JAMES HEARSUM
3/30/17

COMPUTING PRACTICAL PROJECT (7517/C)

Contents

Introduction

For my project, I wanted to investigate artificial intelligence (AI) and machine learning algorithms. This topic fascinated me because of the recent media coverage of AI and learning algorithms in the news. Ranging from self-driving cars, automated robots and computers that play games against humans. Personally, I was very interested in the competition between humans and computers, and how programs can be written that can outsmart a human at certain games and only get smarter as they play. Computers beating humans at games, using learning algorithms rather than a set of best moves, dates back to May 1997, when Deep Blue beat the world chess champion at his own game in a very high profile tournament. More recently, in April 2016, AlphaGo beat the world go champion, go being a very strategic and complicated game (Johson, 2016). Both are regarded as massive achievements. I am interested in the processes a computer goes through in order to play to such a level it can predict and trap human players, no matter what strategy they think of and implement.

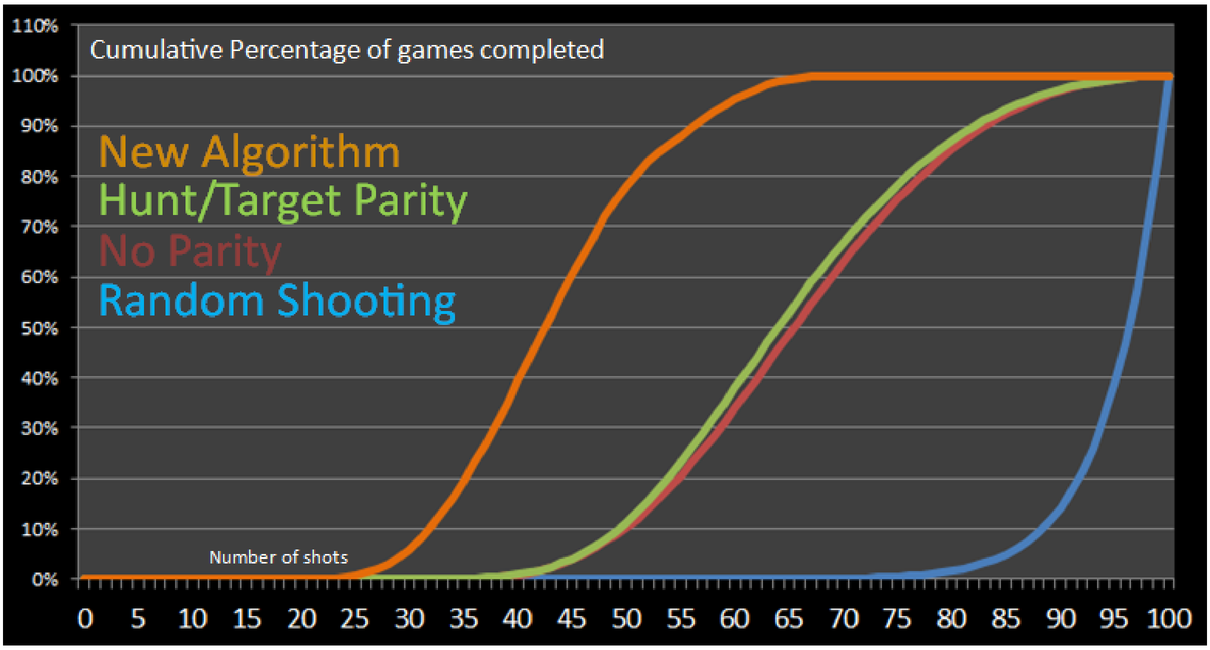
I have been tasked by Mr M, a maths teacher who would like their identity to remain anonymous, to look at what other games could potentially be played by a learning algorithm. They want to be shown data from a computer program I have created, that proves another game that it would be possible to develop a learning program to play. They specified that the user interface should be simple, as they will be the only user, and the results should be written up and explained to them. They would like to have basic control over the program. This includes choosing the amount of training games and choosing whether to reset and retrain the learn data.

I looked into different games that have been learnt and then at those yet to be learnt. After some initial research, I decided on battleships being my game of choice. It has two distinct players and has two quite basic elements to the game, placing your ships and shooting your opponent's ships. I felt this game be a nice structure to go off, show how I can optimise these processes, as well as analyse and play so many games that it will come up with a near optimal solution.

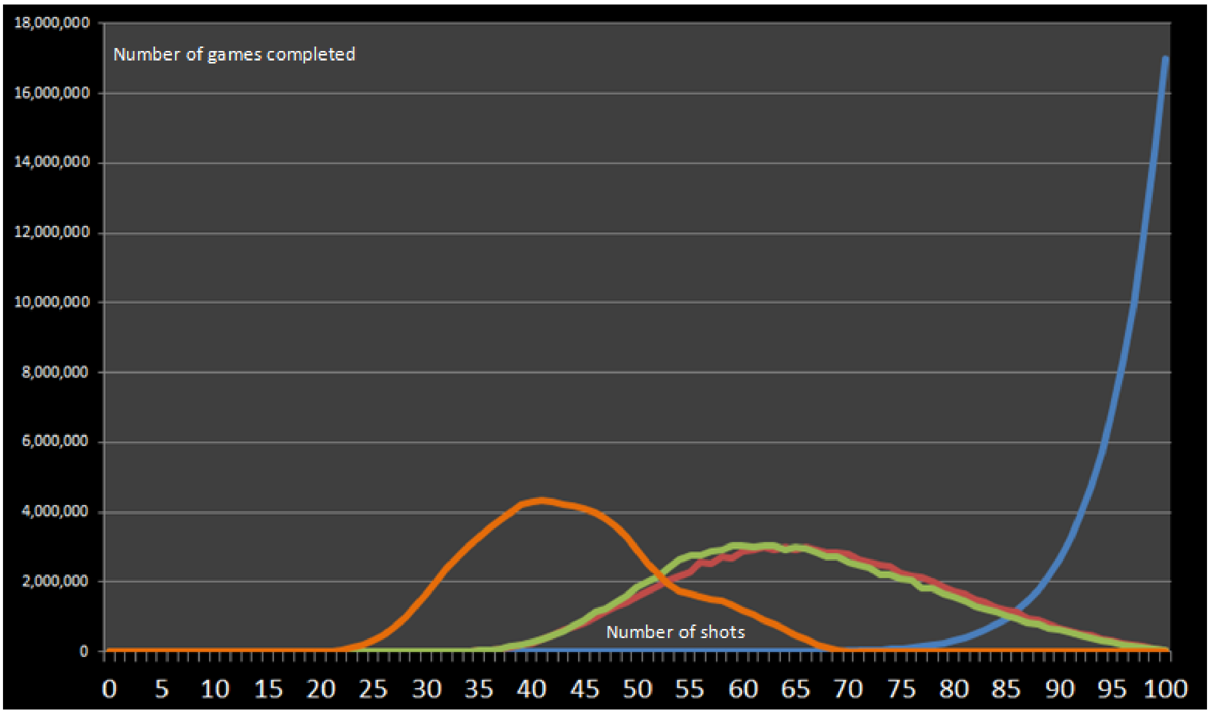
A solution widely regarded as the optimal was created by Nick Berry. My original idea was to use more simple probability algorithms, the method he used, to find the optimal solution. I decided to use machine learning, because I can then test the quality of my program against his solution. His solution is clearly explained on a website he runs called Data Genetics and it is competed with step-by-step examples and explanations, along with data backing up his claims. He compared different targeting strategies. These are the following:

- Random firing – randomly selecting squares to shot, hopefully hitting ships in the process
- Hunt/target – firing randomly until you hit a ship then firing at adjacent squares, to land another hit, then continue firing on an axis until you miss or the ship is sunk.

- Hunt/target with parity – also known as checker board, this means only firing at adjacent spaces once a ship has been found. This is because the minimum ship size is 2, so you theoretically only need to target half as many spaces.
- Probability density functions – to work out the probability of a ship being in a square and fire at the most likely places. This is done like the parity technique but takes into account the sizes of the ships left as well as places they can actually be placed.



(Berry, 2011)



(Berry, 2011)

My end goal will be to create a program that can learn to play the game battle ships and to have it improve its strategy and win more games over time. I will use a similar method to test my end algorithm, games completed over shorts taken to win. I can then use the results from my algorithm and compare them to the various methods that he tested to work out how effective it is.

Initial Research

The initial research covered what battleships is and how it plays. This will help design the game as best I can, as well as give be a structure from which the learning algorithms can fit into.

Battle ships is a relatively simple game to play, but probability calculations and strategic play can be done to greatly increase your probability of winning. There have been other variations of the game that use different sized grids, but I will be ignoring those for now. The aim of the game is to sink all of your opponent's battle ships. Each player has 5 ships, an aircraft carrier (5 grid squares long), a battle ship (4 squares long), a frigate and a submarine (both 3 squares) and lastly a patrol boat (2 squares long). Sometimes different names are used but the sizes should stay the same. The ships get placed before the game, and can be put horizontally or vertically in any place as long as they are sharing a grid or have parts of the grid. Taking it in turns players then fire to try and hit the opposing ships, which they will be hidden from them. If a ship is hit then the player must say the shot hit, but not what ship was hit. Once a ship is sunk the player must declare a ship is sunk, but not which ship it was. The player who sinks their opponent's ships first wins.

In order for it to teach its self and play depending on play styles I have researched different ways of doing this. Two key ways stood out to me: neural networks and Monte Carlo trees. Neural networks are what the majority of modern leaning programs use. Monte Carlo trees were a break though in how to process hundreds of potential paths in seconds. Monte Carlo trees are what allowed Deep Blue to beat the world chess campion in 1997 and AlphaGo to beat the world Go champion in 2016, a significantly more complex game than chess. For battle ships however I don't think my game of battleships is complex enough to use Monte Carlo trees, because it does not require planning ahead and guessing what your opponent will do next. I feel that further, in depth, research is needed in this field, so I will conduct some after I have the frame work of the game in place.

The ultimate end goal will be for it to beat the vast majority of humans it plays against and give reasonable completion against the probability density strategy. It will hopefully achieve all of this by it teaching its self and learning from games, rather than me making the mathematically best strategy and I think this can be achieved if I do further research at a later date, going in depth into how I can implement a learning program.

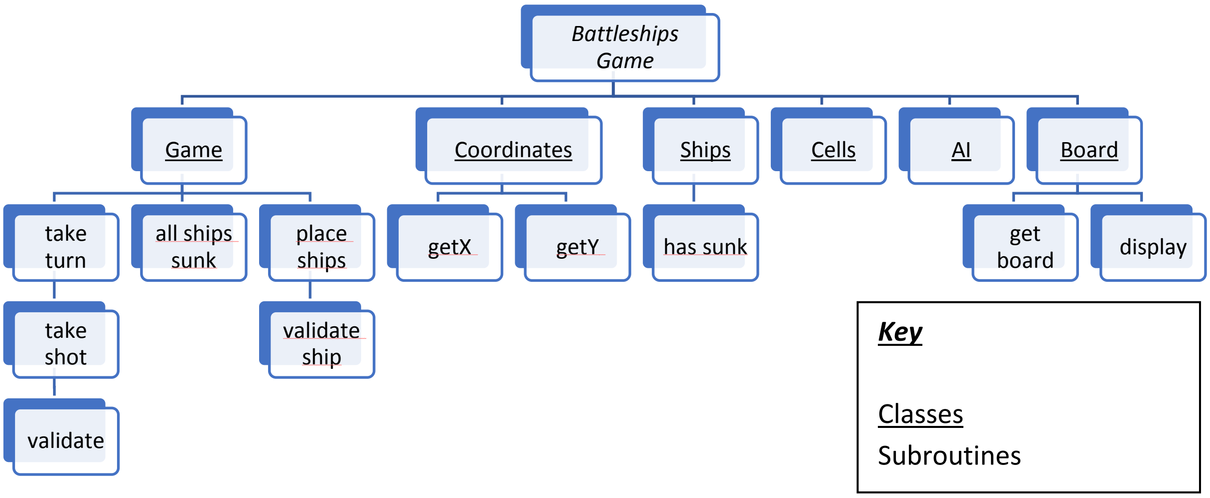
Initial Development

From analysing the research, I have come up with a specification for myself. I will then write the code to match this specification. I started all my planning processes from a high level then brought it closer to a low level, making it easier to work with, making sure I don't miss anything. At this stage in the process the focus is in creating the game of battle ships, from there I can then explore the learning algorithm possibilities that will work with the game.

My personal specification is to create a system that satisfies all the following points:

- Runs a game of battleships
- Has a learning element to the code that:
  - chooses a place to shoot
  - chooses where to put the ship
  - can improve on its first strategy to increase its win percentage
  - Can teach itself to play my Battleships game by learning from playing against opponents
- Can be replayed thousands of times without error
- Stores the intelligence that its learnt
- Could learn to beat a human player

If I meet all these criteria, then I will have been successful in my investigation. To make the program user friendly I will take necessary precautions to prevent input errors and runtime errors, making it very difficult to crash the code. As good programming practise, I will be writing it as an object orientated program, which should make it easier to edit and change things as the task progresses. The hierarchy charts below are how the classes and subroutines will be set up, however there is a high chance this could change when I begin implementing the code.



I plan to have to have the learning algorithm to not only learn how to shoot and sink ships, but also to learn patterns in haw its opponent plays. This is one problem that cannot be calculated from a fixed algorithm, but instead requires a learning algorithm to work out and predict the likely places and patterns to where they like to place their ships. From my initial research, I know that that neural networks can be used for this kind of problem, but I need to research further into how they work.

If I am going to make a program that can play the game properly then it will need to place ships as well. There are two main strategies when it comes to ship placement and there is so far, no definite answer as to what is the best. They are placing them touching or not. Arguments for touching is that they are less likely to hit any if they stick to one side of the board also it can trick your opponent if they think they sunk a large ship, when they really just sunk a smaller one and hit some others. Points against the touching strategy if that they have found all your ships at once which can be catastrophic gameplay wise.

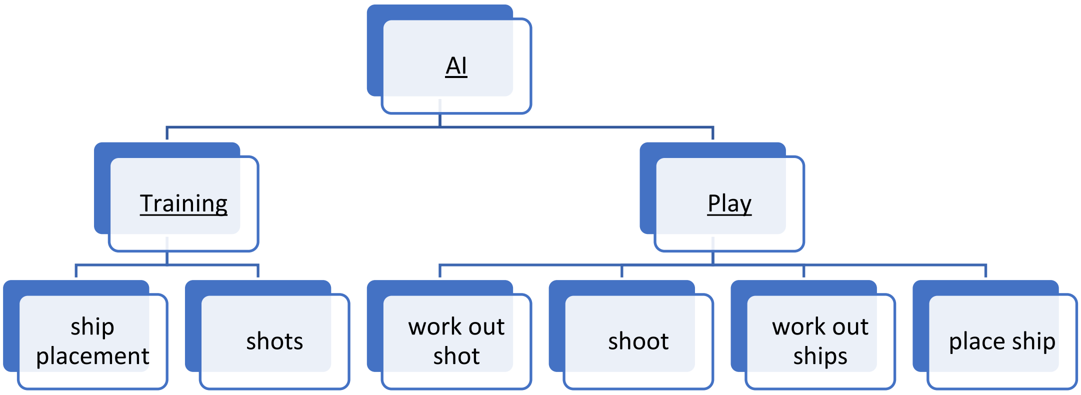
I will try and get the program to learn traits of play that leads to ships being placed in certain areas and orientations more often than in others. This is something an algorithm can work out, but can only be perfected through games with real people. It will be things like this that could allow it to be more successful than the probability algorithm alone, because it can add in preferred orientation and places to the probability mix.

To make the program I will need to firstly make a battleships game for the program to play. This will need to be able to support computer vs computer, as well as computer vs human. The computer vs computer should allow it to play millions of games to teach its self how to win most games. Next it will move to playing people to get it to learn habits and play styles so it can adjust its strategy accordingly.

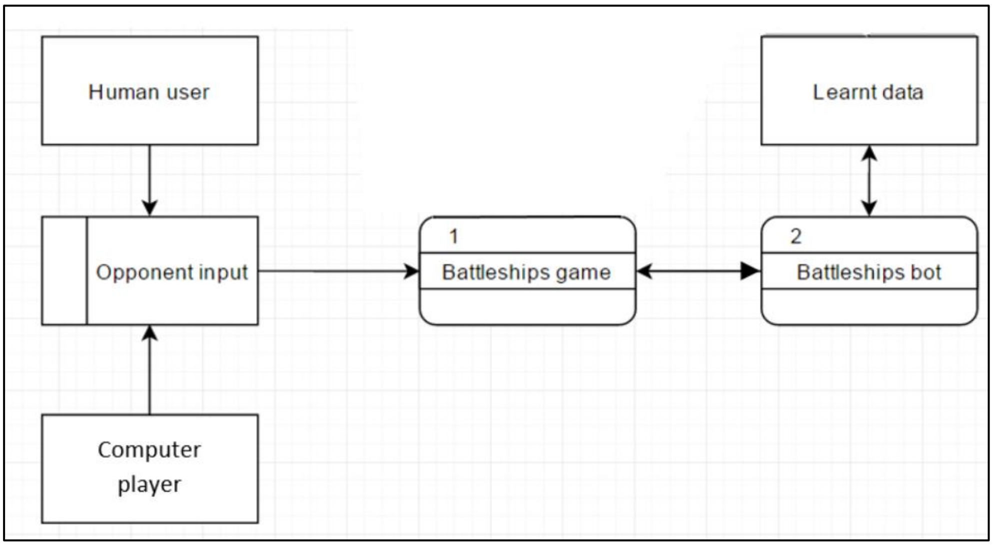
I will need to have a file to store the learnt data so it doesn’t have to be retrained after each run through. I will probably be storing the intelligence file as a comma-separated-values (CSV) file, because the structure will be useful, but if other formats are better suited then I will use them.

While most learning programs use C, Java and python, I will be writing the program in Python because it is a language that I am familiar with and it has features well suited to writing this sort of program. While python is very useful for writing learning programs, it is not eh best for writing complex object orientated programs, due to the way the classes work and are structured.

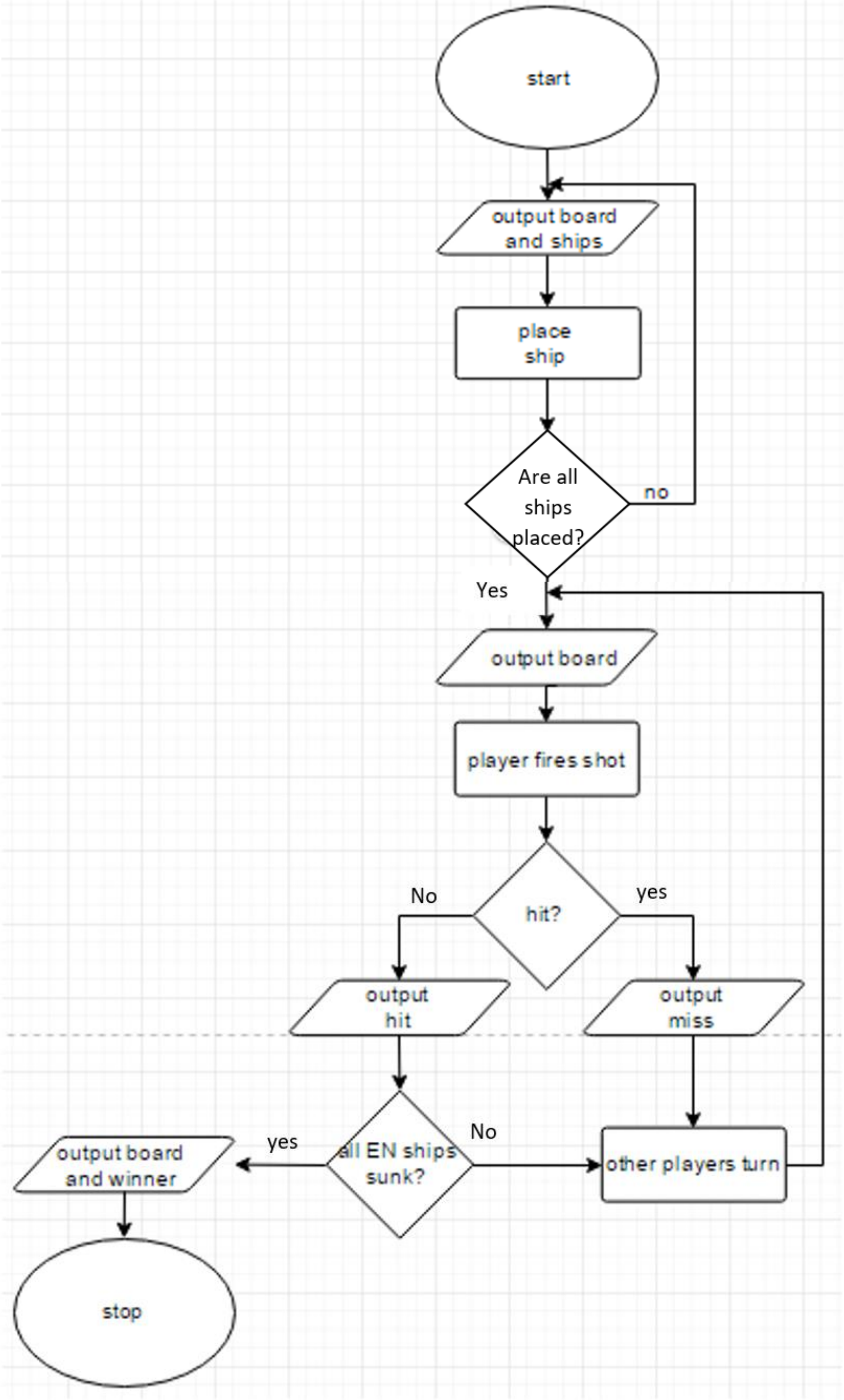
I have had a few ideas as to how to make the learning program work, so I have made a small guideline. At this stage I have not fully research the learning part of the program but I plan to build it to this structure:



This is a data flow diagram for the complete solution. This explains how the data will be moved, passed and stored throughout the completed system. It is just there to act as a rough representation as to how the data will be passed around the program. It is a model and should not be taken literally as it is all in the same program. The “Battleships bot” represents the learning side, the “Battleships game” represent the whole game, the “Learnt data” is the external file and the “Opponent input” is to represent the function used to take the commands from either a human or an automated algorithm.



I made a flow chart to show the basic running of the game of battleships, which will be the part I will create first, before researching further. Once I have the game, with parts ready for the implementation of the learning side, then I will do further research on the machine learning approaches.


start
output board and ships
place ship
Are all ships placed?
no
Yes
output board
player fires shot
hit?
No
yes
output hit
output miss
all EN ships sunk?
yes
No
output board and winner
other players turn
stop

To prove the board has been created properly it will need to be displayed. However, I would not want it to display on ever run if the program is running for thousands of games. I have made the following pseudo code to show how I plan on displaying the board.

```
i <= board_width - 1
FOR x IN board_width:
        PRINT i,
        FOR y IN board_height:
                PRINT board_grid[y][i],
        i <= i - 1
        PRINT
        END FOR
END FOR
PRINT " " WITH NO RETURN
FOR j IN board_height:
        PRINT j WITH NO RETURN
END FOR
```

This displays function of the code prints the board to console in a way that is easy for the user to understand, and it contains all the aspects that you would normally see on a board for battleships. The use of the variable “i” means the numbers displayed downs the side of the board will be the right way around to comply with the standard layout of battleships.

Once I have the base game completed then I will also have a structure to work off, so I know what the tasks the learning algorithm will have to do, and I can start to work on the protocols for that to work. I will treat the creation of the game and the implementation of the machine learning algorithm as two separate sections of the project, to keep my mind focused.

I have made IPSO charts for the AI and the game. This helps me to understand what input is need, what outputs to expect and the processes and storage requirements needed to get these. This chart is very useful when I come to designing the program, because it gives a rough outline as top what is needed. This stops me from missing key features of the code and gives me a base guideline as to how it will work.

IPSO chart for battle ships game:

INPUTS • Who is playing(string) • Ship placement locations [X and Y] (integer) • Shot coordinates [X and Y] (integer)	OUTPUTS • Boards with details • Text commands
PROCESS • Menu options • Make board • Placing ships • Firing • Hit or miss?	STORAGE • Data from the game • Data learnt by the AI

• Has ship sunk? • Has game ended? • Turn swap	

IPSO chart for machine learning section:

INPUTS	OUTPUTS
• Board(array) • Game text commands(string)	• Ship placement locations [X and Y] (integer) • Shot coordinates [X and Y] (integer)
PROCESSES • Adapt to inputs • Decide on outputs	STORAGE • Data learnt from game

I have made a list of requirements. I will try to write a program that meets all of these requirements. If it does then it will be successful, if it meets the majority of the requirements, then there a chance it will still complete the investigation, but will not be able to do so reliably in a user-friendly manner.

These are the initial requirements:

1. To have a game of battleships
   1.1. To have the game play to the standard rules of battle ships
   1.2. To have players that can be human and computer controlled
      1.2.1. To have player inputs
      1.2.2. To have learning program controls
   1.3. To have it be robust and efficient so it
      1.3.1. Doesn't crash after incorrect input
      1.3.2. Can run thousands of times without crashing
      1.3.3. Uses minimal amount of storage and memory
      1.3.4. Can simulate thousands of games in a reasonable amount of time
   1.4. To have it be capable of storing learning data
      1.4.1. Have it stored in a way that easy to read
      1.4.2. Have it stored in a way that easy to write
   1.5. To have a learning program to play battleships
      1.5.1. Be able to produce effective inputs in the game
      1.5.2. Have it learn to:
         1.5.2.1. Play the game
         1.5.2.2. How to become better at the game by:
            1.5.2.2.1. Increasing its hit percentage
            1.5.2.2.2. Placing ships in more tactical places

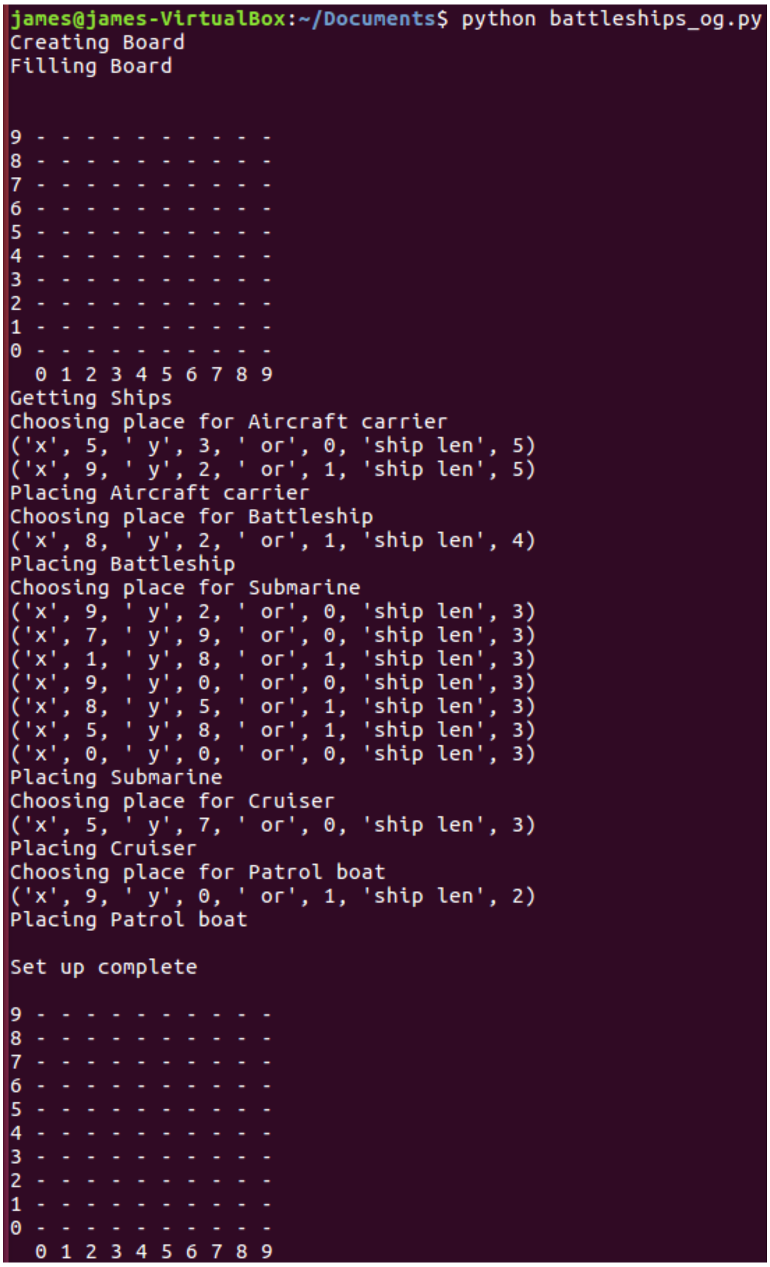
These are the set of requirements that I started writing the game for, and later used for my further research. These have been revised into my final requirements in my Further Research.

Creating Battleships

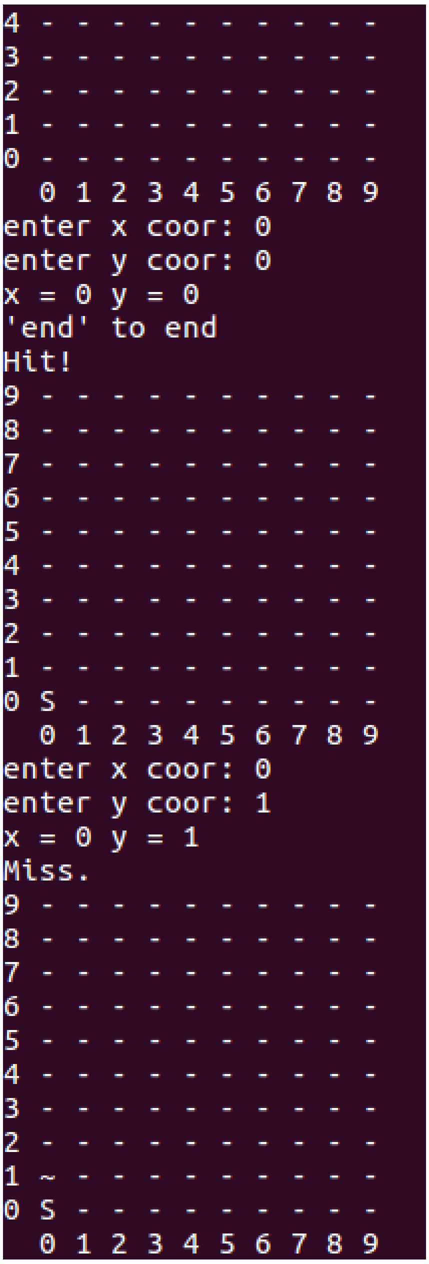
I began to implement some of my code for the battleships game. I was implementing the code while researching in to learning algorithms to try and be time efficient. I go the frame work of the game in place while researching into the feasibility the learning program. Very early on I realised that is should focus on just one element for the learning program, or I would have to make two as the parts to learn are so different. So, from near the start I made the game one player; with the learning program being the shooter, randomly placing ships to represent the other character.

Appendix 2 is my code of the working game of battleships. I built the game first then, then went on to decide what features needed to be changed. Some features got changed by removing them, some had a few variables changes so they worked with different attempts at learning programs, and others were simplified or removed.

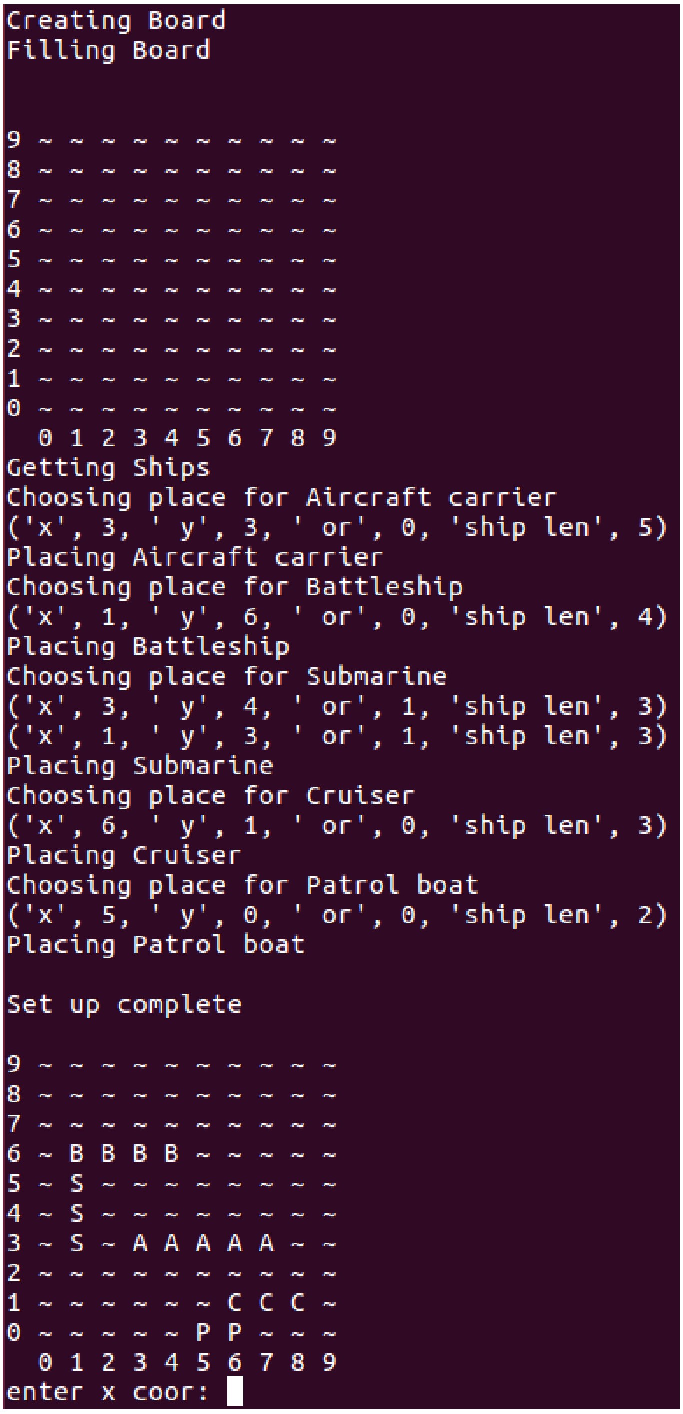
The screen shots below show how the game plays, and how the board is set up.



```
james@james-VirtualBox:~/Documents$ python battleships_og.py
Creating Board
Filling Board


9 - - - - - - - - - -
8 - - - - - - - - - -
7 - - - - - - - - - -
6 - - - - - - - - - -
5 - - - - - - - - - -
4 - - - - - - - - - -
3 - - - - - - - - - -
2 - - - - - - - - - -
1 - - - - - - - - - -
0 - - - - - - - - - -
  0 1 2 3 4 5 6 7 8 9
Getting Ships
Choosing place for Aircraft carrier
('x', 5, ' y', 3, ' or', 0, 'ship len', 5)
('x', 9, ' y', 2, ' or', 1, 'ship len', 5)
Placing Aircraft carrier
Choosing place for Battleship
('x', 8, ' y', 2, ' or', 1, 'ship len', 4)
Placing Battleship
Choosing place for Submarine
('x', 9, ' y', 2, ' or', 0, 'ship len', 3)
('x', 7, ' y', 9, ' or', 0, 'ship len', 3)
('x', 1, ' y', 8, ' or', 1, 'ship len', 3)
('x', 9, ' y', 0, ' or', 0, 'ship len', 3)
('x', 8, ' y', 5, ' or', 1, 'ship len', 3)
('x', 5, ' y', 8, ' or', 1, 'ship len', 3)
('x', 0, ' y', 0, ' or', 0, 'ship len', 3)
Placing Submarine
Choosing place for Cruiser
('x', 5, ' y', 7, ' or', 0, 'ship len', 3)
Placing Cruiser
Choosing place for Patrol boat
('x', 9, ' y', 0, ' or', 1, 'ship len', 2)
Placing Patrol boat

Set up complete

9 - - - - - - - - - -
8 - - - - - - - - - -
7 - - - - - - - - - -
6 - - - - - - - - - -
5 - - - - - - - - - -
4 - - - - - - - - - -
3 - - - - - - - - - -
2 - - - - - - - - - -
1 - - - - - - - - - -
0 - - - - - - - - - -
  0 1 2 3 4 5 6 7 8 9
```



```
4 - - - - - - - - - -
3 - - - - - - - - - -
2 - - - - - - - - - -
1 - - - - - - - - - -
0 - - - - - - - - - -
  0 1 2 3 4 5 6 7 8 9
enter x coor: 0
enter y coor: 0
x = 0 y = 0
'end' to end
Hit!
9 - - - - - - - - - -
8 - - - - - - - - - -
7 - - - - - - - - - -
6 - - - - - - - - - -
5 - - - - - - - - - -
4 - - - - - - - - - -
3 - - - - - - - - - -
2 - - - - - - - - - -
1 - - - - - - - - - -
0 S - - - - - - - - -
  0 1 2 3 4 5 6 7 8 9
enter x coor: 0
enter y coor: 1
x = 0 y = 1
Miss.
9 - - - - - - - - - -
8 - - - - - - - - - -
7 - - - - - - - - - -
6 - - - - - - - - - -
5 - - - - - - - - - -
4 - - - - - - - - - -
3 - - - - - - - - - -
2 - - - - - - - - - -
1 ~ - - - - - - - - -
0 S - - - - - - - - -
  0 1 2 3 4 5 6 7 8 9
```



```
Creating Board
Filling Board


9 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
7 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
6 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
3 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
1 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
0 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  0 1 2 3 4 5 6 7 8 9
Getting Ships
Choosing place for Aircraft carrier
('x', 3, ' y', 3, ' or', 0, 'ship len', 5)
Placing Aircraft carrier
Choosing place for Battleship
('x', 1, ' y', 6, ' or', 0, 'ship len', 4)
Placing Battleship
Choosing place for Submarine
('x', 3, ' y', 4, ' or', 1, 'ship len', 3)
('x', 1, ' y', 3, ' or', 1, 'ship len', 3)
Placing Submarine
Choosing place for Cruiser
('x', 6, ' y', 1, ' or', 0, 'ship len', 3)
Placing Cruiser
Choosing place for Patrol boat
('x', 5, ' y', 0, ' or', 0, 'ship len', 2)
Placing Patrol boat

Set up complete

9 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
7 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
6 ~ B B B B ~ ~ ~ ~ ~
5 ~ S ~ ~ ~ ~ ~ ~ ~ ~
4 ~ S ~ ~ ~ ~ ~ ~ ~ ~
3 ~ S ~ A A A A A ~ ~
2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
1 ~ ~ ~ ~ ~ ~ C C C ~
0 ~ ~ ~ ~ ~ P P ~ ~ ~
  0 1 2 3 4 5 6 7 8 9
enter x coor:
```

The screen shot to the left is from when the code was edited so that none of the cells stared off being hidden, which allowed me to see that the ships had been placed correctly.

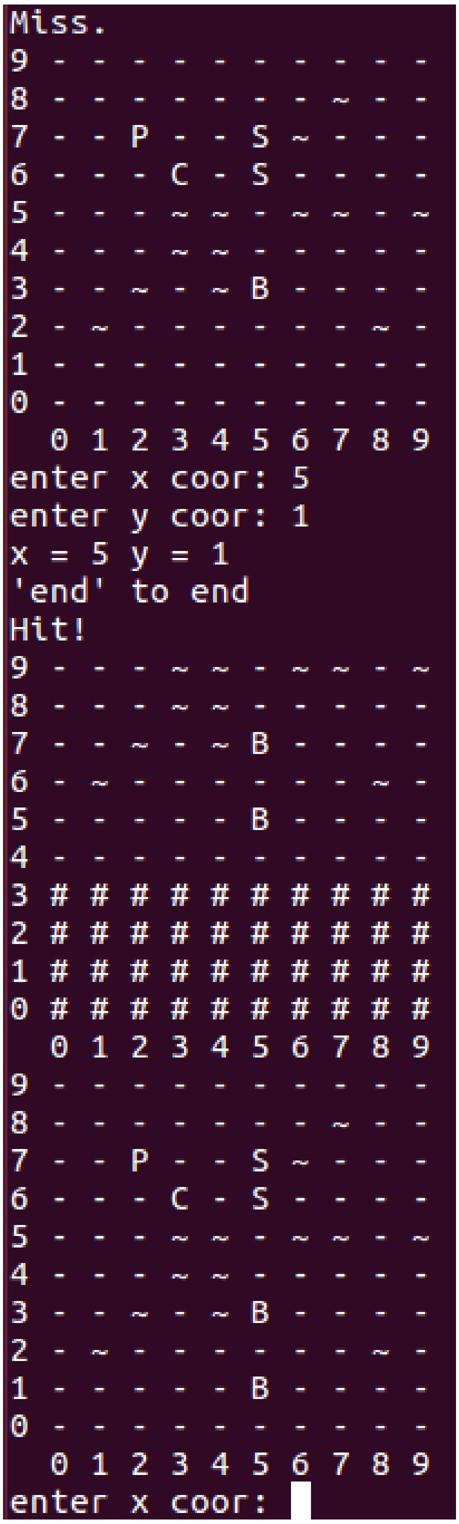
It its self was created from two different pieces of code I began writing, both got too messy, so I cut them down and used so of the useful parts in this, with the rest of this being written from scratch to fill the gaps.

