Shallow Q-Network for Tic-Tac-Toe

Siddharth Chaitra Vivek
2022A7PS0569G
BITS Pilani, Goa Campus

November 16, 2024

Abstract

This report outlines my attempt at implementing a Shallow Q-Network (SQN) to play Tic-Tac-Toe. It includes a brief overview of the problem statement, the idea behind approaching the problem with a SQN, a theoretical explanation of the network and Q-Learning algorithm, the various attempts/methods of implementation and training and finally a discussion on the findings.

Problem Statement

Game Environment

Tic-Tac-Toe is a simple yet strategic 3×3 board game where two players, 'X' and 'O', alternately place their respective marks on empty cells. The goal is to align three marks consecutively in a row, column, or diagonal. The game ends in one of the following scenarios:

- A player wins by achieving a straight line of their marks (row, column, or diagonal).
- The board is completely filled without any player winning, resulting in a draw.

State: The game state is the 3x3 board itself. Each cell can be represented as 0 (empty), 1 (agent's mark), or -1 (opponent's mark). The state in the code is just represented as an array of 9 integers:

$$[1 \quad -1 \quad 1 \quad 0 \quad -1 \quad 0 \quad 0 \quad 1 \quad 0]$$

Action: The action is selecting a cell to place the agent's mark.

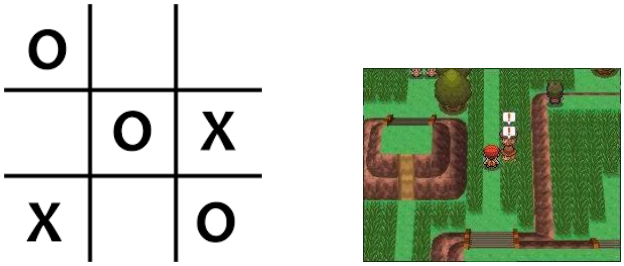
Objective

The primary objective is to design and train a Q-network that can play Tic-Tac-Toe effectively by:

- Selecting actions (placing marks) that maximize the probability of winning or achieving a draw.
- Minimizing the likelihood of losing by avoiding suboptimal moves.

Approaching the Problem

Deep Q-Networks (DQNs) are essential in reinforcement learning for tackling complex environments with vast or continuous state spaces, which traditional Q-learning cannot handle efficiently due to the inability to have a lookup table for Q-values. By using neural networks to approximate the Q-value function, DQNs make it possible to larger state spaces and enable generalization across unseen states..
(eg: Something like a Nintendo DS Pokemon Game where you can walk around the world and interact with so many different objects etc)



(a) *Small state space and low complexity game in the case of tic-tac-toe*

(b) *Much larger and complex state space in video games*

Figure 1: *Two images side by side in a single column.*

In the context of Tic-Tac-Toe, the use of a Shallow Q-Network (SQN) , as opposed to a DQN, is sufficient due to the simplicity of the game's state and action space. Tic-Tac-Toe has a limited and discrete state space, with only 3^9 possible board configurations (considering the three states of each cell: empty, X, or O) and a finite set of legal moves at each turn. This smaller complexity allows a SQN, consisting of just a few layers, to effectively learn a mapping between game states and optimal actions without the need for a deeper network. Furthermore, the task does not involve high-dimensional or continuous input data like images etc, this makes it an ideal choice for solving such a straightforward reinforcement learning problem.

Theoretical Explanation

Q-Function and Network Architecture

- **Q-Function**: The Q-function, $Q(s, a)$, represents the return of taking action a in state s. Here, the return is positive for a win, zero for a draw, and negative for a loss.
- **Shallow Network**: A shallow neural network approximates the Q-function. The network inputs are the current state of the board, and the output is the Q-value for each possible move. Hence 9 nodes in the input layer and 9 nodes in the output layer

Reinforcement learning involves training an agent to take actions in an environment to maximize cumulative rewards. The framework consists of the following components:

- **State** (s): Represents the current configuration of the Tic Tac Toe board.
- **Action** (a): Denotes a move (placing a symbol in an empty cell).
- **Reward** (r): Feedback signal based on the outcome of an action.
- **Policy** ($\pi(a \mid s)$): Defines the agent's strategy, mapping states to actions.
- **Q-Value** ($Q(s, a)$): Quantifies the expected future rewards of taking action a in state s.

Shallow Q-Learning Overview

Shallow Q-Learning extends traditional Q-Learning by approximating the Q-function with a neural network. This is necessary as the conventional approach of using a Q-Table will not work in case of environments with a large state space and many possible actions The network takes a state as input and outputs Q-values for all possible actions. Training involves minimizing the **mean squared error (MSE)** between predicted Q-values and target Q-values derived from the Bellman equation:

$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$$

where:

- r: Reward for the current action.
- γ: Discount factor controlling the importance of future rewards.
- s': The next state resulting from action a.

