Reinforcement Learning - Tic Tac Toe

Vishal Saini
vishalde@buffalo.edu

Rohith Kumar Poshala
rposhala@buffalo.edu

Sreekar Guggilam
sreekarg@buffalo.edu

Abstract: In this project we are training an agent to play the game Tic-Tac-Toe. We used two different reinforcement learning algorithms SARSA and Deep Q learning to train the agent and see how well the agent plays when trained with these algorithms. Front end using HTML (for UI skeleton), CSS (for Styling) and JavaScript (handling conditions in frontend) is also implemented.

I. Introduction:

Machine learning is a field of study where the computers are automated to learn from the data. Reinforcement learning is one of three machine learning paradigms along with supervised learning and unsupervised learning. It's an exciting area of machine learning, where in a given environment it learns by balancing between exploring different areas (exploration) or based on current knowledge chooses the best move (exploitation) . It is basically the learning of an efficient strategy by interacting with the environment. A reward is assigned for a given behavior and the agents learn to reproduce that behavior in order to maximize the long term award. It is an iterative trial and error process since it chooses to explore and exploit as well. For this game, based on the structure of the game, the way we could track the game states and the way rewards are being assigned, we have decided to implement SARSA (State Action Reward State Action) Algorithm and Q learning Algorithm. Both are algorithms for learning a Markov decision process policy, used in reinforcement learning. Q learning is an Off-Policy Algorithm for Temporal Difference Learning*.

SARSA is a modification of the Q-Learning algorithm. It is an On-Policy algorithm for Temporal Difference learning. Major difference between SARSA and Q-Learning is that we do not necessarily use the

Q-learning : $$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

SARSA : $$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

*Temporal Difference Learning refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function. These methods sample from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods.

maximum reward of the next state to update the Q-values of the current state. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. The difference is visible in the update statements for each algorithm :

II. Methodology

Deep Q Learning -

Our Deep Q Learning Pseudocode:

Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
  Sates $\mathcal{X} = \{1, \ldots, n_x\}$
  Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
  Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
  Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
  Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
  Discounting factor $\gamma \in [0, 1]$
  procedure QLearning($\mathcal{X}$, A, R, T, α, γ)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
    while Q is not converged **do**
      Start in state $s \in \mathcal{X}$
      while s is not terminal **do**
        Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow \arg\max_a Q(x, a)$)
        $a \leftarrow \pi(s)$
        $r \leftarrow R(s, a)$ ▷ Receive the reward
        $s' \leftarrow T(s, a)$ ▷ Receive the new state
        $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
        $s \leftarrow s'$
    return Q

The deep learning network was implemented in order to understand the operations of a neural network with multiple layers and use the architecture of a neural network to implement Q-Learning algorithm. The DQN contains 4 layers- One layer for both input and output and two hidden layers. The first hidden layer has an input of 256 nodes and the second hidden layer has an input of 64 nodes which then map the output to a matrix of size 9 which is the board dimensions. Out of these 9 values, the most favored probability is used to select the action for the agent. There are multiple approaches to implement the network such as using a sequential model or using a tabular approach. Similarly, the initial parameters such as learning rate, discount rate, loss function are defined as variables and with every action taken by the agent, the next state is calculated and then the reward is computed. After the game reaches the terminal state, using the reward, the network is updated using back propagation of the network.

To store the experiences i.e, the current state, action, reward,next state and object of class MemoryBuffer is used. The class is nothing but a wrapper of the deque collection. Then after every terminal state, the network is updated using the tensorflow methods. The loss function used is the mean-squared error to calculate the differences between the current action reward and the future reward.

The Environment class is the class that creates the state for the game. By default, the empty space on the board has a value of -1, 1 for cross and 0 for naught. This class is responsible for telling the agents if they won or the game was a draw. To test the terminal conditions, a list of winning combos is stored as a variable providing every possible combination of winning a 3x3 game. Also, the default board size is 3x3 but can be increased and the code trains the agents on the respective board size.

The DeepAgent class is responsible for creating a tensorflow agent with its own deep neural network and variables such as update rate etc. The class is also responsible for agent related methods such as updating the model, reducing the exploration rate or taking a step based on exploration factors.

The training part consists of training two agents that play each other for 200k iterations. The weights and the states of these

agents are stored in the tensorflow session and ultimately dumped into a file.

This file is again used to load the values and test the agents against each other or a human player.

The training stats show that both the players play well against each other with a fair number of draws.

