ENPM808F
HOMEWORK 4

Nithish Kumar S (116316958)

Project Overview:

This report contains details of the Q-learning implementation for the game of dots and boxes.The goal is train the game using self-play and testing its performance with a random agent for a 2x2 grid and 3x3 grid.

Inferences:

The 2x2 grid took lesser number of iterations in the training to give a good winning ratio than the 3x3 grid.

2x2 grid - 2^12 states

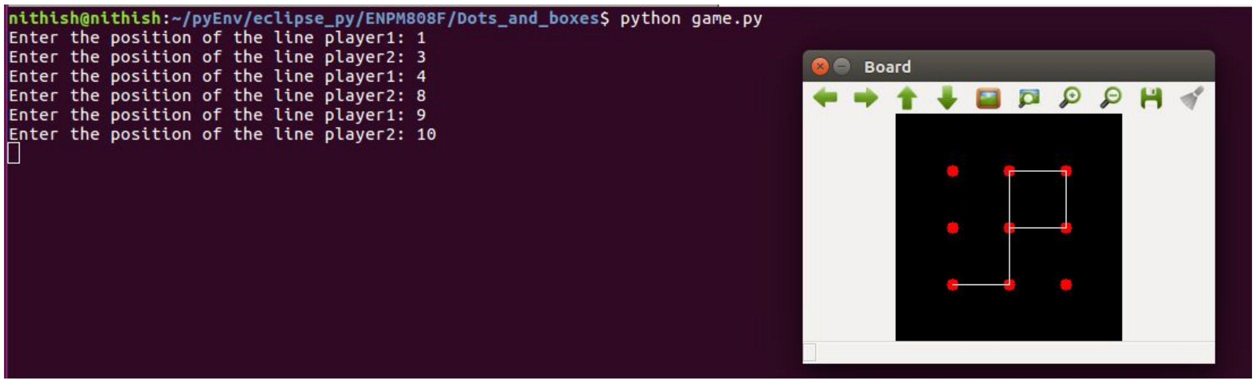
3x3 grid - 2^24 states

This is because the 2x2 grid has lesser number of states, so smaller the Q-table size. So it was able to fill the table faster than the other.

Implementation of a 2x2 Grid:

The GUI

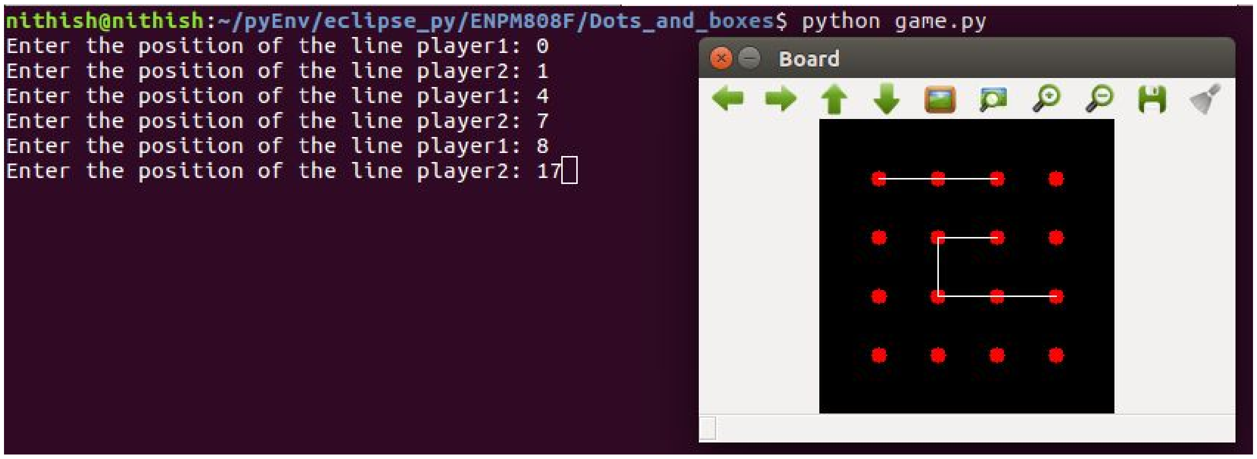
This is a user playable version where each player has to enter the line number to play the game. It starts with 0 going sequentially from top to bottom, horizontal first and vertical next.



```
nithish@nithish:~/pyEnv/eclipse_py/ENPM808F/Dots_and_boxes$ python game.py
Enter the position of the line player1: 1
Enter the position of the line player2: 3
Enter the position of the line player1: 4
Enter the position of the line player2: 8
Enter the position of the line player1: 9
Enter the position of the line player2: 10
```

