Sam Dauncey Candidate No. 4191

Guildford county school

Investigation into the use of Reinforcement learning techniques in timed connect4

Contents

[Analysis 2](#_Toc5111161)

[Project Outline 2](#_Toc5111162)

[Machine Learning Research 6](#_Toc5111163)

[Specification 13](#_Toc5111164)

[Design 14](#_Toc5111165)

[Overall Coding Style / Paradigms 15](#_Toc5111166)

[General Algorithm design 16](#_Toc5111167)

[GUI design 18](#_Toc5111168)

[AI algorithm implementation 19](#_Toc5111169)

[Class Structure 23](#_Toc5111170)

[Technical Solution 25](#_Toc5111171)

[Python Files 25](#_Toc5111172)

[Techniques Used 53](#_Toc5111173)

[Testing 55](#_Toc5111174)

[GUI Testing 55](#_Toc5111175)

[AI Testing 55](#_Toc5111176)

[Evaluation 62](#_Toc5111177)

[Completeness Based on Specification 62](#_Toc5111178)

[Supervisor Feedback 65](#_Toc5111179)

[Potential Improvements 66](#_Toc5111180)

## Analysis

### Project Outline

##### background and Introduction

Originally, I planned to focus this project on the game of Chess, however I quickly realised that, Chess is relatively complex in terms of the number of different rules etc.; making it much tougher to get an AI to learn. I have since chosen to re-focus on the game of connect4, which is comparatively simpler whilst still being classed as a “complete information game”. This will hopefully allow for me to go into more depth within the scope of this fifty-hour project, however I will try to keep my technical solution as general to these types of games as possible so that in theory the game rules could be switched, and the AI would still learn to be relatively proficient. That said, much of my research may focus around Chess as it has many learning materials that are available which talk about applying AI to it with it and is a game that I am more familiar with.

That said, modern Chess “engines” are computer programs which evaluate how strong a player’s possible moves are in each position in a game of Chess. On a small scale, a system like a Chess engine could be used to create an opponent for someone to play against in an app or by high-level players to analyse their games and see where they could improve, as in many areas engines are stronger than the top grand masters in their respective games. On a larger scale, new developments in this technology could have wider implications in the field of game theory and machine learning.

Reinforcement learning is a relatively new branch of machine learning which focuses on creating algorithms which learn to take actions in a certain environment (for example a game), with the only intervention being punishing the AI when its actions lead to it “losing” and rewarding it when it ‘s actions lead to it “winning”. Typically, when investigating reinforcement learning one will create an algorithm to play a certain game, given that it is often relatively easy to generalise the algorithm for games in that class. There have been recent breakthroughs in using reinforcement learning techniques to complete information games such as Google DeepMind’s “AlphaZero” AI, detailed in (Silver et al. 2018), which was able to reach superhuman levels in games including Go and Chess whilst only being given the rules of the game it’s playing and some time to learn how to play.

My goal in this project is to take these new breakthroughs and apply them to connect4, whist slightly adapting them so that the machine learns to play more optimally given there is a timer involved.

##### Supervisor Identification

My supervisor for this project is a third-year university student who has done modules relating to machine learning (namely “Artificial Intelligence” and “Deep Learning and Advanced AI”) and has more than ample knowledge to help guide me through this project.

##### Current solutions: MiniMax

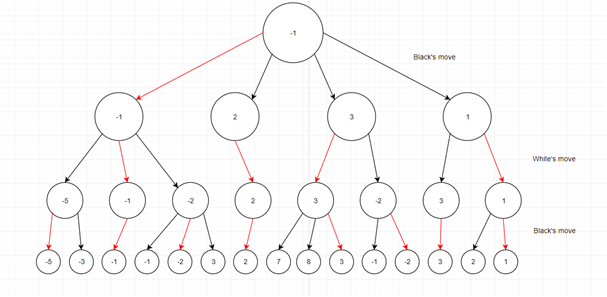


Figure 1: a graphical representation of how the MiniMax algorithm works

Currently, Chess engines work mainly using an algorithm called “MiniMax”, which works by considering each state resulting from every possible action that either player could have played, then rating these states relative to how favourable they are for black or white. After this, it can then be reasoned which actions each player would play if they both played optimally; and therefore, which move the starting player should play. The diagram above is a decision tree which illustrates an example of this process, where each node represents a certain state, and each arc points to a position that would result from a certain action. As this shows, the ratings (negative representing good for black, positive representing good for white) for a “layer” of positions above are dictated by the positions which it points to, if it is black’s move the most negative position will be picked whereas if it is white’s move the most positive position will be selected.

MiniMax is further optimised by “pruning” branches of the decision tree that do not need to be calculated. One of the most simple and effective methods of “lossless” pruning (where you can be sure that pruning the branches will not affect the final result) is the “alpha-beta” pruning algorithm. This is where you can use logic to deduce that a player should not choose to go down a branch of the decision tree. For example, in the decision tree shown in Figure 1, if the computer was calculating from left to right the alpha-beta pruning algorithm could reason that it would not need to calculate the two nodes marked (-2,3) in the purple box, because, even if they are highly positive values, black could select to play to the (-1) node next to them, and since it has already been calculated that, in the best case scenario white can force to play to a score of (-1) anyway, it need not calculate these two nodes. As we can see from the other green marked boxes, alpha beta pruning can be used to prune whole branches of the tree as we get further right on the tree, reducing the total number of calculations to evaluate positions down from 14 to 8.

The main problem that current Chess engines face is that rating the positions for the bottom “layer” of pieces is a tricky problem to solve using a hard-coded solution. This is because, if both players have an equal number of pieces, rating the position becomes an intuitive problem that comes relatively easily to human players but is hard to quantify into set rules which a computer can follow. I am going to be doing an investigation into using machine learning to solve this part of the problem, as techniques such as neural networks have been shown to excel in these types of intuitive problems when other algorithms struggle.

On top of this, even with alpha-beta pruning, the time MiniMax takes for Chess is still O(cn) problem, where n is the depth the algorithm is programmed to go to; as each node in one layer corresponds to multiple nodes on the layer below it. This means that the amount of computing power required explodes as the depth is increased, meaning that even the most efficient modern Chess engines with incredibly powerful supercomputers can only compute about 30 moves deep. In stark contrast, humans can use their intuitions to prune and remove unnecessary calculations at a much more effective way, telling them where they can cut their calculation short and where they should keep calculate deeper (usually based on how “closed” the position is). This is how in 1996, reigning world champion Gary Kasparov was able to beat IBM’s computer Deep Blue in a best out of 6 matches despite Deep Blue being able to analyse roughly 200,000 positions per second in comparison to Kasparov’s 3 per second. Again, intuitions about how “closed” the position is are something that are very tricky to program directly.

##### current solutions: upper Confidence Tree search

As already mentioned, there are many problems with using pure MiniMax to solve the problems presented by AMGs, so when creating their AlphaZero AI, DeepMind used a different algorithm called Upper Confidence Tree (UCT) search, which was first outlined in (Kocsis and Szepesvari, 2006). The underlying idea of UCT search is to slowly build a decision tree by playing “simulated” games from the starting state until we get to a state that hasn’t been analysed; at which point we analyse how favourable this state is for either player and then propagate this information back up the tree.

UCT search relies on some maths called the “multi-armed bandit” problem; which can be explained in terms of standing in front of a row of slot machines, each of which have a different probability of giving out a reward (say £1). The problem comes with the idea that one has some number k tokens which they can use on any one of the slot machines, how can you maximise the amount of pay-out you get from using all your tokens if you don’t know the probabilities of each machine? One solution to this problem comes from generating “upper confidence bounds” for the probabilities of each machine (for example I can say that I can be almost certain that a given machine will not have a probability of above 75%) and then picking the machine with the highest one of these bounds at each time step.

Figure 2: A flowchart detailing how the expand node function works, explained further in the next page

The idea behind UCT search is that we can frame the problem of deciding which move to pick in a simulated game as our multi-armed bandit problem, as both are deciding between the trade-off of “exploring” different moves and machines and “exploiting” what we currently think is the best machine or most likely move. AlphaZero uses a neural network with UCT search to give estimates of what the best move is from a given game state and to estimate what the how favourable a state is for each player as a value in the range [-1 to 1], called the “value”.

More formally, each simulated game involves calling the “expand” function on the root node in our decision tree (corresponding to the current state in the game), which will be recursively called on the child node that has the highest upper confidence bound for its value (called its “UCT value”), a flowchart for this expand function is shown in Figure 2 on the previous page. This process continues with the children of this node until we reach a node/state that we haven’t visited before; known as a leaf node. When this happens, we initialise this node using the value of its state which is outputted by our neural network. We then pass this value back up our tree, using it to increment the “total value” of the states we passed to get to our final one, which is then used to recalculate their UCT values. Using the maths from the multi-armed bandit problem we get the UCT value to be equal to:

Where the policy estimate is also a value outputted by the neural network as an estimate of how good it thinks this move is; and the exploration constant is a value which we can adjust manually to get the best results.

##### project concept and acceptable limitations

As it was used in AlphaZero, the UCT algorithm does a set amount of simulated games per move; meaning roughly the same amount of time is spent thinking on each move. I believe that this could be reducing the effectiveness of the algorithm, as some moves in a given game should be a lot more obvious than others, for example in Chess if there is only one move that white can make to prevent checkmate, he should obviously make that move. On top of this, the neural network does not take into account how much time each player has left when calculating the value and policy for a given state, which could be important.

In this investigation I will therefore be looking at trying to adapt this version of the UCT algorithm to work so that the AI will budget the amount of thinking time it is given in a more efficient manner and incorporate how much thinking time it has left into its strategy. I feel like this project is ambitious, and thus I feel like merely getting the AI to learn and improve with these adaptations in place will be my main goal here. This is also due to the fact that I do not have a large amount of computational resources available to me, which machine learning projects typically require (AlphaZero played roughly 20 million games against itself to train its neural network); I am therefore definitely not trying to create an AI that is incredibly intelligent like AlphaZero.

### Machine Learning Research

##### the basics of neural networks

Starting this project, I have had very limited experience with neural networks before; so I have had to research the basics of how they work. At a very basic level, a Neural Network takes an array of inputs, and performs operations on them to produce an array of outputs; with the idea that the neural network can be “trained” to identify patterns in the input data to produce a desired output. Neural Networks can be represented as a weighted graph with layers of nodes, in a standard “dense” layer all the nodes in a previous layer are “connected” to any given node in the layer in front. This means that to calculate the value of a node you need to get all the values from the previous layer, multiply them by the corresponding weight of their connection to the node and sum up all the results. A sketch of this graph is shown in Figure 3.

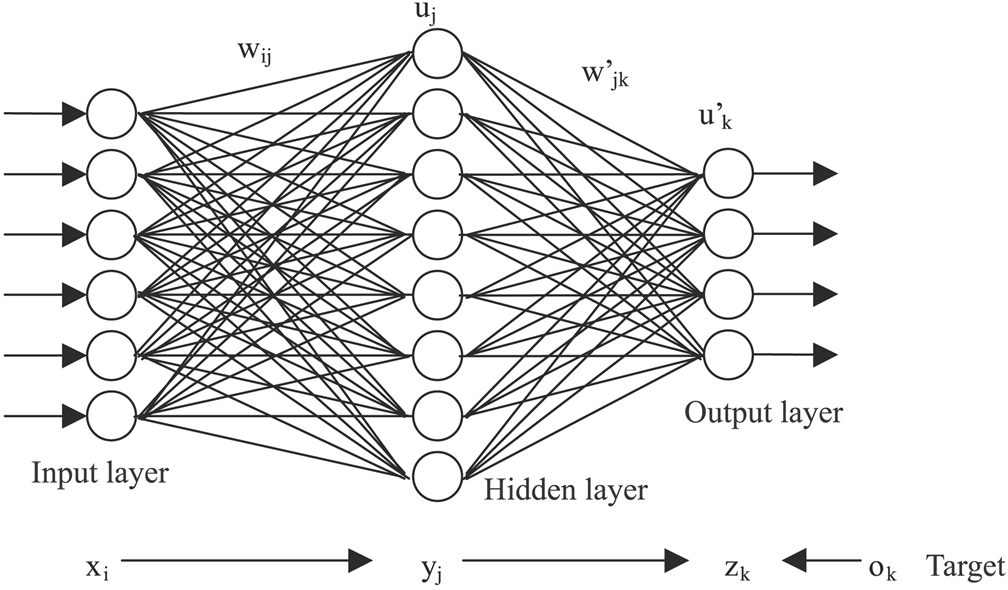
Mathematically, this corresponds to taking the dot product of a vector storing the values of the previous layer with a matrix storing all the weights of the connections to get a vector output giving the values of the next layer along. After this has happened, an “activation function” is applied to the given values to give the final values of that layer of nodes. The activation function is necessary as it ensures that the inputs to not have a linear correspondence with the outputs, which would be mathematically equivalent to neural network with only an input layer and an output layer (which is generally not very useful for machine learning problems).

Figure 3: a graphical representation of a neural network with three dense layers.

What does all this mean in practice? I will likely opt for using a library to do all these neural network calculations. This is because there are many readily available libraries that do these calculations using the GPU, which make them much faster and more efficient and thus greatly increasing the performance of my engine; and doing this myself is beyond my level of technical ability. On top of this, a library will likely be much easier to use, set up and change a given network enabling me to do more investigation into the use of different types and sizes of layers in this application of the neural network. This said, I still do believe that understanding the neural networks function at a fundamental level will greatly increase my ability to apply them; for this reason, I programmed my own Neural Network class in java as part of my research (shown in Figure 4)

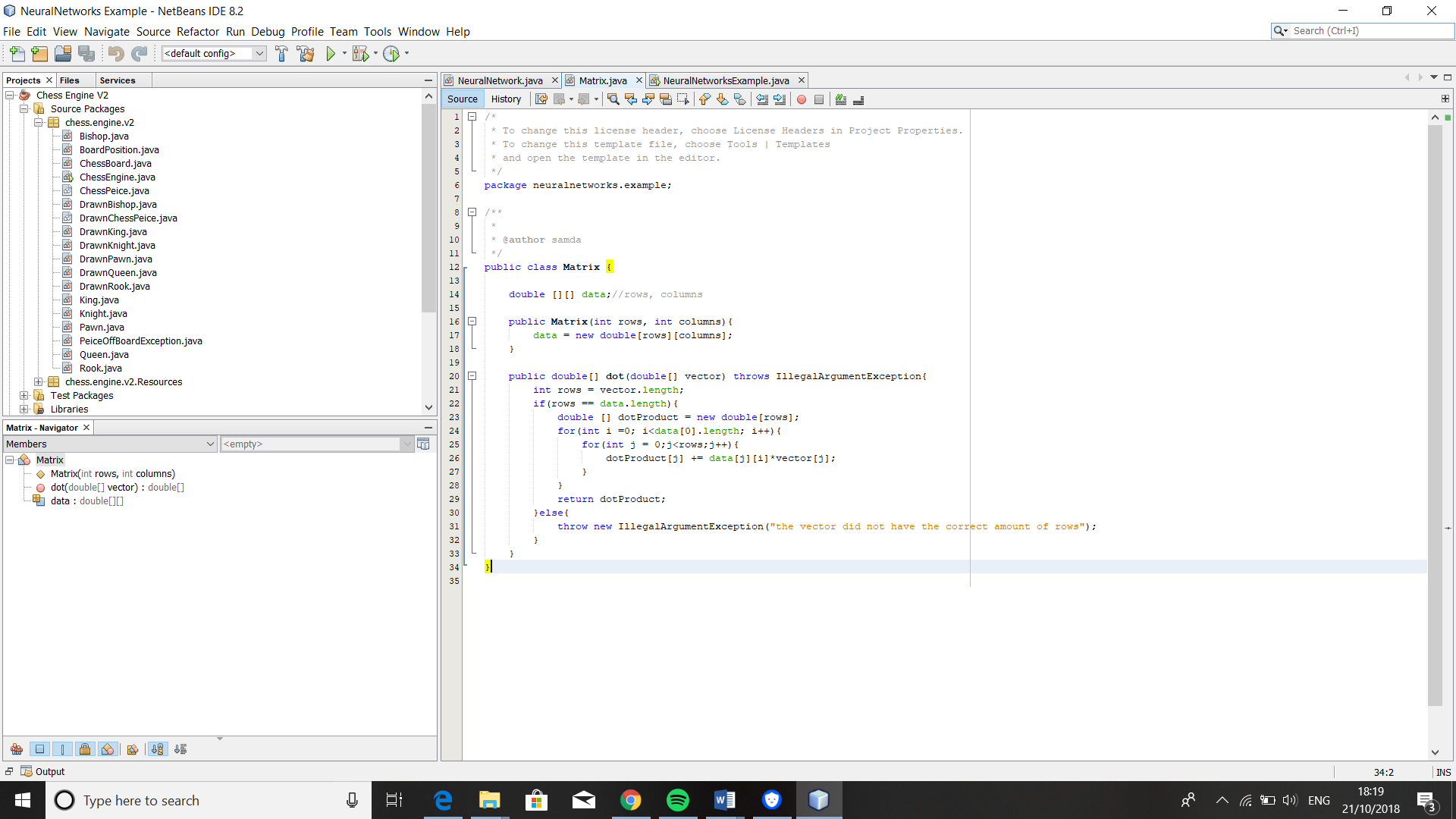
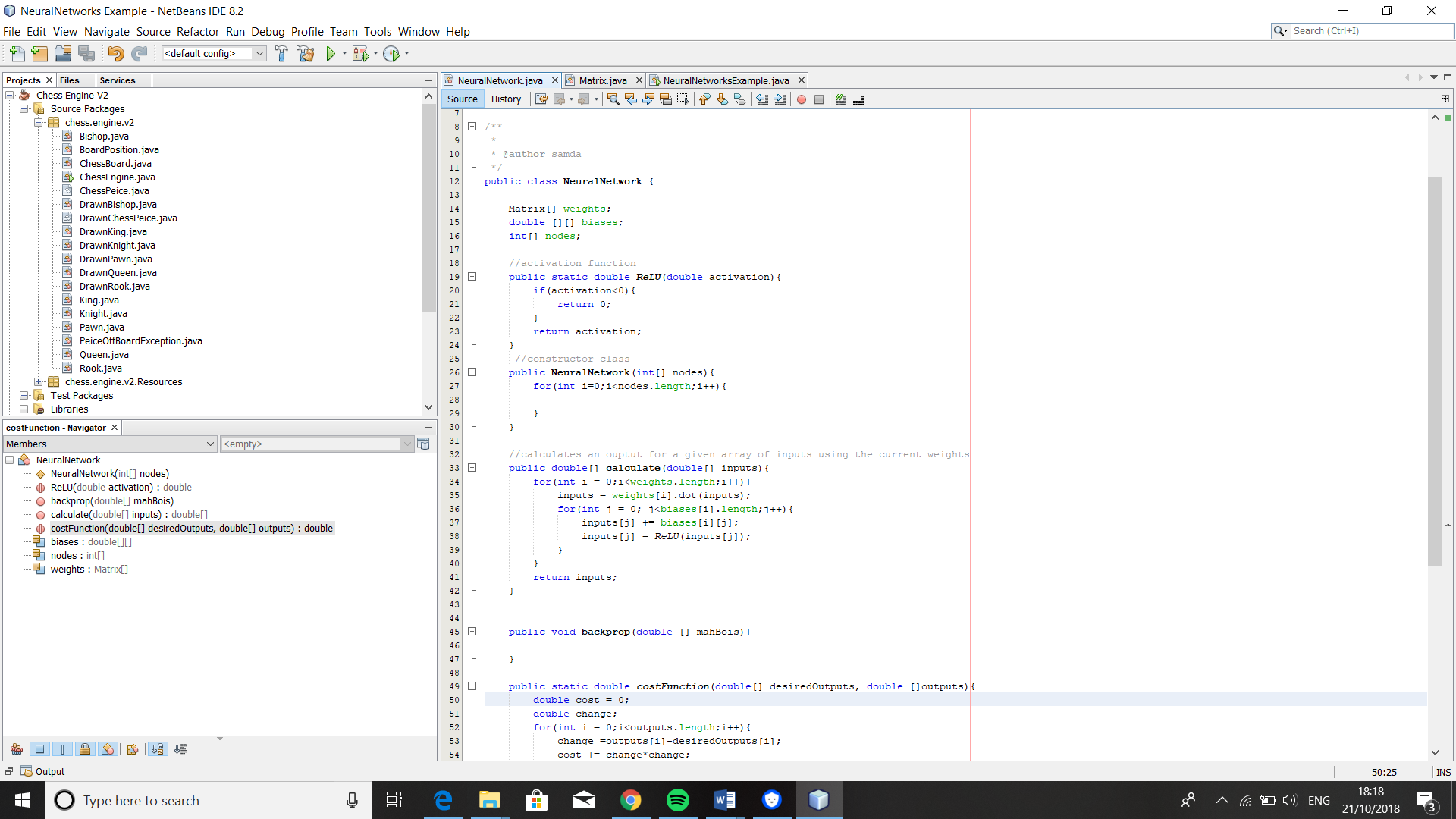


Figure 4: my neural network and matrix classes NeuralNetwork.java and Matrix.java

