Part 1.

Below, in Figure 1, is a plot of the Q-Learning agent's progress when using a Q-Table implementation. The data was collected by having 2 Q-Learning agents play against one another. Both agents shared and updated the Q table throughout playing. An epsilon value was used to determine how often the Q-Learning agents made the "best" move. The epsilon value used throughout this analysis is .1. Meaning the Q-Learning agent explored using other moves about 10% of the time. The plot below shows the progress of the Q-learning agent over 10000 games. Every 1000 games, training was stopped and 1000 games were played pitting the Q-Learning agent against an opponent making random moves. During this evaluation period, the epsilon value was set to 0 to assess the performance of the agent making it's best moves always.

It can be seen that the agent learns to better the opponent making random moves within the first 1000 games. However, there also seems to be some sort of saturation as the number of games won by the Q Learning agent plateaus around the 60% mark. One might expect a more gradual increase in Q-Learning performance if a higher epsilon value was used, due to the increased amount of exploring. One may also find that the level of saturation may be higher due to the fact that one of those exploratory moves might reveal more winning strategies. A rerun was done using a value of .4 for epsilon. The results were captured in figure 2. It seems that the separation between the Q-Learning agent and the random actor increased at a much faster pace. This suggests that the exploration allowed the Agent to discover some better strategies for winning. This plot too seems to plateau around the 60% win range, however the number of times the random agent wins is markedly lower than when epsilon was .1.

While the variability in performance throughout the training makes observation of trends slightly difficult. Both iterations of the Q-Table show that they begin acting ins a somewhat random fashion and learn to improve. The Q Agent with epsilon = 0.1 seemed to start it's plateau around 2000 games while the agent with epsilon = 0.4 had even more variation makingit hard to identify a clear plateau point.

There is a section below titled "Implementing the Q Table" which discusses some hurdles that were met when implementing the Q-Table.

Part 2.

Below, in Figures 3-5, you'll find the results of the Q-Learning agent in the 3x3 case. Figure 3 contains the results of an agent that started from scratch. Figure 4 shows an agent that used some information learned from the results in the 2x2 case. Finally, Figure 5 shows how the Q-Table approach will start to separate from random moves eventually (after the 10,000 game cap). Now with these figures it is important to note that the Q-Learning agents were in fact learning while playing one another, it is evident via the plot of 100,000 games. Looking at the first two graphs of 10,000 games however it seems to be no better than the randomly acting agent. This is a drastically slower rate of increased performance compared to the 2x2. One

could contribute this difference to the fact that there are 4096 possible states in a 2x2 while there are 16,777,216 states in the 3x3. Now in the 2x2 the total number of possible state action pairs came out to be ~25,000 which still is a small fraction of just the states possible in a 3x3. The total combinations of state action pairs of a 3x3 is a large number compared to the number of state action pairs one may come across during the 10,000 games of training. When evaluating the performance during the training, the Q-Learning agent is then more likely to come across state action pairs is has not seen before. In that case it will essentially act randomly because all of the Q values returned from the table will be 0. This seems to be what is happening in Figures 3 and 4 below. The Q learning agent has not populated enough of the table during the 10,000 iterations of training, to properly choose moves most of the time. Note even with the seeded information from the 2x2 this was not enough to win games in the 3x3. Discussion regarding how the 2x2 was used to seed the 3x3 is captured in the Implementing the Q Table section below all figures.

Part 3.

Below in Figure 6 illustrates the performance of a Q Learner using a functional approximation instead of a Q Table. Some more details about how the functional approximation was implemented are provided in the Functional Approximation section below. Now I found that some of the trained functional approximations also showed significant variability. Figure 6. Shows the performance of the best functional approximation I could find for the 2x2 game. From this point I attempted to continue training the agent to further improve it's performance. However, as can be seen here the performance dips quite a bit and seems to plateau around 55%. For this training, 250 games were played before retraining the network. In an attempt to avoid heavily biasing the data to the newly found data points, the original set of data that was used to generate the functional approximation was saved and used. During the 10,000 games of training, the state action pairs that already existed within the training data were replaced by new values. The old values remained. At the 250 game mark all the data was used to fit a new functional approximation. These functional approximations never seemed to be as good as the original one that was seeded from the Q table. Essentially, this resulted in me creating approximations of my approximations that seemed to get worse as training continued. In retrospect I feel that the number of games used in the batch size may have actually contributed negatively to the results. Once the functional approximation was created I should not have used old data mixed with new data points to train a new functional approximation. (I only felt the need to keep the old data because the batch size of 250 games was not enough data points alone to make a sufficient functional approximation.) Larger batch sizes with the new functional approximations may have improved results but that hypothesis was not tested during this study.

Part 4.

Figure 8. Illustrates the results of the table seeded functional approximation. Again the training data was collected during 10,000 training games using the Q Table. Like figure 6, Figure 8 shows the performance over 10,000 games (to be consistent with the other results present in this report) but the Functional approximation was not modified in any way during those games. The results coincident to Figure 7 for the 2x2 were not captured. I attempted a similar process of updating the functional approximation every 250 games but my kernel crashed and I had many issues just reviving my virtual box. However, the code is still capable of generating those plots. I think that instead of using all of the training data that I've accumulated over the set of games played in one batch, I would use a fixed batch size. This may help to avoid what I believe to be the cause of the kernel crash, running out of system memory.

Now the results functional approximation were very promising. Showing it could consistently outperform the random agent just over 80% of the time. Comparing these results to the ones observed in Figure 3 and Figure 5. suggests that the functional approximation is better suited for this game. The Q-Table limits the Q-Agents to situations it has come across in it's training. If it has not had enough games to explore and properly populate the table then it can not be an effective agent. The functional network on the other hand can observe patterns from a significantly smaller set of data to build a model. Because this model is pattern based and not memory based it will be actually be able to handle state action pairs it has not seen before. This is why I believe the functional approximation does so much better than the Q-Table for the allotted training done for this report.

Figures 9 and 10 show the 2 Q agents playing against each other, on using a Q-Table while the other uses a functional approximation for a 2x2 and 3x3 game board respectively. Originally epsilon was set to 0 for the computers to play the best games. However, because the two agents always play their best moves the games all end up being duplicate games played repeatedly. In order to try to stifle this effect I had them play against each other using epsilon values of .05. These are the results I got.

