CS F407 - Artificial Intelligence

Report on Programming Assignment #2

Submitted To :

Professor Dr. Sujith Thomas

BITS Pilani

Submitted By:

Yashi Malik
2021A3PS3056G

On:

Date: 16/11/2024

Table Of Contents

1. Introduction

This assignment focusses on the implementation of a shallow Q-Network(SQN) for the game Tic-Tac-Toe by using reinforcement learning. We use a shallow neural network to approximate the Q-values for different state-action pairs. We allow our SQN agent to play multiple games against player 1 that uses the ε-greedy policy, as a result we achieve a stable training through experience collection and replay.

2. Methodology

Neural Network framework

a) **Input layer-** Takes a 9 dimensional vector representing the Tic-Tac-Toe board state where each position is encoded as -1 (player2), 0 (empty), 1(player1)

b) **Hidden layers-**Two dense layers with 64 neurons each using the RelU activation function.

c) **Output layer-** 9 neurons with linear activation function, corresponding to the Q-values for each possible state.

Experience Replay and Epsilon-Greedy Policy

a) **Experience Replay-** We are using an experience replay buffer to store tuples of the form (s,a,r,s′, done) and then mini-batches of experiences are randomly sampled from this buffer for training.
b) **Epsilon Greedy Policy-**The agent starts with exploring the environment using a high epsilon (ϵ=1). As training progresses the epsilon decays enabling exploitation over exploration.

Updating Q-value

We are using the Bellman equation to update Q-values:

$Q_{target}(s,a)=r+\gamma_{a'}maxQ(s',a')$

r -reward received
γ-discount factor set to 0.95
s′-next state

Prioritizing certain actions

```
# Give -1 to all the used positions
for i in range(9):
    if (state_array[i] != 0): predicted_val[i] = -1
    pass

game = TicTacToe()
game.board = state

# Give 0.9 to block opponent
for position in game.empty_positions():
    game.board[position] = 2
    if game.check_winner(2):
        predicted_val[position] = 0.9
    game.board[position] = 0

# Give +1 if we are winning
for position in game.empty_positions():
    game.board[position] = 1
    if game.check_winner(1):
        predicted_val[position] = 1
    game.board[position] = 0
```

a) Marking all the occupied positions with a Q-value of -1 to prevent choosing invalid actions
b) Assigning a Q-value of 0.9 to actions that would block the opponent from winning.
c) Assigning a Q-value of +1 to moves that result in the immediate win for player 2.

By strategically rewarding winning/blocking moves I observed that the training was much faster(converging to a better win rate) which lead to an exponential fall in the number of losses as compared to my previous models.

Training Process

I have split the training process into two phases to ensure effective learning and to address the issues like data insufficiency and instability:

a) **Phase 1 (Data Collection)-** The agent plays 500 episodes against player 1 using a purely random strategy (`smartMovePlayer1=0`). The agent stores the experience tuple `(state, action, reward, next_state, done)` in the replay buffer.
This phase focuses solely on data collection, and no training of the neural network occurs.

b) Phase 2 (Training loop)- In this phase, the agent trains using the experiences stored in the replay buffer and continues to gather new experiences through self-play.
The agent trains periodically after every 5 episodes, which ensures that the model is not updated too frequently that can lead to overfitting or divergence.
Also the agent randomly samples a mini-batch of experiences from the replay buffer which helps break the correlation between consecutive experiences making the training process more stable.

Dynamically adjusting the difficulty level of opponent

```
if win_rate > 0.5 and current_difficulty < smartMovePlayer1:
    increase = min(0.1, smartMovePlayer1 - current_difficulty)
    current_difficulty = min(current_difficulty + increase, smartMovePlayer1)
    print(f"Increasing difficulty to: {current_difficulty:.2f}")
    consecutive_wins = 0

if win_rate > 0.5 and current_difficulty == smartMovePlayer1: break
```

While training I am giving the target difficulty from the command line and I am increasing the difficulty by 0.1 starting from 0 till it reaches the target difficulty. The increment happens when the win rate exceeds 50%.
The adaptive difficulty mechanism helps prevent overfitting. If the agent only plays against a fixed-level opponent, it may overfit to that specific strategy.