A full play through pf the game can be seen in the video of Appendix 4. There you can see how the program complies with every battleships rule, apart from the only one player taking shots. The displaying of the ship locations at the start was to make it easier for me to error check and so I could cheat. Cheating had no effect on the program, it just made it so I was faster to win. The input validation can be seen at times 1:55 and 2:13 in the Appendix 4 video.

I created a feature, that allowed me to quit a game once I it a ship. This was mainly for testing so I don't have to finish the game each time I want to test a part, as I have the option to leave after every hit.

This game stood as the foundation for the rest of my programming. To ensure it could be easily adapted I started writing it in an object orientated format, and made sure it was very efficient.

When I finished writing the game, I changed parts depending on what I thought was feasible or not. I tried to leave as much of the original code in as I could so that the frame work for the program to be expanded back into the full game was there. For example, I already had a means of placing ships in every space on the board.



```
Miss.
9 - - - - - - - - - -
8 - - - - - - - ~ - -
7 - - P - - S ~ - - -
6 - - - C - S - - - -
5 - - - ~ ~ - ~ ~ - ~
4 - - - ~ ~ - - - - -
3 - - ~ - ~ B - - - -
2 - ~ - - - - - - ~ -
1 - - - - - - - - - -
0 - - - - - - - - - -
  0 1 2 3 4 5 6 7 8 9
enter x coor: 5
enter y coor: 1
x = 5 y = 1
'end' to end
Hit!
9 - - - ~ ~ - ~ ~ - ~
8 - - - ~ ~ - - - - -
7 - - ~ - ~ B - - - -
6 - ~ - - - - - - ~ -
5 - - - - - B - - - -
4 - - - - - - - - - -
3 # # # # # # # # # #
2 # # # # # # # # # #
1 # # # # # # # # # #
0 # # # # # # # # # #
  0 1 2 3 4 5 6 7 8 9
9 - - - - - - - - - -
8 - - - - - - - ~ - -
7 - - P - - S ~ - - -
6 - - - C - S - - - -
5 - - - ~ ~ - ~ ~ - ~
4 - - - ~ ~ - - - - -
3 - - ~ - ~ B - - - -
2 - ~ - - - - - - ~ -
1 - - - - - B - - - -
0 - - - - - - - - - -
  0 1 2 3 4 5 6 7 8 9
enter x coor: 
```

I also began to experiment with centring the board so the AI would only have to learn how to sink a ship from around the centre point. This would make it so much easier, because anything that is not a direct horizontal or vertical translation from that centre point should have no change of containing the rest of that ship.

The centering of the ship worked really well, but more wotrk had to be done for it to be ready to use, and the idea was still in concept.

I had plans to shrink the board to a smaller size for testing purposes, so when I did shrink the board down I left all the parts that would be needed if the program was to be expanded back in to a full size game.

I used this code as the basis for most of my experimental learning techniques. First I out copy this into a new file, then I would start making changes.

After changing the code so that I could centre the ships, I left this file, so the original working game could be used to foundation all my other ideas. I then began to focus all my work on further research of learning algorithms.

I used this code as the basis for most of my experimental learning techniques. First I out copy this into a new file, then I would start making changes.

This first program acted as the skeleton for my final program, where I could then implement and expand the learning part of the program on top

Further Research

I researched into learning programs and similar pieces of code, to let me compare techniques and types. I analysed different pieces of code to extract the relevant parts to teach me how to do certain parts. While having a lot of interest in AI and learning algorithms I had no prior experience in programming one before this project.

I used various online tutorial to teach myself the theory of how they worked before I started writing my own. At first I was surprised by the wide range of approaches to machine learning. I came across “Supervised”, “Unsupervised”,” Semi-supervised”, “Reinforcement” and “Deep” learning (Wikipedia, 2016). On the same website, some of these were then broken down further into subcategories. In total, there were 62 methods listed. I realised that I had to reduce the amount of information that I would have to read through.

After visiting other sources, I broke the types into three categories that are widely regarded as the main types:

- Supervised
- Unsupervised
- Reinforcement

I then taught myself the basic concept of each types and what they often are used for, so I could then make an informed decision on what path to look down further.

Supervised is when there are example inputs and desired outputs, the program then has to link the inputs with the outputs. Unsupervised being when it is not given input labels and is required to see how far towards an end goal it can get. Finally, Reinforcement learning being when it has a changing environment, and isn’t told how close to the goal it got, just whether it achieved it or not.

I saw that one example given of Reinforcement learning was “learning to play a game” (Wikipedia, 2016), which indicated that it was the right type of learning program for my investigation. Furthermore, a key difference between reinforcement and unsupervised learning is that example answers are not given, and small problems in the learning are not automatically corrected, rather it must work them out. Also in the game of battleships the ship placement is changing every game, meaning that it is a changing environment, something that is often overcome with reinforcement learning.

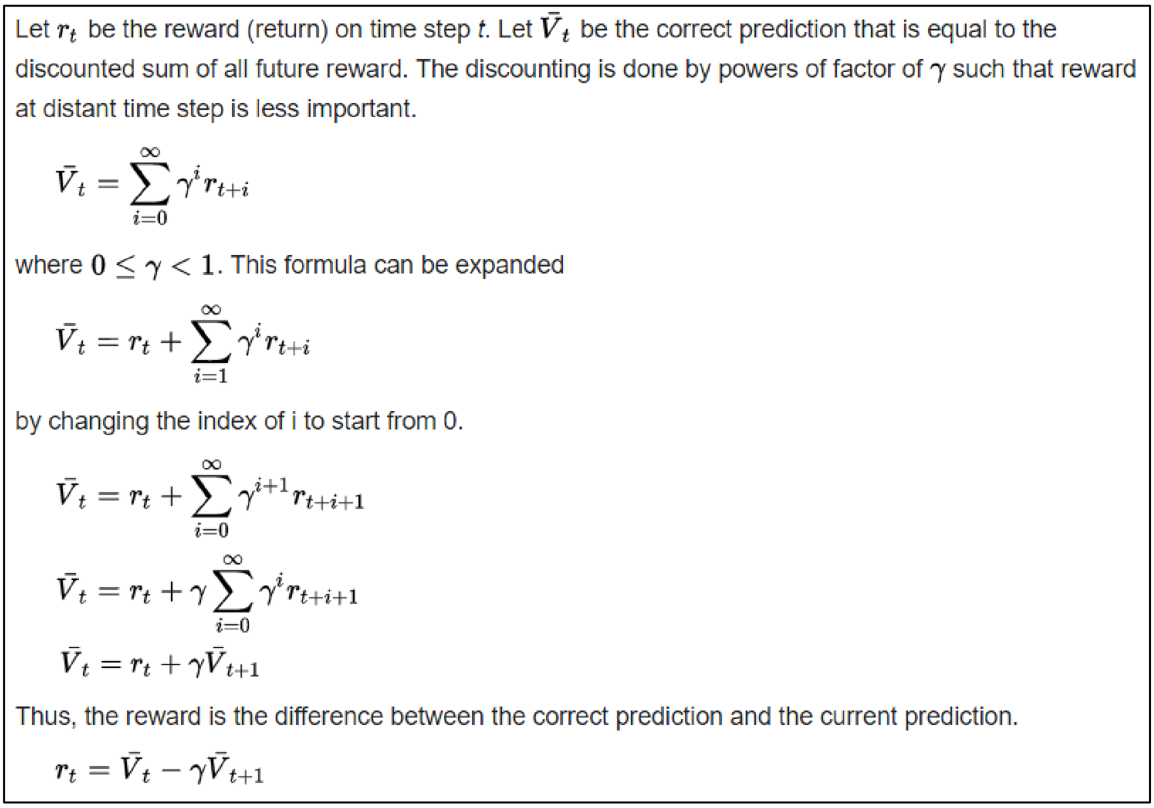
While visiting Leeds University I talked to Dr Mehmet Dogar about what machine learning approach I should use, supervised, unsupervised or reinforcement learning. He agreed that Reinforcement learning was probably the best choice for my code but we had limited time and couldn’t go into depth.

I found four methods of reinforcement learning from the Wikipedia, List of machine learning concepts, web page. These were:

- Temporal difference learning
- Q-learning
- Learning Automata
- State-Action-Reward-State-Action (SARSA)

Temporal Difference Learning

The Temporal Difference (TD) method involves looking at the reactions caused by its action and weighing that action accordingly. If the action creates a positive reaction, then the weight of the said action gets increased by a value. The weights of all the values are increased by amounts that reflect on the action and the previous actions. It can be formulated in the following way:



Let r_t be the reward (return) on time step t. Let $\bar{V}_t$ be the correct prediction that is equal to the discounted sum of all future reward. The discounting is done by powers of factor of γ such that reward at distant time step is less important.

$$\bar{V}_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

where $0 \leq \gamma < 1$. This formula can be expanded

$$\bar{V}_t = r_t + \sum_{i=1}^{\infty} \gamma^i r_{t+i}$$

by changing the index of i to start from 0.

$$\bar{V}_t = r_t + \sum_{i=0}^{\infty} \gamma^{i+1} r_{t+i+1}$$

$$\bar{V}_t = r_t + \gamma \sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$$

$$\bar{V}_t = r_t + \gamma \bar{V}_{t+1}$$

Thus, the reward is the difference between the correct prediction and the current prediction.

$$r_t = \bar{V}_t - \gamma \bar{V}_{t+1}$$

(Wikipedia, 2016)

I used skills that I learn in my maths classes to help me make sense of these equations, coming to the conclusion that the reward value is the difference between the full values from complete the task and the significance of that action to completing that task. Eventually that the reward value is dependent on how important that action was in completing the overall given task.

TD-Lambda

Developed in the 1990s by Richard S. Sutton, the algorithm works by assigning the variable lambda between the values of 0 and 1, where 0 is used of a completely wrong action and 1 being used to greatly reward key events. This means that more events can be covered, and the reward system remains relative because all the values will remain between 0 and 1.

This algorithm's most famous application is probably in the program TD-Gammon, a program that learnt to play backgammon, and could beat the world's greatest players.

Overview

This approach defiantly seems like a plausible method. It has been used to learn games in the past and can be used to reward many different actions, which in a game of battle ships there are many. It could allow me to reward the program as it plays each game, meaning it will be "smarter" for the next game.

Q-learning

Q-learning is a method of learning that can be applied to many different programs and is designed to find the optimal path through a Markov decision process (MDP). Q-learning learns different actions then applies values to the actions, and builds larger actions from the previously learnt actions, based on the values that these actions have. It is useful for showing the effects of certain actions without the need for a model environment. Another pro for Q-learning is that it learns and can work out a lot of things so once given a task often it won't require adaptations.

Markov decision processes are mathematical representations of decision making problems, where the problems outcomes are partly random and partly controlled. They can be represented in finite state machines or as recursive algorithms.

Overview

Q-learning would be the perfect option for my code, because the game can be put into a MDP format. The placement of the ships and the shots are up to the program, but the placement of the opponent's shots and ships can be entirely random. After attempting to put the game into this format I quickly realised that it was quite a struggle and there is only a limited amount of information I can get from online articles and tutorials. I knew that this topic was well beyond my ability.

Despite it being the perfect option on a theoretical level, I could not use it because I did not fully understand all the equations and notations. It would have too hard to write an attempt at this method and it would take too long for me to teach myself all the necessary information to understand.

Learning automata

A machine learning technique that's been studied since the 1970s, this approach learns from its environment. It uses a Markov chain (method of predicting future outcomes from a collection of past outcomes) in an attempt to learn what its future outputs should be. Like Q learning, it is another way to process Markov decision processes, but this time it doesn't learn in such a dynamic way, rather it performs lots of actions then works out what to do next from that and tries again. It learns from the changes in its environment and can predict with great accuracy what could happen next, then they can change a factor in this process.

Overview

From what I understand, this technique will be of much use on the shooting side of the game because the inputs have to depend on a relatively unpredictable lay out and the data learnt would not be able to be passed on, because the opponents ships will be changing location each game. However, it could potentially be used in ship placement, as an attempt to learn where the opponent usually shoots and therefore predict where they will not shoot and place the ship there.

State-Action-Reward-State-Action (SARSA)

The name comes from the process of working out the Q value. The best explanation I found was: "Q-value depends on the current state of the agent '**S**$_1$', the action the agent chooses '**A**$_1$', the reward '**R**' the agent gets for choosing this action, the state '**S**$_2$' that the agent will now be in after taking that action, and finally the next action '**A**$_2$' the agent will choose in its new state. Taking every letter in the quintuple (s_t, a_t, r_t, s_{t+1}, a_{t+1}) yields the word *SARSA*" (Wikipedia, 2016)

The algorithm below shows how the state-action value is updated on an error and gets adjusted by the learning rate.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

(Sutton, 2012)

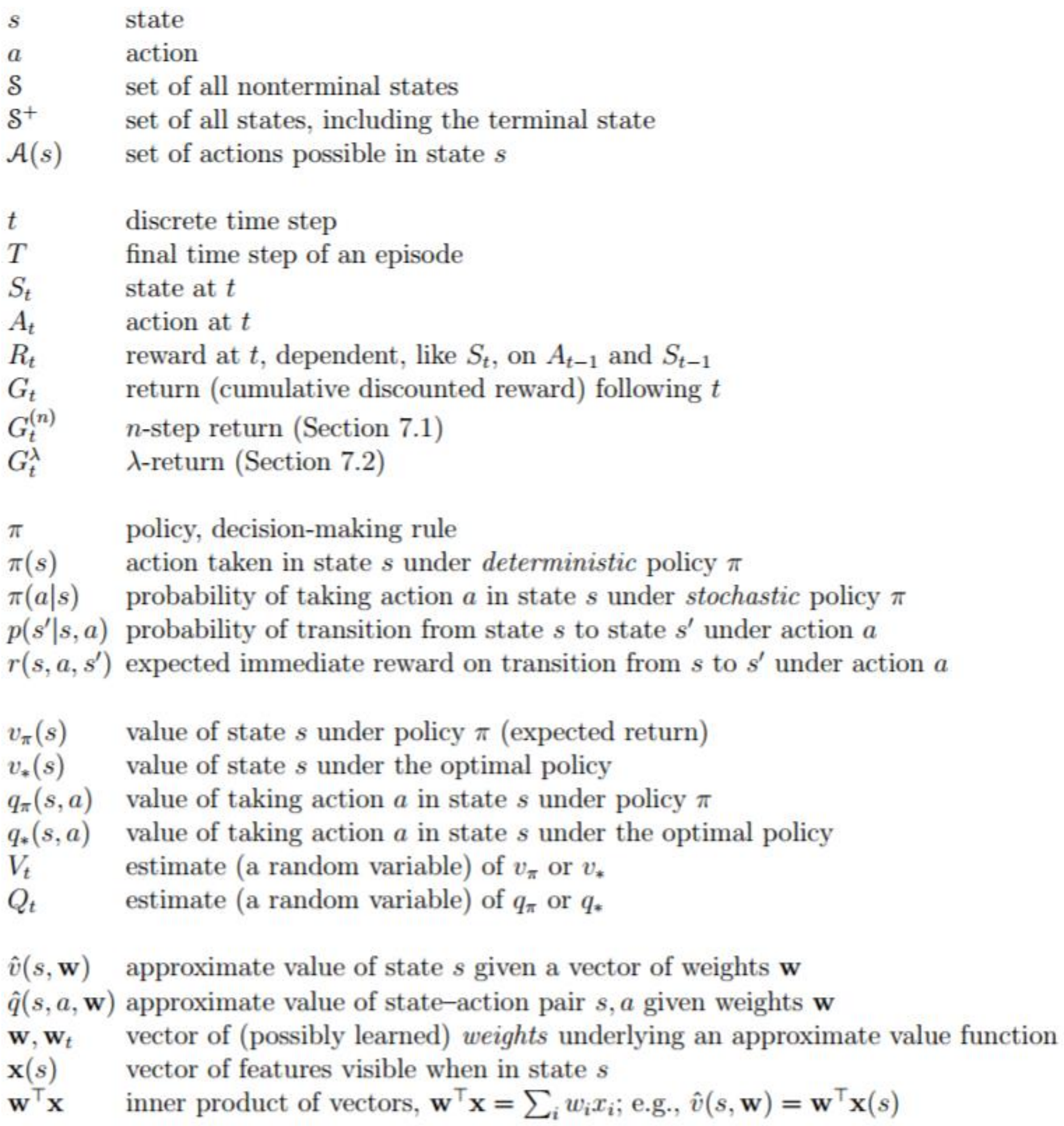
Overview

SARSA is very similar to Q-learning, the main difference being that Q-learning is given an algorithm that is set to learn and retrieve values, but SARSA has its base algorithm which it learns to adapt. In the way they run, there are only a few difference between Q-learning and SARSA, but I believe both are too complex for me to implement efficiently.

Conclusion

I found that Q-learning and SARSA would be the most effective methods to use, but they are far too complex for me to effectively implement. The other approaches too were rather complex. I understood them to a base level but I struggled to understand the notation used in the equations and therefore struggled to follow the algorithms.   I tried teaching myself some of the topics so that I would understand, but the task was too great and would have consumed too much of my time.

However, I did learn the standard character notations for each of the elements from a paper written by Richard S. Sutton. This not only helped me understand the algorithms I found, but also showed me what the key parts of the learning program were, so I knew what variables I would have to consider.



s	state
a	action
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{A}(s)$	set of actions possible in state s
t	discrete time step
T	final time step of an episode
S_t	state at t
A_t	action at t
R_t	reward at t, dependent, like S_t, on A_{t-1} and S_{t-1}
G_t	return (cumulative discounted reward) following t
$G_t^{(n)}$	n-step return (Section 7.1)
G_t^λ	λ-return (Section 7.2)
π	policy, decision-making rule
$\pi(s)$	action taken in state s under *deterministic* policy π
$\pi(a\|s)$	probability of taking action a in state s under *stochastic* policy π
$p(s'\|s,a)$	probability of transition from state s to state s' under action a
$r(s,a,s')$	expected immediate reward on transition from s to s' under action a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy
$q_\pi(s,a)$	value of taking action a in state s under policy π
$q_*(s,a)$	value of taking action a in state s under the optimal policy
V_t	estimate (a random variable) of v_π or v_*
Q_t	estimate (a random variable) of q_π or q_*
$\hat{v}(s,\mathbf{w})$	approximate value of state s given a vector of weights $\mathbf{w}$
$\hat{q}(s,a,\mathbf{w})$	approximate value of state–action pair s,a given weights $\mathbf{w}$
$\mathbf{w},\mathbf{w}_t$	vector of (possibly learned) *weights* underlying an approximate value function
$\mathbf{x}(s)$	vector of features visible when in state s
$\mathbf{w}^\top\mathbf{x}$	inner product of vectors, $\mathbf{w}^\top\mathbf{x} = \sum_i w_i x_i$; e.g., $\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top\mathbf{x}(s)$

(Sutton, 2012)

To effectively complete this project, I will either have to finds another method of creating the learning program, whether that be through finding another approach or by creating my own method.

I also decided that this task would be more difficult than I initially thought, so I shifted my focus to one part of the program, the shooting, and decided to abstract that problem further. I separated it out into potential ways to represent the problem, this then made me more open to different techniques I could use. After breaking down the shooting part of the program, I decided that getting a program to work out where to shoot next will be my main priority.

I broke the problem down as so:

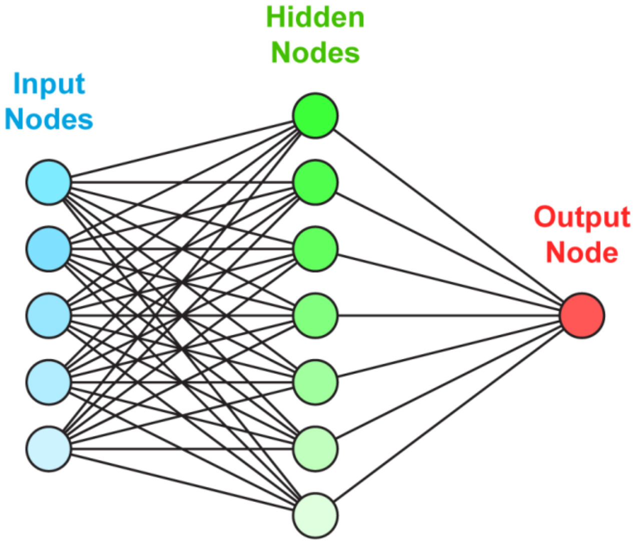
Neural Networks

I researched neural networks because they are a common feature in many learning programs. There are a wide range of online sources that cover this topic. The concept seemed confusing at first, but with the right tutorials and videos I quickly understood the concept.

They are often used for learning patterns and coming up with conclusions from a variety of inputs, which would just be too much data and possibilities for a person to comprehend. I felt this could be useful in my project because if I could get it to recognise the pattern of the ships, then it could predict where to shoot next.

By neural networks I refer to artificial neural networks (ANN), as to natural neural networks, which are a collection of interconnected brain cells. They have similar names because the structure of an ANN is loosely based on the structure of the brain. (Woodford, 2017)

Neural networks are very useful in learning programs because they don't have to be programmed to specifically learn, they are just given the rules and learn by themselves. The concept behind an ANN is that you make a single machine act as if it was millions of small interconnected cells/units. The units are arranged in layers and a theoretical network can have infinite layers, of infinite size. However in reality this is not necessary because just by adding just one new layer can greatly increase the computing capability of the network, because every unit in a layer connects to every unit in the next layer.

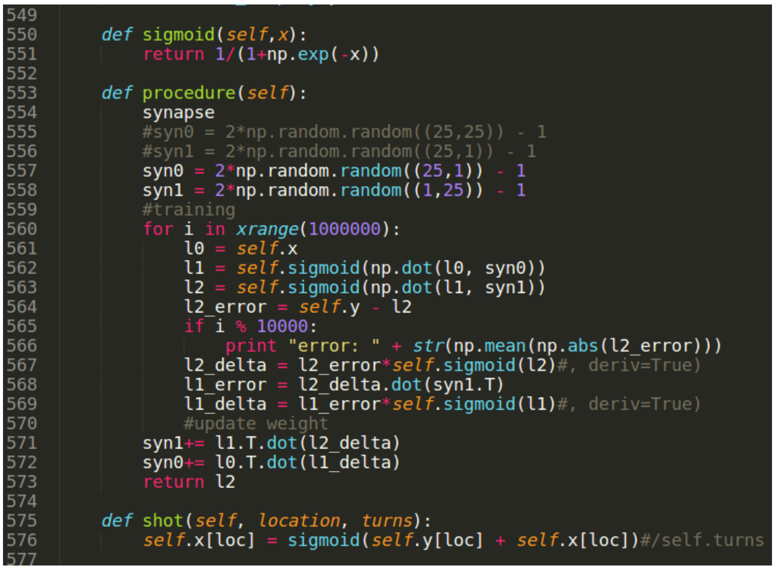


(Butler, 2015)

There are input layers and output layers, all the layers in-between are considered as hidden layers. The weights that connect each unit are what calculates and allows the network to "learn", and the quantity of cells and layers determine how many weights are added before the final solution is come to. It passes the values from the input units through all the hidden units, adding the weights as it goes, then once it reaches the output it can back track through the network to find the best fitting route.

I attempted to create my own ANN, which I managed successfully. I followed an online tutorial video: https://www.youtube.com/watch?v=h3l4qz76JhQ (Raval, 2016)

By creating my own, very basic, ANN I could then test to see if I could adapt it to work with my battle ships game. I tried to make it so it would recognise the ship as a pattern.



```
    def sigmoid(self,x):
        return 1/(1+np.exp(-x))

    def procedure(self):
        synapse
        #syn0 = 2*np.random.random((25,25)) - 1
        #syn1 = 2*np.random.random((25,1)) - 1
        syn0 = 2*np.random.random((25,1)) - 1
        syn1 = 2*np.random.random((1,25)) - 1
        #training
        for i in xrange(1000000):
            l0 = self.x
            l1 = self.sigmoid(np.dot(l0, syn0))
            l2 = self.sigmoid(np.dot(l1, syn1))
            l2_error = self.y - l2
            if i % 10000:
                print "error: " + str(np.mean(np.abs(l2_error)))
            l2_delta = l2_error*self.sigmoid(l2)#, deriv=True)
            l1_error = l2_delta.dot(syn1.T)
            l1_delta = l1_error*self.sigmoid(l1)#, deriv=True)
            #update weight
        syn1+= l1.T.dot(l2_delta)
        syn0+= l0.T.dot(l1_delta)
        return l2

    def shot(self, location, turns):
        self.x[loc] = sigmoid(self.y[loc] + self.x[loc])#/self.turns
```

This was the very basic neural network I created and tried to adapt, but it was never successfully adapted to work with the rest of my code. This was the final state it was left in, however there are unsaved versions that were more complete.

The problem with this is that the ship remains hidden, until shot. This was a great struggle and while trying to adapt the network, I realised that it would probably not work. There were many reasons that I realised it wouldn't work.

The main reason was that neural networks need sample data to train from (Spencer-Harper, 2015), which it couldn't have because of the hidden nature of all the cells. It means that I would have to try to recognise several different variations of the ship depending on what was showing and what was hit. This means either adding more layers or creating multiple networks. Another issue which comes from trying to get away with no training data, is that it will try to learn from a board that is constantly changing, meaning it will be very unlikely for there to be any discoverable patterns.

Overview

After realistically looking at the problems to overcome, and trying to come up with and later looking up ways to solve them, I concluded that I would not be able to use a neural network in an efficient or effective manner. This is mostly because of the lack of training data and the problems that I encounter from trying to overcome this, which come from the different states it would have to recognise the ship in and the hidden and changing nature of the cells.

For the neural network to work it would have to have lots of changes made to the structure and many elements would have to be custom made to the point where there is no point in using a neural network. Even then this is theoretical and I decided it was not worth the effort as there was an unlikely chance that I would get it to work.

Conclusion

After considering a variety of existing techniques, I have not found an existing approach that will fit my requirements. I have found algorithms that would be perfect for my program, but they have been too complex for me to understand and implement in the given time. I reviewed other methods, but they too were either not suited in the way that they work, or they are too complex for me to understand. From looking at the Q-learning, I started getting into deep learning, which is well beyond my level of computational competence.

I started to research more basic approaches, and found a few videos that described very basic learning programs. I was shown one source which dynamically works out if a shape is a triangle. It had a set of variables; blue, red, 3_corners, 4_corners, perimeter_greater_than_5, perimeter_less_than_5; to name a few. It then was given data that represented shapes and had to work out whether they were triangles. It worked by weighting each, which correlated to how much it considered that factor in its decision. After it had started running it worked out that 3 sides and 3 corners are the most important factors in deciding if a shape is a triangle. As a test the input data was changed to make all triangles red and the other shapes blue. As expected I worked out the key factors in a triangle are it having 3 sides, 3 corners and it being blue.

I felt like this sort of weighting and very basic structure would be best suited to my ability and complexity of the task. I also having researched further I will update my requirements to something more feasible and to my ability.

Requirements

These are the final, revised requirements:

2. To have a learning program that learns to play elements of battleships
   - 2.1. To have the elements of the game to follow the standard rules of battleships
     - 2.1.1. Ships are placed on a board
     - 2.1.2. The ships are then shot at
     - 2.1.3. When a square on the board is shot, it is either a hit, when it hits a ship, or a miss.
     - 2.1.4. If all the cells that make up a shit get hit then the ship sinks
     - 2.1.5. The player wins when all the opponent's ships are sunk
   - 2.2. For there to be one player controlled by the learning algorithm
     - 2.2.1. Have it calculate and fire shots at the ships
     - 2.2.2. Have it learn to become better at the game by, taking less turns to sink the ships as time goes on
     - 2.2.3. There to be no deterministic code to control firing
   - 2.3. To have randomly placed ships to represent the other player
     - 2.3.1. Full game places ships as per battleship rules
     - 2.3.2. Test case will use a single ship
   - 2.4. To have it be robust and efficient so it
     - 2.4.1. Doesn't crash after incorrect input
     - 2.4.2. Can run thousands of times without crashing
     - 2.4.3. Uses minimal amount of storage and memory
     - 2.4.4. Can simulate thousands of games in a reasonable amount of time
       - 2.4.4.1.
     - 2.4.5. Doesn't crash if file isn't accessible
   - 2.5. To have it be capable of storing learning data
     - 2.5.1. Have it stored in a way that easy to read
     - 2.5.2. Have it stored in a way that easy to write
     - 2.5.3. Have it set up and reset the file
   - 2.6. Must have a basic user interface that
     - 2.6.1. Allows the user to choose to retrain the learning program
     - 2.6.2. Allows the user to choose how many games to loop through
     - 2.6.3. Displays data that can prove that the program is learning

Explanation of requirements

These are the requirements that I will write the rest of my program to. My end goal is to have all these requirements met, which means I will have successfully completed my task.

I have set the requirements to what they are so that given more time the program can be expanded to cover the complete game of battleships. I have set it do that it only shoots the ships because it makes the game much simpler, because then I don't have to worry about it taking turns, and working out what the opponent shoots least to place the ships is a relatively easy task, as it would just be board representation that increases each cells value when that cell is targeted.

It could be made more complex by searching for the location where there will be the least cumulative shots for the place of the whole ship, but this will just involve some for loops to iterate through looking for that optimum point. I feel this would not add a lot to the game, as the board placement is not the most important feature; it only happens at the start of the game and would have minimal effect on the play of the overall game, especially against a random firing bot which is what it would most likely be trained against.

The sinking of the ships is the key part of the game, because that is how you win; smart ship placement could increase how long it takes to lose, but it is very much up to chance.

The overall goal was to create a program that learns to play certain aspects of the game battleships. This way I'm still investigating what games can be learnt by a learning program, just I'll be exploring elements of the game rather than the whole game itself. I have chosen to try and teach it the shooting part, because I would consider it the most skilful and useful part of the game. It is also the most complex part of the game. This is because it not only must find the ship on the hidden board, but once it has been hit the most effective way to paly would be to shoot around the hit cell in the board. This can be quiet easily coded in, but I imagine getting a program to learn that patterning would be more of a test. Also, there is a chance that it could find patterns that mean it will be more effective than a choosing a random place nearby to shoot.

To achieve this goal, I will have to meet sub criteria. These being the requirements 2.1 to 2.6.

Requirement 2.1 means that the parts of the game I do have must follow the standard rules of the battleships game. For example, the current design would not let there be two players taking on in turn to play because that is not being included, however the learning algorithm will shoot at the opponent's board, and it must work out where to shoot from what any normal player would see, it cannot see through the hidden layer and shoot where the ships are. Likewise, the ships that it must shoot will have to be the placed in a way that they comply with the rules. If elements of the code have different rules to the actual game, then I will not actually be testing whether the game is learnable by a machine, because it will be playing to different rules. All the sub requirements just state the rules of the game that it must abide by, to stay representative of the game battleships:

2.1.1. Ships are placed on a board
2.1.2. The ships are then shot
2.1.3. When a square on the board is shot, it is either a hit, when it hits a ship, or a miss.
2.1.4. If all the cells that make up a shit get hit then the ship sinks

2.1.5. The player wins when all the opponent's ships are sunk

2.2 is the learning algorithm being in my program. It has to be a learning algorithm and has to learn, or at least have a reasonable attempt, at trying to learn the game. This part of the program will be very complex and is required to do numerous tasks. I have made sub requirements of this requirement; 2.2.1, 2.2.2 and 2.2.3; to make sure that the learning algorithm does everything it is supposed to.

Requirement 2.2.1 is that the learning can work out where to shoot and shoot that location. This is a critical aspect of the game battleships as it is a way in which a player wins. The machine learning part of the program has to be making the decision on where to shoot or it has ultimately failed in learning to play the given element of the game it was given. If it can learn to take shots, then it will only progress further to play other parts of the game, like identifying ships, placing ships and spotting patterns in where its opponent places ships.

2.2.2 is a key part of the learning process. For the program to truly learn then it should improve at the game over time. There are two way to measure its ability to learn, the accuracy of its shooting and the amount of moves it takes to win. These are very similar and one will result in the other, but I feel that they will be two ways to measure and check the same thing. I made this separate from the main requirement, because it could be possible for the AI to learn to play, but not get any better than just firing. Theoretically the longer the program runs for the smarter it gets, so there should be a significant different in the time it takes to win from a game after 5000 training runs and a game after 100000 training runs, however I do not yet know the efficient the learning algorithm will be, it may learn over a shorter space of time, or it could take longer.

Requirement 2.2.3 means that no aspect of the learning program is designed in a way that tells it where to shoot, without it having to learn. It should not have any elements of the algorithms that tell it where to shoot, because then it is not entirely a learning algorithm, and its performance can't be relied on from the learning part alone.

Requirement 2.3 is to have ships placed randomly on a board to represent the other player. This is a necessary requirement as battle ships is normally a two-player game, but because the learning algorithm will only shoot, this will only place ships. This is critical to simulate a real game like scenario for the AI to learn from.

Requirement 2.4 is for the program to be robust and efficient. There are many factors for this to be asses, which I have put in their own sub requirements of this requirements, 2.4.1 to 2.4.5. For the code to be at a satisfactory level of efficiency and robustness, these requirements must be met. The code should be efficient so it doesn't take a long time to iterate through the games, and to keep good programming technique. The program should be robust so it doesn't crash and so it is user friendly for ease of use.

Requirement 2.4.1 is so the program does not bug or crash if an incorrect input was entered. There should not be much user inputs, but it would be a pain if the program crashed after an accidental input.

Requirement 2.4.2 means the code should run thousands of times without crashing, which it needs to do for the program to learn. It needs to manage thousands of runs in a row without crashing to give the AI an opportunity to learn.

2.4.3 is for the program to use a minimal amount of storage and memory while running. There are a few reasons this is important. Firstly, after the memory could fill up if all the data from thousands of iterations was stored in memory, especially as I will have to run the program on a virtual machine, so it has very limited memory. The same goes for storage, because reading and writing from storage can take time and the virtual machine has limited storage.

Requirement 2.4.4 means the program can run through thousands of games in a reasonable amount of time. This is so the user doesn't have to wait a long time for their results. This is good for time management and testing, because I don't want to waste time and have to wait a long time, just to find a bug and have to wait all over again.

2.4.5 means the program doesn't crash if it cannot open the file. This is important because it will be a waste of time and learnt data if the program crashed part way through a large training session, because the file cannot be opened. It would be best for the user to get the option to try again, to save the data, or at least a message to inform them of the error.

The requirement 2.5 is for the storage of the learnt data. This is important because saving to a file means the data will stay there then the program finishes and it means the saved data is non-volatile so it doesn't have be retrained. This data will also have to be easy to read and write. Easy meaning it can be accessed and read in a quick and efficient way. This makes up the sub requirements 2.5.1 and 2.5.2. Also, the file will have to be created and reset, as explained in 2.5.3

2.5.1 is for the learnt data to be stored in a way that is easy to read. This is important so the program doesn't have to decipher or search for the data, as the speed the program runs is very important due to the amount of potential iterations. It must also be easy for the read for the user, so they can analyse the data. The user will have to understand the data so they can see if the AI is behaving as it should and to check that it is learning the correct values.

Requirement 2.5.2 is for the data to be stored in a way that easy to write. It is important for the program to be able to write to the data file easily. It shouldn't have to take time because the program need to write lots of things to the file so it shouldn't be a slow process or it would take a long time to run thousands of games. Also, it should be a simple writing process so noting can go wrong, because if it does lots of data can be lost.

Requirement 2.5.3 is about the creation and resetting of the file. They are in the same category because resetting the file is the same as making a new file with the same name. The file must be created so that the file before learning is still ready to be read from, and ready to receive the learn data when it comes. Resetting the file is important because the learnt data may be incorrect and it must be tested multiple times.

The requirement 2.6 is for the user interface. This has to display a range of things to satisfy the client. It was specified to be simple, so I will not add any more features than what they ask for. The features that are needed are described in the sub requirements of requirement 2.6. they are import because without a working user interface the client will have no control over the program.

2.6.1 states that there has to be an option for retraining the program. This is essentially an option to set up the file again. This feature is important as it will save the client from having to manually reset the file. The reason the user must have this feature are explained in 2.5.3.

2.6.1 means there must be a means of allowing the user to choose how many games to loop through. This is important as this will allow the user to work out how long it takes for the program to have learnt to a sufficient level. Also, it will allow for testing to be completed in a faster time, as if one feature is needed to be tested then only one game is need not thousands.

The requirement 2.6.3 means that the user interface displays data that can prove that the program is learning. This is so that in the evaluation and analysis of the program, as well as the testing, one can ensure the program is learning. Without this then there is no evidence that the program is learning.

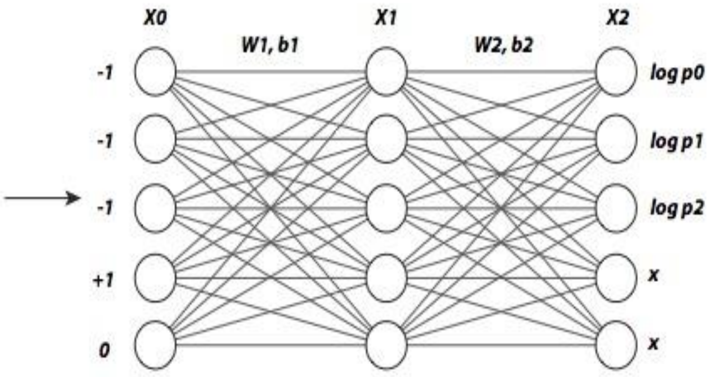
These are all the requirements that I will writing my program to meet and testing it against. At the end of the project, I will evaluate the program against these requirements. If it meets all the requirement s then it will satisfy the user.

It is key that I stick as best I can to these requirements and use them as a guideline, because I don’t want to start writing code that is not needed, and missing out features that are needed.

Designing of the final program

A lot of design went into creating the final program. Some of the ideas that I come up with were too complex to iterate through by hand, so many had to be implemented into a test file, to see if they were actually feasible. Some of the ideas were much more suited than others, however, I made sure I tried anything because machine learning was a topic I knew very little about, so I didn't want to miss a simple, but effective, idea.

One test that I mentioned in my further research was artificial neural networks (ANN). This would have been a very useful resource had I got it to work, but the task was beyond me. I found a resource source that was very helpful in my research.



Their ANN for a board that is 5 by 1 with a ship of length 3. They said the process "can be a challenge, and rewards function design is therefore something of an art form." (Landy, Deep reinforcement learning, battleship, 2016) Which proved to me the complexity of my task.

Jonathan Landy, the author of the article, describes himself on his web page: "I worked for eight years in theoretical physics, primarily statistical mechanics. This included two postdocs, one at UC Berkeley and one at UC Santa Barbara. These days, I work as a data scientist at Square in San Francisco." (Landy, Jonathan Landy, 2015) I felt this showed the difficulty of the task, and the sort of team that would be needed to solve it. It also shows how the topic is far beyond A-level standard.

The source showed that it was possible to use an ANN to learn the game battle ships, but in the article, it says how they encountered veracious problems with their program. The main part of their article was centred around many complicated equations that I, nor my teacher understood.

This was their final game function

$$r(a; t_0) = \sum_{t \geq t_0} \left(h(t) - \overline{h(t)} \right) (0.5)^{t-t0}$$

(Landy, Deep reinforcement learning, battleship, 2016)

I could make sense of some parts but not enough to fully understand their equations and what they tried to solve the issues they had. They also used lots of download packages: jupyter, tensorflow, numpy, and matplotlib. I only used numpy because I didn't not want to rely on the packages, because it meant that the technical parts are done for you; little programming is required.

My original idea was to simplify the problem was to convert the board into a one-dimensional format, but still I could not get the program to run as it should, so I didn't even get to work on any form of training.

The following quote explains my issue with the aid of a formula. It explains how the neural network requires a set of learning data, which cannot be provided. Therefore, the first runs have to have approximated values, and the learning data becomes the output of from the first runs.

"Here, the p(a) values are the action probability outputs of our network.

$$\begin{aligned}\partial_\theta \langle r(a|s)\rangle &\equiv \partial_\theta \int p(a|\theta,s)r(a|s)da \\ &= \int p(a|\theta,s)r(a|s)\partial_\theta \log(p(a|\theta,s))da \\ &\equiv \langle r(a|s)\partial_\theta \log(p(a|\theta,s))\rangle.\end{aligned}$$

Unfortunately, we usually can't evaluate the last line above. However, what we can do is approximate it using a sampled value: We simply play a game with our current network, then replace the expected value above by the reward actually captured on the i-th move" (Landy, Deep reinforcement learning, battleship, 2016)

This proved to be a very difficult thing to do, and I never managed it. Most of the work I did on the integration of ANNs into my program was done with my research, but some examples of my work are in, however they are very jumbled as I did not get a working version of the code before I mentally worked out it was not feasible. I stopped trying to use ANNs then I realised that that continuing down this route would be too tricky.
From the knowledge, I gained through my research, I feel confident that I can create my own algorithm using the concepts and ideas from the other ones I have looked at. My concept is to have a board where all the cells are weighted, and the weight depends on where the program finds ships. This is a similar concept to how all the other learning algorithms would have worked, but I am choosing the values to weight it by and I am choosing how the weights are added. It will not as effective or efficient, but it should work, and it will be my only option as I cannot get any of the others to work.