Please observe the figures below as well as some discussion regarding implementation details and hurdles that was handled during this project.

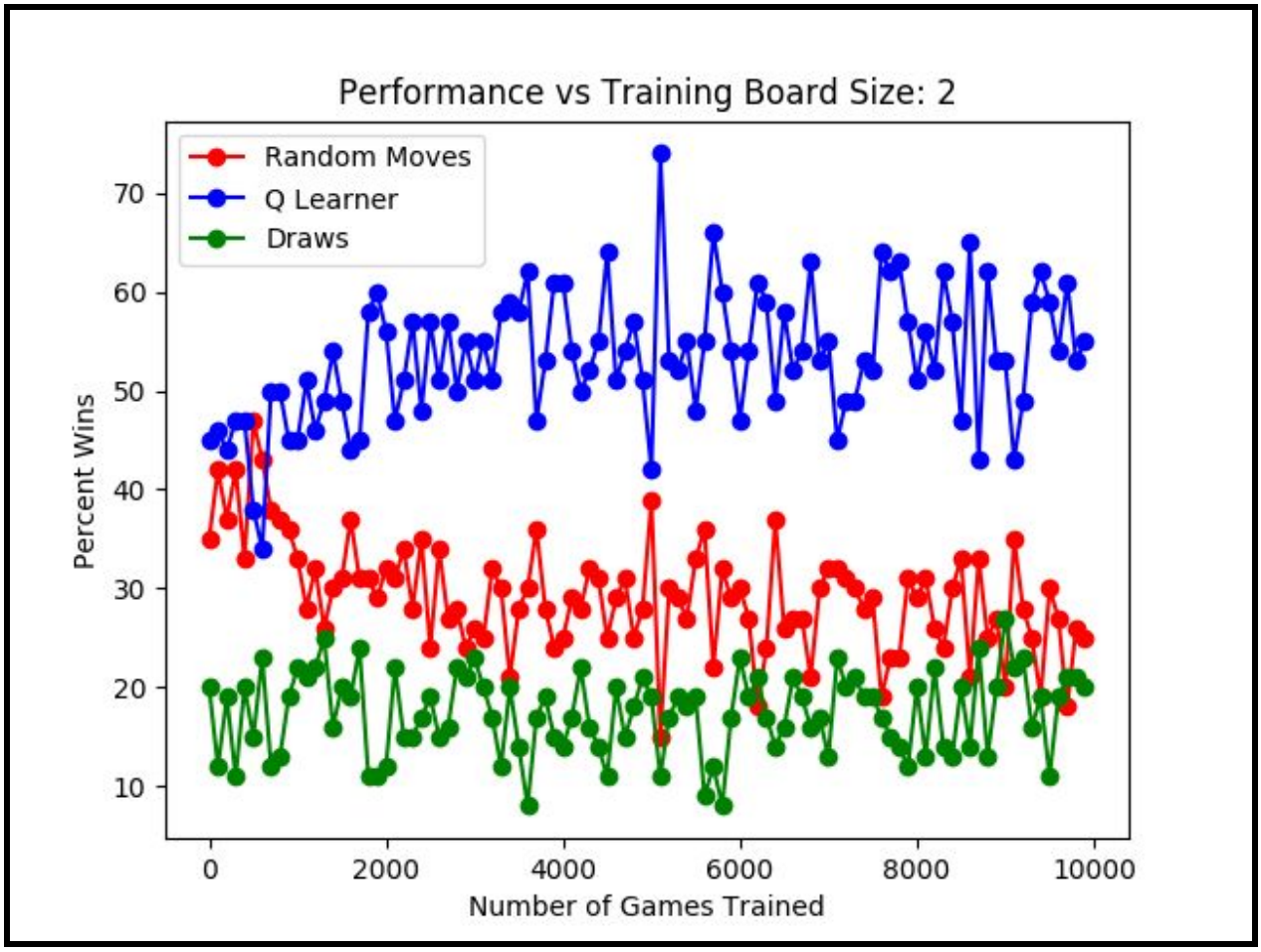

Figure 1. Results of a Q-Learning Agent vs. Random Moves Agent, $\varepsilon = 0.1$

