Artificial Intelligence

Connect 4: A great example for a Neural Network

Connect 4

The aim of the game is to get four coloured pieces in a horizontal, vertical or diagonal row.
Essentially each player chooses between one of seven columns; they choose a number between 1 and 7.

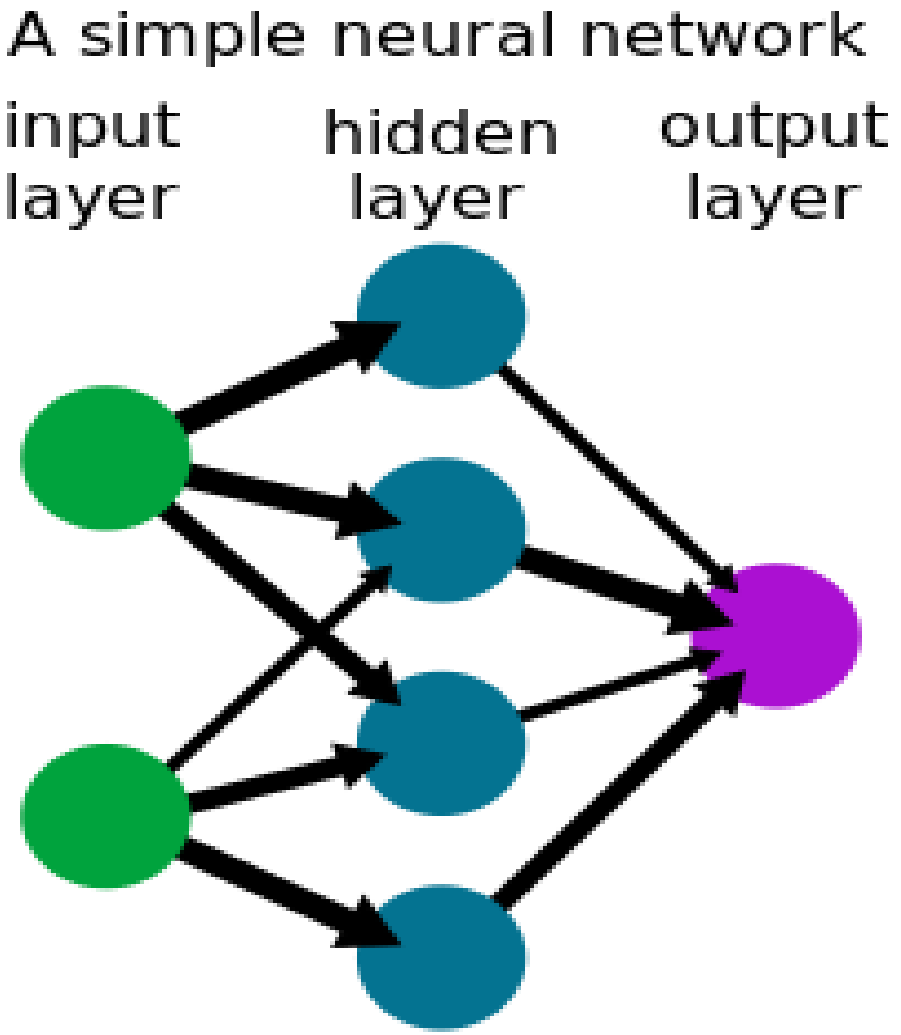
Background

I had been planning to write a program to play Connect 4 for many years, 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 this, but there was 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 move is just choosing a number 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.



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.

