Q-Learning & SARSA Agent for Tic-Tac-Toe

Professor Sidney Givigi
CISC 474 - Group 13
December 5, 2021

Brian Herman (20095996) - Baadshah Verma (10189808)
Makan Sabeti (20118161) - Matthew Rezkalla (20103458)

Table of Contents

1 Problem Formulation

1.1 Problem

For this project, we decided to implement the agent for Tic-Tac-Toe. It is a two-player, zero-sum game, played on a 3x3 grid. Two players will take turns placing their letter ‘X’ or ‘O’ on an open space. The game ends when one player has their letter in a full row, column, or diagonal. If there are no open spaces and no one has won, the game is a tie. This game is a solved game – which means that the outcome of the game can be correctly predicted from any position if both players play optimally. The goal of our project is to create a reinforcement learning model where the agent will always win or draw every game.

1.2 Model Architecture

State Space

Tic-Tac-Toe has a state space of 255,168 possible combinations of the board. This is derived from 9! subtracting the number of combinations that follow boards that have already been won. Each state can hold either an “X”, an “O”, or be left empty.

Action Space

The action space is the possible moves a player can make on any given board such that the next move is theirs. For example, ‘X’ won’t need to consider the states where it is the turn of ‘O’. The number of actions in each state is based on the total possible number (9) minus the number of moves made.

Reward Scheme

The rewards will be given after each game completes and will be as follows: 1 for a board where x wins, 0 for a tied board, -1 for a board where o wins.

Chosen Models

We implemented Q-learning and SARSA for this project. Coming into the project, we predicted that Q-learning would perform much better than SARSA given the results of assignment 2 and what we have learned. The purpose of this was to try multiple algorithms in order to find the most efficient one to use.

2 Algorithm and Implementation

2.1 Q-Learning

The first algorithm we decided to implement was Q-learning. Q-learning is a value-based learning algorithm using TD (temporal difference). This approach uses the maximum expected value for taking an action at each step to calculate the TD. This algorithm will find the sequence of actions that will produce the maximum total reward for each state. Q-learning is an off-policy approach that is guaranteed to converge to the optimal path given enough episodes, while SARSA will tend to learn the safer path. Both algorithms are similar, though Q-Learning uses the best approximation vs the expected value as seen below. This method will iteratively update state action values visited during the episodes using the Bellman equation seen below.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Q-learning was selected due to the fact it is guaranteed to identify an optimal action-selection policy. Since SARSA will find the safest path, we hypothesised that the policy found may be worse. This would be because it may avoid going towards "dangerous" states, even though moves are determinate. Thus, SARSA may tend to try to tie rather than try to win.

2.2 SARSA

The second model that we decided to implement was SARSA. This algorithm is also a value-based learning algorithm using TD. SARSA's name reflects the method used for updating the Q-values. It is based on the initial state S_t, the chosen action A_t, the reward for taking action A_t at S_t, new state S_{t+1} and the next action chosen at S_{t+1}, A_{t+1}.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

We chose to implement the SARSA algorithm over others due to the fact that we are already implementing Q-learning as the first algorithm. We were curious about what the results would look like due to the similar process. Specifically, it was interesting to see which model took less runs to find the optimal policy and how those policies differed.

2.3 Problem with Tic-Tac-Toe models

Model 1

Model 1 was mostly a naive attempt at fleshing out functions and methods for the agent. Though this model did not converge correctly, the methods were helpful and carried over into models 4 and 5. Model 1 was a single class for the agent which controlled both the environment and the algorithm. It was concluded that this caused many problems with both the code and the formatting. After a few trials with this model it was decided that further implementation should be continued with a model that has a

separate environment and agent. The hope was that this change would result in a working algorithm. Work was then started on model 4.

Model 2

When coming up with the model, we decided that the best approach would be if everyone designs their own models such that we get a diverse set of codes and can continue with the most efficient one. Model 2 presented a problem with winning the game when it goes first. The model used both a Q-learning and SARSA implementation to try and find an optimal policy. After around 1 000 000 episodes, the policy seemed to stop getting better, even at this point some issues were still present.

A major issue that we found was that when it had chances to finish the game, it would not take it. Instead the agent would more highly value trying to trap the player (getting a state in which any move by the player would result in their loss). However, the issue with that is, if you notice that plan, you can prevent losing and force a tie, even from a guaranteed losing state.

There were two different ways we attempted to resolve this issue. The first was to change the epsilon, gamma, and alpha values to try and achieve faster convergence. After multiple attempts at this we realized that the issue is probably something more fundamental to the model. We came to the conclusion that the source of this problem was the way we were selecting spots. We were selecting the first move randomly, which made it very difficult to train and get an optimal policy. Therefore, the solution was to figure out a way where we can teach the agent to choose an intelligent move first, instead of a random one.