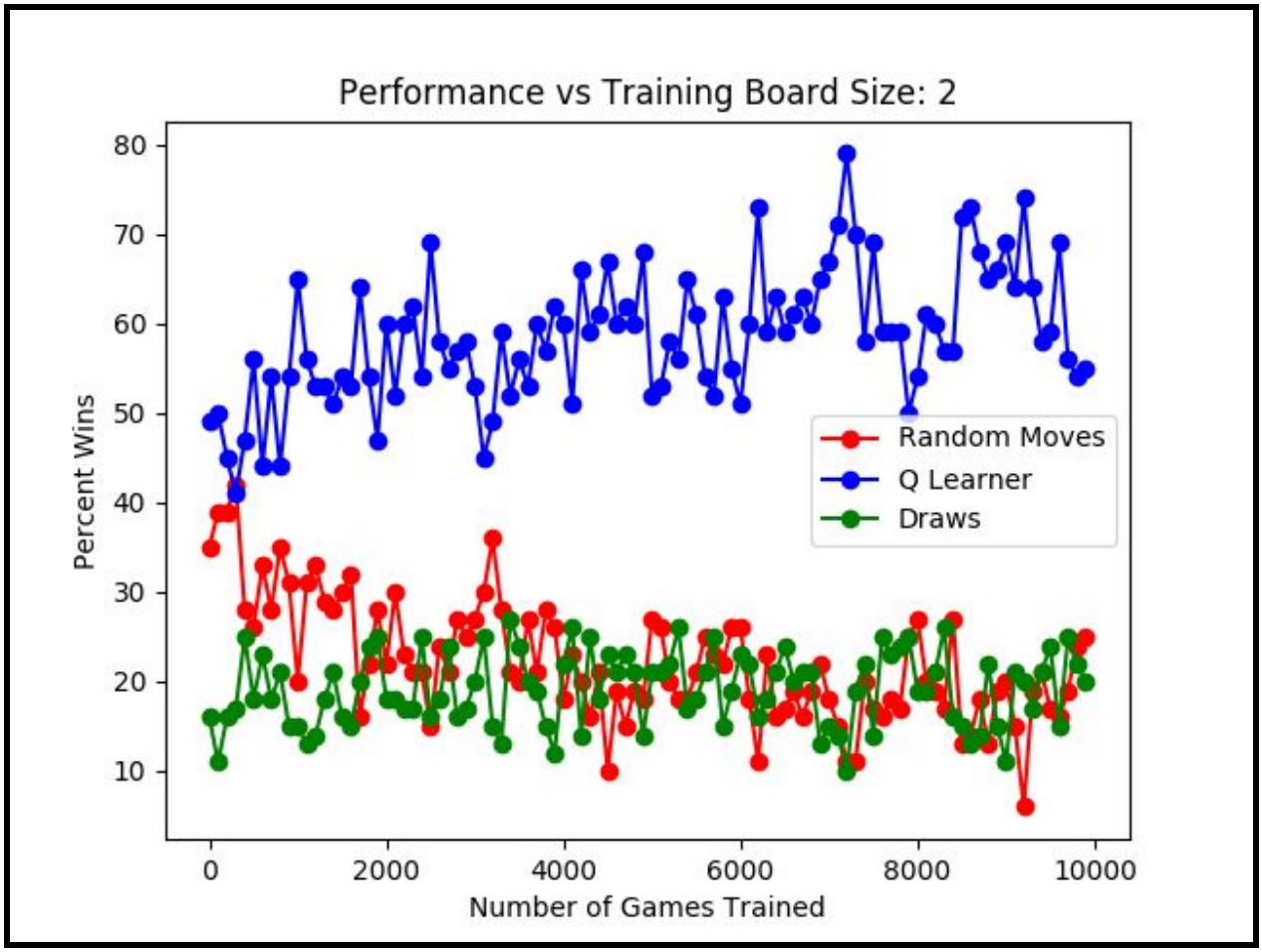

Figure 2. Results of a Q-Learning Agent vs. Random Moves Agent, $\varepsilon = 0.4$