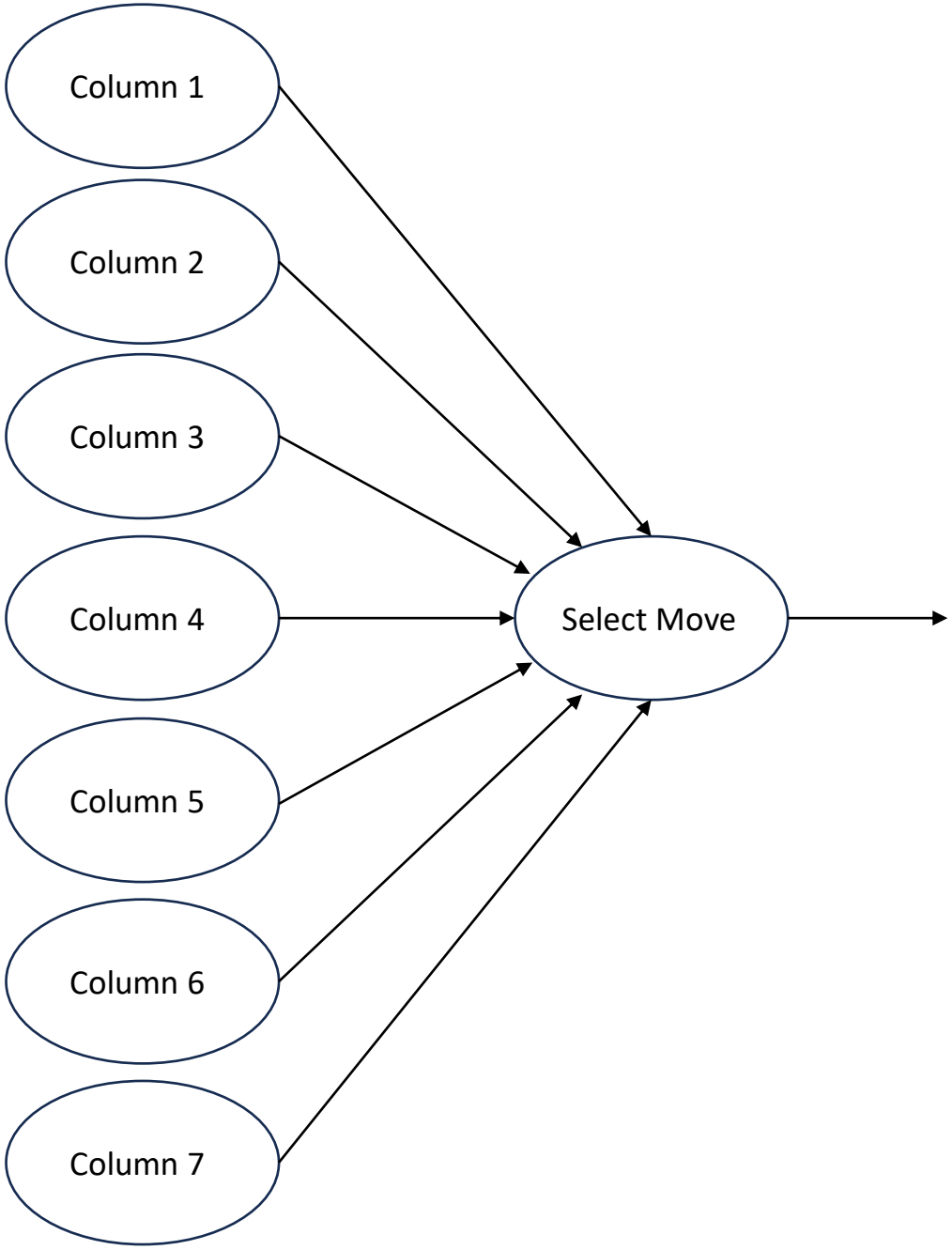
Designing the Neural Network

The description sounds easy, but the devil is in the detail. To play the game a set of neurons (calculations / bits 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.

Just 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.