Model 3

Model 3 was built using 2 classes, one representing the environment and the other for the learning agent. The model worked well acting as the first player (X), but had issues with the second player (O). This issue was unresolved, but it could have been a result of the initial states that the model used. This model used both Q-learning and SARSA implementations to learn. After running 100,000 episodes and testing a variety of parameters, it was still unable to perform well as the second-playing agent. The model rarely lost going first, but a path that would beat it could still be found. This suggests an improvement that could likely be made in the general way the model trains if we had decided to continue working on this model.

Model 4

Model 4 was based on Model 1 with the mentioned changes. The code class was split for functionality between an Agent and a Board class. The two Agent classes that would be responsible for training either using the Q-Learning or SARSA algorithm for X or O and held an instance of the Board class. The Board class was responsible for representing the environment for which each Agent could act on and get information from. This consisted of: taking actions, getting rewards or checking the winner. Training was done separately between two classes X and O that inherited the Agent class. For each agent, training would be done against random actions made in the Board class. Though this model did prove far more accurate than the previous Model 1, problems were found with the policies and the required number of episodes to find a policy that was good enough. We found that it was very plausible to instead have a single agent. This would be done simply by swapping between X and O choosing actions, and letting X

attempt to maximize the rewards while O would try to minimize. This allows us to consolidate the code and generate a single model to play against. These changes were made and the code was re-written as Model 5 seen in section 2.4 below.

2.4 Implementation

Documentation of the final implementation (Model 5) is found in the README.md file located here. This is an in depth description of our classes and the functions for the implementation of our models.

3 Results

Originally, we chose 0.9 as the discount value as this is somewhat conventional for TD algorithms. However, we soon realized that episodes are a maximum of 9 steps and thus there is no reason not to use a value of 1.0 as the discount as theoretically there is a very small amount of data to remember. We also want to avoid initial moves not being affected by rewards as they are given by the final move of an episode. In conclusion, we found that a discount of 1.0 was the best choice going forward.

We chose 0.3 as our epsilon value. Higher values for this parameter will cause the agent to spend a lot of time searching states that will almost never be seen in games. An example of this would be states that can only be achieved if both players are intentionally avoiding winning. Since the agent is assumed to always try to win, there is no need to search these states as they will not be seen. If epsilon is lower, the agent will tend to play the same games over and over since both X's and O's moves are chosen by the q values. Therefore we decided on 0.3 for the final epsilon.

We found that an alpha of 0.5 was overall adequate for the algorithm. After testing some other values, higher and lower, we did not find a reason to change it.

With these values, we ran the code for 100 000, 1 000 000, 10 000 000, and 50 000 000 episodes. To analyze the conversion, we graphed the largest change over all steps per episode **Figure 1**.

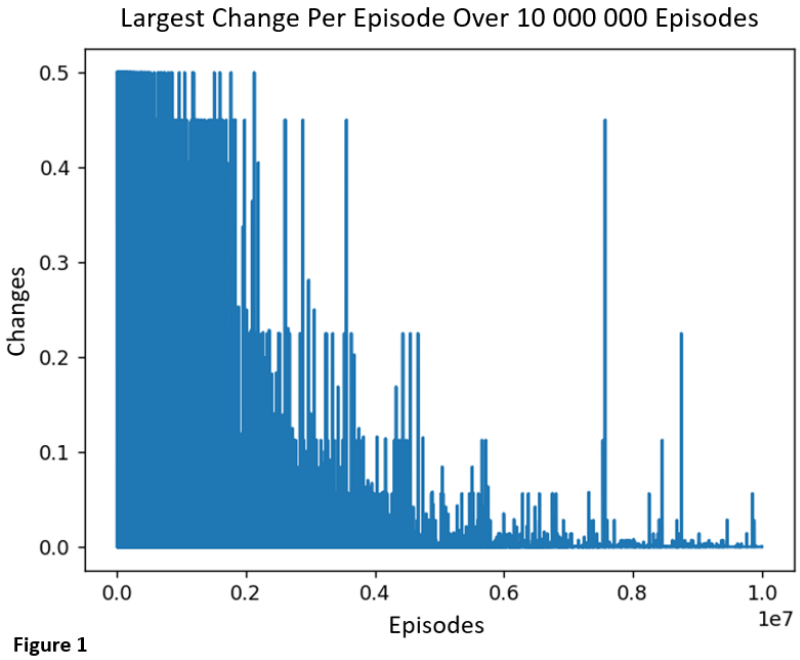

Figure 1

We found that the values were relatively converged around 6 000 000 episodes and further around 8 000 000 with a few outliers. These outliers are unlikely states resulting from the repeated choice of epsilon random actions.

We then tested to see the percentage of X wins, O wins and ties over the course of 100 000 episodes. Based on Tic-Tac-Toe statistics, X is far more likely to win than O due to the fact they play first and get an extra move. As well, if both players are taking optimal actions, then every game will be a tie. Thus we assumed that for a naive agent (run for a small number of episodes) such that actions chosen are relatively random or uninformed, both players may win games with X winning more than O. We also assumed that as the episodes progressed the rate of either X or O winning would go down as policies converged to optimal for both players and rate of ties would increase towards 100% (disregarding games won/lost based on the agent choosing epsilon actions). Final results of this test are found below in **Figure 2**.