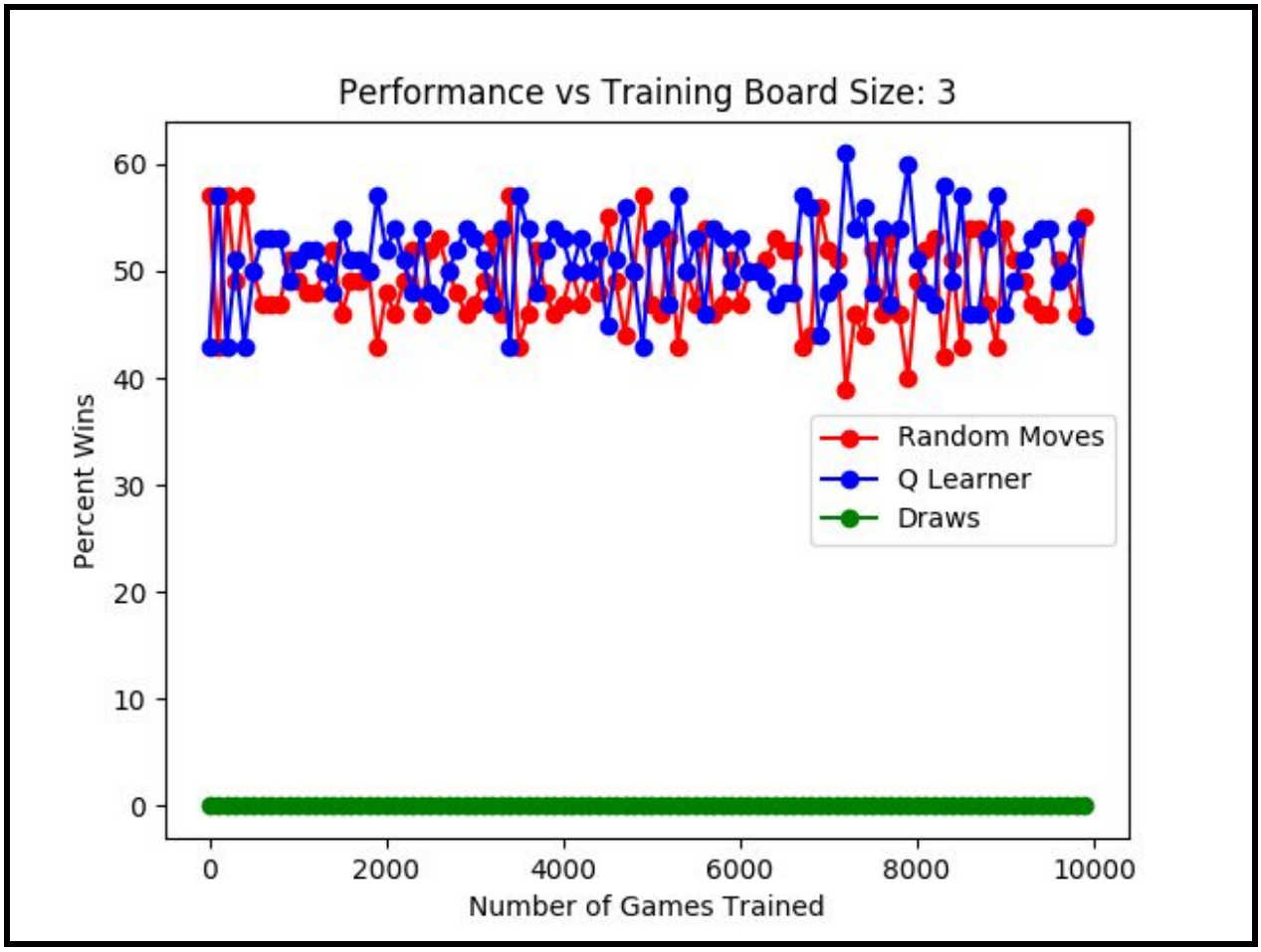

Figure 3. Results of a Q-Learning Agent vs. Random Moves Agent on a 3x3

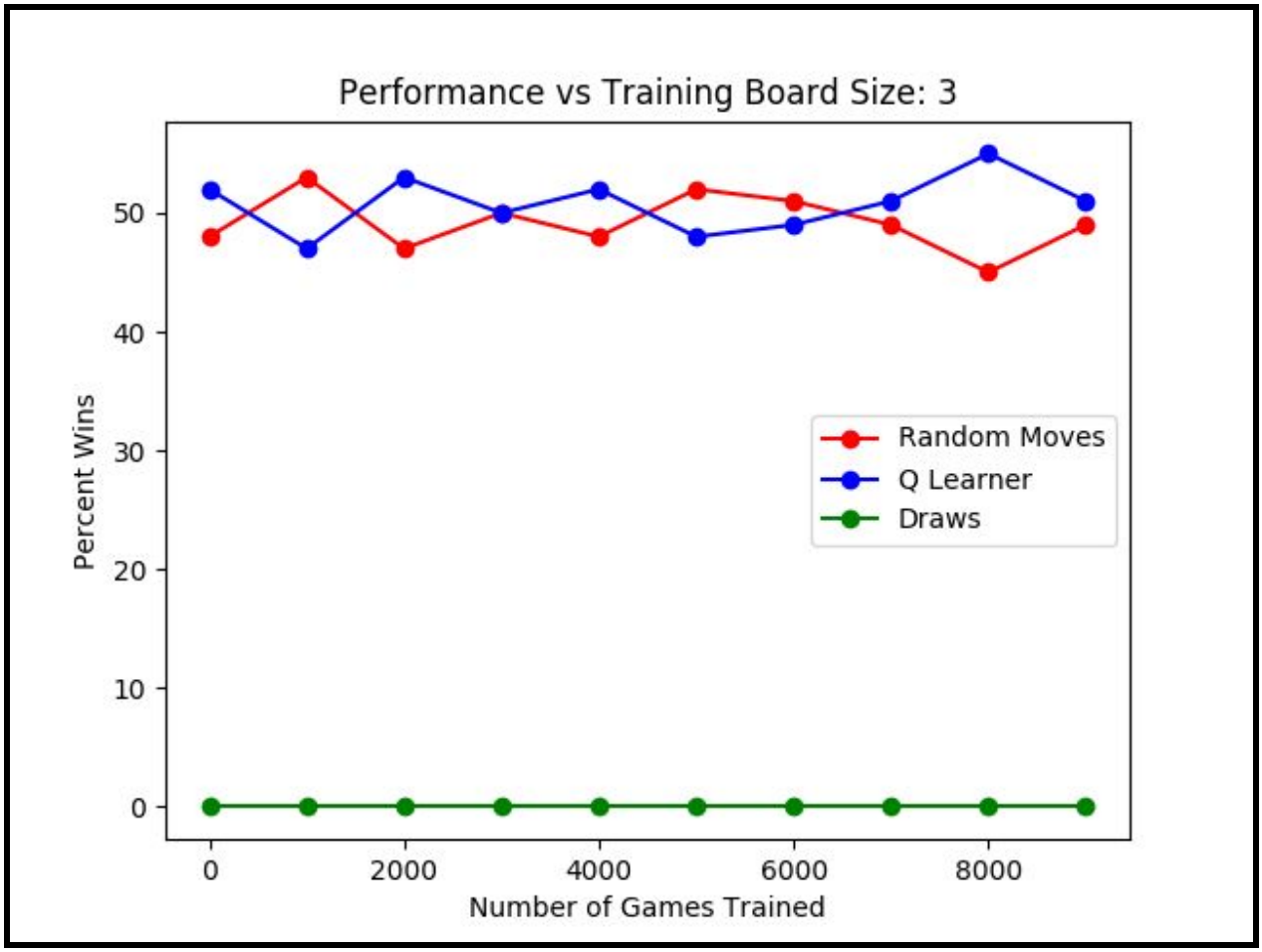

Figure 4. Results of a Q-Learning Agent vs. Random Moves Agent on a 3x3 With Seeding from the 2x2