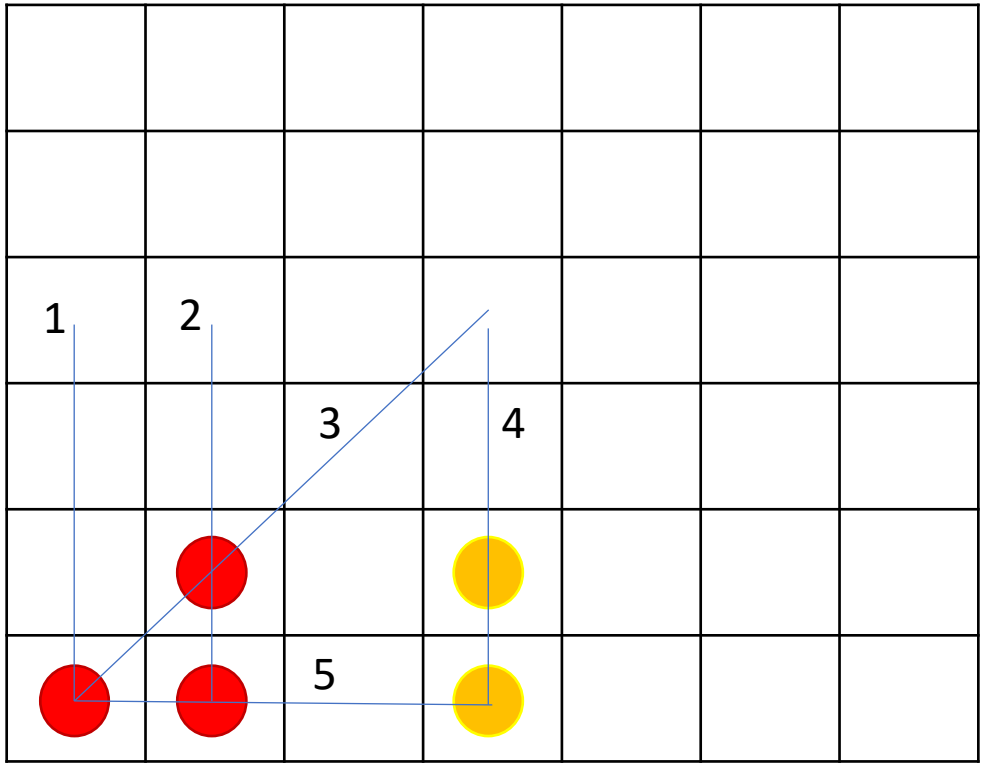
The Output

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 Input

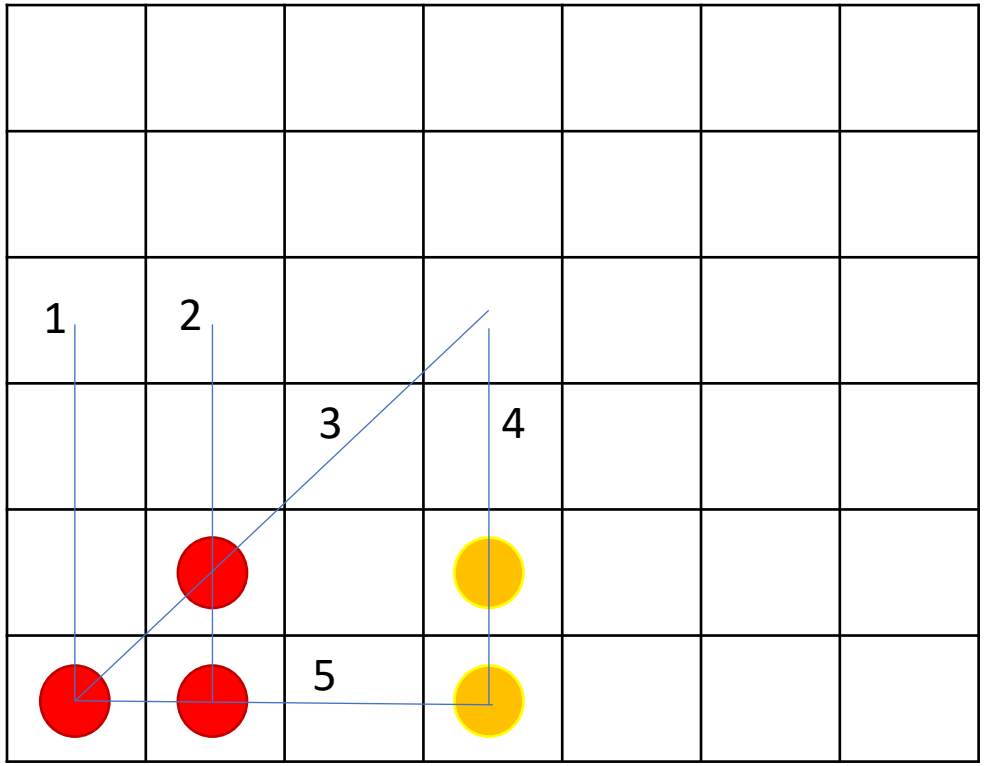
I had the idea that the first neurons needed to work out the score for each possible winning row. The starting point is to count the number of red (R) and number of yellow (Y) pieces within a potential row of four. This can be 1, 2 or 3 (4 means the game has already been won). Each neuron counts the number of pieces. If both red and yellow have pieces, the group of 4 cannot be won, so both R and Y are set to 0.



