COMP 560
HW 2

For this assignment, we were given the task to train an AI to be able to play a game of tic-tac-toe on a 3D plane. The AI is supposed to get better and better after each game by calculating utility values for each individual tile in the board. In order to accomplish this, we had to implement a form of reinforcement learning, so that the AI could learn from past mistakes/accomplishments. After searching on google for various reinforcement learning techniques, we came across the idea of "temporal difference learning."

This form of reinforcement learning uses previous states in order to be able to predict future states. For example, given a set of past states and their associated outcomes/rewards, we can use these to be able to make the best choice for a future action.

In our code, we used a HashMap to map a serialization of the board's state with the utility values of the board. For example, if we were playing on a 3x3 board and no moves have been made, our serialization would be "000000000" and it would map to a matrix consisting of utility values in each index.

Our temporal difference method takes in an integer "reward", a String representing the serialization of the previous state, and a String representing the serialization of the current state. This method then applies the temporal difference formula:

**X = X + a(v(G) – S))**

**X** = the 3D matrix that holds the utility values after successfully completing the formula

**a** = a scalar that represents the learning rate of the program. In our case, the learning rate is 0.5

**v** = a scalar that represents the "reward"

**G** = a 3D matrix representing the "current" state

**S** = a 3D matrix representing the "previous" state

For our "reward" values, we put two AIs against each other and played an N amount of games. We assigned 1 to an AI that won the game and -1 to the AI that lost the game when training. In the case of a tie, we would just assign 0 to both. When we have calculated the final 3D matrix, we would replace those values into the HashMap with the "current" state key. This would update our utility values and would allow us to use these to predict future moves.

Another important part of our code is our "explore" and "exploit" methods. These methods are called when an AI makes a new move. The AI can either "explore" the game board and pick a random index that hasn't been used yet or the AI can "exploit" the game board and choose the index that has the highest utility value. We learned about exploration and

exploitation in class with respect to Markov Decision Processes, and we used the same idea in our code. In order for the AI to pick one, we have a “exploration constant” of 0.9, meaning 90% of the time while training, we want to explore. This is because we want to be able to find the best utility values for each action while training, so that the AI could improve.

With each successful move, we check if the game has been won or not by having helper methods that check rows, columns, diagonals, ect. for similar values. These methods can be found in the Board class. After training is done, we set the AI’s exploration value to 0.0, so that it will now only expoit the board, meaning it will only make moves that will most likely result in a win.

In each of our classes, we have comments explaining the process and what each code block does, so take a lot at those!

For instructions on how to run the program, read the README file in the repository.