LAHORE UNIVERSITY OF MANAGEMENT SCIENCES
SBASSE: ELECTIRCAL ENGINEERING

Project Group: 2 December 25, 2023

Dynamic Programming & Reinforcement Learning Final Project Report

by

Syed Muqeem Mahmood Hamza Saleem

Abstract: The purpose of this project was to make the students familiar with a handful of dynamic programming and reinforcement learning algorithms covered comprehensively in the course by making them solve the Tic Tac Toe problem in three different scenarios: Tac Tac Toe in 2D (3x3 and 4x4) and in 3D (4x4x4).
Throughout the duration of the project, we encountered many challenges such as managing the huge state space of 2D 4x4 case and approximating the Q-values using a Deep Neural Network for the 3D 4x4x4 case.
Satisfying results were achieved for the 2D cases through self-training.

1 Phase I: 2D Tic Tac Toe (3x3 Grid)

To get ourselves accustomed to the game environment and its rules, we had to start by solving the basic 2D 3x3 Tic Tac Toe problem. We used Value Iteration because the state space is manageable and the model is known.

1.1 Algorithm

We adopted the following algorithm for value iteration:

Algorithm 1: Value Iteration

```
Result: Find $V^{\pi^*}(s)$ and $\pi^*(s), \forall s$
$V(s) \leftarrow 0, \forall s \in S$;
$\Delta \leftarrow \infty$;
while $\Delta \geq \Delta_o$ do
    $\Delta \leftarrow 0$;
    for each $s \in S$ do
        temp $\leftarrow V(s)$;
        $V(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$;
        $\Delta \leftarrow \max_a(\Delta, |$ temp $- V(s)|)$;
$\pi^*(s) \leftarrow \text{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')], \forall s \in S$;
return $\pi^*(s), \forall s \in S$;
```

The key idea is to compute the values of all observable states on the board and select a policy based the max/min values of states computed in the while loop. Since the game is played in turns, we know beforehand the all the next possible states and choose between min and max values for each alternating turn.
The total observable state space for this problem was computed using `generate_state_space(board_size = 9)` and came out to be `5478` which is manageable. A total of `100` epochs were run with γ set to `0.99`.

1.2 Results

A total of `1000` games were played between `player_1` and `player_2` resulting in `1000` draws, meaning each player was able to make optimal moves at each given state.

The policy table for each observable state in the game was saved in the local directory in `.json` format.

2 Phase II: 2D Tic Tac Toe (4x4 Grid)

The biggest hurdle faced during this phase was managing the huge state space of 4x4 board (10^6 states). This was the main motivation for using online Q learning where we interact with the environment to discover new states and compute Q values for each state action pair on the fly.

2.1 Algorithm

We have used self training approach where `player 'X'` is minimizing and `player 'O'` is maximising their rewards over their turns respectively during learning. The algorithm proposed for this approach is given below (note: S' is the discovered state space during environment exploration):

Algorithm 2: Online Q Learning

```
Result: Find $Q^{\pi^*}(s,a)$ where $s \in S'$
1  for $i \leftarrow 1$ to $N$ do
2      $s \leftarrow$ initialize environment;
3      while not terminated do
4          $a \leftarrow$ epsilon greedy;
5          $s'$, $r$, terminated $\leftarrow$ env.step($a$);
6          $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a') - Q(s,a)]$;
7          $s \leftarrow s'$;
8          if terminated then
9              break
10 return $\pi^*(s), \forall s \in S'$;
```

The above algorithm is repeated for both players starting from `line 4`. Also note that s' is the next state after opponent's turn. The player must wait for its opponent's turn when deciding s'.

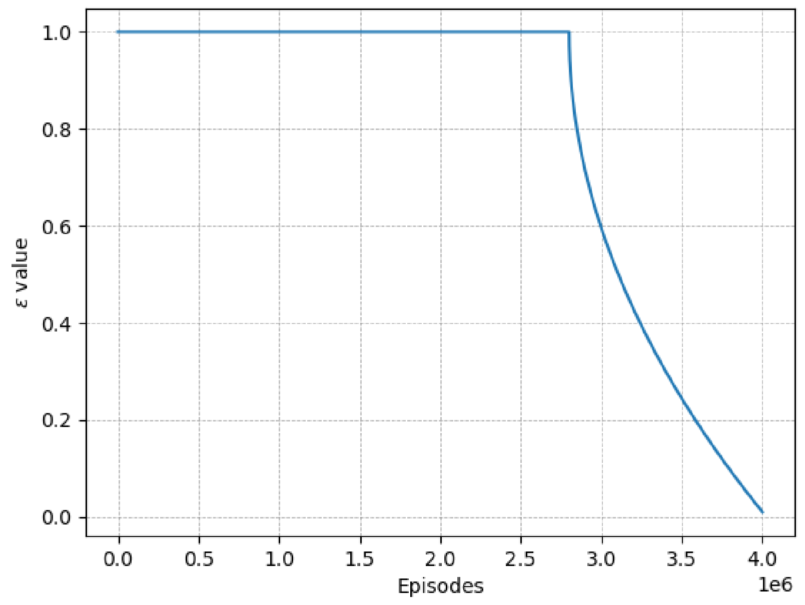

Figure 1: ϵ across episodes

A total of `4,000,000` episodes were run to compute the Q table for this problem. To reduce the computational complexity of the problem, we used hash tables (`python dictionary`) to store Q values. We initialized the learning rate 'α' to `0.7` and decayed it exponentially across the episodes to allow convergence of Q values. The tables were stored in the local directory in `.json` format.
Our exploration strategy was to take random actions for the first `3 million` episodes and then for the remaining `1 million` episodes, use epsilon greedy to select an optimal action with probability $1 - \epsilon$ and take a random action with probability ϵ. This allowed us to explore a reasonable number of states from the state space in the first `3 million` episodes and then start exploiting.

