ENPM 808F Robot Learning Homework 4

Chethan Mysore Parameshwara

Program a Tic-Tac-Toe game. Assign a reward of +1 for a win (X wins), -1 for a loss (O wins), and 0 otherwise. Have it train itself to play through self-play and Q-learning. Use a table to store the Q-values. Does it achieve an optimal policy?

Here is the pseudo code which I referred to program the tic-tac-toe game. (https://www.cs.swarthmore.edu/~meeden/cs63/f11/lab6.php). The program is written in python by using the dictionary data structure for storing the Q values. Here state is 1 x 9 array representing the game board. And, actions are coordinates of the move taken.

```
gameLearning(startState, maxGames)
  state = startState
  games = 0
  while games < maxGames
    stateKey = makeKey(state, player)
    if stateKey not in table
       addKey(stateKey)
    action = chooseAction(stateKey, table)
    nextState = execute(action)
    nextKey = makeKey(nextState, opponent(player))
    reward = give_reward(nextState)
    if nextKey not in table
       addKey(nextKey)
    updateQvalues(stateKey, action, nextKey, reward)
    if game over
       reset game
       state = startState
       games += 1
    else
       state = nextState
       switchPlayers()
```

The game is designed in such a way that two players are trained together by sharing the Q-table. Both the players gradually improve their strategy by competing with each other. This approach has the advantage over playing against a random player because the opponent is also competent to play strategical moves. The approach also helps in reducing the number of iterations required to train the Q-values. The pseudo code explains how this is implemented.

```
updateQValues(stateKey, action, nextKey, reward)
  if game over
     expected = reward
  else
     if player is X
        expected = reward + (discount * lowestQvalue(nextKey))
     else
        expected = reward + (discount * highestQvalue(nextKey))
  change = learningRate * (expected - table[stateKey][action])
  table[stateKey][action] += change
```

I ran the Q-learning for 200,000 interations, it gave very good results. It's almost impossible to beat the Q-player. It can be seen in https://www.youtube.com/watch?v=oeHcHDRLfoA

Replace the Q-value table with a functional approximation, such as a neural network. Compare the results of Q-learning using the table representation with those achieved using your functional approximation.

For this problem, I used 3 layer Deep Q Network to replace the Q table. There are 10 inputs and 9 outputs. The input contains a state and player information. The output is nothing but the q-values for all the actions. The reward function for this problem is designed differently than conventional Q-learning. Because, the output gives q-values to all the actions irrespective of whether a particular action is valid or not. Hence, I give penalty to those actions which are not valid. Here, I am training only one player and other player is a random player. Next move is chosen by taking action corresponding to maximum q value. Further, weights are stored in a file and while testing, weights are loaded to the model. Even after running the DQN for more than 1 million times, I didn't get the optimal results. The player is making suboptimal moves. After debugging the code extensively, I am confident that my code is correct. I feel that I may need to train both the player together rather than playing against a random player.

In conclusion, Q-learning is better than DQN from performance, complexity and computational effort point of view. I got optimal results for only Q-learning but not for the DQN.