Implementation of a 3x3 Grid:

The GUI



```
nithish@nithish:~/pyEnv/eclipse_py/ENPM808F/Dots_and_boxes$ python game.py
Enter the position of the line player1: 0
Enter the position of the line player2: 1
Enter the position of the line player1: 4
Enter the position of the line player2: 7
Enter the position of the line player1: 8
Enter the position of the line player2: 17
```

Qtable training process:

The states are stored as a dictionary of state of all the line in the game in a row major form. State is flattened to a 1D array for a computational efficiency.

Q table has the reward for each action for each state. So the size of qtable is the product of total number of possible states and the number of actions.
2x2 grid has 12 possible lines and so

no of states	: 2^12 = 4096 states
Actions	: 12
Qtable size	: 49152

3x3 grid has 24 possible lines and so

no of states	: 2^24 = 1677216 states
Actions	: 24
Qtable size	: 402653184

Given the size of q table I chose to use a dictionary data structure. Dictionary is also effective since the key can be accessed directly without the need to search.

Rewards:

Win :	+5
Box :	+1
Other:	0

Training:

The training process is simple and it utilizes the bellman equation to update the rewards for each action corresponding to a state. Initially the agent has to explore all possible states and action spaces and later it has to exploit the known Q value. So I implemented an epsilon greedy algorithm which explores initially using random states and later goes only to actions with highest q values.

Q table is updated in each iteration of the game.

Decay rate helps us control this and I set it to .01
I store the Qtable in a text file, so that training need not be done everytime we need to run the game.

Some results:
Greedy policy

Epsilon - 1.0
Decay_rate - 0.01
Discount factor - 0.9
Learning rate - .35

2x2 grid:
Training 100 iterations:
Wins : 52 / 100
Training 1000 iterations:
Wins : 68 / 100
Training 10000 iterations:
Wins : 85 / 100

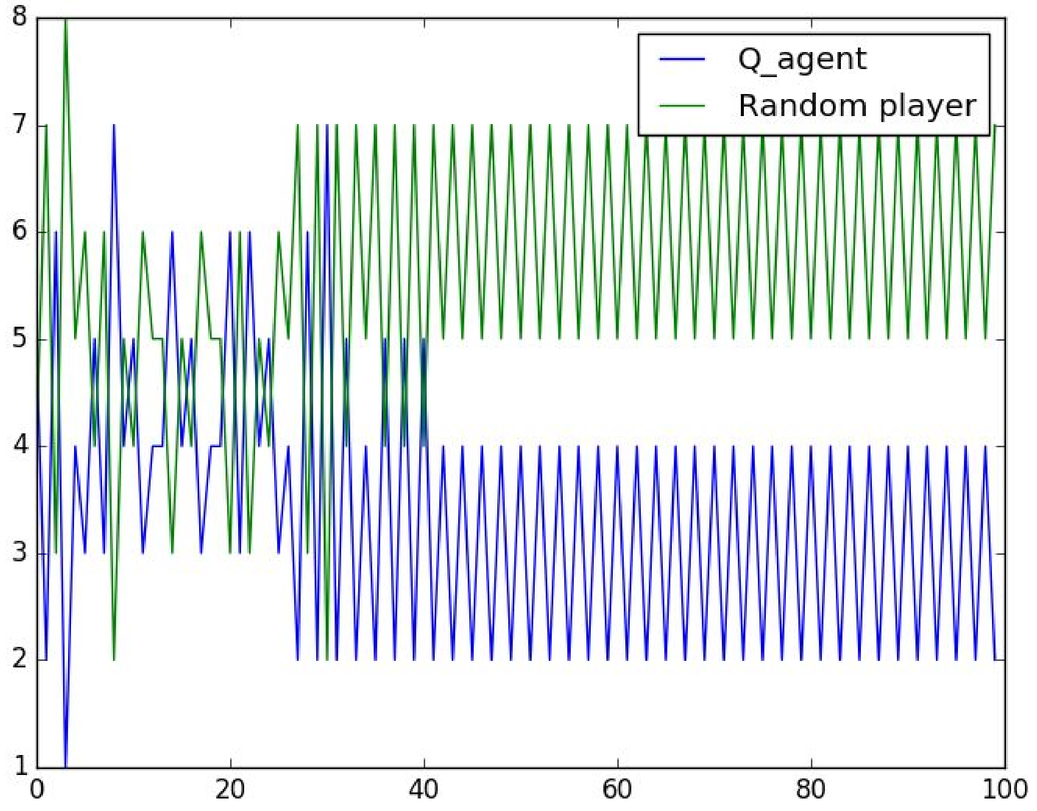
3x3 grid:

Training 100 iterations:
Wins : 16 / 100
Training 1000 iterations:
Wins : 55 / 100
Training 10000 iterations:
Wins : 63 /100

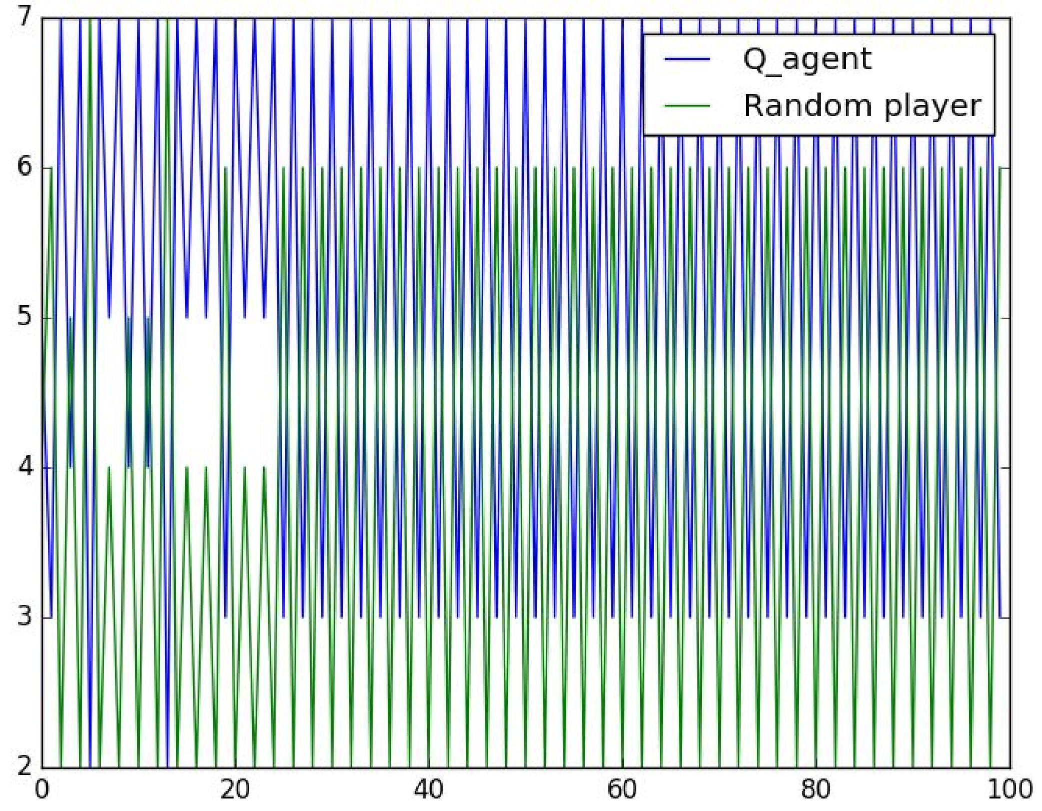
These results were not always consistent. These are some of the best results.