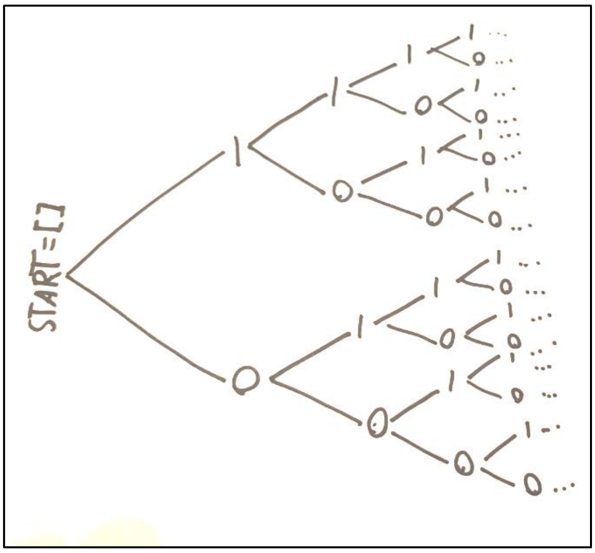
My focus from this point was to work out how to weight the cells in the board. I left the cells in the classes so each individual cell can have a weighted value to go with its other properties. My main problem now what that I realised that the board is dynamic, so the values will have to be dynamic as well. To simplify the problem, I reduced the number of ships to one of length three. This meant I would no longer have to worry of how it would have to distinguish between two ships. This greatly reduced number of different board combination there could be, because the other cells of the ship will only a change in one of the axis. This meant I could use the centred board idea I had theorised about. Just so the board was a reasonable size, I reduced it down to 5. All these changes I made are revisable so if working they can then be scaled up to larger ships and larger board by only changing one or two variables.

By simplify the problem I tried to go back to the neural network concept. I got a network working on a one-dimensional version of the board, where it could find the pattern of a ship, horizontal or vertical. The problem was that when the ship moved then it would have to retrain. Before I could fix this, worked out if there was any point doing so. The recognition if the pattern of the ship was only possible because all the cells were not hidden. As soon as the cell become hidden then there is no pattern and the only knowledge it would have learnt is where the last ship was. For the program to then learn where the next ship it is and how to sink it would need learning data. It would need to learn every state of the ship, when parts are hidden and when parts have been shot.

I felt that this was a major setback and there was no point in trying to proceed further with the neural network approach. However, in light of this, I did learn that I would need a separate set of weights for each scenario. I realised that I would need a set of storage locations to hold the different combinations of the board. Originally, I thought of having one class with all the different boards as properties of it. This would be a very good class structure and similar to how iterative learning trees work, but I decided against the idea. If I had gone for this idea then the data wouldn’t get saved when the program had finished; the only way I could store the data would be through flat files, due to time constraints at this stage. I realised that I could always use a CSV file like I originally intended. This would give me structure to store each line individually in rows, with the weights of all the boards cells stored in the same row.

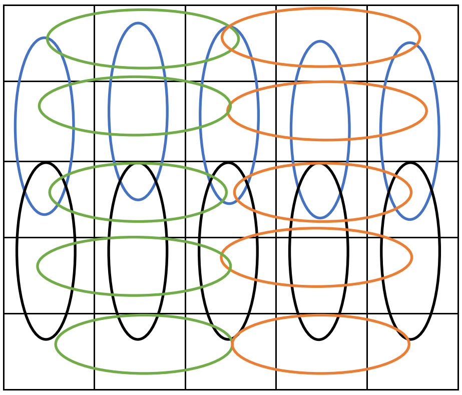
This plan seemed feasible, but there would always be the issue of a having a very large amount of data. The data could be sorted into a tree to make it easy to iterate, but there would still be a lot of data. If the layers were being created dynamically as they were being encountered then there would be significantly less data. The tree would be very useful to store this as for each cell it could branch a different way, and the boards could be added in to the right place as they were being created.

Even if the data was being created dynamically there would still be a lot. I decided to get the numbers to see if it would be too many and as a first trial to see if the method worked. I planned as if I were to create them all algorithmically at the start and not dynamically, then come to creating it dynamically if it works. For each of the three symbols currently used to represent a cell; -, ~, #, S (hidden, empty, off the board, and ship); there would be the number of symbols possibilities, to the power of the total number of cells on the board. That would be 4^(5*5) which is equal to 1.26*10^15. Considering that each of these rows would have 1(for the board) + 25 (for the values) items.



That would be a file of a total of 2.927*10^16 items, which is much too large.

I set out to reduce the number of possibilities. Firstly, I worked out that all of these would not be possible, because the ship would only occur in a line of 3 and the off the board marks only in rows and columns. I then worked out that shooting off the board is as much of a negative penalty as shooting a cell that's already been shot, so the same marker can be used for off the board and miss. This reduced the number down further, but it would still be too many. I decided to work out the possible combinations again.



The 25 different places for the ship on this board.

So, to the number of symbols; - and ~; to the power of the number of cells, all multiplied by the number of potential locations for the ship.

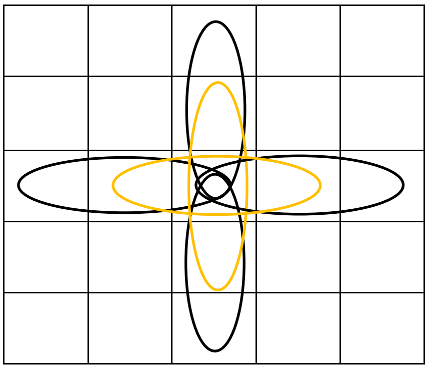
Rows of data) 25*(2)^(5*5) = 8.389*10^8

Items in file) (1+25)*25*(2)^(5*5) = 2.181*10^8

I decided to experiment with the board being 4 by 4.

Rows of data) 16*(2)^(4*4) = 1.049*10^6

Items in file) (1+16)*16*(2)^(4*4) = 1.782*10^7



I decided the board being 4 by 4 was a good move because the number of possible combinations (rows of data) is 0.125% of that from the 5 by 5 board. This was only now that I remembered the board could still be centred so that would reduce the number of possible ship places. I then ran the function again with the centred total number of ship placements, of course only the 5 by 5 board can be centred.

Rows of data) 6*(2)^(5*5) = 2.013*10^8

Items in file) (1+25)*6*(2)^(5*5) = 5.234*10^9

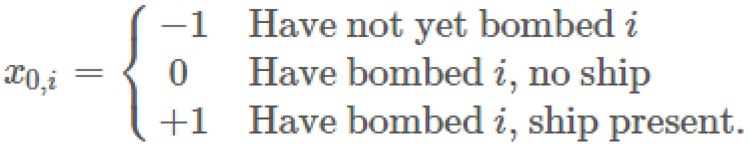
However, this still isn’t a large enough reduction. By reducing the length of the ship to 2, you only create 2 less possibilities on the 5 by 5 and no less on the 4 by 4. It was here I decided that actually reducing the ship to size two was the right this to do, then the board could become size 3 by 3. Which would reduce the size to a reasonable amount.

Rows of data) 4*(2)^(3*3) = 2048

Items in file) (1+9)*4*(2)^(3*3) = 20480

My reasoning behind this was that if I can get it to work on the 3 by 2 scale then that can be used anywhere on the ship. The centre piece will always be a hit, and the board changes after the next hit. This meant that in what the boards look like, only the centre cell will be hit, and that will already be hit to centre the board, and the other eight cells will either be hidden or empty. Underneath a hidden cell there will be the ship, but that doesn’t change how the board looks and that can be held in the cells properties.

As a representation, the values that are completely wrong are -1, moves to be discouraged, like missing, and plus 0.1 is to encourage behaviour, like hitting a ship. So, if the cell is not hidden then it gets -1, if the cell is a miss it gets -0.1, as this is not such a big penalty, and for a hit it is 0.1 to balance out the penalty for a miss. These are the rewards, which would normally be given by a machine learning algorithm, I had to create myself.



$$x_{0,i} = \begin{cases} -1 & \text{Have not yet bombed } i \\ 0 & \text{Have bombed } i\text{, no ship} \\ +1 & \text{Have bombed } i\text{, ship present.} \end{cases}$$

The weighting method of the ANN solution, by EFAVDB, is different, but this is because these are the weights given to them before they are put through the ANN and relate to how they should be processed for the final best shot to be found, where as I am keeping a record of all the values and just iteration g through manually to find the best one. These values would normally be generated in the training part of the machine learning process.

1)	1	2	3	2)	~	-	-	3)	-1	0.1	-0.1	4)	-1	-0.1	-0.1
	4	5	6		-	S	~		-0.1	-1	-1		0.1	-1	-1
	7	8	9		~	-	~		-1	-0.1	-1		-1	-0.1	-1
5)	-1	-0.1	-0.1	6)	-1	0.1	-0.1	7)	~	S	-	8)	-1	-1	-0.1
	-0.1	-1	-1		-0.1	-1	-1		-	S	~		-0.1	-1	-1
	-1	0.1	-1		-1	0.1	-1		~	-	~		-1	0.1	-1

1) Shows the cell labelling of all the squares
2) Show what the board will look like, it will be represented as ~---S~~-~ in the file
3) 4) & 5) all show the different length 2 ship possibilities that are possible with this board.
6) Shows how a longer length ship could be place, for when the code is expanded
7) & 8) show how the board could then be after the ship on grid 6 gets shot at cell 2

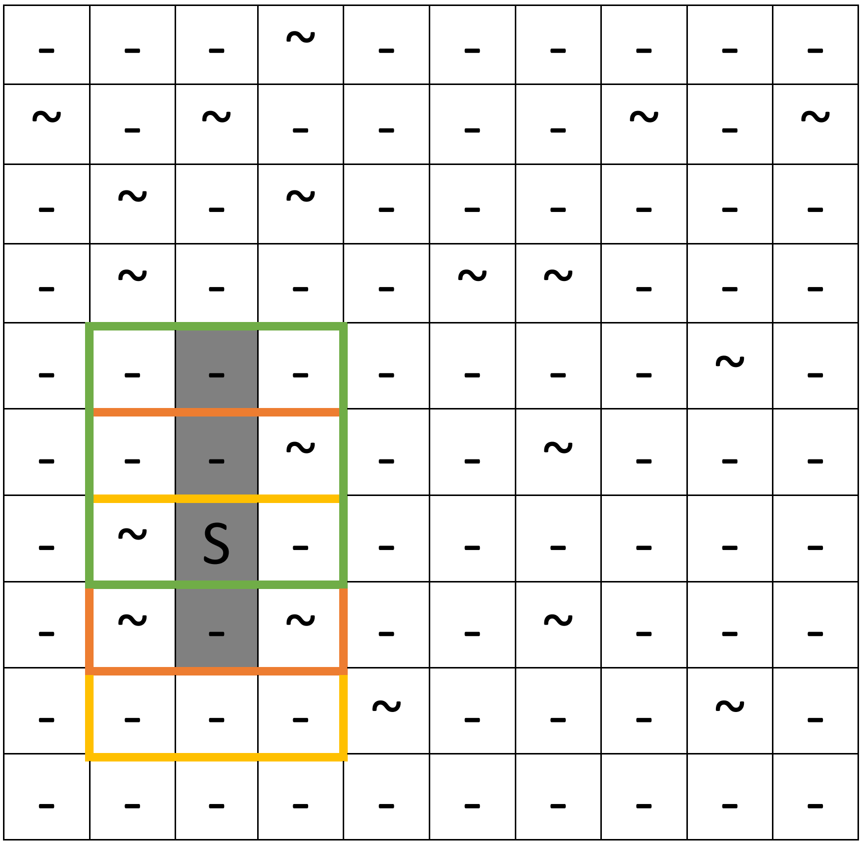
What all this means is that you only need to have the two symbols, to the power of the cells that are not the centred hit. Only what the board looks like has to be represented by the storage file.

- / ~	- / ~	- / ~
- / ~	S	- / ~
- / ~	- / ~	- / ~

Rows of data) (2)^(3*3-1) = 256

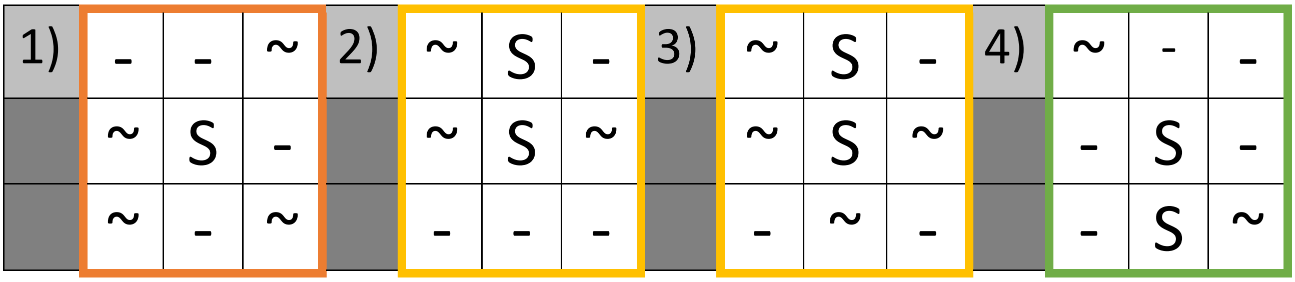
Items in file) (1+9)*(2)^(3*3-1) = 2560

This eliminated all size of the data file aspect, and allows the code to be built on. Once this works it could be built so it then iterates along the whole ship, with relative ease. Realistically there will only be the 3 by 3 section of the board being looked at.



-	-	-	~	-	-	-	-	-	-
~	-	~	-	-	-	-	~	-	~
-	~	-	~	-	-	-	-	-	-
-	~	-	-	-	~	~	-	-	-
-	-	-	-	-	-	-	-	~	-
-	-	-	~	-	-	~	-	-	-
-	~	S	-	-	-	-	-	-	-
-	~	-	~	-	-	~	-	-	-
-	-	-	-	~	-	-	-	~	-
-	-	-	-	-	-	-	-	-	-

This is a representation on the board and one ship. I am going to use this to explain why only a 3 by 3 grid is needed. The ship is shown by the shaded area, as its potion is unknown to the program. The initial hit is marked with the S and the centred board is marked in orange. From this board, it shoots down hitting another, centring the board to the yellow square, it shoots down again, but misses. It knows there are no more reasonable shots, so moves up to the orange, then shoots the to where it gets centred in the green board. It then shoots up from the green board and sinks the ship. This will work with all sizes and once the message is displays showing the ship to be sunk, it can move back to normal firing.



1)	-	-	~	2)	~	S	-	3)	~	S	-	4)	~	-	-
	~	S	-		~	S	~		~	S	~		-	S	-
	~	-	~		-	-	-		-	~	-		-	S	~

The down side to this is on its own it does not look like much, but it is a key step on working out a solution. Also, it means that shooting the board before the first hit, to centre the board, is a completely different task. For now, this task will be a firing rom randomly generated coordinates, because even if it did learn, the learning how to shoot a blank board will not be as crucial to winning the game as it sinking a ship is. This function will remain random until the sinking the ship side of the code is complete.

To learn and work out what the weights were every cell would start with an even weight. This weight would then get increased or decreased depending on where the shot was fired. If the AI shot a cell that it had already shot, or one that was off the board then it would get a punishment weight of -1 added to the total value of the cell. For a miss its -0.1 and a hit is +0.1. This encourages shots on target, this is so on a board representation where there can be multiple reasonable shots, there is an even chance that it is any of these, and not just the first it comes to. The following diagram explains this.

1)	1	2	3	2)	~	-	-	3)	-	-	-	4)	-1	0.1	-0.1
	4	5	6		-	S	~		-	S	~		-0.1	-1	-1
	7	8	9		~	~	~		-	~	-		-1	-1	-1
5)	-1	-0.1	-0.1	6)	-0.1	0.1	-0.1	7)	-0.1	-0.1	-0.1	8)	0	50%	0
	0.1	-1	-1		-0.1	-1	-1		0.1	-1	-1		50%	0	0
	-1	-1	-1		-0.1	-1	-0.1		-0.1	-1	-0.1		0	0	0

1) Shows the cell labelling of all the squares
2) & 3) Show what the board will look like, in two different scenarios
4) & 5) all show the different possibilities for the ship placement and values of grid 2
6) & 7) all show the different possibilities for the ship placement and values of grid 3
8) Shows how the percentage chance of the cells being the ship is split between 2 and is the same for both.

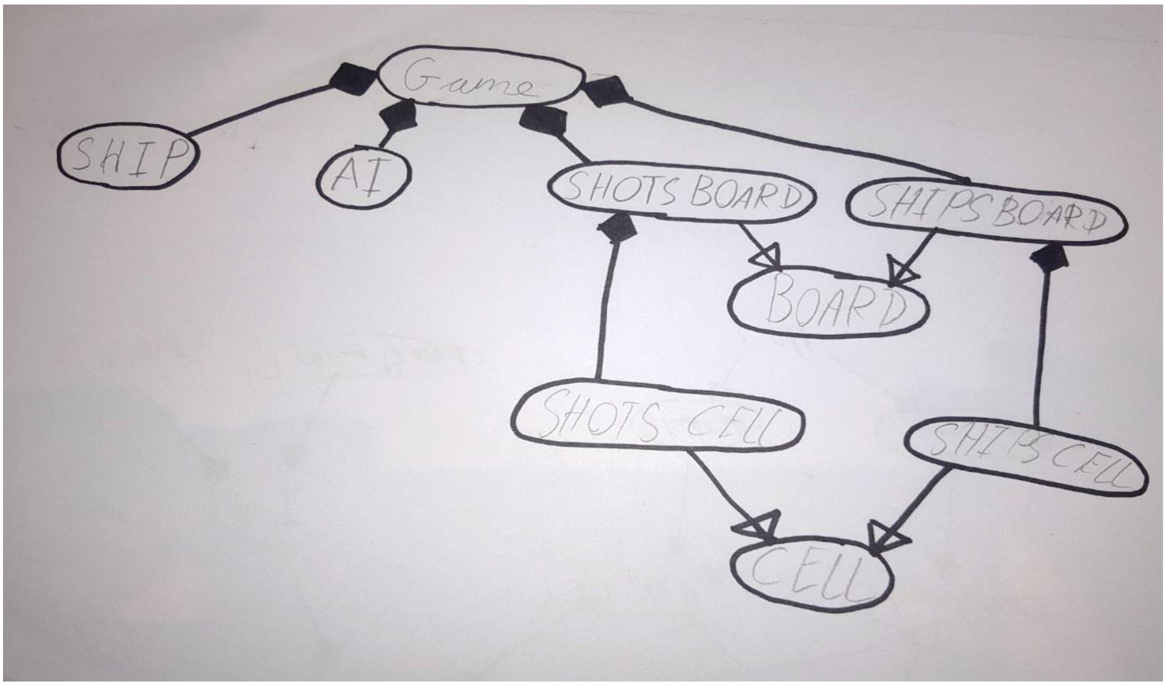
This shows how the total values for each cell in each board scenario, must be treated as equal and must balance out. The way the program will work is it will choose the cell on the board with the height weight to be the one it shoots, and the consequential value from that shot will be the one added to the chosen cell. The values for all the cells will be talked out of the file, and the consequential values for each shot will be created dynamically depending on what that cell holds. The new weights of all the cells will then be put back into a list and stored in the file.

If the tree structure does not work then another search, like binary could be used. However, a temporary way could be to just remember the location of that instance of the board in the file, so it could be move straight back into place when writing over the file. I think for now I will create all the instances of the boards in the file at the start of the game, as there will only be 256.

I experimented with having two separate programs that ran simultaneously, and only communicated through a text file. This was an early prototype, which I ended up scraping, but I did get it working at some stage. The idea behind it was that the text file would represent the game, and the learning program would be getting all of its information form the game, so it would be like it was actually playing the game. I realised that this idea was highly impractical, because the games happened so quickly they could not be followed and there was a high chance of error. Also, many of the pre-existing classes were designed to work with the l learning program, so it being in separated in different files made little sense. Examples from some of these files are in Appendix 6.

While this learning algorithm will be very simple, it should learn and should be very effective. It is also built so that it could later be developed into a full working algorithm that plays battleships. This program will not directly meet all the requirements, but they were set to a high standard before the full complexity of the task was discovered. Also, if it works as intended, it would still partial complete the assigned task. It will be a working frame work that shows that elements of battleships can be learnt by a learning algorithm, and the structure will be there that it can be enhanced into making a program that does fully learn to pay battleships.

I made a high-level plan for my design, to incorporate all the classes. This will be the foundation that I will use for the rest of my program.

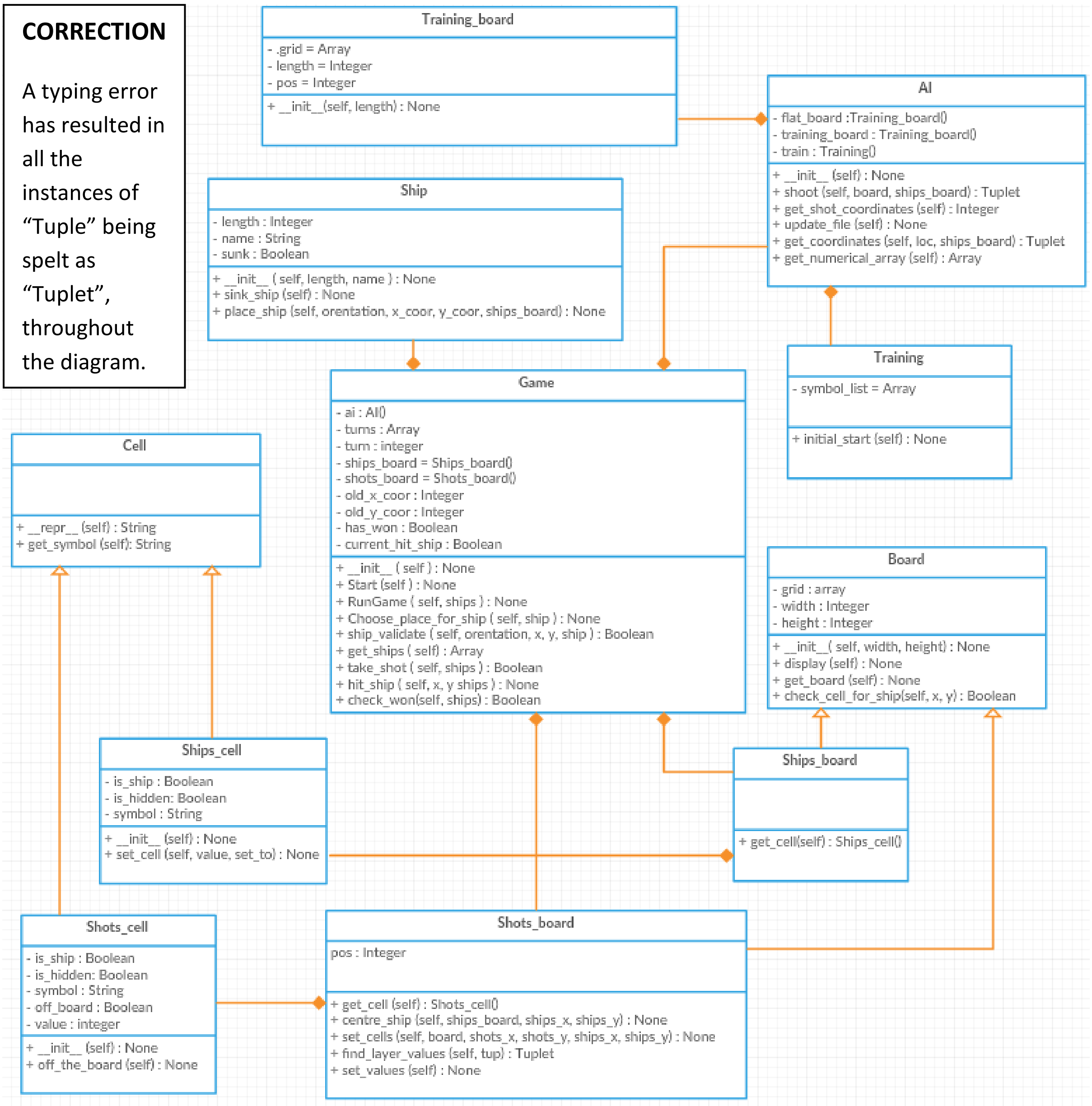


I feel like this will be a lot of work to implement all of the code, so I have decided to leave out the additional features for now. These include the tree search and storage of the board files.

I expanded on the previous diagrams and have produced the following class diagram for how I plan on structuring my program. It includes all the properties and methods, so it is easy to follow.

CORRECTION

A typing error has resulted in all the instances of "Tuple" being spelt as "Tuplet", throughout the diagram.


Training_board
- .grid = Array
- length = Integer
- pos = Integer
+ __init__(self, length) : None
AI
- flat_board :Training_board()
- training_board : Training_board()
- train : Training()
+ __init__ (self) : None
+ shoot (self, board, ships_board) : Tuplet
+ get_shot_coordinates (self) : Integer
+ update_file (self) : None
+ get_coordinates (self, loc, ships_board) : Tuplet
+ get_numerical_array (self) : Array
Ship
- length : Integer
- name : String
- sunk : Boolean
+ __init__ (self, length, name) : None
+ sink_ship (self) : None
+ place_ship (self, orentation, x_coor, y_coor, ships_board) : None
Training
- symbol_list = Array
+ initial_start (self) : None
Game
- ai : AI()
- turns : Array
- turn : integer
- ships_board = Ships_board()
- shots_board = Shots_board()
- old_x_coor : Integer
- old_y_coor : Integer
- has_won : Boolean
- current_hit_ship : Boolean
+ __init__ (self) : None
+ Start (self) : None
+ RunGame (self, ships) : None
+ Choose_place_for_ship (self, ship) : None
+ ship_validate (self, orentation, x, y, ship) : Boolean
+ get_ships (self) : Array
+ take_shot (self, ships) : Boolean
+ hit_ship (self, x, y ships) : None
+ check_won(self, ships) : Boolean
Cell
+ __repr__ (self) : String
+ get_symbol (self): String
Board
- grid : array
- width : Integer
- height : Integer
+ __init__(self, width, height) : None
+ display (self) : None
+ get_board (self) : None
+ check_cell_for_ship(self, x, y) : Boolean
Ships_cell
- is_ship : Boolean
- is_hidden: Boolean
- symbol : String
+ __init__ (self) : None
+ set_cell (self, value, set_to) : None
Ships_board
+ get_cell(self) : Ships_cell()
Shots_cell
- is_ship : Boolean
- is_hidden: Boolean
- symbol : String
- off_board : Boolean
- value : integer
+ __init__ (self) : None
+ off_the_board (self) : None
Shots_board
pos : Integer
+ get_cell (self) : Shots_cell()
+ centre_ship (self, ships_board, ships_x, ships_y) : None
+ set_cells (self, board, shots_x, shots_y, ships_x, ships_y) : None
+ find_layer_values (self, tup) : Tuplet
+ set_values (self) : None

These are the new designs I came up with after editing and changing the design laid out in the initial phase, where I created the base game. The classes Shots_board and AI have methods to access the file. In the Shots_board class the method is: find_layer_values(self, Tip)

The learnt data will be stored in a CSV file. This is because the format of a CSV file will provide it with the structure that it needs and this file type does not take up large amounts of storage. They are being chosen because they are efficient to write to, read from and don't take up much storage. This means by using this file type I will keep to the requirements. The way in which the data will be arranged in the file will be like the following table.

Board representation	Value of cell 1	Value of cell 2	Value of cell 3	Value of cell 4	Value of cell 5	Value of cell 6	Value of cell 7	Value of cell 8	Value of cell 9
Before learning example									
----S----	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
-~-~S~--~	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
After learning example									
----S----	-1	2.5	-1	2.5	-1	2.5	-1	2.5	-1
-~-~S~--~	-1	-1	-1	-1	-1	-1	-1	10	-1

I will write an algorithm to produce this file before the program starts learning. The itertools module has a useful function, product(a, b), that can generate every possible combination for the characters in a, for the number of items there are, b. The product function is very similar to the following piece of code in how it works.

```python
def product(*args, **kwds):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = map(tuple, args) * kwds.get('repeat', 1)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

(Python Software Foundation, No Date)

Once I have all the possible boards, I will just create the nine starting values needed. These starting values are all, 0, as no weight has been assigned yet.

I have decided to hard code all the possible boards because they will not take up too much storage and iterating directly through 256 items will not take too much time. This will allow me to get a basic frame work in place so the code can be expanded further, to allow for the boards and data entries to be created, and stored, in a data structure like a linked list or tree.

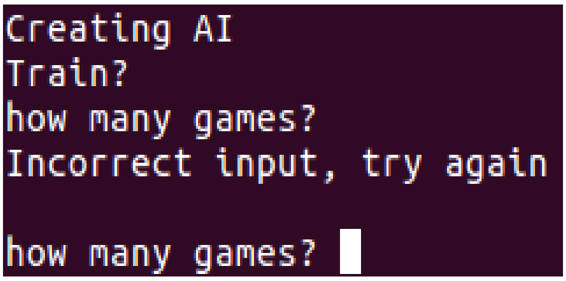
I have written pseudo code to explain the algorithm I will use to generate the file:

```
symbol_list <= ["-","~"]
        array <= itertools.product(self.symbol_list, repeat = 8)
                OPEN("layers.csv", "write")
                FOR i IN array:
                        list <= LIST ITEMS OF i
                        INSERT 'S' AS 4th ITEM OF list
                        String <= ITEMS OF list AS ONE STRING
                        WRITE ((s, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)) TO "layers.csv"
                END FOR
                CLOSE "layers.csv"
```

This code will be written in the training class, as the file creation Is part of the initial training of the program. The training part of the code woul dnormally be where the training takes place, but my program will always be learning dynamically so no separate training section is needed. I have kept the name, because this part of the program will still be involved in the training aspects. It will be where the file that stores all the learnt data gets created and reset, both of which are important requirements of the program.

I plan on having a very basic user interface. Similar to how the first instance of the game was displayed, but it needs to have the options to choose: whether to retrain the learning algorithm, or to keep the current intelligence, which will be stored in the file; and the option of how many games you would like it to loop through. Also, the at the end of the program there need to be a display of some measurable numbers to prover it is learning. These all are key features and are specified in the requirements, so must be implemented.

I have made a mock-up of how the user interface will look. I was given no indication as to how the program should display, so I have kept it to a minimum. The mock up can be seen pictured below.

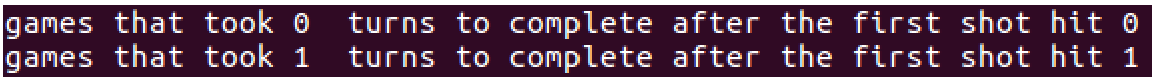


```
Creating AI
Train?
how many games?
Incorrect input, try again

how many games? 
```

As you can see it is a very basic format. It displays what stage of the process it is currently at, then asks to the number of games. The instructions maybe considered vague, but they do not require much explaining. The only users will be me and the client, so they do not have to be informative. I do like the idea of there being a message that displays if they a wrong input is entered, but its not really needed, as it would probably be a miss type, rather than the user not knowing what to do. Once a valid input has been given, then the program will play games until it reaches the specified number.

I think that my safest option for displaying values that can be used to calculate the learn rate, is to display the number of turns it takes to complete each game after the first shot hits a ship, so on the centred board, as this is the board it is learning to play on. Displaying each one individually will leave too much data to look through, so I will print how many games took each number of turns. In the number of turns it takes can be a varying number, so I will have to make a system that dynamically creates the list of how long each game took to win. These values could then be written to a file, but I feel this is not necessary, and it would use more storage.

The following picture is a visual representation of what the console output would look like:



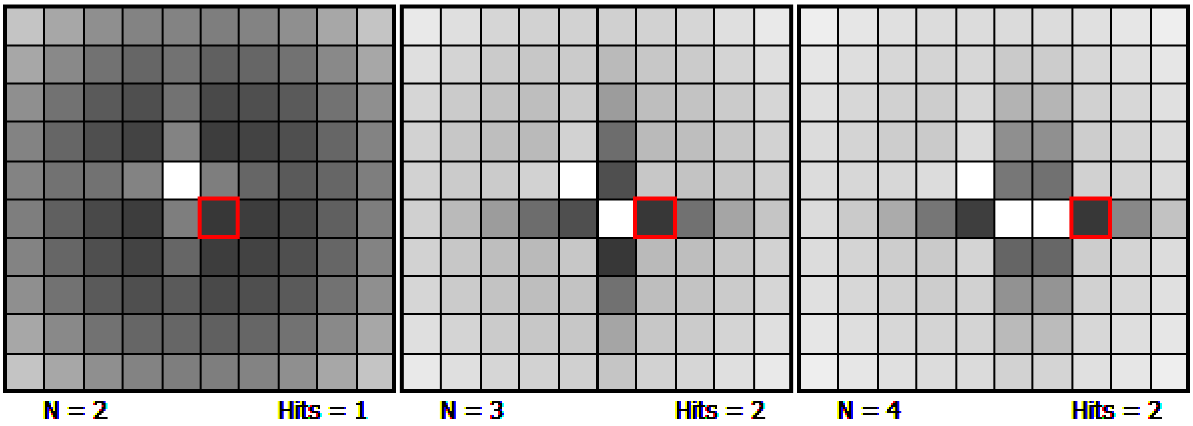
```
games that took 0  turns to complete after the first shot hit 0
games that took 1  turns to complete after the first shot hit 1
```

This part of the program will not be in a class, but rather at the end of the document, where the class gets called. It is here outside all the classes that the program will be called and the results will be processed. It is also where most the important user interface will be.

The cells for the boards will be inherited from a parent cell, so they can share the methods they need. The reason the parent cell has not properties is be the way python inheritance works, I would have to either override the initial function, or would have to create another method to inherit and call just to keep the three properties they share. Neither of these seemed to be the sensible option, so I left them to declare the properties separately.

The theory behind centring the board in the way that I do, is so that the program only has to learn the smaller grid. On this grid is learnt then it can be used anywhere on the board to sink any ship. It represents the feasible area for a well-educated shot. If while iterating along a hit ship all the shots it can chose from have negative probabilities then the code can be expanded to search the other end of the ship it had been hitting. This way the learning algorithm is still only learning a small board, which leaves less room for error. A separate learning program can be used to learn how to iterate along the ships. It can be taught directly from the weights received by the grid searching algorithm and they will work in together. This is a much better strategy as there is much less room for error, and is the AI did learn to play on a large board then only shots it will take after hitting ship will be in the immediate area, due do the nature of probability.

It is easy to see with these images. They represent a heat map of the likely hood of a cell containing a ship in the play through battleships.



(Berry, 2011)

In the first image the ship was hit. The other two images show where the probabilitys now lay for the other cells being hit. The strongest amost probable chances for a hit usually occur in a one square raduis. When the shot fires to the most likely position and its not a ship, image 3, this is when the program would, once expanded, move to the other end of the ship to try there.

When choosing a random location for the ship to be placed there are two things that must be considered regarding the validity of the placement. Firstly all the parts of the ship must be on the board, secondly will it be placed over any other ships. I have a function that should work these out, depending of the ships orientation, length and desired coordinates it should first work out if the ship is too long for the place, and secondly work out if there is already a ship there. In doing them in that order it will prevent an error occurring if it tried to check of a ship in a location outside of the board. The pseudo code for such a task is as follows.