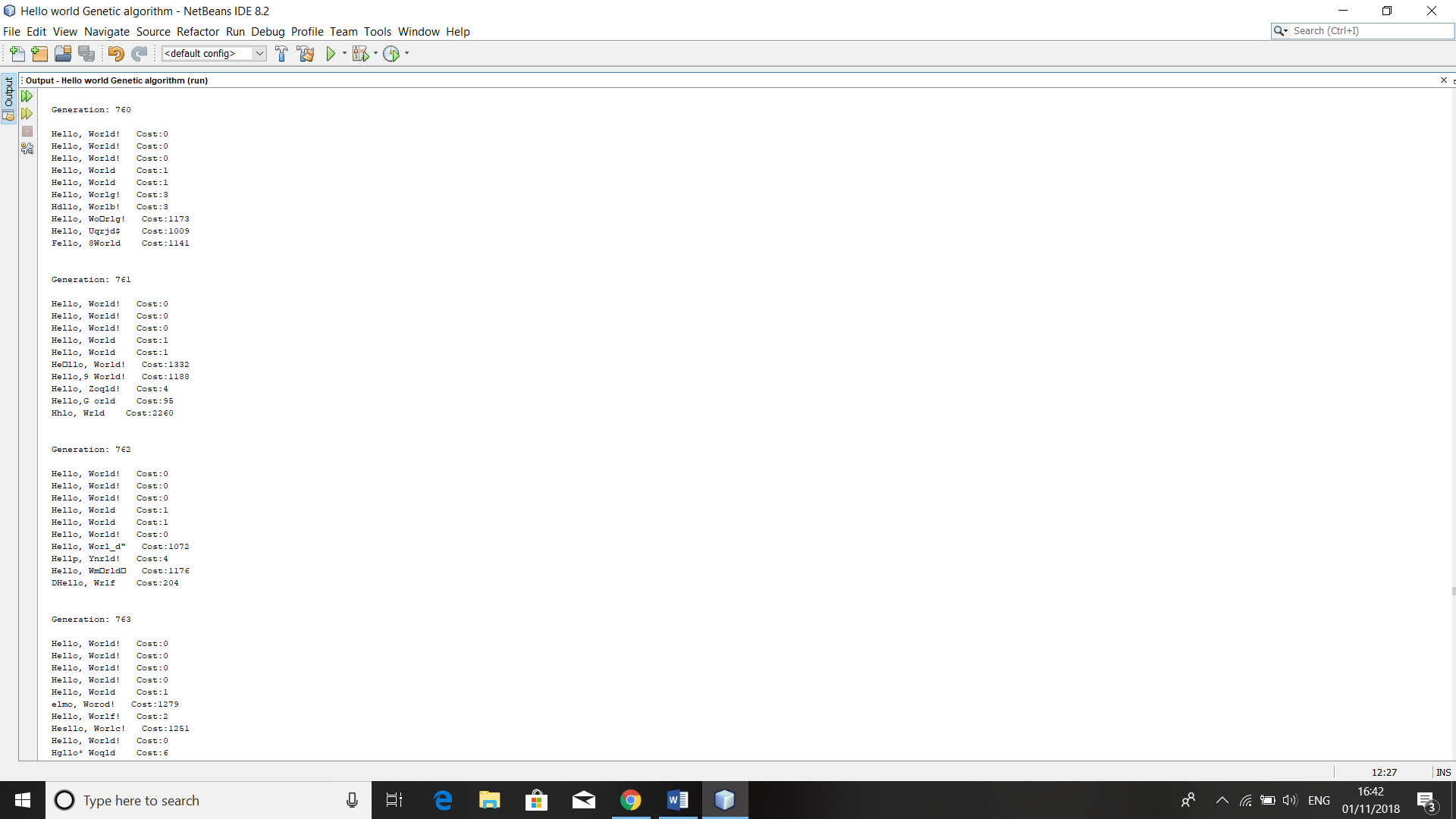
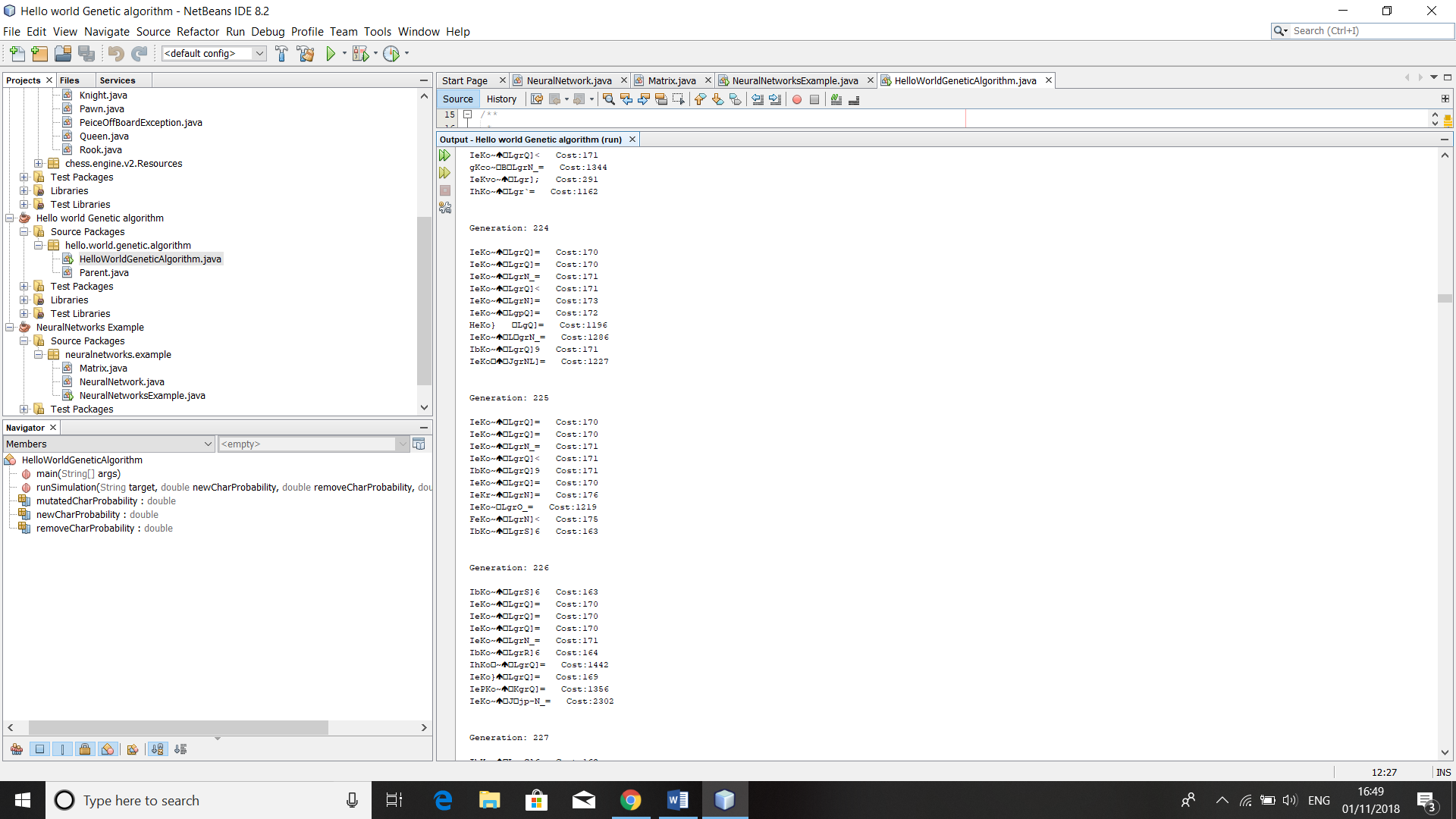
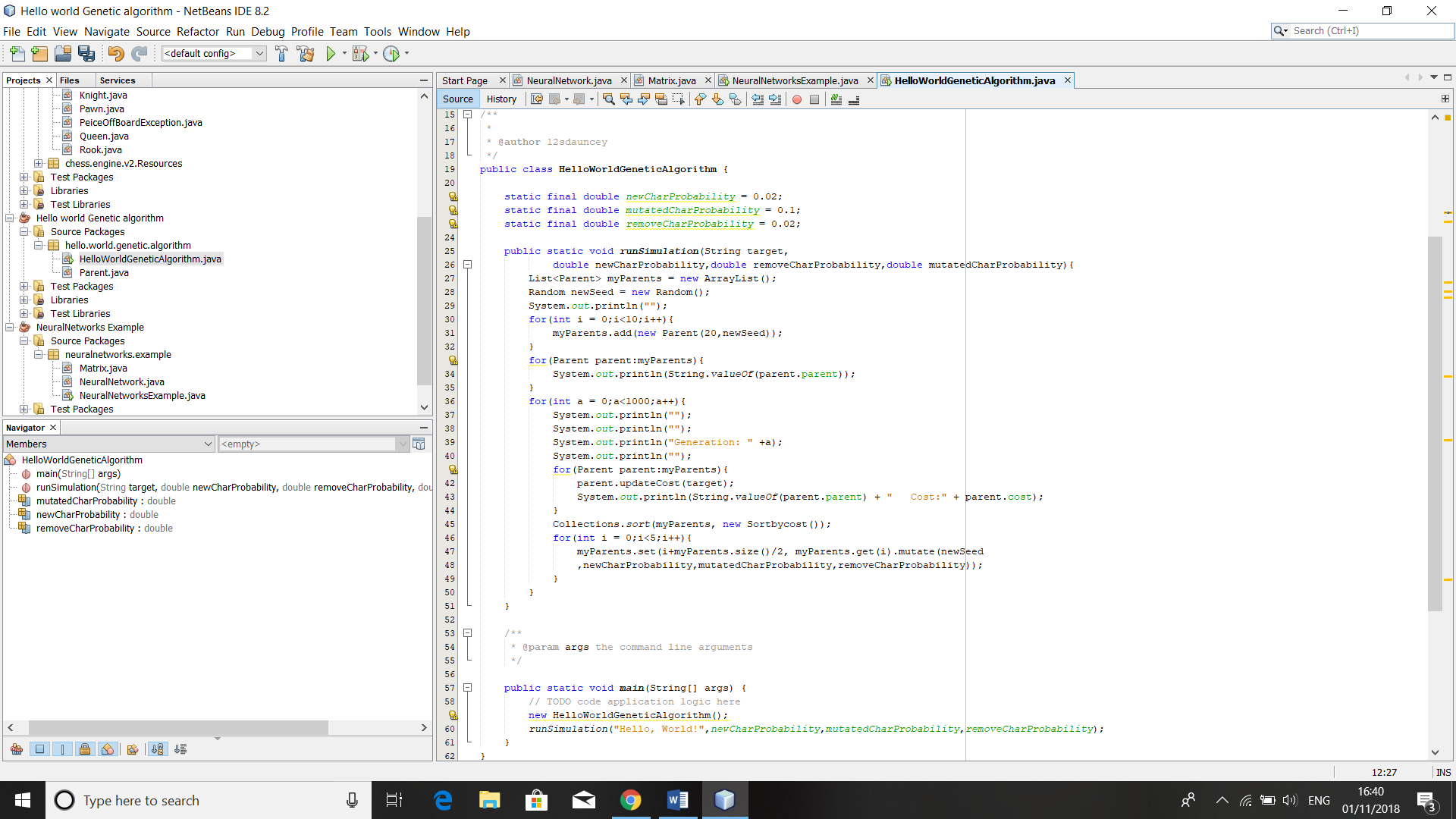
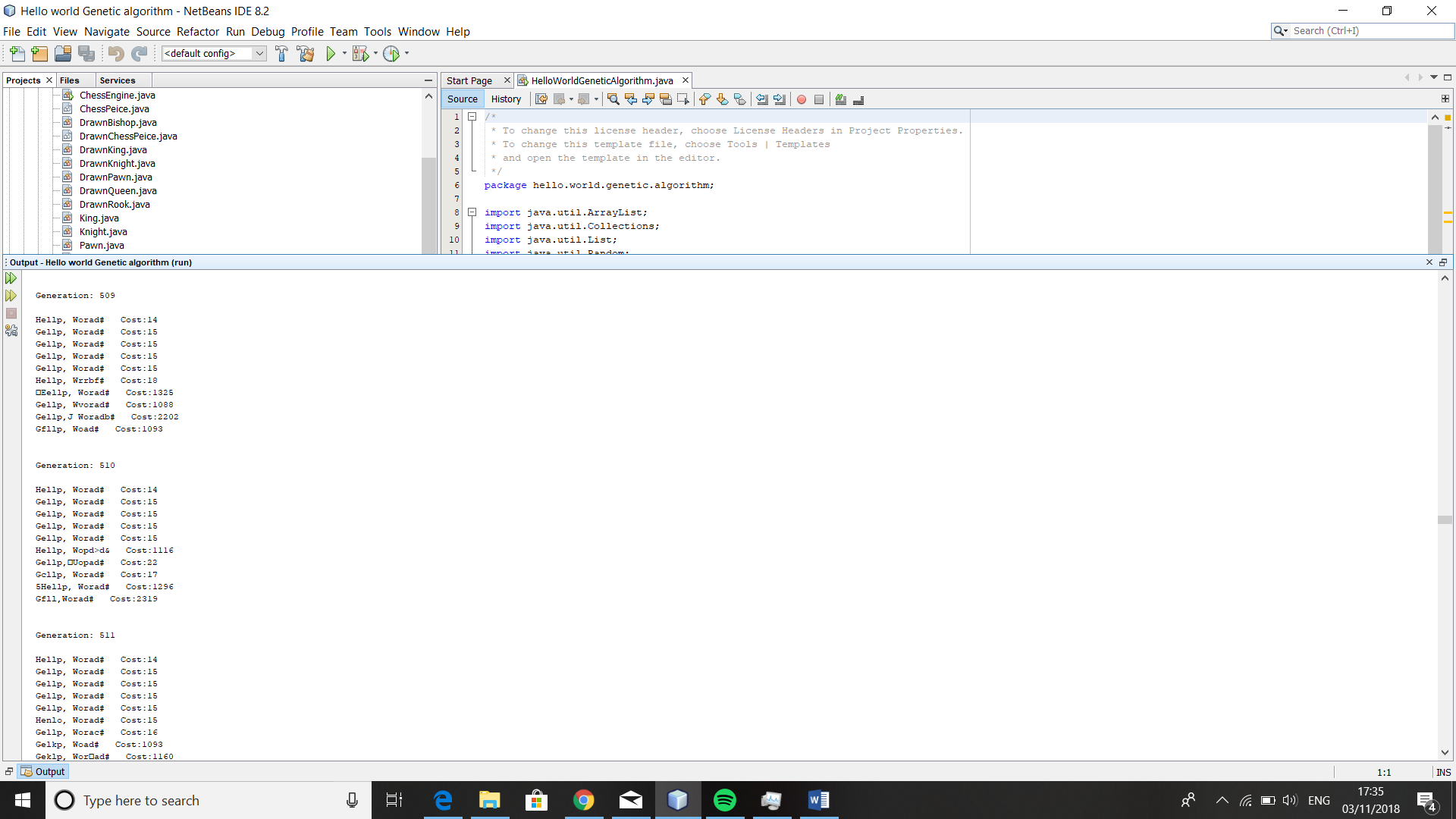
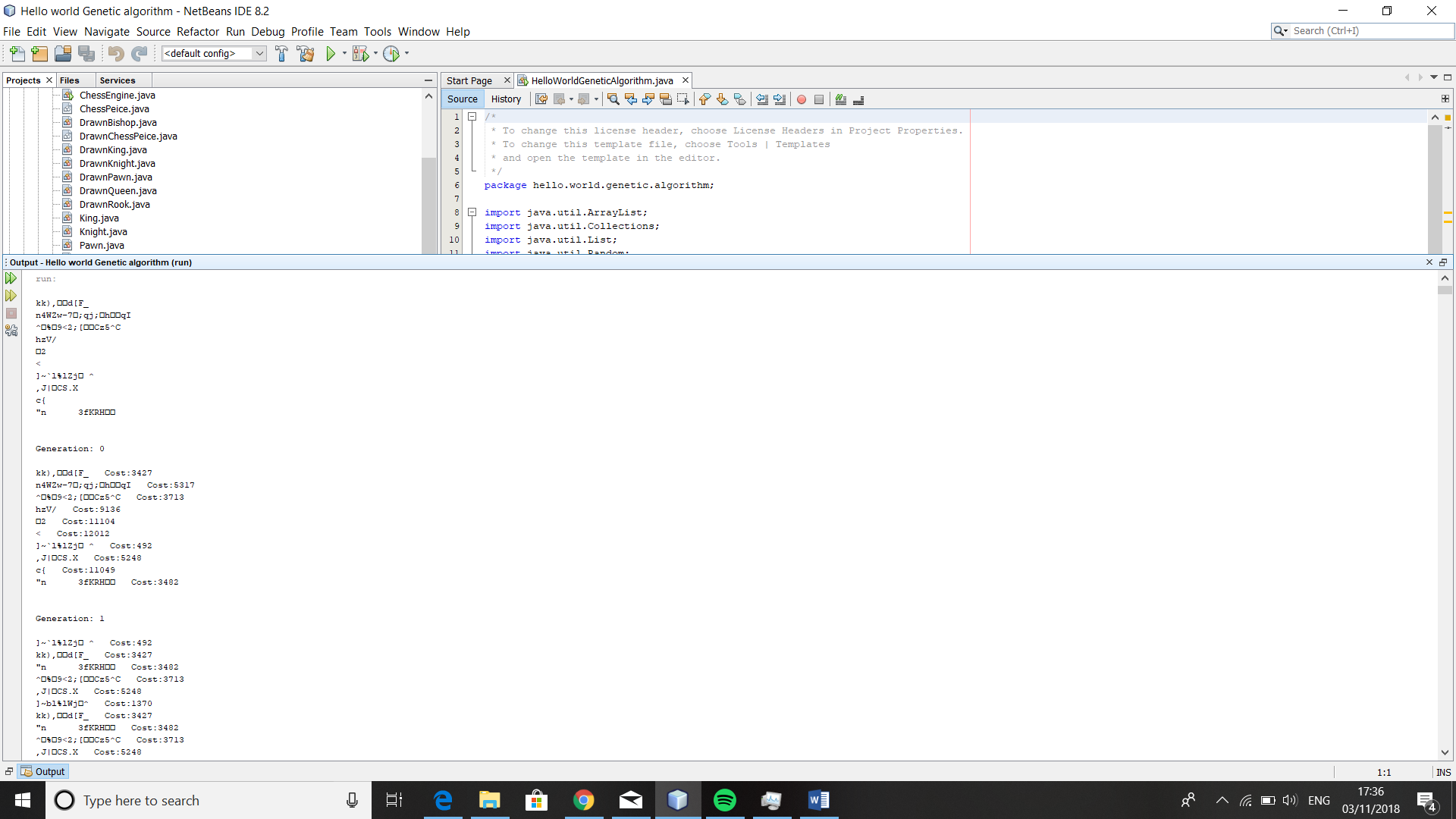


##### training algorithms

Without training algorithms, neural networks would be practically useless as all the weights would have to be adjusted by hand to get the desired output, and the result would likely be a relatively inefficient solution to the problem in relation to how much work would have to be put into it. However, training algorithms can tune the weights to give the desired output, which is what make neural networks so versatile and able to tackle problems that would have previously considered unsolvable for computers.

One of the simplest training algorithms is a genetic algorithm. This is where a pool of neural networks is randomly generated, then evaluated based on how well they perform a given task. The ones that perform badly are thrown out and the ones that perform well are “mutated”, that is the weights are randomly adjusted, to create child networks and then evaluated again in a new iteration. The idea is that after each generation, the networks will become more and more adapted perform better. The problem with genetic algorithms is that they take a very long time to adapt into a suitable network for any complex problem; this would be a problem for me as I have a limited amount of computing power available to me. One of the benefits of using this type of algorithm is that by definition it is reinforcement learning, as labelled data is not needed, not only this but it means the networks would adapt to work optimally with the type of MiniMax algorithm that I choose to use the network with. To further research into how I would use a genetic algorithm and just how long it would take to produce a viable network, I also wrote a program to model the use of a genetic algorithm; the source code for which is shown on the next page.

The way this program worked was to model the networks as “parent” strings which were initially of random length and filled with random characters. These strings were evaluated with a cost function which assessed how far away from the phrase “Hello, World!” they are, with a large penalty for a difference in the amount of characters in the string and then a



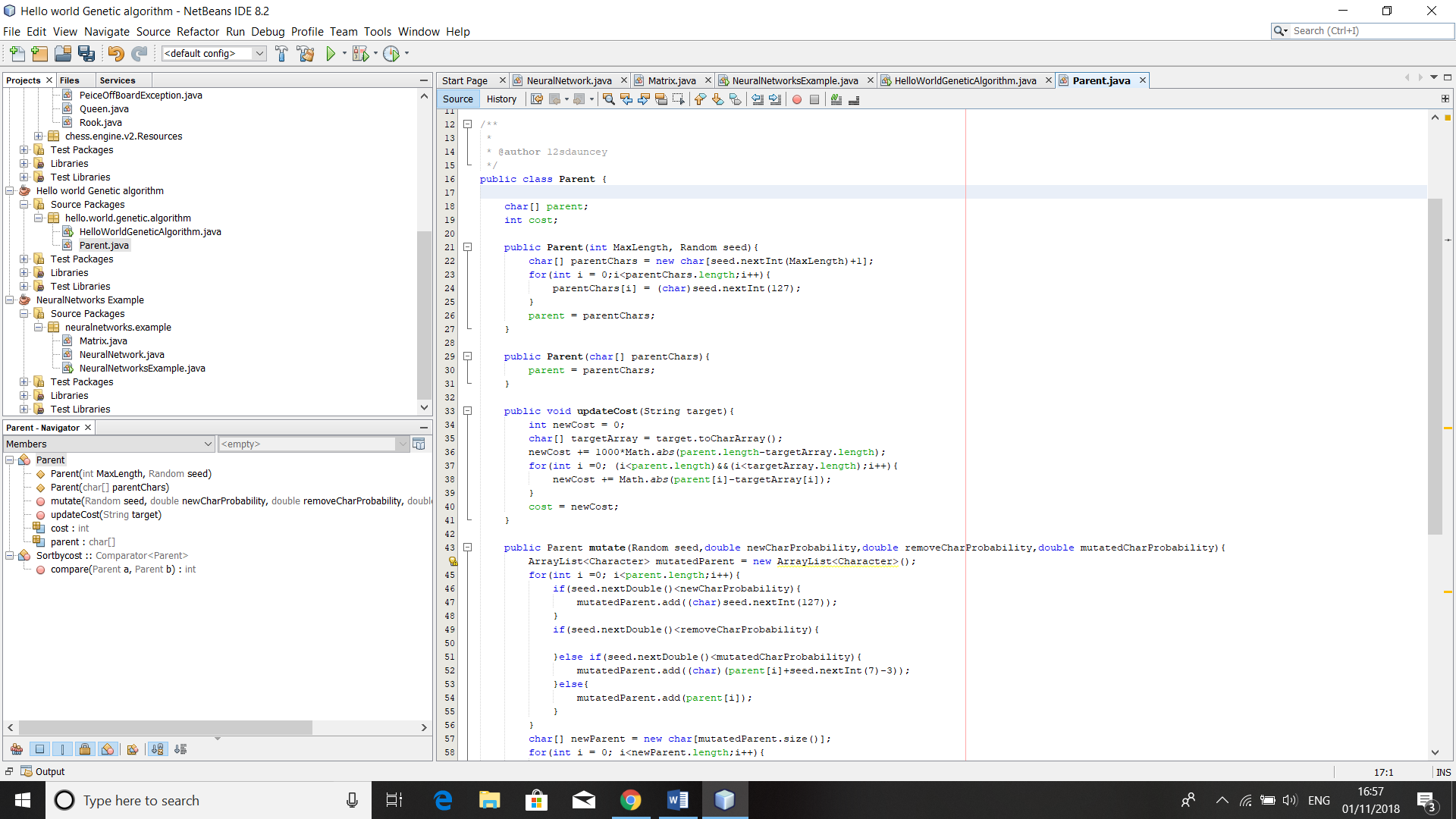
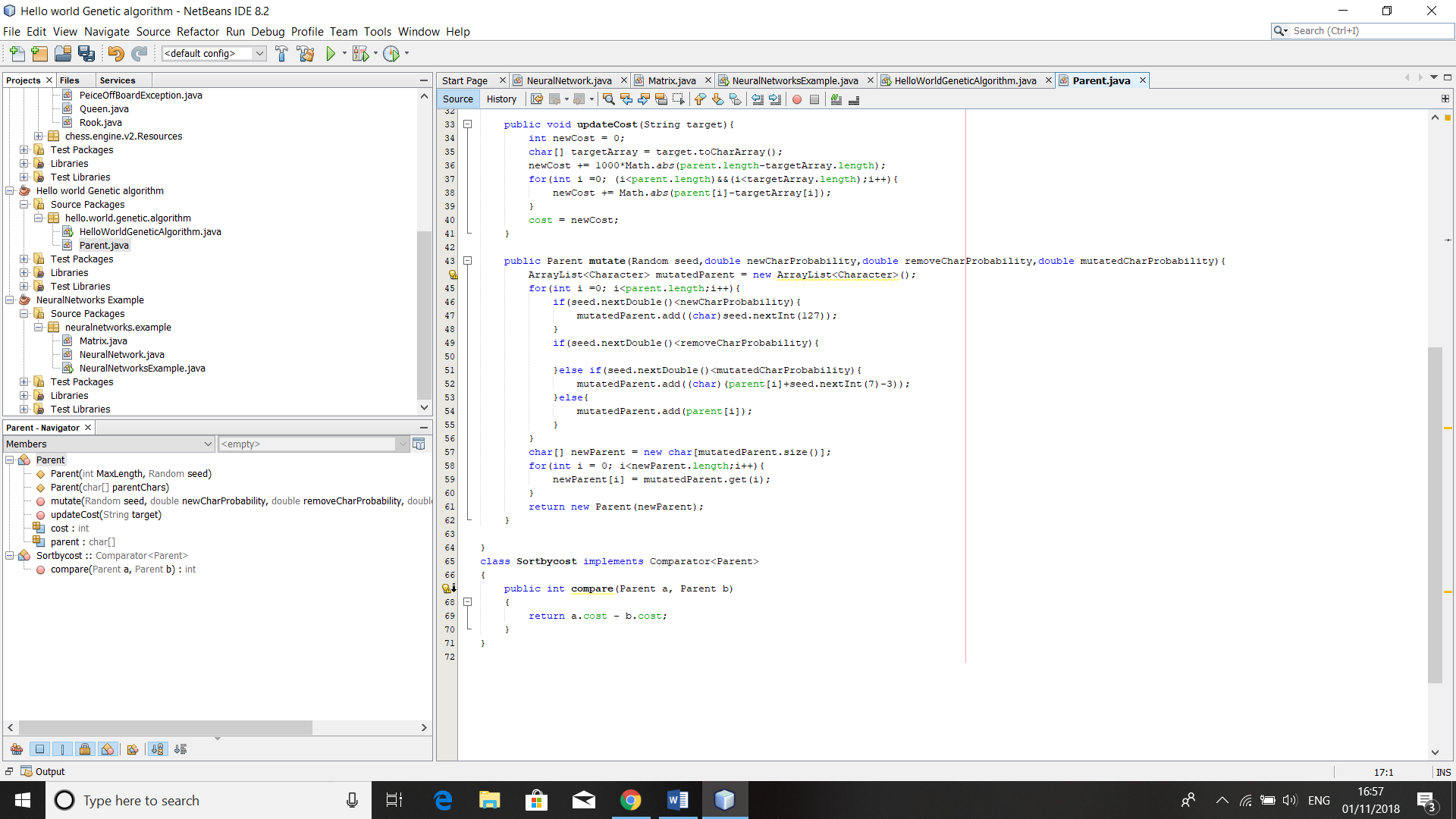


Figure 5: My program to model a genetic algorithm, HelloWorldGeneticAlgorithm.java , Sortbycost.java and Parent.java and some samples from its output

smaller penalty per character depending on how far away its ASCII value was from the desired one for the character in its position. The parents with the highest costs were replaced with mutated versions of the parents with the lower costs. In this context, mutating the parents meant having a small probability to alter the ascii value of a given character along with an even smaller probability that it will gain or lose a character.

As I expected, the algorithm took a large number of generations (roughly 750) on each test to converge on the string “Hello, world!”; this will only take more and more time when I introduce a more complex problem such as the game of Chess. Whilst with some tweaking of the variables that define the probabilities of the different types of mutation, I suspect this number could be reduced, the number will always be in the hundreds. For this reason, I will likely not invest many resources into investigating the use of this algorithm, however I may investigate using it to “fine tune” a network after it has been trained using a different algorithm.

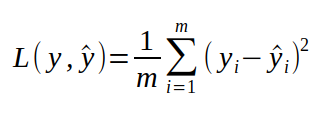
For these reasons, genetic algorithms are virtually never used in actual machine learning, with most programmers opting for “gradient descent” algorithms instead. These types of algorithms require labelled training data (in this case, this means they need to know what we value want each output neuron to have for each piece of training data). They use these “desired outputs” along with the actual outputs from the network to compute the “loss function”, which is used to determine for how far away the actual outputs are from the desired outputs. To quickly explain how the loss function is typically calculated, essentially for each output neuron the difference between the actual value and the desired value is squared (to: 1.remove any minus signs and 2.punish the network more for being very far away than being close-ish to the desired output) and then we take an average of this score across all the output neurons. This type of loss function is called mean squared error, a mathematical formula is shown for it in Figure 6 and will likely be the one I choose to use in this project.

Figure 6: Mathematical formula for a mean squared error loss function, y and ŷ are vectors containing the actual and desired outputs, and m is the total number of output neurons

The key here is that we can use calculus to calculate how changing each weight will change the loss function, more specifically, we can see whether increasing a weight will increase or decrease the loss function across the whole “batch” of data that we are training our network with and by how much. Using this, we can use this technique on all our training data to calculate how we should change each weight to reduce the loss function over every training example. We can then change all the weights accordingly, making them each a small amount higher or lower and then repeat this process where we run our training data, calculate the loss function etc. perhaps hundreds of times until our network stops improving. There are many types of “stochastic gradient descent” algorithms, each with a different method of how exactly we calculate how much we should change each weight by, however they all are pretty much variants on this same basic concept and all get the network to “converge” onto an effective one relatively quickly.

In the above paragraph, I slightly glossed over the idea that pretty much all stochastic gradient descent type algorithms have “hyperparameters” (named so because they cannot be trained) that need to be set to train the network effectively. In older algorithms, the most important of these used to be a “learning rate” parameter which defined how much we change our weights, if it was too low our network would take too long to train, whereas if it was too high we may over-adjust our weights , meaning our loss would not actually decrease to as low as it could go. Newer algorithms, such as the “adam” optimisation algorithm detailed in (Kingma and Ba 2015) build on this by having a different learning rate for each weight in our network and adapting this to optimally adjust our weights as we train our network. The “adam” optimisation algorithm is currently thought of as the default optimisation algorithm for machine learning problems.

Quick sidenote:  
stochastic gradient descent algorithms are also referred to as backpropagation, as to calculate the gradients for each of the weights we need to calculate the gradients of the weights of the layer closer to the output of the network first. In this way, people often find it useful to visualise gradients as flowing from the output of the network down into the deeper layers, this will likely be useful in the next section.

##### problems with training deep neural networks

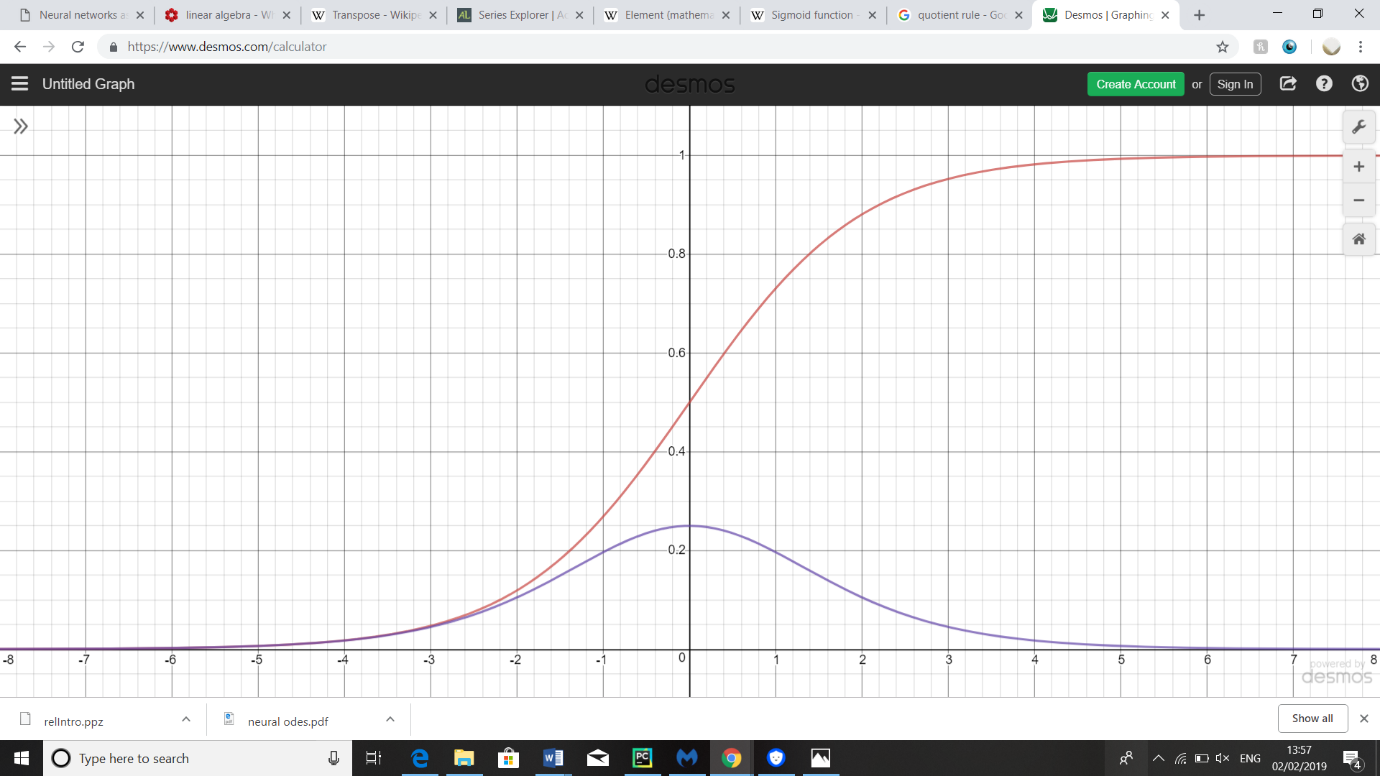
In general, when one tries to increase the size of a network to hopefully increase its accuracy, one will first try to make the network “deeper” (meaning increasing the number of layers) rather than “wider” (meaning increasing the amount of neurons per layer). This is because in general, deeper networks are better at abstracting the patterns in the data whilst wider networks tend to be more prone to simply memorising the training data you give them (this is more widely referred to as “overfitting” in the machine learning world).