Replay buffer in SQN

In SQN, the **experience replay buffer** plays a key role in helping the agent learn more effectively. When the agent interacts with the environment, it generates a series of experiences. These experiences are stored in the buffer and consist of a tuple of the following form:

$$(s, a, r, s', \text{done})$$

Where:

- s represents the current state of the environment, before taking any action.
- a is the action the agent decides to take in state s.
- r is the reward the agent receives after taking action a.
- s' is the next state the agent moves to after taking action a.
- done is a flag that tells whether s' is a terminal state (i.e., the episode has ended). If true, it means the agent has reached a terminal state; otherwise, the episode continues.

The replay buffer helps by storing past experiences and randomly sampling mini-batches from them to train the Q-network. This random sampling helps break the correlation between consecutive experiences.

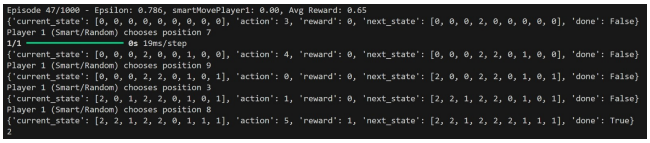


Figure 2: *Representation of the contents of the replay buffer (experiences) done using a 5-tuple stored as a dictionary*

Implementation approach 1: Simultaneous Training and Data Collection

My original idea and approach was as follows:
As the agent plays, it stores the experiences from each move in a replay buffer. Once a game (episode) ends, the agent samples a random batch of experiences from the replay buffer to train its neural network.

Throughout the process the ϵ value continues to decay to complement the fact that the model is improving and hence needs to do less exploration and more exploitation.

There is also a separate deque called scores that keeps track of the outcomes of the games. 1 for win, -1 for loss and 0 for a draw. This deque has

a fixed window size and is used to decide when a model is performing well for a certain smartness level and can play against a harder opponent. Average performance is calculated by taking the average of the scores within the window stored in the deque and if it meets a certain threshold and enough episodes have been played, the smartness of opponent gets incremented.

The pseudocode of this method is shown below:

Algorithm 1 Simultaneous Training and Data Collection

Function main(smartMovePlayer1)
Initialize playerSQN, number of episodes, window size for performance tracking.
Initialize smartMovePlayer1 as 0.
for each episode **do**
  Initialize the game and state. Player 1 starts of with a move
  while game is not over **do**
    Player 2 (SQN-based) makes a move.
    If move is valid, calculate reward and next state.
    Store experience in replay buffer.
    if game ends **then**
      Update performance scores.
      Decay exploration rate of Player 2.
      Exit the loop.
    end if
    Player 1 (random) makes a move.
  end while

  Sample a batch size of 32 from the replay buffer and train the model on this minibatch for a given number of epochs

  Calculate average performance of recent episodes.
  if performance improves **then**
    Gradually increase smartMovePlayer1.
  end if
  Print episode summary.
end for
Save the trained model.

Results and Discussion

The main issue with this approach is due to instability in training. Upon running with various different hyper parameter settings this methods always converges way too slowly each time struggling to move past 0.3/0.4 smartness over 1000 episodes played or it doesn't converge at all.

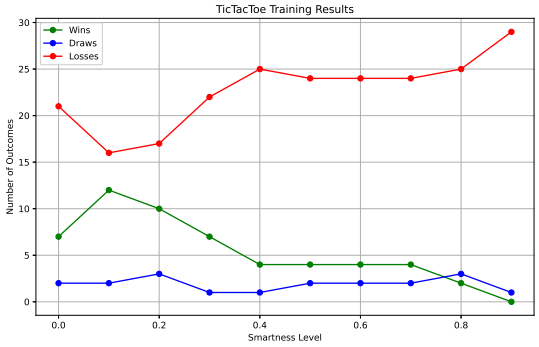


Figure 3: *Testing the model against different smartness levels for 30 runs at each level*

NOTE: Interesting Results on letting SQN Play first

An interesting observation is the large improvement in performance when the SQN is allowed to play first. Intuitively this obviously means better results as the first player has an advantage in Tic-Tac-Toe. However this also led to much lesser fluctuation and instability and it led to the model performing really well and winning a large percentage of the time even again a very smart opponent.
Shown below are the results of this configuration:

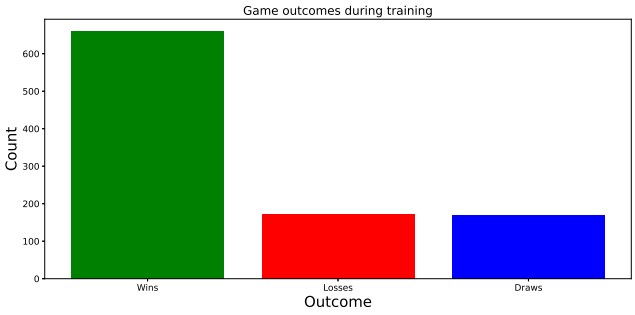


Figure 4: *The model won a large number of times over the 1000 episodes*

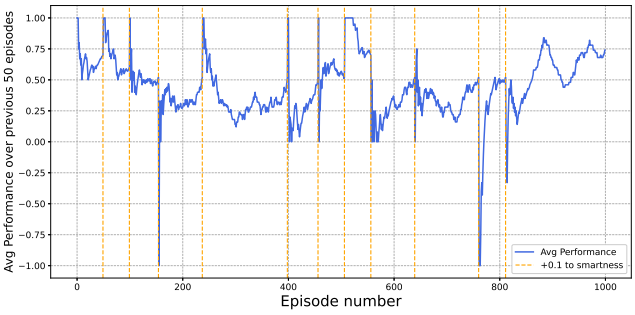


Figure 5: *Shows the average performance of the model versus the episode, which is here an average of the most recent outcomes. The yellow lines indicate when the smartness of opponent increased*