```
IF orientation IS HORAZONTAL
    IF (ship_length + x_cooordinate) > WIDTH OF THE BOARD - 1
        return False
    ELSE
        FOR i IN LENTH OF SHIP
            IF THERE IS A BOARD IN THE CELL (x+i, y):
                RETURN False
            END IF
        END FOR
    END IF
ELSE
    IF (ship_length + y_cooordinate) > HEIGHT OF THE BOARD - 1
        return False
    ELSE
        FOR i IN LENTH OF SHIP
            IF THERE IS A BOARD IN THE CELL (x, y+i )
                RETURN False
            END IF
        END FOR
    END IF
END IF
RETURN True
```

The function that allow the AI to shoot on the board is shoot. It takes in the board. It then flattens this board to a one-dimensional array. If two boards were passed in then it means the board has been centred so the AI must take the shot, but if only one board was entered then the board is the ships board and the shot should be taken by the random coordinate generator.

Any inputs into the program must be validated to ensure the program is robust. This means validate the users direct input with a TRY statement. While in a TRY statement if the code were to crash, it instead would pull up an exception. It must be told what sort of error to expect, in order to catch it. A ValueError is what I would expect from an invalid string being converted to an integer, which will be useful for when the user enters the number of games they would like the algorithm to play. An IOError can be expected when the program attempts to read from a file that cannot be accessed. I will encase all instances of opening a file in the program TRY statements, expecting an IOError; with the option to try again.

get_numerical_array will be the function that returns a list of what all the consequence weights for all of the tiles on the board. It will iterate through the board, building a list consisting of float numbers. These will be -1, -0.1, 0.1; I have already explained why I have chosen these numbers. The following pseudo code shows the process

```
temp <= []
FOR cell IN board
    IF CELL IS HIDDEN OR CELL IS OFF THE BOARD:
        APPEND -1 TO TEMP
    ELSE IF CELL IS SHIP:
        APPEND 0.1 TO TEMP
    else:
        APPEND -0.1 TO TEMP
    END IF
END FOR
RETURN temp
```

The function that works out what cell to shoot, once the first hit has been had, is a very simple function that iterates through the list of cells and finds the one with the greatest value. This cell is then chosen as the cell to shoot. The location of this cell in the one - dimensional array is then returned so its coordinates can be retrieved, for the shot to then be taken. In a neural network instead of iterating through every item it would choose every cell and weigh up the weights and continue to play until enough moves have been taken and backtracked for the optimal rout to be taken.

I allowed the learning algorithms shots to land on cells it shouldn’t shoot, like the ones that have already been shot. This was so that I could prove that I could learn not to shoot those cells. I wanted the program to get as little help as possible when making it choices over where to shoot.

I could have used sigma and logarithmic functions to get nicer numbers to work with, but due to the simplicity of the learning algorithm I could just choose the largest value.

get_coordinates is a function that works out the coordinates of a cell on the ships_board just from its location in the one-dimensional array. A counter is used to work out when the loops have reached the coordinates of the cell. The counter gets passed through two for loops, one for the x-axis, one for the y-axis. Once the counter reaches the values of the cells one-dimensional location, then the current loops from the x and y loops will be equal to that cells coordinates. The pseudo code explains.

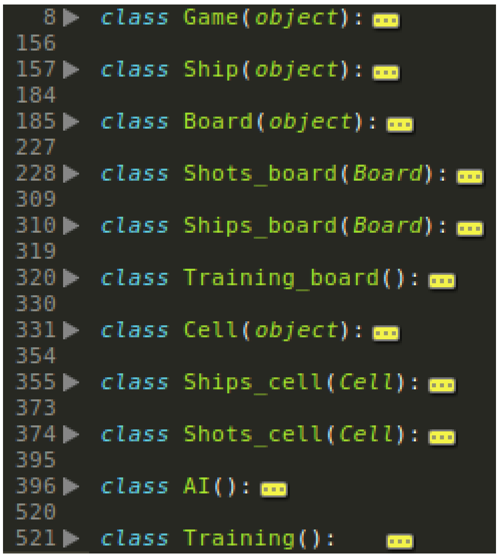
```
i <= 0
FOR y IN WIDTH OF THE BOARD
    FOR y IN HEIGHT OF THE BOARD
        IF i =1D LOCATION
            RETURN (x,y)
    END FOR
END FOR
```

Sinking of the ships is relatively simple. I use the length of the ships as a durability metre to work out how many hits they can take. I can do this because if a ship shoots a cell that has already been hit then it counts as a miss and I do not have to worry about the same cell being hit twice to sink the ship. This method is really efficient because no new variables have to be made, and the ships length is not used for anything else after its initial placing ion the board.

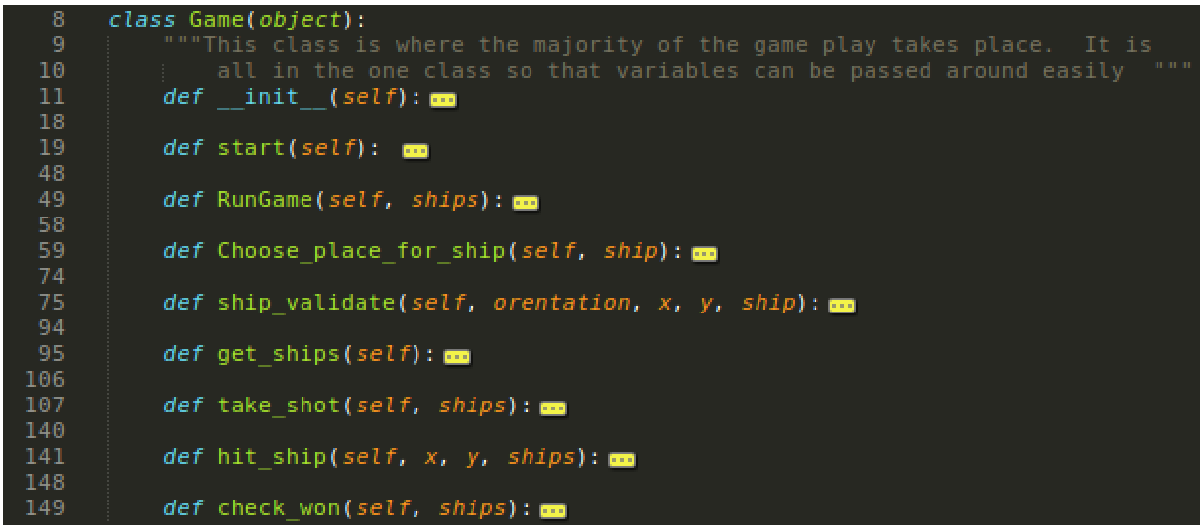
In the next section, you can see the final layout of the classes and subroutines in the final program. These were all put into place before the rest of the code was implemented. They are also labelled with a short description of what they do. For more detail on what the classes contain and how they work, all the code has been annotated to a very high level. The code can be found in Appendix 1. The comments on each part of the code explain the running of each function and the purposes of most the lines.

Implementation of the final program

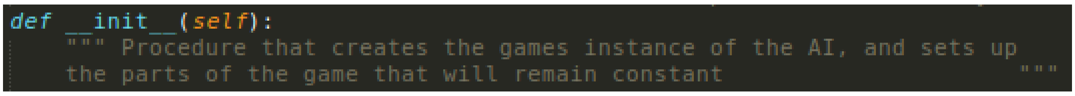
Appendix 1 is my final program. Throughout the code there are hundreds of comments explaining what each line, loop and conditional statement is doing. Also at the start of every class and subroutine there are short sentences explain what they do. I will not go on to explain some of the key elements of the code further, to ensure they are properly understood.



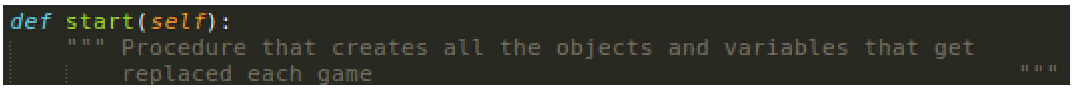
```
class Game(object):
class Ship(object):
class Board(object):
class Shots_board(Board):
class Ships_board(Board):
class Training_board():
class Cell(object):
class Ships_cell(Cell):
class Shots_cell(Cell):
class AI():
class Training():
```



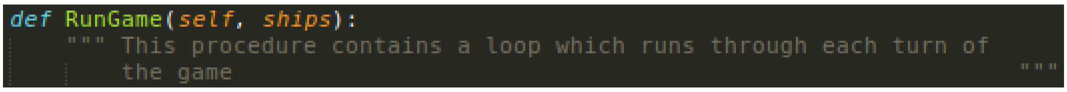
```
class Game(object):
    """This class is where the majority of the game play takes place.  It is
        all in the one class so that variables can be passed around easily  """
    def __init__(self):

    def start(self):

    def RunGame(self, ships):

    def Choose_place_for_ship(self, ship):

    def ship_validate(self, orentation, x, y, ship):

    def get_ships(self):

    def take_shot(self, ships):

    def hit_ship(self, x, y, ships):

    def check_won(self, ships):
```



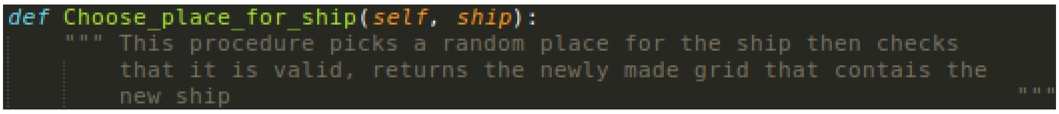
```
def __init__(self):
    """ Procedure that creates the games instance of the AI, and sets up
    the parts of the game that will remain constant                    """
```



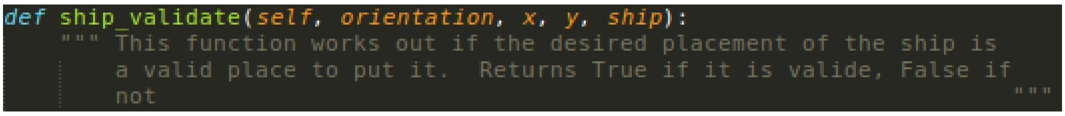
```
def start(self):
    """ Procedure that creates all the objects and variables that get
        replaced each game                                             """
```



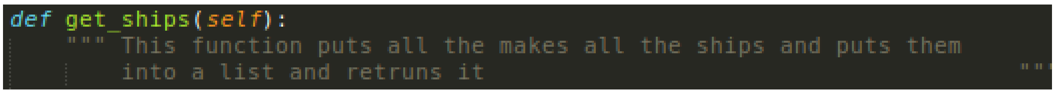
```
def RunGame(self, ships):
    """ This procedure contains a loop which runs through each turn of
        the game                                                       """
```



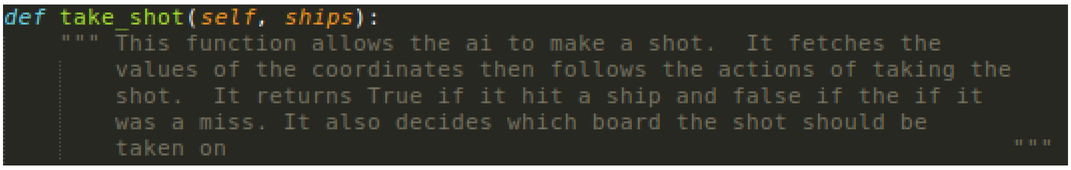
```
def Choose_place_for_ship(self, ship):
    """ This procedure picks a random place for the ship then checks
        that it is valid, returns the newly made grid that contains the
        new ship                                                      """
```



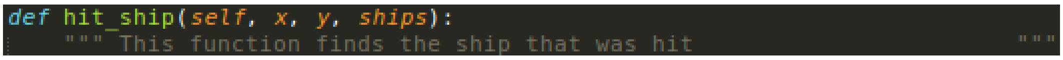
```
def ship_validate(self, orientation, x, y, ship):
    """ This function works out if the desired placement of the ship is
        a valid place to put it.  Returns True if it is valide, False if
        not                                                            """
```



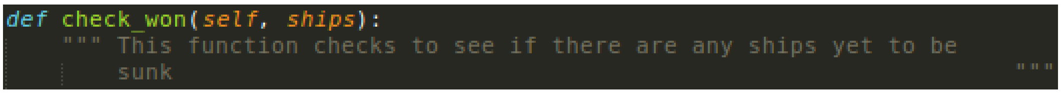
```
def get_ships(self):
    """ This function puts all the makes all the ships and puts them
        into a list and retruns it                                    """
```



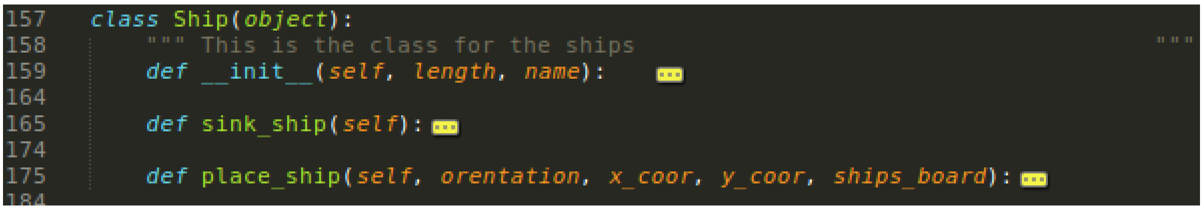
```
def take_shot(self, ships):
    """ This function allows the ai to make a shot.  It fetches the
        values of the coordinates then follows the actions of taking the
        shot.  It returns True if it hit a ship and false if the if it
        was a miss. It also decides which board the shot should be
        taken on                                                       """
```



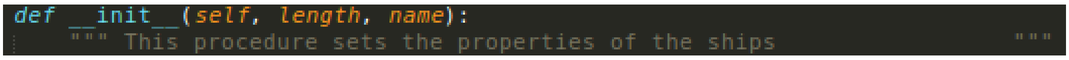
```
def hit_ship(self, x, y, ships):
    """ This function finds the ship that was hit                      """
```



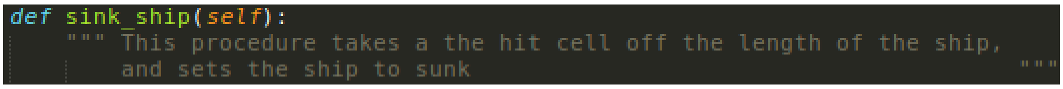
```
def check_won(self, ships):
    """ This function checks to see if there are any ships yet to be
        sunk                                                           """
```



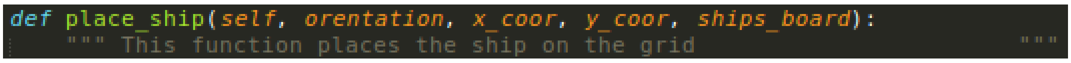
```
class Ship(object):
    """ This is the class for the ships                                         """
    def __init__(self, length, name):   ...

    def sink_ship(self): ...

    def place_ship(self, orentation, x_coor, y_coor, ships_board): ...
```



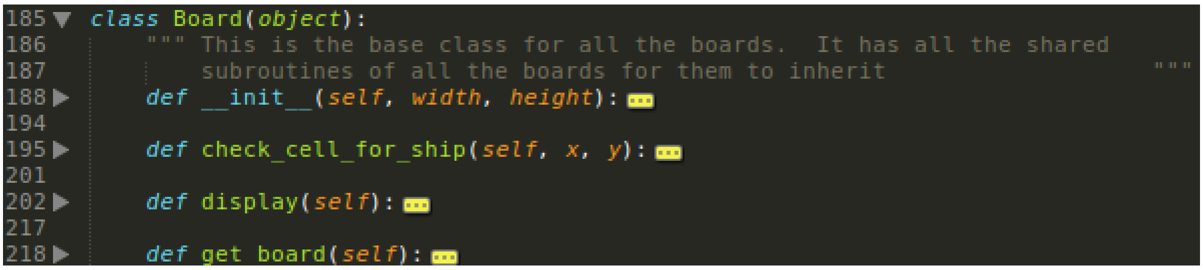
```
def __init__(self, length, name):
    """ This procedure sets the properties of the ships                """
```



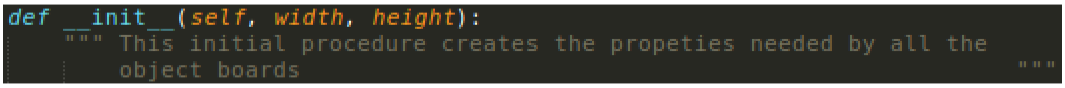
```
def sink_ship(self):
    """ This procedure takes a the hit cell off the length of the ship,
        and sets the ship to sunk                                      """
```



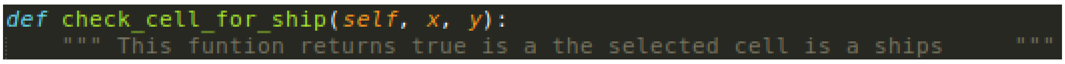
```
def place_ship(self, orentation, x_coor, y_coor, ships_board):
    """ This function places the ship on the grid                      """
```



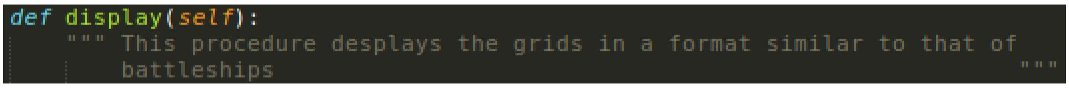
```
class Board(object):
    """ This is the base class for all the boards.  It has all the shared
        subroutines of all the boards for them to inherit                        """
    def __init__(self, width, height): ...

    def check_cell_for_ship(self, x, y): ...

    def display(self): ...

    def get_board(self): ...
```



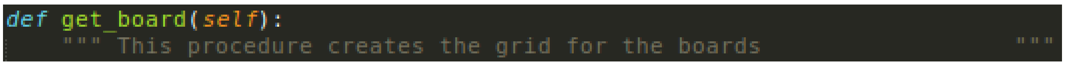
```
def __init__(self, width, height):
    """ This initial procedure creates the propeties needed by all the
        object boards                                                  """
```



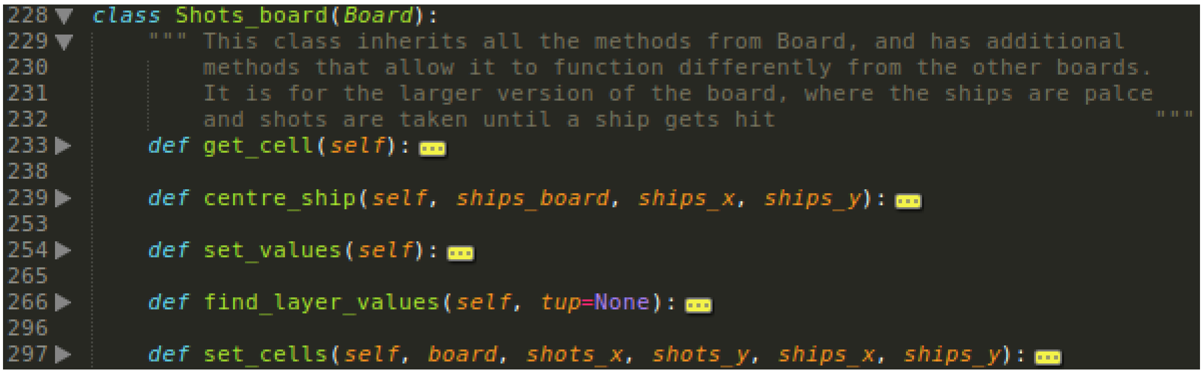
```
def check_cell_for_ship(self, x, y):
    """ This funtion returns true is a the selected cell is a ships    """
```



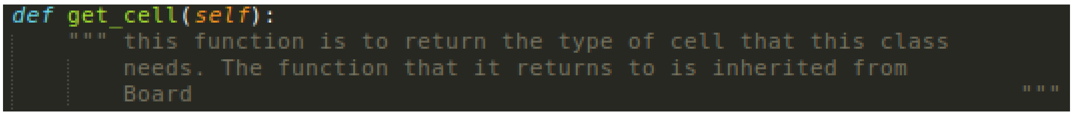
```
def display(self):
    """ This procedure desplays the grids in a format similar to that of
        battleships                                                       """
```



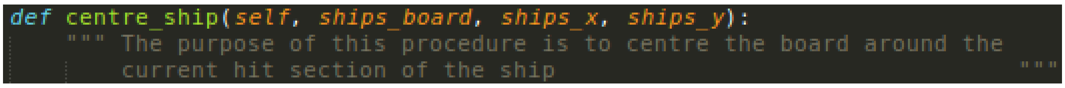
```
def get_board(self):
    """ This procedure creates the grid for the boards                 """
```



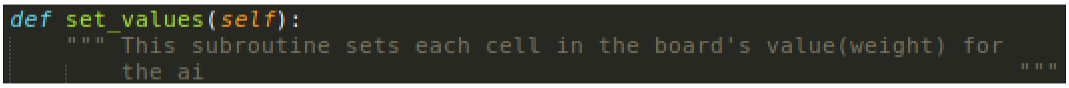
```
class Shots_board(Board):
    """ This class inherits all the methods from Board, and has additional
        methods that allow it to function differently from the other boards.
        It is for the larger version of the board, where the ships are palce
        and shots are taken until a ship gets hit                            """
    def get_cell(self): ...

    def centre_ship(self, ships_board, ships_x, ships_y): ...

    def set_values(self): ...

    def find_layer_values(self, tup=None): ...

    def set_cells(self, board, shots_x, shots_y, ships_x, ships_y): ...
```



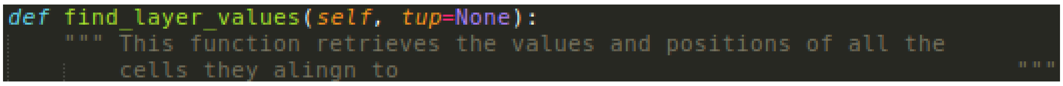
```
def get_cell(self):
    """ this function is to return the type of cell that this class
        needs. The function that it returns to is inherited from
        Board                                                          """
```



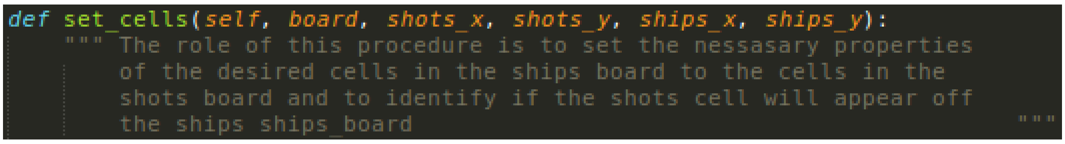
```
def centre_ship(self, ships_board, ships_x, ships_y):
    """ The purpose of this procedure is to centre the board around the
        current hit section of the ship                                  """
```



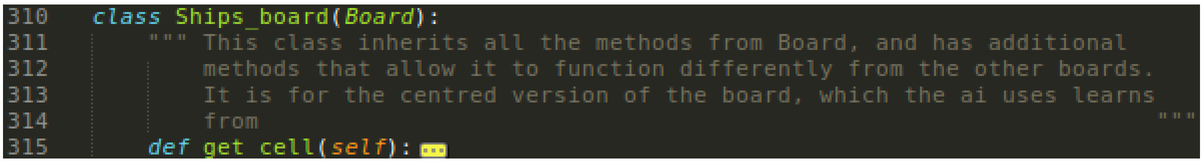
```
def set_values(self):
    """ This subroutine sets each cell in the board's value(weight) for
        the ai                                                           """
```



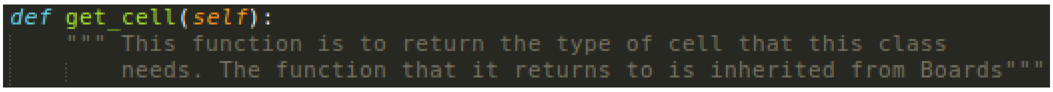
```
def find_layer_values(self, tup=None):
    """ This function retrieves the values and positions of all the
        cells they alingn to                                             """
```



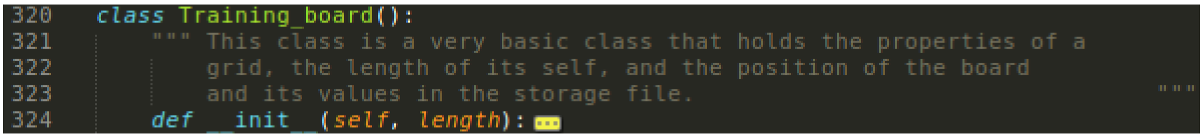
```
def set_cells(self, board, shots_x, shots_y, ships_x, ships_y):
    """ The role of this procedure is to set the nessasary properties
        of the desired cells in the ships board to the cells in the
        shots board and to identify if the shots cell will appear off
        the ships ships_board                                            """
```



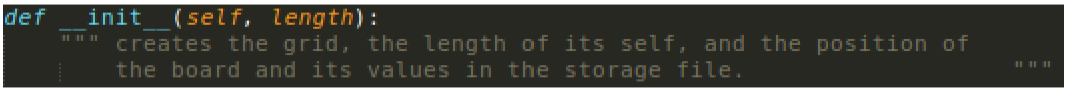
```
class Ships_board(Board):
    """ This class inherits all the methods from Board, and has additional
        methods that allow it to function differently from the other boards.
        It is for the centred version of the board, which the ai uses learns
        from                                                                  """
    def get_cell(self): ...
```



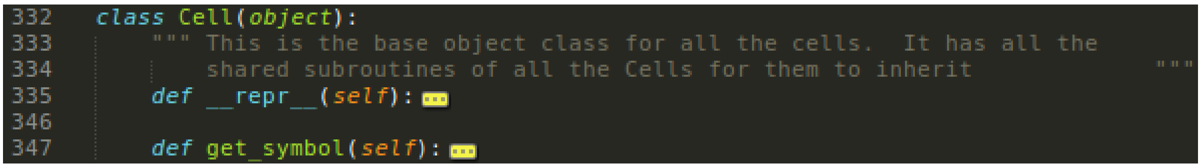
```
def get_cell(self):
    """ This function is to return the type of cell that this class
        needs. The function that it returns to is inherited from Boards"""
```



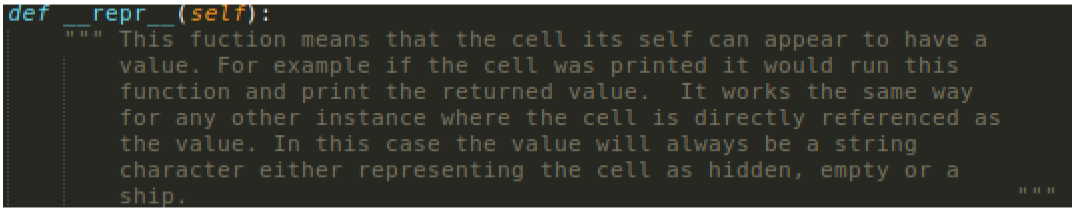
```
class Training_board():
    """ This class is a very basic class that holds the properties of a
        grid, the length of its self, and the position of the board
        and its values in the storage file.                                """
    def __init__(self, length): ...
```



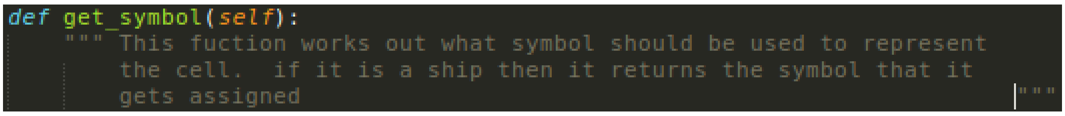
```
def __init__(self, length):
    """ creates the grid, the length of its self, and the position of
        the board and its values in the storage file.                   """
```



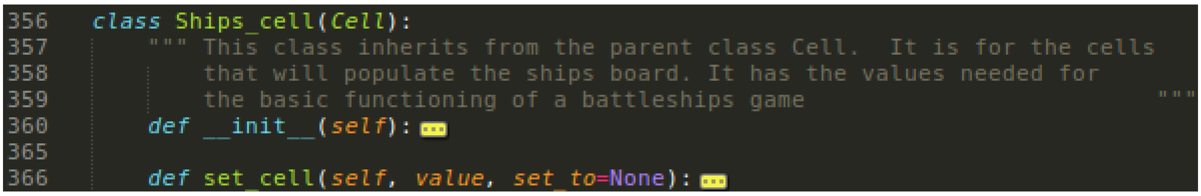
```
class Cell(object):
    """ This is the base object class for all the cells.  It has all the
        shared subroutines of all the Cells for them to inherit                 """
    def __repr__(self):

    def get_symbol(self):
```



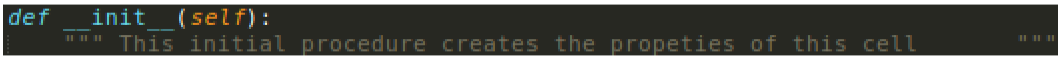
```
def __repr__(self):
    """ This fuction means that the cell its self can appear to have a
        value. For example if the cell was printed it would run this
        function and print the returned value.  It works the same way
        for any other instance where the cell is directly referenced as
        the value. In this case the value will always be a string
        character either representing the cell as hidden, empty or a
        ship.                                                             """
```



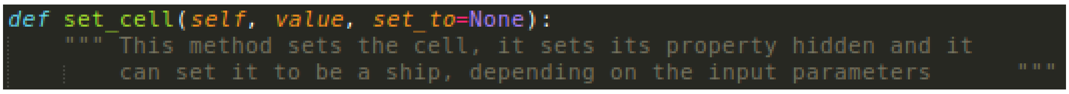
```
def get_symbol(self):
    """ This fuction works out what symbol should be used to represent
        the cell.  if it is a ship then it returns the symbol that it
        gets assigned                                                     """
```



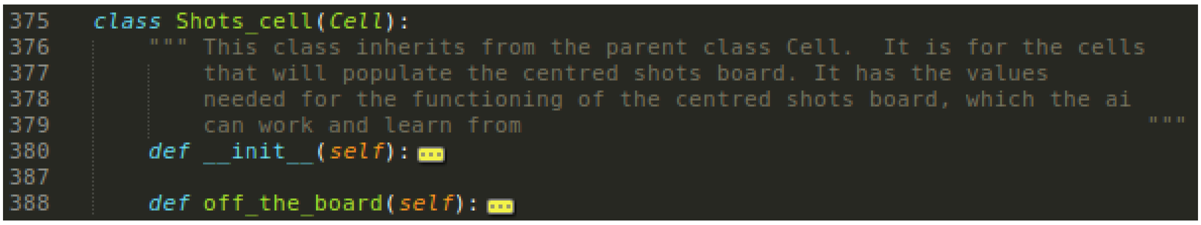
```
class Ships_cell(Cell):
    """ This class inherits from the parent class Cell.  It is for the cells
        that will populate the ships board. It has the values needed for
        the basic functioning of a battleships game                              """
    def __init__(self):

    def set_cell(self, value, set_to=None):
```



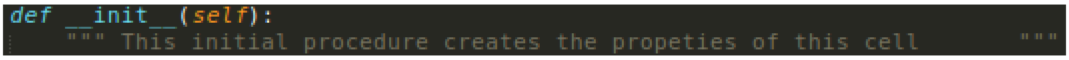
```
def __init__(self):
    """ This initial procedure creates the propeties of this cell          """
```



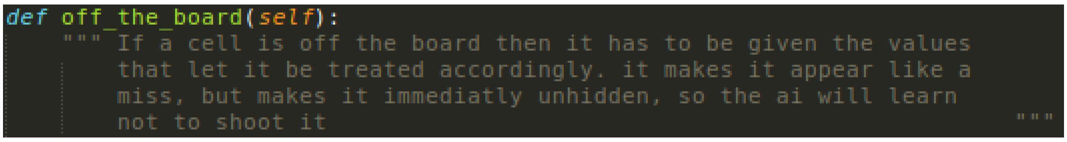
```
def set_cell(self, value, set_to=None):
    """ This method sets the cell, it sets its property hidden and it
        can set it to be a ship, depending on the input parameters         """
```



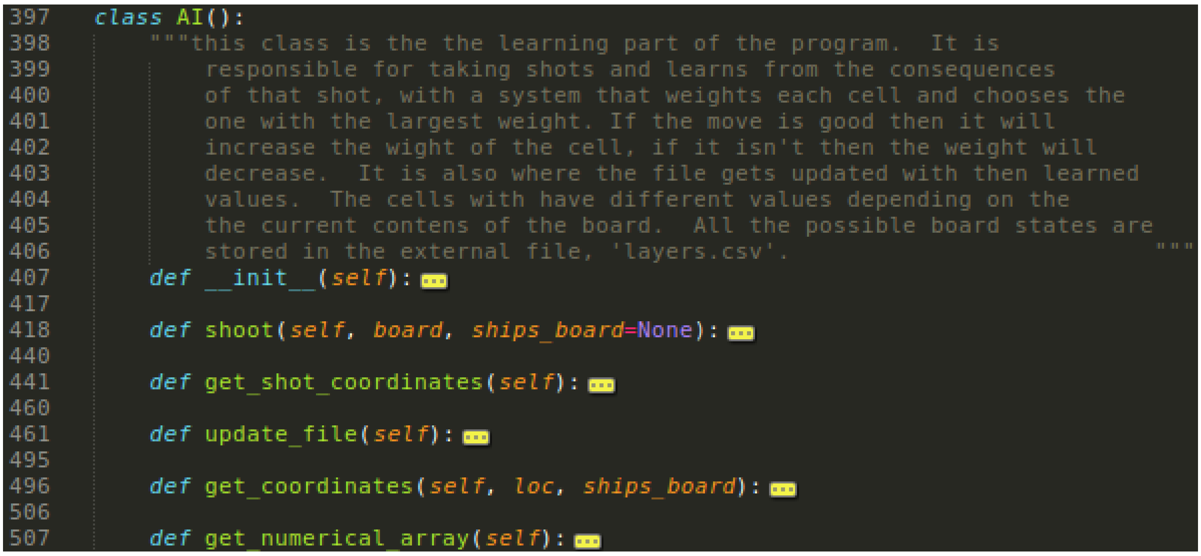
```
class Shots_cell(Cell):
    """ This class inherits from the parent class Cell.  It is for the cells
        that will populate the centred shots board. It has the values
        needed for the functioning of the centred shots board, which the ai
        can work and learn from                                                  """
    def __init__(self):

    def off_the_board(self):
```



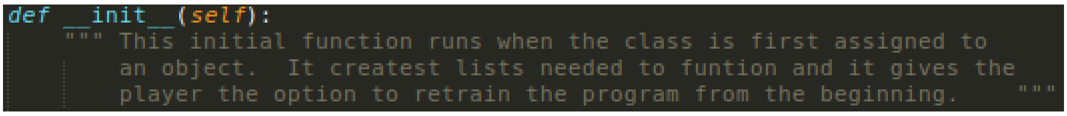
```
def __init__(self):
    """ This initial procedure creates the propeties of this cell          """
```



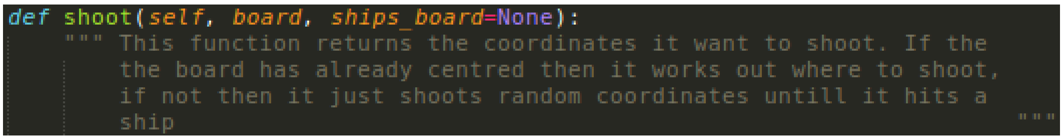
```
def off_the_board(self):
    """ If a cell is off the board then it has to be given the values
        that let it be treated accordingly. it makes it appear like a
        miss, but makes it immediatly unhidden, so the ai will learn
        not to shoot it                                                    """
```



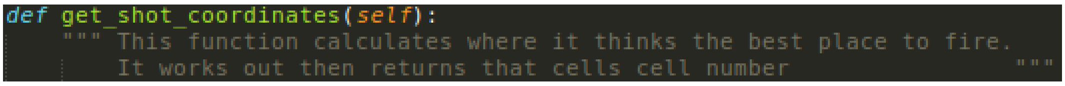
```
class AI():
    """this class is the the learning part of the program.  It is
        responsible for taking shots and learns from the consequences
        of that shot, with a system that weights each cell and chooses the
        one with the largest weight. If the move is good then it will
        increase the wight of the cell, if it isn't then the weight will
        decrease.  It is also where the file gets updated with then learned
        values.  The cells with have different values depending on the
        the current contens of the board.  All the possible board states are
        stored in the external file, 'layers.csv'.                         """
    def __init__(self): …

    def shoot(self, board, ships_board=None): …

    def get_shot_coordinates(self): …

    def update_file(self): …

    def get_coordinates(self, loc, ships_board): …

    def get_numerical_array(self): …
```



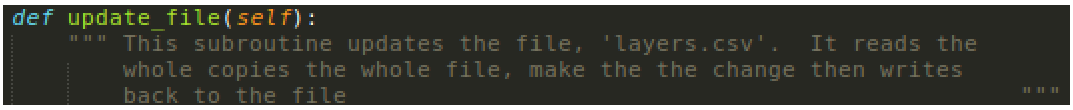
```
def __init__(self):
    """ This initial function runs when the class is first assigned to
        an object.  It createst lists needed to funtion and it gives the
        player the option to retrain the program from the beginning.   """
```



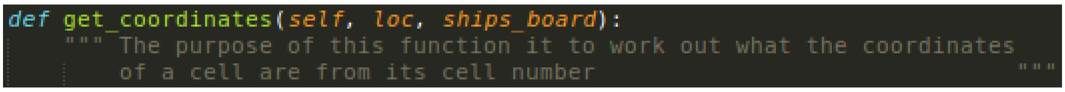
```
def shoot(self, board, ships_board=None):
    """ This function returns the coordinates it want to shoot. If the
        the board has already centred then it works out where to shoot,
        if not then it just shoots random coordinates untill it hits a
        ship                                                             """
```



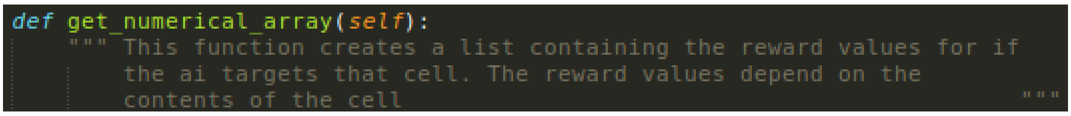
```
def get_shot_coordinates(self):
    """ This function calculates where it thinks the best place to fire.
        It works out then returns that cells cell number                 """
```



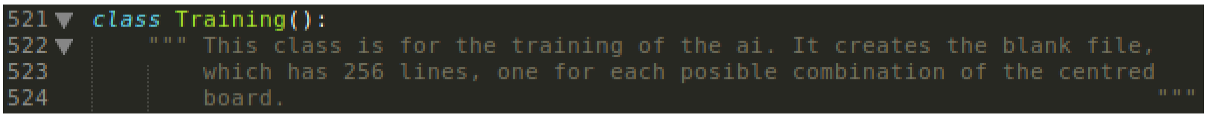
```
def update_file(self):
    """ This subroutine updates the file, 'layers.csv'.  It reads the
        whole copies the whole file, make the the change then writes
        back to the file                                                 """
```



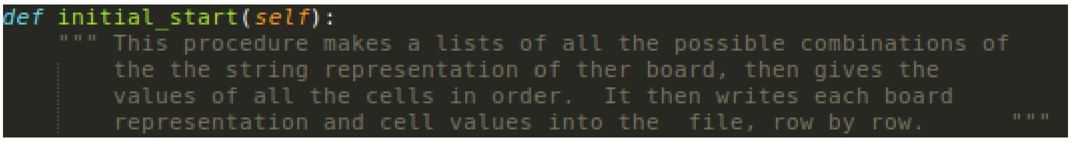
```
def get_coordinates(self, loc, ships_board):
    """ The purpose of this function it to work out what the coordinates
        of a cell are from its cell number                               """
```



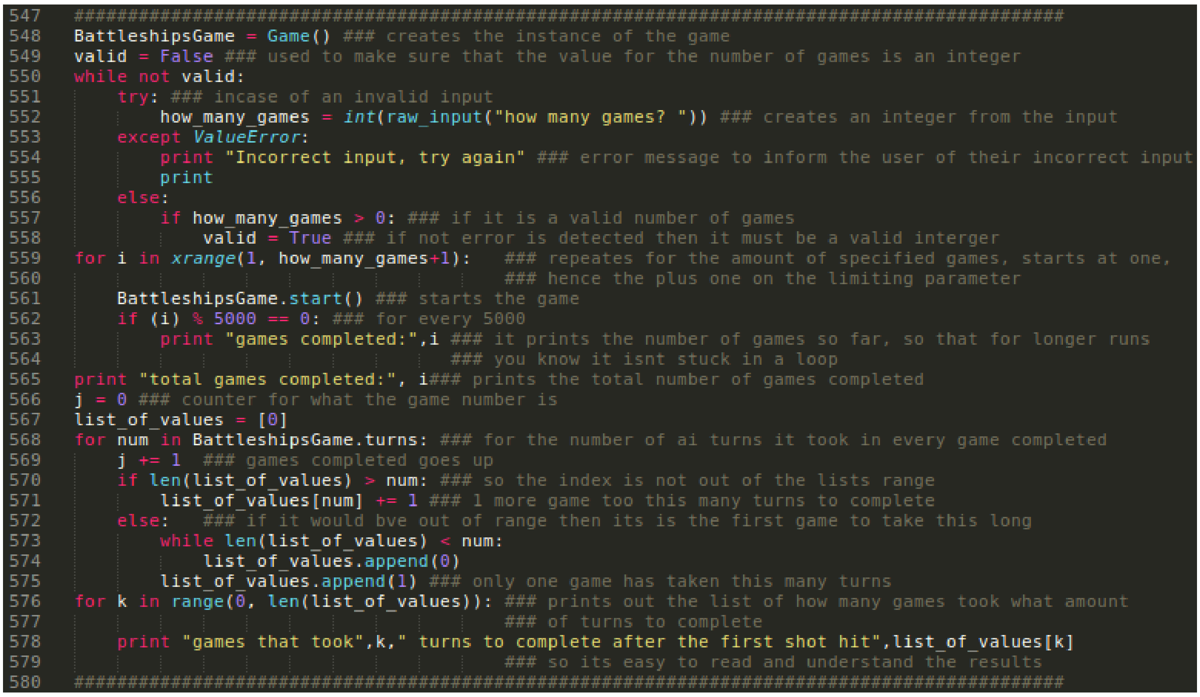
```
def get_numerical_array(self):
    """ This function creates a list containing the reward values for if
        the ai targets that cell. The reward values depend on the
        contents of the cell                                             """
```



```
class Training():
    """ This class is for the training of the ai. It creates the blank file,
        which has 256 lines, one for each posible combination of the centred
        board.                                                              """
```



```
def initial_start(self):
    """ This procedure makes a lists of all the possible combinations of
        the the string representation of ther board, then gives the
        values of all the cells in order.  It then writes each board
        representation and cell values into the  file, row by row.      """
```



```
############################################################################################
BattleshipsGame = Game() ### creates the instance of the game
valid = False ### used to make sure that the value for the number of games is an integer
while not valid:
    try: ### incase of an invalid input
        how_many_games = int(raw_input("how many games? ")) ### creates an integer from the input
    except ValueError:
        print "Incorrect input, try again" ### error message to inform the user of their incorrect input
        print
    else:
        if how_many_games > 0: ### if it is a valid number of games
            valid = True ### if not error is detected then it must be a valid interger
for i in xrange(1, how_many_games+1):   ### repeates for the amount of specified games, starts at one,
                                        ### hence the plus one on the limiting parameter
    BattleshipsGame.start() ### starts the game
    if (i) % 5000 == 0: ### for every 5000
        print "games completed:",i ### it prints the number of games so far, so that for longer runs
                                   ### you know it isnt stuck in a loop
print "total games completed:", i### prints the total number of games completed
j = 0 ### counter for what the game number is
list_of_values = [0]
for num in BattleshipsGame.turns: ### for the number of ai turns it took in every game completed
    j += 1  ### games completed goes up
    if len(list_of_values) > num: ### so the index is not out of the lists range
        list_of_values[num] += 1 ### 1 more game too this many turns to complete
    else:   ### if it would bve out of range then its is the first game to take this long
        while len(list_of_values) < num:
            list_of_values.append(0)
        list_of_values.append(1) ### only one game has taken this many turns
for k in range(0, len(list_of_values)): ### prints out the list of how many games took what amount
                                        ### of turns to complete
    print "games that took",k," turns to complete after the first shot hit",list_of_values[k]
                                        ### so its easy to read and understand the results
############################################################################################
```

Further descriptions to every piece of my code can be found in Appendix 1, and footage of it running can be found in Appendix 3.

The learning algorithm does take shots at the board. However, the size of the board does change and some of its shots are random. The reason that it is built like this is so that the program is within a reasonable scope for me, but allows me room to expand the program in the future. I have made it so that the shots taken on a board before the ship was uncovered were random, this is because those initial shots are not too important because there is a random element to it, as the player would not know where the ship were. However, as soon a ship was hit, then it would have to shoot in the immediate area of that ship. This is why the board gets shrunk to 3 by 3. Once the first hit has been made then the only reasonable place for it to fire would be in the immediate area of that initial hit. I focused on this part mainly because once is has leant to fire in the immediate area then I it can sink any length ship. My next move would be to make it so that the algorithm learns that if there is not ship in the only reasonable space, it should move to the other end of the ship and start firing there. After this the next step would be for it to learn that is moving in a straight line up and down a ship doesn't sink it, then what it thought was a ship is actually two ships placed together, and so it will have to learn how to deal with that.

I would expect that when changes were made to the code to display the game to the user, it was clear that there is ship placement, turns, shooting, ships being hit and sunk as well as boards. This shows that program does simulate elements from a game of battleships. The major difference is that the players don't take turns, but after changing my requirements I was no longer trying to achieve this. However, I was trying to get the board and ships to look like a game of battleships. Here I have failed because the board is only 4 by 4 rather than 10 by 10, and also because there is only one 2 by 1 ship, rather than the full set of ships, of lengths 5, 4, 3, 3 and 2. This is because I reduced the ship count to one ship of length one, so that it is easier for the AI to learn. This is so that is didn't have to worry about breaking down the ship size and alternating between the centred board and the full board. The full board was only made to 4 by 4, even though with very minor changes it can be any size, was so that the random firing at the start does not take too long.

Testing and evaluating

I tested the program as I was writing it, by here is the final testing of the program against my requirements. These are the requirements that I am testing against:

2. To have a learning program that learns to play elements of battleships
    - 2.1. To have the elements of the game to follow the standard rules of battleships
        - 2.1.1. Ships are placed on a board
        - 2.1.2. The ships are then shot at
        - 2.1.3. When a square on the board is shot, it is either a hit, when it hits a ship, or a miss.
        - 2.1.4. If all the cells that make up a shit get hit then the ship sinks
        - 2.1.5. The player wins when all the opponent’s ships are sunk
    - 2.2. For there to be one player controlled by the learning algorithm
        - 2.2.1. Have it calculate and fire shots at the ships
        - 2.2.2. Have it learn to become better at the game by, taking less turns to sink the ships as time goes on
        - 2.2.3. There to be no deterministic code to control firing
    - 2.3. To have randomly placed ships to represent the other player
    - 2.4. To have it be robust and efficient so it
        - 2.4.1. Doesn’t crash after incorrect input
        - 2.4.2. Can run thousands of times without crashing
        - 2.4.3. Uses minimal amount of storage and memory
        - 2.4.4. Can simulate thousands of games in a reasonable amount of time
            - 2.4.4.1.
        - 2.4.5. Doesn’t crash if file isn’t accessible
    - 2.5. To have it be capable of storing learning data
        - 2.5.1. Have it stored in a way that easy to read
        - 2.5.2. Have it stored in a way that easy to write
        - 2.5.3. Have it set up and reset the file
    - 2.6. Must have a basic user interface that
        - 2.6.1. Allows the user to choose to retrain the learning program
        - 2.6.2. Allows the user to choose how many games to loop through
        - 2.6.3. Displays data that can prove that the program is learning

Test of requirement: 2. To have a learning program that learns to play elements of battleships

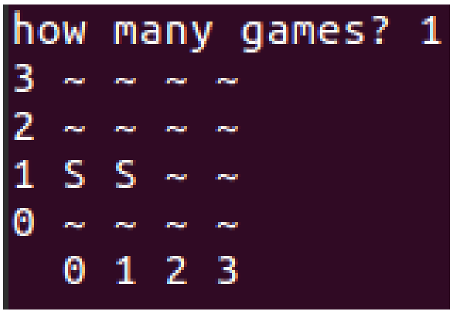
If the rests of the tests are successful, then the task has been completed and the overall goal is met. I set up each test to test each element for the requirements. I feel that separating the testing like this ensures that all the key parts of the program work, and then the white box testing ensured that key run as expected.

Test of requirement: 2.1. To have the elements of the game to follow the standard rules of battleships

I would expect that when changes were made to the code to display the game to the user, it was clear that there is ship placement, turns, shooting, ships being hit and sunk as well as boards. This shows that program does simulate elements from a game of battleships. The major difference is that the players don't take turns, but after changing my requirements I was no longer trying to achieve this. However, I was trying to get the board and ships to look like a game of battleships. Here I have failed because the board is only 4 by 4 rather than 10 by 10, and also because there is only one 2 by 1 ship, rather than the full set of ships, of lengths 5, 4, 3, 3 and 2. This is because I reduced the ship count to one ship of length one, so that it is easier for the AI to learn. This is so that is didn't have to worry about breaking down the ship size and alternating between the centred board and the full board. The full board was only made to 4 by 4, even though with very minor changes it can be any size, was so that the random firing at the start does not take too long.

Test of requirement: 2.1.1. Ships are placed on a board

One shit should get placed on the board, meaning requirement 2.1.1 has been partially reached. While a ship is placed on the board, it is not "ships", as stated in the requirements. However, in the placing of this ship, it lays down the frame works to shrink any ship of any size. The screen shots in the upcoming section, Test summary 2.1 requirements and sub requirements, show a ship being placed at the start if the game, and be un hiding the ships, by changing the value assigned at line 362 and removing the '#' at line 39, you can see the placement of the ship on the board, as seen below.



