Assignment 05

I found it difficult to decide on a reinforcement learning algorithm for the Tetris game. I ultimately decided to implement the TD-learning for Q-values algorithm because it seemed to fit best for the problem at hand. In order to use the TD-learning for Q-values algorithm, I had to create a state representation for the agent and a reward function.

That state representation I implemented holds a block and an integer array. The block refers to the current block that is being used. The integer array holds the amount of free space from the top level down in each column. In order to find this I created a method that loops through the current full board and looks for a free space. It checks to make sure that the free space is uncovered from the top. In order to accomplish this I created a Boolean array that corresponds to each column. When a free space is found, it checks the Boolean array to make sure that the column hasn't been blocked earlier. This gives a good representation of the shape/contour of the current blocks.

Each state has to have an action. The actions in this problem are to move the block between 0 and 5 spots, and to rotate it four different ways. This equals out to 24 different actions for each state representation. In order to hold the state action pairs in the best possible way, I decided to use a HashMap that has a key of a State and a value of another HashMap that holds a key of an Integer and a value of the Q value. The Integer corresponds to one of the 24 possible actions. This set-up allowed for constant time lookups, which was incredibly important for this problem. If your state representation, or how you held your state action pairs, took up too much data, your algorithm would run very slowly.

My reward function for this problem was very simple. If one of the rows in the Tetris game was filled, then that state action pair would get a reward of 100. If, however, a row was higher than 3 or a row dropped off the bottom, the state action pair would receive a reward of -50. The reason I chose these numbers were experimental, but they seemed to have worked well.

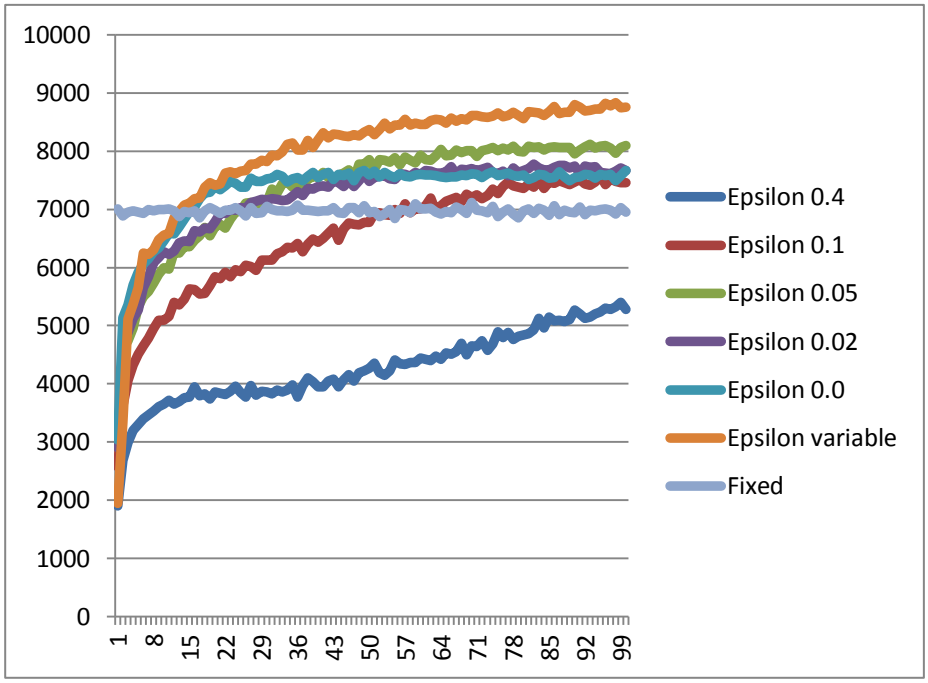
The TD-learning for Q-values algorithm used a few different variables. It used the value from the current state action pair Q(s, a); the proceeding state action pair Q(s',a'); an epsilon value, which dictated how often to use random values; an alpha value, or how much to care about the current value (starts high and goes lower as you learn); and a gamma value, which is a constant scaling factor of 0.9. The algorithm will execute a given number of runs per game, with each run holding a given number of iterations.

In order for the algorithm to pick an action, it would pick a random number between 0 – 1. If the number was smaller than the set epsilon value, it would choose a random action.

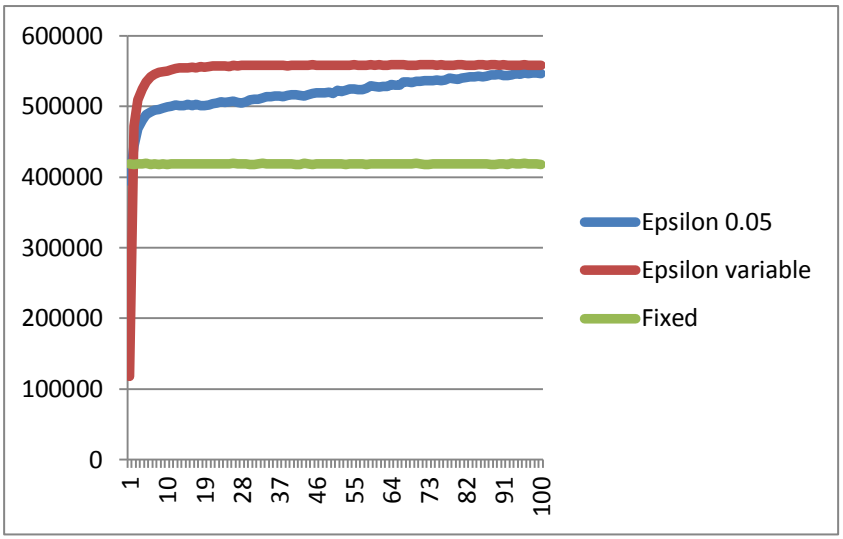
Otherwise, we would loop through all 24 possible actions and take the first highest action found.