1 R=1, Y=0
2 R=2, Y=0
3 R=2, Y=0
4 R=0, Y=2
5 R=0, Y=0 (no one can win this line)

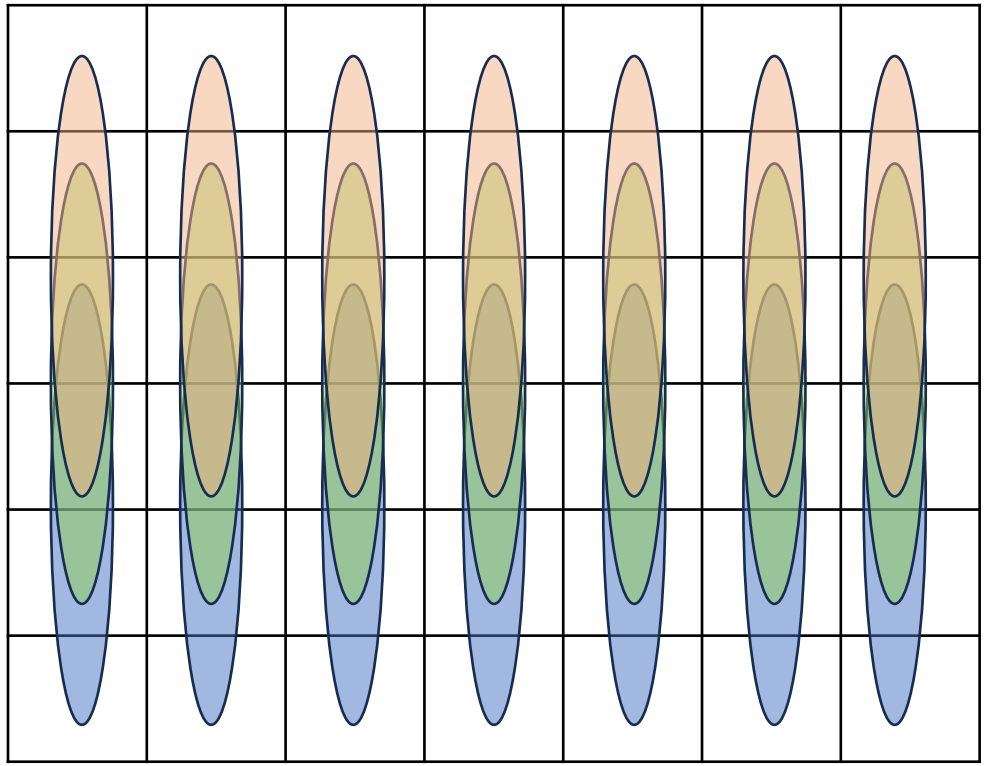
The Input

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

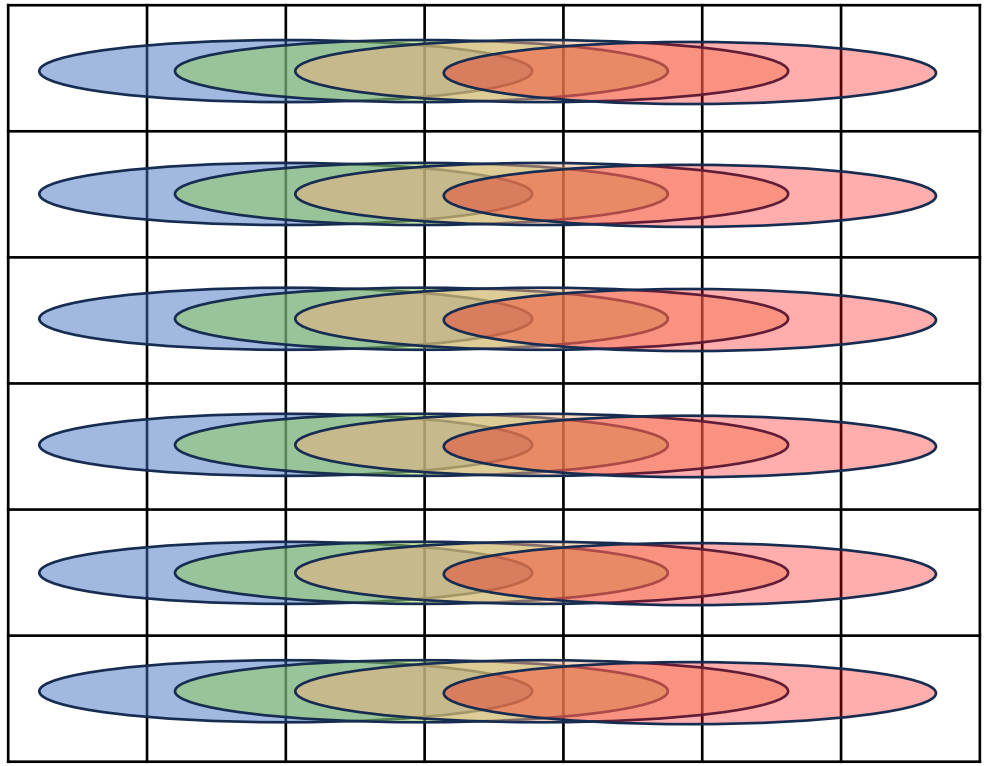


1 R=x^1, Y=x^0