Using the Q values from the 2x2 grid for the corresponding seemed to improve the performance a notch.

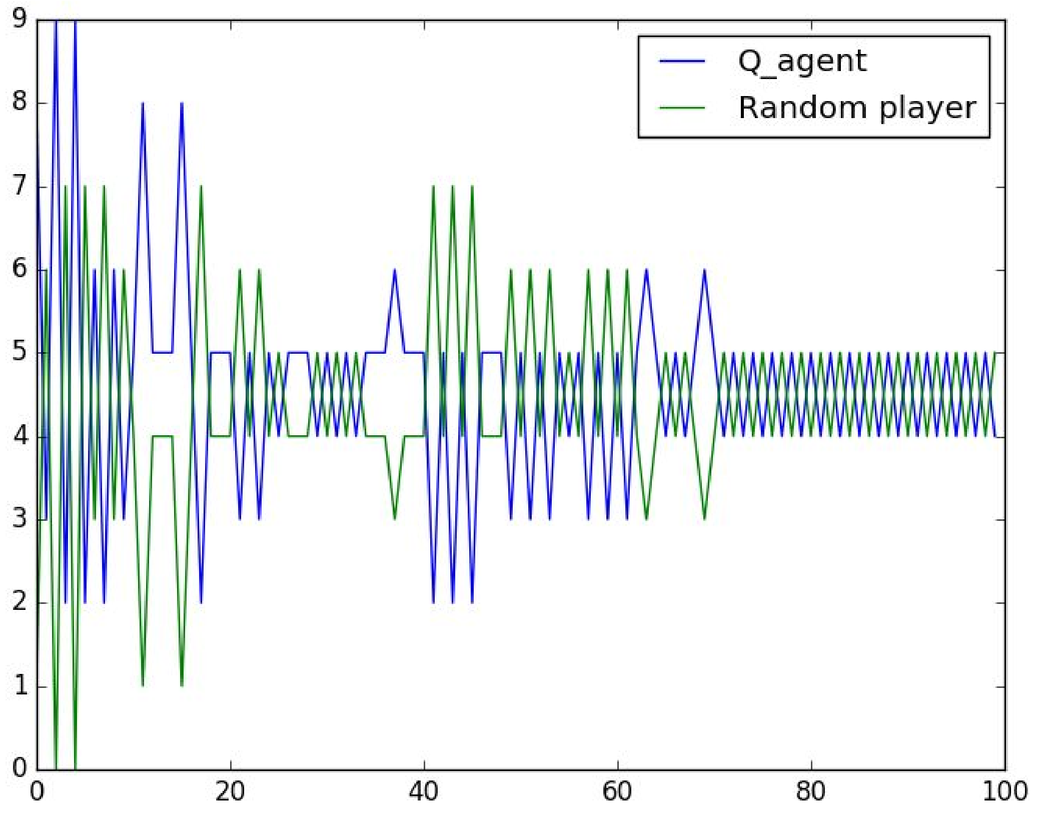
Out of 9 maximum boxes that can be scored in the 3x3 grid, the output is shown here for a test run of 100 games.



After 100 iterations



After 1000 iterations



After 10000 iterations

I clearly notice a difference in output when the learning rate is changed. For smaller learning rates it takes longer to learn but the output is consistent then.

Functional Approximation Implementation:
The difference lies in updating and using the neural network instead of the qtable in each iteration of the game. The Bellman equation is updated using the neural network

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a)w_i$$

Weights are updated just like the qtable but using the following equations.

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$
$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

Since the neural network computes the output it is not accurate as the q table but can manage larger number of states.
But I couldn't get the output to converge using my code. So I have just uploaded the code where I used the Multi layered Perceptron from the scipy toolbox.

It was bad for 100, 1000 and 10000 iterations as well. The Q agent kept losing to the random agent.