For my first run I used a constant epsilon value of .4, which means that 40% of the time the algorithm would take a random value. I used an alpha value that was equal to (# of possible steps - # of steps so far) / # of possible steps. That way, alpha would start at 1 and slowly go down to 0. The algorithm would go for 100 runs with 10,000 iterations per run. I found that my algorithm worked exceptionally fast. The values it got, however, were not great. I got a Q value of 5,276 after 100 runs. This was much less than the fixed policy made by Dr. Allen, which ended at 6,954. I decided the problem resided with my epsilon value.

I then lowered my epsilon value to 0.1, followed by 0.05, followed by 0.02 and ultimately 0. The algorithm seemed to work best with an epsilon of 0.05. After this, I decided to use a variable epsilon. That is, I set my epsilon to (1 / (# of runs + 1)). This would obviously start at 1, and become insignificant as the number of runs become higher. The results of the epsilon changes are shown in the graph below:

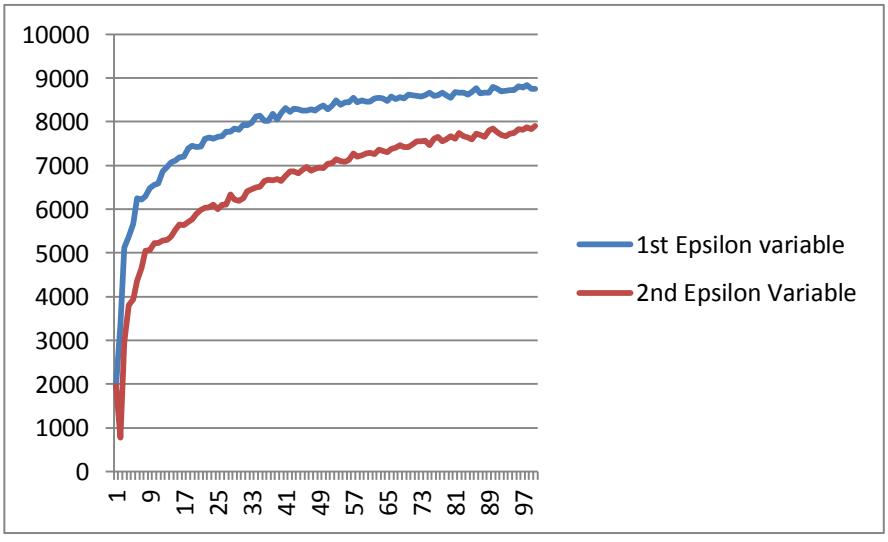


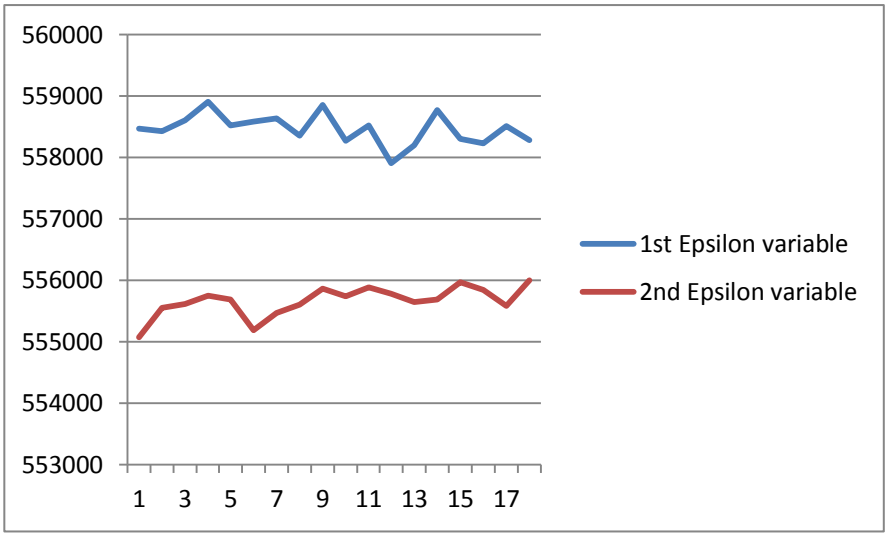
As you can see, any epsilon less than .1 beat the fixed policy. The best fixed epsilon was .5, but the variable epsilon beat the rest by far. I wanted to make sure that these results followed for larger data sets. In order to test this, I ran the algorithm with 600,000 iterations and 100 runs. I used epsilon values of 0.05 and variable. The results are as follows:



The results were fantastic, with the variable epsilon beating both the fixed epsilon and the fixed policy. As we can see, the variable epsilon learned rapidly for about 31 iterations and leveled off for the remaining 69. It finished with a Q value of 558,279. 11,896 better than the epsilon of 0.05, and 140,377 better than the fixed policy by Dr. Allen. The results of this state representation were phenomenol.

I then decided to try a more data heavy state representation. I kept the current block and Integer array, but added the next block. That way, there would be many more possible states, and the algorithm would know more each time. I then reran the same experiements. I compared the results from the 1st state representation and the 2nd state representation. The results were as shown:





As we can see the first state representation performed better. It also ran in about half the time as the second state representation. Although I expected the first state representation to run faster than the second, I didn't expect it to perform better. I assumed that more information directly correlated to better performance, but this didn't prove to be true.

Overall, we saw that a state representation consisting of just the current block and an integer array was able to learn and perform significantly better than a fixed policy, as well as a state representation that held more data. We also learned that a constantly decreasing epsilon gives better results than a fixed epsilon, although a fixed epsilon of 0.05 seemed to be an optimal fixed value.