2 R=x^2, Y=x^0
3 R=x^2, Y=x^0
4 R=x^0, Y=x^2
5 R=x^0, Y=x^0 (no one can win this line)

Rows and Columns

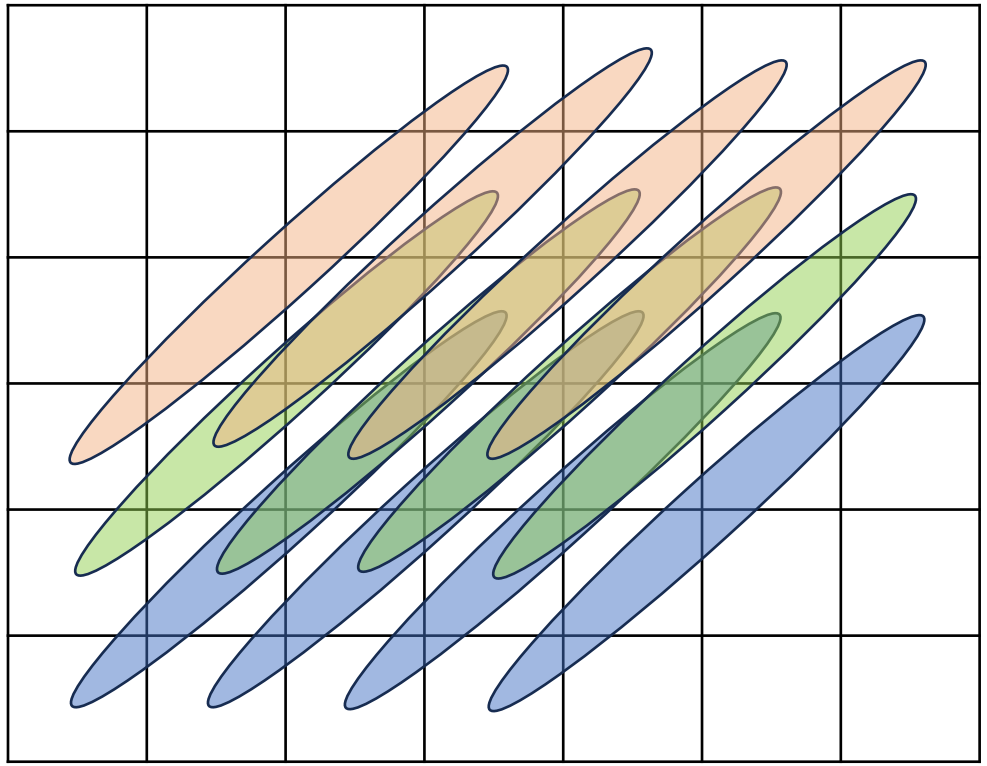


21 possible vertical groups of 4

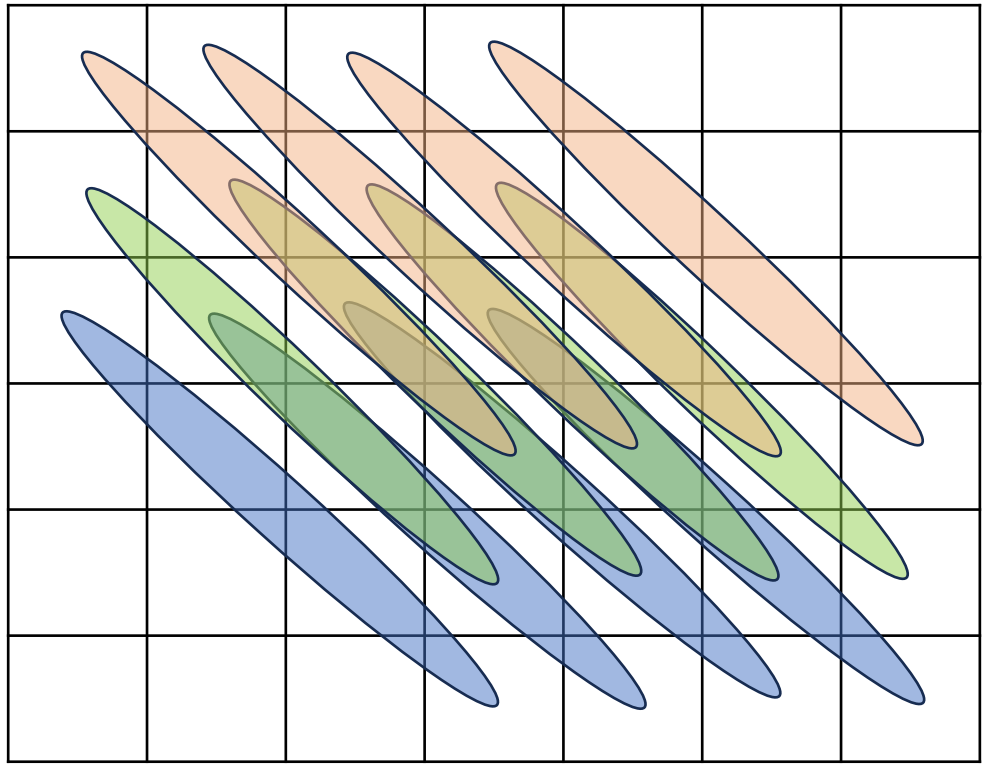


24 possible horizontal groups of 4

