Project Report
Dynamic Programming and Reinforcement Learning

By:
Ahmed Mozammil Iqbal (Roll no. 24100085)
Javeria Siddique (Roll no. 24100246)

25th December 2023

Table of Contents

1.0 Project Phase 1

For Phase 1, we were tasked with making a 3 by 3 Tic-Tac-Toe agent using Value Iteration. Our approach was done in the following stages:

1. **Setup and Initialisation:**

   We setup the states as the allowed Tic-Tac-Toe board configurations, which we did as follows:

   a. Using a recursive function we generated all possible board configurations.
   b. We removed all impossible states from the list of configurations which are:

      i. All states where both X and O win (which is impossible)
      ii. All states where number of X is not equal to number of O's or where O is not 1 more than X (which is also impossible)

      We also included the initial board as part of the configuration.

   We initialized the value table and policy table arrays as the size of all states. As there are a total of 9 possible places on a 3 by 3 grid where a player can make a turn, we created the Q table with rows being the number of states and columns being equal to 9.

   In the Q table we set NaN on the places in every state where the next player can not make a move (i.e: it is already occupied).

2. **Value Iteration:**

   The general setup was using a Minimax approach.

   For every episode we traverse over all states:

   For every state we find all next states and calculate the bellman values from the bellman equation according to each of the next values. We update the bellman values in the Q table.

   In the bellman equation, if it is X's turn, we set the reward of +1 if X wins in one of the next states and for O's turn we set the reward of -1 if O wins, else for all other next states we give the reward of 0.

From all of the generated bellman values in the Q table for the given state, we take a max of the values if it is X's turn and assign it to the value table. We take a min of the values if it is O's turn and assign it to the value table.

Hyper Parameter gamma: We set the learning rate gamma as equal to 0.99 as we seek to give more importance to later rewards. In our setup, the non zero rewards are only generated near to a terminal state, therefore we fealt it was appropriate to use gamma close to 1.

3. Calculating Optimal Policy:

In the Q table, we took argmax on the states corresponding to X's turn, and argmin corresponding to states corresponding to O's turn. The optimal policy table can then be used for playing the game.

We also faced the following challenges initially:

1. Non-convergence of Policy:

We realized this was because of an error in our bellman equation. Initially in our bellman equation we were multiplying the discount factor with the Value of the current state as opposed to the value of the next states.

As a result this leads to only the value of the current state being updated for all episodes and hence there was no propagation of the Q values in the Q table.

When we fixed the error in our bellman equation, our policy started to converge and the Q values began to propagate in the Q table.

2. Very long training time:

We identified all bottlenecks in our code and optimized them. Generally this was done as follows:

- We converted all list based functions to numpy based functions due to numpy's speed of execution.
- Before we were updating all states, but later we only updated those states which were not terminal, which reduced the training time as well.
- Initially we used linear search for any searches needed during value iteration such as: finding the index in the value table to be updated.

To optimize this, we created dictionaries of state to index in the value_table, and state to which player's turn it is.

By creating dictionaries we changed all searches from O(n) to O(1). which drastically increased the execution time of our code.

2.0 Project Phase 2

For Phase 2, we were tasked with making a 4 by 4 Tic-Tac-Toe agent using Q learning. Our approach was done in the following stages:

1. Setup and Initialisation:

Our approach for generating all states was similar to how we generated all states for Project Phase 1 except scaled for 4 by 4.

However as there are alot more states, which means a longer generation time for all states with the typical, we instead used np.mesh to quickly generate all 43 million states (including both legal and illegal states).

We then filtered all the states and removed the illegal states the same way as in phase 1 except scaled for a 4 by 4 grid until we were left with ~ 10 million states.

For faster computation during Q learning we set all the data structures, Q table, value table, reward table, turns table as dictionaries as it allows for faster indexing and searching.

The Q table has states as keys. And each state has actions as keys.

2. Q learning:

Q Learning uses a greedy epsilon approach with an initial epsilon of 0.9 so for the most part, it generates random actions for better exploration of the state space and environment. We implemented epsilon decay with the reduction rate of (total episodes - current episode number)/(total episodes). This decay rate reduces the epsilon by a small amount so that the exploration policy stays for a longer time in the start. Minimum epsilon value of 0.1 ensures that the epsilon value doesn't just vanish towards the end iterations, but co-exists with the dominant exploitation one (based on our previous learning pattern.)