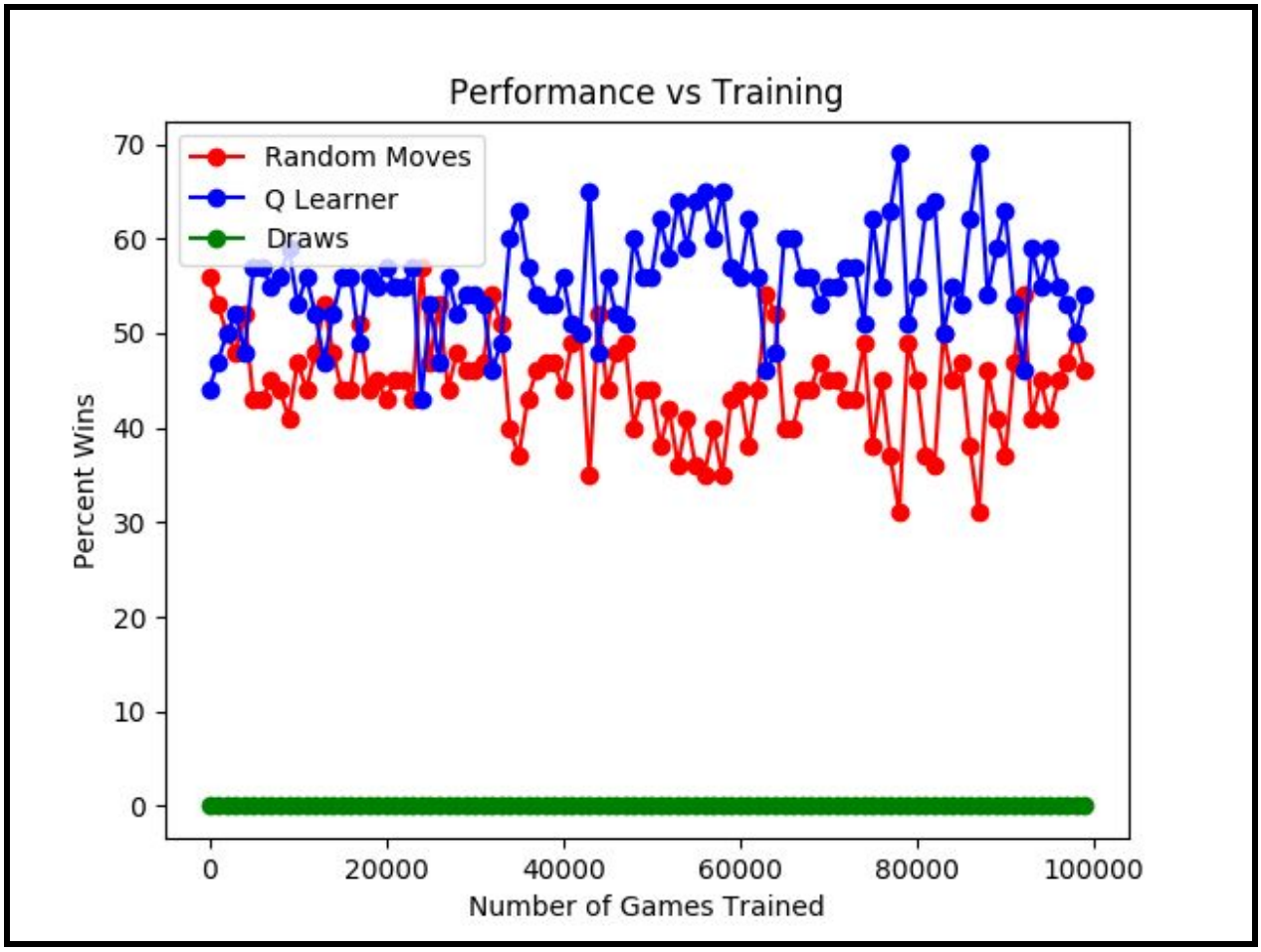

Figure 5. Results of a Q-Learning Agent vs. Random Moves Agent on a 3x3 100000

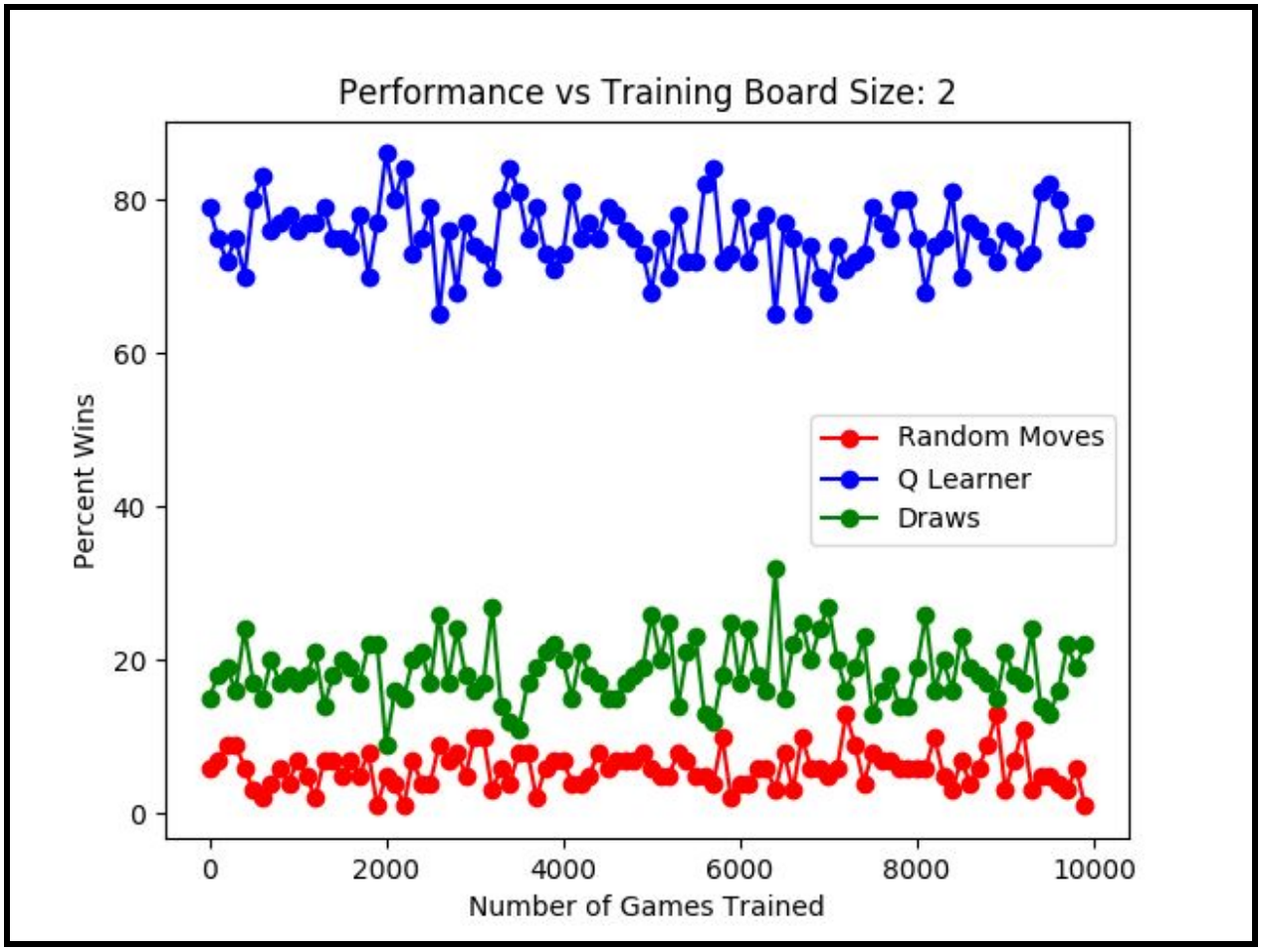

Figure 6. Results of a Q-Learning Agent vs. Random Moves Agent with Fixed Fcn Approx 2x2