```
how many games? 1
3 ~ ~ ~ ~
2 ~ ~ ~ ~
1 S S ~ ~
0 ~ ~ ~ ~
  0 1 2 3
```

In the original game created before the learning program, multiple ships could be placed on a board, and the boards and ships could be any size. This can be seen at time 0:00:00 in Appendix 2, with the creation of the ships. Also by reading through the earlier section, Creating Battleships, there are screen shots showing and explaining the ship placement.

Test of requirement: 2.1.2. The ships are then shot at

Shots should be fired at the board every turn, and at a ship when its location becomes known. The screen shots in the upcoming section, Test summary 2.1 requirements and sub requirements, show shots are fired at the board throughput the running of the code. Once the first hit connected the learning program took over the firing and sank the ship in one shot. This is sufficient evidence that shots are fired at the ship and that it is tested to work as expected.

Test of requirement: 2.1.3. When a square on the board is shot, it is either a hit, when it hits a ship, or a miss.

On the game board; - represents a square that is yet to fire at, ~ represents a miss, and S, or whatever the first letter in the name of the ship, represents a hit. The screen shots in section, Test summary 2.1 requirements and sub requirements, show when the board was shot the shots showed up as misses, ~, or as hits, marked as an 'S'. This shows that the requirement was reached and that it works as expected.

Test of requirement: 2.1.4. If all the cells that make up a shit get hit then the ship sinks

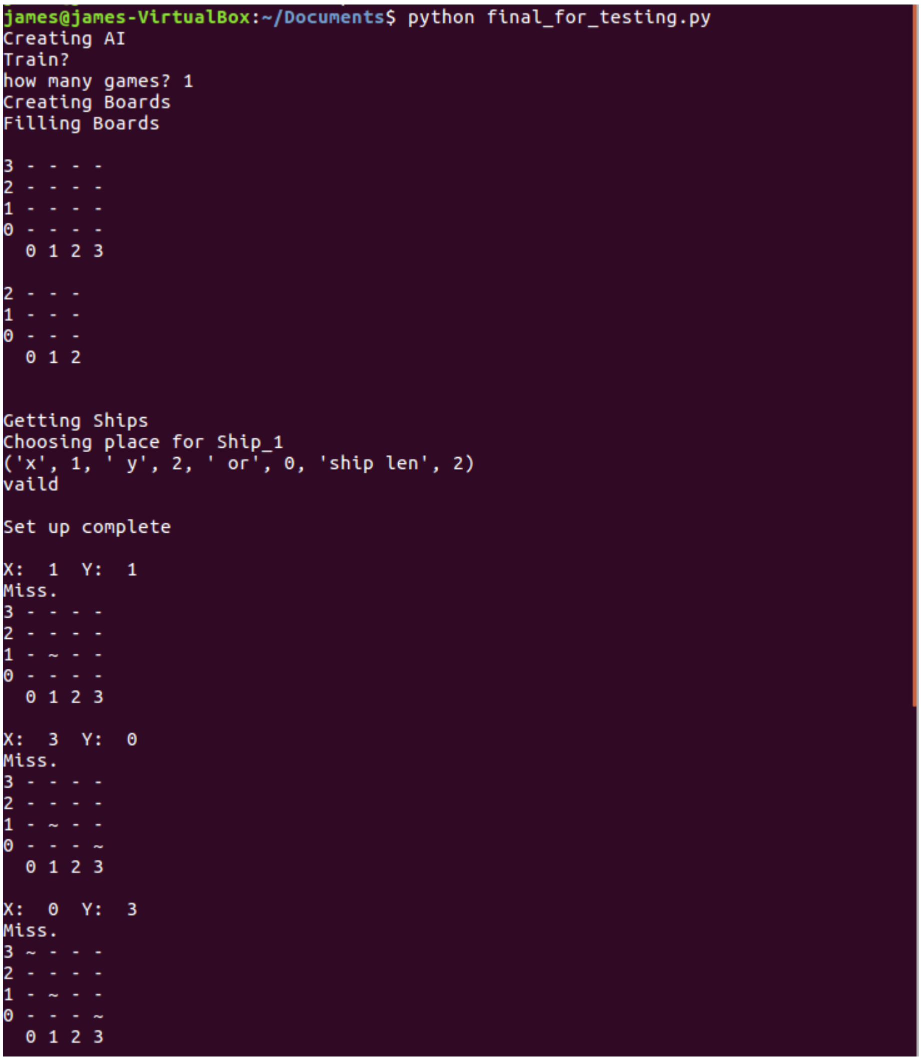
When all of the tiles of the ship are hit, the ship should sink. As seen in the screen shots in section, Test summary 2.1 requirements and sub requirements, after the second hit is announced the ship sunk message appears, showing that the ship has sunk. This proves that this element of the program works as intended, and therefore that this requirement has been met.

Test of requirement: 2.1.5. The player wins when all the opponent's ships are sunk

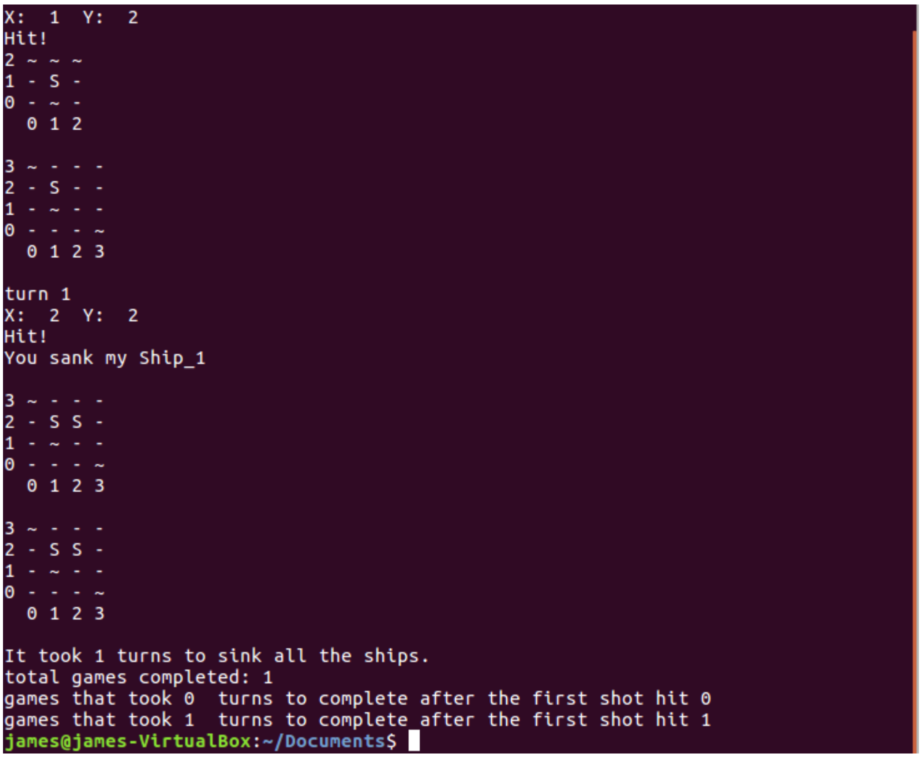
When all the ships have been sunk, the player should win the game. As seen the screen shots in the next section, this requirement has been met. When the ship was sunk, it displays two instances of the board, before allowing the learning algorithm to win, displaying that all ships were sunk and therefore the game has ended.

Test summary 2.1 requirements and sub requirements

These screen shots were produced when with the minor modification of removing the '#' at the start of lines; 15,24, 28, 31, 32, 33, 34, 35, 38, 41, 42, 43, 56, 57, 58, 68, 71, 74, 125, 139, 172, 173, 178, 252, 418, 440, 505 (which are all just related to how the game is displayed); in the final program.



```
james@james-VirtualBox:~/Documents$ python final_for_testing.py
Creating AI
Train?
how many games? 1
Creating Boards
Filling Boards

3 - - - -
2 - - - -
1 - - - -
0 - - - -
  0 1 2 3

2 - - -
1 - - -
0 - - -
  0 1 2


Getting Ships
Choosing place for Ship_1
('x', 1, ' y', 2, ' or', 0, 'ship len', 2)
vaild

Set up complete

X:  1  Y:  1
Miss.
3 - - - -
2 - - - -
1 - ~ - -
0 - - - -
  0 1 2 3

X:  3  Y:  0
Miss.
3 - - - -
2 - - - -
1 - ~ - -
0 - - - ~
  0 1 2 3

X:  0  Y:  3
Miss.
3 ~ - - -
2 - - - -
1 - ~ - -
0 - - - ~
  0 1 2 3
```



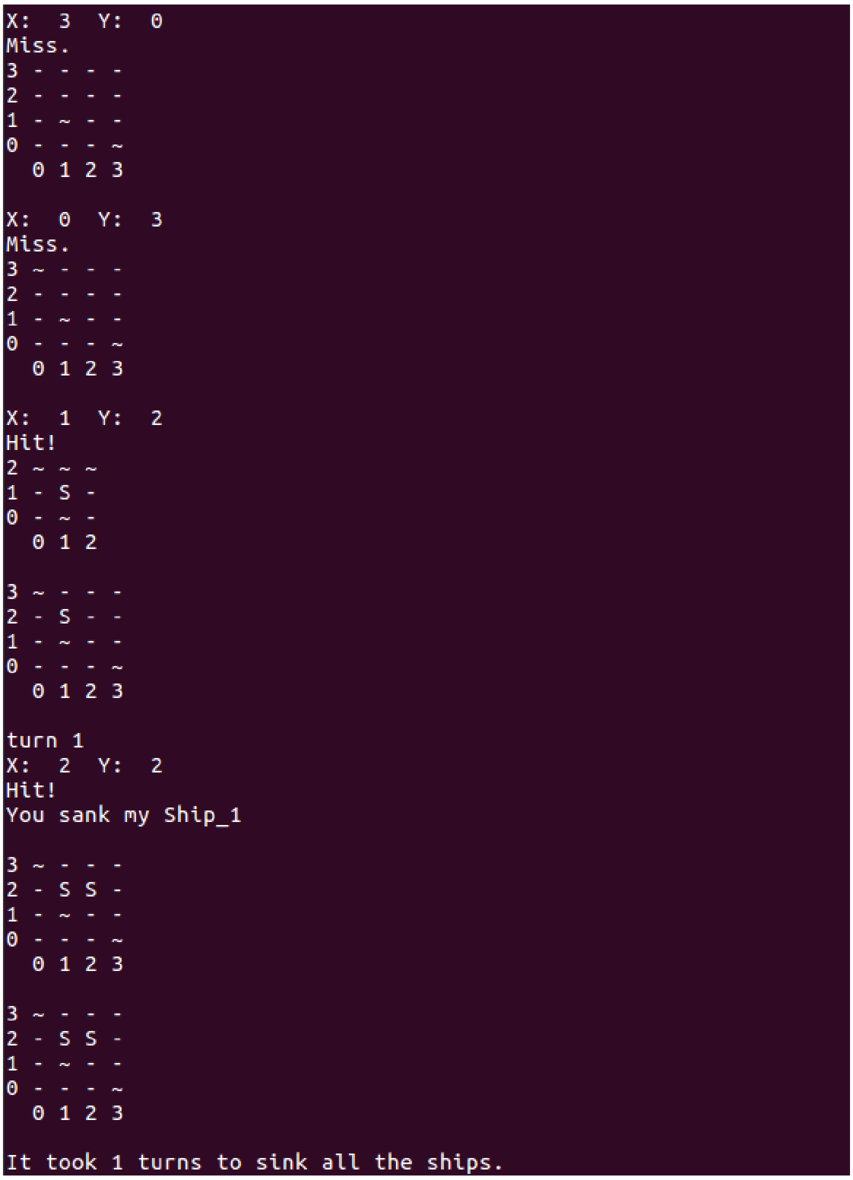
```
X:  1  Y:  2
Hit!
2 ~ ~ ~
1 - S -
0 - ~ -
  0 1 2

3 ~ - - -
2 - S - -
1 - ~ - -
0 - - - ~
  0 1 2 3

turn 1
X:  2  Y:  2
Hit!
You sank my Ship_1

3 ~ - - -
2 - S S -
1 - ~ - -
0 - - - ~
  0 1 2 3

3 ~ - - -
2 - S S -
1 - ~ - -
0 - - - ~
  0 1 2 3

It took 1 turns to sink all the ships.
total games completed: 1
games that took 0  turns to complete after the first shot hit 0
games that took 1  turns to complete after the first shot hit 1
james@james-VirtualBox:~/Documents$ 
```

Test of requirement: 2.2. For there to be one player controlled by the learning algorithm

The shooting player is controlled by a learning algorithm. While it only learns some aspects of the game, it is still a learning algorithm that controls the shots.

Test of requirement: 2.2.1. Have it fire shots at the ships

The learning algorithm does take shots at the board. However, the size of the board does change and some of its shots are random. The reason that it is built like this is so that the program is within a reasonable scope for me, but allows me room to expand the program in the future. I have made it so that the shots taken on a board before the ship was uncovered were random, this is because those initial shots are not too important because there is a random element to it, as the player would not know where the ship were. However, as soon a ship was hit, then it would have to shoot in the immediate area of that ship. Overall it does shoot at the ship, as seen in the screen shots of the console displayed game. X being the x-coordinate of the shot and Y being the y-coordinate.



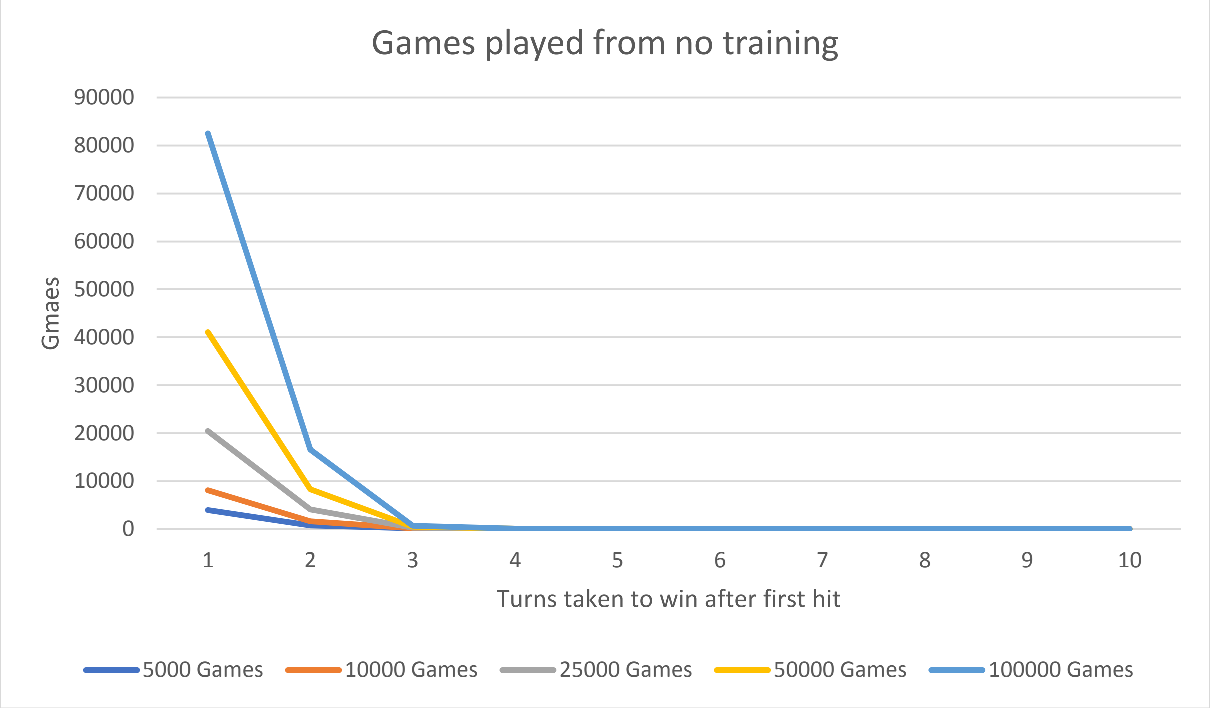
```
X:  3  Y:  0
Miss.
3 - - - -
2 - - - -
1 - ~ - -
0 - - - ~
  0 1 2 3

X:  0  Y:  3
Miss.
3 ~ - - -
2 - - - -
1 - ~ - -
0 - - - ~
  0 1 2 3

X:  1  Y:  2
Hit!
2 ~ ~ ~
1 - S -
0 - ~ -
  0 1 2

3 ~ - - -
2 - S - -
1 - ~ - -
0 - - - ~
  0 1 2 3

turn 1
X:  2  Y:  2
Hit!
You sank my Ship_1

3 ~ - - -
2 - S S -
1 - ~ - -
0 - - - ~
  0 1 2 3

3 ~ - - -
2 - S S -
1 - ~ - -
0 - - - ~
  0 1 2 3

It took 1 turns to sink all the ships.
```

Test of requirement: 2.2.2. Taking less turns to sink the ships as time goes on

Data collected from the output of the program shows that it gets much better at hitting ships. It can be seen clearly as it learns. By taking the values printed in the console at the end of the game, and graphing them with an application like Microsoft Excel creates clear visual evidence. I used a numpy seed for my random numbers. This meant that it produced the same random numbers each time, this made it easy to test, because I could compare the program with the exact same inputs and number generations, meaning that there were no other factors that could affect the program. By doing this only the programs ability to learn is being tested and assessed.

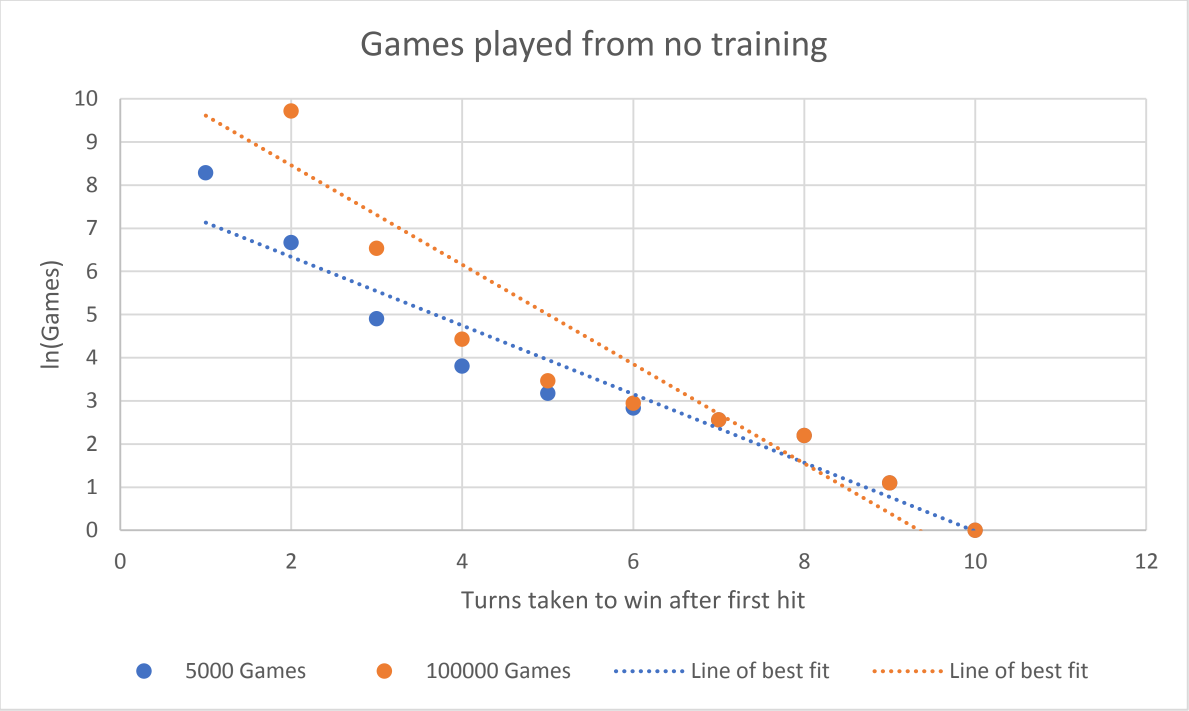
This first graph shows how the program performed after being trained from nothing. It compares data from when 5000 games were played, up to when 100000 games were played. The problems with this is that the range in numbers is so great that you cannot see in detail. However, from the table you can see how not all the values were increase even in proportion to the total number of games played. If it wasn't learning then I would expect the values to double when the games played doubled, however it is clear that they don't.

Turns taken	5000 Games	10000 Games	25000 Games	50000 Games	100000 Games
1	3965	8104	20456	41049	82555
2	788	1593	4102	8306	16593
3	135	180	307	498	691
4	45	51	62	72	84
5	24	28	29	31	32
6	17	18	18	18	19
7	13	13	13	13	13
8	9	9	9	9	9
9	3	3	3	3	3
10	1	1	1	1	1



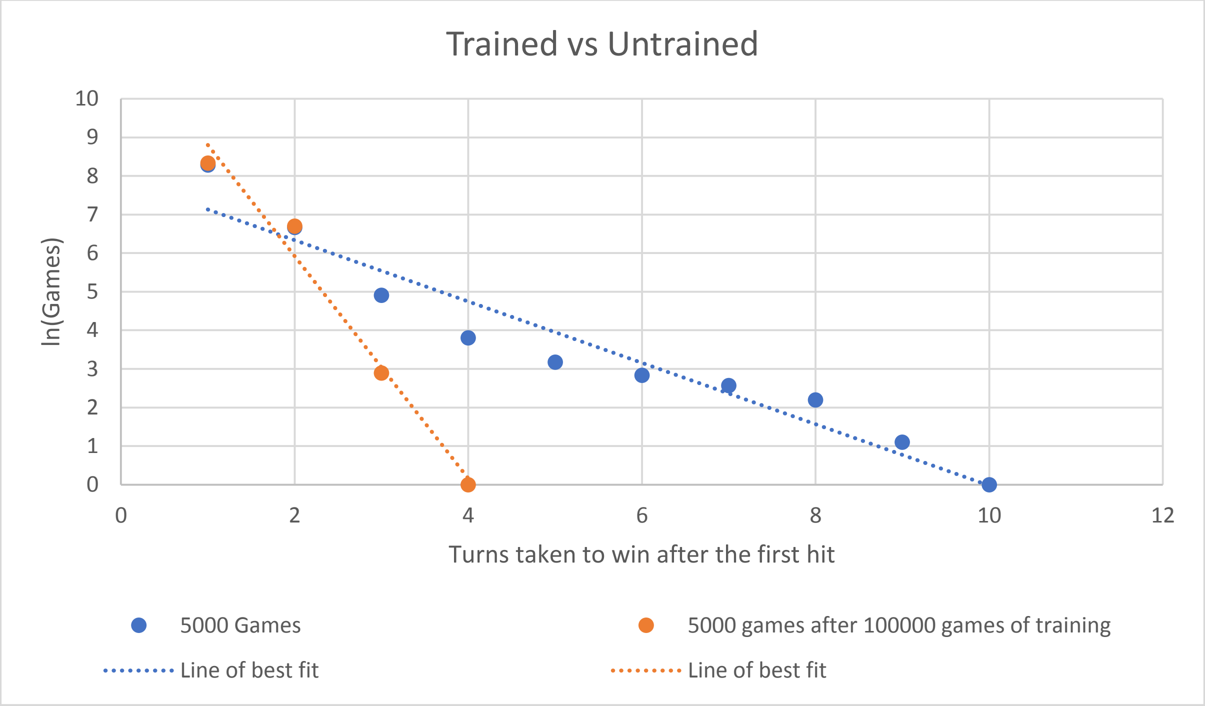
To represent the change in values better I decided to use the natural log of the games played, as this should give me a straight line, so the gradient is easy to compare. The steeper gradient the better it is at shooting. It also brought the scale down so they were easier to compare in terms of the scale and range of values. From the graph, you can see that after more games the overall line of best fit gets steeper, but it is still close, I decided this was because the line plotted still included the earlier results.

Turns taken	ln(Games) for 5000 games played from no training	ln(Games) for 100000 games played from no training
1	8.285261134	11.32122002
2	6.66949809	9.716736199
3	4.905274778	6.538139824
4	3.80666249	4.430816799
5	3.17805383	3.465735903
6	2.833213344	2.944438979
7	2.564949357	2.564949357
8	2.197224577	2.197224577
9	1.098612289	1.098612289
10	0	0

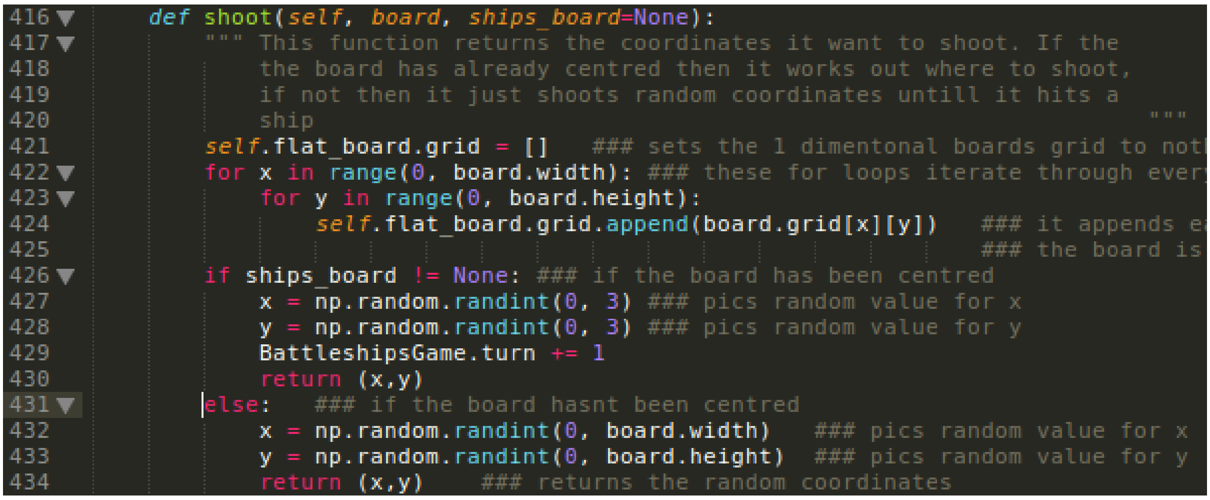


The next graph here, shows how long it took to compete 5000 games for when the AI had no training, and for after the AI had 100000 games of training. It is clear here how the gradient of the trained AI is much steeper than the untrained AI. The correlations in the data are clear. There is a distinct difference in the lines which shows the program has learnt and has become better at shooting.

Turns taken	5000 games played from no training	5000 games after 100000 games of training	ln(Games) ;5000 games played from no training	ln(Games) ;5000 games after 100000 games of training
1	3965	4173	8.285261134	8.336390481
2	788	809	6.66949809	6.695798917
3	135	18	4.905274778	2.890371758
4	45	0	3.80666249	0
5	24	0	3.17805383	0
6	17	0	2.833213344	0
7	13	0	2.564949357	0
8	9	0	2.197224577	0
9	3	0	1.098612289	0
10	1	0	0	0



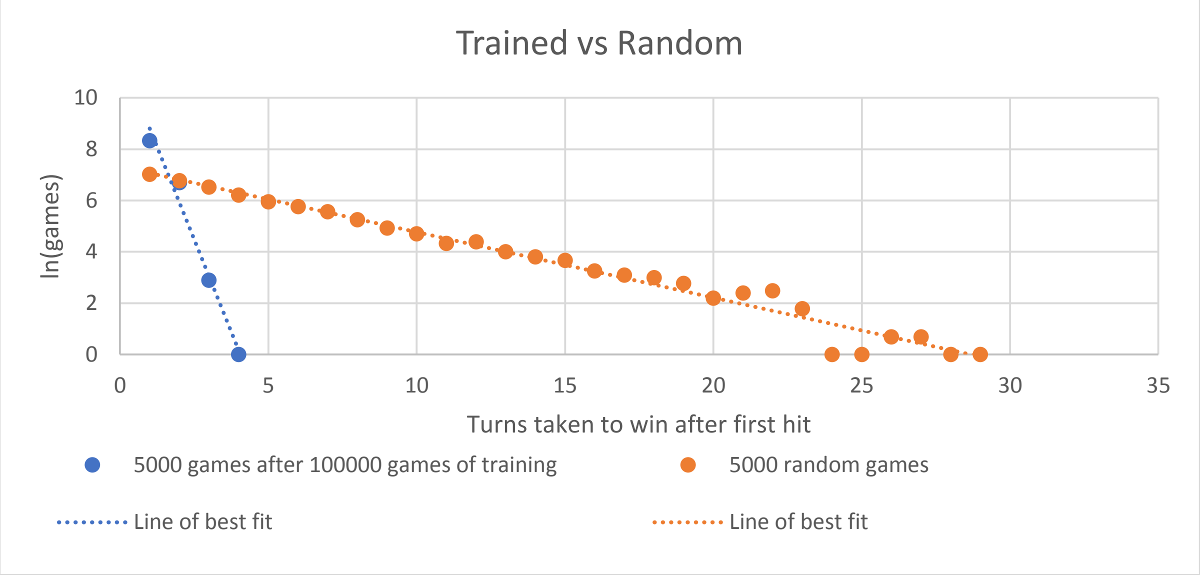
It is clear here how the gradient of the trained AI is much steeper than the untrained AI. I feel that this is sufficient evidence that the program is leaning, as it is clear that the amount of turns it take to win the game. The untrained program has a gradient of -0.795, while the trained program has a gradient of -2.882, both to three decimal places. That's a very large difference and is significant evidence that it learnt and improved at the game.



```
    def shoot(self, board, ships_board=None):
        """ This function returns the coordinates it want to shoot. If the
            the board has already centred then it works out where to shoot,
            if not then it just shoots random coordinates untill it hits a
            ship                                                          """
        self.flat_board.grid = []   ### sets the 1 dimentonal boards grid to not
        for x in range(0, board.width): ### these for loops iterate through ever
            for y in range(0, board.height):
                self.flat_board.grid.append(board.grid[x][y])   ### it appends e
                                                                ### the board is
        if ships_board != None: ### if the board has been centred
            x = np.random.randint(0, 3) ### pics random value for x
            y = np.random.randint(0, 3) ### pics random value for y
            BattleshipsGame.turn += 1
            return (x,y)
        else:   ### if the board hasnt been centred
            x = np.random.randint(0, board.width)   ### pics random value for x
            y = np.random.randint(0, board.height)  ### pics random value for y
            return (x,y)    ### returns the random coordinates
```

The screen shot above shows the changes I made to the code to render the AI redundant and produces completely random coordinates. This is so I could then compare the gradients from the learnt program and the randomly firing one.

Turns taken	5000 games after 100000 games of training	ln(games) ;5000 games after 100000 games of training	5000 random games	ln(games) ;5000 random games
1	4173	8.33639	1119	7.020191
2	809	6.695799	875	6.774224
3	18	2.890372	677	6.517671
4	0	0	500	6.214608
5			386	5.955837
6			318	5.762051
7			260	5.560682
8			191	5.252273
9			138	4.927254
10			110	4.70048
11			76	4.330733
12			81	4.394449
13			55	4.007333
14			45	3.806662
15			39	3.663562
16			26	3.258097
17			22	3.091042
18			20	2.995732
19			16	2.772589
20			9	2.197225
21			11	2.397895
22			12	2.484907
23			6	1.791759
24			1	0
25			1	0
26			2	0.693147
27			2	0.693147
28			1	0
29			1	0



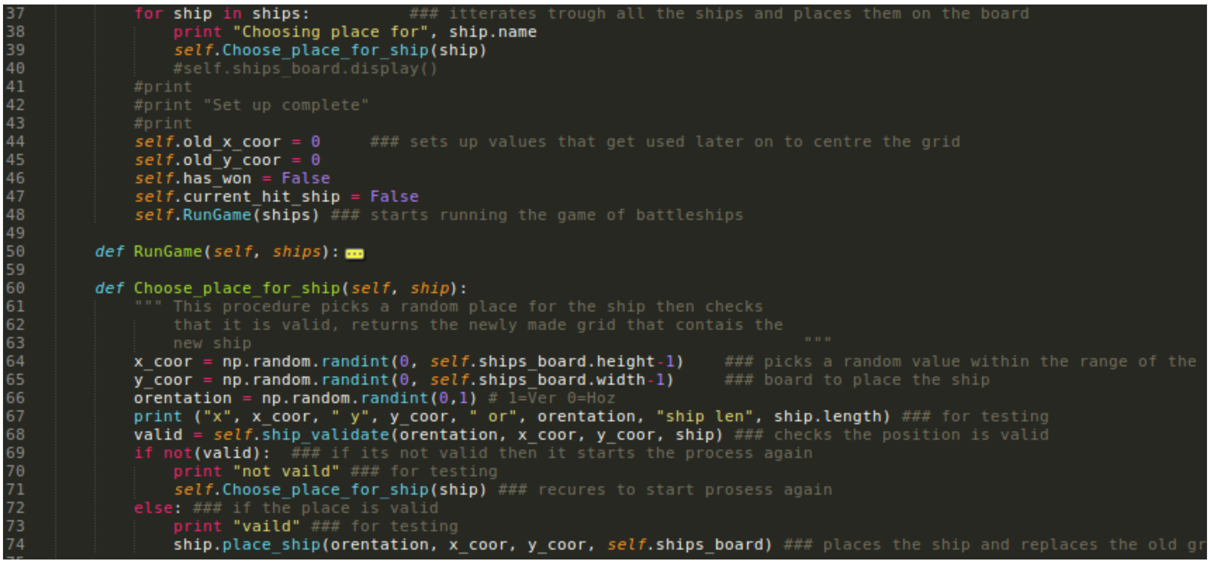
The gradient of the trend line for the random games is approximately -0.255, the untrained program has a gradient of -0.795, while the trained program has a gradient of -2.882, all to three decimal places. This is clear evidence that the program is much better than a random system, and the untrained system.

Test of requirement: 2.2.3. There to be no deterministic code to control firing

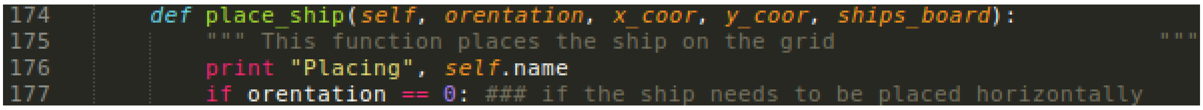
By looking at the algorithms used it is easy to see that no deterministic firing has been used when the program is learning, however the learning algorithm does not control where the shots go before the first hit. My reasons behind this are in the Design of the Final Program section of this booklet. It can be argued the random nature of the firing does not affect the programs ability to learn, but they are still shots that are out of control of the learning algorithm. However, in creating the program in the is way it can be expanded in to learn to sink a ship of any size, and can then be implemented with an algorithm to learn when shooting in between the sinking of ships. This could learn things like where the ship is more likely to be placed.

Test of requirement: 2.3. To have randomly placed ships to represent the other player

By adding a line of code to print the out puts of the ships placement to the console terminal, it is clear to see that the "opponent" is placing the ship in a random location each time. I only made changes to 5 lines; 38, 67, 70, 73, 176; and this was just removing the '#' at the start of the line, so it was no longer a comment, as seen below, which and be compared to the raw code in Appendix 1.

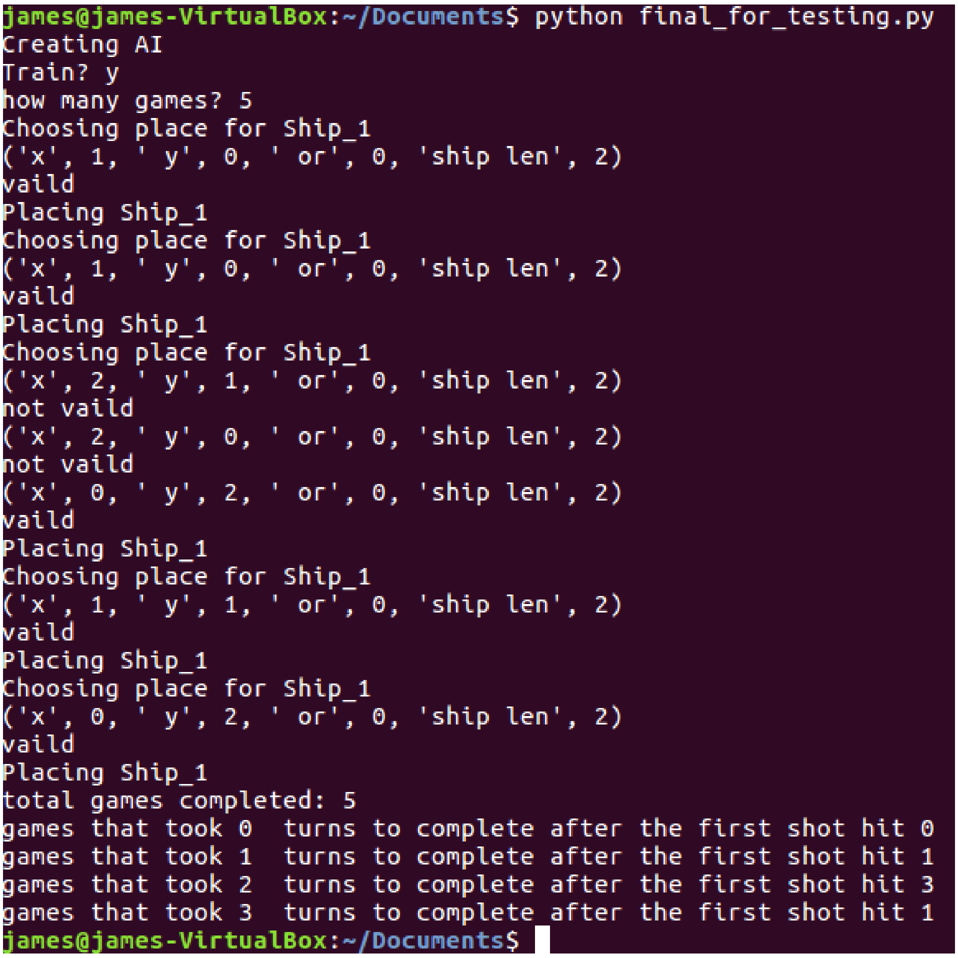


