Severian Adams
GITHUB REPO FOR THE PROJECT: https://github.com/Lufferly/csc496FinalThing

- The only real important file is the final.ipynb. The other files are from some of the homeworks you gave us that I put in the same folder so that I could reference them when building the environment and agents.

I have been implementing reinforcement learning for the capture go environment. So I have implemented the environment, which took a while, and implemented deep-q learning, though it is FAR from perfect.

The environment took a while to perfect, mostly making sure that we could properly train the agents, considering that I wanted to have multiple agents training against each other. The hardest part was detecting when the environment was terminated, because we need to calculate whether a piece had been captured, and then deal with making sure both sides know.

I think I came up with a pretty cool solution for this, involving recursion but not needing to traverse the entire board every single move, only the parts of the board that could possibly be captured, leading to faster episode times. We also just use one environment that two agents act on simultaneously. We continuously pass off whose turn it is when training, and the environment will detect if something went wrong.

Another cool thing I did with the environment is make it so that it won't break if an agent tries to make an invalid move. Instead it will punish the agent with a lower score than it would otherwise get. This means I don't even need to tell the agents how to play the game, they will naturally learn how to play just by learning in the environment.

After doing the environment, it was time to implement the agents. I decided to first use deep-q learning, since thats what google used for their go algorithm. I also used epsilon-greedy in when picking states. I found two tutorials that helped me learn how to implement deep-q learning in pytorch:

- https://docs.pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
- https://www.geeksforgeeks.org/python/plot-multiple-lines-in-matplotlib/
- All of the pytorch docs were very useful in helping me do this as well, as there are a lot of neat tools they have to make implementing reinforcement learning easier.

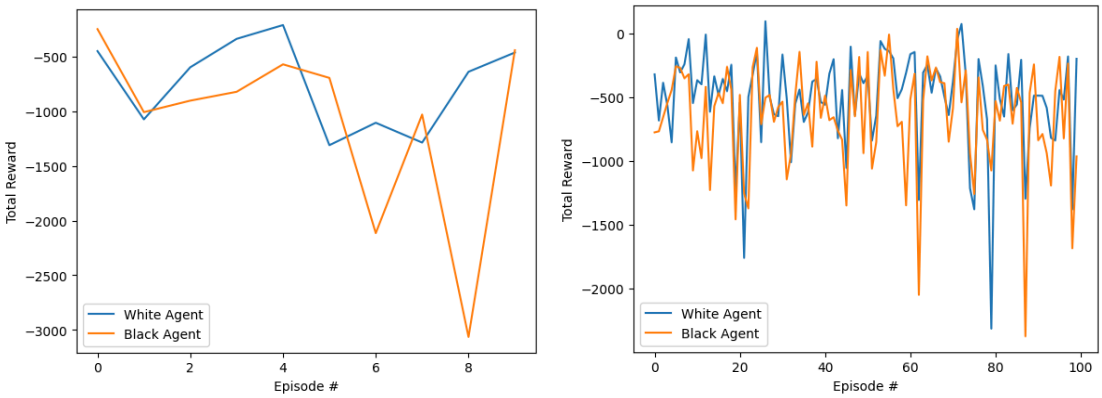
Implementing the deep-q learning was pretty hard, and took most of my time. It was made harder because I wanted to implement deep-q learning for two agents playing against each other at the same time, which the tutorials did not teach me how to do.

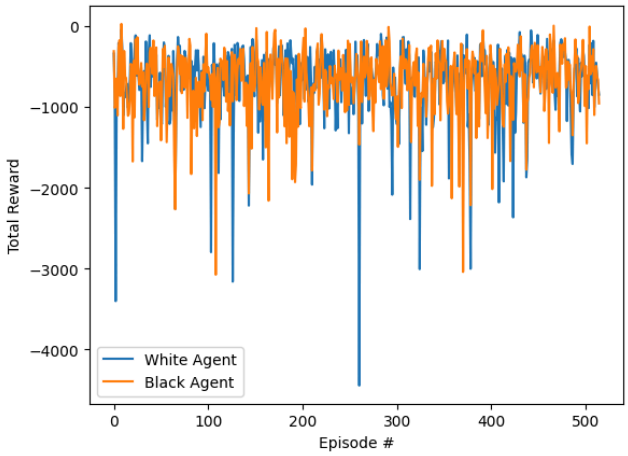
The deep-q learning network that I created takes each "tile" of the go-board as an input. Each tile element is 0 if it is empty, 1 if it has a white stone in it, and -1 if it has a black stone. The network then goes through two hidden layers before coming to the output layer. The output layer has a number of nodes equal to the amount of tile on the board; each tile corresponding to an action. An agent chooses an action based on whatever output node has the highest signal.

In the end it is still very flawed, and my agents are not learning as fast as I hoped they would. It still needs some work, especially on the optimization side of things. It took about 2 hours 30 minutes to run ~500 episodes, so 1000 episodes total since we have two agents.

- One of the things that I think made it take even longer was the fact that I decided to have two hidden layers in my deep-q network, instead of just one. I wanted to be able to capture more complex behaviors like was shown in class with the convolution layers, but I think in the future it will be worth it to test if just 1 hidden layer would be best.

Here is the data generated by some runs:





As you can see, I did not get the improvement that I was hoping for (though there is some slight improvement). I still have a long way to go for the final project. There were several main roadblocks that I think I am having:

- Training took so long, so it was hard to make changes and then see if those changes had any effect. A possible solution to this is to optimize my code, like reducing the hidden layers in my deep-q networks, making my environment code resolve faster, etc.
- My agents are not told how to play the game. This means that they need to learn how to play go at the same time that they are learning to play against each other. Perhaps if I ran it for even longer, I would see slow progress as they learn the game, and faster progress once they actually can start making consistent legal moves?

- I don’t have one agent playing against itself, I have two separate agents playing against each other. This might mean that my graphs might be somewhat in accurate in terms of how well they are learning; both agents are probably about the same skill level so even if they are getting better at the game the games still tend to last the same amount of time. Additionally one of them HAS to win and the other HAS to lose, meaning they are consistently getting high and low return cycles. Perhaps it would be a better solution to just have one agent play against itself? That way that agent would also get twice the amount of learning, since it would train twice as much.
- Of course some tuning of the hyperparameters is probably in order, and my final project will probably be focused on what good hyperparameters for this environment look like, and how the hyperparameters affect the learning.

I think for my final project, I would also like implementing some other reinforcement algorithms, since this one is not working so well. I think I bit off a little bit more than I could chew by tackling deep-q first, even though it seems the coolest. Perhaps I could implement a tabular method on a REALLY small board size, say 3x3, and see how that does. One cool thing about using multiple reinforcement learning algorithms is in the end I could have them play against each other since we are modeling a board game. This would be an interesting way of comparing reinforcement learning algorithms that isn’t just comparing their graphs or how much reward they are getting.