Figure 7: a graph of the sigmoid function (in red) and its derivative / gradient function (in purple)

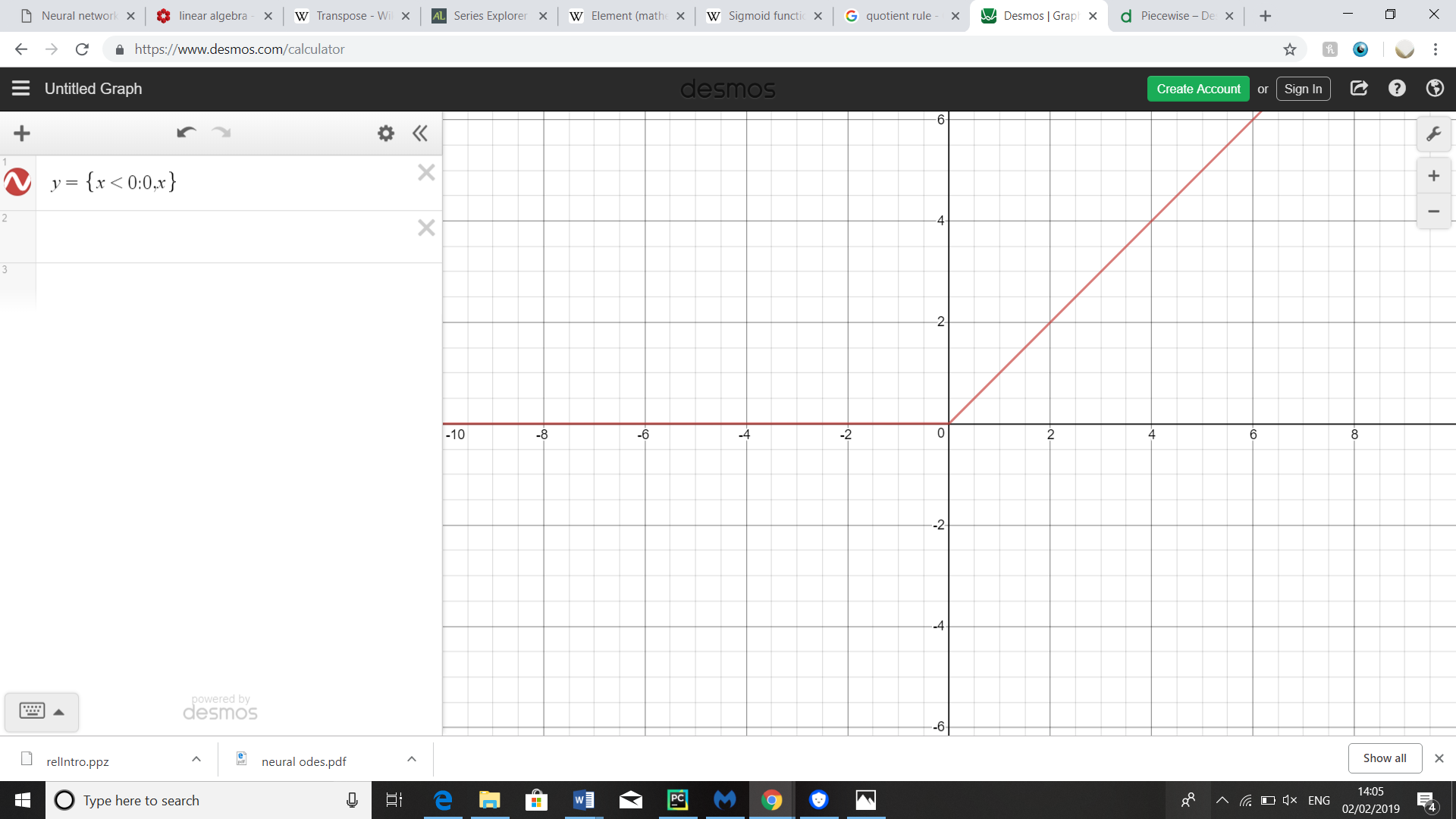
As already mentioned, applying activation functions to each layer after we calculated the results of the matrix multiplications is necessary to make the output a non-linear transformation of the input. The most popular activation function used to be the sigmoid (a graph of which is shown in Figure 7 in red) as it kept the outputs in the range [0,1], meaning it wouldn’t result in an enormous, practically untrainable output when the weights are initialised randomly. The problem with this is that the maximum gradient of the sigmoid graph (a graph of which is shown above in purple) is ¼ in practice, this means that a small change in the activations in a layer closer to the input which multiple sigmoid functions will be applied to calculate the final output will have a much smaller effect on the output than a small change in the activations closer to the output. This is a problem for our stochastic gradient descent algorithms which mathematically assume all the activations in each layer have roughly the same effect on the output. This means that if you increase the number of layers in a network with sigmoid activation functions past roughly 3, it doesn’t improve much in accuracy at all, as it is nearly impossible to train the deeper layers.

Figure 8: graphs of the ReLU (left) and leaky ReLu (right) functions

To fix this, in more modern neural networks nowadays it is more popular to use a Rectified Linear Unit (commonly abbreviated to “ReLU”) activation function, which also comes with the benefit of being much simpler mathematically, as it is equal to x if x>0 and zero (or alpha\*x where alpha is small in a leaky ReLU function) otherwise, illustrated by the graphs on the left. Using these activation functions will make vanishing gradients less likely to occur; however, they can leave a network vulnerable to the opposite problem, called exploding gradients. This is where if during training or initialisation some of the weights close to the output get too large, thus leading to changes in weights in deeper layers of the network having a much larger effect on the output than changes in other weights, meaning our stochastic gradient descent algorithm is prone to over-adjust said weights and our network will likely not reduce its accuracy in training.

##### modern solutions to training neural networks

Training deep neural networks with stochastic gradient descent type algorithms seems to be a fundamentally unstable process, as gradients tend to exponentially collapse/explode depending on whether the gradients tend to be greater or less than 1. To solve this problem, typically “batch normalisation” is performed on the activations before they are passed through the activation function, this is where the activations have a parameter typically named ”beta”(which is initialised as the mean of all the activations across all the training data) and are divided by “gamma” (initialised as the standard deviation, which measures how spread out a dataset is). Mathematically, this ensures that the gradients of our weights will themselves be around 0 and have a standard deviation of 1 to begin with, however we can also learn these “beta” and “gamma” parameters through stochastic gradient descent to allow them to adapt to more optimal values. With these techniques, people were able to increase the depths of state-of-the-art networks roughly 20 layers and still see increases in accuracy after training them…

…However, in 2015 researchers at Microsoft made another breakthrough, published in (He et al. 2015), creating what they called “Residual Neural Networks” (Resnets) with which they were able to effectively train a network with hundreds of layers, to win the Google ImageNet competition. The idea behind them was to introduce “skip connections” every two or so layers, meaning the values for input of one layer would be directly added to the input of another layer two down from it. This also helps to combat vanishing or exploding gradients as it means that the activation values even very close to the input will have a significant effect on the inputs and therefore outputs of layers even one hundred layers downstream, in a sense every layer is forced to add something new to the output during training.

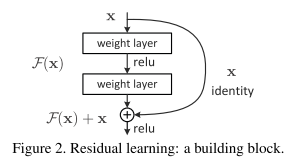
ResNets have been even further improved on even more recently with Neural Ordinary Differential Equations, however I feel implementing such cutting-edge technologies would be beyond the scope of this fifty-hour project. Especially given that DeepMind’s AlphaZero AI “only” used a forty-layer deep ResNet to accomplish Its incredible results.

Figure 9: a data flow diagram showing how a residual block would work as part of a neural network

##### convolutional Neural Networks

In the basic example, all the layers in the neural network are “dense” layers (meaning every neuron in one layer is connected to every other neuron in the layer in front of it); however, different neural network “architectures” have been shown to be more effective in certain machine learning tasks. One of the most popular of these different architectures is a Convolutional Neural Network (CNN), which is applicable to problems where there are adjacent cells of data; most commonly being pixels in an image but also being the main architecture used in AlphaZero when analysing a Chess board.

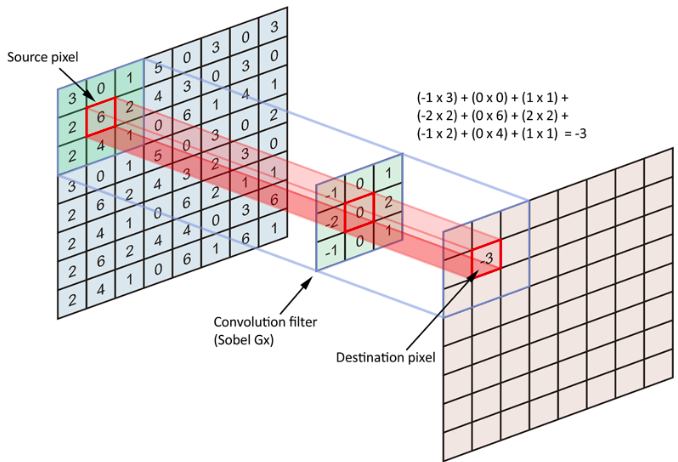
A CNN is built up of convolutional layers, which can be visualised as a set of 2D grids of cells (each of which store values and can be thought of to be like the activations in a dense network) all with the same dimensions. The values of the cells in a grid in the next convolutional layer are calculated by multiplying each of the values by the same 33 “filter” (which can be thought of as containing the weights of the network) as shown above and sliding this filter over the whole grid to calculate all the values of the cells in this grid. Typically, there will actually be multiple filters per layer, which will create multiple grids per layer, and each filter for the next layer will be connected to every grid in the preceding layer. The reason why CNNs have been found to be so effective at image recognition tasks is that they force the network to make “local abstractions”, for example recognising edges and points in the earlier layers and using this information to recognise features in later layers, while still keeping the parameter count low to avoid overfitting.

Figure 10: representation of how a filter is used to calculate the value of a cell in a grid

### Specification

I have decided to split my specification into two parts: the AI and the front end application that goes with it. Each part is ordered in terms of importance

1. The AI should be able to:
   1. Be able to learn to play connect4 independently, based on only the input of the game rules, this includes:
      1. Collecting data from playing games against itself and the rewards it receives
      2. Using this data to train a neural network to estimate the policy and value of different actions and states
      3. Learning other parameters such as when to optimally cut off calculation
   2. Choose moves based on:
      1. The state that the game is in
      2. How much time it and the other player have remaining
   3. Be reasonably proficient once fully trained:
      1. Be able to beat a simple MiniMax algorithm consistently
      2. Show clear improvements against lesser-trained versions of itself
   4. The simple MiniMax algorithm should
      1. Be able to play reasonably proficiently, including looking a certain number of moves ahead to try to force a win or draw
      2. Use techniques such as hash tables and pruning to speed its decision process up
2. The GUI application should be able to:
   1. Play timed games between two players, where a player can either be a human player interacting with the GUI or a chosen AI, including:
      1. Displaying a representation of the board on the screen
      2. Being easy enough to use that a naïve user could play a game
      3. Displaying a live representation of the timer
      4. Displaying live information about the game such as whose move it is and if someone has won
   2. Enable a naïve user to interact with the game on screen whilst it is not being played, including:
      1. Enabling the placing of counters and having a “clear board” functionality
      2. Allocating different amounts of time to each player
      3. Being able to select a different type of player to play for red and yellow and having functionality to play from the current position displayed on the board

## Design

##### coding language and library

The first decision I must make is whether I am going to use a library to do the processing for the machine learning side of my project. The cons of using a library is that all of the processing is done “under the hood”, meaning it may be harder to diagnose problems should the neural network not learn like it is supposed to; on top of this it may make it hard to customise the training of the network (such as using a custom loss function) which is applicable to this project. On the other hand, using a library would enable me to focus more on the actual application of the machine learning techniques (such as the data collection, search algorithm etc.); which I believe to be the core of the investigation. On top of this, a library would probably be written in a very efficient manner, taking advantage of the fact that most machine learning operations are simple matrix operations such as dot products which have been heavily optimised in CPU instruction sets and interfaces with the graphics cards such as CUDA. Overall this would lead to performance improvements (and therefore a reduction in the required computational resources) which I do not have the technical skills to re-create. With all this taken into consideration, I have elected to use a pre-compiled library to do the bulk of the neural network processing such as training and making predictions.

I have already narrowed down my options for the coding language and library that I will use for this project down to two options: using Python and a machine learning package made by Google called TensorFlow or choosing Java as my language and a lesser-known library called Deeplearning4J. I have narrowed my options down to these two as I don’t have much experience with other languages and think it would be beyond the scope of this 50-hour project to learn a new language to complete it. These two libraries seemed like the best choices for the deep-learning type of machine learning that my project is based on for their respective languages; and whilst I could, for instance, use TensorFlow with Java there is a lot less support (in terms of online information etc.) for such a combination which could complicate my project significantly as I likely would not have the technical skills required to fix any problems to do with this part of the project. The pros of using Python and TensorFlow is that they are currently the most popular combination for data scientists and people interested in machine learning; mainly because Python’s NumPy module is very efficient at manipulating “n”-dimensional arrays. This also means that there will be a lot of more online support and information available, which will be especially useful as I don’t have any prior experience in using either of these libraries. I am currently more comfortable with using Java, especially when programming relatively large projects such as this one, however I think that the pros of TensorFlow and Python definitely outweigh this and thus I have elected to use them in this project.

### Overall Coding Style / Paradigms

I have identified four overarching programming philosophies or paradigms that I will try to implement in this project to maximise the effectiveness, maintainability and completeness of my technical solution. These are:

##### Object oriented programming

My finished project will be of a reasonable size, and therefore writing it using a modular, object-oriented paradigm will make it much easier to maintain, develop and test. This means that I will therefore structure my code using groups of classes all interacting with each other through their interfaces.

##### Pythonic coding style

To make my technical solution maintainable and easy to work on, I need to maximise the readability of my code. Luckily, python is built to be written in a certain “pythonic” style that is very readable and should “self-comment” to a certain extent (Although comments are obviously still essential to help with this). To ensure that my code fits with this style I am going to attempt to follow python’s pep8 style guide when writing my code, along with the philosophy laid out in the “zen of python”. One important note to make in my choice to use the python language is that it doesn’t have explicit public and private methods, variables, interfaces or abstract methods, so I will have to make sure that I make it clear when something is an instance of these. Other quirks of the python language include the ability to change an objects default get and set methods, meaning it is not considered “pythonic” to create getters and setters to encapsulate data.

##### Defensive programming

Although defensive programming is not as important in a self-contained investigative project as in a system intended for public consumption, it is still a good habit to get into and will likely make development easier for me as I will encounter fewer bugs etc. To do this, I will design my system (especially the GUI) in a way which “whitelists” any user input to validate it. On top of this, I will need to validate that all the data that is being passed to the machine learning side of the project through testing, as if this data is erroneous it will generally lead to “silent errors”, meaning it will likely result in the AI producing disappointing results rather than an exception being thrown.

##### event driven GUI

In order to make my GUI work efficiently and keep in line with all the other ideas outlined here, it will need to follow the “event-driven” paradigm. This means that processes should be executed solely as the response to a user input. I anticipate that this may make it tricky to interface with the training section of my program, which will likely be executed in the standard fashion.

### General Algorithm design

##### board manipulation

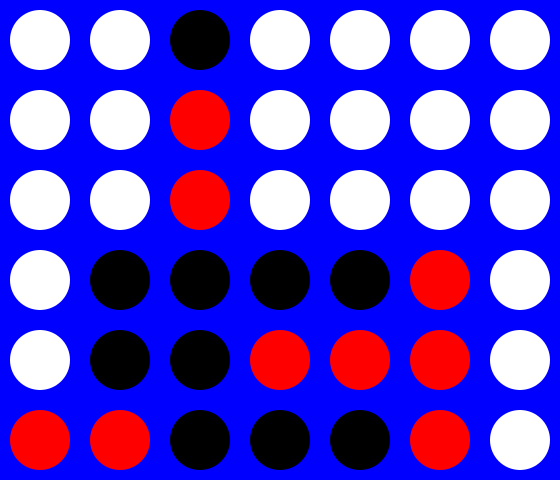
My finished project will need to be able to do the following checks/manipulations on the object representing the board. Ideally, I would maximize the efficiency of said algorithms as the speed of any AI algorithms will be affected by how quickly they can manipulate the board to check different moves.

Placing counters given a column selection

Check if placing a counter in a column is a valid move (check that a column is not full)

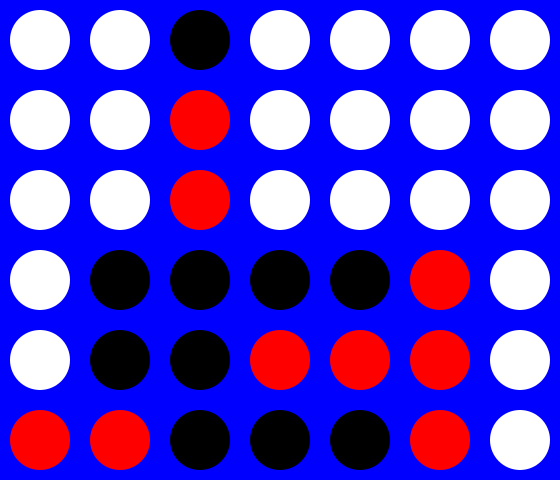
Check if placing a counter in a given column results in a line of 4 like-coloured counters

I have elected to use a 2-dimensional array (that will be wrapped in Board a class) to represent the board as this seems like the most elegant data structure to encode the positions of a grid of counters and will make implementing the above algorithms simple. Checking if a move is valid is relatively trivial: one simply needs to check if the top cell in that column is empty. Similarly, placing counters given a column selection could be done by checking up the column until one finds an empty cell and then placing a counter there.



I have two potential solutions for a win check algorithm:

Algorithm 1 includes checking every row, column and most of the diagonals (we can exclude the diagonals of less than length 4) for a row of counters. This could be implemented using nested for loops looping over the array with different increment conditions (to represent whether the line was a column, row or diagonal), keeping a counter for the number of like coloured counters; which is reset every time we run into a counter of the opposite colour, an empty cell or start a new line check.



Algorithm 2 will require the information of the position of the cell where the previous counter was placed. It will then check from this cell along each row, diagonal and column until it runs into an empty cell or counter of the opposite colour. A count can be kept for the number of like coloured counters in a given direction and True can be returned if this number becomes greater than 4. This algorithm can be further optimised by not checking a diagonal if the cell is in one of the corners of the board or the column if a cell is in one of the top three rows.

Figure 11: representations of the how win check algorithms would work. Algorithm 1 is shown above, the sections of the board in algorithm 2 show places where different checks would not need to be carried out if the previous counter was placed there.

The benefit of using Algorithm 1 will be that it can be used at any time in the program as the information of where the last counter was placed is not required; on top of this it assumes a lot less about how the counters are placed (for instance it does not assume that no counters are above the previous counters’ position), meaning it will be more in line with the idea of defensive programming. This said, I will elect to use algorithm 2 as it performs many fewer checks, making it much more efficient, on top of this, the counter position could be returned from the place counter method and I do not see a situation where I would need to preform a win check without placing a counter first; meaning the increased versatility of algorithm 1 is not that important.

game execution

Another problem I came across whilst conceptualising the system was how games are executed. The obvious initial solution to me was to have an interface for a player (human or machine) that contains a method that returns the column that the player has chosen given the current game state. I then thought that I could have a game class that loops for each move, getting the chosen move from the current player and then updating the board and user interface. The problem with this is that it violated the idea of the event driven GUI design as for a human GUI player, the choose move method would have to make buttons appear, and leave the thread hanging until one of the buttons is clicked.

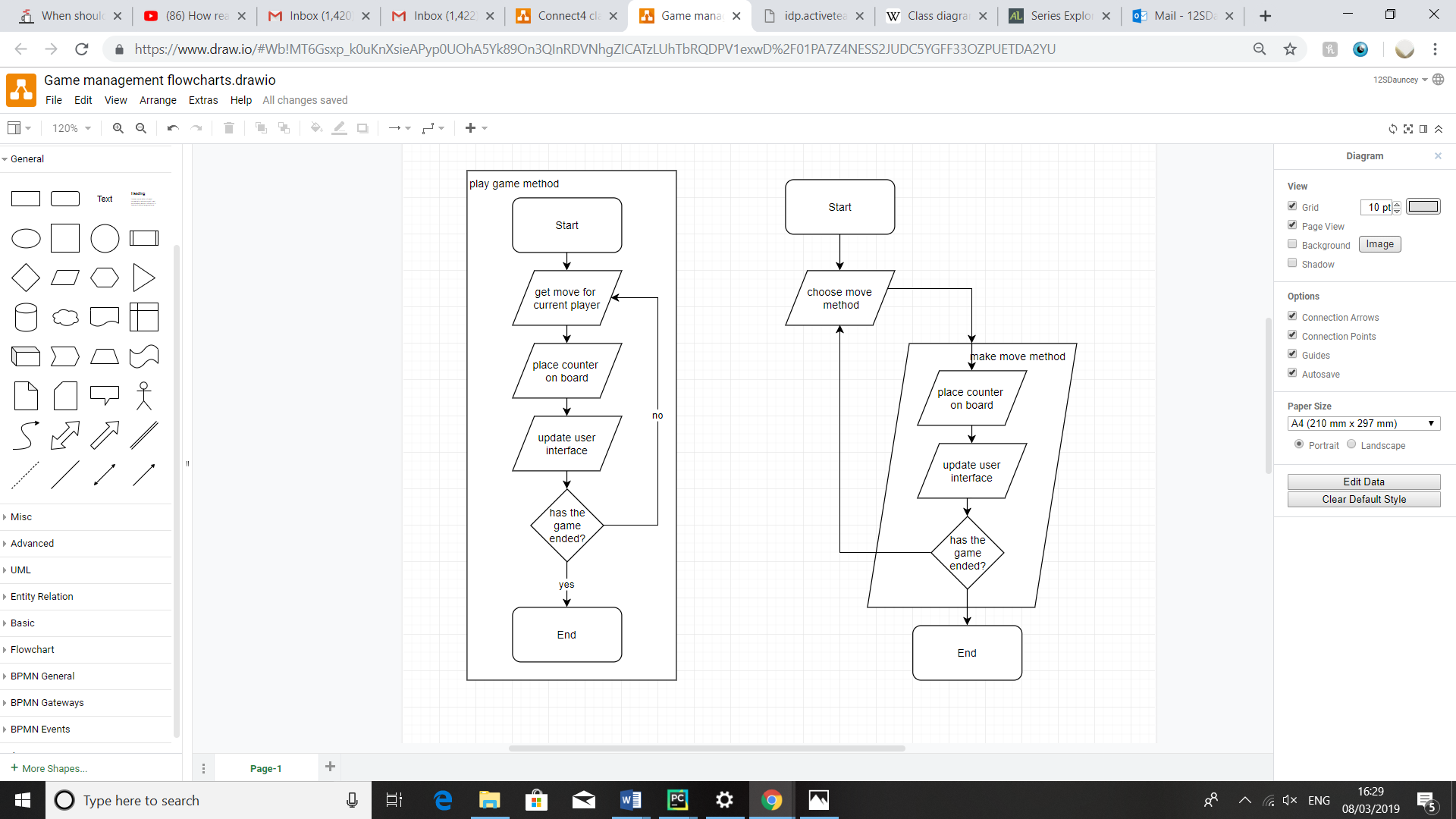
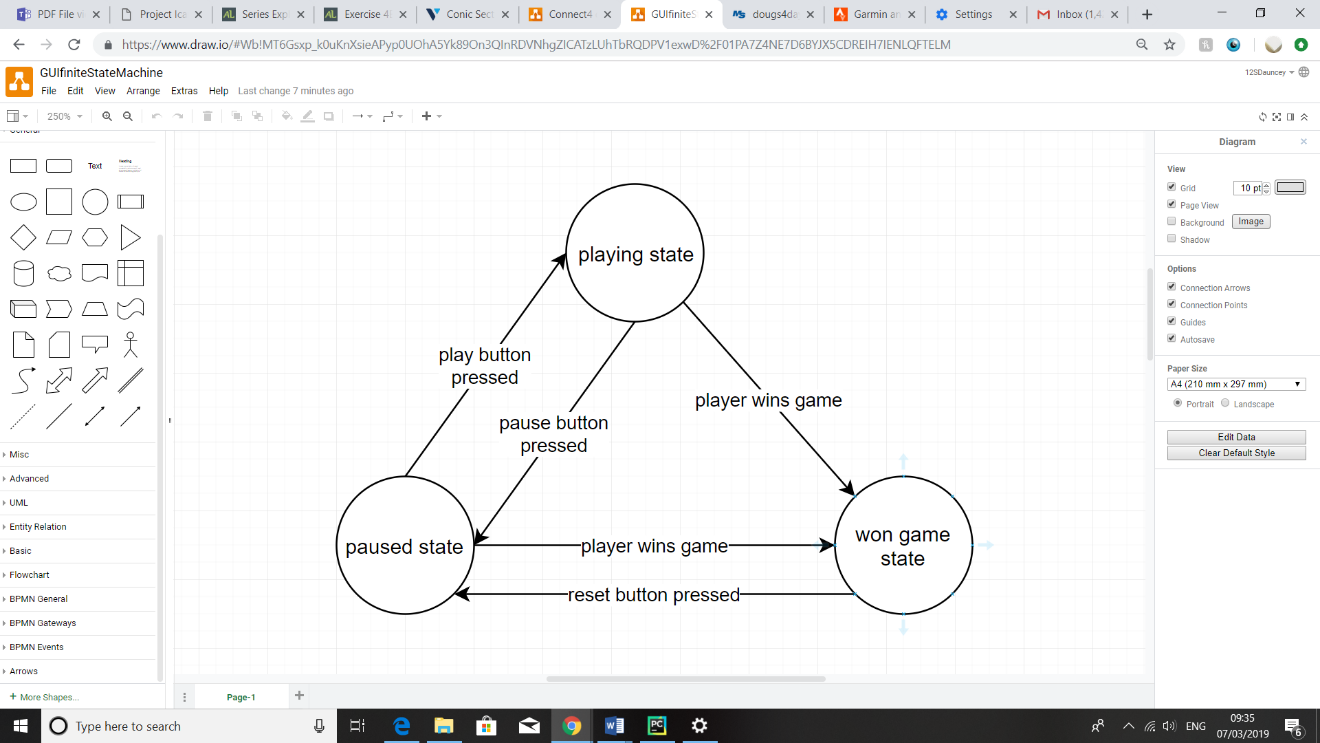
The other solution I had was to instead have a game state object, which would have a make move method which would call the choose move method for the next player after it had finished executing; the choose move method would then recursively call this make move method once its decision had been determined rather than returning a value. This means that the choose move method for a human gui player would be able to show some buttons which have the make move method as their “on-click” method; wich is much more in-line with the event-based paradigm.

Figure 12: flowcharts representing my initial solution on the left and the second solution on the right

I have decided to go with this second solution due to this, even though the end result may be that the program uses more memory due to the recursive nature of the second solution (as there will be more stack frames).

### GUI design

In line with the paradigm of event driven GUI design, I have decided to model my GUI as a finite state machine, shown in Figure 13 where the user’s inputs transition the GUI from different states. This will force me to consider what the user should be allowed to input from a given state, making my programming defensive. I have decided that giving the GUI three states will give it the functionality to achieve the specification, these being:

A “paused game” state that allows the user to edit the board and information about the game such as the players that are currently playing, how much time each player has, etc.

Figure 13: a finite state machine representing the function of the GUI

A “playing game” state where two AI objects can play against each other, with the information about the game being displayed but not able to be edited

A “won game” state where information about the game is displayed and a button can be pressed to reset the game to a paused state with a cleared board

From this, I have devised a list of widgets that my GUI will need, and have sketched out a design for the interface in Figure 14:

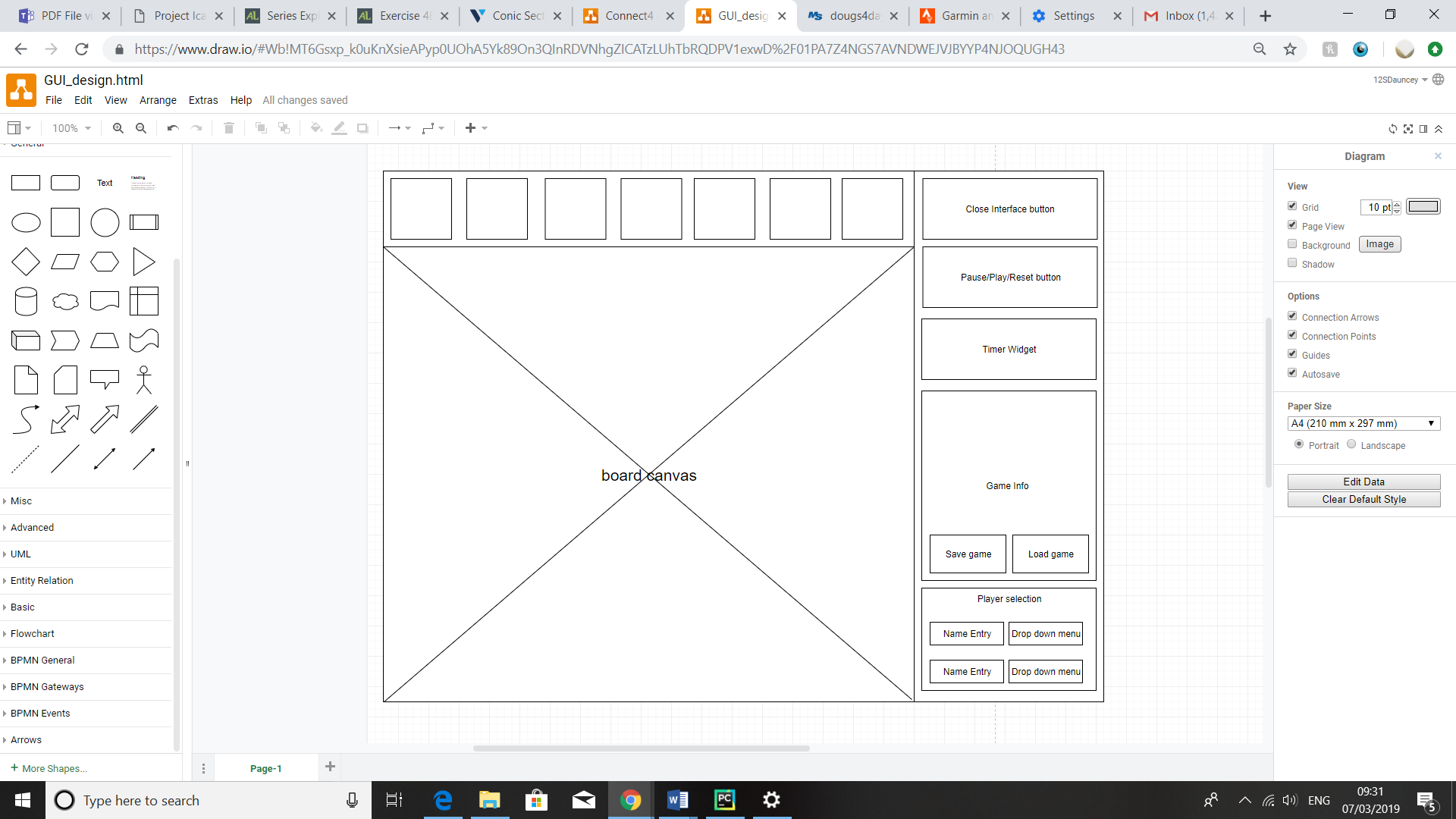


Figure 14: A sketch of the GUI layout

What each widget will do:

* Player move buttons - An array of buttons that appear in the paused game state that enable the placing of counters on the board. In the playing game state, if it is a human AI object’s turn, they will appear for the human to play their chosen move.
* Pause/Play from here/reset button - Transitions from the paused to the playing state and vice versa. In the won game state it should reset the game and transition the state to paused
* Timer widget - In playing state and won game state just displays how much time each player has left. In paused state serves as a text entry widget for someone to enter a certain amount of time for a player to have.
* Game Info – In all states just displays info about the game, such as a move list for each player, whose turn it is, and the winner.
* Player Selection - In paused state, enables editing of who’s playing/name of player, changing red and yellow objects as well as loading parameters for trainable AIs. In playing and won game states, just displays this information

### AI algorithm implementation

##### UCT search

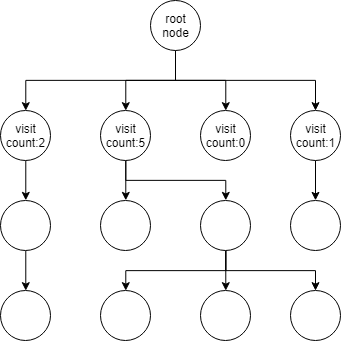
Have already explained how much of the UCT tree search algorithm will work in my analysis; however, I glossed over some key details that I will need to implement to get the algorithm to work properly.  One detail I missed out was how a move is selected once the AI has played a sufficient amount of simulated games to generate a large enough tree of nodes. When the AI is playing “deterministically”, the chosen move is simply the one that corresponds to the node that is child to the root node and has the highest visit count. The problem with always playing deterministically is that it would mean that the AI would constantly play the same set of moves when playing a batch of self-play games, which would lead to the neural network overfitting on those games each time we train it. To get around this problem, we play the fist few moves of each game semi randomly; to do this we generate an array of values called the “improved policy”, by normalising the array of the visit counts of the child nodes. This means that we take each visit count and divide by the sum of all of them, returning [,] for the example in Figure 15. We can then treat this array as a probability distribution and then sample from it to generate our chosen move, again using the example this would mean there would be a 5/8 probability of picking the second-from-right node.