2.2 Results

The state space discovered during training was `7.962 million` A total of `2,000` games were played with alternating turns for each player resulting in `1014 draws`, `402 wins for player 'X'`, and `584 wins for player 'O'`.

Random actions were selected for undiscovered states during training. The results are satisfactory since the training is done in such a way as to maximise the number of wins and draws for each player. No negative reward is set for losing for either of the players which could be the possible reason for such results when compared to **Phase I** implementation.

3 Phase III: 3D Tic Tac Toe (4x4x4 Grid)

This was by far the most challenging setup. Not only did we had to deal with a state space of 10^{20} but also had to find a function which approximates the Q values in this huge state space. We initially implemented Deep Q learning where we used a Deep Neural Network (DNN) as a quality function approximator. The structure of DNN is provided here.

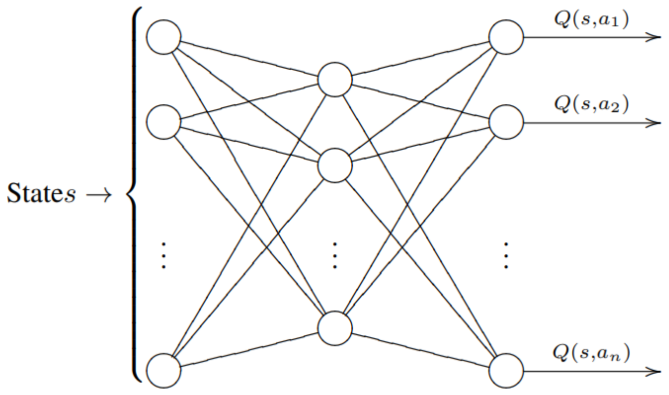

Figure 2: Deep Q learning

For the setup we have `64 input nodes`, `304 nodes in hidden layer 1`, `32 nodes in hidden layer 2`, and `64 notes in output layer`. The input layer corresponds to the board position, while the output layer corresponds to the possible Q values for all actions corresponding to the input state. Note that this architecture is inspired from the paper ”Reinforcement Learning in the Game of Othello: Learning Against a Fixed Opponent and Learning from Self-Play (Michiel van der Ree)”

3.1 Algorithm

The algorithm used in this failed setup is as follows:

Algorithm 3: Deep Q Learning

```
1 Observer the current state $s_t$;
2 For all possible actions $a'_t$ in $s_t$ use NN to compute $\hat{Q}(s_t, a'_t)$;
3 Select an action $a_t$ using epsilon greedy;
4 $\hat{Q}^{new}(s_{t-1}, a_{t-1}) \leftarrow r_{t-1} + \gamma \max_a \hat{Q}(s_t, a)$;
5 Use NN to compute $\hat{Q}(s_{t-1}, a_{t-1})$;
6 Adjust NN by back propagating the error $\hat{Q}^{new}(s_{t-1}, a_{t-1}) - \hat{Q}(s_{t-1}, a_{t-1})$;
7 $s_{t-1} \leftarrow s_t$, $a_{t-1} \leftarrow a_t$;
8 Execute action $a_t$
```

This steps are repeated for both `player 'X'` and `'O'` until one of them terminates the game. This procedure is repeated for `5 million episodes`.
This algorithm was taken from the same paper mentioned above. The main problem we encountered here was the assignment of the previous reward r_{t-1} for the estimation of $\hat{Q}^{new}(s_{t-1}, a_{t-1})$. The update is only possible while the games hasn't terminated and by structure of the algorithm, it is not possible to enter the `inner for loop` when the game has terminated.
Therefore, we had to let go of this methodology and stick to making `policy functions for player 'X' and player 'O'` separately exploiting the structure of the game itself.
The new proposed algorithm for this problem is to observe the board and check if the opponent or the player itself are close to predefined winning condition. If this is true, then the player should play an action to avoid the opponent winning or make itself win. Otherwise it should always play randomly in the hope that it encounters `3 'O's or 'X's` in a row upon which it should place optimally in order to maximise (or minimize) its reward. The policy function algorithm for `player 'X'` is provided here for convenience.

3.2 Results

The proposed policy function gives `100 percent` win rate against randomly policy, meaning it is able to win every time against a player always choosing random

Algorithm 4: Policy Function

```
1 Observe the current state $s_t$;
2 if 3 'O's or 'X's in a row then
3     Place 'X' next to the row of 'O's or 'X's
4 else
5     Play a random action from the available action space
```

actions. A total of `100 games` were played between this player and with opponent using random policy and our player was able to win all `100 games`.

4 Conclusion

Overall, this project provided us with a setup to implement most of the reinforcement learning algorithms covered in the course. Being able to learn the optimal actions from the environment after million of trials was cumbersome but the end results we achieved were satisfying.