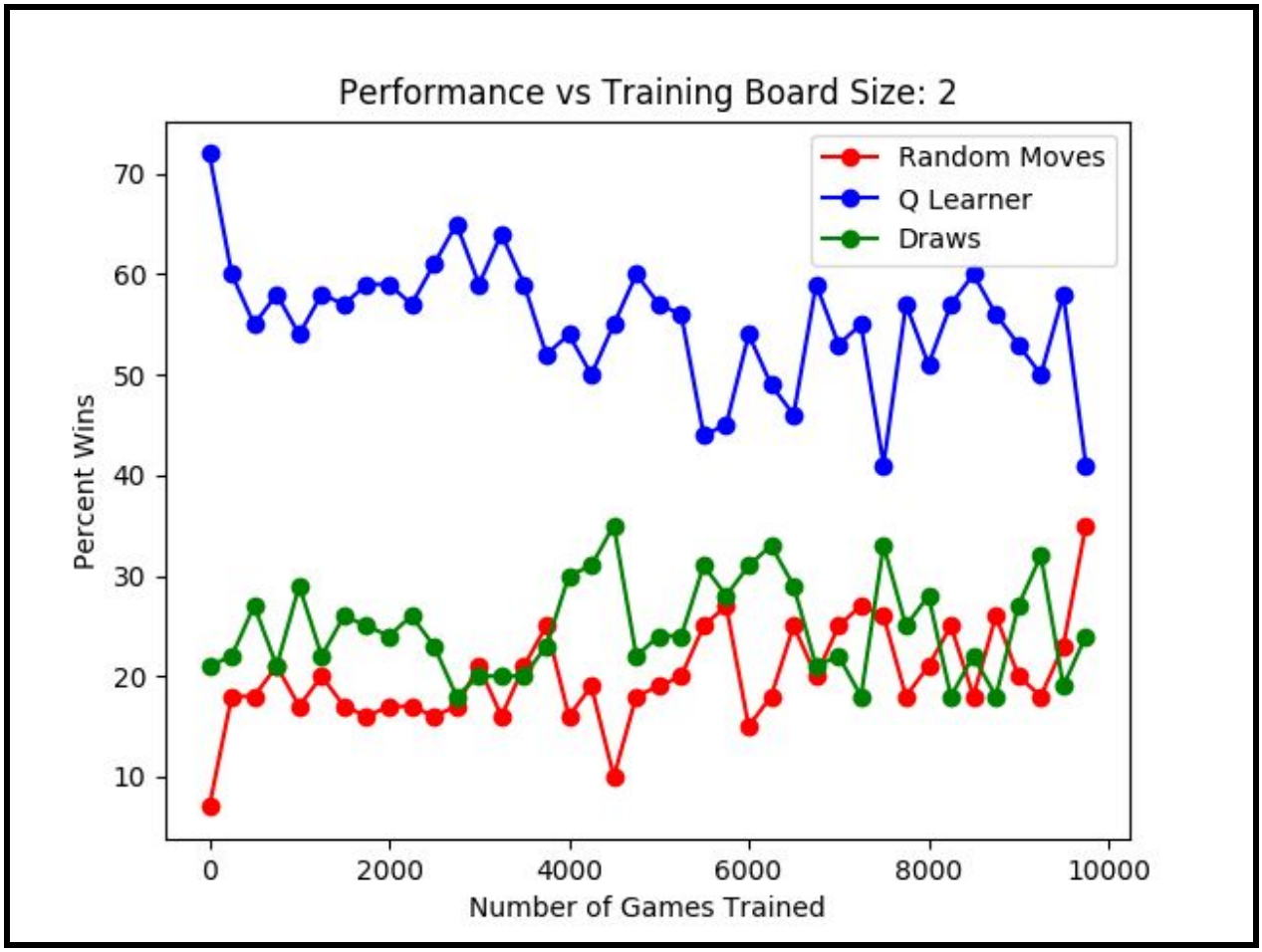

Figure 7. Q-Learning Agent vs. Random Moves Agent withFcn Approx 2x2 Updated Every 250

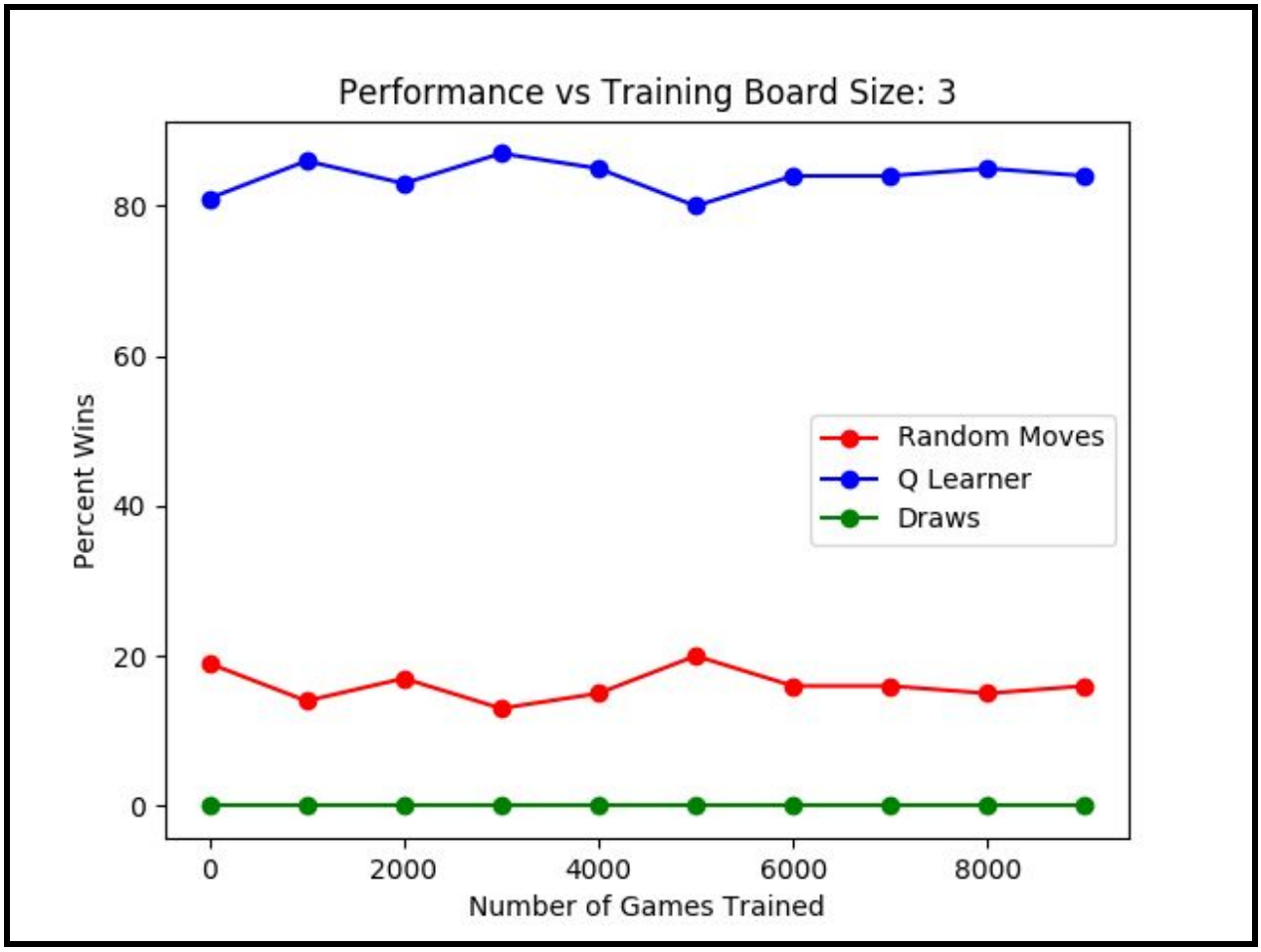

Figure 8. Results of a Q-Learning Agent vs. Random Moves Agent with a Fixed Fcn Approx