SARSA -

Our SARSA Pseudocode:

SARSA(λ): Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
- Sates $\mathcal{X} = \{1, \ldots, n_x\}$
- Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
- Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
- Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
- Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
- Discounting factor $\gamma \in [0, 1]$
- $\lambda \in [0, 1]$: Trade-off between TD and MC
- **procedure** QLEARNING($\mathcal{X}$, A, R, T, α, γ, λ)
  - Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
  - Initialize $e : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ with 0. ▷ eligibility trace
  - **while** Q is not converged **do**
    - Select $(s, a) \in \mathcal{X} \times \mathcal{A}$ arbitrarily
    - **while** s is not terminal **do**
      - $r \leftarrow R(s, a)$
      - $s' \leftarrow T(s, a)$ ▷ Receive the new state
      - Calculate π based on Q (e.g. epsilon-greedy)
      - $a' \leftarrow \pi(s')$
      - $e(s, a) \leftarrow e(s, a) + 1$
      - $\delta \leftarrow r + \gamma \cdot Q(s', a') - Q(s, a)$
      - **for** $(\tilde{s}, \tilde{a}) \in \mathcal{X} \times \mathcal{A}$ **do**
        - $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \alpha \cdot \delta \cdot e(\tilde{s}, \tilde{a})$
        - $e(\tilde{s}, \tilde{a}) \leftarrow \gamma \cdot \lambda \cdot e(\tilde{s}, \tilde{a})$
      - $s \leftarrow s'$
      - $a \leftarrow a'$
  - **return** Q

This code has two parts, one for the backend code which is in python programming language and the other is for the frontend developed using HTML (for UI skeleton), CSS (for Styling) and JavaScript (handling conditions in frontend). For this application to integrate both the frontend and backend we have used Flask framework.

Backend part -

Firstly, necessary libraries have been imported. Here we need an Agent to play in a Tic Tac Toe gaming environment. A SARSA Agent class has been defined which helps us define a standard player/agent so that the attributes and functions are consistent among all players. It also provides good reusability of the code which makes the program more efficient and a good structure to the code which increases the readability of the program. In the SarsaAgent class, necessary attributes like discounting gamma of the next state values, exploration rate of the agent, learning rate to learn optimal Q value to the current state and other attributes are defined. Among which one stores all the states of the particular game and the other stores a dictionary of combinations and its corresponding values.

These attributes are used in the methods/functions of this class which performs operations like computing and storing the Q-values for the current state with the help of SARSA algorithm where after every game, a reward is collected and a Q-value is learned based on the reward. Later Q-values of the precedent states are learned using Q-value of subsequent state as reward. Few functions in the class are defined for picking up the next action which has the maximum Q-value among all combinations and to store the current state as a list which is hashed and used as a key representing the current state along with its Q-value. This class also have function which can save the obtained policies (which are stored as a dictionary with hashed state as key and its Q-value as value) from the agent training by appending them to existing policies file for that player and function which can load the saved policies and use them for the Agent performance helping its decision of taking next step.

Game Environment(board) is defined as a basic list of values from 0 to 8. Functions are defined, to determine the winner by

evaluating the current state of the board and to assign respective reward based on results like win, lose and tie for both the players individually which are later used to determine and store the Q-values as mentioned above.

Function named agent_training is defined which takes SarsaAgent objects and the number of iterations as arguments. In this function, for each iteration we initiate the board state and each player picks its next turn based on the Q-value, stores the current board state to respective attributes and the loop exits when either of the player wins or when it's a tie. At the end of the loop, we get the reward for each player. By the end of defined iterations, both of the player instances get updated with all combinations of the board states and its corresponding Q-values.

Training the Agents:

Agents are initialized for SarsaAgent class and are trained with the help of agent_training function. The perfection in the agent training is directly proportional to the number of iterations given to the above training function. For example, a greater number of iterations trains gets us an unplayable agent and a smaller number of iterations gets you an agent who plays below par (most likely to lose to a human). Later the policies of these trained agents are saved.

HTML implementation for Tic Tac Toe:

Firstly, necessary variables are initialized. Initial page allows us to give 3 inputs: name, player's choice and difficulty level (Easy, Medium & Hard). Then it has a Start button which on clicking redirects to page 2 where we have our Tic Tac Toe matrix. Whenever a user picks(clicks) or agent picks a grid, the value of that grid changes from 0 to 1 and enables it with the respective symbol and color. At each step, it evaluates the board state and if it could determine that's one of the end states, it displays the message depending on who won or it's a tie. By clicking on the Exit button below this matrix, board state and input variables get reset and redirects us to page1.