Diagonals



12 possible up/right groups of 4



12 possible down groups of 4

69 possible winning lines

21 vertical

24 horizontal

24 diagonal

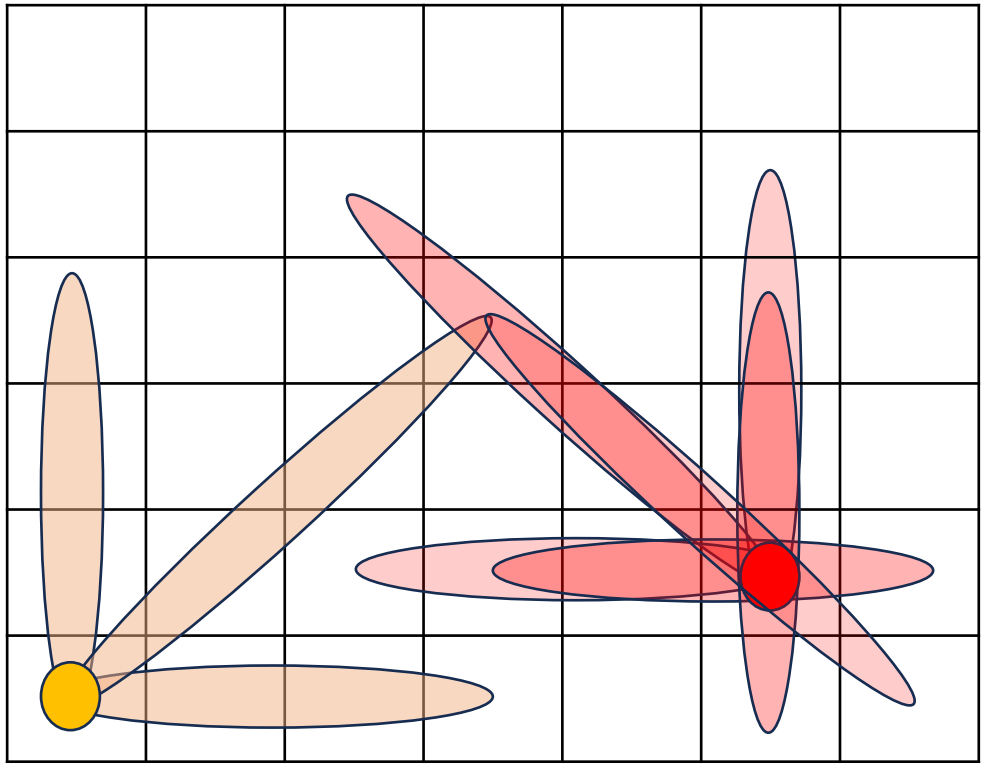
Total 69 possible winning groups of four.