```
        for ship in ships:          ### itterates trough all the ships and places them on the board
            print "Choosing place for", ship.name
            self.Choose_place_for_ship(ship)
            #self.ships_board.display()
        #print
        #print "Set up complete"
        #print
        self.old_x_coor = 0     ### sets up values that get used later on to centre the grid
        self.old_y_coor = 0
        self.has_won = False
        self.current_hit_ship = False
        self.RunGame(ships) ### starts running the game of battleships

    def RunGame(self, ships):

    def Choose_place_for_ship(self, ship):
        """ This procedure picks a random place for the ship then checks
            that it is valid, returns the newly made grid that contais the
            new ship                                                    """
        x_coor = np.random.randint(0, self.ships_board.height-1)    ### picks a random value within the range of the
        y_coor = np.random.randint(0, self.ships_board.width-1)     ### board to place the ship
        orentation = np.random.randint(0,1) # 1=Ver 0=Hoz
        print ("x", x_coor, " y", y_coor, " or", orentation, "ship len", ship.length) ### for testing
        valid = self.ship_validate(orentation, x_coor, y_coor, ship) ### checks the position is valid
        if not(valid):  ### if its not valid then it starts the process again
            print "not vaild" ### for testing
            self.Choose_place_for_ship(ship) ### recures to start prosess again
        else: ### if the place is valid
            print "vaild" ### for testing
            ship.place_ship(orentation, x_coor, y_coor, self.ships_board) ### places the ship and replaces the old gr
```



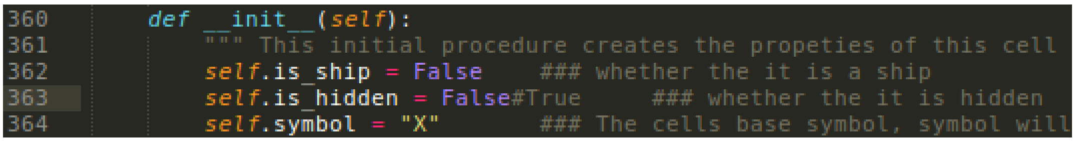
```
    def place_ship(self, orentation, x_coor, y_coor, ships_board):
        """ This function places the ship on the grid                                     """
        print "Placing", self.name
        if orentation == 0: ### if the ship needs to be placed horizontally
```

This is representative of another player, who would also place the ships randomly, or there would be thought behind the process, but that is insignificant due to the wide range of possibilities.

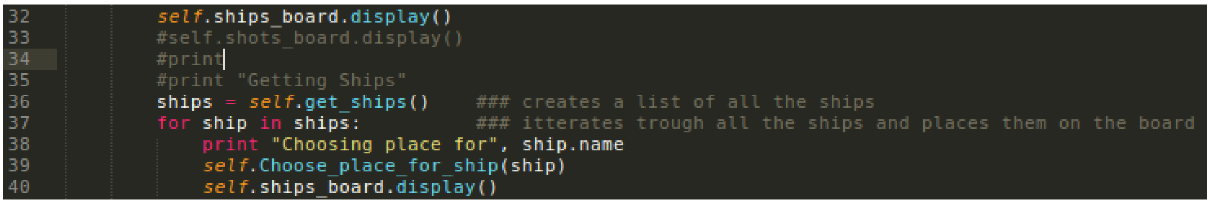


```
james@james-VirtualBox:~/Documents$ python final_for_testing.py
Creating AI
Train? y
how many games? 5
Choosing place for Ship_1
('x', 1, ' y', 0, ' or', 0, 'ship len', 2)
vaild
Placing Ship_1
Choosing place for Ship_1
('x', 1, ' y', 0, ' or', 0, 'ship len', 2)
vaild
Placing Ship_1
Choosing place for Ship_1
('x', 2, ' y', 1, ' or', 0, 'ship len', 2)
not vaild
('x', 2, ' y', 0, ' or', 0, 'ship len', 2)
not vaild
('x', 0, ' y', 2, ' or', 0, 'ship len', 2)
vaild
Placing Ship_1
Choosing place for Ship_1
('x', 1, ' y', 1, ' or', 0, 'ship len', 2)
vaild
Placing Ship_1
Choosing place for Ship_1
('x', 0, ' y', 2, ' or', 0, 'ship len', 2)
vaild
Placing Ship_1
total games completed: 5
games that took 0  turns to complete after the first shot hit 0
games that took 1  turns to complete after the first shot hit 1
games that took 2  turns to complete after the first shot hit 3
games that took 3  turns to complete after the first shot hit 1
james@james-VirtualBox:~/Documents$
```

As the outputs above show, there are many places the ship has been placed, and the places that have been deemed invalid. The notation I used is: 'x' for the X-coordinate, 'y' for the Y-coordinate, 'or' is the orientation of the ship (where 1 is vertical and 0 is horizontal) and 'ship len' is the length of the ship. If is set the cells to unhidden, by changing the value set at line 363, from True to False, and removed the '#' from the start of lines 32, 40.

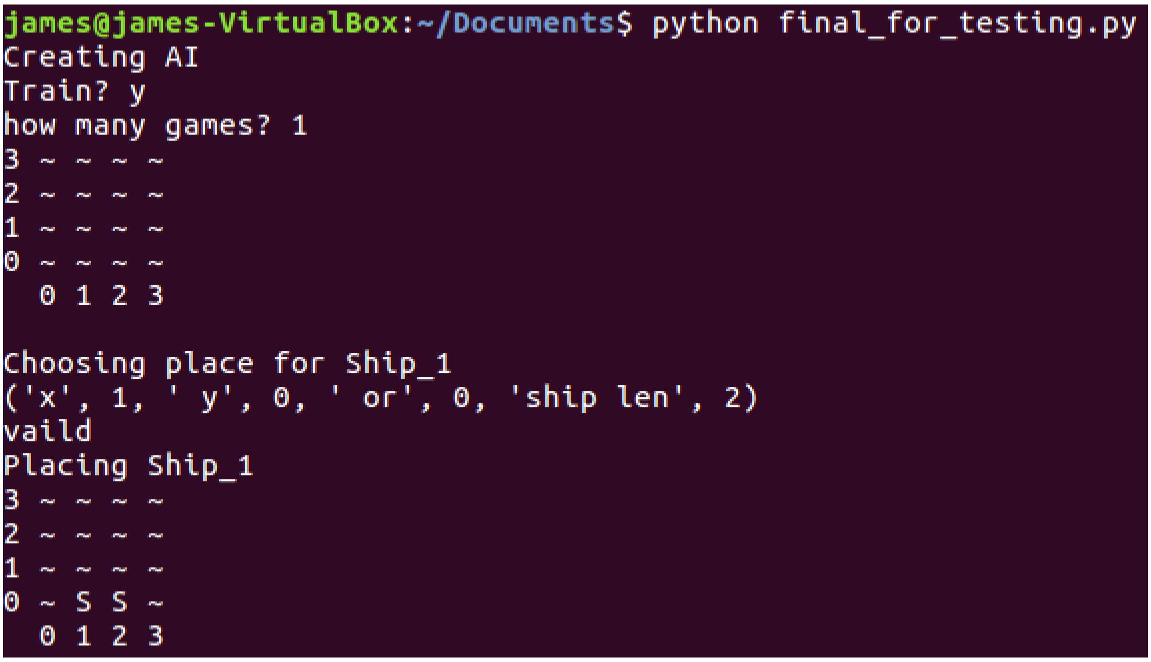


```
360     def __init__(self):
361         """ This initial procedure creates the propeties of this cell
362         self.is_ship = False     ### whether the it is a ship
363         self.is_hidden = False#True      ### whether the it is hidden
364         self.symbol = "X"        ### The cells base symbol, symbol will
```



```
32          self.ships_board.display()
33          #self.shots_board.display()
34          #print
35          #print "Getting Ships"
36          ships = self.get_ships()    ### creates a list of all the ships
37          for ship in ships:          ### itterates trough all the ships and places them on the board
38              print "Choosing place for", ship.name
39              self.Choose_place_for_ship(ship)
40              self.ships_board.display()
```

By displaying the board can also be displayed, which shows its place on the board, as seen below.



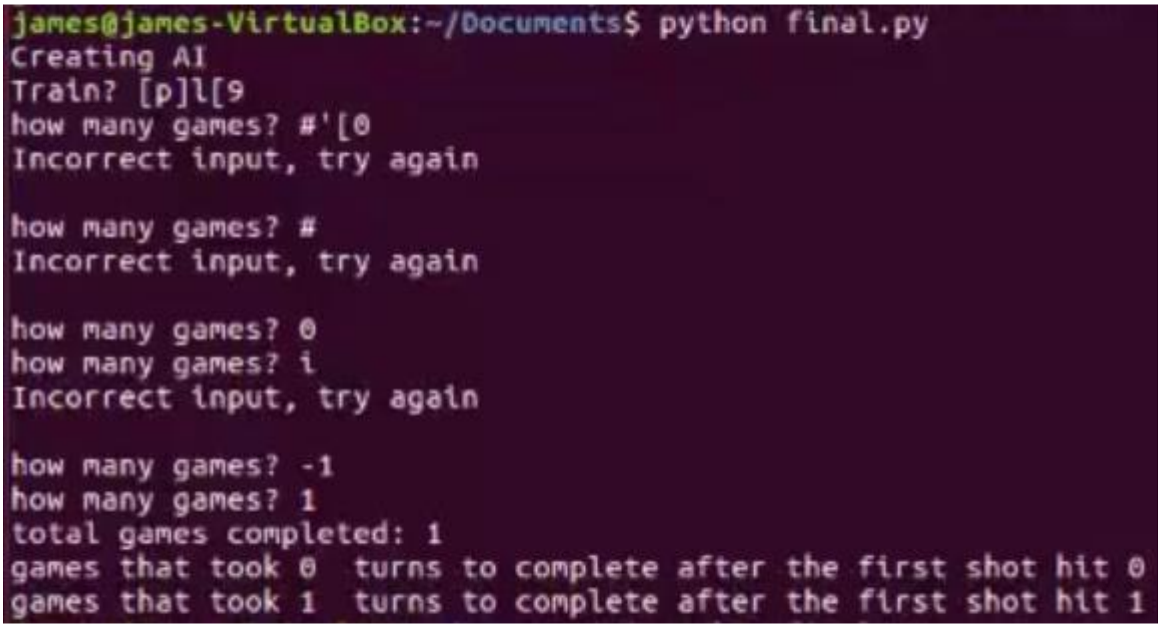
```
james@james-VirtualBox:~/Documents$ python final_for_testing.py
Creating AI
Train? y
how many games? 1
3 ~ ~ ~ ~
2 ~ ~ ~ ~
1 ~ ~ ~ ~
0 ~ ~ ~ ~
  0 1 2 3

Choosing place for Ship_1
('x', 1, ' y', 0, ' or', 0, 'ship len', 2)
vaild
Placing Ship_1
3 ~ ~ ~ ~
2 ~ ~ ~ ~
1 ~ ~ ~ ~
0 ~ S S ~
  0 1 2 3
```

Test of requirement: 2.4. To have it be robust and efficient

For the program to be both robust and efficient, it must meet all to the requirements in this section, which it dose. This means that the code has been tested to a sufficient amount of testing against its robustness and efficiency, and has passed all the tests.

Test of requirement: 2.4.1. Doesn't crash after incorrect input

To ensure the program doesn't crash after an invalid input, all the user inputs are checked before the program progresses. This validation makes sure the inputs are in the correct character set and are a valid range of values. The input for if the user wants to train the program from the start or retry opening a file, there is only on accepted character ‘y’. This means all other inputs are accepted as not wanting to do what was offered. This is a very simple way to combat incorrect inputs and isn’t the most user friendly, but as it is an investigation then user input was not a high priority. For registering how many games are wanted to be played is slightly harder to validate. This is because it can only accept integer values, and those values must be positive. When tested with multiple incorrect inputs it rejected them as expected, as seen below, or between 0:06:40 and 0:07:24 in Appendix 3

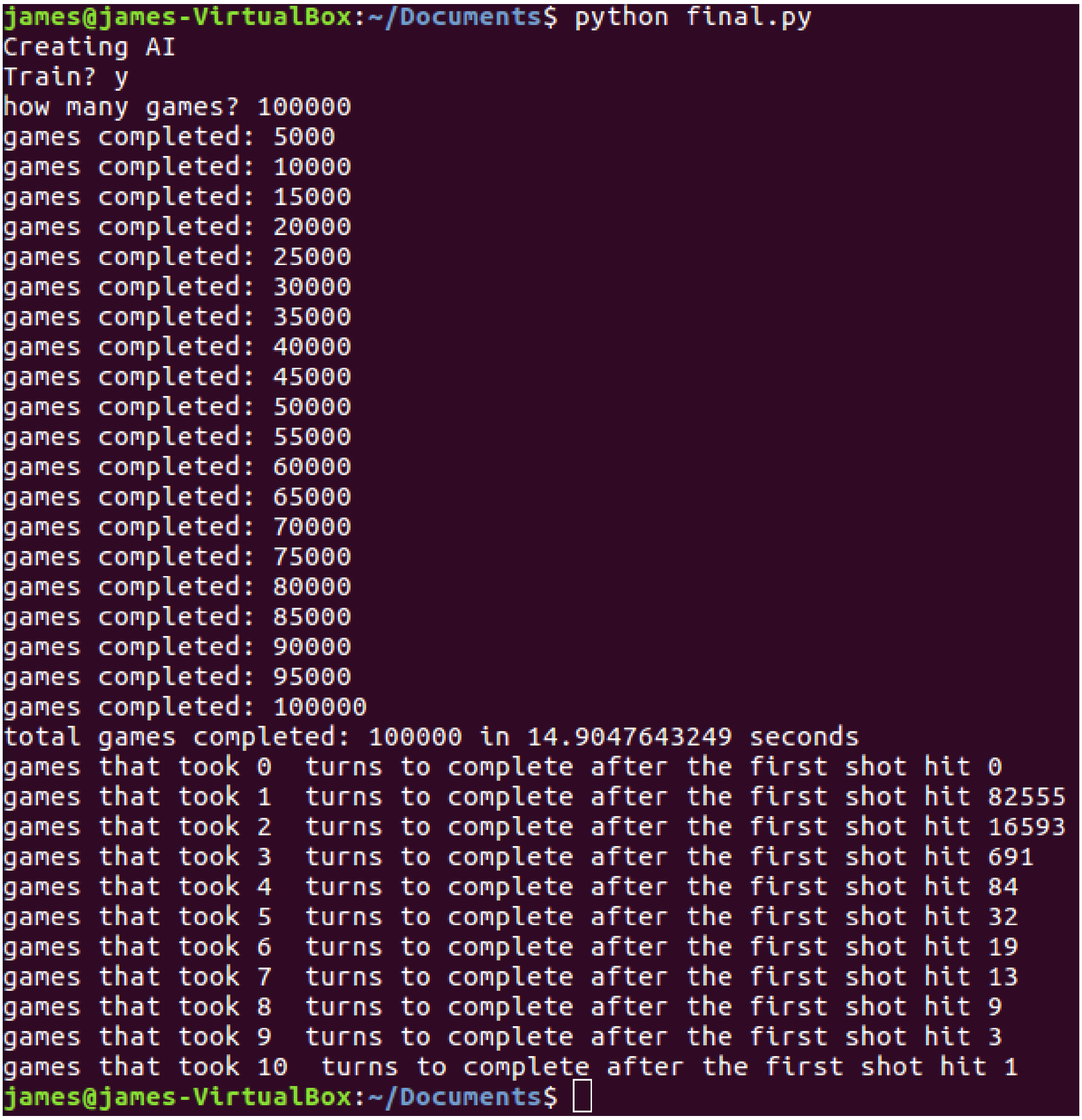


```
james@james-VirtualBox:~/Documents$ python final.py
Creating AI
Train? [p]l[9
how many games? #'[0
Incorrect input, try again

how many games? #
Incorrect input, try again

how many games? 0
how many games? i
Incorrect input, try again

how many games? -1
how many games? 1
total games completed: 1
games that took 0  turns to complete after the first shot hit 0
games that took 1  turns to complete after the first shot hit 1
```

The input for whether to retrain was counted as not to, as expected, and only the correct input, of ‘1’, was accepted. Inputs that are not integers are rejected, with an error message, and integers that were lower than one, and therefore invalid, are rejected but not messages are displayed, because I don’t believe it is necessary to have a message. This is because there would be no reason for the user to put in a less than one value, however a non-integer value could be entered from a miss reading of the prompt message.

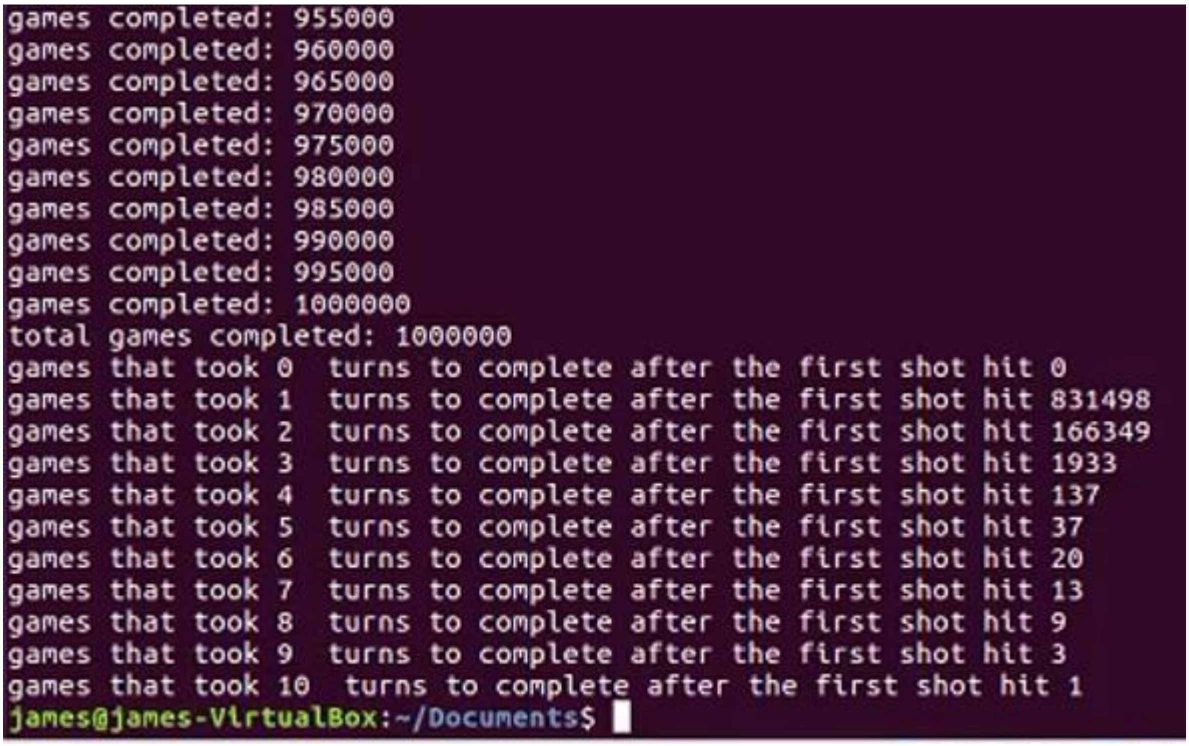
Test of requirement: 2.4.2. Can run thousands of times without crashing

It is crucial that the game can run thousands of times without crashing. When I first started testing I would only test for 5000 turns, but I started testing the code for more games when the game was in a stable state. I can get the program to run for a million games, without crashing. Pictured below is the console output from a 100000 game run through.



```
james@james-VirtualBox:~/Documents$ python final.py
Creating AI
Train? y
how many games? 100000
games completed:  5000
games completed:  10000
games completed:  15000
games completed:  20000
games completed:  25000
games completed:  30000
games completed:  35000
games completed:  40000
games completed:  45000
games completed:  50000
games completed:  55000
games completed:  60000
games completed:  65000
games completed:  70000
games completed:  75000
games completed:  80000
games completed:  85000
games completed:  90000
games completed:  95000
games completed:  100000
total games completed: 100000 in 14.9047643249 seconds
games that took 0  turns to complete after the first shot hit 0
games that took 1  turns to complete after the first shot hit 82555
games that took 2  turns to complete after the first shot hit 16593
games that took 3  turns to complete after the first shot hit 691
games that took 4  turns to complete after the first shot hit 84
games that took 5  turns to complete after the first shot hit 32
games that took 6  turns to complete after the first shot hit 19
games that took 7  turns to complete after the first shot hit 13
games that took 8  turns to complete after the first shot hit 9
games that took 9  turns to complete after the first shot hit 3
games that took 10  turns to complete after the first shot hit 1
james@james-VirtualBox:~/Documents$ 
```

From what I have seen the game can handle unlimited games, the way that it is programmed means that by running the game thousands of times would not use up any more memory that only playing through on game. This is mostly because the game is all a class, meaning that no new instances of the game have to be made just the same function being called over and over. The reason that I did not leave it to run longer is the time it took to run. The one million run game took approximately 57 minutes to run Appendix 3 (0:08:58-1:06:14). A screen shot of the outcome is below.



```
games completed: 955000
games completed: 960000
games completed: 965000
games completed: 970000
games completed: 975000
games completed: 980000
games completed: 985000
games completed: 990000
games completed: 995000
games completed: 1000000
total games completed: 1000000
games that took 0  turns to complete after the first shot hit 0
games that took 1  turns to complete after the first shot hit 831498
games that took 2  turns to complete after the first shot hit 166349
games that took 3  turns to complete after the first shot hit 1933
games that took 4  turns to complete after the first shot hit 137
games that took 5  turns to complete after the first shot hit 37
games that took 6  turns to complete after the first shot hit 20
games that took 7  turns to complete after the first shot hit 13
games that took 8  turns to complete after the first shot hit 9
games that took 9  turns to complete after the first shot hit 3
games that took 10  turns to complete after the first shot hit 1
james@james-VirtualBox:~/Documents$
```

Test of requirement: 2.4.3. Uses minimal amount of storage and memory

Each game that is played is a function of a class that gets called. This reduces the memory use of the program, because calling the game once will use no more memory than having thousands of plays of the game. The data from each game gets stored in an external file, meaning the data does not have to be stored in memory, but in storage. This means that the learnt data can be stored permanently and is not depended on the program always running. This can also mean that if it were to crash, it would still keep the memory that it stored up to that point, considering that it did not crash with writing to the file.

The file is not compressed because it would be impractical to do so. This is because the file only takes up between 12 to 20kilobytes, depending on how much data is being stored. This means that the storage is of no concern because of the relatively small file size. This system could be improved because as it can be seen it Appendix 7, there are some board combinations that are not possible, and these could be removed. If the dynamic creation of the storage gets implemented then this will not happen, and it will be more storage efficient.

Test of requirement: 2.4.4. Can simulate thousands of games in a reasonable amount of time

I was expecting the time for it to take to complete a game to be below 0.05 seconds. From the video of Appendix 3, the timings of the program can be worked out. The running of one million games took from 0:08:58 to 1:06:14, and of the 100000 games took from 0:00:39 to 0:05:51; taking approximately 57 minutes and 5 minutes 12 seconds respectively. That means on average it games took (60*(57+5)+12)/110000 = 0.00339 (to 3 significant figures)] seconds to complete, which I would consider a reasonable amount of time, considering all the other factors, like memory speed, read/write speed of the hard drives and the clock rate of the CPU.

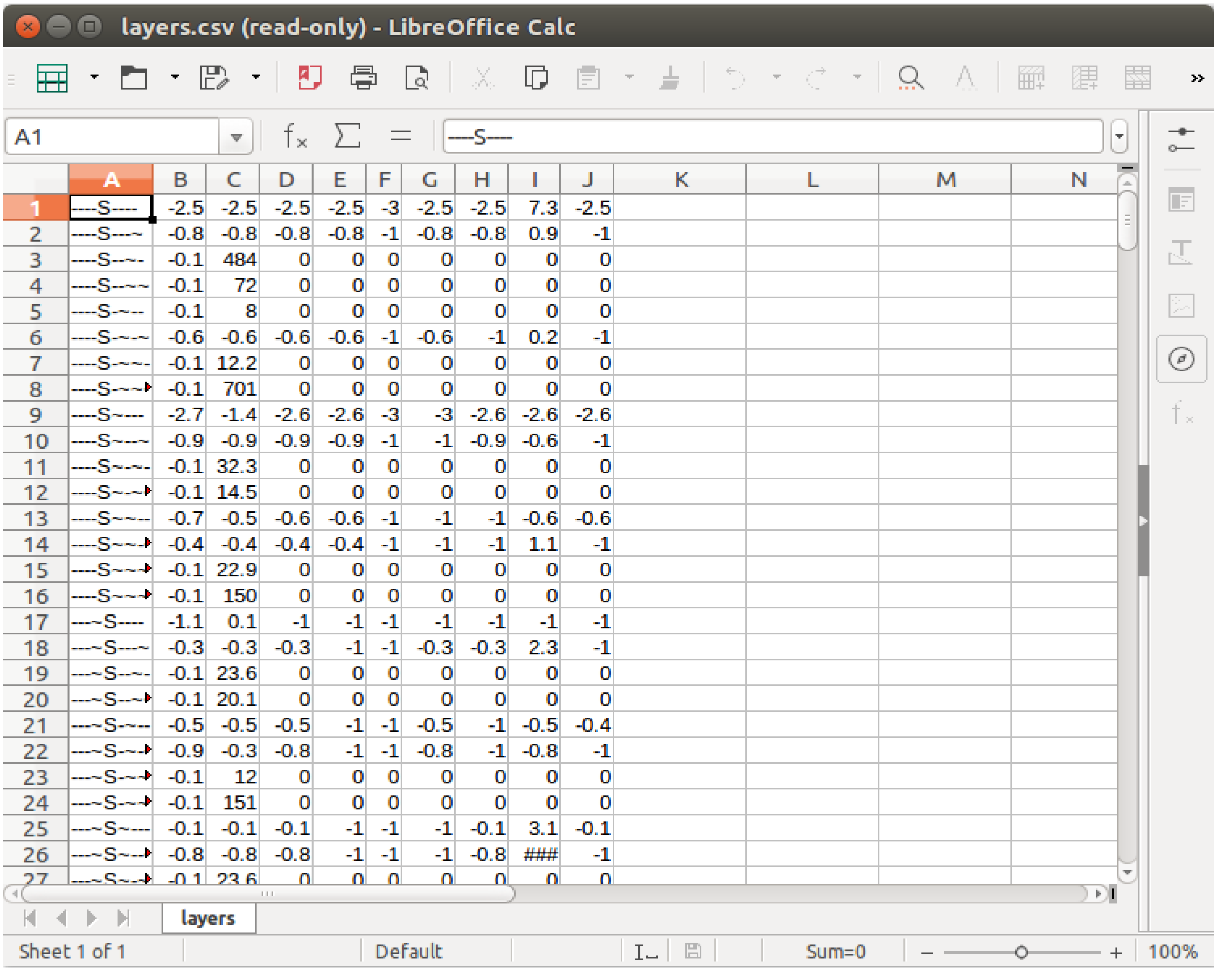
Test of requirement: 2.4.5. Doesn't crash if file isn't accessible

As seen in the video of Appendix 3, the program handles the file not being accessible. When I removed the file from the folder it showed the message and when prompted to search again and the file was there, it carried on as it would if the file didn't disappear. If the program was going through a long run, and the file was removed, or became inaccessible then it would pause mid run and wait for the users input before carrying on as normal. This can be seen between times 0:07:50 and 0:08.16 in Appendix 3.

Test of requirement: 2.5. To have it be capable of storing learning data

I wrote the game data to a comma-separated-values (CSV) file. This made it easy to read and write to, because it is all in rows and columns. I did not try to compress the file, because I felt like it would not be as easy to read and write, and could lead to further issues is the process fails. Also, there is no need to compress the file, because it doesn't take up a significant amount of storage, only averaging between 12 – 20 kilobytes.

The screen shot below shows the file and Appendix 7 is the final game data after one million games were played. Below is the stored data from 250000 runs, which shows how the data is stored and how it can be viewed by the user.



	A	B	C	D	E	F	G	H	I	J
1	----S----	-2.5	-2.5	-2.5	-2.5	-3	-2.5	-2.5	7.3	-2.5
2	----S---~	-0.8	-0.8	-0.8	-0.8	-1	-0.8	-0.8	0.9	-1
3	----S--~-	-0.1	484	0	0	0	0	0	0	0
4	----S--~~	-0.1	72	0	0	0	0	0	0	0
5	----S-~--	-0.1	8	0	0	0	0	0	0	0
6	----S-~-~	-0.6	-0.6	-0.6	-0.6	-1	-0.6	-1	0.2	-1
7	----S-~~-	-0.1	12.2	0	0	0	0	0	0	0
8	----S-~~	-0.1	701	0	0	0	0	0	0	0
9	----S~---	-2.7	-1.4	-2.6	-2.6	-3	-3	-2.6	-2.6	-2.6
10	----S~--~	-0.9	-0.9	-0.9	-0.9	-1	-1	-0.9	-0.6	-1
11	----S~-~-	-0.1	32.3	0	0	0	0	0	0	0
12	----S~-~	-0.1	14.5	0	0	0	0	0	0	0
13	----S~~--	-0.7	-0.5	-0.6	-0.6	-1	-1	-1	-0.6	-0.6
14	----S~~-	-0.4	-0.4	-0.4	-0.4	-1	-1	-1	1.1	-1
15	----S~~~	-0.1	22.9	0	0	0	0	0	0	0
16	----S~~~	-0.1	150	0	0	0	0	0	0	0
17	---~S----	-1.1	0.1	-1	-1	-1	-1	-1	-1	-1
18	---~S---~	-0.3	-0.3	-0.3	-1	-1	-0.3	-0.3	2.3	-1
19	---~S--~-	-0.1	23.6	0	0	0	0	0	0	0
20	---~S--~	-0.1	20.1	0	0	0	0	0	0	0
21	---~S-~--	-0.5	-0.5	-0.5	-1	-1	-0.5	-1	-0.5	-0.4
22	---~S-~-	-0.9	-0.3	-0.8	-1	-1	-0.8	-1	-0.8	-1
23	---~S-~~	-0.1	12	0	0	0	0	0	0	0
24	---~S-~~	-0.1	151	0	0	0	0	0	0	0
25	---~S~---	-0.1	-0.1	-0.1	-1	-1	-1	-0.1	3.1	-0.1
26	---~S~--	-0.8	-0.8	-0.8	-1	-1	-1	-0.8	###	-1
27	---~S~-	-0.1	23.6	0	0	0	0	0	0	0

Test of requirement: 2.5.1. Have it stored in a way that easy to read

The data being stored in CSV format made it easy to read, because I just had to iterate through the lines until I found the one which was my relevant to current situation. Then I just had to split the line into a list and set all the values. This a relatively simple task and there is little chance it could go wrong, and through all the testing I have done, I am yet to see it fail.

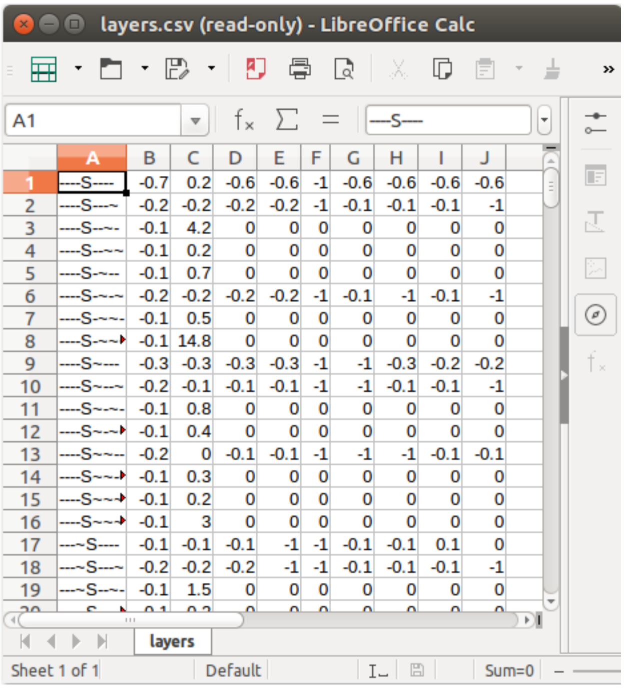
Test of requirement: 2.5.2. Have it stored in a way that easy to write

The CSV file makes it very easy to write back into the file, because the program keeps note of the line number that the data originally came from so that line can be replaced, while the other lines are being rewritten. This save the program from iterating through the list each time, because it knows what line had to change. This works because the data in the file has set rows and columns.

Test of requirement: 2.5.3. Have it set up and reset the file

When 'y' is input when prompted if they would like to retrain the program, in the initial user interface, the file should reset to how it started, from whatever state it was in before. This can be seen in Appendix 1 at time 01:09:00 to 01:10:08. Having just finished an extensive amount of training the program does a run of 5000 and receives very good results. When the same was done, but after setting the file at the start the program got the same result as before when it was untrained, as the random key remained constant. Also. by comparing the layers.csv file, after training and after resetting you can see it has been completely reset to the original file; as seen in the screen shots below.

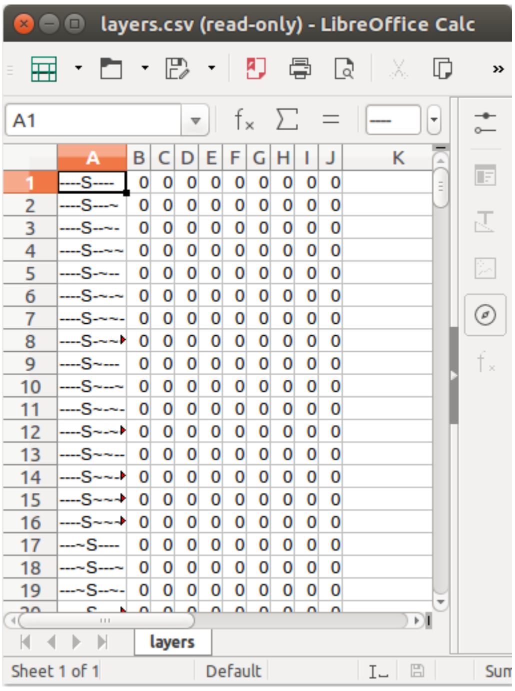
After 5000 games


layers.csv (read-only) - LibreOffice Calc

A1 ---S---

	A	B	C	D	E	F	G	H	I	J
1	----S----	-0.7	0.2	-0.6	-0.6	-1	-0.6	-0.6	-0.6	-0.6
2	----S---~	-0.2	-0.2	-0.2	-0.2	-1	-0.1	-0.1	-0.1	-1
3	----S--~-	-0.1	4.2	0	0	0	0	0	0	0
4	----S--~~	-0.1	0.2	0	0	0	0	0	0	0
5	----S-~--	-0.1	0.7	0	0	0	0	0	0	0
6	----S-~-~	-0.2	-0.2	-0.2	-0.2	-1	-0.1	-1	-0.1	-1
7	----S-~~-	-0.1	0.5	0	0	0	0	0	0	0
8	----S-~~	-0.1	14.8	0	0	0	0	0	0	0
9	----S~---	-0.3	-0.3	-0.3	-0.3	-1	-1	-0.3	-0.2	-0.2
10	----S~--~	-0.2	-0.1	-0.1	-0.1	-1	-1	-0.1	-0.1	-1
11	----S~-~-	-0.1	0.8	0	0	0	0	0	0	0
12	----S~-~	-0.1	0.4	0	0	0	0	0	0	0
13	----S~~--	-0.2	0	-0.1	-0.1	-1	-1	-1	-0.1	-0.1
14	----S~~-	-0.1	0.3	0	0	0	0	0	0	0
15	----S~~~	-0.1	0.2	0	0	0	0	0	0	0
16	----S~~~	-0.1	3	0	0	0	0	0	0	0
17	---~S----	-0.1	-0.1	-0.1	-1	-1	-0.1	-0.1	0.1	0
18	---~S---~	-0.2	-0.2	-0.2	-1	-1	-0.1	-0.1	-0.1	-1
19	---~S--~-	-0.1	1.5	0	0	0	0	0	0	0

After reset, before playing any games

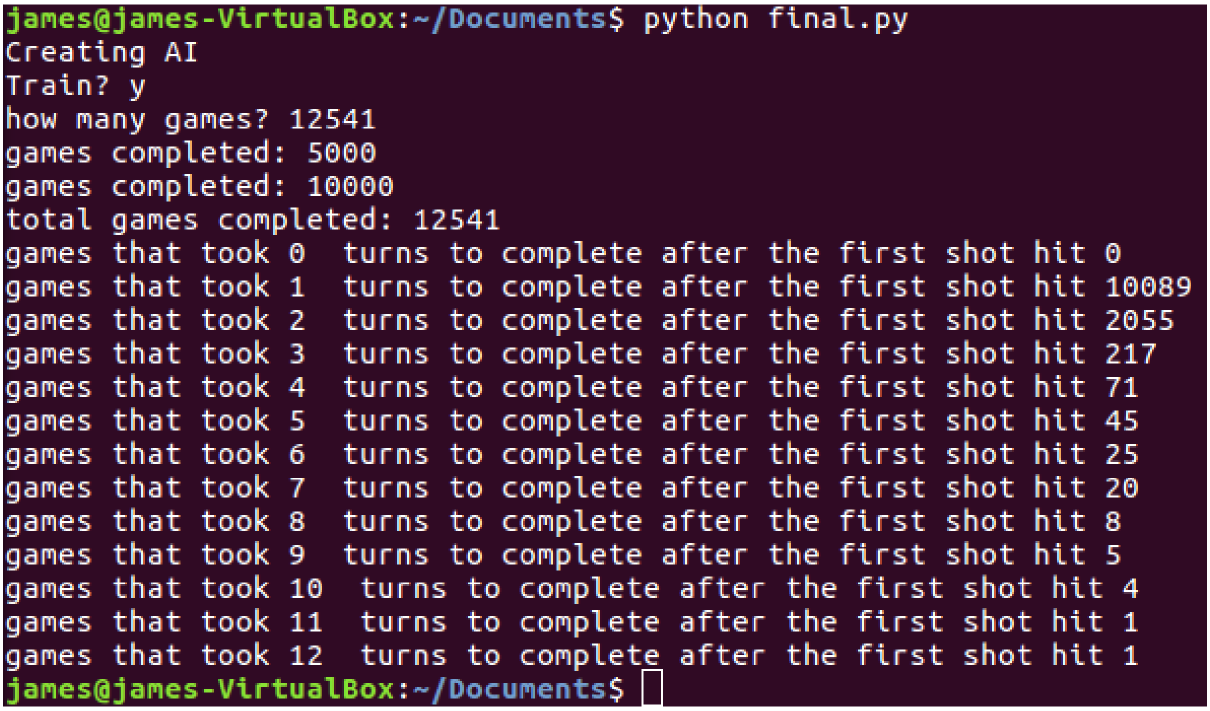

layers.csv (read-only) - LibreOffice Calc

A1 ---

	A	B	C	D	E	F	G	H	I	J	K
1	----S----	0	0	0	0	0	0	0	0	0	
2	----S---~	0	0	0	0	0	0	0	0	0	
3	----S--~-	0	0	0	0	0	0	0	0	0	
4	----S--~~	0	0	0	0	0	0	0	0	0	
5	----S-~--	0	0	0	0	0	0	0	0	0	
6	----S-~-~	0	0	0	0	0	0	0	0	0	
7	----S-~~-	0	0	0	0	0	0	0	0	0	
8	----S-~~	0	0	0	0	0	0	0	0	0	
9	----S~---	0	0	0	0	0	0	0	0	0	
10	----S~--~	0	0	0	0	0	0	0	0	0	
11	----S~-~-	0	0	0	0	0	0	0	0	0	
12	----S~-~	0	0	0	0	0	0	0	0	0	
13	----S~~--	0	0	0	0	0	0	0	0	0	
14	----S~~-	0	0	0	0	0	0	0	0	0	
15	----S~~~	0	0	0	0	0	0	0	0	0	
16	----S~~~	0	0	0	0	0	0	0	0	0	
17	---~S----	0	0	0	0	0	0	0	0	0	
18	---~S---~	0	0	0	0	0	0	0	0	0	
19	---~S--~-	0	0	0	0	0	0	0	0	0	

Test of requirement: 2.6. Must have a basic user interface

The program should display a very basic user interface that allows the user to have basic control over the program and for fills the requirement of 2.6.1 to 2.6.3. I added one additional feature, the number of games completed gets printed and every 5000 games. This is because for longer runs I could not tell if it had frozen or was just taking time. As shown in the test of requirement 2.4.1, the user interface is robust to invalid inputs. The screen shot bellow shows the full user interface after a