Integration of HTML and Python code using Flask API:

Flask enabled us to send and receive data from front end to back end. We have defined two functions to post and get the data from both the pages. Necessary variables which are likely to be used in both the functions are defined globally.

In the first function (start ()), we take the player selection between 'X' or 'O' and difficulty level as input. According to player select input, Symbols for human & agent are determined and based on level of difficulty picked, we load corresponding policy (for easy level, we load policies extracted from agent who is trained for lesser iterations and for medium and hard levels, we load policies of agent who has been trained for greater number of iterations). If the user chooses to be player 2, the first move from the agent has been sent to the frontend to mark the agent's move.

In the second function (getPlace ()), the function takes the user's input, updates the board state, determines the available positions, evaluates the board state to check if it's a terminal state and gets the winner message. Based on the message from the check winner function, we choose to either give the agent its turn (if the winner message is empty) or return back the winner message

along an empty place variable. If the winner message is empty, the agent gets its turn. Agent sends the current state of the board, its available options to pick on the board and its symbol to the NextStep function and gets the next step looking at the Q-values loaded based on the options chosen in the first function. If the game terminates after agent moves, we display the winner message on the Tic Tac Toe page.

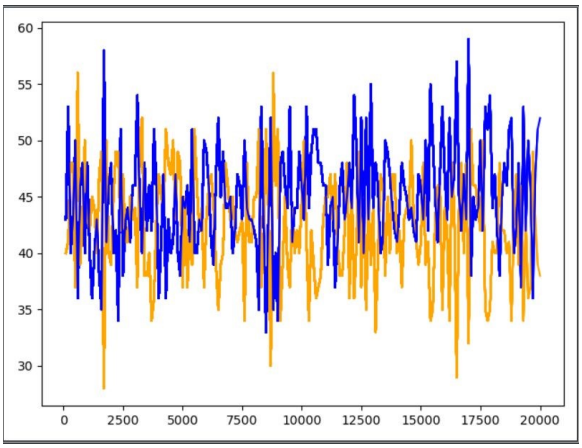

Fig- 1 Deep Q agent player wins every 100 games ; Player 1- Orange, Player 2- Blue; Y-axis-Number of wins by the player in the past 100 games; the agent is trained for 20000 episodes X-axis - Number of episodes of training done from beginning

From Fig 1 we can see that both the players are competing against each other to win. Both the players are playing well but both the agents can still be tricked into losing because they are trying to win but in few cases not trying to draw the game, preventing other player from winning, indicating that the agent still hasn't learned the concrete competitive strategies.

III. Results

1. *Deep Q agent :*

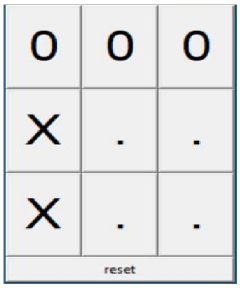

Fig 2 - Deep Q Agent vs Human where the agent won the game; X is Human; O is Agent

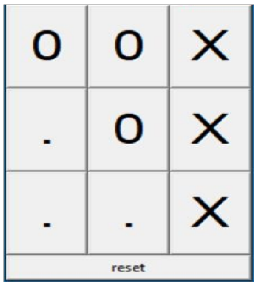

Fig 3 - Deep Q Agent vs Human where player won the game; X is Human; O is Agent

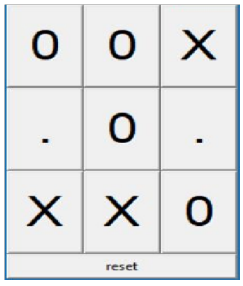

Fig 4 - Deep Q Agent vs Deep Q agent the where 1st player wins the game; X is 2nd player; O is 1st player

2. *SARSA Agent:*

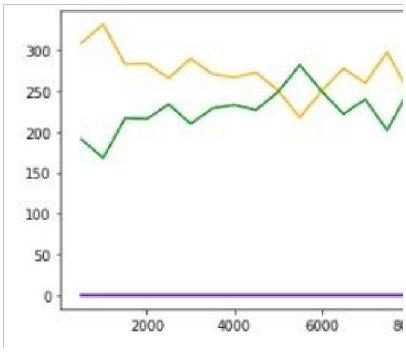

Fig- 5 SARSA agent player wins every 500 games ; Player 1- Orange, Player 2- Blue; (training the agents for 10000 episodes)
Y-axis-Number of wins by the player in the past 500 games;
X-axis - Number of episodes of training done from beginning

