Zi Yan Zhang

For the basic decaying epsilon greedy agent, a Q-Learning agent was the basic agent prepared to be used for the Hex Game competition if the Neural-Network based agent did not function properly. The only set of hyper parameters this agent has are the values of epsilon, gamma, and alpha. These values are set to Ɛ=0.1, α=0.02, and γ=0.98, which are the values used for this class for the Q-learning agent. The Q-Learning agent saves the Q-value tables into a JSON file after each game play. When a new game is launched, the old Q-value files are loaded and are used as the Q-values of the current game and are updated from that point on. Since Q-Learning agents use epsilon-greedy methods that relies on Q-values to get the most optimal actions, by saving the Q-values after each successive play and re-using them as the starting point for a new game, the Q-values become more accurate when the agent plays more games and thus, is being trained. The agent handles the difference between two different reward systems by saving to and loading from two separate files based on whether the reward system is sparse or dense. The agent takes an initialization parameter that specifies whether the current game will have dense rewards. If a dense reward system is used, then it will save and load the Q-values from dense reward environments. This goes the same for sparse reward systems. So dense reward Q-values and sparse reward Q-values are stored in separate files and are used accordingly

When tuning the hyper-parameters for the Neural Network based agent, many of them had to do with learning and training, however, the main thing I needed to touch is the UCB1 exploration constant. This is because that would determine if the model would take an action that is exploratory or exploitive, and tuning the constant applied in the formula would weigh exploration depending on the magnitude of the constant. To tune it, I looked at the opponent’s architecture, and if it more resembled my own model’s architecture, or if I predicted that it would produce more random outputs (less human like), I would decrease the constant to make it exploit more because I knew that my model would likely win because it had been trained on similar actions. If it seemed like it was a more strongly fit model (specifically made to go in a straight line or seemed to have an architecture less like mine, I would increase the constant in order to attempt to have the agent take an exploratory action in an attempt to potentially block the actions of a more overfit model or actions that were less familiar to my agent’s training.

In the training loop, the agent periodically saves a checkpoint of the model weights so I can evaluate the performance after a certain number of training rounds. When I want to load the weights that I’m happy with, I simply pass the relative path of the .pt weight file. To handle the sparse or dense reward system, I had tuned my model for sparse and dense environments slightly differently, and in the instantiation of the model, I checked for if the environment is dense or sparse and load the weights accordingly.