Figure 15: an example of a UCT tree after performing 12 simulations. the lower nodes' visit counts have not been included for clarity.

As I have already mentioned, the neural network is used to output a single number for the estimated “value” and an array of length seven for the estimated “policy” for each given game state; this means that I will need to generate “labels” for each of these outputs if I want to train the network to output useful information. The most obvious thing to use as a label for the policy estimate is the improved policy, as this is generated by looking a few moves deeper and thus will generally be better than the policy estimate; meaning the network should gradually learn to produce a more and more intelligent policy estimate.

The label that was used by DeepMind for their value estimate was simply the reward given to the AI at the end of the game (1 if the AI won from this position and -1 if it lost). The idea behind this is that the network should learn to output a value in the range [-1, 1] as what it expects to be the average reward if it plays from this position many times. To see why this is the case, we could imagine a position which is completely even for both players; ideally, when given this position, the output of the network for the value head should be close to zero; however, one might think that if we only train the network with labels of 1 or -1 it will learn to output numbers very close to 1 or -1 even in drawn positions as it might as well gamble on the 50% chance of getting the right answer? The key here is that if we use a mean-squared error loss function, the average loss on the value head if the network continuously predicts 0 for this even position will be one, whereas if the network was to randomly pick between 1 and -1, the average loss would be two. Since the network is trained to minimise the loss across a whole batch of these training examples, this should hopefully mean that it won’t learn to simply guess 1 or -1 for the value head on every input.

##### Timed UCT

As I already mentioned, in this project I will be Adapting this UCT algorithm so the amount of computation dedicated to each move can be modulated depending on how sure the AI is of a given move. The idea here is that, in UCT search we already calculate a confidence interval for the value of a given state; meaning we know how sure we are of a valuation of this state. This means that we can calculate a rough estimate of, on average, how much value will gain from performing more computation. This is done by subtracting the Lower confidence bound of the child node with the highest average value from the upper confidence bound of the other nodes (and limiting this value to being zero instead of going negative) and summing this value across all the root nodes’ children. I have called this the “total confidence interval overlap”, which can be combined with information about the game to give a value scoring whether the AI should perform more computation:

I could then compare this value with a constant, where if it is greater than the constant another simulation is performed, otherwise the computation for this move is cut off. This constant could be learned through increasing it slightly after very game but then dialling it back substantially when the AI times out; meaning its value should balance out over time.

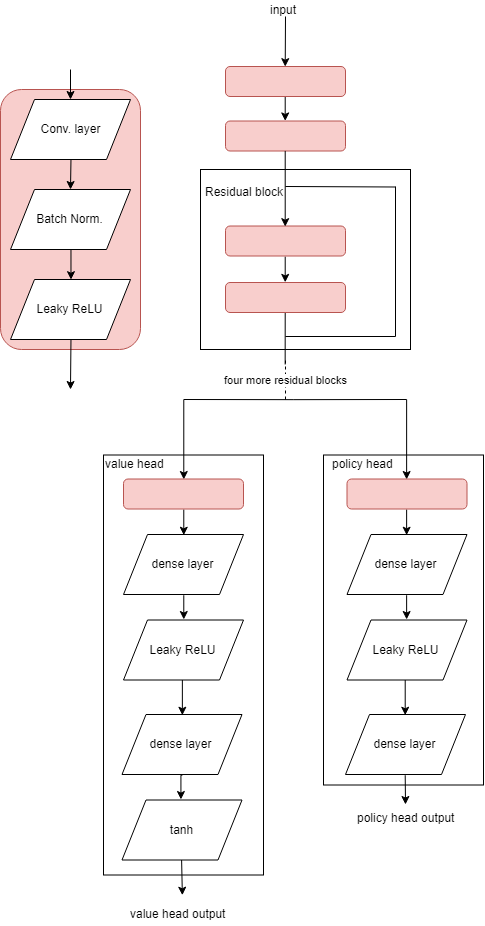
I also plan to pass the neural network an input detailing how much time each player has left so that it could hopefully formulate strategies to do with how much time each player has (for instance, if the enemy player is low on time, the AI may choose to play a more drawn-out game to get its opponent to timeout). This input will likely be two extra 67 arrays containing how much time their corresponding player has left divided by the total amount of time that player was given in every cell.

##### Training environment

As already discussed, the AI will be timed when it is making its decisions in training so that it can learn how to budget its thinking time most effectively. The most obvious solution to this is to measure how much time it took from when the “choose move” method is called to when the AI comes back with its decision, however this leads to a problem, which is that the time a given amount of computation takes varies on a number of factors such as background processes that are running, how hot the processor is, etc. This means that if the training games were to be played using absolute time, the AI could end up being punished for things beyond its own control, such as a piece of computation taking longer than expected. To solve this problem, I have elected to use a system of computation “tokens” in the training environment, in which each AI keeps track of how many tokens it has used and the game treats these tokens like the time left. Luckily, one simulated game in the UCT search algorithm takes a roughly constant amount of time as the main bulk of computation occurs at the end of the simulation when generating the neural network prediction.

This means that I should hopefully be able to then feed the data of how much time each player into the AI in a similar way to how it was given the data of how many tokens it had remaining in training, and thus the AI will be able to play timed games despite training with games where the absolute time taken was not measured

##### Neural Network specifics

The main purpose of this project is not to assess the proficiencies of different architectures as this has already been investigated by others in detail. I have therefore elected to use a Residual network architecture very similar to the one DeepMind used for AlphaZero (albeit scaled down to account for the lower amount of resources I have available to me) as this type of network has already been tried and tested for this type of task. This said, the nature of machine learning in its current state is such that the architecture I have sketched out in Figure 16 will likely need some tweaking to get it to work effectively.

I will also need to do some formatting of the input to the neural network to make it possible for it to recognise patterns in the data that it is fed. The input that I will give the network will be a 6x7 array containing values of {1, 0, -1} the reasoning behind this is that many of the inner workings of the optimisation and gradient descent algorithms assume that the input data has a mean of roughly zero and a standard deviation of roughly one. Luckily, this was how I was thinking of representing the board apart from one key difference: in the representation of the board in the rest of my program will use 1 to represent a red counter and -1 to represent a yellow counter. However, if I want my AI to learn I need to give it the board from the perspective of the current player, this means using 1 to represent counters for the current player and -1 to represent counters for the enemy player. This means that when I pass the board to the neural network, I will first have to check whether it is yellow’s move and “invert” the board if it is.

Figure 16: a sketch of the Neural Network architecture I plan to use

##### mimimax algorithm

As already discussed, I will program a MiniMax entity into my program as well to test the trained AIs against. I have already discussed much of how this algorithm will work, the only thing I intend to implement that has not already discussed is a “transposition table”. This is a HashMap of positions that have already been analysed by the algorithm along with the rating it gave them, this will allow it to re-use the calculations it has already made by looking up every board it encounters on the transposition table before taking the time to analyse this board. The reason why the algorithm may encounter the same board multiple times is because different arrangements of the same moves could produce the same results.

##### training example container

One problem I encountered when thinking about the training infrastructure was how I was going to feed the network a stream of data that it would be able to learn from. The most obvious solution to the problem of what data to feed the network is to play a set of games and, create a batch of training examples based on every position in these games and then train the neural network on this batch. The problem with this approach is it leads to having certain types of training example overrepresented in the batch, (for example, if in one of the games there happen to be lots of counters placed in column 7, there will be roughly 20 training examples showing this specific arrangement of counters). This could lead to overfitting, as the model could just learn to recognise this specific game and output the result.

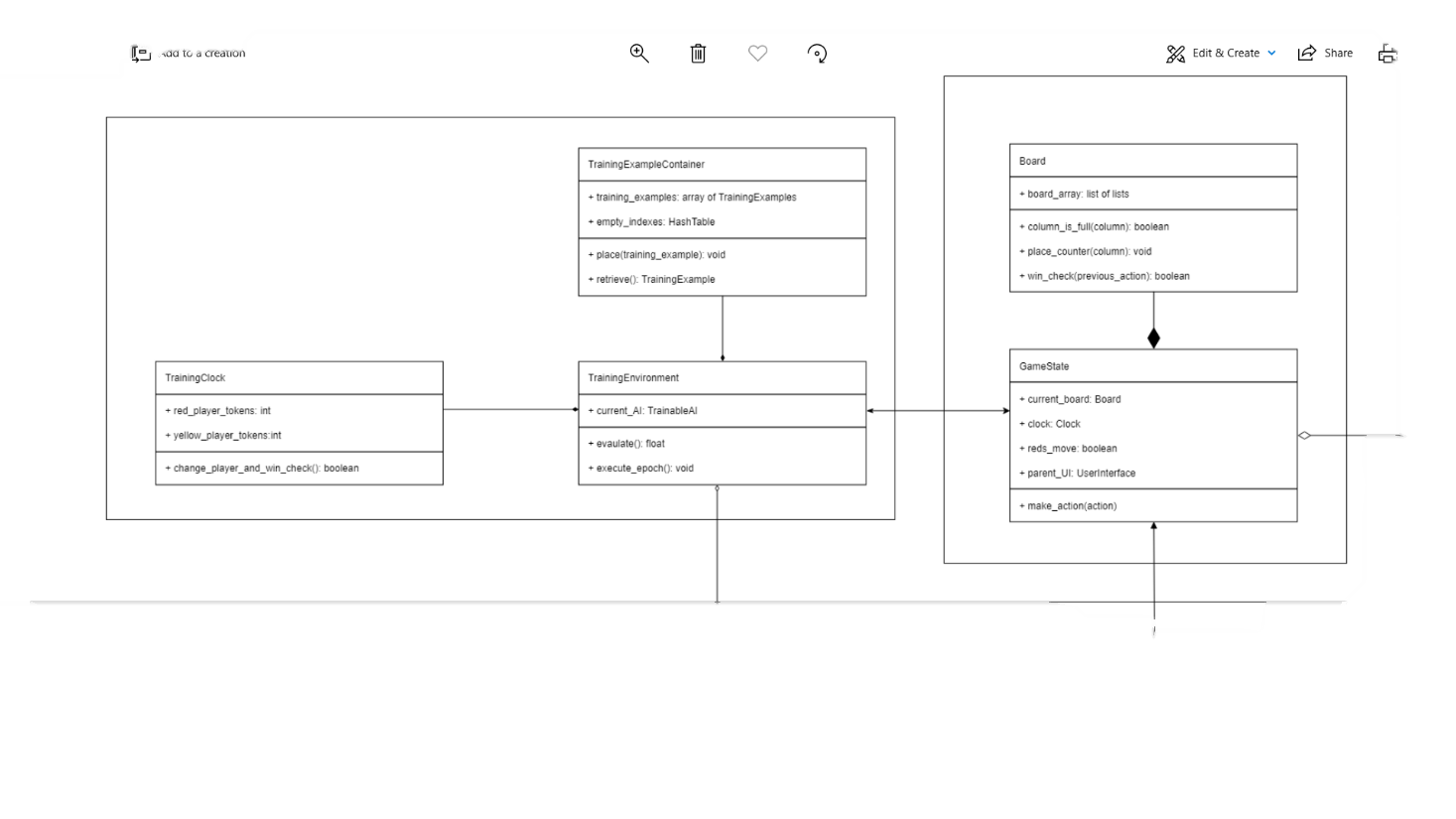
One solution to this problem would be to generate training examples from randomly sampling one position from each game and throwing the rest away; I think this is the method that was used by DeepMind from their paper, however it was not one hundred percent clear in their paper. However, this method would increase the amount of computation needed to generate a batch by an order of magnitude, making it not very practical for me. Instead, I have elected to try and combine some training data from the newer games with training data from slightly older games, one important note is that we want training data that is generated from a relatively new version of the AI as this will allow for the AI to build on the new intelligence it has gained.

To do this I have designed a training example container data structure that works by having a pool of training examples from which a batch of randomly selected training examples can be popped before the pool is refilled with new training examples; this will mean that the probability of a given training example “surviving” n of these cycles will decay exponentially as n increases. Because this container will be handling large volumes of data, I have attempted to make retrieving and placing a given training example O(1). This means that when I pop a randomly selected example from this array of training data, I will not be able to simply shuffle all the data along the array as this would make the algorithm O(n), so I will therefore have to maintain a hash table which keeps track of the indexes of the “holes” in this container. This information could then be used to place new training examples in these “holes” with constant runtime.

### Class Structure

To finish my design, I have decided to draw a UML class diagram to represent all of the components of the project, this will ensure that I am not overlooking any areas of my project by breaking it down into small segments of code which I am confident that I can write. In line with the object-oriented paradigm that I am trying to adhere to, I have selected to split my project into four modules, one to handle the game, board etc; one to handle the AI, one to handle training and one to handle the user interface.

##### training and game modules



UserInterface interface

Figure 17: a UML class diagram for the training and game modules

Trainable AI interface

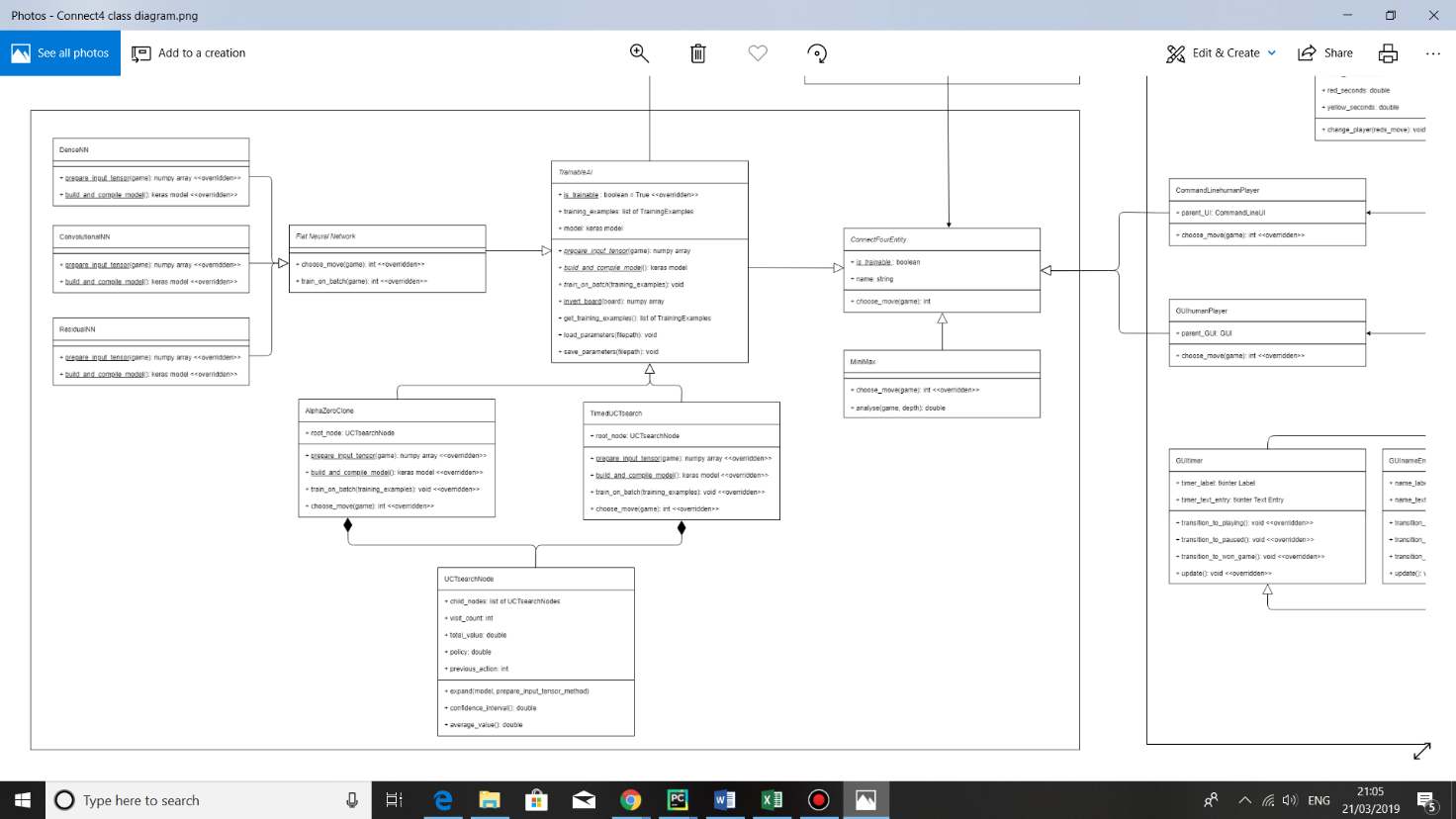
ConnectFourEntity interface

Trainable AI interface

Notes:

* I am considering creating a separate “TrainingGameState” class which inherits from the GameState class, depending on how clean implementing the code turns out as-is.

##### AI module



GameState class

TrainningEnvironment class

HumanPlayer classes

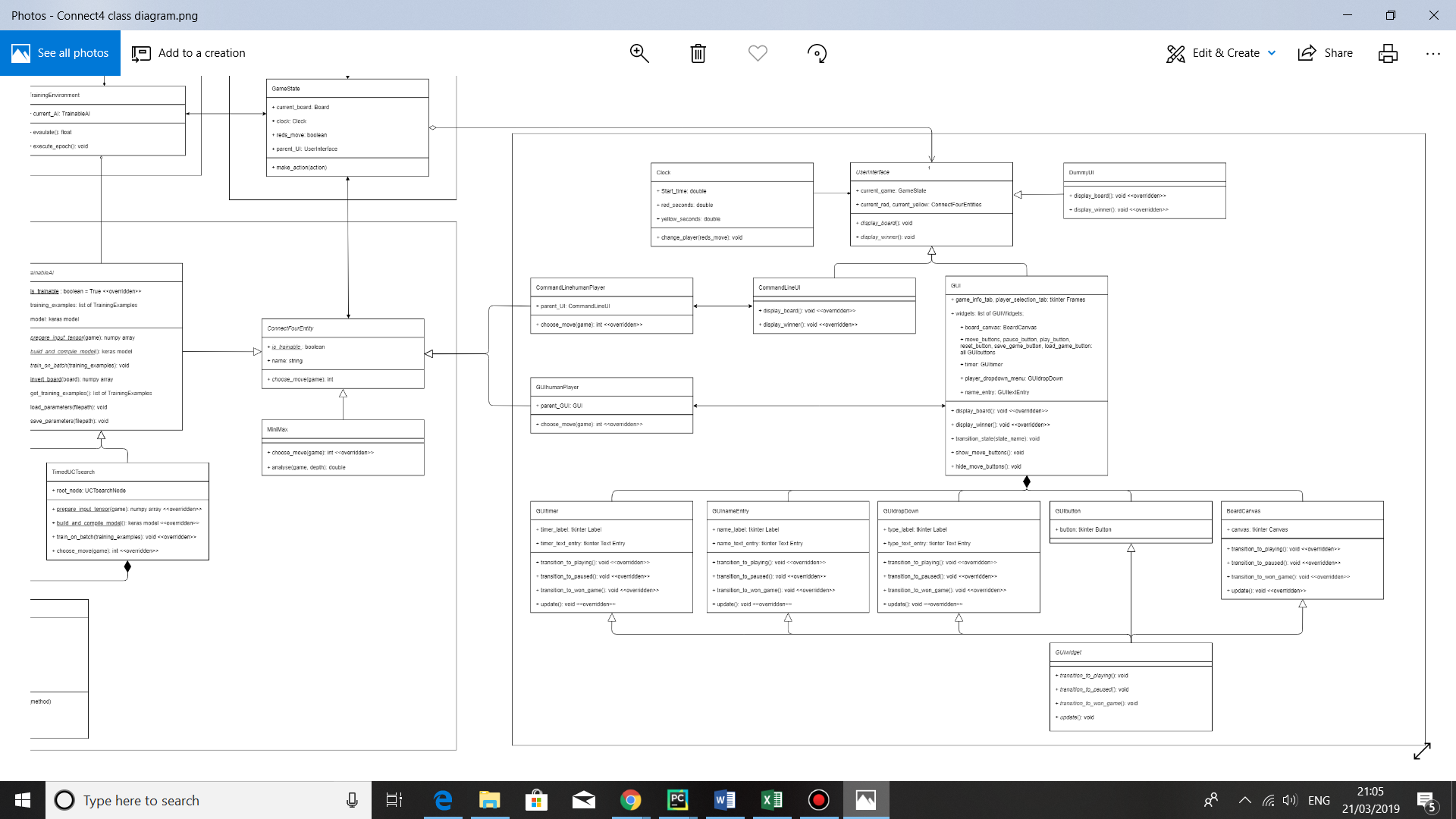
Figure 18: a UML class diagram for the AI module

Notes:

* A “flat” neural network simply analyses the each of the positions resulting from the possible moves of the current game state and the chooses the position it thinks is best for it.
* I have decided to create both flat neural networks and then an AlphaZero clone before moving on to my timed UCT algorithm. This should hopefully give me practice with applying reinforcement learning as I increase the complexity of the algorithms I am implementing. This should also allow me to compare these algorithms against the timed UCT to see if it actually ends up playing more effectively.

##### User interface module

GameState class



ConnectFourEntity interface

Figure 19: a UML class diagram for the user interface module

Notes:

* I have decided to use the inbuilt python library (“Tkinter”) to write most of the GUI as there are many learning materials available for it.
* I decided to create a GUIwidget class that all the different widgets I need will inherit from so that I will be able to loop over all of them whenever I need to transition the state of the user interface.
* I have also decided to create a command line user interface to help with the early stages of testing.

## Technical Solution

### Python Files

##### connect4game.py

**class** FullColumnException(Exception):  
 *"""Raised when one attempts place a counter in an already a full column."""* **pass  
  
  
class** Board:  
 *"""Stores a 6\*7 2D array of integers representing the board,  
 with 1 being a red counter, -1 being yellow and 0 for no counter."""* **def** \_\_init\_\_(self):  
 *"""initialises the board in a blank state by default"""* self.array = [[0] \* 7 **for** i **in** range(6)]  
  
 **def** print(self):  
 *"""Prints a representation of the board into the console."""* **for** i **in** range(7):  
 print(i + 1, end=**" "**)  
 print()  
 **for** row **in** self.array:  
 **for** cell **in** row:  
 **if** cell == 0:  
 character = **"#"  
 elif** cell == 1:  
 character = **"R"  
 else**:  
 character = **"Y"** print(character, end=**" "**)  
 print()  
  
 **def** column\_is\_full(self, column):  
 *"""Determines whether a column is full."""* **if** self.array[0][column] == 0:  
 **return False  
 else**:  
 **return True  
  
 def** place\_counter(self, reds\_move, column):  
 *"""Places a in a given column,  
 returns a tuple representing where it was placed."""  
 # raises FullColumnException if the column is full* **if** self.column\_is\_full(column):  
 **raise** FullColumnException  
 *# otherwise, checks up from the bottom for a free cell* **for** i **in** range(5, -1, -1):  
 **if** self.array[i][column] == 0:  
 new\_counter\_position = i, column  
 **if** reds\_move:  
 self.array[i][column] = 1  
 **else**:  
 self.array[i][column] = -1  
 **return** new\_counter\_position

This is an example of where I have defined a multi-dimensional array to store a representation of a connect4 board. This has been encapsulated by a Board class that manipulates it.

Board class continued in the next page.

Continuing in the Board class:

**def** win\_check(self, reds\_move, position):  
 *"""Returns True if a player won with their last move."""* **if** reds\_move:  
 counter\_code = 1  
 **else**:  
 counter\_code = -1  
 *# checks the row of the position for line of length for* **if** self.check\_full\_line(counter\_code, position, -1, 0):  
 **return True** *# checks the column of the position, which is only  
 # necessary when the 'y' value is greater than or equal to 4* **if** position[0] < 3:  
 **if** self.check\_line(counter\_code, position, 1, 0, 1) == 4:  
 **return True** *# checks the diagonals, in some cases where the counter is close  
 # a corner of the board, checking one of the diagonals is unnecessary* position\_sum = position[0] + position[1]  
 position\_difference = position[0] - position[1]  
 **if** 2 < position\_sum < 9:  
 **if** self.check\_full\_line(counter\_code, position, -1, 1):  
 **return True  
 if** -4 < position\_difference < 3:  
 **if** self.check\_full\_line(counter\_code, position, 1, 1):  
 **return True  
  
 def** check\_full\_line(self, counter\_code, position,  
 column\_index\_iterator, row\_index\_iterator):  
 *"""Used by the win\_check method, returns True if there is a line of at least  
 four like coloured counters either way in the specified direction."""* full\_line\_length\_counter = 1  
 full\_line\_length\_counter = self.check\_line(  
 counter\_code, position, full\_line\_length\_counter,  
 column\_index\_iterator, row\_index\_iterator, )  
 **if** full\_line\_length\_counter == 4:  
 **return True** *# counts the length of the line in the opposite direction* full\_line\_length\_counter = self.check\_line(  
 counter\_code, position, full\_line\_length\_counter,  
 -column\_index\_iterator, -row\_index\_iterator)  
 **if** full\_line\_length\_counter == 4:  
 **return True  
 return False  
  
 def** check\_line(self, counter\_code, position, line\_length\_counter,  
 column\_index\_iterator, row\_index\_iterator):  
 *"""Used by check full line method, returns the length of a line of  
 like coloured counters from a point in one specified direction"""* **for** i **in** range(1, 4):  
 column\_index = position[1]+i\*column\_index\_iterator  
 row\_index = position[0]+i\*row\_index\_iterator  
 **if** -1 < column\_index < 7:  
 **if** -1 < row\_index < 6:  
 **if** self.array[row\_index][column\_index] == counter\_code:  
 line\_length\_counter += 1  
 **if** line\_length\_counter == 4:  
 **return** 4  
 **else**:  
 **break  
 else**:  
 **break  
 else**:  
 **break  
 return** line\_length\_counter

connect4game.py file continued in the next page.