From Fig 5 and Fig 6 it can be observed that both the agents are competing against each other. For 1000 iterations, both the agents (SARSA agents one is player 1 and other player 2) haven't learned any strategies so it is randomly selecting moves

while for 7000 episodes both the agents are still learning the strategies but they are performing better. After 50000 episodes

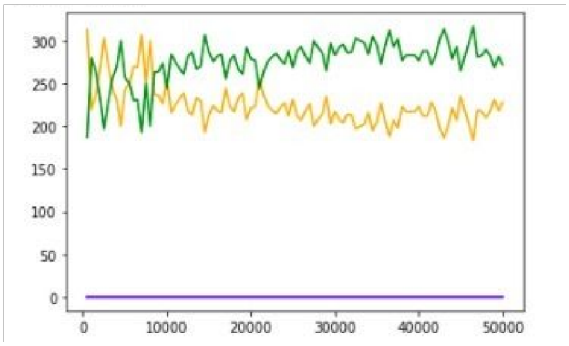

Fig- 6 SARSA agent player wins every 500 games ; Player 1- Orange, Player 2- Blue; (training the agents for 50000 episodes)
Y-axis-Number of wins by the player in the past 500 games;
X-axis - Number of episodes of training done from beginning

both the agents are playing competitively and doesn't allow the other player to win at least try for a draw if it is not winning. Below are the few games between human player and different levels of agent (level of the agent depends on how well the agent has trained).

Level 'Easy'-

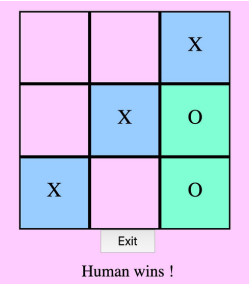

Fig 7 - SARSA agent vs Human where Human won the game;
X is Human; O is Agent

Level 'Medium'-

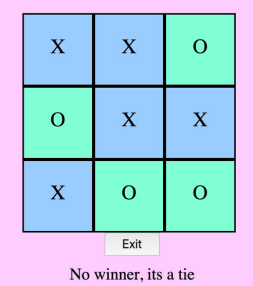

Fig 8 - SARSA agent vs Human where agent ties the game;
X is Human; O is Agent

Level 'Hard' -

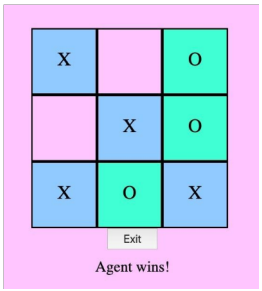

Fig 9 - SARSA agent vs Human where agent wins the game; X is Agent; O is Human

IV. Conclusion

Based on our results we found that SARSA agents were performing better than the Deep Q learning agents. This is because Deep Q learning requires very fine tuning of hyperparameters and even if fine tuned Q learning requires a lot of episodes to optimize itself to deliver a flawless performance.

V. Acknowledgments

We would like to express our deep gratitude to Professor Shreyasee, our course Instructor, for her enthusiastic encouragement of this project work. I would also like to thank TAs Yanjun Zhu and Tiehang Duan, for their advice and assistance.

VI. Contributions

Rohith Poshala has worked on training agent using SARSA learning, User Interface of the game and its deployment in AWS. Vishal and Sreekar worked on the Deep Q learning code and Report.

VII. Deployment

Game has been deployed on AWS server, below is the link for the game: http://18.223.101.138:8080/

Policies of both the players of each level are saved when the agents are trained previously so that each time the end user plays the game accessing the above link with no need to train the agent again and again.
Running the app in the local system:

Submitted saved policies, python source code file with name server.py along with a folder named template which has front end code with name tictactoe.html (basic structure of file placement for flask api to integrate both front and back end) should be placed in the same folder. And from the terminal we need to execute the python code (server.py) from the above folder using command: **python server.py** It works for Mac.

Once the application starts, the Tic Tac Toe game can be accessed in the local system using http://localhost:5000/ link.

The DQN can be run using **python game.py**

Fig 10 - UI screenshot

Fig 10 this is how the first page of UI looks like.

VIII. References

1. https://stackoverflow.com/questions/28021734/q-learning-algorithm-for-tic-tac-toe/28022044#28022044
2. https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0
3. https://www.instagram.com/p/B_kgrBPgG_Z/?igshid=1bt8erpvig0jx
4. https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html
5. https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html
6. https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542