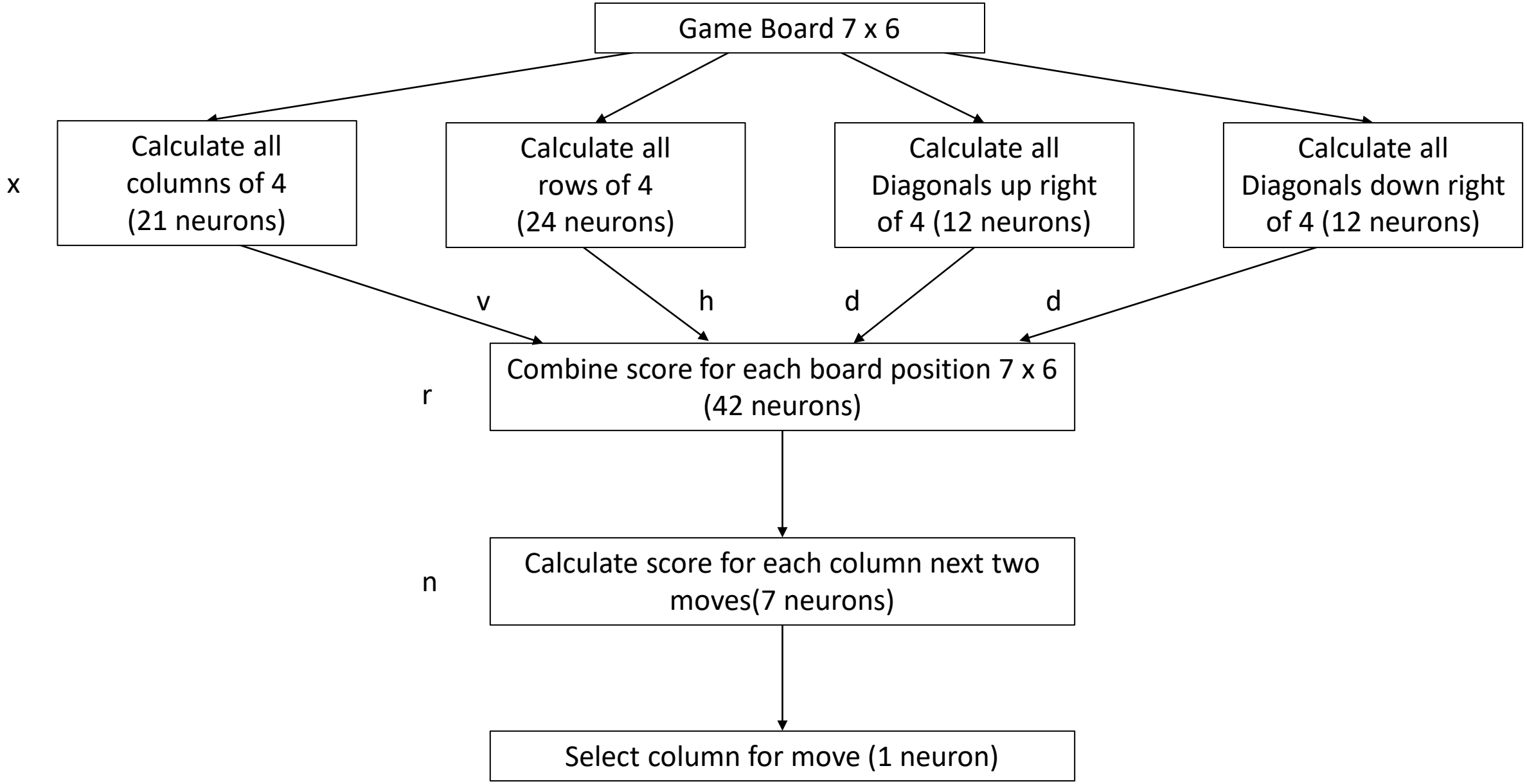
This means 69 neurons to determine the inputs to the neural network.

The Board

To process the scores to get to columns, the different scores from the different possible lines need to be combined into a table of 7x6 places (42 neurons), to determine the score for each position on the board. For the bottom left corner this is the score for the bottom left horizontal, vertical and diagonal group. Towards the middle of the table the positions are in more groups of 4. For diagonals not all positions can be used. Each position on the board has a random number added, so that the computer does not always make the same move.

This gives the following neural network:




Game Board 7 x 6
x
Calculate all columns of 4 (21 neurons)
Calculate all rows of 4 (24 neurons)
Calculate all Diagonals up right of 4 (12 neurons)
Calculate all Diagonals down right of 4 (12 neurons)
v
h
d
d
r
Combine score for each board position 7 x 6 (42 neurons)
n
Calculate score for each column next two moves(7 neurons)
Select column for move (1 neuron)

Columns

The score for each column is calculated by finding the next free position (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 just chooses 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 piece count (found 4 or 5 best)

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 (max 2 or 3 seems to work best)

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 was optimised by the computer playing against itself. A set of default weightings was chosen. The weightings were then varied across the ranges shown. The program played 100 games of the new weightings against default weightings. If the new weightings won more games, these new weightings were then played against the current set of weightings. If the new weightings won again, the current weightings were set to match the new set.

This approach was run several times with different x and r values. Originally the x and r values were also varied, but this created unpredictable results. Increasing the random factor made the outcome too unrepeatable. Changing x impacted other weightings, so it was fixed at 4 or 5 during optimisation runs.

The program now uses weightings from the optimisation process and seems to be a good player.

Implementation

The program is written as a single class in Java and built into a single jar file. The program runs as a console program but has a command line option. The command line is:

Java –jar Connect4.jar “ YRYRY YRY R “

The string is 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 from to the command line.

Next

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. Python can call the Java program as follows:

```
subprocess.call("Java –jar Connect4.jar "" YRYRY   YRY    R                    """,
shell=True)
```