Continuing in connect4game.py:

**class** GameState:  
 *"""An abstract class which manages games in play."""* **def** \_\_init\_\_(self, red, yellow, reds\_move, board, starting\_move\_list):  
 self.red = red  
 self.yellow = yellow  
 self.current\_board = board  
 self.move\_list = starting\_move\_list  
 self.reds\_move = reds\_move  
 self.is\_drawn = **False  
  
 def** start\_game\_procedure(self):  
 **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** won\_game\_procedure(self):  
 **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** next\_move\_procedure(self):  
 **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** change\_clock\_and\_win\_check(self):  
 **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** make\_move(self, move):  
 *"""Places a counter in the given column on the board."""* **if** self.place\_counter\_and\_win\_check(move):  
 self.won\_game\_procedure()  
 **elif** self.change\_clock\_and\_win\_check():  
 self.won\_game\_procedure()  
 **else**:  
 self.next\_move\_procedure()  
 current\_player = self.get\_current\_player()  
 current\_player.play\_chosen\_move(self)  
  
 **def** place\_counter\_and\_win\_check(self, move):  
 self.move\_list.append(move)  
 **try**:  
 new\_move = self.current\_board.place\_counter(self.reds\_move, move)  
 **except** FullColumnException:  
 self.red\_winner = **not** self.reds\_move  
 self.is\_drawn = **False** self.timeout = **False  
 return True  
 else**:  
 **if** self.current\_board.win\_check(self.reds\_move, new\_move):  
 self.red\_winner = self.reds\_move  
 self.is\_drawn = **False** self.timeout = **False  
 return True  
 if** len(self.move\_list) >= 42:  
 self.is\_drawn = **True** self.timeout = **False  
 return True** self.reds\_move = **not** self.reds\_move  
 **return False  
  
 def** play(self):  
 self.start\_game\_procedure()  
 **if** self.reds\_move:  
 self.red.play\_chosen\_move(self)  
 **else**:  
 self.yellow.play\_chosen\_move(self)  
  
 **def** get\_current\_player(self):  
 **if** self.reds\_move:  
 **return** self.red  
 **else**:  
 **return** self.yellow

This is an example of a Board object being composed onto a GameState object

GameState class continued in the next page.

Continuing in the GameState class:

**def** get\_enemy\_player(self):  
 **if** self.reds\_move:  
 **return** self.yellow  
 **else**:  
 **return** self.red

**def** get\_possible\_moves(self):  
 *"""Returns a list of integers representing  
 which columns can be played in."""* possible\_moves = []  
 **for** column **in** range(7):  
 **if not** self.current\_board.column\_is\_full(column):  
 possible\_moves.append(column)  
 **return** possible\_moves  
  
 **def** get\_normalised\_clocks(self):  
 *"""Abstract method used by the TimedUCT AI to return tuple containing  
 how much time each player has left on a normalised scale."""* **raise** NotImplementedError(**"Please Implement this method"**)

##### connect4AI.py

**import** math  
**import** random  
**import** pickle  
**from** copy **import** deepcopy  
  
**import** numpy **as** np  
**import** tensorflow **as** tf  
**from** tensorflow **import** keras  
  
**import** connect4game **as** game  
  
  
**class** ConnectFourEntity:  
 *"""Interface for different types of AI to play connect 4."""* is\_trainable = **False  
  
 def** \_\_init\_\_(self, name):  
 self.name = name  
  
 **def** play\_chosen\_move(self, game\_state):  
 chosen\_move = self.choose\_move(game\_state)  
 game\_state.make\_move(chosen\_move)  
  
 **def** choose\_move(self, game\_state):  
 *"""Abstract method that returns an integer between 0 and 6  
 for the column that the AI wants to play in"""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
  
**class** Dummy(ConnectFourEntity):  
 **def** \_\_init\_\_(self):  
 **pass  
  
 def** choose\_move(self, game\_state):  
 **pass**

This is an example of a method that is implemented many times (Polymorphism)

connect4AI.py file continued on the next page.

Continuing in connect4AI.py:

This is an example of a class that inherits from another class

**class** TrainableAI(ConnectFourEntity):  
 *"""An abstract class that can interact with the training module as well  
 as the game module."""* is\_trainable = **True  
  
 def** \_\_init\_\_(self, name, is\_training):  
 self.name = name  
 *# Training examples is an array that stores data from the current game.* self.training\_examples = []  
 self.root\_node\_initialised = **False** self.model = self.build\_and\_compile\_model()  
 self.is\_training = is\_training  
 *# Used computation tokes counts how many times the model has been used  
 # to generate a prediction in the current game.* self.used\_computation\_tokens = 0  
  
 **def** get\_training\_examples(self):  
 *"""Returns and clears the list of currently stored training examples."""* training\_examples = self.training\_examples  
 self.training\_examples = []  
 **return** training\_examples

**def** train\_on\_batch(self, examples):  
 *"""Abstract method that takes an array of training examples  
 and trains the AI's model on them."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** load\_parameters(self, filepath):  
 self.model.load\_weights(filepath+**".h5"**)  
  
 **def** save\_parameters(self, filepath):  
 self.model.save\_weights(filepath+**".h5"**)  
  
 @staticmethod  
 **def** build\_and\_compile\_model():  
 *"""Abstract method which returns an instance of this AI's model  
 architecture with randomly initialised weights."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** prepare\_input\_tensor(self, game\_state):  
 *"""Abstract method that Returns a numpy array representing the  
 game\_state that can be inputted into the model."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** punish\_for\_timeouts(self, timeout\_count, game\_count):  
 **pass**

**class** FlatNeuralNetwork(TrainableAI):  
 *"""An abstract class for an AI that simply looks one move deep and uses  
 a neural network to analyse the resulting positions."""* **class** TrainingExample:  
 *"""Class which stores examples from games to train the AI's model on."""* **def** \_\_init\_\_(self, game\_state, parent\_AI):  
 self.state\_tensor = parent\_AI.prepare\_input\_tensor(game\_state)  
 self.reward\_tensor = np.array([0])  
  
 **def** set\_reward(self, reward):  
 self.reward\_tensor = np.array([reward])

FlatNeuralNetwork class continued in next page.

Continuing in the FlatNeuralNetwork class:

**def** choose\_move(self, game\_state):  
 **if** self.is\_training:  
 *# Places the current game\_state into the training\_examples array.* self.training\_examples.append(FlatNeuralNetwork.TrainingExample(  
 game\_state, self))  
 *# A move is counted of having a rating in the range [1 ,-1] depending  
 # on how good it is for the current player (1 being a win).  
 # Predictions stores an array of these ratings.* predictions = []  
 best\_move\_rating = -1  
 best\_move = 0  
 possible\_moves = game\_state.get\_possible\_moves()  
 **for** move **in** possible\_moves:  
 game\_state\_copy = make\_copy(game\_state)  
 *# If making the move on the board results in a finished game,  
 # use the default move ratings* **if** game\_state\_copy.place\_counter\_and\_win\_check(move):  
 **if** game\_state\_copy.is\_drawn:  
 move\_rating = 0  
 **else**:  
 move\_rating = 1  
 *# Otherwise, the move rating is the output of the neural network.* **else**:  
 input\_tensor = self.prepare\_input\_tensor(game\_state\_copy)  
 input\_tensor = np.expand\_dims(input\_tensor, 0)  
 *# The neural network predicts how favourable the game state is  
 # for the current player, so this rating must be flipped.* move\_rating = np.squeeze(-self.model.predict(input\_tensor))  
 self.used\_computation\_tokens += 1  
 *# If the game is longer than 6 moves, the AI plays what it thinks is  
 # the best move, otherwise it will sample from the prediction array.* **if** len(game\_state.move\_list) > 6:  
 **if** move\_rating > best\_move\_rating:  
 best\_move = move  
 best\_move\_rating = move\_rating  
 **else**:  
 predictions.append(move\_rating)  
 **if** len(game\_state.move\_list) > 6:  
 chosen\_move = best\_move  
 **else**:  
 probability\_distribution = softmax(predictions)  
 chosen\_move = possible\_moves[sample(probability\_distribution)]  
 **return** chosen\_move  
  
 **def** train\_on\_batch(self, examples):  
 state\_tensors = []  
 reward\_tensors = []  
 **for** example **in** examples:  
 state\_tensors.append(example.state\_tensor)  
 reward\_tensors.append(example.reward\_tensor)  
 **return** self.model.fit(np.array(state\_tensors), np.array(reward\_tensors))  
  
  
**class** ResidualNN(FlatNeuralNetwork):  
 *"""Uses a residual neural network architecture to predict how favourable  
 a given game state is for the current player."""* @staticmethod  
 **def** build\_and\_compile\_model(residual\_blocks=5, filter\_size=3, filter\_no=75):  
 *# Uses the keras functional api to build a residual model.* input\_layer = keras.Input(shape=(6, 7, 1))  
 resnet = add\_resnet(input\_layer, residual\_blocks, filter\_size, filter\_no)  
 resnet = add\_dense\_head(resnet)  
 output\_layer = keras.layers.Dense(1)(resnet)  
 model = keras.Model(inputs=input\_layer, outputs=output\_layer)  
 model.compile(optimizer=tf.train.AdamOptimizer(),  
 loss=keras.losses.mean\_squared\_error,  
 metrics=[**'accuracy'**])  
 **return** model

ResidualNN class continued in next page.

Continuing in the ResidualNN class:

**def** prepare\_input\_tensor(self, game\_state):  
 board\_array = deepcopy(game\_state.current\_board.array)  
 *# the board has to be inverted if it is yellow's move to make it from  
 # the perspective of the current player.* **if not** game\_state.reds\_move:  
 invert\_board(board\_array)  
 input\_tensor = np.array(board\_array)  
 *# Keras' convolutional layer expects an array of values per cell.* input\_tensor = np.expand\_dims(input\_tensor, 3)  
 **return** input\_tensor  
  
  
**class** MiniMax(ConnectFourEntity):  
 *"""Uses the MiniMax algorithm to choose moves."""* MOVE\_PRIORITIES = (3, 4, 2, 5, 1, 6, 0)

This is an example of manipulating an input matrix before it is fed into Keras’ fit or predict methods

DEFAULT\_SEARCH\_DEPTH = 6

**def** \_\_init\_\_(self, name, search\_depth):  
 self.name = name

self.search\_depth = MiniMax.DEFAULT\_SEARCH\_DEPTH

*# transposition\_table is a dictionary indexed by boards  
 # that is used to prevent repeated analysis of the same board.* self.transposition\_table = {}  
  
 **def** choose\_move(self, game\_state):  
 *# The transposition\_table has to be cleared for every new move.* self.transposition\_table = {}  
 winning\_moves\_list = []  
 drawing\_moves\_list = []  
 losing\_moves\_list = []  
 winning\_score = MiniMax.get\_winning\_score(game\_state)  
 self.get\_move\_evaluations(  
 winning\_moves\_list, drawing\_moves\_list, losing\_moves\_list,  
 winning\_score, game\_state, self.search\_depth)  
 *# If there are some winning moves, randomly choose from them. Otherwise,  
 # choose the drawing move that is highest in the move priorities* **if** winning\_moves\_list:  
 chosen\_move = random.choice(winning\_moves\_list)  
 **elif** drawing\_moves\_list:  
 **for** move **in** MiniMax.MOVE\_PRIORITIES:  
 **if** move **in** drawing\_moves\_list:  
 chosen\_move = move  
 **break  
 else**:  
 chosen\_move = random.choice(losing\_moves\_list)  
 **return** chosen\_move

**def** analyse(self, game\_state, depth):  
 *"""Returns either 1, -1 or 0 to represent a win, loss or draw for the  
 current player as the evaluation of this game state."""* **if** depth == 0:  
 **return** 0  
 **else**:  
 winning\_moves\_list = []  
 drawing\_moves\_list = []  
 losing\_moves\_list = []  
 winning\_score = MiniMax.get\_winning\_score(game\_state)  
 *# Recursively calls get\_move\_evaluations.* self.get\_move\_evaluations(  
 winning\_moves\_list, drawing\_moves\_list, losing\_moves\_list,  
 winning\_score, game\_state, depth)  
 **if** winning\_moves\_list:  
 **return** winning\_score  
 **if** drawing\_moves\_list:  
 **return** 0  
 **elif** losing\_moves\_list:  
 **return** -winning\_score  
 **else**:  
 *# In the case where the board is full, returns zero for a draw.* **return** 0

This is an example of recursion as the analyse and get\_move\_evaluations methods call each other (shown on next page)

MiniMax class continued in next page

Continuing in MiniMax class:

**def** get\_move\_evaluations(self, winning\_moves\_list, drawing\_moves\_list,  
 losing\_moves\_list, winning\_score, game\_state, depth):  
 *"""Fills the move lists with the moves of the given evaluation by  
 recursively calling analyse."""* possible\_moves = game\_state.get\_possible\_moves()  
 **for** move **in** possible\_moves:  
 game\_state\_copy = make\_copy(game\_state)  
 **if** game\_state\_copy.place\_counter\_and\_win\_check(move):  
 **if** game\_state\_copy.is\_drawn:   
 drawing\_moves\_list.append(move)  
 **else**:  
 winning\_moves\_list.append(move)  
 **else**:  
 *# If making the move does not result in a terminal game state,  
 # first check if this position has aready been analysed.* hashable\_board = get\_board\_tuple(game\_state\_copy)  
 **if** hashable\_board **in** self.transposition\_table:  
 move\_analysis = self.transposition\_table[hashable\_board]  
 **else**:  
 *# If the position is not in the transposition table,  
 # recursively call analyse.* move\_analysis = self.analyse(game\_state\_copy, depth - 1)  
 self.transposition\_table[hashable\_board] = move\_analysis  
 **if** move\_analysis == winning\_score:  
 winning\_moves\_list.append(move)  
 **elif** move\_analysis == 0:  
 drawing\_moves\_list.append(move)  
 **else**:  
 losing\_moves\_list.append(move)  
  
 @staticmethod  
 **def** get\_winning\_score(game\_state):  
 **if** game\_state.reds\_move:  
 **return** 1  
 **else**:  
 **return** -1  
**class** AZtypeAI(TrainableAI):  
 *"""An abstract class for AI which use types of the AlphaZero algorithm."""* **class** TrainingExample(FlatNeuralNetwork.TrainingExample):  
 *"""Used to store data used to train AlphaZero's neural network."""* **def** \_\_init\_\_(self, game\_state, improved\_policy, parent\_AI):  
 super().\_\_init\_\_(game\_state, parent\_AI)  
 self.policy\_tensor = np.array(improved\_policy)  
  
 **def** \_\_init\_\_(self, name, is\_training):  
 super().\_\_init\_\_(name, is\_training)  
 self.root\_node\_initialised = **False  
  
 def** choose\_move(self, game\_state):  
 self.setup\_for\_simulations(game\_state)  
 self.set\_root\_node(game\_state)  
 *# Plays simulated games until it is decided enough have been played.* **while** self.more\_simulations\_needed():  
 self.root\_node.expand(self)  
 self.used\_computation\_tokens += 1  
 *# Sets the improved policy to be the normalised counts  
 # of the possible moves and 0 otherwise* improved\_policy = [0 **for** \_ **in** range(7)]  
 **for** child\_node **in** self.root\_node.child\_nodes:  
 improved\_policy[child\_node.preceding\_action] = \  
 child\_node.visit\_count/(self.root\_node.visit\_count-1)  
 **if** self.is\_training:  
 self.training\_examples.append(AZtypeAI.TrainingExample(  
 make\_copy(game\_state), improved\_policy, self))  
 *# Chooses the child node either deterministically (if the game is longer  
 # than 4 moves) or probabilistically using the improved policy.* chosen\_child\_node = self.get\_chosen\_child\_node(  
 len(game\_state.move\_list), improved\_policy)  
 self.root\_node = chosen\_child\_node  
 **return** chosen\_child\_node.preceding\_action

This is an example of where I have used a dictionary to prevent my MiniMax algorithm from Calculating the same position multiple times.

This is an example of recursion as the analyse and get\_move\_evaluations methods call each other (shown on previous page)

AZtypeAI class continued in next page.

Continuing in the AZtypeAI class:  
  
 **def** more\_simulations\_needed(self):  
 *"""Abstract method returns False when computation should be cut off."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** setup\_for\_simulations(self, game\_state):  
 *"""Abstract method called at the start of the choose move method."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** get\_chosen\_child\_node(self, move\_list\_length, improved\_policy):  
 *"""Chooses the child node either deterministically (if the game is longer  
 than 4 moves) or probabilistically using the improved policy."""* **if** move\_list\_length < 5:  
 chosen\_action = sample(improved\_policy)  
 **for** child\_node **in** self.root\_node.child\_nodes:  
 **if** child\_node.preceding\_action == chosen\_action:  
 **return** child\_node  
 **else**:  
 chosen\_child\_node\_list = []  
 max\_visit\_count = 0  
 *# Keeps a list if there are multiple child nodes with the same  
 # highest visit count and chooses randomly in the event of a tie.* **for** child\_node **in** self.root\_node.child\_nodes:  
 **if** child\_node.visit\_count > max\_visit\_count:  
 max\_visit\_count = child\_node.visit\_count  
 chosen\_child\_node\_list = [child\_node]  
 **elif** child\_node.visit\_count == max\_visit\_count:  
 chosen\_child\_node\_list.append(child\_node)  
 **return** random.choice(chosen\_child\_node\_list)  
  
 **def** set\_root\_node(self, game\_state):  
 *# If the root node is initialised, checks all the child nodes to see  
 # if the game state they represent is the same as the passed game state.* **if** self.root\_node\_initialised:  
 **for** child\_node **in** self.root\_node.child\_nodes:  
 **if** child\_node.game\_state.current\_board.array ==\  
 game\_state.current\_board.array:  
 self.root\_node = child\_node  
 **break  
 else**:  
 *# If no such child node is found, a new root node is made.* self.root\_node\_initialised = **False** *# Makes a new root node using a copy of the game state.* **if not** self.root\_node\_initialised:  
 game\_copy = make\_copy(game\_state)  
 self.root\_node = UCTsearchNode(**None**, game\_copy, **None**)  
 self.root\_node.expand(self)  
 self.root\_node\_initialised = **True  
  
 def** train\_on\_batch(self, examples):  
 *"""Trains the neural network on a list of training examples."""* state\_tensors = []  
 reward\_tensors = []  
 policy\_tensors = []  
 **for** example **in** examples:  
 state\_tensors.append(example.state\_tensor)  
 reward\_tensors.append(example.reward\_tensor)  
 policy\_tensors.append(example.policy\_tensor)  
 state\_tensors = np.array(state\_tensors)  
 reward\_tensors = np.array(reward\_tensors)  
 policy\_tensors = np.array(policy\_tensors)  
 **return** self.model.fit(state\_tensors, {**"policy\_head"**: policy\_tensors,  
 **"value\_head"**: reward\_tensors})

AZtypeAI class continued in next page.

Continuing in the AZtypeAI class:

@staticmethod  
 **def** build\_AZ\_network(input\_shape, residual\_blocks, filter\_size, filter\_no):  
 *"""Uses the Keras functional api to make an alphazero-type  
 residual neural network model with a specified input shape."""* input\_layer = keras.Input(shape=input\_shape)  
 resnet = add\_resnet(input\_layer, residual\_blocks, filter\_size, filter\_no)  
 policy\_head = add\_dense\_head(resnet)  
 policy\_head = keras.layers.Dense(7, name=**"policy\_head"**,  
 activation=**"sigmoid"**)(policy\_head)  
 value\_head = add\_dense\_head(resnet)  
 value\_head = keras.layers.Dense(1, name=**"value\_head"**,  
 activation=**"tanh"**)(value\_head)  
 model = keras.Model(inputs=input\_layer,  
 outputs=[policy\_head, value\_head])  
 model.compile(optimizer=tf.train.AdamOptimizer(),  
 loss={**"policy\_head"**: keras.losses.binary\_crossentropy,  
 **"value\_head"**: keras.losses.mean\_squared\_error},  
 metrics=[**'accuracy'**])  
 **return** model  
  
  
**class** AZclone(AZtypeAI):  
 *""" An implementation of the AlphaZero algorithm for connect4."""* **def** more\_simulations\_needed(self, simulations\_per\_move=100):  
 *# Returns false when 100 simulations have been used for this move.* **if** self.simulation\_count == simulations\_per\_move:  
 **return False  
 else**:  
 self.simulation\_count += 1  
 **return True  
  
 def** setup\_for\_simulations(self, game\_state):  
 self.simulation\_count = 0  
  
 **def** \_\_init\_\_(self, name, is\_training):  
 super().\_\_init\_\_(name, is\_training)  
 self.root\_node\_initialised = **False** @staticmethod  
 **def** build\_and\_compile\_model(residual\_blocks=5, filter\_size=3, filter\_no=75):  
 **return** AZtypeAI.build\_AZ\_network((6, 7, 1),  
 residual\_blocks, filter\_size, filter\_no)  
  
 **def** prepare\_input\_tensor(self, game\_state):  
 **return** ResidualNN.prepare\_input\_tensor(self, game\_state)  
  
  
**class** TimedUCT(AZtypeAI):  
 DEFAULT\_TIMER\_CONSTANT = 3  
  
 **def** \_\_init\_\_(self, name, is\_training):  
 super().\_\_init\_\_(name, is\_training)  
 self.timer\_constant = TimedUCT.DEFAULT\_TIMER\_CONSTANT  
  
 **def** setup\_for\_simulations(self, game\_state):  
 *# collects data used when preparing the input tensor and  
 # deciding whether more simulations are needed.* self.simulation\_count = 0  
 self.initially\_reds\_move = game\_state.reds\_move  
 self.remaining\_move\_count = 42 - len(game\_state.move\_list)  
 self.normalised\_clocks = game\_state.get\_normalised\_clocks()  
 self.current\_player\_clock\_tensor = \  
 np.array([[[self.normalised\_clocks[0]]  
 **for** \_ **in** range(7)] **for** \_ **in** range(6)])  
 self.enemy\_player\_clock\_tensor = \  
 np.array([[[self.normalised\_clocks[1]]  
 **for** \_ **in** range(7)] **for** \_ **in** range(6)])

TimedUCT class continued in next page.

Continuing in the TimedUCT class:

**def** more\_simulations\_needed(self):  
 *# Atleast 5 simulations are done to prevent anomolies.* self.simulation\_count += 1  
 **if** self.simulation\_count < 5:  
 **return True** *# calculates the lower confidence bound of the "best" node* most\_negative\_UCT\_node = self.root\_node.get\_most\_negative\_UCT\_node()  
 lower\_confidence\_bound = \  
 most\_negative\_UCT\_node.average\_value() \  
 + most\_negative\_UCT\_node.confidence\_interval(  
 self.root\_node.visit\_count)  
 *# Calculates how much this bound overlaps with the UCT values  
 # of the other child nodes.* total\_confidence\_overlap = 0  
 **for** child\_node **in** self.root\_node.child\_nodes:  
 **if** child\_node **is not** most\_negative\_UCT\_node:  
 total\_confidence\_overlap += self.confidence\_overlap(  
 child\_node, lower\_confidence\_bound)  
 *# calculates timer\_value, a score of how much  
 # could be gained from doing more simulations* timer\_value = (total\_confidence\_overlap \* self.normalised\_clocks[0]\*42)\  
 / (self.remaining\_move\_count)  
 *# compares this value with a learned parameter  
 # to decide whether to cut of calculation  
 # The simulation count is limited to mitigate against anomolies.* **if** self.simulation\_count > 500:  
 **return False  
 elif** timer\_value < self.timer\_constant:  
 **return False  
 else**:  
 **return True  
  
 def** confidence\_overlap(self, child\_node, lower\_confidence\_bound):  
 *"""Returns the difference between this node's UCT value  
 and the lower confidence bound."""* child\_UCT = child\_node.average\_value() - child\_node.confidence\_interval(  
 self.root\_node.visit\_count)  
 confidence\_overlap = lower\_confidence\_bound - child\_UCT  
 **if** confidence\_overlap > 0:  
 **return** confidence\_overlap  
 **else**:  
 **return** 0  
  
 **def** punish\_for\_timeouts(self, timeout\_count, game\_count):  
 *# If the AI timed out in over 5 percent of the games, the timer  
 # constant is increased to make it use less time per move.* **if** timeout\_count / game\_count > 0.05:  
 *# We multiply rather than add to the timer constant so that  
 # it will converge on an appropriate value quicker.* self.timer\_constant = self.timer\_constant\*1.01  
 **else**:  
 self.timer\_constant = self.timer\_constant\*0.99  
 print(**"timer\_constant: "**, self.timer\_constant)  
  
 @staticmethod  
 **def** build\_and\_compile\_model(residual\_blocks=2, filter\_size=3, filter\_no=75):  
 **return** AZtypeAI.build\_AZ\_network((6, 7, 3),  
 residual\_blocks, filter\_size, filter\_no)  
  
 **def** prepare\_input\_tensor(self, game\_state):  
 board\_tensor = ResidualNN.prepare\_input\_tensor(self, game\_state)  
 **if** game\_state.reds\_move == self.initially\_reds\_move:  
 **return** np.concatenate((board\_tensor,  
 self.current\_player\_clock\_tensor,  
 self.enemy\_player\_clock\_tensor), 2)  
 **else**:  
 **return** np.concatenate((board\_tensor,  
 self.enemy\_player\_clock\_tensor,  
 self.current\_player\_clock\_tensor), 2)

TimedUCT class continued in next page.

Continuing in the TimedUCT class:

**def** load\_parameters(self, filepath):  
 *"""Overrides TrainableAI's load\_parameters method to  
 load the timer constant as well"""* self.model.load\_weights(filepath + **".h5"**)  
 timer\_constant\_file = open(filepath + **".p"**, **"rb"**)  
 self.timer\_constant = pickle.load(timer\_constant\_file)  
  
 **def** save\_parameters(self, filepath):  
 *"""Overrides TrainableAI's save\_parameters method to  
 save the timer constant as well"""* self.model.save\_weights(filepath + **".h5"**)  
 timer\_constant\_file = open(filepath + **".p"**, **"wb"**)  
 pickle.dump(self.timer\_constant, timer\_constant\_file)  
  
  
  
**class** UCTsearchNode:  
 *"""Used by the AZclone and TimedUCT classes, each node represents a game  
 state with child nodes that are the result of different moves from this state."""* C\_PUCT = 4  
  
 **def** \_\_init\_\_(self, preceding\_action, game\_state, node\_policy):  
 *"""Initialises this node as a leaf node"""* self.preceding\_action = preceding\_action  
 self.game\_state = game\_state  
 self.is\_leaf\_node = **True** self.visit\_count = 0  
 self.total\_value = 0  
 self.node\_policy = node\_policy  
 self.child\_nodes = []  
  
 **def** expand(self, AI):  
 *"""Returns the change in this node's value resulting from the simulation."""* **if** self.is\_leaf\_node:  
 *# get prediction from neural network* input\_tensor = AI.prepare\_input\_tensor(self.game\_state)  
 input\_tensor = np.expand\_dims(input\_tensor, 0)  
 model\_output = AI.model.predict(input\_tensor)  
 policy\_head = np.squeeze(model\_output[0])  
 value\_head = np.squeeze(model\_output[1])  
 *# use prediction to estimate own value and likely moves* self.initialise\_child\_nodes(policy\_head)  
 value\_change = value\_head  
 self.is\_leaf\_node = **False  
 else**:  
 *# recursively expand the child node with the most negative  
 # upper confidence value, which corresponds to the  
 # game state that is worst for the other player* most\_negative\_UCT\_node = self.get\_most\_negative\_UCT\_node()  
 value\_change = most\_negative\_UCT\_node.expand(AI)  
 self.total\_value += value\_change  
 self.visit\_count += 1  
 *# the parent node is from the perspective of the other player,  
 # hence the value change returned is negative* **return** -value\_change  
  
 **def** get\_most\_negative\_UCT\_node(self):  
 *"""Returns the child node with the highest upper confidence (UCT) value"""* most\_negative\_UCT = 1  
 most\_negative\_UCT\_node = **None  
 for** child\_node **in** self.child\_nodes:  
 child\_UCT = child\_node.average\_value() \  
 - child\_node.confidence\_interval(self.visit\_count)  
 **if** child\_UCT < most\_negative\_UCT:  
 most\_negative\_UCT\_node = child\_node  
 most\_negative\_UCT = child\_UCT  
 **return** most\_negative\_UCT\_node

This shows a UCT search node object having multiple UCT search node children to create a tree data structure

This is an example of the expand function being called recursively to play a simulated game and generate a decision tree

UCTsearchNode continued in next page

UCTsearchNode class continued:

**def** initialise\_child\_nodes(self, policy\_head):  
 *# get possible moves and normalise the policy head  
 # to only take into account allowed moves* policy\_sum = 0  
 possible\_moves = []  
 **for** i, policy **in** enumerate(policy\_head):  
 **if not** self.game\_state.current\_board.column\_is\_full(i):  
 policy\_sum += policy  
 possible\_moves.append(i)  
 *# initialise child nodes corresponding to game states  
 # resulting from each possible move* **for** move **in** possible\_moves:  
 game\_state\_copy = make\_copy(self.game\_state)  
 adjusted\_policy = policy\_head[move] / policy\_sum  
 **if** game\_state\_copy.place\_counter\_and\_win\_check(move):  
 **if** game\_state\_copy.is\_drawn:  
 node\_value = 0  
 **else**:  
 node\_value = -1  
 self.child\_nodes.append(TerminalNode(  
 node\_value, move, game\_state\_copy, adjusted\_policy))  
 **else**:  
 self.child\_nodes.append(UCTsearchNode(  
 move, game\_state\_copy, adjusted\_policy))   
  
 **def** confidence\_interval(self, parent\_n):  
 **return** (UCTsearchNode.C\_PUCT \* self.node\_policy \*  
 math.sqrt(parent\_n)) / (self.visit\_count + 1)  
  
 **def** average\_value(self):  
 **if** self.is\_leaf\_node:  
 **return** 0  
 **else**:  
 **return** self.total\_value / self.visit\_count  
  
  
**class** TerminalNode(UCTsearchNode):  
 *"""Class for nodes that represent a won or drawn position, a terminal node's  
 value is -1 for a loss for the current player and 0 for a draw."""* **def** \_\_init\_\_(self, value, preceding\_action, game\_state, node\_policy):  
 super().\_\_init\_\_(preceding\_action, game\_state, node\_policy)  
 self.value = value  
  
 **def** average\_value(self):  
 **return** self.value  
  
 **def** confidence\_interval(self, parent\_n):  
 *"""We can be one hundred percent sure of this node's value;  
 therefore the confidence interval on it's value is zero."""* **return** 0  
  
 **def** expand(self, AI):  
 self.visit\_count += 1  
 **return** -self.value  
  
  
**def** add\_dense\_head(y):  
 *"""Adds a set of dense layers using the Keras functional api."""* y = keras.layers.Flatten()(y)  
 y = keras.layers.Dense(20)(y)  
 y = keras.layers.LeakyReLU()(y)  
 y = keras.layers.Dense(20)(y)  
 y = keras.layers.LeakyReLU()(y)  
 **return** y

This is an example of where I have used mathematical techniques to computed a confidence interval on a valuation of a game state

connect4AI.py file continued in next page

connect4AI.py file continued  
  
**def** add\_resnet(y, residual\_blocks, filter\_size, filter\_no):  
 *"""Uses the Keras functional api to add a convolutional residual model."""* y = add\_convolutional\_layer(y, filter\_size, filter\_no)  
 y = add\_convolutional\_layer(y, filter\_size, filter\_no)  
 **for** i **in** range(residual\_blocks):  
 y = add\_residual\_block(y, filter\_size, filter\_no)  
 **return** y  
  
  
**def** add\_residual\_block(block\_input, filter\_size, filter\_no):  
 *"""Uses the Keras functional api to add a convolutional residual block."""* y = add\_convolutional\_layer(block\_input, filter\_size, filter\_no)  
 y = add\_convolutional\_layer(y, filter\_size, filter\_no)  
 **return** keras.layers.add([block\_input, y])  
  
  
**def** add\_convolutional\_layer(y, filter\_size, filter\_no):  
 *# the padding is "same" to keep the input and output a constant shape* y = keras.layers.Conv2D(filter\_no, filter\_size, padding=**"same"**)(y)  
 y = keras.layers.BatchNormalization()(y)  
 **return** keras.layers.LeakyReLU()(y)  
  
  
**def** invert\_board(board):  
 *"""Returns a list with the counter codes switched."""* **for** i **in** range(6):  
 **for** j **in** range(7):  
 board[i][j] = -board[i][j]  
 **return** board  
  
  
**def** softmax(prediction):  
 *"""Turns an array of expected rewards into a probability distribution."""* prediction = np.exp(prediction)  
 probability\_sum = sum(prediction)  
 **for** i, probability **in** enumerate(prediction):  
 prediction[i] = probability / probability\_sum  
 **return** prediction  
  
  
**def** sample(probability\_distribution):  
 *"""Returns the index of the randomly selected probability."""* random\_probability = random.random()  
 **for** i, probability **in** enumerate(probability\_distribution):  
 random\_probability -= probability  
 **if** random\_probability < 0:  
 **return** i  
 **return** len(probability\_distribution) - 1  
  
  
**def** make\_copy(game\_state):  
 dummy = Dummy()  
 board\_copy = deepcopy(game\_state.current\_board)  
 move\_list\_copy = deepcopy(game\_state.move\_list)  
 **return** game.GameState(dummy, dummy,  
 game\_state.reds\_move, board\_copy, move\_list\_copy)  
  
  
**def** get\_board\_tuple(game\_state):  
 **return** tuple(tuple(row) **for** row **in** game\_state.current\_board.array)

This is an example of a use of mathematical techniques to compute SoftMax on a vector.