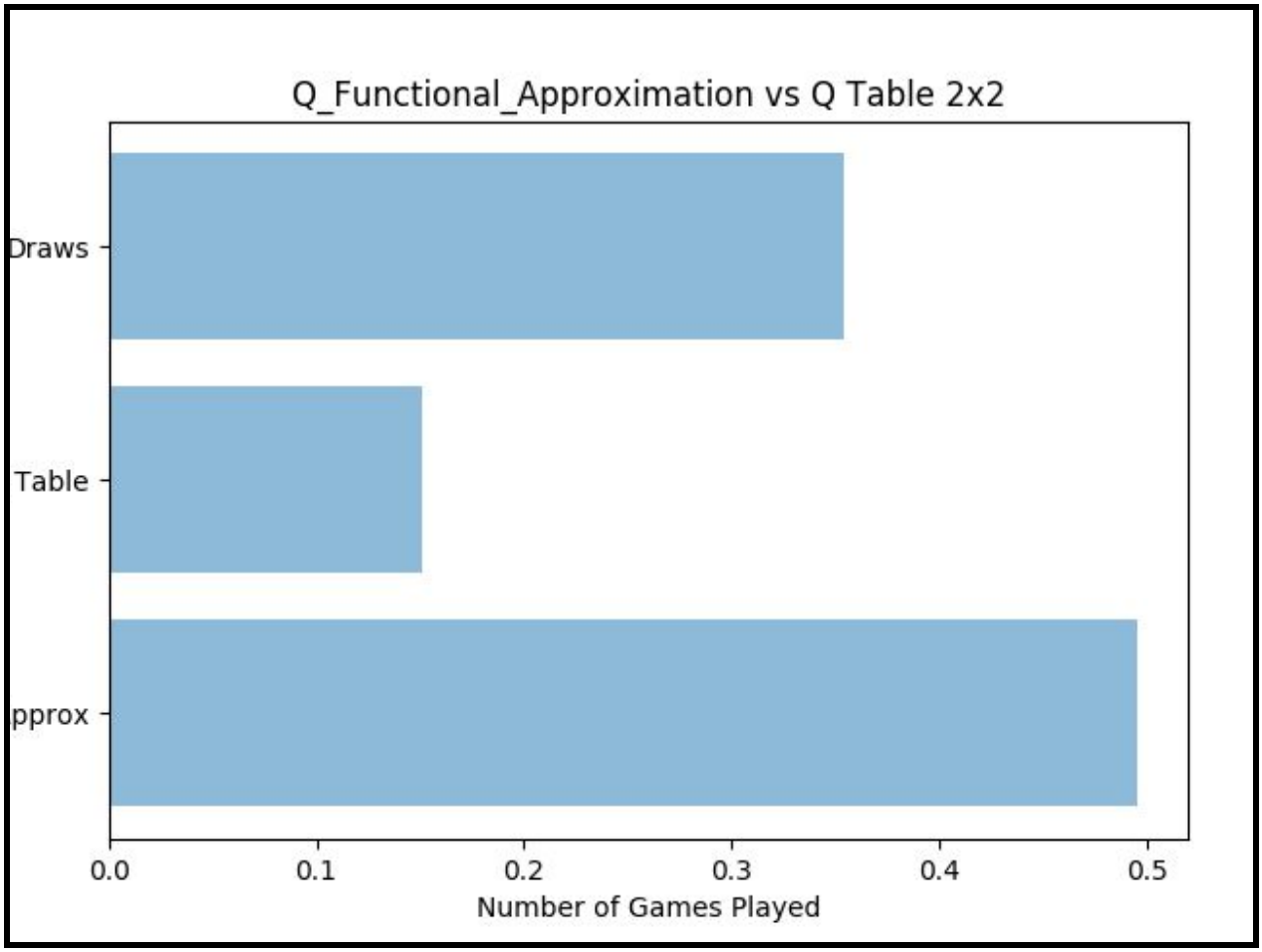

Figure 9. 2x2 Q-Learning Table vs. Function Approximation

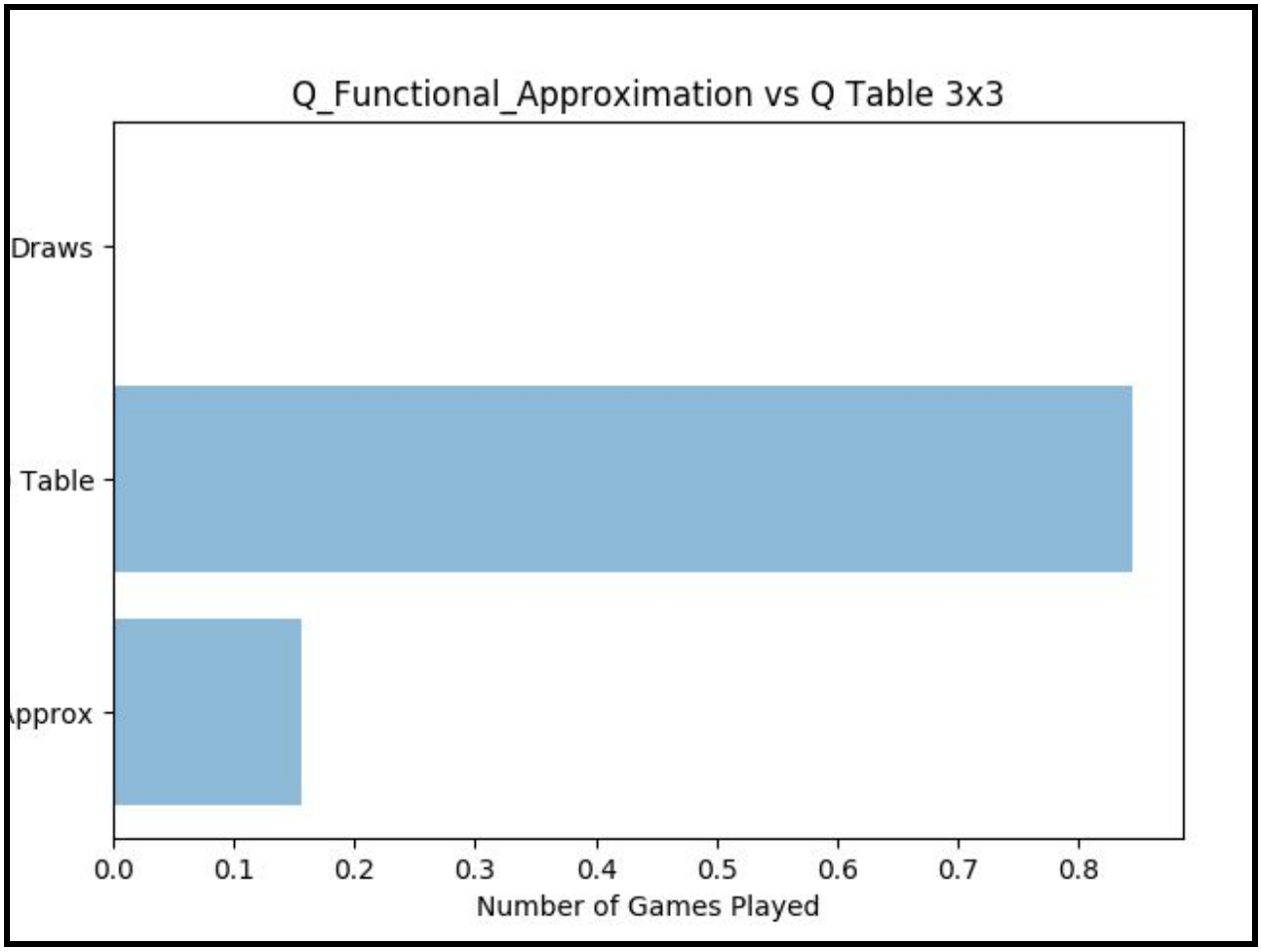

Figure 10. 3x3 Q-Learning Table vs. Function Approximation

Implementing a Q-Table

In order for a Q-Table to work, one must have all of the State-Action Pairs available to look up in the table and retrieve a Q value. Originally, this lead me to write scripts exploring different ways to reduce the representation of these states and action in order to preserve time and memory allocating all of the combinations. Considering that each Q-Value is initialized to 0 and eventually updated the number of combinations were quite large for the 2x2 and would grow exponentially as the board size increased. The State_Reduction_Script.py contains the two methods I'd come up with for representing states and saves them to another python file. Originally, I used this list of states as the keys in a python dictionary that corresponded to the Q values (initially set to 0). There were two revelations regarding this methodology.

The first revelation was that the get() method provided by python dictionaries allows a default value. So instead of storing every conceivable combination of states and actions I elected to start with an empty dictionary. When requesting Q values for state-action pairs the agent had never seen before, it would return the default value of 0. Then later during the update phase it would update and store those state action pairs in the dictionary for future reference.

This implementation allowed me to save time and memory for both the 2x2 and 3x3 implementation without sacrificing any functionality in the Q-Table implementation.

The second revelation related to how the actions were selected. Early on when all state action pairs return a Q value of 0 they should all essentially be equally likely to be chosen. However, the argmax function will return the index of the first max value if multiple max values are present. Originally, this only became a problem because the list of available actions were passed in the same order the same order every time. When building a game of Dots and boxes all possible lines or sides are created at the beginning of the game. They are then allocated to squares via an algorithm. The list of possible lines were therefore provided in such and order that would complete squares. Now when the Q-Learning Agent was choosing moves and most often selecting the first move in the list it was unfairly biased to select square completing moves. In order to remove this bias all available actions are shuffled before passing them to the Q-Agent.