Implementation approach 2: Alternating data collection and Training

This approach has the same components as the previous method except for one change where instead of training on a sampled batch after each episode is simulated , a large number of experiences are collected in the buffer through simulating episodes and then multiple samples are drawn from this buffer to train on. The buffer is then cleared and process is repeated.
The pseudocode is shown below:

Algorithm 2 Alternating data collection and training

Function main(smartMovePlayer1)

Initialize PlayerSQN, number of episodes per smartness level, and smartness increment parameters.
Set initial smartness to 0.
while smartness is within the allowed range **do**
  Simulate episodes for the current smartness level:
  for each episode **do**
    Initialize the game and state.
    while game is not over **do**
      Player 2 (SQN-based) makes a move.
      If move is valid, calculate reward and store experience.
      If game ends, decay Player 2 exploration rate.
      Player 1 makes a move.
    end while
  end for
  Train the model on multiple batches from the replay buffer.
  Clear the replay buffer after training.
  Increment the smartness level for Player 1.
end while
Save the trained model.

This method generates moderately good results but it doesn't perform well against higher smartness opponents indication that it hasn't converged. Performance decreases drastically towards the end and the only wins/draws it is getting is due to luck where the smart opponent is playing in an exploratory fashion.

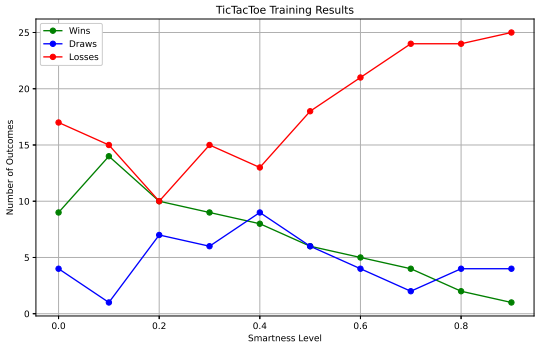


Figure 6: *Testing the model against different smartness levels for 30 runs at each level*

Discussion and Future work

The experiments produced results that were less than ideal. Here, here are some limitations of shallow Q-learning and potential reasons behind its underperformance in developing a TicTacToe player. These challenges, along with possible explanations, are discussed to provide a clearer understanding of the observed outcomes.

Being Player 2

Being the second Player is inherently harder, and you are more likely to win being player 1. As seen in an earlier section, the results were significantly better when I tested for a case where the SQN played first. The same advantage applies to how easily a model can improve and play

Insufficient Exploration

The balance between exploration and exploitation is critical for Q-learning. Poorly tuned exploration parameters (e.g., ϵ in ϵ-greedy strategies):

- Can lead to **premature convergence** to suboptimal strategies.
- May result in the agent never experiencing certain critical states (e.g., endgames where winning patterns emerge).

Reward Design Issues

A lack of more comlex reward structures can impede effective learning. For example:

- Providing rewards only at the end of the game might delay reinforcement signals critical for strategy learning.
- Lack of intermediate rewards (e.g., for blocking the opponent or making advantageous moves) can result in the agent failing to understand game dynamics.

Hyper-Parameter Sensitivity

Hyperparameter tuning in Q-learning is non-trivial. One possible improvement would have been more rigourous hyperparameter finetuning

- **Learning Rate (α):** A high learning rate can lead to unstable updates, while a low rate may result in slow convergence.
- **Discount Factor (γ):** A poorly chosen discount factor might either overemphasize immediate rewards or fail to prioritize future gains.

Overfitting to Opponent Behavior

A TicTacToe player trained using shallow Q-learning might inadvertently overfit to the specific behavior of the opponent during training. This is particularly true when training against static or suboptimal opponents, leading to poor generalization against unseen strategies.

Conclusion

In conclusion, this research aimed to apply Shallow Q Networks (DQN) to the game of Tic-Tac-Toe, offering an interesting approach to solving a classic game with reinforcement learning. While the results didn't reach the ideal performance levels, they provided valuable insights into both the strengths and challenges of using SQN for this task. The model was able to learn basic strategies, but it struggled to consistently make optimal decisions. However, the process highlighted several key areas for improvement, such as fine-tuning hyperparameters, refining the reward system, and improving the exploration strategy.

I will spend more time on the project to try and explore the ideas from future work as well as run the current codebase for a higher number of training episodes to see if there is any improvement using the current ideas.