End of File.

##### Connect4Train.py

**import** random  
  
**import** numpy **as** np  
**from** matplotlib **import** pyplot **as** plt  
  
**import** connect4game **as** game  
**import** connect4user\_interface  
  
INITIAL\_COMPUTATION\_TOKENS = 2000  
EXAMPLE\_CONTAINER\_SIZE = 10000  
BATCH\_SIZE = 1024  
  
  
**class** FullContainerException(Exception):  
 *"""Raised when an attempt is made to place an example into the container  
 when it is already full."""* **pass  
  
  
class** TrainingExampleContainer:  
 **def** \_\_init\_\_(self, container\_size=EXAMPLE\_CONTAINER\_SIZE):  
 self.container\_size = container\_size  
 self.empty\_index\_stack = [i **for** i **in** range(self.container\_size)]  
 self.empty\_index\_set = set(self.empty\_index\_stack)  
 self.training\_examples = [0 **for** \_ **in** range(self.container\_size)]  
  
 **def** place(self, training\_example):  
 **if** self.is\_full():  
 **raise** FullContainerException  
 **else**:  
 empty\_index = self.empty\_index\_stack.pop()  
 self.empty\_index\_set.remove(empty\_index)  
 self.training\_examples[empty\_index] = training\_example  
  
 **def** retrieve(self):  
 *"""Returns a randomly selected training example."""* **while True**:  
 random\_index = random.randint(0, self.container\_size-1)  
 **if** random\_index **not in** self.empty\_index\_set:  
 self.empty\_index\_stack.append(random\_index)  
 self.empty\_index\_set.add(random\_index)  
 **return** self.training\_examples[random\_index]  
  
 **def** retrieve\_batch(self, batch\_size=BATCH\_SIZE):  
 *"""Returns a randomly selected batch of training examples."""* batch = []  
 **for** \_ **in** range(batch\_size):  
 batch.append(self.retrieve())  
 **return** batch  
  
 **def** is\_full(self):  
 **if** self.empty\_index\_stack:  
 **return False  
 else**:  
 **return True  
  
 def** get\_percent\_full(self):  
 proportion = (EXAMPLE\_CONTAINER\_SIZE -  
 len(self.empty\_index\_stack))/EXAMPLE\_CONTAINER\_SIZE  
 **return** proportion\*100

This is an example of where I have used a hash table to keep track of holes in the container so that checking if a randomly selected training example is already a hole is O(1).

This is an example of where I have used a stack to keep track of the holes in the container so that finding a hole’s index is O(1)

connect4train.py file continued in next page.

connect4train.py file continued:

**class** TrainingClock:  
 *"""Keeps track of how many 'computation tokens' each player has remaining  
 To ensure training games are fair."""* **def** \_\_init\_\_(self):  
 self.current\_player\_tokens = INITIAL\_COMPUTATION\_TOKENS  
 self.enemy\_tokens = INITIAL\_COMPUTATION\_TOKENS  
  
 **def** change\_player(self, current\_player\_used\_tokens):  
 *"""Returns true if the current player runs out of tokens."""* self.enemy\_tokens = self.current\_player\_tokens  
 self.current\_player\_tokens = \  
 INITIAL\_COMPUTATION\_TOKENS - current\_player\_used\_tokens  
 **if** self.current\_player\_tokens < 0:  
 **return True  
 else**:  
 **return False  
  
  
class** TrainingGameState(game.GameState):  
 *"""Type of GameState that does not have a User Interface for training."""* **def** \_\_init\_\_(self, red, yellow, reds\_move=**True**, board=**None**,  
 starting\_move\_list=**None**):  
 **if** board **is None**:  
 board = game.Board()  
 **if** starting\_move\_list **is None**:  
 starting\_move\_list = []  
 super().\_\_init\_\_(red, yellow, reds\_move, board, starting\_move\_list)  
 self.clock = TrainingClock()  
  
 **def** start\_game\_procedure(self):  
 self.red.used\_computation\_tokens = 0  
 self.yellow.used\_computation\_tokens = 0  
 self.clock = TrainingClock()  
  
 **def** won\_game\_procedure(self):  
 **pass  
  
 def** next\_move\_procedure(self):  
 **pass  
  
 def** change\_clock\_and\_win\_check(self):  
 *"""Returns true if a player runs out of computation tokens."""* **if** self.clock.change\_player(  
 self.get\_current\_player().used\_computation\_tokens):  
 self.red\_winner = self.reds\_move  
 self.is\_drawn = **False** self.timeout = **True  
 return True  
 else**:  
 **return False  
  
 def** get\_normalised\_clocks(self):  
 *"""Used by the TimedUCT AI to return tuple containing  
 how much time each player has left on a normalised scale."""* normalised\_current\_player\_tokens = \  
 self.clock.current\_player\_tokens/INITIAL\_COMPUTATION\_TOKENS  
 normalised\_enemy\_player\_tokens = \  
 self.clock.current\_player\_tokens/INITIAL\_COMPUTATION\_TOKENS  
 **return** normalised\_current\_player\_tokens, normalised\_enemy\_player\_tokens  
  
  
**class** TrainingEnvironment:  
 *"""Used to train AI."""* **def** \_\_init\_\_(self, AI, epoch\_count=0):  
 self.AI = AI  
 self.epoch\_count = epoch\_count  
 self.training\_example\_container = TrainingExampleContainer()  
 *# loss\_history stores average loss from each batch of training.* self.loss\_history = []

TrainingEnvironment class continued in next page.

TrainingEnvironment class continued:

**def** execute\_episode(self, red, yellow):  
 *"""Executes a game of self play and places examples in the container."""* new\_game = TrainingGameState(red, yellow)  
 new\_game.play()  
 **if** new\_game.timeout:  
 self.timeout\_counter += 1  
 **if** new\_game.is\_drawn:  
 red\_reward = 0  
 **elif** new\_game.red\_winner:  
 red\_reward = 1  
 **else**:  
 red\_reward = -1  
 self.assign\_rewards\_and\_place(red.get\_training\_examples(), red\_reward)  
 self.assign\_rewards\_and\_place(yellow.get\_training\_examples(), -red\_reward)  
  
 **def** assign\_rewards\_and\_place(self, training\_examples, reward):  
 *# If the container is already full, no more examples are placed into it.* **for** training\_example **in** training\_examples:  
 training\_example.set\_reward(reward)  
 **try**:  
 self.training\_example\_container.place(training\_example)  
 **except** FullContainerException:  
 **break  
  
 def** execute\_epoch(self):  
 *"""Saves the current AI, executes episodes until the container is full,  
 and then trains the AI using a batch retrieved from the container."""* self.timeout\_counter = 0  
 self.episode\_counter = 0  
 print(**"epoch: "**, self.epoch\_count)  
 filepath = self.AI.name+**"\_"**+str(self.epoch\_count)+**"\_epochs"** self.epoch\_count += 1  
 *# saves the current AI* self.AI.save\_parameters(filepath)  
 self.fill\_container(filepath)  
 self.AI.punish\_for\_timeouts(self.timeout\_counter, self.episode\_counter)  
 metric\_history = self.AI.train\_on\_batch(  
 self.training\_example\_container.retrieve\_batch())  
 self.loss\_history.append(metric\_history.history[**"loss"**])  
  
 **def** fill\_container(self, filepath):  
 *"""Plays games until the container is full."""* red, yellow = self.create\_copies(filepath)  
 i = 0  
 print(**"filling container"**)  
 **while not** self.training\_example\_container.is\_full():  
 *# Prints how full the container is every fifth game.* **if** i % 5 == 0:  
 print(str(round(  
 self.training\_example\_container.get\_percent\_full(), 1)),  
 **"percent full"**)  
 i += 1  
 self.episode\_counter += 1  
 self.execute\_episode(red, yellow)  
 print(**"games played: "**, i)  
  
 **def** create\_copies(self, filepath):  
 *"""Returns a tuple of copies of the current AI being trained."""* red = type(self.AI)(self.AI.name, **True**)  
 yellow = type(self.AI)(self.AI.name, **True**)  
 red.load\_parameters(filepath)  
 yellow.load\_parameters(filepath)  
 **return** red, yellow

connect4train.py file continued in next page.

connect4train.py file continued:  
  
**def** evaluate(red, yellow, num\_games\_per\_colour=50):  
 *"""Returns the percentage of non-drawn  
 games won by the AI that is currently red"""* draw\_count = 0  
 player\_one\_win\_count = 0  
 timeout\_count = 0  
 *# Plays 50 games and logs the results.* **for** \_ **in** range(num\_games\_per\_colour):  
 new\_game = TrainingGameState(red, yellow)  
 new\_game.play()  
 **if** new\_game.is\_drawn:  
 draw\_count += 1  
 **elif** new\_game.red\_winner:  
 player\_one\_win\_count += 1  
 **if** new\_game.timeout:  
 timeout\_count += 1  
 *# Switches the sides and then logs the results again.* red, yellow = yellow, red  
 **for** \_ **in** range(num\_games\_per\_colour):  
 new\_game = TrainingGameState(red, yellow)  
 new\_game.play()  
 **if** new\_game.is\_drawn:  
 draw\_count += 1  
 **elif not** new\_game.red\_winner:  
 player\_one\_win\_count += 1  
 **if** new\_game.timeout:  
 timeout\_count += 1  
 print(**"red win count: "**, player\_one\_win\_count)  
 print(**"draw count: "**, draw\_count)  
 print(**"timeout count: "**, timeout\_count)  
 **try**:  
 red\_win\_percentage = \  
 100\*player\_one\_win\_count/(2\*num\_games\_per\_colour-draw\_count)  
 **except** ZeroDivisionError:  
 red\_win\_percentage = 50  
 **return** red\_win\_percentage

##### Connect4User\_interface.py

**import** time  
**import** os  
**import** math  
**import** tkinter **as** tk  
**from** copy **import** deepcopy  
  
**import** connect4AI **as** AI  
**import** connect4game **as** game  
  
DEFAULT\_PLAYER\_SECONDS = 300  
HUMAN\_PLAYER\_STRING = **"human player"**RESNET\_STRING = **"flat residual network"**MINIMAX\_STRING = **"MiniMax"**AZ\_CLONE\_STRING = **"Alpha Zero clone"**TIMED\_UCT\_STRING = **"Timed UCT"**AI\_TYPE\_STRINGS = (HUMAN\_PLAYER\_STRING, MINIMAX\_STRING, RESNET\_STRING,  
 AZ\_CLONE\_STRING, TIMED\_UCT\_STRING)  
  
  
**def** AI\_type\_dict(string):  
 **return**{  
 HUMAN\_PLAYER\_STRING: PlayingGUI.GUIHumanPlayer,  
 MINIMAX\_STRING: AI.MiniMax,  
 RESNET\_STRING: AI.ResidualNN,  
 AZ\_CLONE\_STRING: AI.AZclone,  
 TIMED\_UCT\_STRING: AI.TimedUCT  
 }[string]

connect4user\_interface.py file continued in next page.

connect4user\_interface.py file continued:

**def** AI\_default\_time\_dict(AI\_type):  
 **return**{  
 PlayingGUI.GUIHumanPlayer: 300,  
 AI.MiniMax: 20,  
 AI.ResidualNN: 10,  
 AI.AZclone: 20,  
 AI.TimedUCT: 20  
 }[AI\_type]  
  
  
**class** PlayingClock:  
 *"""Stores how much time each player has left."""* **def** \_\_init\_\_(self, player\_seconds):  
 *""" Player seconds is a tuple saying how much time each player has."""* self.current\_player\_seconds = player\_seconds[0]  
 self.enemy\_player\_seconds = player\_seconds[1]  
 *# the start method needs to be applied before the clock can be used* self.start\_time = **None  
  
 def** start(self):  
 *"""Sets the clock's start time to the current time"""* self.start\_time = time.time()  
  
 **def** change\_player(self):  
 *"""Takes away the correct amount of time from the current player  
 and switches the current and other player."""* new\_start\_time = time.time()  
 self.current\_player\_seconds = self.current\_player\_seconds\  
 + self.start\_time - new\_start\_time  
 self.start\_time = new\_start\_time  
 self.enemy\_player\_seconds, self.current\_player\_seconds = \  
 self.current\_player\_seconds, self.enemy\_player\_seconds  
  
  
**class** PlayingGameState(game.GameState):  
 *"""Game state used when playing games with a user interface."""* **def** \_\_init\_\_(self, red, yellow, parent\_UI, reds\_move=**True**, board=**None**,  
 starting\_move\_list=**None**, player\_seconds=**None**):  
 *# the default values for board and starting move list are for a new game* **if** board **is None**:  
 board = game.Board()  
 **if** starting\_move\_list **is None**:  
 starting\_move\_list = []  
 super().\_\_init\_\_(red, yellow, reds\_move, board, starting\_move\_list)  
 self.parent\_UI = parent\_UI  
 *# if no value is given for player seconds, the game is not timed* **if** player\_seconds **is None**:  
 self.is\_timed = **False  
 else**:  
 self.is\_timed = **True** self.clock = PlayingClock(player\_seconds)  
  
 **def** start\_game\_procedure(self):  
 *"""Overridden from game module."""* **if** self.is\_timed:  
 self.clock.start()  
 self.parent\_UI.display\_state(self)  
  
 **def** won\_game\_procedure(self):  
 *"""Overridden from game module."""* self.parent\_UI.display\_winner(self)  
  
 **def** next\_move\_procedure(self):  
 *"""Overridden from game module."""* self.parent\_UI.display\_state(self)

PlayingGameState class continued in next page.

PlayingGameState class continued:  
 **def** change\_clock\_and\_win\_check(self):  
 *"""Returns True if the current player runs out of time."""* **if** self.is\_timed:  
 self.clock.change\_player()  
 **if** self.clock.enemy\_player\_seconds < 0:  
 self.red\_winner = self.reds\_move  
 self.is\_drawn = **False** self.timeout = **True  
 return True  
 else**:  
 **return False  
 else**:  
 **return False  
  
 def** get\_normalised\_clocks(self):  
 *"""Returns a tuple containing how much time each player has divided by  
 their default amount."""* current\_player\_type = type(self.get\_current\_player())  
 default\_current\_player\_seconds = AI\_default\_time\_dict(current\_player\_type)  
 enemy\_player\_type = type(self.get\_enemy\_player())  
 default\_enemy\_player\_seconds = AI\_default\_time\_dict(enemy\_player\_type)  
 normalised\_current\_player\_clock = \  
 self.clock.current\_player\_seconds / default\_current\_player\_seconds  
 normalised\_enemy\_player\_clock = \  
 self.clock.enemy\_player\_seconds / default\_enemy\_player\_seconds  
 print(**"normalised clocks: "**, normalised\_current\_player\_clock, normalised\_enemy\_player\_clock)  
 **return** normalised\_current\_player\_clock, normalised\_enemy\_player\_clock  
  
  
**class** ConnectFourUserInterface:  
 *"""abstract interface for different types of UI"""* **def** display\_state(self, game\_state):  
 *"""Abstract method that is called after a move is made."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** display\_winner(self, game\_state):  
 **raise** NotImplementedError(**"Please Implement this method"**)  
  
  
**class** CommandLineUI(ConnectFourUserInterface):  
 *"""abstract interface for different types of UI"""* **class** CommandLineHumanPlayer(AI.ConnectFourEntity):  
 *"""For a human player to play."""* **def** choose\_move(self, game\_state):  
 *"""prompts the player to enter an integer between 1 and 7"""* invalid\_input = **True  
 while** invalid\_input:  
 player\_input = input(**"Select a column"**)  
 **if** player\_input.isdigit():  
 player\_input = int(player\_input)  
 **if** player\_input **in** range(1, 8):  
 **if** game\_state.current\_board.column\_is\_full(player\_input - 1):  
 print(**"that column is full"**)  
 **else**:  
 invalid\_input = **False  
 else**:  
 print(**"please enter a number between 1 and 7"**)  
 **else**:  
 print(**"the imput need to be an integer"**)  
 **return** player\_input-1  
  
 **def** display\_state(self, game\_state):  
 *"""Abstract method that is called after a move is made."""* current\_player\_name = game\_state.get\_current\_player().name  
 enemy\_player\_name = game\_state.get\_enemy\_player().name  
 *#at this time, the current and enemy players have been switched on the clock* print(**"{0}'s move, \n"  
 "{0} has {1} seconds left and {2} has {3} seconds left"** .format(current\_player\_name, game\_state.clock.current\_player\_seconds,  
 enemy\_player\_name, game\_state.clock.enemy\_player\_seconds))  
 game\_state.current\_board.print()

CommandLineUI class continued in next page.

CommandLineUI class continued:  
  
 **def** display\_winner(self, game\_state):  
 **if** game\_state.is\_drawn:  
 print(**"draw!"**)  
 **else**:  
 **if** game\_state.timeout:  
 print(**"timeout!"**)  
 **if** game\_state.red\_winner:  
 print(game\_state.red.name, **" wins!"**)  
 **else**:  
 print(game\_state.yellow.name, **" wins!"**)  
 game\_state.current\_board.print()  
  
  
**class** GUIwidget:  
 *"""Interface used so all the widgets can be updated in a loop."""* **def** display\_state(self, game\_state):  
 *"""Abstract method which displays information about the current game."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** transition\_to\_playing(self):  
 *"""Abstract method which changes the widget to the 'playing' state."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** transition\_to\_paused(self):  
 *"""Abstract method which changes the widget to the 'paused' state."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
 **def** display\_winner(self, game\_state):  
 *"""Abstract method which changes the widget to the 'won game' state."""* **raise** NotImplementedError(**"Please Implement this method"**)  
  
  
**class** NonResizableButton(tk.Frame):  
 *"""Used by MoveButton and PausePlayReset button to create a button which  
 is not automatically re-sized by tkinter."""* **def** \_\_init\_\_(self, master, width, height, command, text=**""**):  
 tk.Frame.\_\_init\_\_(self, master, width=width, height=height, bg=**"white"**)  
 self.grid\_propagate(**False**)  
 self.rowconfigure(0, weight=1)  
 self.columnconfigure(0, weight=1)  
 self.button = tk.Button(self, command=command, text=text)  
 self.show()  
  
 **def** show(self):  
 self.button.grid(column=0, row=0, sticky=**"nesw"**)  
  
 **def** hide(self):  
 self.button.grid\_remove()  
  
  
**class** TransitioningWidget(GUIwidget, tk.Frame):  
 *"""Abstract class for widgets which become labels in the 'playing' state"""* **def** \_\_init\_\_(self, master, \*options, \*\*kwargs):  
 *"""Options is a tuple of strings and kwargs is a dictionary of tkinter  
 settings for a given widget."""* tk.Frame.\_\_init\_\_(self, master)  
 self.child\_text = tk.StringVar(value=options[0])  
 self.child\_widget = self.get\_child\_widget(\*options, \*\*kwargs)  
 *# use the default font unless it is passed as a keyword argument* self.child\_label = tk.Label(self, text=options[0],  
 font=kwargs.get(**"font"**, **None**))  
  
 **def** display\_winner(self, game\_state):  
 self.transition\_to\_playing()  
  
 **def** transition\_to\_playing(self):  
 *# Replaces the widget with the label and updates the label text.* self.child\_widget.grid\_remove()  
 self.child\_label.configure(text=self.child\_text.get())  
 self.child\_label.grid()

This is an example of an interface.

TransitioningWidget class continued in next page.

TransitioningWidget class continued:   
  
 **def** transition\_to\_paused(self):  
 *# Replaces the label with the widget.* self.child\_label.grid\_remove()  
 self.child\_widget.grid()  
  
 **def** get\_child\_widget(self, options, command, font):  
 *"""Abstract method which gets the widget which is being used."""* **raise** NotImplementedError  
  
 **def** display\_state(self, game\_state):  
 **pass  
  
  
class** TransitioningDropdown(TransitioningWidget):  
 **def** get\_child\_widget(self, \*options, \*\*kwargs):  
 **return** tk.OptionMenu(self, self.child\_text, \*options, \*\*kwargs)  
  
  
**class** TransitioningEntry(TransitioningWidget):  
 **def** get\_child\_widget(self, \*options, \*\*kwargs):  
 **return** tk.Entry(self, textvariable=self.child\_text, \*\*kwargs)  
  
  
**class** TransitioningPlayerTimer(TransitioningEntry):  
 *"""Displays how much thinking time one player has left."""* **def** \_\_init\_\_(self, master, is\_reds\_clock, \*options):  
 *# validation is used to ensure the entry contents are numeric* validate\_command = (master.register(self.validate\_user\_entry), **"%S"**)  
 super().\_\_init\_\_(master, \*options, font=(**"Times"**, 35, **"bold"**),  
 validate=**"key"**, validatecommand=validate\_command, width=5)  
 self.is\_reds\_clock = is\_reds\_clock  
  
 **def** display\_state(self, game\_state):  
 *# if in the 'playing' state, the player's time left updated* **if** game\_state.is\_timed:  
 **if** game\_state.reds\_move == self.is\_reds\_clock:  
 player\_seconds = game\_state.clock.current\_player\_seconds  
 background\_colour = **"green"  
 else**:  
 player\_seconds = game\_state.clock.enemy\_player\_seconds  
 background\_colour = **"grey"** self.child\_label.configure(  
 text=str(round(player\_seconds, 1)), bg=background\_colour)  
  
 **def** validate\_user\_entry(self, new\_character):  
 *"""Returns true if the new contents could be converted to a float."""* **try**:  
 float(self.child\_text.get() + new\_character)  
 **except** ValueError:  
 **return False  
 else**:  
 **return True  
  
 def** transition\_to\_paused(self):  
 *# changes the default text in the entry to how much time is left.* super().transition\_to\_paused()  
 self.child\_text.set(self.child\_label.cget(**'text'**))  
  
 **def** display\_winner(self, game\_state):  
 *# changes te background to red if this timer ran out.* **if** game\_state.timeout:  
 **if** game\_state.red\_winner != self.is\_reds\_clock:  
 self.child\_label.configure(text=**"0.0"**, bg=**"red"**)

connect4user\_interface.py file continued in next page.

connect4user\_interface.py file continued:

**class** PlayingGUI(ConnectFourUserInterface):  
 *"""Uses the tkinter library to create a GUI for playing connect4."""* **def** \_\_init\_\_(self, master):  
 master.title(**"connect4"**)  
 *# widgets is an array of TransitioningWidgets to be looped over* self.widgets = []  
 self.setup\_base\_frames(master)  
 self.board\_canvas = PlayingGUI.BoardCanvas(master)  
 self.widgets.append(self.board\_canvas)  
 self.move\_buttons = []  
 self.setup\_move\_button\_array()  
 self.pause\_play\_reset\_button = PlayingGUI.PausePlayResetButton(self)  
 self.widgets.append(self.pause\_play\_reset\_button)  
 self.game\_info\_tab = PlayingGUI.GameInfoTab(self)  
 self.widgets.append(self.game\_info\_tab)  
 self.timer = PlayingGUI.GUItimer(self)  
 self.widgets.append(self.timer)  
 self.player\_info\_tab = PlayingGUI.PlayerInfoTab(self)  
 self.widgets.append(self.player\_info\_tab)  
 *# sets up the board with a new game and in the paused state* self.reset()  
  
 **def** copy\_game(self, red, yellow, timed\_game):  
 *"""Used by pause and play to create a copy of current\_game"""* board\_copy = deepcopy(self.current\_game.current\_board)  
 move\_list\_copy = deepcopy(self.current\_game.move\_list)  
 **if** timed\_game:  
 self.current\_game = PlayingGameState(  
 red, yellow, self, self.current\_game.reds\_move,  
 board\_copy, move\_list\_copy,  
 self.timer.get\_player\_seconds(self.current\_game.reds\_move))  
 **else**:  
 self.current\_game = PlayingGameState(  
 red, yellow, self, self.current\_game.reds\_move,  
 board\_copy, move\_list\_copy)  
  
 **def** display\_state(self, game\_state):  
 **for** widget **in** self.widgets:  
 widget.display\_state(game\_state)  
  
 **def** display\_winner(self, game\_state):  
 **for** widget **in** self.widgets:  
 widget.display\_winner(game\_state)  
  
 **def** pause(self):  
 *"""Stops the timer and allows for the editing of the current game."""  
 # copies the current\_game,  
 # but makes both players human so that the board can be edited* red = PlayingGUI.GUIHumanPlayer(**"red"**, self)  
 yellow = PlayingGUI.GUIHumanPlayer(**"yellow"**, self)  
 self.copy\_game(red, yellow, **False**)  
 **for** widget **in** self.widgets:  
 widget.transition\_to\_paused()  
 self.current\_game.play()  
  
 **def** play(self):  
 *"""Plays the game from the current game displayed on the GUI."""* red = self.player\_info\_tab.get\_AI(**True**)  
 yellow = self.player\_info\_tab.get\_AI(**False**)  
 self.copy\_game(red, yellow, **True**)  
 **for** widget **in** self.widgets:  
 widget.transition\_to\_playing()  
 self.current\_game.play()  
  
 **def** reset(self):  
 *"""Clears the board and transitions the GUI to a 'paused' state."""* dummy = AI.Dummy()  
 self.current\_game = PlayingGameState(dummy, dummy, self)  
 self.pause()  
 self.timer.reset\_timer()  
 self.display\_state(self.current\_game)  
PlayingGUI class continued in next page.