```
james@james-VirtualBox:~/Documents$ python final.py
Creating AI
Train? y
how many games? 12541
games completed: 5000
games completed: 10000
total games completed: 12541
games that took 0  turns to complete after the first shot hit 0
games that took 1  turns to complete after the first shot hit 10089
games that took 2  turns to complete after the first shot hit 2055
games that took 3  turns to complete after the first shot hit 217
games that took 4  turns to complete after the first shot hit 71
games that took 5  turns to complete after the first shot hit 45
games that took 6  turns to complete after the first shot hit 25
games that took 7  turns to complete after the first shot hit 20
games that took 8  turns to complete after the first shot hit 8
games that took 9  turns to complete after the first shot hit 5
games that took 10  turns to complete after the first shot hit 4
games that took 11  turns to complete after the first shot hit 1
games that took 12  turns to complete after the first shot hit 1
james@james-VirtualBox:~/Documents$
```

Test of requirement: 2.6.1. Allows the user to choose to retrain the learning program

The user interface has the option to retrain the program, by resetting all the learnt data. As shown in 2.5.3 the data gets completely reset when the input of ‘y’ is entered on the ‘Train?’ prompt. I expected this to happen, and this evidence proves that the user has the ability to retrain the program from the very basic interface.

Test of requirement: 2.6.2. Allows the user to choose how many games to loop through

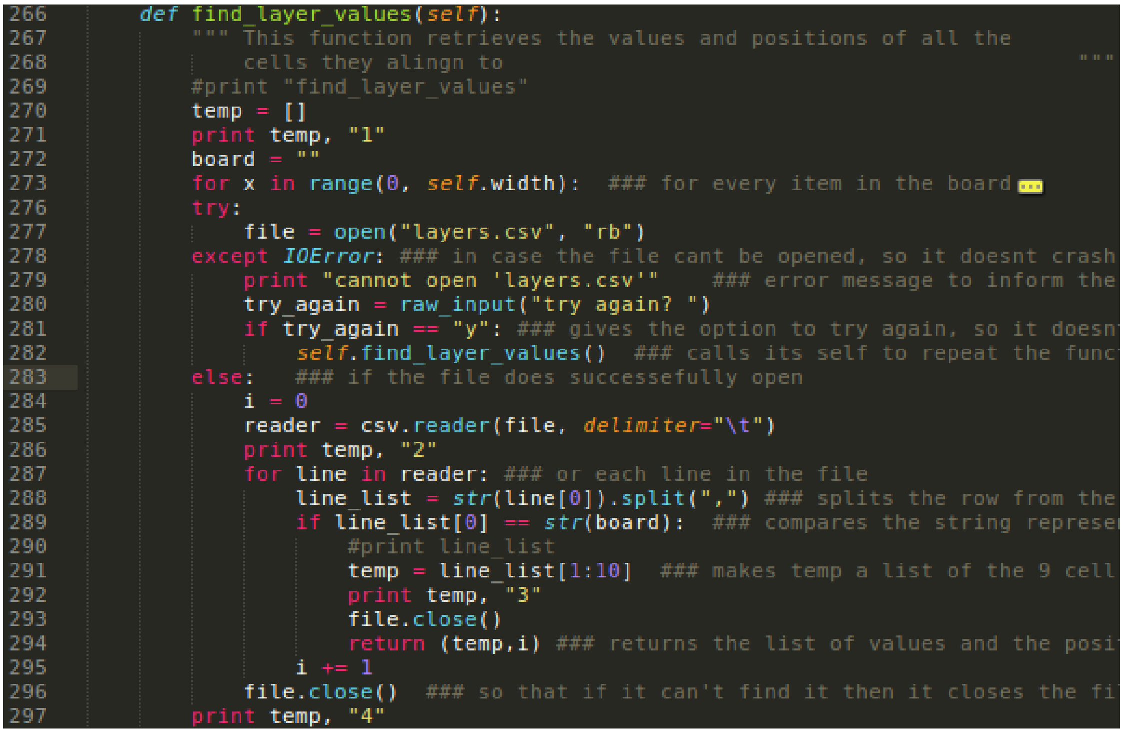
When prompted by the console line ‘how many games?’ the user can input any positive integer and that many games will be played. If they do not enter a positive integer, they will be prompted again until a valid input is given. This can be seen in 2.4.1, and an example of the code iterating through the games can be seen in Appendix 1, the same user interface is used throughout. This is sufficient evidence that I have a working system for choosing the amount of games.

Test of requirement: 2.6.3. Displays data that can prove that the program is learning

Once all the games have finished it is supposed to print a set of lines, like the ones shown in the design, displaying the amount of games that took k turns to complete. In the screen shot of requirement 2.6, and at the ends of all the test runs in Appendix 1 show this feature working. This is how I met this requirement.

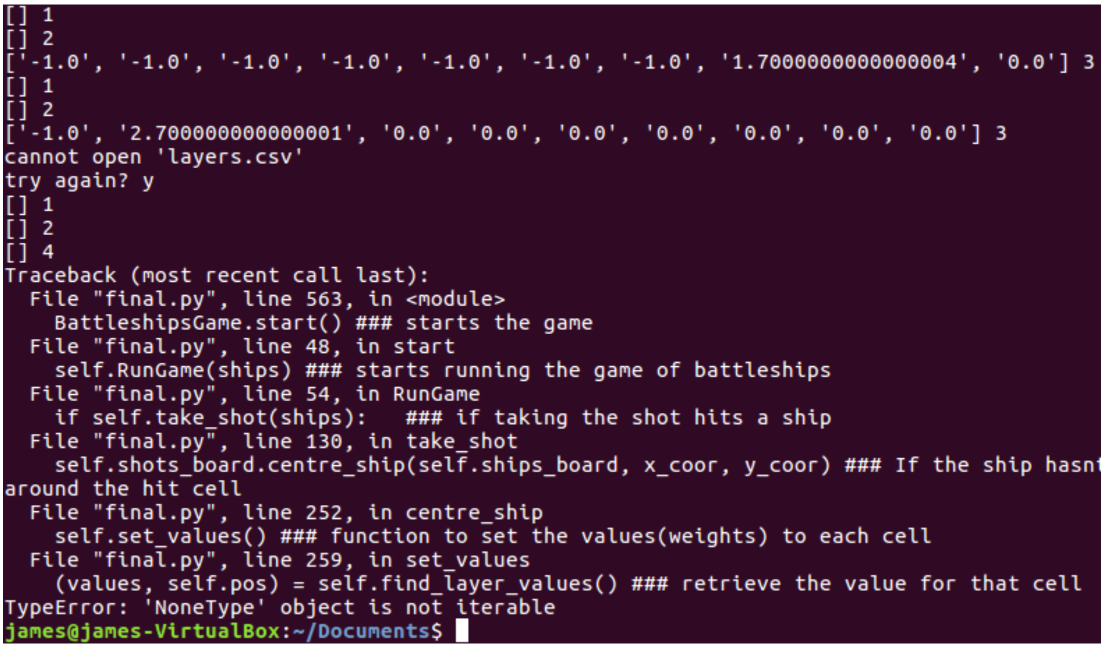
White box testing

This testing that when on during the implementation of the code. The video in Appendix 3 shows a small part of the testing that went on between 0:08:24 and 0:08:57. Throughout the code there are many different lines that print values to the console. Some of these are only for display of the program but most have been use in various tests. Toward the start of the programming many of the test were to see if the techniques were working, as I still had not worked out exactly how I was going to get the program to learn. A common way I tested to see the states of variables at certain stages was as follows.



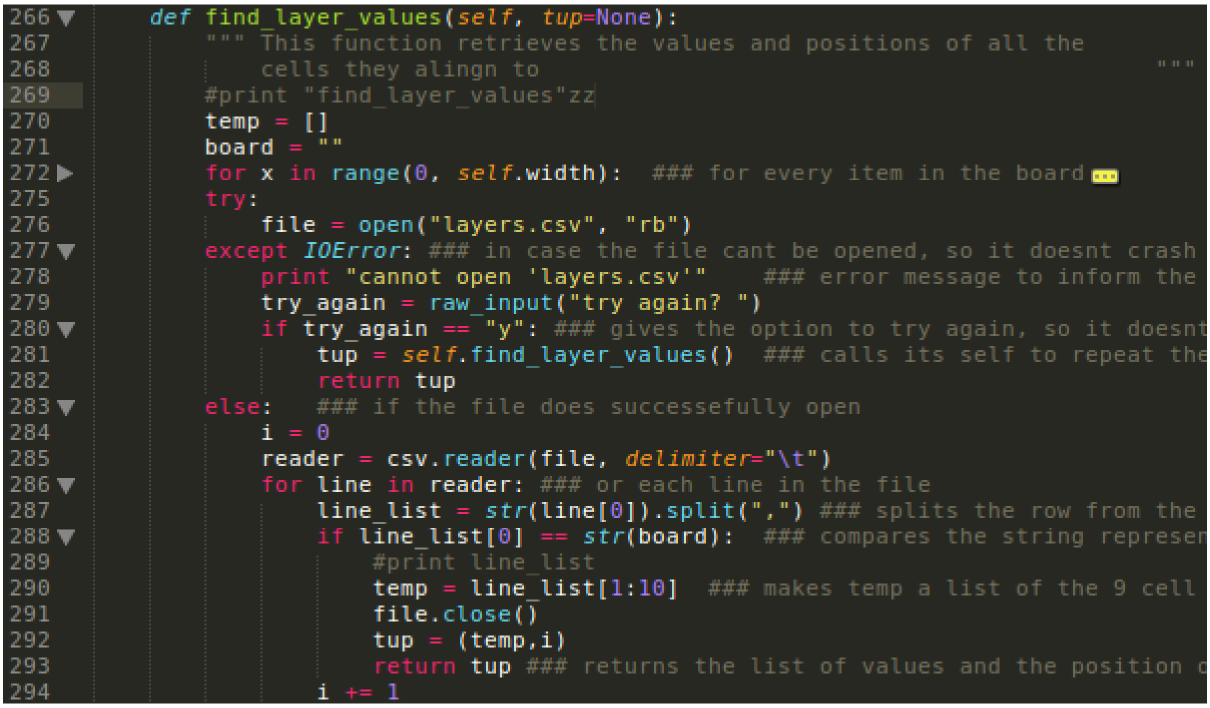
```
    def find_layer_values(self):
        """ This function retrieves the values and positions of all the
            cells they alingn to                                              """
        #print "find_layer_values"
        temp = []
        print temp, "1"
        board = ""
        for x in range(0, self.width):  ### for every item in the board
        try:
            file = open("layers.csv", "rb")
        except IOError: ### in case the file cant be opened, so it doesnt crash
            print "cannot open 'layers.csv'"     ### error message to inform the
            try_again = raw_input("try again? ")
            if try_again == "y": ### gives the option to try again, so it doesn
                self.find_layer_values()  ### calls its self to repeat the func
        else:   ### if the file does successefully open
            i = 0
            reader = csv.reader(file, delimiter="\t")
            print temp, "2"
            for line in reader: ### or each line in the file
                line_list = str(line[0]).split(",") ### splits the row from the
                if line_list[0] == str(board):  ### compares the string represe
                    #print line_list
                    temp = line_list[1:10]  ### makes temp a list of the 9 cell
                    print temp, "3"
                    file.close()
                    return (temp,i) ### returns the list of values and the posi
                i += 1
            file.close()  ### so that if it can't find it then it closes the fi
        print temp, "4"
```

I printed the value out, along with a number. This is because the interface I was using did not give me a way to follow variables, and with all the loops and return values, I found this was the quickest and easiest way to test values. However, it did become more difficult when trying to track a variable across multiple functions, but if used similar to a binary search, in that you close the gap between the marks, you can isolate any problems quite quickly.



```
[] 1
[] 2
['-1.0', '-1.0', '-1.0', '-1.0', '-1.0', '-1.0', '-1.0', '1.7000000000000004', '0.0'] 3
[] 1
[] 2
['-1.0', '2.700000000000001', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0'] 3
cannot open 'layers.csv'
try again? y
[] 1
[] 2
[] 4
Traceback (most recent call last):
  File "final.py", line 563, in <module>
    BattleshipsGame.start() ### starts the game
  File "final.py", line 48, in start
    self.RunGame(ships) ### starts running the game of battleships
  File "final.py", line 54, in RunGame
    if self.take_shot(ships):    ### if taking the shot hits a ship
  File "final.py", line 130, in take_shot
    self.shots_board.centre_ship(self.ships_board, x_coor, y_coor) ### If the ship hasn
around the hit cell
  File "final.py", line 252, in centre_ship
    self.set_values() ### function to set the values(weights) to each cell
  File "final.py", line 259, in set_values
    (values, self.pos) = self.find_layer_values() ### retrieve the value for that cell
TypeError: 'NoneType' object is not iterable
james@james-VirtualBox:~/Documents$ 
```

This example is from when I was trying to get the code to deal with the file being unable to open. You can see the program running fine up until the error was induced, by removing the file from the folder. After trying to access the file again, you can see the error happened. Here I could tell the problem was that it was reaching the end of the function before returning a value. This helped me identify how to fix it, which can be seen in Appendix 3 at time 0:07.50 to 0:08.16, and in the code below.



```
    def find_layer_values(self, tup=None):
        """ This function retrieves the values and positions of all the
            cells they alingn to                                          """
        #print "find_layer_values"zz
        temp = []
        board = ""
        for x in range(0, self.width):  ### for every item in the board
        try:
            file = open("layers.csv", "rb")
        except IOError: ### in case the file cant be opened, so it doesnt crash
            print "cannot open 'layers.csv'"    ### error message to inform the
            try_again = raw_input("try again? ")
            if try_again == "y": ### gives the option to try again, so it doesnt
                tup = self.find_layer_values()  ### calls its self to repeat the
                return tup
        else:   ### if the file does successefully open
            i = 0
            reader = csv.reader(file, delimiter="\t")
            for line in reader: ### or each line in the file
                line_list = str(line[0]).split(",") ### splits the row from the
                if line_list[0] == str(board):  ### compares the string represen
                    #print line_list
                    temp = line_list[1:10]  ### makes temp a list of the 9 cell
                    file.close()
                    tup = (temp,i)
                    return tup ### returns the list of values and the position o
                i += 1
```

This was the main way I solved bugs in my code that I could not immediately see. The debugger that identified the line of code that caused the problem, and its trace back path was very useful. It gave me the line that the error occurred, and the lines where it was being called from, which came in use when tracking values across multiple functions.

Conclusion

To conclude my testing and evaluating, I think I have done a reasonable job at meeting all the requirements. The majority of the requirements were met; however, I fell short of some. The ones where I didn't meet however, I feel I shill gave a good attempt at, as I at least made some progress on all of them. For all the requirements that I had not met, I have at least made progress in competing those goal in the long term. The plan which I laid out in my design, if continued, I am confident will produce a learning algorithm that is capable of playing, and excelling, at the game battle ships, while remaining within my computational ability.

Feedback

I took my code and analysis of my results to Mr M and he was very impressed. The paper work I showed and explained to him was my further research, my design and my testing proof that the program learnt. I felt this was sufficient evidence that it was possible to create a learning program for battleships, which gave an answer to the task he gave me.

He was impressed that I got an algorithm that could learn and how I had a set plan to further expand what I had. I showed him what I had in terms so far with a demonstration of my program and the data analysis I did to prove that it was learning in my testing. He liked my use of the natural logarithm to manipulate the data into a form that made it easy to read and visualise the results, and then explain what these results meant in terms of leaning.

The process of breaking down and simplifying the problem that I documented in the design, he also appreciated. The way that the I broke down the problem, by applying problem solving skills, which ended up with a better solution, albeit a long one that would take too long to complete. He said that the work I had produced and the the future plans I had planned proved to him that the current method could have worked on the full game with more time.

I went on to show him my research, and how I could have used neural networks, Q-learning and even deep learning had I the skills. He understood that the original task was far too complex for the level I am at, but I could also understand the research and could see how some of these techniques could potentially be used.

Why looking at the program he said that all the components he had asked for had been included, however he would have preferred the learning outputs top have gone to a file, but understood that he did not also me to do so. He said the code looked very professional in the way that it was written, and that had potential moving forward if I were to stay in this subject area.

We both agreed the original requirements were too hard, and that the secondary updated ones were more to my level, but were still optimistic. Next time we agreed that I should break down subject area before accepting the task, and that I should make requirements that were achievable.

Overall he was impressed at the work I had produced, the code I had written and said that I did achieve the task he set, despite not all of it being put into code.

Bibliography

Berry, N. (2011, December 3). *Battleship*. Retrieved from Data Genetics: http://www.datagenetics.com/blog/december32011/index.html

Butler. (2015, June 14). *Neural network data mining explained.* Retrieved from Butler Analytics: http://www.butleranalytics.com/neural-network-data-mining-explained/

Johson, G. (2016, April 4). *To Beat Go Champion, Google's Program Needed a Human Army*. Retrieved from New York Times: https://www.nytimes.com/2016/04/05/science/google-alphago-artificial-intelligence.html?_r=2

Landy, J. (2015, January 9). *Jonathan Landy*. Retrieved from Google sites: https://www.sites.google.com/site/jslandy/

Landy, J. (2016, October 15). *Deep reinforcement learning, battleship*. Retrieved from EFAVDB: http://efavdb.com/battleship/

Python Software Foundation. (No Date). *9.7. itertools — Functions creating iterators for efficient looping*. Retrieved from Python: https://docs.python.org/2/library/itertools.html

Raval, S. (2016, April 4). *Build a Neural Net in 4 Minutes.* Retrieved from YouTube: https://www.youtube.com/watch?v=h3l4qz76JhQ

Spencer-Harper, M. (2015, July 21). *How to build a simple neural network in 9 lines of Python code.* Retrieved from Medium: https://medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1#.runof0n4w

Sutton, R. S. (2012). *Reinforcement Learning: An Introduction.* Retrieved from people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf: http://people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf

Wikipedia. (2016, August 17). *List of machine learning concepts*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/List_of_machine_learning_concepts

Wikipedia. (2016, August 29). *Machine learning*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Machine_learning

Wikipedia. (2016, September 05). *State-Action-Reward-State-Action*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/State-Action-Reward-State-Action

Wikipedia. (2016, September 21). *Temporal_difference_learning*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Temporal_difference_learning

Woodford, C. (2017, Febuary 24). *Introduction to neural networks.* Retrieved from Explain That Stuff: http://www.explainthatstuff.com/introduction-to-neural-networks.html

Appendix 1 – The Raw Code

```
import csv

import sys

import itertools

import numpy as np



#np.random.seed(1)



class Game(object):
    """This class is where the majority of the game play takes place.  It is
        all in the one class so that variables can be passed around easily    """
    def __init__(self):
        """        Procedure that creates the games instance of the AI, and sets up
        the parts of the game that will remain constant                                            """
        print "Creating AI"
        self.ai = AI()           ### the ai
        self.turns = [] ### list of all the total ai turns taken
        self.turn = 0            ### the current amount of ai turns take for that game

    def start(self):
        """        Procedure that creates all the objects and variables that get
                   replaced each game                                                                              """
        self.turn = 0
        #print "Creating Boards"
        self.ships_board = Ships_board(4, 4)        ### board which the ships will be placed on
        self.shots_board = Shots_board(3, 3)        ### board which the ship gets centred too
        #print "Filling Boards"
        self.ships_board.get_board()      ### fills the board with cells
        self.shots_board.get_board()      ### fills the board with cells
        #print
        #self.ships_board.display()
        #self.shots_board.display()
        #print
        #print "Getting Ships"
        ships = self.get_ships()            ### creates a list of all the ships
        for ship in ships:                          ### iterates through all the ships and places them on the board
                #print "Choosing place for", ship.name
                self.Choose_place_for_ship(ship)
                #self.ships_board.display()
        #print
        #print "Set up complete"
        #print
        self.old_x_coor = 0     ### sets up values that get used later on to centre the grid
        self.old_y_coor = 0
        self.has_won = False
        self.current_hit_ship = False
        self.RunGame(ships)   ### starts running the game of battleships

def RunGame(self, ships):
        """     This procedure contains a loop which runs through each turn of
                the game                                                                                        """
        while not self.has_won: ### keeps looping until the game is won
                if self.take_shot(ships):           ### if taking the shot hits a ship
                        self.current_hit_ship = True                ### sets that there is currently a hit ship, so the board centres
        #self.ships_board.display()
        #print "It took", turn, "turns to sink all the ships."
        self.turns.append(self.turn) ### adds the amount ai turns it took to finish the game to the overall list

def Choose_place_for_ship(self, ship):
        """     This procedure picks a random place for the ship then checks
                that it is valid, returns the newly made grid that contains the
                new     ship                                                                                        """
        x_coor = np.random.randint(0, self.ships_board.height-1)            ### picks a random value within the range of the
        y_coor = np.random.randint(0, self.ships_board.width-1)             ### board to place the ship
        orentation = np.random.randint(0,1) # 1=Ver 0=Hoz
        #print ("x", x_coor, " y", y_coor, " or", orentation, "ship len", ship.length) ### for testing
        valid = self.ship_validate(orentation, x_coor, y_coor, ship) ### checks the position is valid
        if not(valid):            ### if its not valid then it starts the process again
                #print "not valid" ### for testing
                self.Choose_place_for_ship(ship) ### recurs to start process again
        else: ### if the place is valid
                #print "vaild" ### for testing
                ship.place_ship(orentation, x_coor, y_coor, self.ships_board) ### places the ship and replaces the old grid with one with the ship on it
```

```
def ship_validate(self, orentation, x, y, ship):
    """ This function works out if the desired placement of the ship is
        a valid place to put it.  Returns True if it is valid, False if
        not                                                                                                   """
    if (orentation == 0): ### if the ship is to be placed horizontal
        if (ship.length + x) > self.ships_board.width-1: ### if the end of the ship will go off the end of the grid
            return False ### returning false as soon as it is not valid so the rest doesn’t not have to be checked
        else:
            for i in range(0,ship.length): ### to check each space it will be, to see if there’s already a ship there
                if self.ships_board.check_cell_for_ship(x+i, y):
                    return False
    else: ### if the ship is to be placed vertical
        if (ship.length + y) > self.ships_board.height: ### if the end of the ship will go off the end of the grid
            return False
        else:
            for i in range(0,ship.length): ### to check each space it will be, to see if there’s already a ship there
                if self.ships_board.check_cell_for_ship(x, y+i) == True:
                    return False
    return True ### returns true if no faults in the placement have been found

def get_ships(self):
    """ This function puts all the makes all the ships and puts them
        into a list and returns it                                                                  """
    ships = [Ship(2, "Ship_1")]
             #Ship(5, "Aircraft carrier"),
             #Ship(4, "Battleship"),
             #Ship(3, "Submarine"),
             #Ship(3, "Cruiser"),
             #Ship(2, "Patrol boat")]
        ### the code only works with a ship of length 2, but can be built on to extend its use to any size ships
    return ships ### returns the list of ships

def take_shot(self, ships):
    """ This function allows the ai to make a shot.  It fetches the
        values of the coordinates then follows the actions of taking the
        shot.  It returns True if it hit a ship and false if the if it
        was a miss. It also decides which board the shot should be
        taken on                                                                                   """
    x_coor = 0
    y_coor = 0
    coor = (x_coor,y_coor) ### puts the coordinates into a tuple so the values can be assigned by a single run of a function
    if self.current_hit_ship == True: ### if there is a ship that’s currently hit. This decides what board it need to fire the shot on
```

```
                coor = self.ai.shoot(self.shots_board, self.ships_board)
            else: ### if the broad isn't centred then it must shoot the ships board
                coor = self.ai.shoot(self.ships_board)
            (x_coor,y_coor) = coor ### the tuple is unpacked into the coordinates again
            #if self.ships_board.check_cell_for_ship( x_coor, y_coor):
            if self.ships_board.grid[x_coor][y_coor].is_hidden and self.ships_board.check_cell_for_ship( x_coor, y_coor): ### checks to see if it was hidden and a hit
                self.ships_board.grid[x_coor][y_coor].set_cell(False) ### sets the cell to no longer hidden
                #print "Hit!"
                self.hit_ship(x_coor, y_coor, ships) ### goes through the process of hitting a ship
                self.has_won = self.check_won(ships) ### works out if all the ships are sunk
                if self.has_won: ###  if the game is won then updates the file
                    self.ai.update_file()
                    return True ### returns the True to signafy a hit
                self.shots_board.centre_ship(self.ships_board, x_coor, y_coor) ### If the ship hasn't been sunk then it centres the board around the hit cell
                self.old_y_coor = y_coor ### save the old coordinates so it can update the shots board next time if it misses
                self.old_x_coor = x_coor
                return True ### returns the True to signafy a hit
            else: ### if it wasnt a hit
                self.ships_board.grid[x_coor][y_coor].set_cell(False) ### sets the cell to no longer hidden
                if self.current_hit_ship == True: ### if there has been a hit it resets the centred board
                    self.shots_board.centre_ship(self.ships_board, self.old_x_coor, self.old_y_coor)
                #print "Miss."
                return False ### returns the False to signify a miss

        def hit_ship(self, x, y, ships):
            """ This function finds the ship that was hit                                    """
            for ship in ships: ### iterates through all the possible ships
                if self.ships_board.grid[x][y].symbol == ship.name[0]: ### the symbol matches that of the start of the ship's name, then that was the ship that was hit
                    ship.sink_ship() ### registers the hit and sinks the ship if necessary
                    return ### breaks out once the ship has been dealt with
            return

        def check_won(self, ships):
            """ This function checks to see if there are any ships yet to be
                sunk                                                                                              """
            for ship in ships: ### Checks all the ships
                if ship.sunk == False: ### if a ship is not sunk then the game is not over
                    return False ### returns here to prevent further unnecessary checks
            return True ### if they are all sunk then the game is over

    class Ship(object):
        """ This is the class for the ships                                                  """
        def __init__(self, length, name):
```

```
        """ This procedure sets the properties of the ships                                  """
        self.length = length
        self.name = name
        self.sunk = False ### sunk is used in checking if the game is done, this could have just been done by checking the overall lengths, but this way it can be expanded and makes the code easier to follow

    def sink_ship(self):
        """     This procedure takes a the hit cell off the length of the ship,
                and sets the ship to sunk                                                                                   """
        self.length -= 1 ### takes 1 off the length so the remaining length is the remaining amount of the ships cells still on the board
        if self.length == 0: ### if the length is 0 then all the cells must have been hit, so the ship gets sunk
            self.sunk = True
            #print "You sank my", self.name
            #print
        return

    def place_ship(self, orentation, x_coor, y_coor, ships_board):
        """     This function places the ship on the grid                                             """
        #print "Placing", self.name
        if orentation == 0: ### if the ship needs to be placed horizontally
            for i in range(0,self.length): ### for each of the cells the ship will take up
                ships_board.grid[x_coor+i][y_coor].set_cell(True, self.name[0]) ### placing the ship increasing in the x axis
        else:   ### if the ship needs to be placed vertically
            for i in range(0,self.length): ### for each of the cells the ship will take up
                ships_board.grid[x_coor][y_coor+i].set_cell(True, self.name[0]) ### placing the ship increasing in the y axis

class Board(object):
    """     This is the base class for all the boards.  It has all the shared
            subroutines of all the boards for them to inherit                                         """
    def __init__(self, width, height):
        """ This initial procedure creates the properties needed by all the
                object boards                                                                                                    """
        self.grid = []          ### grid is the grid that hold the cells which makes up what the player can see as the board
        self.width = width ### these variables set the dimensions of the boards
        self.height = height

    def check_cell_for_ship(self, x, y):
        """ This funtion returns true is a the selected cell is a ships        """
        if self.grid[x][y].is_ship == True: ### is the cell a ship
            return True ### returns true if it is
        else:
            return False ### false if it isn’t
    def display(self):
        """ This procedure displays the grids in a format similar to that of
            battleships                                                                                                              """
        i = self.width - 1 ### i is being used so the list prints in the correct order
        for x in range(0, self.width):
            print i, ### prints out the y axis index number
            for y in range(0, self.height): ### these loops iterate through every cell and get values so they can be printed in the correct order
                print self.grid[y][i], ###  prints the cell
            i -= 1 ### increments the i value by -1 so the cells print out in the correct order
            print
        print " ",
        for j in range(0, self.height): ### prints out the x axis index
            print j,
        print
        print

    def get_board(self):
        """ This procedure creates the grid for the boards                                   """
        cell_no = 0
        for x in range(0, self.width):        ### this loops through, creating a dimensional list for the width of the grid
            self.grid.append([])     ### this empty list makes the grid 2 dimensional, one dimension for the width and the other for the length
            for y in range(0, self.height):        ### this loop then fills the dimensional board with a cell for the length fo the grid
                cell_no += 1 ### adding 1 to the cell_no so it gives each cell a new number
                self.grid[x].append(self.get_cell()) ### appends a cell to the current, get cell makes sure the correct type of cell is appended
                self.grid[x][y].number = cell_no ### so the cells have a unique number to help identify them

class Shots_board(Board):
    """ This class inherits all the methods from Board, and has additional
        methods that allow it to function differently from the other boards.
        It is for the larger version of the board, where the ships are place
        and shots are taken until a ship gets hit                                              """
    def get_cell(self):
        """ this function is to return the type of cell that this class
            needs. The function that it returns to is inherited from
            Board                                                                                                                    """
        return Shots_cell()

    def centre_ship(self, ships_board, ships_x, ships_y):
        """ The purpose of this procedure is to centre the board around the
            current hit section of the ship                                                      """
        half_x = int(np.ceil(self.width/2)) ### works out the coordinates of the centre point
        half_y = int(np.ceil(self.height/2))
        alignment_x = 0-half_x ### so it starts at the lowest x coor of the new grid
        for shots_x in range(0, self.width):            ### to make sure all the x axis cells are set
            alignment_y = 0-half_y  ### so it starts at the lowest y coor of the new grid
            for shots_y in range(0, self.height):            ### to make sure all the y axis cells are set
                self.set_cells(ships_board, shots_x, shots_y, ships_x+alignment_x, ships_y+alignment_y) ### to set the cell of the specific location with all the necessary data
                alignment_y += 1     ### to iterate through the y axis
            alignment_x += 1     ### to iterate through the x axis
        #self.display()
        self.set_values() ### function to set the values(weights) to each cell

    def set_values(self):
        """ This subroutine sets each cell in the board's value(weight) for
            the ai                                                                                                                                              """
        values = []
        self.pos = 0
        (values, self.pos) = self.find_layer_values() ### retrieve the value for that cell
        i = 0       ### a counter to make the value in the list be the right value for the cell
        for x in range(0, self.width):      ### these for loops iterate through every cell in the 2D array
            for y in range(0, self.height):
                self.grid[x][y].value = float(values[i]) ### sets the value of the cell to the retrieved value
                i += 1

    def find_layer_values(self, tup=None):
        """         This function retrieves the values and positions of all the
            cells they alingn to                                                                                                      """
        #print "find_layer_values"
        temp = []
        board = ""
        for x in range(0, self.width):      ### for every item in the board
            for y in range(0, self.height):
                board += str(self.grid[x][y]) ### adds the cells string symbol to a string representing the board
        try:
            file = open("layers.csv", "rb")
        except IOError:        ### in case the file cant be opened, so it doesnt crash
            print "cannot open 'layers.csv'"   ### error message to inform the user of the error
            try_again = raw_input("try again? ")
            if try_again == "y": ### gives the option to try again, so it doesn’t end up constantly recurring
                tup = self.find_layer_values()  ### calls its self to repeat the function
                return tup
        else:       ###         if the file does successfully open
            i = 0
            reader = csv.reader(file, delimiter="\t")
            for line in reader: ### or each line in the file
                line_list = str(line[0]).split(",")     ### splits the row from the file in to a list of the columns
                if line_list[0] == str(board):          ### compares the string representation of the current board to the board representation on the row in the file
                    #print line_list
                    temp = line_list[1:10]  ### makes temp a list of the 9 cell values in the row
                    file.close()
                    tup = (temp,i)
                    return tup ### returns the list of values and the position of the row corresponding to the current board in the file
                i += 1
            file.close()  ### so that if it can't find it then it closes the file before it crashes

    def set_cells(self, board, shots_x, shots_y, ships_x, ships_y):
        """ The role of this procedure is to set the necessary properties
            of the desired cells in the ships board to the cells in the
            shots board and to identify if the shots cell will appear off
            the ships ships_board                                                                """
        if ships_x >= 0 and ships_x < self.width and ships_y >= 0 and ships_y < self.height: ### if the cell is in a valid place(on the board)
            self.grid[shots_x][shots_y].is_hidden       = board.grid[ships_x][ships_y].is_hidden ### sets all the required values
            self.grid[shots_x][shots_y].is_ship         = board.grid[ships_x][ships_y].is_ship
            self.grid[shots_x][shots_y].symbol                  = board.grid[ships_x][ships_y].symbol
            self.grid[shots_x][shots_y].number                  = board.grid[ships_x][ships_y].number
        else: ### if the cell is not in a valid place then it is off the board
            self.grid[shots_x][shots_y].off_the_board()  ### dets the cell to the values it has when its off the board

class Ships_board(Board):
    """ This class inherits all the methods from Board, and has additional
        methods that allow it to function differently from the other boards.
        It is for the centred version of the board, which the ai uses learns
        from                                                                                          """
    def get_cell(self):
        """ This function is to return the type of cell that this class
            needs. The function that it returns to is inherited from Boards"""
        return Ships_cell()

class Training_board():
    """ This class is a very basic class that holds the properties of a
        grid, the length of its self, and the position of the board
        and its values in the storage file.                                          """
    def __init__(self, length):
        """ creates the grid, the length of its self, and the position of
            the board and its values in the storage file.                    """
        self.grid = []
        self.length = length
        self.pos = 0
```

```
class Cell(object):
    """     This is the base object class for all the cells.  It has all the
            shared subroutines of all the Cells for them to inherit                     """
    def __repr__(self):
        """ This fuction means that the cell its self can appear to have a
                value. For example, if the cell was printed it would run this
                function and print the returned value.  It works the same way
                for any other instance where the cell is directly referenced as
                the value. In this case, the value will always be a string
                character either representing the cell as hidden, empty or a
                ship.                                                                               """
        if self.is_hidden: ### if the cell is hidden, it returns the character used to show a hidden cell
                return '-'
        return self.get_symbol()        ### so if its not hidden then it works out what symbol should be given

    def get_symbol(self):
        """ This fuction works out what symbol should be used to represent
                the cell.  if it is a ship then it returns the symbol that it
                gets assigned                                                                   """
        if self.is_ship:
                return self.symbol
        else:
                return '~'

class Ships_cell(Cell):
    """ This class inherits from the parent class Cell.  It is for the cells
            that will populate the ships board. It has the values needed for
            the basic functioning of a battleships game                             """
    def __init__(self):
        """ This initial procedure creates the properties of this cell              """
        self.is_ship = False     ### whether the it is a ship
        self.is_hidden = True   ### whether the it is hidden
        self.symbol = "X"               ### The cells base symbol, symbol will be the character that represents the cell when its not hidden

    def set_cell(self, value, set_to=None):
        """ This method sets the cell, it sets its property hidden and it
                can set it to be a ship, depending on the input parameters          """
        if set_to == None: ### if the set to value is None, then the hidden value needs to change
                self.is_hidden = value
        else:   ### if there is a set to value then ship becomes the value, and symbol becomes the set to value
                self.is_ship = value     ### I made the function work in this way to deduce the abound of code, the same function can perform different tasks depending on the parameters.
                self.symbol = set_to    ### this saves me from writing two different subroutines for similar tasks that only need one
```

```
class Shots_cell(Cell):
        """ This class inherits from the parent class Cell.  It is for the cells
                that will populate the centred shots board. It has the values
                needed for the functioning of the centred shots board, which the ai
                can work and learn from                                                                                                   """
        def __init__(self):
                """ This initial procedure creates the properties of this cell                          """
                self.is_ship = False      ### whether the it is a ship
                self.is_hidden = True  ### whether the it is hidden
                self.symbol = "X"                       ### the cells base symbol, symbol will be the character that represents the cell when its not hidden
                self.off_board = False ### whether the cell would be off the sips board, if it was put on that board, relative to its neighbouring cells
                self.value = 0                                          ### the numerical value or weight of the certain cell, so the ai knows where to shoot

        def off_the_board(self):
                """ If a cell is off the board then it has to be given the values
                        that let it be treated accordingly. it makes it appear like a
                        miss, but makes it immediately unhidden, so the ai will learn
                        not to shoot it                                                                                           """
                self.is_hidden = False  ### sets it to unhidden because it is not on the board, therefore cant be shot and so cant be made un hidden
                self.off_board = True  ### sets the cell to off the board, the symbol doesn’t change because in terms of the gameplay whether the cell is off the board or hit, it still shouldn’t be a target so the
ai will quickly learn not to shoot there.
                                                                ### It being the same symbol means that there are a lot less possible board combinations

class AI():
        """this class is the learning part of the program.  It is
                responsible for taking shots and learns from the consequences
                of that shot, with a system that weights each cell and chooses the
                one with the largest weight. If the move is good then it will
                increase the weight of the cell, if it isn't then the weight will
                decrease.  It is also where the file gets updated with then learned
                values.  The cells with have different values depending on the
                the current contents of the board.  All the possible board states are
                stored in the external file, 'layers.csv'.                                                  """
        def __init__(self):
                """ This initial function runs when the class is first assigned to
                        an object.  It creates lists needed to function and it gives the
                        player the option to retrain the program from the beginning.        """
                self.training_board = Training_board(0) ### creating the training board, which acts as a list
                self.flat_board = Training_board(0)             ###             the one dimensional representation of the centred, shots, board
                if raw_input("Train? ") == "y": ###             if, when prompted, the user shows they want to retrain the program
                        self.train = Training()  ### creates an object for training
                        self.train.initial_start() ### starts up the training phase
                        #print "Start up complete"
```

```
	def shoot(self, board, ships_board=None):
		""" This function returns the coordinates it want to shoot. If the
			the board has already centred then it works out where to shoot,
			if not then it just shoots random coordinates untill it hits a
			ship                                                                                                                        """
		self.flat_board.grid = []		### sets the 1 dimentonal boards grid to nothing
		for x in range(0, board.width):	### these for loops iterate through every item in the board got from the parameters
			for y in range(0, board.height):
				self.flat_board.grid.append(board.grid[x][y])		### it appends each cell in the board to the flat_board's grid.
																			### the board is converted to a one
dimentional array so that it is easy to understand what is happening
		if ships_board != None:		### if the board has been centred
			self.training_board.grid = self.get_numerical_array()	### the training board becomes a list of all the set cell values, to then be added to the overall weight where appropriate
			#print self.training_board.grid
			self.flat_board.pos = board.pos ### so the new list has the position of the data it needs to change in the layers.csv file
			loc = self.get_shot_coordinates() ### gets the coordinates where the weight is highest
			self.update_file()	### updates the file with the new values
			return self.get_coordinates(loc, ships_board)		### returns the coordinates of this location
		else:	### if the board hasnt been centred
			#print "X: ", x," Y: ", y
			x = np.random.randint(0, board.width)	### pics random value for x
			y = np.random.randint(0, board.height)	### pics random value for y
			return (x,y)		### returns the random coordinates

	def get_shot_coordinates(self):
		""" This function calculates where it thinks the best place to fire.
			It works out then returns that cells cell number                                              """
		flat_loc = 0### this varible will hold the location in the list of the cell that it thinks is the best place to fire
		k = 0	### k is a counter variable that will be used to find the location of the cell currently being checked
		chosen_cell = self.flat_board.grid[0]		### sets the first chosen cell to the first cell in the list
		for i in self.flat_board.grid:		### iterates through all the items in the 1 dimensional version of the grid.  Its 1 dimensional so its easier to follow and understand
			if i.value > chosen_cell.value:	### if the current value is the largest so far
				flat_loc = k  ### the location of the largest cell becomes the location of this one
				chosen_cell = i ### and the cell chossen to shot becomes this cell
			k += 1 ### increase by one each loop to count the loops
		#print chosen_cell.value
		#print "num= ",chosen_cell.number-1
		#print "loc= ", flat_loc
		chosen_cell.value += self.training_board.grid[flat_loc]   ### the consequence weight is then added to the value of the cell, greatly decreasing it, if it is not hidden, slightly decreasing it, if it is a
miss and slightly increasing it, if it is a hit
		#print chosen_cell.value
		BattleshipsGame.turn += 1 ### increasing the turn count by one, signifying the end of the turn where it calculates the cell, random guesses are not counted
```