There was some difficulty trying to use the learned information from the 2x2 to seed a 3x3 game. One proposed method suggested representing the 3x3 board with 4 overlapping 2x2 games. However, I felt that this may be problematic because a move that would be considered great in 1 2x2 game might be a terrible move to make in one or all of the remaining overlapping games. Due to this, I resolved not to do that. Instead I took the state action pairs present in the 2x2 Q table and laid it over the top left corner of the 3x3. I set the remaining walls of the 3x3 to be off and adjusted the actions to map accordingly. As mentioned previously, the seeded information was not sufficient to give the seeded 3x3 agent any advantage over the non seeded agent during the first 10,000 games of training.

State Action Representation

The state representation chosen used a boolean bit for each wall on each square. The first bit referred to the left wall of the box and the remaining bits represented the walls moving clockwise around. Now there are walls that are shared by squares, making the boolean in that case, somewhat redundant because it appears in two places. Nonetheless, it was provided for consistency's sake. This representation was 16 bits for a 2x2 and 36 btis for the 3x3. Accompanying each state was a integer corresponding to an action. The total number of actions per game was equivalent to the number of walls (Shared or not) present in the game. For the 2x2 game that was 12 moves while the 3x3 game there were 24. The states and actions were stored as a pair and then added to a python dictionary representing the Q-Table.

Implementing a Functional Approximation

The functional approximation for both the 2x2 and 3x3 games used the Q-Tables from the 10,000 game training sessions. All of the keys that were stored as state action pairs were converted into a 1-D input matrix for a Neural Network. The corresponding values were passed to the network as labels. The entire dictionary that was accumulated over the course of those games was used for training data for the Neural network. The network consisted of 3 Layers all using relu activation functions except for the output layer. The network was structured as follows:

- Dense Input Layer - Length Depending on the board size
- Dense 128 Neurons
- Dense 64 Neurons
- Dense 32 Neurons
- Dense 16 Neurons
- Dense 1 Neuron (Output)

Dots

The game was implemented with the following rules:

- Player 1 or Player 2 randomly selected to start
- Players completing a square will must go again and take another turn.
- Rewarded 1 Point for Completing a Square
- Rewarded 5 Points for Winning