PlayingGUI class continued:

**def** show\_move\_buttons(self, game\_state):  
 **for** move **in** game\_state.get\_possible\_moves():  
 self.move\_buttons[move].show()  
  
 **def** hide\_move\_buttons(self):  
 **for** button **in** self.move\_buttons:  
 button.hide()  
  
 **def** create\_move\_button\_command(self, move):  
 *"""Hides the move buttons and makes the chosen move on the board"""* self.hide\_move\_buttons()  
 self.current\_game.make\_move(move)  
  
 **def** setup\_move\_button\_array(self):  
 *"""Creates an array of buttons for human players to make moves with."""* **for** i **in** range(7):  
 self.move\_buttons.append(PlayingGUI.MoveButton(self, i))  
 self.move\_buttons[i].grid(column=i, row=0, padx=10, pady=10)  
 self.hide\_move\_buttons()  
  
 **def** setup\_base\_frames(self, master):  
 *"""Places three frames on the root frame to fill them with widgets."""* master.resizable(**False**, **False**)  
 self.master\_frame = tk.Frame(master)  
 self.master\_frame.grid()  
 self.above\_board\_frame = tk.Frame(  
 master, width=700, height=100, bg=**"white"**)  
 self.right\_hand\_frame = tk.Frame(  
 master, width=250, height=700, bg=**"white"**)  
 self.above\_board\_frame.grid\_propagate(**False**)  
 self.above\_board\_frame.grid(column=0, row=0)  
 self.right\_hand\_frame.grid\_propagate(**False**)  
 self.right\_hand\_frame.grid(column=1, row=0, rowspan=2)  
  
 **class** GUIHumanPlayer(AI.ConnectFourEntity):  
 *"""For a human player to play using the move buttons."""* **def** \_\_init\_\_(self, name, parent\_GUI):  
 super().\_\_init\_\_(name)  
 self.parent\_GUI = parent\_GUI  
  
 **def** play\_chosen\_move(self, game\_state):  
 self.parent\_GUI.show\_move\_buttons(game\_state)  
  
 **class** BoardCanvas(tk.Canvas, GUIwidget):  
 *"""Displays the board in it's current state"""* **def** \_\_init\_\_(self, root):  
 super().\_\_init\_\_(root, width=700, height=600, bg=**"dark blue"**)  
 *# makes a 2D array of ovals of width 80 and a spacing of 20.* self.oval\_array = [[self.create\_oval((10 + 100 \* i, 10 + 100 \* j,  
 100 \* i + 90, 100 \* j + 90),  
 fill=**"white"**)  
 **for** i **in** range(7)] **for** j **in** range(6)]  
 self.grid(column=0, row=1)  
  
 **def** display\_state(self, game\_state):  
 **for** i **in** range(6):  
 **for** j **in** range(7):  
 self.update\_canvas\_location(  
 (i, j), game\_state.current\_board.array[i][j])  
  
 **def** update\_canvas\_location(self, position, counter\_code):  
 *"""Updates a specific position on the canvas with a counter code."""* **if** counter\_code == 0:  
 self.itemconfig(self.oval\_array[position[0]][position[1]],  
 fill=**"white"**)  
 **elif** counter\_code == 1:  
 self.itemconfig(self.oval\_array[position[0]][position[1]],  
 fill=**"red"**)  
 **else**:  
 self.itemconfig(self.oval\_array[position[0]][position[1]],  
 fill=**"yellow"**)

BoardCanvas class continued in next page.

BoardCanvas class continued:

**def** transition\_to\_playing(self):  
 **pass  
  
 def** transition\_to\_paused(self):  
 **pass  
  
 def** display\_winner(self, game\_state):  
 **if not** game\_state.timeout:  
 self.display\_state(game\_state)  
  
 **class** MoveButton(NonResizableButton):  
 *"""Enables a human player to place counters on the board"""* **def** \_\_init\_\_(self, parent\_GUI, i):  
 command = **lambda** move=i: parent\_GUI.create\_move\_button\_command(move)  
 super().\_\_init\_\_(parent\_GUI.above\_board\_frame, 80, 80, command, **""**)  
  
 **class** PausePlayResetButton(NonResizableButton, GUIwidget):  
 *"""Enables the player to transition the GUI state."""* **def** \_\_init\_\_(self, parent\_GUI):  
 self.play\_command = **lambda**: parent\_GUI.play()  
 self.pause\_command = **lambda**: parent\_GUI.pause()  
 self.reset\_command = **lambda**: parent\_GUI.reset()  
 super().\_\_init\_\_(parent\_GUI.right\_hand\_frame, 230, 80,  
 self.play\_command, text=**"play from here"**)  
 self.grid(row=0, column=0, padx=10, pady=10)  
  
 **def** display\_state(self, game\_state):  
 **pass  
  
 def** transition\_to\_playing(self):  
 self.button.configure(text=**"pause"**, command=self.pause\_command)  
  
 **def** transition\_to\_paused(self):  
 self.button.configure(text=**"play from here"**, command=self.play\_command)  
  
 **def** display\_winner(self, game\_state):  
 self.button.configure(text=**"reset"**, command=self.reset\_command)  
  
 **class** GUItimer(GUIwidget):  
 *"""Displays or enables editing of how much time each player has left."""* **def** \_\_init\_\_(self, parent\_GUI):  
 *# creates a frame to place the timers in* self.timer\_frame = tk.Frame(parent\_GUI.right\_hand\_frame,  
 width=230, height=80, bg=**"white"**)  
 self.timer\_frame.grid(row=1, column=0, padx=10, pady=10)  
 self.timer\_frame.grid\_propagate(**False**)  
 self.red\_title\_label = tk.Label(self.timer\_frame,  
 text=**"red time:"**)  
 self.yellow\_title\_label = tk.Label(self.timer\_frame,  
 text=**"yellow time:"**)  
 *# Initialises TransitioningPlayerTimer objects for each player* self.red\_timer = TransitioningPlayerTimer(  
 self.timer\_frame, **True**, str(DEFAULT\_PLAYER\_SECONDS))  
 self.yellow\_timer = TransitioningPlayerTimer(  
 self.timer\_frame, **False**, str(DEFAULT\_PLAYER\_SECONDS))  
 self.red\_title\_label.grid(row=0, column=0)  
 self.yellow\_title\_label.grid(row=0, column=1)  
 self.red\_timer.grid(row=1, column=0, sticky=**"e"**)  
 self.yellow\_timer.grid(row=1, column=1, sticky=**"w"**)  
  
 **def** display\_state(self, game\_state):  
 self.red\_timer.display\_state(game\_state)  
 self.yellow\_timer.display\_state(game\_state)  
  
 **def** transition\_to\_playing(self):  
 self.red\_timer.transition\_to\_playing()  
 self.yellow\_timer.transition\_to\_playing()  
  
 **def** transition\_to\_paused(self):  
 self.red\_timer.transition\_to\_paused()  
 self.yellow\_timer.transition\_to\_paused()

GUItimer class continued in next page.

GUItimer class continued:

**def** display\_winner(self, game\_state):  
 self.red\_timer.display\_winner(game\_state)  
 self.yellow\_timer.display\_winner(game\_state)  
  
 **def** get\_player\_seconds(self, reds\_move):  
 *"""Returns a tuple containing how much time each player has."""* **if** reds\_move:  
 current\_player\_timer = self.red\_timer  
 enemy\_player\_timer = self.yellow\_timer  
 **else**:  
 enemy\_player\_timer = self.red\_timer  
 current\_player\_timer = self.yellow\_timer  
 current\_player\_seconds = float(current\_player\_timer.child\_text.get())  
 enemy\_player\_seconds = float(enemy\_player\_timer.child\_text.get())  
 **return** current\_player\_seconds, enemy\_player\_seconds  
  
 **def** reset\_timer(self):  
 self.red\_timer.child\_text.set(str(DEFAULT\_PLAYER\_SECONDS))  
 self.yellow\_timer.child\_text.set(str(DEFAULT\_PLAYER\_SECONDS))  
  
 **class** GameInfoTab(GUIwidget):  
 *"""Displays whose move it is and the list of previous moves"""* **def** \_\_init\_\_(self, parent\_GUI):  
 self.game\_info\_frame = tk.Frame(parent\_GUI.right\_hand\_frame,  
 width=230, height=230)  
 self.game\_info\_frame.grid\_configure()  
 self.game\_info\_frame.grid(row=2, column=0, padx=10, pady=10)  
 self.game\_info\_frame.grid\_propagate(**False**)  
 self.player\_move\_label = tk.Label(self.game\_info\_frame, text=**""**)  
 *# creates arrays of labels to show the list of previous moves* self.red\_move\_labels = [  
 tk.Label(self.game\_info\_frame, text=**" "**) **for** \_ **in** range(21)]  
 self.yellow\_move\_labels = [  
 tk.Label(self.game\_info\_frame, text=**" "**) **for** \_ **in** range(21)]  
 self.player\_move\_label.grid(row=0, columnspan=6)  
 *# places the move labels in a grid* **for** i **in** range(7):  
 **for** j **in** range(3):  
 self.red\_move\_labels[i+j\*7].grid(row=i+1, column=j,  
 padx=13, pady=4)  
 self.yellow\_move\_labels[i+j\*7].grid(row=i+1, column=j+3,  
 padx=13, pady=4)  
  
 **def** display\_state(self, game\_state):  
 *# updates whose move it is and the* **if** game\_state.reds\_move:  
 current\_player\_name = game\_state.red.name  
 text\_colour = **"red"  
 else**:  
 current\_player\_name = game\_state.yellow.name  
 text\_colour = **"yellow"** current\_player\_name += **"'s move"** self.player\_move\_label.configure(text=current\_player\_name,  
 fg=text\_colour)  
 self.update\_move\_labels(game\_state.move\_list)  
  
 **def** transition\_to\_playing(self):  
 **pass  
  
 def** transition\_to\_paused(self):  
 **pass**

GameInfoTab class continued in next page.

GameInfoTab class continued:

**def** display\_winner(self, game\_state):  
 **if** game\_state.timeout:  
 text\_colour = **"black"** message = **"timeout!"  
 elif** game\_state.is\_drawn:  
 text\_colour = **"black"** message = **"draw!"  
 elif** game\_state.red\_winner:  
 text\_colour = **"red"** message = **"red wins!"  
 else**:  
 text\_colour = **"yellow"** message = **"yellow wins!"** self.player\_move\_label.configure(text=message, fg=text\_colour)  
 **if not** game\_state.timeout:  
 self.update\_move\_labels(game\_state.move\_list)  
  
 **def** update\_move\_labels(self, move\_list):  
 *"""Prints the move list into the grid of labels"""* **def** get\_move\_label(counter):  
 **if** counter % 2 == 0:  
 **return** self.red\_move\_labels[int(counter/2)]  
 **else**:  
 **return** self.yellow\_move\_labels[int((counter-1)/2)]  
 **for** i, move **in** enumerate(move\_list):  
 get\_move\_label(i).configure(text=str(move+1))  
 **for** i **in** range(len(move\_list), 42):  
 get\_move\_label(i).configure(text=**" "**)  
  
 **class** PlayerInfoTab(GUIwidget):  
 *"""Lets user change player types when the game in a 'paused' state."""* **def** \_\_init\_\_(self, parent\_GUI):  
 self.transitioning\_widgets = []  
 self.parent\_GUI = parent\_GUI  
 self.player\_info\_frame = tk.Frame(parent\_GUI.right\_hand\_frame,  
 width=230, height=220)  
 self.player\_info\_frame.grid(row=3, column=0, padx=10, pady=10)  
 self.player\_info\_frame.grid\_propagate(**False**)  
 *# creates dropdown menus which show a load parameter entry box if  
 # the AI selected is trainable* **def** red\_type\_dropdown\_command(value):  
 self.show\_if\_trainable\_AI(**True**, value)  
 self.red\_type\_dropdown = TransitioningDropdown(  
 self.player\_info\_frame, \*AI\_TYPE\_STRINGS,  
 command=red\_type\_dropdown\_command,)  
 self.transitioning\_widgets.append(self.red\_type\_dropdown)  
  
 **def** yellow\_type\_dropdown\_command(value):  
 self.show\_if\_trainable\_AI(**False**, value)  
 self.yellow\_type\_dropdown = TransitioningDropdown(  
 self.player\_info\_frame, \*AI\_TYPE\_STRINGS,  
 command=yellow\_type\_dropdown\_command,)  
 *# Creates widgets for a filepath to a model's weights be entered.* self.transitioning\_widgets.append(self.yellow\_type\_dropdown)  
 self.red\_weights\_entry = TransitioningEntry(  
 self.player\_info\_frame, **""**)  
 self.transitioning\_widgets.append(self.red\_weights\_entry)  
 self.yellow\_weights\_entry = TransitioningEntry(  
 self.player\_info\_frame, **""**)  
 *# Creates widgets for players' names to be entered.* self.transitioning\_widgets.append(self.yellow\_weights\_entry)  
 self.red\_name\_entry = TransitioningEntry(  
 self.player\_info\_frame, **"red"**)  
 self.transitioning\_widgets.append(self.red\_name\_entry)  
 self.yellow\_name\_entry = TransitioningEntry(  
 self.player\_info\_frame,**"yellow"**)  
 self.transitioning\_widgets.append(self.yellow\_name\_entry)  
 self.setup\_labels()  
 self.place\_widgets()

PlayerInfoTab class continued in next page.

PlayerInfoTab class continued:  
 **def** show\_if\_trainable\_AI(self, red\_AI, string):  
 *"""Makes the weights entry widget show if the AI is trainable."""* **if** red\_AI:  
 parameter\_filepath\_label = self.red\_weights\_filepath\_label  
 parameter\_entry = self.red\_weights\_entry  
 row=4  
 **else**:  
 parameter\_filepath\_label = self.yellow\_weights\_filepath\_label  
 parameter\_entry = self.yellow\_weights\_entry  
 row=9  
 **if** AI\_type\_dict(string).is\_trainable:  
 parameter\_filepath\_label.configure(text=**"weights filepath:"**)  
 parameter\_entry.grid(row=row, column=1, sticky=**"w"**)  
 **else**:  
 parameter\_filepath\_label.configure(text=**" "**)  
 parameter\_entry.grid\_remove()  
  
 **def** setup\_labels(self):  
 self.title\_label = tk.Label(self.player\_info\_frame,  
 text=**"player information"**)  
 self.red\_label = tk.Label(self.player\_info\_frame, text=**"red"**)  
 self.yellow\_label = tk.Label(self.player\_info\_frame, text=**"yellow"**)  
 self.name\_label\_1 = tk.Label(self.player\_info\_frame, text=**"name:"**)  
 self.name\_label\_2 = tk.Label(self.player\_info\_frame, text=**"name:"**)  
 self.type\_label\_1 = tk.Label(self.player\_info\_frame, text=**"type:"**)  
 self.type\_label\_2 = tk.Label(self.player\_info\_frame, text=**"type:"**)  
 self.red\_weights\_filepath\_label = tk.Label(  
 self.player\_info\_frame, text=**" "**)  
 self.yellow\_weights\_filepath\_label = tk.Label(  
 self.player\_info\_frame, text=**" "**)  
  
 **def** place\_widgets(self):  
 self.title\_label.grid(row=0, columnspan=2)  
 self.red\_label.grid(row=1, column=0, sticky=**"w"**)  
 self.name\_label\_1.grid(row=2, column=0, sticky=**"e"**)  
 self.red\_name\_entry.grid(row=2, column=1, sticky=**"w"**)  
 self.type\_label\_1.grid(row=3, column=0, sticky=**"e"**)  
 self.red\_type\_dropdown.grid(row=3, column=1, sticky=**"w"**)  
 self.red\_weights\_filepath\_label.grid(row=4, column=0, sticky=**"e"**)  
 self.player\_info\_frame.grid\_rowconfigure(5, weight=10)  
 self.yellow\_label.grid(row=6, column=0, sticky=**"w"**)  
 self.name\_label\_2.grid(row=7, column=0, sticky=**"e"**)  
 self.yellow\_name\_entry.grid(row=7, column=1, sticky=**"w"**)  
 self.type\_label\_2.grid(row=8, column=0, sticky=**"e"**)  
 self.yellow\_type\_dropdown.grid(row=8, column=1, sticky=**"w"**)  
 self.yellow\_weights\_filepath\_label.grid(row=9, column=0, sticky=**"e"**)  
  
 **def** get\_AI(self, red\_AI):  
 *"""Returns an AI generated form the user's entry in the tab"""  
 # Gets the correct widgets and variables for if the requested  
 # AI is the red player or the yellow player.* **if** red\_AI:  
 AI\_type\_string = self.red\_type\_dropdown.child\_text.get()  
 name = self.red\_name\_entry.child\_text.get()  
 weights\_entry = self.red\_weights\_entry  
 **else**:  
 AI\_type\_string = self.yellow\_type\_dropdown.child\_text.get()  
 name = self.yellow\_name\_entry.child\_text.get()  
 weights\_entry = self.yellow\_weights\_entry  
 *# Gets the AI type from the string retrieved from the dropdown menu.* AI\_type = AI\_type\_dict(AI\_type\_string)  
 *# If the AI type is trainable, an attempt is made to load the model  
 # in from the filepath entered, otherwise random weights are used.* **if** AI\_type.is\_trainable:  
 AI = AI\_type(name, **False**)  
 weights\_filepath = weights\_entry.child\_text.get()  
 **if** os.path.isfile(weights\_filepath):  
 AI.load\_parameters(weights\_filepath)  
 **else**:  
 weights\_entry.child\_text.set(**"no weights found."**)  
 **return** AI  
 **elif** AI\_type == PlayingGUI.GUIHumanPlayer:  
 **return** AI\_type(name, self.parent\_GUI)  
 **else**:  
 **return** AI\_type(name)  
connect4user\_interface file class continued:  
 **def** display\_state(self, game\_state):  
 **pass  
  
 def** transition\_to\_playing(self):  
 **for** widget **in** self.transitioning\_widgets:  
 widget.transition\_to\_playing()  
  
 **def** transition\_to\_paused(self):  
 **for** widget **in** self.transitioning\_widgets:  
 widget.transition\_to\_paused()  
  
 **def** display\_winner(self, game\_state):  
 self.transition\_to\_playing()  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 *# If the user interface module is explicitly run, the GUI is opened.* root = tk.Tk()  
 PlayingGUI(root)  
 root.mainloop()

These are examples of AI objects being dynamically generated at runtime based on the user input.

### Techniques Used

##### Object Oriented Techiniques

I tried to implement my technical solution using the object-oriented paradigm, so I have many examples of when I have used a given technique. I have thus only picked out the most obvious examples of where a technique has been used. The overall structure of my OOP model is sketched out in a UML class diagram in the design section, although my finished technical solution does not completely adhere to it.

* Interfaces. The most obvious example of an Interface in my code is the GUIwidget class in the connect4user\_interface module on page 45 where all classes that inherited from it had to transition their GUI state in a different way. Note that python does not include explicit definitions of interfaces, instead; one simply defines a class with only abstract methods.
* Polymorphism and Inheritance. One example of where I used polymorphism and inheritance for the AI is in the connect4AI module, where all types of AI inherit from the ConnectFourEntity class and override the abstract method choose\_move differently. These are highlighted in the AI module on the pages 28 and 29.
* Composition. I have defined the GameState class to encapsulate Board objects, as a result, the board object is composed onto the GameState. This is highlighted in the game module on page 27.
* Dynamic generation of objects. In my the connect4user\_interface module ConnectFourEntity objects Are created at runtime depending on the user’s input. This is highlighted on page 52.

##### Algorithms

Group A:

* Hash tables and Stacks. I have composed both a stack and a hash table onto my TrainingExampleContainer data structure; which I have highlighted on page 39. Using both of these was necessary as I needed to ensure that operations to do with the container were O(1) as the container is so large in size.
* Trees. I have defined the UCTsearchNode object to represent a node in a decision tree in my UCT search algorithm. This node keeps an array of, references to it’s child nodes, which are initialise when the expand function is called on this node. This is highlighted on page 36 in the constructor for the UCT search node.
* Recursive algorithms. Due to the nature of my project, I have used two examples of recursion. One of these is my implementation of the MiniMax algorithm highlighted on pages 31 and 32 in the expand and analyse functions. UCT tree search also incorporates a recursive function, namely the expand node function highlighted on page 36, to play simulated games and build a decision tree.
* Advanced matrix operations. I’m less sure about whether my project satisfies this guideline as, whilst in using TensorFlow to train and get predictions from neural networks many matrix operations are happening, much of this is abstracted away from me as a user of the library. I have, however, highlighted my manipulations of a numpy n-dimensional array before it is fed into the neural network as an input matrix on page 31.

Group B:

* Multi-dimensional arrays. I have used a two dimensional array on page 25 to store a representation of the board.
* Dictionaries. I have used a dictionary as part of my implementation of the MiniMax algorithm in the form of transposition table, highlighted on page 32.
* Statistical and mathematical operations. I have performed SoftMax on an array on page 38 to transform a vector of values into a probability distribution. I have also implemented a function which calculates the UCT value of a game state on page 37.

## Testing

For the appropriate tests, I have recorded video evidence of them being carried out for clarity. These testing videos can be found on the ALevel\_CompSci\_NEA\_GCS youtube channel in, I have included audio transcripts and the timestamps for the different tests in this document.

### GUI Testing

##### test 1.1 - General functionality

video timestamp: 0:00

audio transcript: “As we can see, the GUI application can be used to play games between two human players by clicking the ‘play’ button. In this playing state, the GUI displays information about the game such as the current players, a list of moves and how much time each player has left. At any point, the pause game button can be pressed to transition the GUI state back to a paused state, where we can change the players, their names, how much time they have left etc. If at any point, a counter is placed on the board which results in a win in the paused or playing states, the GUI is transitioned to a won game state, form here, the only thing that can be done is reset the game.”

##### Test 1.2 - Further timer testing

video timestamp: 1:35

audio transcript: “I have included the key presses in this recording to help illustrate this test. As already shown, we can change the player’s time left in the paused state, if we try to type in a value that would make the timer no – longer a float, it is rejected. If a timer runs out of time, the game is stopped and visual indicators are shown, such as a timeout message and the timer colour going red.”

### AI Testing

##### general Minimax test – 2.1

video timestamp: 2:37

audio transcript: “We can use the GUI application’s board editing features to set up a position where yellow needs to play to the third or sixth column to prevent red from getting a three in the bottom row which would lead to them being able to force a win. As we can see if yellow is then set to choose based on the MiniMax algorithm, it plays the correct move to prevent red from winning whilst still playing close to the centre. The algorithm will choose the columns closest to the central one unless it is forced to prevent the other player from forcing a win or it can make a four in a row itself.”

##### transposition table test – 2.2

video timestamp: 4:13

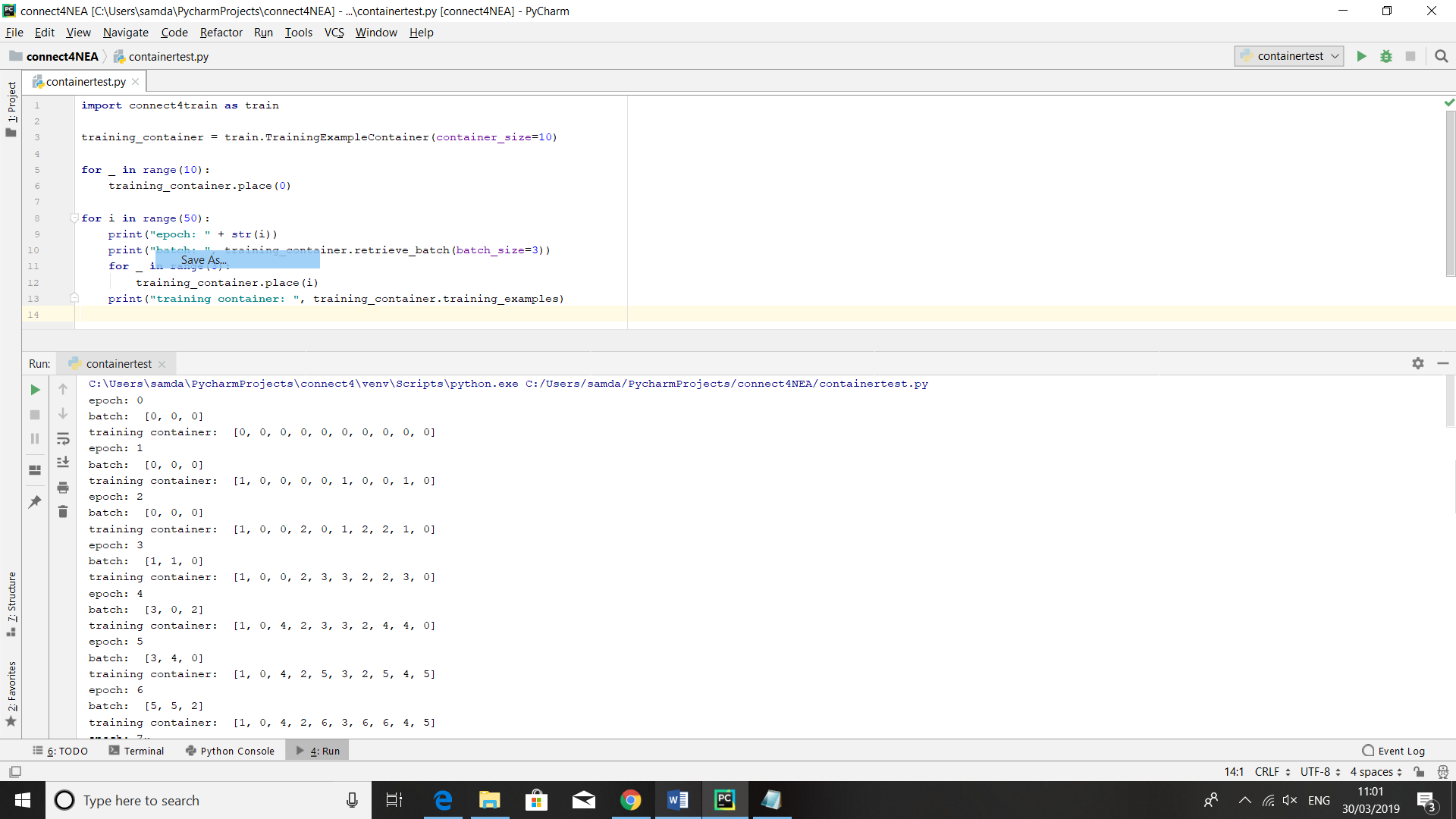
audio transcript: For this test, I have changed the search depth of the MiniMax algorithm to three moves deep and added in a couple lines of code such that if the algorithm finds that it is trying to analyse the same board twice, it prints the board into the console. With the GUI application, we can then get the algorithm to choose a move and see which boards have been found to be repeated in the calculations because they can be reached through different orders of the same moves.”