Another hyper parameter of this approach is the learning rate. We initialize the learning rate with a very high value of 0.9 as well because in the start, where q table is sparse and majority of our rewards is going to be 0, we would want our terminal states to propagate the non-zero values with a faster rate. Higher learning rate will help with the initial fast propagation, but once we have explored the environment enough, we go towards fine tuning and reducing the learning rate so the values propagated in the Q table takes the precise values to give accurate optimal policy. We restrict the learning rate to be a minimum of 0.01 so we don't get stuck with the same Q table for infinite convergence time. Moreover, we update the learning rate (alpha) once every 1000 iterations for it to give some time for processing and propagating values.

3. Computing Optimal Policy:

Once we have obtained the Q table, we work through the min and max q values for each state of each player to decide on their optimal actions. X will take the max q value while O takes the minimum one.

4. Limitations of this Approach:

Since Q learning takes random steps and a few based on its previous learning to learn the states effectiveness, its convergence towards the optimal policy is really slow. Furthermore. We had over 10 million states and generating correct q values for all possible actions over them was going to take an infinite amount of overall episodes during the training process. Therefore, due to the limited time, space, and computational resources, we train it over a few million episodes which can give an estimate but not an optimal policy.

3.0 Project Phase 3:

For Phase 3, we were tasked with making a 4 by 4 by 4 Tic-Tac-Toe agent using any approach of our choice.

We followed 2 techniques for Phase 3:

1. Deep Q Networks

DQN model uses a deep neural network to estimate q values based on the Q learning formula incorporated in the loss function i.e., r + (gamma) * max (over all actions) (s', a') - Q(s, a) which helps the model to go towards optimal policy during backpropagation.

The 3D 4-by-4-by-4 environment has 64 tile values per state. Therefore, the input layer of our deep Q network consists of 64 cells. The output of the model calculates the Q values for all actions over the state. Now, since the output layer size of a neural network is fixed and the maximum possible actions over a state is 64 i.e., with an initial empty board state, the output layer of the model would also consist of 64 neuron cells. RELU activation between the linear dense layers addresses the vanishing gradient problem and introduces non-linearity in the model to capture for more complex state to actions dependent q values. Moreover, Adam optimizer adjusts the model parameters during the training process.

Here are some pointers related to our DQN model approach:

- Epsilon-greedy approach is used in this process too to balance out the exploration and exploitation in environment learning.
- Model is trained over a certain number of episodes i.e., 20000 in our case due to resource limitations.
- After the model training is complete, we invoke the model to generate the q values for a state and shortlist the useful values by refining over the possible (non-occupied) action locations.
- argmin-argmax strategy for alternate players generates their optimal actions over any possible state.
- Training over larger dataset (more iterations of state space) would have resulted in a better and accurate model

2. Rule based system

For the rule based system we used the following rules:

Rule 1: Check if we are winning in the next state, if so take the action that will make us win in the next state.

Rule 2: Check if we are losing in the next state, if so take the action that will block the opponent from winning in the next state.

Rule 3: Check if any of the action spaces allow for us to make a fork (opportunity to win in 2 ways or “double trap” the opponent. If so, take the action that will create the fork.

Rule 4: Check if any of the action spaces allow for the opponent to make a fork. If so, take the action that will block the opponent from creating the fork.

Rule 5: Create a situation where we have 3 of our action in a line where the 4th action is an empty space. This will help us win in the next to next turn.

Rule 6: Block the opponent from creating a situation where 3 of their action are in a line where the 4th action is an empty space.

Rule 7: If the opponent is occupying a corner, then occupy the opposite corner from the opponent.

Rule 8: Choose any of the 8 corners if they are unoccupied

Rule 9: Choose any other location once all the 7 rules are checked.

To check for if the agent is winning/losing we repurposed the given check_win() method as part of the WinConditions class in the provided Jupiter notebook for 3D Tic Tac Toe.

The other rules were adapted using the check_win() function as a base to create check_three() which sees if the agent has an opportunity to make 3 of it's moves/ block 3 of the opponent's moves. And check_fork() was also created using check_three() as a base.

The challenges we faced were as follows:

1. For Deep Q learning we were finding it difficult to map the states to the input neurons in the Neural Network for this we did…

2. For the rule based system it was difficult to decide how to make a fork in the 3D grid so that the agent can create a fork and also block the opponent from making a fork

4.0 References

We found the following links helpful:

[1]https://blogs.cornell.edu/info2040/2022/09/13/56590/#:~:text=The%20Minimax%20Tic%2DTac%2DToe,100%25%20chance%20of%20a%20draw

[2]https://www.youtube.com/watch?v=nOBm4aYEYR4

[3]https://www3.ntu.edu.sg/home/ehchua/programming/java/javagame_tictactoe_ai.html

To understand how the rule based system for standard 3 by 3 Tic Tac Toe works.