One unusual aspect of our algorithm is that no initial state space is recorded. States are discovered and recorded as the algorithm continues. This is done for two reasons, the state space is very large and inefficient to iterate through, and many states will not be achievable in real games while the agent makes optimal moves. As described in the decision of the epsilon value, if the agent makes optimal moves, many states cannot be achieved in games as they require the players to intentionally avoid winning. To illustrate this point, we recorded the number of visited states over the course of 100 000. As seen in **Figure 3,** after around 20 000 episodes the number of new states visited is close to, or is 0. Thus we can conclude that this is an appropriate way to complete the algorithm without having to record the entire state space.

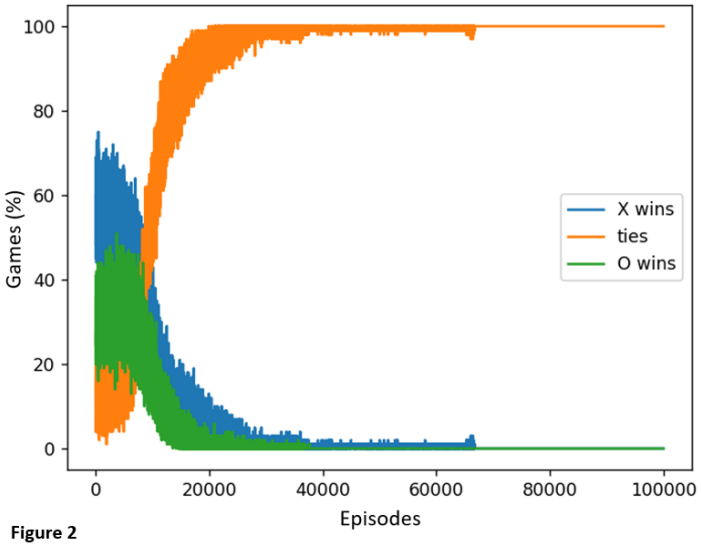

Figure 2

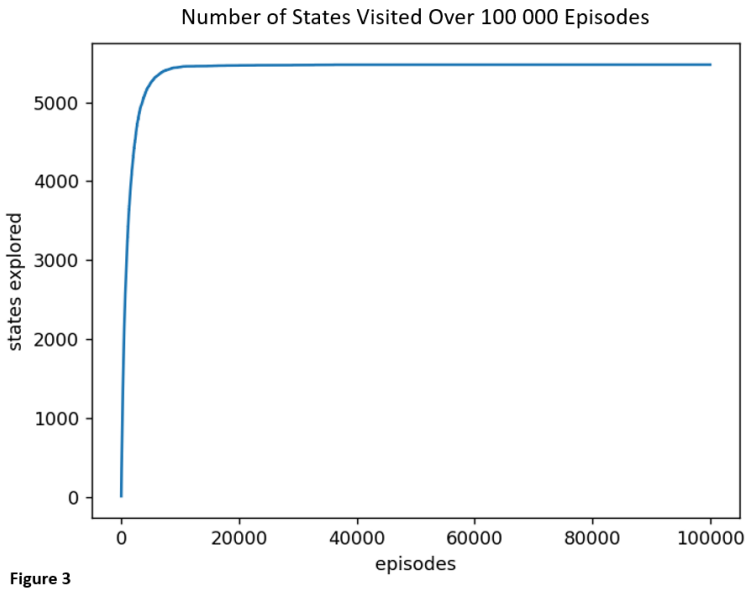

Figure 3

4 Discussion

Overall, the project was a great learning experience. Each of the group members worked on an individual model and was able to try and take the challenge of the problem head-on. Tic-Tac-Toe is a straightforward game, but as we built our models, the unforeseen obstacles began to come up. Questions regarding how we trained our models, if we needed multiple policies, if we should account for who starts, and many more arose. Through this process, 2 models produced non-optimal yet somewhat decent policies and only 1 consistently generated the optimal policy. The other models either did not run properly or outputted policies with many faults. The final model remains the most efficient and straightforward solution. We learned a lot about working with artificial intelligence models and algorithms and some of the common difficult problems that we will likely face in the future. The project taught each of us important lessons in the planning and building of AI Models and provided a good platform to continue learning and designing new models.

5 Group Contributions

Member	**Contributions**
Brian Herman	Final model (Model 5), Model 1, Model 4 Results Problems with tic-tac-toe models (Model 1, Model 4)
Matthew Rezkalla	Model 2, Model 3 Discussion Problems with tic-tac-toe models (Model 2, Model 3)
Makan Sabeti	Model 2 Problem formulation Algorithm & Implementation (SARSA) Problems with tic-tac-toe models (Model 2)
Baadshah Verma	Model 2 Problem formulation Algorithm & Implementation (Q-Learning) Problems with tic-tac-toe models (Model 2)