```
        #print turn
        return (chosen_cell.number-1)    ### returns the cell number of the chosen cell, so it can be found in the ships board so the correct coordinates can be found

    def update_file(self):
        """ This subroutine updates the file, 'layers.csv'.  It reads the
            whole copies the whole file, make the the change then writes
            back to the file                                                                                                              """
        data_list = []          ### empty list that will hold the the data from the file
        try:        ###         so if the file fails to open the program doesnt not crash
            file = open("layers.csv", "rb")      ### opens the file 'layers.csv' to read
        except IOError:     ###         the error of the file not opening will be an IO Error
            print "cannot open 'layers.csv'"    ### error message to inform the user of the error
            try_again = raw_input("try again? ")            ### gives the option to try again, so it doesn’t end up constantly recurring
            if try_again == "y":
                self.update_file()      ### calls its self to repeat the process
        else:       ###         if the file does successfully open
            data = csv.reader(file) ### assigns the file contents to a variable
            data_list.extend(data)### puts the contents of the variable into a list format
            file.close() ### closes the file
        as_string = ''.join(str(item) for item in self.flat_board.grid)          ### puts the 1 dimentional grid into a string of characters
        as_list = [as_string] ### creates a list containing the sting representation of the grid
        #print as_string
        for j in self.flat_board.grid: ###for each cell in the 1 dimentional list
            as_list.append(j.value)             ### it appends the value of that cell
        line_to_override = {self.flat_board.pos:as_list} ### sets the line which will get over written
        try:        ###         so if the file fails to open the program doesnt not crash
            file = open("layers.csv", "wb")      ### opens the file 'layers.csv' to write
        except IOError:     ###         the error of the file not opening will be an IO Error
            print "cannot open 'layers.csv'"    ### error message to inform the user of the error
            try_again = raw_input("try again? ")            ### gives the option to try again, so it doesn’t end up constantly recurring
            if try_again == "y":
                self.update_file()      ### calls its self to repeat the process
        else:       ###         if the file does successefully open
            writer = csv.writer(file)           ### sets the writer to the file
            for line, row in enumerate(data_list):          ### for each line in the file, line number(number) and
                write_data = line_to_override.get(line, row) ### separates the it into line(the position of the board's line in the file) and row(the string representation of the board and
the values of the cells) and puts them as a single variable
                writer.writerow(write_data)        ### writes the variable to the file as a row
            file.close() ### closes the file

    def get_coordinates(self, loc, ships_board):
        """ The purpose of this function it to work out what the coordinates
            of a cell are from its cell number                                                                              """
        i = 0 ### a counter used to work out when the loops have reached the coordinates of the cell
        for x in range(0, ships_board.width):       ### this is a for loop that iterates through the size of the ships_board
            for y in range(0, ships_board.height):
                if i == loc:    ### when it reaches the same amount of iterations as it would tack to reach the cell
                    #print "X: ", x," Y: ", y
                    return (x,y) ### it returns the values of x and y as a coordinate, because this is the coordinates of the cell
                i += 1

    def get_numerical_array(self):
        """ This function creates a list containing the reward values for if
            the ai targets that cell. The reward values depend on the
            contents of the cell                                                                                   """
        temp = []   ### empty list that will contain all the reward values
        for cell in self.flat_board.grid:       ### for each item in this grid
            if cell.is_hidden == False or cell.off_board == True:       ### if the is not hidden or is off the board
                temp.append(-1)         ### a large negative weight is added to deter the ai from shooting there again
            elif cell.is_ship == True:              ### if the cell contains a ship
                temp.append(0.1) ### a smaller positive weight is added to reward it
            else:  ### so if it is a miss
                temp.append(-0.1)       ### a smaller minus value is added.  This is too counter the value given by the ship, so if there are multiple places the ship could be in a certain
board scenario, then it learns there is an even chance of it being in either
        return temp

class Training():
    """ This class is for the training of the ai. It creates the blank file,
        which has 256 lines, one for each possible combination of the centred
        board.                                                                                                                      """
    def initial_start(self):
        """ This procedure makes a list of all the possible combinations of
            the the string representation of ther board, then gives the
            values of all the cells in order.  It then writes each board
            representation and cell values into the file, row by row.              """
        self.symbol_list = ["-","~"]          ### a list of the symbols that need to be used in the product function.
        arr = itertools.product(self.symbol_list, repeat = 8)       ### creates an itertools array of all the combinations of the two characters in symbol_list when there are 8 characters.  this could also be
done by hand in a recursive fuction, but itertools makes it easier
        try:        ###         so if the file fails to open the program doesnt not crash
            file = open("layers.csv", "wb")     ### opens the file 'layers.csv' to write
        except IOError:         ###         the error of the file not opening will be an IO Error
            print "cannot open 'layers.csv'"   ### error message to inform the user of the error
            try_again = raw_input("try again? ")            ### gives the option to try again, so it doesnt end up constantly recuring
            if try_again == "y":
                self.initial_start()        ### calls its self to repeat the process
        else:       ###         if the file does successefully open
```

```
                        writer = csv.writer(file) ### sets the writer to the file
                        for i in arr: ### for each element in the array
                                l = list(i) ### creates a list of the element
                                l.insert( 4, 'S') ### inputs the character 'S' to the centre of the string.  this represents the ship in the 3 by 3 centred grid
                                s = ''.join(str(item) for item in l) ### this sets the list back into a sting so it is ready to be put into the file
                                #print s
                                writer.writerow((s, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)) ### writes row(string board, followed by the values) to the file
                        file.close() ### closes the file

################################################################################################
BattleshipsGame = Game() ### creates the instance of the game
valid = False ### used to make sure that the value for the number of games is an integer
while not valid:
        try: ### incase of an invalid input
                how_many_games = int(raw_input("how many games? ")) ### creates an integer from the input
        except ValueError:
                print "Incorrect input, try again" ### error message to inform the user of their incorrect input
                print
        else:
                if how_many_games > 0: ### if it is a valid number of games
                        valid = True ### if not error is detected then it must be a valid interger
for i in xrange(1, how_many_games+1):       ### repeates for the amount of specified games, starts at one,
                                                        ### hence the plus one on the limiting parameter
        BattleshipsGame.start() ### starts the game
        if (i) % 5000 == 0: ### for every 5000
                print "games completed:",i ### it prints the number of games so far, so that for longer runs
                                                ### you know it isnt stuck in a loop
print "total games completed:", i### prints the total number of games completed
j = 0 ### counter for what the game number is
list_of_values = [0]
for num in BattleshipsGame.turns: ### for the number of ai turns it took in every game completed
        j += 1          ### games completed goes up
        if len(list_of_values) > num: ### so the index is not out of the lists range
                list_of_values[num] += 1 ### 1 more game too this many turns to complete
        else:           ### if it would bve out of range then its is the first game to take this long
                while len(list_of_values) < num:
                        list_of_values.append(0)
                list_of_values.append(1) ### only one game has taken this many turns
for k in range(0, len(list_of_values)): ### prints out the list of how many games took what amount
                                                        ### of turns to complete
        print "games that took",k," turns to complete after the first shot hit",list_of_values[k]
                                                        ### so its easy to read and understand the results
################################################################################################
```

Appendix 2

```
import random
import numpy as np

class Game():
    def start(self):
        turns = 0
        temp = []
        print "Creating Board"
        ships_board = Ships_board()
        shots_board = Shots_board()
        print "Filling Board"
        ships_board.get_board()
        print
        shots_board.get_board()
        print
        ships_board.display()
        #shots_board.display()
        print "Getting Ships"
        ships = self.get_ships()
        for ship in ships:
            print "Choosing place for", ship.name
            ships_board.grid = self.Choose_place_for_ship(ships_board, ship)
            #ships_board.display() ### for testing
        print
        print "Set up complete"
        print
        self.RunGame(ships_board, shots_board, ships, turns)

    def RunGame(self, ships_board, shots_board, ships, turns):
        has_won = False
        while not has_won:
            #for ship in ships:
            #   print ship.length #testing only
            ships_board.display()
            #ships_board.write_board()
            turns += 1
            if self.take_shot(ships_board, ships, shots_board):
                has_won = self.check_won(ships)
        ships_board.display()
        print "It took", turns, "turns to sink all the ships."
        """ put game data into a list as it iterates out of the procedures """
```

```
def Choose_place_for_ship(self, ships_board, ship):
        x_coor = random.randint(0, ships_board.height-1)
        y_coor = random.randint(0, ships_board.width-1)
        orentation = random.randint(0,1) # 1=Ver 0=Hoz
        print ("x", x_coor, " y", y_coor, " or", orentation, "ship len", ship.length) ### for testing
        valid = self.ship_validate(orentation, x_coor, y_coor, ship, ships_board)
        if not(valid):
                #print "not vaild" ### for testing
                self.Choose_place_for_ship(ships_board, ship)
        #if valid == True:
        else:
                #print "vaild" ### for testing
                ships_board.grid = self.place_ship(orentation, x_coor, y_coor, ship, ships_board)
        return ships_board.grid

def place_ship(self, orentation, x_coor, y_coor, ship, board):
        print "Placing", ship.name
        if orentation == 0:
                for i in range(0,ship.length):
                        board.grid[x_coor+i][y_coor].set_cell(True, ship.name[0])
        else:
                for i in range(0,ship.length):
                        board.grid[x_coor][y_coor+i].set_cell(True, ship.name[0])
        return board.grid

def ship_validate(self, orentation, x, y, ship, board):
        if (orentation == 0):
                if (ship.length + x) > board.width-1:
                        return False
                else:
                        for i in range(0,ship.length):
                                if board.check_cell_for_ship(x+i, y):
                                        return False
        else:
                if (ship.length + y) > board.height:
                        return False
                else:
                        for i in range(0,ship.length):
                                if board.check_cell_for_ship(x, y+i) == True:
                                        return False
        return True
```

```
def get_ships(self):
    ships = [Ship(5, "Aircraft carrier"),
            Ship(4, "Battleship"),
            Ship(3, "Submarine"),
            Ship(3, "Cruiser"),
            Ship(2, "Patrol boat")]

    return ships

def take_shot(self, ships_board, ships, shots_board):
    for i in xrange(5000):
        x_coor = self.get_coor( ships_board, "Input X axis of the target grid: ", True)
        y_coor = self.get_coor( ships_board, "Input Y axis of the target grid: ", False)
        if ships_board.grid[x_coor][y_coor].is_hidden == True:
            print "x =", x_coor, "y =", y_coor
            break
    if i == 5000:
        print "over flow in take_shot"
    self.hit_ship(x_coor, y_coor, ships_board, ships)
    ships_board.grid[x_coor][y_coor].set_cell(False, None)
    if ships_board.check_cell_for_ship( x_coor, y_coor):
        if raw_input("'end' to end ") == "end":
            for ship in ships:
                ship.sunk = True
        print "Hit!"
        #self.hit_ship(x_coor, y_coor, ships_board, ships)
        shots_board.center_ship(ships_board, x_coor, y_coor)
        return True
    else:
        print "Miss."
        return False

def get_coor(self, board, text, x_axis_coor):
    valid = False
    i = 0
    while valid == False and i < 5000:
        i += 1
        try:
            if x_axis_coor:
                coor = raw_input("enter x coor: ")
            else:
                coor = raw_input("enter y coor: ")
            coor = int(coor)
```

```
                except Exception as e:
                    print "Please enter a valid number"
                else:
                    if (coor >= 0) and (coor < board.width) and x_axis_coor == True:
                        valid = True
                    elif (coor >= 0) and (coor < board.height) and x_axis_coor == False:
                        valid = True
                    else:
                        print "Please enter a valid number"
            if i> 5000:
                print "over flow in get_coor"
            return coor

    def hit_ship(self, x, y, board, ships):
        for ship in ships:
            if board.grid[x][y].symbol == ship.name[0]:
                ship.sink_ship()
        return

    def check_won(self, ships):
        for ship in ships:
            if ship.sunk == False:
                return False
        return True

class Ship(object):
    def __init__(self, length, name):
        self.length = length
        self.name = name
        self.sunk = False

    def sink_ship(self):
        self.length -= 1
        if self.length == 0:
            self.sunk = True
            print "You sank my", self.name
            print
        return


class Board(object):
    def __init__(self):
        self.grid = []
        self.width = 10
        self.height = 10

    def check_cell_for_ship(self, x, y):
        # returns true if cell is not empty or hidden
        if self.grid[x][y].is_ship == False:
            return False
        else:
            return True

    def display(self):
        i = self.width - 1
        for x in range(0, self.width):
            print i,
            for y in range(0, self.height):
                print self.grid[y][i],
            i -= 1
            print
        print " ",
        for j in range(0, self.height):
            print j,
        print

    def get_board(self):
        cell_no = 0
        for x in range(0, self.width):
            self.grid.append([])
            for y in range(0, self.height):
                cell_no += 1
                self.grid[x].append(self.get_cell())
                self.grid[x][y].number = cell_no

class Shots_board(Board):
    def get_cell(self):
        return Shots_cell()

    def center_ship(self, ships_board, ships_x, ships_y):
        half_x = int(np.ceil(self.width/2))
        half_y = int(np.ceil(self.height/2))
        alignment_x = 0-half_x
        for shots_x in range(0, self.width):
            alignment_y = 0-half_y
            for shots_y in range(0, self.height):
                self.set_cells(ships_board, shots_x, shots_y, ships_x+alignment_x, ships_y+alignment_y)
                alignment_y += 1
            alignment_x += 1
        #self.display()

    def set_cells(self, board, shots_x, shots_y, ships_x, ships_y):
        if ships_x >= 0 and ships_x < self.width and ships_y >= 0 and ships_y < self.height:
            self.grid[shots_x][shots_y].is_hidden       = board.grid[ships_x][ships_y].is_hidden
            self.grid[shots_x][shots_y].is_ship         = board.grid[ships_x][ships_y].is_ship
            self.grid[shots_x][shots_y].symbol                     = board.grid[ships_x][ships_y].symbol
            self.grid[shots_x][shots_y].number                     = board.grid[ships_x][ships_y].number
        else:
            self.grid[shots_x][shots_y].off_the_board()

class Ships_board(Board):
    def get_cell(self):
        return Ships_cell()

class Cell(object):
    def __init__(self):
        self.is_ship = False
        self.is_hidden = True
        self.symbol = "X"
        self.number = 0

    def __str__(self):
        if self.is_hidden:
            return '-'
        return self.get_symbol()

    def get_symbol(self):
        if self.is_ship:
            return self.symbol
        else:
            return '~'

class Ships_cell(Cell):
        def set_cell(self, value, secondary):
            if secondary == None:
                self.is_hidden = value
            else:
                self.is_ship = value
                self.symbol = secondary ### while testing
```

```
class Shots_cell(Cell):
        def off_the_board(self):
                self.is_hidden = False
                off_board = True
                self.symbol =  "#"

        def get_symbol(self):
                if self.is_ship or self.symbol ==  "#":
                        return self.symbol
                else:
                        return '~'

class AI():
        def __init__():
                return


BattleshipsGame = Game()
BattleshipsGame.start()
```

Appendix 3

https://youtu.be/AP4lqbiNXk0

Appendix 4

https://youtu.be/qKCeBHJTjXE

Appendix 5

```
"""class AI():
    def __init__(self):
        self.x = []
        self.y = []

    def update(self,shots_board, turns):
        numbers = []
        self.one_dim_board = self.get_data(shots_board)
        for cell in self.one_dim_board:
        #   print cell,
            numbers.append(cell.number)
        #print
        self.y = np.array([self.get_weighted_array()])
        self.x = np.array([self.get_numerical_array()])

        self.list_display(numbers)
        self.list_display(self.one_dim_board)
        self.list_display(self.x)
        self.list_display(self.y)

        probs = self.procedure(turns)

        self.list_display(probs)

    def sigmoid(self,x):
        return 1/(1+np.exp(-x))

    def get_data(self, board):
        one_dim_board = []
        for column in range(0, shots_board.width):
            for row in range(0, shots_board.height):
                one_dim_board.append(shots_board.grid[column][row])
        return one_dim_board


    #def get_numerical_array(self):
        #           temp = []
        #           for cell in self.one_dim_board:
        #               if cell.is_hidden == True:
        #                   temp.append(0)
        #               elif cell == "~":
        #                   temp.append(0)
        #               elif cell.symbol == "#":
        #                   temp.append(-1)
        #               else:
        #                   temp.append(1)
                return temp


    def get_weighted_array(self):
        temp = []
        for cell in self.one_dim_board:
            if cell.is_hidden == False:
                temp.append(-1)
            elif cell.off_board == True:
                temp.append(-1)
            elif cell.is_ship:
                temp.append(1)
            else:
                temp.append(0)
        return temp

    #input

    #output
    def procedure(self):
                    #synapse
                    #syn0 = 2*np.random.random((25,25)) - 1
                    #syn1 = 2*np.random.random((25,1)) - 1
        #           syn0 = 2*np.random.random((25,1)) - 1
        #           syn1 = 2*np.random.random((1,25)) - 1
                    #training
        #           for i in xrange(1000000):
        #               l0 = self.x
        #               l1 = self.sigmoid(np.dot(l0, syn0))
        #               l2 = self.sigmoid(np.dot(l1, syn1))
        #               l2_error = self.y - l2
        #               if i % 10000:
        #                   print "error: " + str(np.mean(np.abs(l2_error)))
        #               l2_delta = l2_error*self.sigmoid(l2)#, deriv=True)
        #               l1_error = l2_delta.dot(syn1.T)
        #               l1_delta = l1_error*self.sigmoid(l1)#, deriv=True)
        #           #update weight
        #               syn1+= l1.T.dot(l2_delta)
        #               syn0+= l0.T.dot(l1_delta)
```

```
        #
        #                                   return l2


    def procedure(self, turns):
        biggest_val = self.one_dim_board.value[0]
        for j in len(self.one_dim_board):
            if self.one_dim_board[j].value > biggest_val:
                biggest_val = self.one_dim_board[j].value
                loc = j
        self.shot(loc, turns)

    def shot(self, location, turns):
        self.x[loc] = sigmoid(self.y[loc] + self.x[loc])#/self.turns


    def get_numerical_array(self):
        j=[]
        try:
            comms = open("memory_table.txt", "r")
        except IOError:
            print "Failed to open"
            return None
        j = comms.readlines()
        j = [line[:-1] for line in j]
        j = j,dtype = float
        comms.close()
        return j

    def list_display(self, list25):
        i=0
        for cell in list25:
            i += 1
            print cell,
            if i == 5:
                print
                i=0
        print

        #print l2"""


“””import numpy as np
```

```
class Neural_Network(object):
    def __init__(self, Lambda=0):
        #Define Hyperparameters
        self.inputLayerSize = 2
        self.outputLayerSize = 1
        self.hiddenLayerSize = 3

        #Weights (parameters)
        self.W1 = np.random.randn(self.inputLayerSize,self.hiddenLayerSize)
        self.W2 = np.random.randn(self.hiddenLayerSize,self.outputLayerSize)

        #Regularization Parameter:
        self.Lambda = Lambda

    def forward(self, X):
        #Propogate inputs though network
        self.z2 = np.dot(X, self.W1)
        self.a2 = self.sigmoid(self.z2)
        self.z3 = np.dot(self.a2, self.W2)
        yHat = self.sigmoid(self.z3)
        return yHat

    def sigmoid(self, z):
        #Apply sigmoid activation function to scalar, vector, or matrix
        return 1/(1+np.exp(-z))

    def sigmoidPrime(self,z):
        #Gradient of sigmoid
        return np.exp(-z)/((1+np.exp(-z))**2)

    def costFunction(self, X, y):
        #Compute cost for given X,y, use weights already stored in class.
        self.yHat = self.forward(X)
        J = 0.5*sum((y-self.yHat)**2)/X.shape[0] + (self.Lambda/2)*(np.sum(self.W1**2)+np.sum(self.W2**2))
        return J

    def costFunctionPrime(self, X, y):
        #Compute derivative with respect to W and W2 for a given X and y:
        self.yHat = self.forward(X)

        delta3 = np.multiply(-(y-self.yHat), self.sigmoidPrime(self.z3))
        #Add gradient of regularization term:
        dJdW2 = np.dot(self.a2.T, delta3)/X.shape[0] + self.Lambda*self.W2
        delta2 = np.dot(delta3, self.W2.T)*self.sigmoidPrime(self.z2)
        #Add gradient of regularization term:
        dJdW1 = np.dot(X.T, delta2)/X.shape[0] + self.Lambda*self.W1

        return dJdW1, dJdW2

    #Helper functions for interacting with other methods/classes
    def getParams(self):
        #Get W1 and W2 Rolled into vector:
        params = np.concatenate((self.W1.ravel(), self.W2.ravel()))
        return params

    def setParams(self, params):
        #Set W1 and W2 using single parameter vector:
        W1_start = 0
        W1_end = self.hiddenLayerSize*self.inputLayerSize
        self.W1 = np.reshape(params[W1_start:W1_end], \
                             (self.inputLayerSize, self.hiddenLayerSize))
        W2_end = W1_end + self.hiddenLayerSize*self.outputLayerSize
        self.W2 = np.reshape(params[W1_end:W2_end], \
                             (self.hiddenLayerSize, self.outputLayerSize))

    def computeGradients(self, X, y):
        dJdW1, dJdW2 = self.costFunctionPrime(X, y)
        return np.concatenate((dJdW1.ravel(), dJdW2.ravel()))”””
```

Appendix 6

```
### ai_write.py ###
import random
class ai(object):
def clear_file(file):
    file.seek(0)
    file.truncate()
x_coor = 1
y_coor = 1
ready = False

while True:
    x_coor = random.randint(0,4)
    y_coor = random.randint(0,4)
    ready = False
    while not ready:
        try:
            comms = open("communication.txt", "r")
        except IOError:
            print "Failed to open"
            ready = False
        else:
            if comms.read(5) == "Ready":
                print "ready"
                ready = True
        comms.close()

    comms = open("communication.txt", "w")
    clear_file(comms)
    comms.write("Done")
    comms.write("\n")
    comms.write(str(x_coor))
    comms.write("\n")
    comms.write(str(y_coor))
    comms.write("\n")
    comms.close()
    comms = open("communication.txt", "r")
```

Appendix 7

----S----	-10.7	11.3	-10.7	-10.7	-11	-10.7	-10.7	-10.7	-10.7
----S---~	-0.1	6	0	0	0	0	0	0	0
----S--~-	-0.1	440.8	0	0	0	0	0	0	0
----S--~~	-0.1	63.7	0	0	0	0	0	0	0
----S-~--	-2.6	7.9	-2.5	-2.5	-3	-2.5	-3	-2.5	-2.5
----S-~-~	-1.6	2.7	-1.5	-1.5	-2	-1.5	-2	-1.5	-2
----S-~~-	-0.1	77.9	0	0	0	0	0	0	0
----S-~~~	-0.1	2822.6	0	0	0	0	0	0	0
----S~---	-1.1	-1	-1.1	-1.1	-1	-1	-1.1	5.8	-1
----S~--~	-1	-1	-1	-1	-1	-1	-1	3.8	-1
----S~-~-	-0.1	257.2	0	0	0	0	0	0	0
----S~-~~	-0.1	91	0	0	0	0	0	0	0
----S~~--	-2.2	-2.2	-2.2	-2.2	-3	-3	-3	4.8	-2.1
----S~~-~	-2	-2	-2	-2	-2	-2	-2	-2.7	-2
----S~~~-	-0.1	92	0	0	0	0	0	0	0
----S~~~~	-0.1	614.5	0	0	0	0	0	0	0
---~S----	-2	10.7	-2	-2	-2	-2	-2	-2	-2
---~S---~	-1.7	-1.4	-1.7	-2	-2	-1.7	-1.7	-1.7	-2
---~S--~-	-0.1	78.9	0	0	0	0	0	0	0
---~S--~~	-0.1	58.6	0	0	0	0	0	0	0
---~S-~--	-1.7	-1.5	-1.6	-2	-2	-1.6	-2	-1.6	-1.6
---~S-~-~	-2	-2	-2	-2	-2	-2	-2	-1.8	-2
---~S-~~-	-0.1	47.9	0	0	0	0	0	0	0
---~S-~~~	-0.1	584.3	0	0	0	0	0	0	0
---~S~---	-1.5	-1.5	-1.5	-2	-2	-2	-1.5	-0.7	-1.5
---~S~--~	-1.6	-1.6	-1.6	-2	-2	-2	-1.6	-1.3	-2
---~S~-~-	-0.1	96.5	0	0	0	0	0	0	0
---~S~-~~	-0.1	51.3	0	0	0	0	0	0	0
---~S~~--	-0.1	-0.1	-0.1	-1	-1	-1	-1	0.5	0
---~S~~-~	-1.5	0.6	-1.4	-2	-2	-2	-2	-1.4	-2
---~S~~~-	-0.1	61.1	0	0	0	0	0	0	0
---~S~~~~	-0.1	316.2	0	0	0	0	0	0	0
--~-S----	-1.8	-1.8	-2	-1.8	-2	-1.8	-1.8	1.6	-1.8
--~-S---~	-2.3	-1.4	-3	-2.2	-3	-2.2	-2.2	-2.2	-3
--~-S--~-	-0.1	217.4	0	0	0	0	0	0	0
--~-S--~~	-0.1	58.4	0	0	0	0	0	0	0
--~-S-~--	-2.3	-0.1	-3	-2.2	-3	-2.2	-3	-2.2	-2.2
--~-S-~-~	-0.2	2	-1	-0.1	-1	-0.1	-1	-0.1	-1
--~-S-~~-	-0.1	72.1	0	0	0	0	0	0	0
--~-S-~~~	-0.1	576.2	0	0	0	0	0	0	0
--~-S~---	-0.1	-0.1	-1	-0.1	-1	-1	-0.1	4.9	0
--~-S~--~	-15	-14.9	-15	-15	-15	-15	-15	6.9	-15

--~-S~-~-	-0.1	95	0	0	0	0	0	0	0
--~-S~-~~	-0.1	2919.6	0	0	0	0	0	0	0
--~-S~~--	-1.8	-1.8	-2	-1.8	-2	-2	-2	-1.6	-1.7
--~-S~~-~	-5.3	-5.2	-6	-5.3	-6	-6	-6	2.3	-6
--~-S~~~-	-0.1	55.4	0	0	0	0	0	0	0
--~-S~~~~	-0.1	5004.1	0	0	0	0	0	0	0
--~~S----	-2.1	-2.1	-3	-3	-3	-2.1	-2.1	5	-2
--~~S---~	-1.3	-1.3	-2	-2	-2	-1.3	-1.3	0.4	-2
--~~S--~-	-0.1	87.1	0	0	0	0	0	0	0
--~~S--~~	-0.1	50.3	0	0	0	0	0	0	0
--~~S-~--	-0.1	0.5	0	0	0	0	0	0	0
--~~S-~-~	-1.3	-1.3	-2	-2	-2	-1.3	-2	-1.2	-2
--~~S-~~-	-0.1	28.5	0	0	0	0	0	0	0
--~~S-~~~	-0.1	342.5	0	0	0	0	0	0	0
--~~S~---	-0.7	3	-1	-1	-1	-1	-0.6	-0.6	-0.6
--~~S~--~	-3.8	-3.8	-4	-4	-4	-4	-3.8	-3.4	-4
--~~S~-~-	-0.1	28.3	0	0	0	0	0	0	0
--~~S~-~~	-0.1	820.5	0	0	0	0	0	0	0
--~~S~~--	-0.8	-0.8	-1	-1	-1	-1	-1	-0.5	-0.7
--~~S~~-~	-5.2	-5.2	-6	-6	-6	-6	-6	-2	-6
--~~S~~~-	-0.1	36.3	0	0	0	0	0	0	0
--~~S~~~~	-0.1	2157.3	0	0	0	0	0	0	0
-~--S----	-0.1	-1	-0.1	-0.1	-1	-0.1	-0.1	1853.1	0
-~--S---~	-0.1	-1	-0.1	-0.1	-1	-0.1	-0.1	292.6	0
-~--S--~-	0	0	0	0	0	0	0	0	0
-~--S--~~	0	0	0	0	0	0	0	0	0
-~--S-~--	-0.1	-1	-0.1	-0.1	-1	-0.1	-1	276.2	0
-~--S-~-~	-0.1	-1	-0.1	-0.1	-1	-0.1	-1	88.9	0
-~--S-~~-	0	0	0	0	0	0	0	0	0
-~--S-~~~	0	0	0	0	0	0	0	0	0
-~--S~---	-0.1	-1	-0.1	-0.1	-1	-1	-0.1	110.3	0
-~--S~--~	-0.1	-1	-0.1	-0.1	-1	-1	-0.1	36.8	0
-~--S~-~-	0	0	0	0	0	0	0	0	0
-~--S~-~~	0	0	0	0	0	0	0	0	0
-~--S~~--	-0.1	-1	-0.1	-0.1	-1	-1	-1	40.8	0
-~--S~~-~	-0.1	-1	-0.1	-0.1	-1	-1	-1	39.1	0
-~--S~~~-	0	0	0	0	0	0	0	0	0
-~--S~~~~	0	0	0	0	0	0	0	0	0
-~-~S----	-0.1	-1	-0.1	-1	-1	-0.1	-0.1	277.4	0
-~-~S---~	-0.1	-1	-0.1	-1	-1	-0.1	-0.1	71.1	0
-~-~S--~-	0	0	0	0	0	0	0	0	0
-~-~S--~~	0	0	0	0	0	0	0	0	0
-~-~S-~--	-0.1	-1	-0.1	-1	-1	-0.1	-1	78.3	0

-~-~S-~-~	-0.1	-1	-0.1	-1	-1	-0.1	-1	54.4	0
-~-~S-~~-	0	0	0	0	0	0	0	0	0
-~-~S-~~~	0	0	0	0	0	0	0	0	0
-~-~S~---	-0.1	-1	-0.1	-1	-1	-1	-0.1	36.1	0
-~-~S~--~	-0.1	-1	-0.1	-1	-1	-1	-0.1	35.2	0
-~-~S~-~-	0	0	0	0	0	0	0	0	0
-~-~S~-~~	0	0	0	0	0	0	0	0	0
-~-~S~~--	-0.1	-1	-0.1	-1	-1	-1	-1	26.4	0
-~-~S~~-~	-0.1	-1	-0.1	-1	-1	-1	-1	50.8	0
-~-~S~~~-	0	0	0	0	0	0	0	0	0
-~-~S~~~~	0	0	0	0	0	0	0	0	0
-~~-S----	-0.1	-1	-1	-0.1	-1	-0.1	-0.1	146.6	0
-~~-S---~	-0.1	-1	-1	-0.1	-1	-0.1	-0.1	70.4	0
-~~-S--~-	0	0	0	0	0	0	0	0	0
-~~-S--~~	0	0	0	0	0	0	0	0	0
-~~-S-~--	-0.1	-1	-1	-0.1	-1	-0.1	-1	61.1	0
-~~-S-~-~	-0.1	-1	-1	-0.1	-1	-0.1	-1	61.6	0
-~~-S-~~-	0	0	0	0	0	0	0	0	0
-~~-S-~~~	0	0	0	0	0	0	0	0	0
-~~-S~---	-0.1	-1	-1	-0.1	-1	-1	-0.1	34.3	0
-~~-S~--~	-0.1	-1	-1	-0.1	-1	-1	-0.1	1033.5	0
-~~-S~-~-	0	0	0	0	0	0	0	0	0
-~~-S~-~~	0	0	0	0	0	0	0	0	0
-~~-S~~--	-0.1	-1	-1	-0.1	-1	-1	-1	36.3	0
-~~-S~~-~	-0.1	-1	-1	-0.1	-1	-1	-1	633.8	0
-~~-S~~~-	0	0	0	0	0	0	0	0	0
-~~-S~~~~	0	0	0	0	0	0	0	0	0
-~~~S----	-0.1	-1	-1	-1	-1	-0.1	-0.1	40.8	0
-~~~S---~	-0.1	-1	-1	-1	-1	-0.1	-0.1	37	0
-~~~S--~-	0	0	0	0	0	0	0	0	0
-~~~S--~~	0	0	0	0	0	0	0	0	0
-~~~S-~--	-0.1	-1	-1	-1	-1	-0.1	-1	65.7	0
-~~~S-~-~	-0.1	-1	-1	-1	-1	-0.1	-1	44.1	0
-~~~S-~~-	0	0	0	0	0	0	0	0	0
-~~~S-~~~	0	0	0	0	0	0	0	0	0
-~~~S~---	-0.1	-1	-1	-1	-1	-1	-0.1	56.9	0
-~~~S~--~	-0.1	-1	-1	-1	-1	-1	-0.1	387.1	0
-~~~S~-~-	0	0	0	0	0	0	0	0	0
-~~~S~-~~	0	0	0	0	0	0	0	0	0
-~~~S~~--	-0.1	-1	-1	-1	-1	-1	-1	56.2	0
-~~~S~~-~	-0.1	-1	-1	-1	-1	-1	-1	546.6	0
-~~~S~~~-	0	0	0	0	0	0	0	0	0
-~~~S~~~~	0	0	0	0	0	0	0	0	0

~---S----	-1	3	-0.8	-0.8	-1	-0.8	-0.8	-0.8	-0.8
~---S---~	-1	8.5	-0.4	-0.4	-1	-0.4	-0.4	-0.4	-1
~---S--~-	-1	83.4	0	0	0	0	0	0	0
~---S--~~	-1	33.2	0	0	0	0	0	0	0
~---S-~--	-1	-0.2	-0.2	-0.2	-1	-0.2	-1	7	-0.1
~---S-~-~	-4	-3.7	-3.7	-3.7	-4	-3.7	-4	-3.3	-4
~---S-~~-	-1	99.8	0	0	0	0	0	0	0
~---S-~~~	-1	597.7	0	0	0	0	0	0	0
~---S~---	-2	-1.6	-1.6	-1.6	-2	-2	-1.6	2.2	-1.5
~---S~--~	-1	-0.4	-0.3	-0.3	-1	-1	-0.3	4.7	-1
~---S~-~-	-1	83	0	0	0	0	0	0	0
~---S~-~~	-1	56.2	0	0	0	0	0	0	0
~---S~~--	-1	0.1	-0.7	-0.7	-1	-1	-1	-0.7	-0.7
~---S~~-~	-1	0.3	-1	-1	-1	-1	-1	-1	-1
~---S~~~-	-1	28	0	0	0	0	0	0	0
~---S~~~~	-1	323.1	0	0	0	0	0	0	0
~--~S----	-3	0.9	-2.4	-3	-3	-2.4	-2.4	-2.4	-2.4
~--~S---~	-1	-0.3	-1	-1	-1	-1	-1	-1	-1
~--~S--~-	-1	55.9	0	0	0	0	0	0	0
~--~S--~~	-1	31.1	0	0	0	0	0	0	0
~--~S-~--	-7	-3.4	-6.9	-7	-7	-6.9	-7	-6.9	-6.9
~--~S-~-~	-2	8.2	-1.7	-2	-2	-1.7	-2	-1.7	-2
~--~S-~~-	-1	1535.4	0	0	0	0	0	0	0
~--~S-~~~	-1	4779.5	0	0	0	0	0	0	0
~--~S~---	-1	3.3	-0.9	-1	-1	-1	-0.9	-0.9	-0.9
~--~S~--~	-3	-1.2	-2.1	-3	-3	-3	-2.1	-2.1	-3
~--~S~-~-	-1	34.8	0	0	0	0	0	0	0
~--~S~-~~	-1	52.2	0	0	0	0	0	0	0
~--~S~~--	-5	-4.9	-4.9	-5	-5	-5	-5	-4.3	-4.9
~--~S~~-~	-1	4.9	-0.7	-1	-1	-1	-1	-0.7	-1
~--~S~~~-	-1	455.4	0	0	0	0	0	0	0
~--~S~~~~	-1	2011.3	0	0	0	0	0	0	0
~-~-S----	-2	-0.7	-2	-1.5	-2	-1.5	-1.5	-1.5	-1.5
~-~-S---~	-1	4.7	-1	-0.6	-1	-0.6	-0.6	-0.6	-1
~-~-S--~-	-1	71.6	0	0	0	0	0	0	0
~-~-S--~~	-1	37.3	0	0	0	0	0	0	0
~-~-S-~--	-3	-0.8	-3	-2.2	-3	-2.2	-3	-2.2	-2.2
~-~-S-~-~	-1	-1	-1	-1	-1	-1	-1	1.6	-1
~-~-S-~~-	-1	33.1	0	0	0	0	0	0	0
~-~-S-~~~	-1	333.7	0	0	0	0	0	0	0
~-~-S~---	-1	-1	-1	-1	-1	-1	-1	3	-0.9
~-~-S~--~	-9	-8.9	-9	-8.9	-9	-9	-8.9	-7.5	-9
~-~-S~-~-	-1	52.9	0	0	0	0	0	0	0

~-~-S~-~~	-1	635.1	0	0	0	0	0	0	0
~-~-S~~--	-1	3	-1	-0.5	-1	-1	-1	-0.5	-0.5
~-~-S~~-~	-1	-0.4	-1	-0.4	-1	-1	-1	7.9	-1
~-~-S~~~-	-1	38.7	0	0	0	0	0	0	0
~-~-S~~~~	-1	2348.9	0	0	0	0	0	0	0
~-~~S----	-1	-1	-1	-1	-1	-1	-1	1.2	-0.9
~-~~S---~	-1	-0.8	-1	-1	-1	-0.8	-0.8	-0.1	-1
~-~~S--~-	-1	47.5	0	0	0	0	0	0	0
~-~~S--~~	-1	53.9	0	0	0	0	0	0	0
~-~~S-~--	-6	-1	-6	-6	-6	-5.4	-6	-5.3	-5.4
~-~~S-~-~	-2	5.7	-2	-2	-2	-1.7	-2	-1.7	-2
~-~~S-~~-	-1	399.8	0	0	0	0	0	0	0
~-~~S-~~~	-1	2028.4	0	0	0	0	0	0	0
~-~~S~---	-1	-0.2	-1	-1	-1	-1	-0.4	-0.4	-0.4
~-~~S~--~	-5	-5	-5	-5	-5	-5	-5	1.1	-5
~-~~S~-~-	-1	57	0	0	0	0	0	0	0
~-~~S~-~~	-1	620.3	0	0	0	0	0	0	0
~-~~S~~--	-7	-0.8	-7	-7	-7	-7	-7	-6.3	-6.3
~-~~S~~-~	-1	-0.3	-1	-1	-1	-1	-1	20.9	-1
~-~~S~~~-	-1	410.2	0	0	0	0	0	0	0
~-~~S~~~~	-1	5291.3	0	0	0	0	0	0	0
~~--S----	-1	-1	-0.1	-0.1	-1	-0.1	-0.1	284.4	0
~~--S---~	-1	-1	-0.1	-0.1	-1	-0.1	-0.1	89.9	0
~~--S--~-	0	0	0	0	0	0	0	0	0
~~--S--~~	0	0	0	0	0	0	0	0	0
~~--S-~--	-1	-1	-0.1	-0.1	-1	-0.1	-1	32.3	0
~~--S-~-~	-1	-1	-0.1	-0.1	-1	-0.1	-1	37.6	0
~~--S-~~-	0	0	0	0	0	0	0	0	0
~~--S-~~~	0	0	0	0	0	0	0	0	0
~~--S~---	-1	-1	-0.1	-0.1	-1	-1	-0.1	47.7	0
~~--S~--~	-1	-1	-0.1	-0.1	-1	-1	-0.1	26.7	0
~~--S~-~-	0	0	0	0	0	0	0	0	0
~~--S~-~~	0	0	0	0	0	0	0	0	0
~~--S~~--	-1	-1	-0.1	-0.1	-1	-1	-1	59.5	0
~~--S~~-~	-1	-1	-0.1	-0.1	-1	-1	-1	57	0
~~--S~~~-	0	0	0	0	0	0	0	0	0
~~--S~~~~	0	0	0	0	0	0	0	0	0
~~-~S----	-1	-1	-0.1	-1	-1	-0.1	-0.1	75.7	0
~~-~S---~	-1	-1	-0.1	-1	-1	-0.1	-0.1	56.3	0
~~-~S--~-	0	0	0	0	0	0	0	0	0
~~-~S--~~	0	0	0	0	0	0	0	0	0
~~-~S-~--	-1	-1	-0.1	-1	-1	-0.1	-1	2426.3	0
~~-~S-~-~	-1	-1	-0.1	-1	-1	-0.1	-1	870.8	0

~~-~S-~~-	0	0	0	0	0	0	0	0	0
~~-~S-~~~	0	0	0	0	0	0	0	0	0
~~-~S~---	-1	-1	-0.1	-1	-1	-1	-0.1	58.4	0
~~-~S~--~	-1	-1	-0.1	-1	-1	-1	-0.1	43.2	0
~~-~S~-~-	0	0	0	0	0	0	0	0	0
~~-~S~-~~	0	0	0	0	0	0	0	0	0
~~-~S~~--	-1	-1	-0.1	-1	-1	-1	-1	750	0
~~-~S~~-~	-1	-1	-0.1	-1	-1	-1	-1	643	0
~~-~S~~~-	0	0	0	0	0	0	0	0	0
~~-~S~~~~	0	0	0	0	0	0	0	0	0
~~-S----	-1	-1	-1	-0.1	-1	-0.1	-0.1	2836.5	0
~~-S---~	-1	-1	-1	-0.1	-1	-0.1	-0.1	599.2	0
~~-S--~-	0	0	0	0	0	0	0	0	0
~~-S--~~	0	0	0	0	0	0	0	0	0
~~-S-~--	-1	-1	-1	-0.1	-1	-0.1	-1	617	0
~~-S-~-~	-1	-1	-1	-0.1	-1	-0.1	-1	313.6	0
~~-S-~~-	0	0	0	0	0	0	0	0	0
~~-S-~~~	0	0	0	0	0	0	0	0	0
~~-S~---	-1	-1	-1	-0.1	-1	-1	-0.1	579.9	0
~~-S~--~	-1	-1	-1	-0.1	-1	-1	-0.1	4989.9	0
~~-S~-~-	0	0	0	0	0	0	0	0	0
~~-S~-~~	0	0	0	0	0	0	0	0	0
~~-S~~--	-1	-1	-1	-0.1	-1	-1	-1	337.7	0
~~-S~~-~	-1	-1	-1	-0.1	-1	-1	-1	2035.3	0
~~-S~~~-	0	0	0	0	0	0	0	0	0
~~-S~~~~	0	0	0	0	0	0	0	0	0
~~S----	-1	-1	-1	-1	-1	-0.1	-0.1	587.8	0
~~S---~	-1	-1	-1	-1	-1	-0.1	-0.1	325.6	0
~~S--~-	0	0	0	0	0	0	0	0	0
~~S--~~	0	0	0	0	0	0	0	0	0
~~S-~--	-1	-1	-1	-1	-1	-0.1	-1	5248.3	0
~~S-~-~	-1	-1	-1	-1	-1	-0.1	-1	2311.8	0
~~S-~~-	0	0	0	0	0	0	0	0	0
~~S-~~~	0	0	0	0	0	0	0	0	0
~~S~---	-1	-1	-1	-1	-1	-1	-0.1	325.6	0
~~S~--~	-1	-1	-1	-1	-1	-1	-0.1	2065.7	0
~~S~-~-	0	0	0	0	0	0	0	0	0
~~S~-~~	0	0	0	0	0	0	0	0	0
~~S~~--	-1	-1	-1	-1	-1	-1	-1	2266.5	0
~~S~~-~	-1	-1	-1	-1	-1	-1	-1	4636.5	0
~~S~~~-	0	0	0	0	0	0	0	0	0
~~S~~~~	0	0	0	0	0	0	0	0	0