##### Test 2.3 - Training container

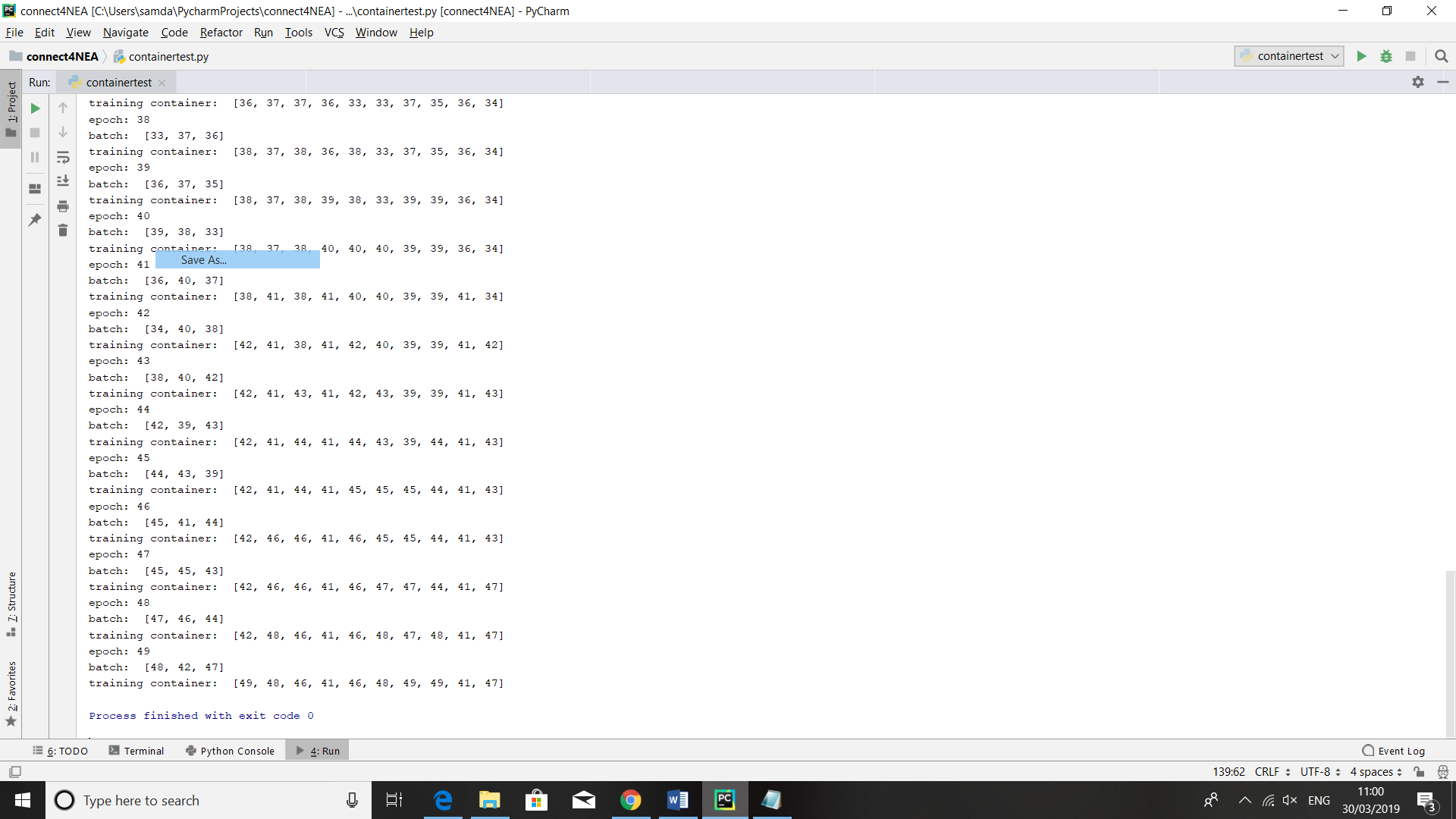
I first ran a test to ensure that my TrainingExampleContainer data structure was working the way I intended it to. To do this, I created a container that was small and placed Integer values into it to represent training examples. I then ran number of simulated “epochs” where I retrieved a batch of three values and replaced them with three integers with the same value as the epoch number. Here is the code I ran in a file called containertest.py to do this:

**import** connect4train **as** train  
  
training\_container = train.TrainingExampleContainer(container\_size=10)  
  
**for** \_ **in** range(10):  
 training\_container.place(0)  
  
**for** i **in** range(50):  
 print(**"epoch: "** + str(i))  
 print(**"batch: "**, training\_container.retrieve\_batch(batch\_size=3))  
 **for** \_ **in** range(3):  
 training\_container.place(i)  
 print(**"training container: "**, training\_container.training\_examples)

result sample:

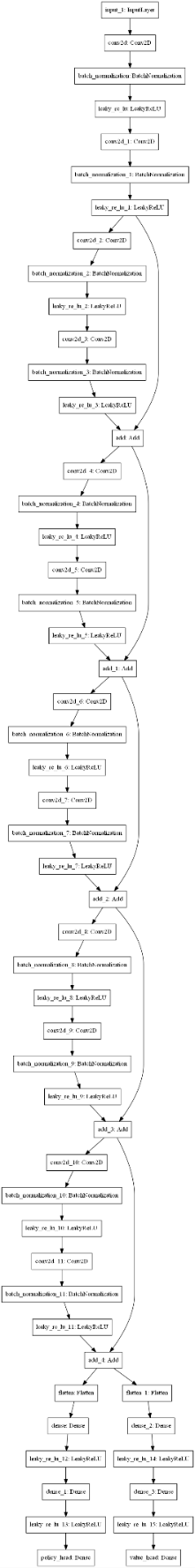
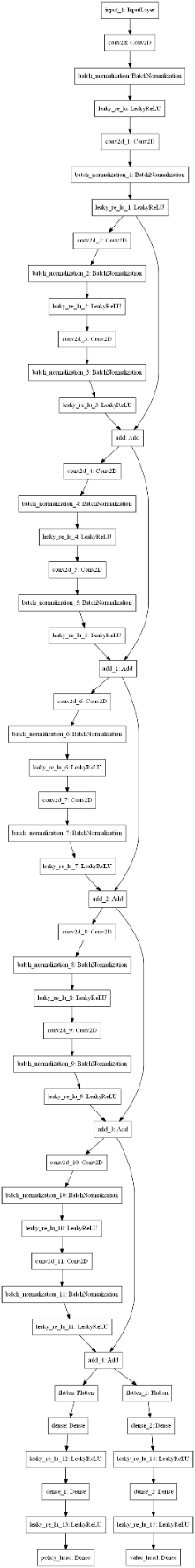


…



This clearly shows the TrainingExampleContainer randomly retrieving the batches successfully and filling in the “holes” that are left. On top of this, we can see that the batches are sufficiently varied in terms of epoch number to avoid the overfitting problems I discussed in my design whilst still retrieving relatively recent training examples so that the AI can learn new tactics.

##### Test 2.4 – Model Architecture check

I ran a test to check that the model architecture keras had interpreted from my use of it’s functional API (this is generally considered good practice to prevent “silent” bugs where the architecture of the model isn’t what one thought it was). To do this, I initialised an alphazero clone with random weights and used the keras’ plot model method which uses a package called Graphviz to plot the model. The I used code to do this is shown below along with a sample from the output (which has been shortened as there were many repeated residual blocks)

**import** connect4AI **as** AI

azclone = AI.AZclone(**"azclone"**, **True**)

keras.utils.plot\_model(azclone.model,to\_file=**"azclone\_architecture.png"**)

##### Test 2.4 – Trained Flat residual NN

Before running this code, I wrote a python script that used my training interface to train the residual network, called training.py:

**import** connect4AI **as** AI  
**import** connect4train **as** train

resnet = AI.ResidualNN(**"resnet\_v7"**, **True**)  
train\_env = train.TrainingEnvironment(resnet)  
**for** \_ **in** range(100):  
 train\_env.execute\_epoch()

(There were six attempts before this successful one which failed due to either bugs in the training interface, overfitting or exploding gradients for which I had to tweak the model architecture; hence the version number)

video timestamp: 5:08

audio transcript: “For this test I have made the flat residual network algorithm print out it’s move ratings for the first few moves and trained the algorithm using one hundred epochs of self-play. As we can see using the GUI application the network has learned to play towards the centre and will eventually create a row of four counters if you let it. However, the AI is incapable of generating long term strategies and will not look moves ahead if we put it in a situation where it needs to do so to prevent a forced win from its opponent.”

Overall, the development of the Flat resnet was helpful for me as a learning experience however it showed to me that “flat” networks were very unlikely to be able to solve the problem on their own. For this reason, I decided to not spend time developing “flat” dense and convolutional networks as I had planned to in my design as I feel it would not have contributed much to the investigation and taken up many of my time and computational resources.

##### test 2.5 – AlphaZero clone data collection

As part of my specification, I included the ability to collect training data through playing games, on top of this, it is good practice to check one’s input data as it can often cause “silent” errors where the neural network is trained based on erroneous data. Again, I wrote a python script to do this for my AlphaZero clone class and edited the assign\_ rewards\_and\_place method in the training interface class (page 41):

**import** connect4AI **as** AI  
**import** connect4train **as** train

azclone = AI.AZclone(**"azclone"**, **True**)  
train\_env = train.TrainingEnvironment(azclone)  
red, yellow = train\_env.create\_copies()  
train\_env.execute\_episode(red, yellow)

the editions to the training interface class:

**def** assign\_rewards\_and\_place(self, training\_examples, reward):  
 *# If the container is already full, no more examples are placed into it.* **for** training\_example **in** training\_examples:  
 training\_example.set\_reward(reward)  
 print(**"reward:"**, np.squeeze(training\_example.reward\_tensor))  
 print(**"stochastic policy"**, training\_example.policy\_tensor)  
 print(**"state: "**, np.squeeze(training\_example.state\_tensor))  
 **try**:  
 self.training\_example\_container.place(training\_example)  
 **except** FullContainerException:  
 **break**

A sample of the output:

This clearly shows the training examples being assigned roughly appropriate policy and value labels to train the network with. From prior experimentation, I changed the value labels to being an average of the reward and the improved value (the average value of this node) to help speed up the learning process, hence why the value labels are no longer simply 1, 0 or -1.

##### Test 2.6 – Trained Alphazero clone

Before this test, I ran a training script like the one used for the residual network overnight, in which it completed 40 epochs of training (Each epoch took significantly longer than with the flat residual network as the AI took longer to choose moves and therefore play games against itself).

video timestamp: 6:58

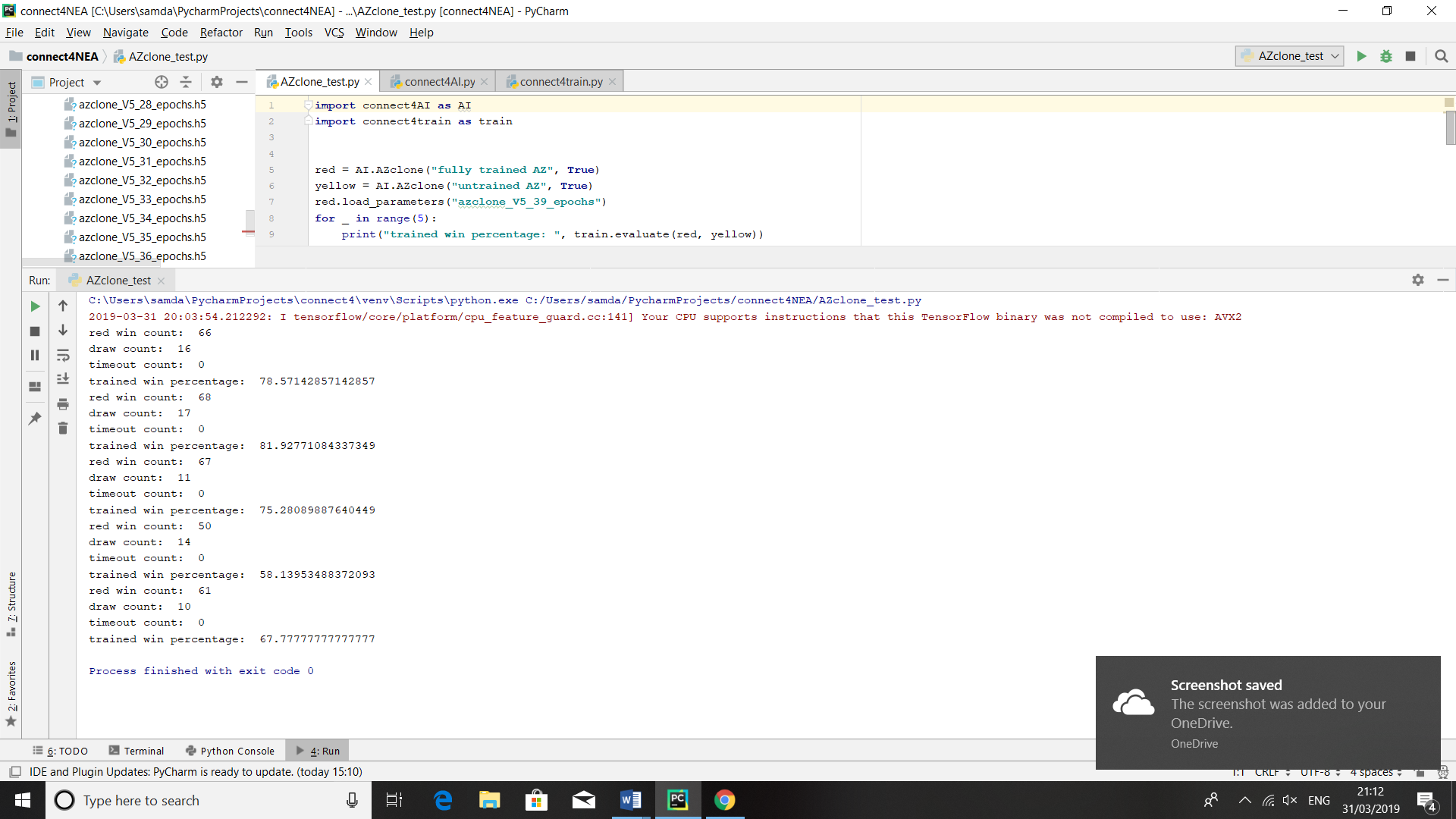
audio transcript: “For this test, I have made the AlphaZero algorithm printout the initial policy and value predictions from the network for the current state along with the improved policy and value once the calculation had been completed. With the GUI we can see that, after training from 40 epochs of self-play, the algorithm has learned to play towards the centre at the beginning of the game, will block its opponents’ attempts to make a four in-a-row and will make a row of four itself if allowed to. We can also see that the algorithm is capable of looking forward enough moves to prevent simple forced wins”

##### Test 2.7 – AlphaZero clone Improvement

In this test I used the evaluate function from my training module to test whether the AlphaZero clone I trained had actually improved in playing the game. The script I used to do this is shown below, note that the AIs had to be initialised to be “training” so that they would play randomly for the first few moves and not play the same game repetitively:

**import** connect4AI **as** AI  
**import** connect4train **as** train  
  
  
red = AI.AZclone(**"fully trained AZ"**, **True**)  
yellow = AI.AZclone(**"untrained AZ"**, **True**)  
red.load\_parameters(**"azclone\_V5\_39\_epochs"**)  
**for** \_ **in** range(5):  
 print(**"trained win percentage: "**, train.evaluate(red, yellow))

A screenshot of the output:



From analysing the output, I found that the average win rate for the trained version of the algorithm using the trained network was 72% across the 432 games played. This is definitely significant enough to show the algorithm has improved through the training (and may perhaps improve more if trained more).

##### test 2.8 – Timed UCT Functionality

video timestamp: 8:59

audio transcript: “For this test, I have made the Timed UCT algorithm print out how many simulations it has used to calculate a given move. Using the GUI, we can see that the untrained version of the algorithm will use fewer simulations when the move it needs to make is obvious, such as when it needs to block its opponent from making a row of four or it can make a row of four itself. In top of this, if we give the algorithm half the amount of time, it will use many fewer simulations to calculate its first move.”

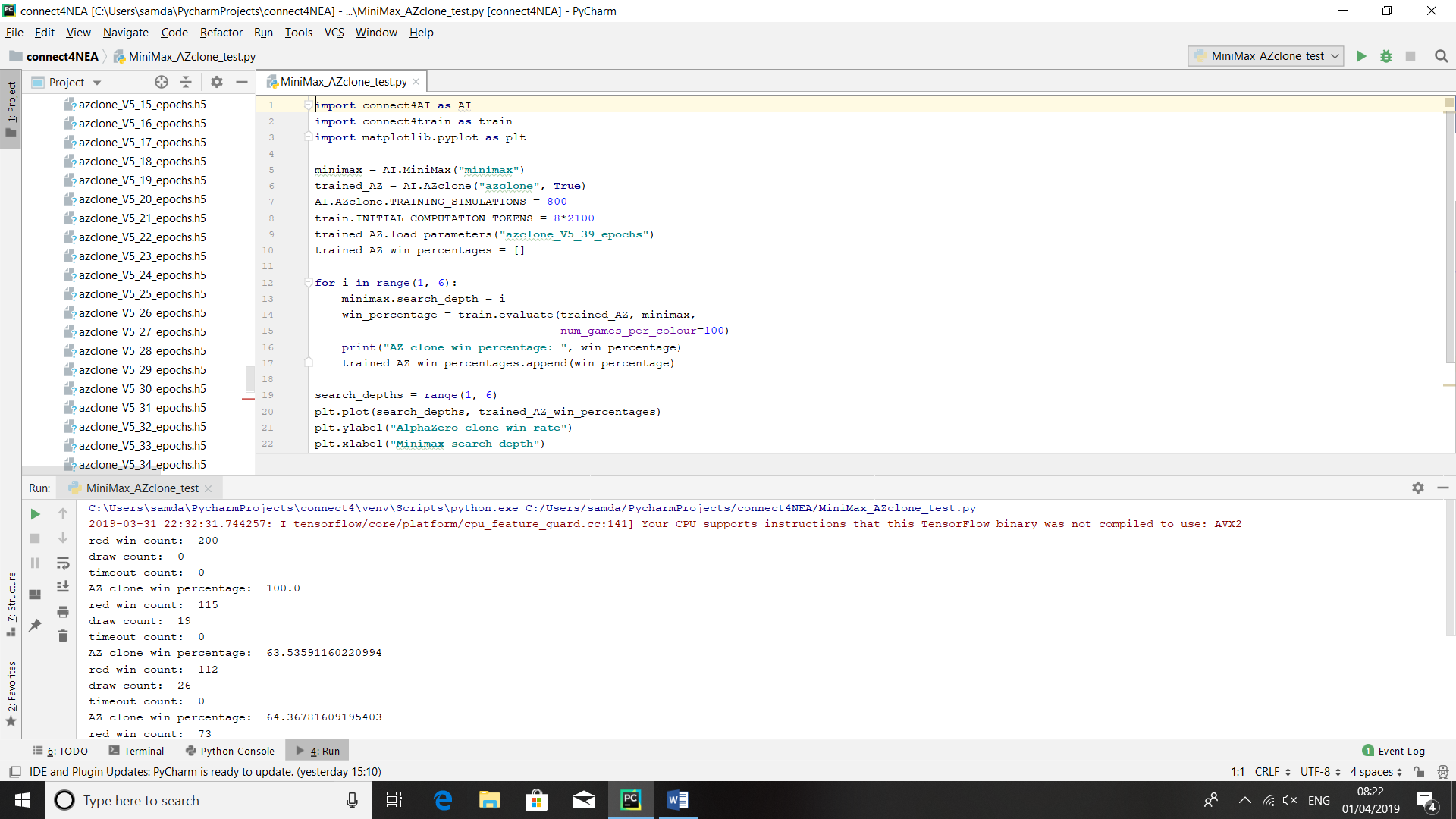
I was unable to train my timed UCT algorithm because of exploding gradients causing the network to give erroneous outputs. However, I believe this test shows that the idea behind the algorithm is promising and that someone with more experience with training neural networks would be able to get the algorithm working as intended.

##### test 2.9 – MiniMax vs alphazero clone

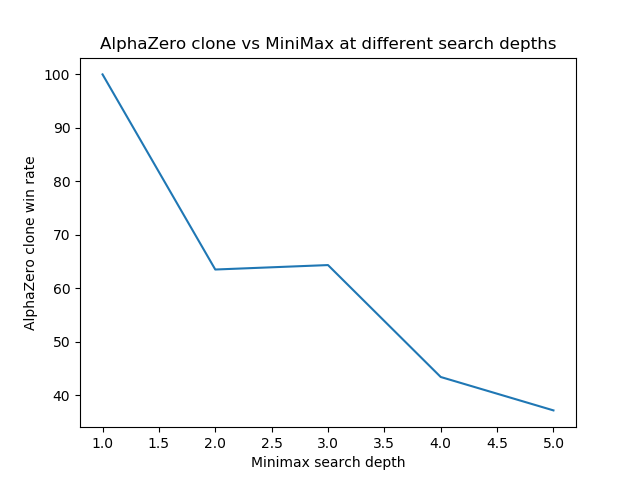
I also decided to test the AlphaZero clone against MiniMax at different search depths to measure its ability on a more absolute scale. One problem I encountered with this test was that, if the AlphaZero clone was not in the “training” state, the two AIs would just play the same game over and over. To remedy this, I set the AlphaZero to the “training” state but gave it the same amount of computation time as it would usually have when playing games on the GUI. This put the AlphaZero clone at a disadvantage as it meant it had to play randomly for the first few moves. The python script I used to do this compared the AlphaZero clone with the MiniMax algorithm at different search depths and then used a library called Matplotlib to draw a graph of the results.

**import** connect4AI **as** AI  
**import** connect4train **as** train  
**import** matplotlib.pyplot **as** plt  
  
minimax = AI.MiniMax(**"minimax"**)  
trained\_AZ = AI.AZclone(**"azclone"**, **True**)  
AI.AZclone.TRAINING\_SIMULATIONS = 800  
train.INITIAL\_COMPUTATION\_TOKENS = 8\*2100  
trained\_AZ.load\_parameters(**"azclone\_V5\_39\_epochs"**)  
trained\_AZ\_win\_percentages = []  
  
**for** i **in** range(1, 6):  
 minimax.search\_depth = i  
 win\_percentage = train.evaluate(trained\_AZ, minimax,  
 num\_games\_per\_colour=100)  
 print(**"AZ clone win percentage: "**, win\_percentage)  
 trained\_AZ\_win\_percentages.append(win\_percentage)  
  
search\_depths = range(1, 6)  
plt.plot(search\_depths, trained\_AZ\_win\_percentages)  
plt.ylabel(**"AlphaZero clone win rate"**)  
plt.xlabel(**"Minimax search depth"**)  
plt.title(**"AlphaZero clone vs MiniMax at different search depths"**)  
plt.savefig(**"azclone\_vs\_minimax.png"**)  
plt.show()

Screenshot of the output:



The graph outputted by Matplotlib is shown on the next page.



As we can see, the fully trained AlphaZero clone was capable of beating minimax most of the time at lower search depths, however, the scales tipped to MiniMax’s favour once the search depth went above four.

## Evaluation

### Completeness Based on Specification

##### AI completeness

Overall, the AI side of my technical solution has the potential to fulfil all the requirements that were laid out in the specification. However, my lack of experience and resources to train neural networks lead to me not getting as impressive results as I had hoped for

|  |  |  |  |
| --- | --- | --- | --- |
| Objective referenced | description | Completed? | comments |
| 1) | The AI should be able to: | | |
| 1) a) | Be able to learn to play connect4 independently, based on only the input of the game rules, this includes: | | |
| 1) a) i) | Collecting data from playing games against itself and the rewards it receives | Yes | This functionality is shown to successfully work for the AlphaZero clone in test 2.5, since this is done by the same code in the training environment for all AI, it is proven to work for them too. |
| 1) a) ii) | Using this data to train a neural network to estimate the policy and value of different actions and states | Yes | The results of using data like what was collected in 2.5 is shown most clearly in test 2.6, where the policy and value outputs of the neural network are clearly shown to have been trained to give useful information to the rest of the AlphaZero algorithm. |
| 1) a) iii) | Learning other parameters such as when to optimally cut off calculation | Potentially | The timed UCT implementation has the infrastructure to do this (shown in the method punish\_for\_timeouts defined on page 35 and used on page 41), however this was hard to test because of the other problem with training this algorithm. |
| 1) b) | Choosing moves based on: | | |
| 1) b) i) | The state that the game is in | Yes | Test 2.6 clearly shows the AlphaZero algorithm choosing the good moves to play in a given state. |
| 1) b) ii) | How much time it and the other player have remaining | Potentially | Test 2.8 shows timed UCT controlling how much calculation it and the other player have remaining. In terms of being able to use this information strategically, in theory the information passed to the timed UCT algorithm in the prepare\_input\_tensor method on page 35 should affect the value head output, However I could not train this algorithm to find this out. |
| 1) c) | Be reasonably proficient once fully trained: | | |
| 1) c) i) | Be able to beat a simple MiniMax algorithm consistently | Mostly | Test 2.9 shows the AlphaZero clone was able to beat MiniMax at lower search depths however couldn’t compete with the raw amount of computation that MiniMax could use to force wins once the MiniMax search depth went above four. |
| 1) c) ii) | Show clear improvements against lesser-trained versions of itself | Yes | Test 2.7 clearly shows the trained version of the AlphaZero clone was able to beat the untrained version by a substantial margin. |
| 1) d) | The simple MiniMax algorithm should: | | |
| 1) d) i) | Be able to play reasonably proficiently, including looking a certain number of moves ahead to try to force a win or draw | Yes | This functionality is shown in test 2.1. |
| 1) d) ii) | Use techniques such as hash tables and pruning to speed its decision process up | Mostly | The use of a dictionary was shown to work in test 2.2, I decided to not implement pruning into final solution as after doing further research, it seems it would not have a big impact in calculation times for connect4. |

##### GUI completeness

|  |  |  |  |
| --- | --- | --- | --- |
| Objective referenced | Description | Completed? | comments |
| 2) | The GUI application should be able to: | | |
| 2) a) | Play timed games between two players, where a player can either be a human player interacting with the GUI or a chosen AI, including: | | |
| 2) a) i) | Displaying a representation of the board on the screen | Yes | This functionality is clearly shown in test 1.1. |
| 2) a) ii) | Being easy enough to use that a naïve user could play a game | Yes | Test 1.1 shows the GUI in use. I would argue that the GUI is “signposted” well enough that someone would be able to play a game if they hadn’t seen it in use before |
| 2) a) iii) | Displaying a live representation of the timer | Mostly | The timer’s full functionality is shown in tests 1.1 and 1.2; however, I did not have the time to implement it in such a way that each player’s time is shown ticking down as a commercial app would do. |
| 2) a) iv) | Displaying live information about the game such as whose move it is and if someone has won | Yes | This again is shown clearly in test 1.1 |
| 2) b) | Enable a naïve user to interact with the game on screen whilst it is not being played, including: | | |
| 2) b) i) | Enabling the placing of counters and having a “clear board” functionality | Yes | The “clear board” functionality is shown at the end of test 2.1. Placing of counters whilst in the paused state is also shown in test 2.1 but perhaps more clearly in test 1.X |
| 2) b) ii) | Allocating different amounts of time to each player | Yes | This is shown in test 2.2 |
| 2) b) iii) | Being able to select a different type of player to play for red and yellow and having functionality to play from the current position displayed on the board | yes | This is shown in test 2.1 and in tests 1.X, etc. |

### Supervisor Feedback

To finish with the project, I asked my supervisor questions about how she thought the project went.

##### interview

In what aspects would you say this project is most successful?

“For the scope of this project, I would say that the results for the AlphaZero algorithm show that you were successful; although there’s definitely room for future improvement, I do not think it would be realistic to do all of that in the given timeframe.”

What aspects of the project need more investigation?

“I would say it would be interesting to experiment with different algorithms, trying to tweak the AlphaZero one to work better with connect4. Also, I think trying different neural network architectures could help you get even better results with less training.”

If I were to do another project on machine learning, what advice would you give?

“I would say that whilst you researched the fundamentals of how neural networks worked well, you should have focussed a bit more of your research on the more applied side of machine learning, as how to diagnose and fix problems that you get from training neural networks. I’m sure if you had done this you would find fixing the exploding gradients that your timed algorithm suffered from much easier.”

Would you say more investigations into machine learning should focus on trying to solve real-word problems rather than man-made games?

“I’d say there’s room for doing research in everything, at the end of the day it is all progress that could contribute to real-world problems. That said, I do think it research into things that improve everyday life can be more satisfying.”

##### conclusions from feedback

I think this feedback was very valuable in terms of shedding light on what I could have done to spend my time and computation resources more efficiently. After discussing this further with my supervisor, I think that I probably could have accomplished more if I had used a simpler convolutional architecture as I would have been able to prevent exploding gradients more easily and my laptop would have been able to process more predictions and therefore play more training games.

### Potential Improvements

##### Gui potential improvements

Overall, I’m pretty happy with how my GUI turned out, but there are a few things that I think the finished solution is missing. Probably the first thing I would address if I was to re-visit the GUI is to try to prevent the GUI from freezing when types of AI are calculating. Initially, I was going to solve this problem by running the choose move method on a separate thread. However, after doing further research it appears that this would not completely work as a solution because the Tkinter framework is limited in terms of working with threading, and the only solution to my problem would be a polling queue. So if I were to do my project again, I would do more research into which graphics library I wanted to use and what functionality it supported rather than simply picking the one that seems to be most popular. Other improvements to my current GUI include using threading to make the timers “tick down” and having a drag-and-drop functionality to allow the player to place counters.

At the start I also envisioned this project having a GUI that would allow the user to train different AIs and interact with the playing GUI (potentially showing the training games live as they happened). However, I quickly realised that this would take a substantial amount of time and would not really enhance the research side of the project and therefore chose to only implement the playing interface. This could also be another potential way that one could expand on my project if one had more time.

##### AI potential improvements

The most obvious thing that still needs work on the AI side of my project is getting the timed UCT algorithm to be able to train. To do this, I would need to experiment with different model architectures; which would take lots more processing time hence why I ran out of time to do it for this project. After this, I could improve on the AIs by tweaking all of the parameters to see whether I can get them to perform better and perhaps beat the MiniMax algorithm in the higher search depths; this could include training larger neural networks for more epochs.

One potential way I could have completed the aforementioned processes in the scope of this project would be to increase the amount of training that I could perform in a given timeframe. To do this, I could optimise my training interface to process the data more efficiently, for example by using a NumPy array to store the training examples rather than python lists or running multiple training games in parallel. On top of this, I could run the training on hardware more suited to do deep-learning style computations; for example if I ran the neural network processing on a graphics card it would be able to generate predictions and fit the model significantly quicker because it is more suited to doing matrix operations than the CPU. Another idea I had whilst doing this project would be to let the AlphaZero algorithm generate its own “opening book” whilst training so that it didn’t repetitively recalculate the same fist few moves in training games; however, I did not have time to implement this.

Finally, I could expand on my project by generalising the algorithms to more complete information games, such as Chess and Go, although this again would require perhaps an order of magnitude more computation.