3. Results

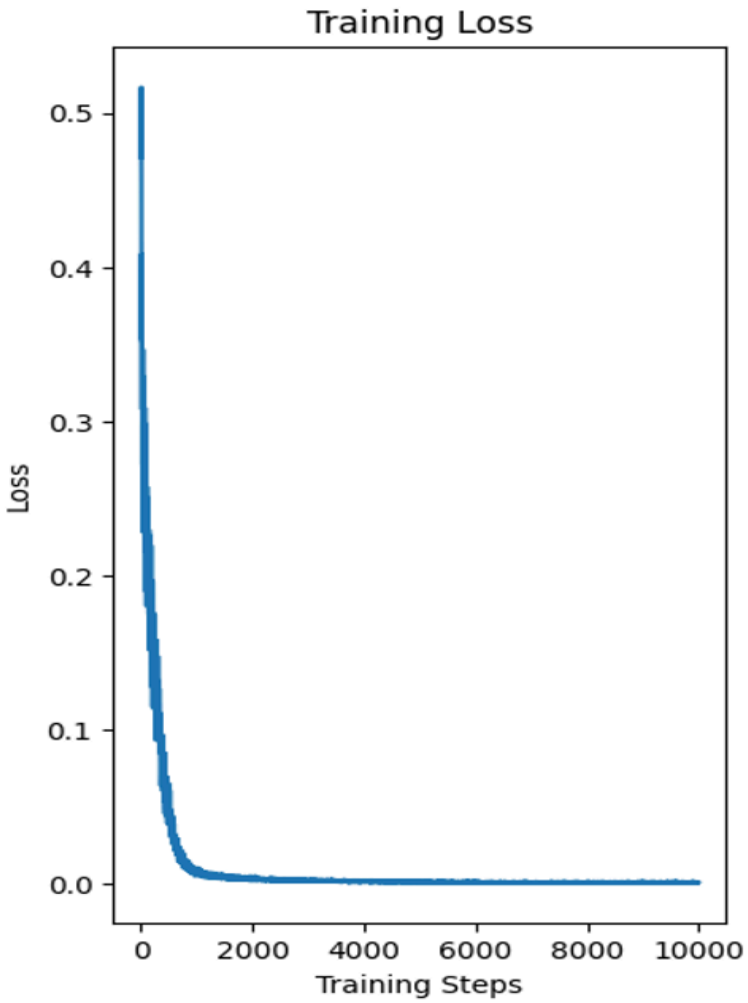
Evaluation results over 100 episodes for 0, 0.5 and 1.0 smartness

```
Evaluation Results for 2021A3PS3056G.py
================================================


Smartness 0:
Wins: 71 (71.0%)
Draws: 22 (22.0%)
Losses: 7 (7.0%)
------------------------------

Smartness 0.5:
Wins: 46 (46.0%)
Draws: 41 (41.0%)
Losses: 13 (13.0%)
------------------------------

Smartness 1.0:
Wins: 11 (11.0%)
Draws: 57 (57.0%)
Losses: 32 (32.0%)
------------------------------
```

The agent performed best against the opponent playing randomly and as the smartness was increased the win rate dropped, there were still not many losses because of the reward that I gave for blocking the opponents's moves.

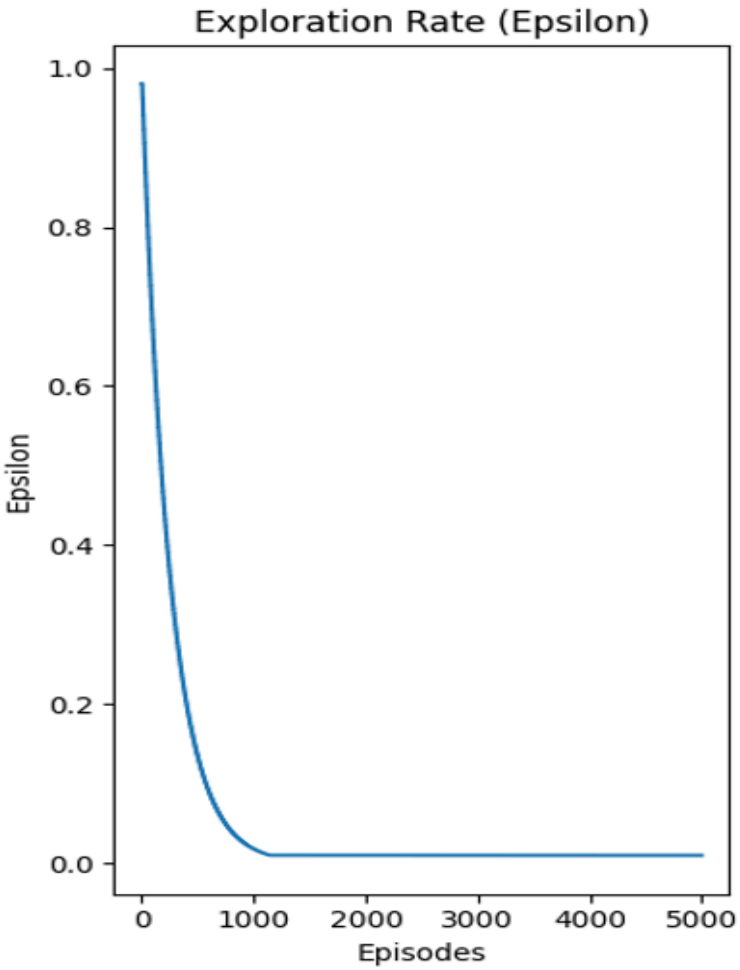
Plots

a) Training Loss



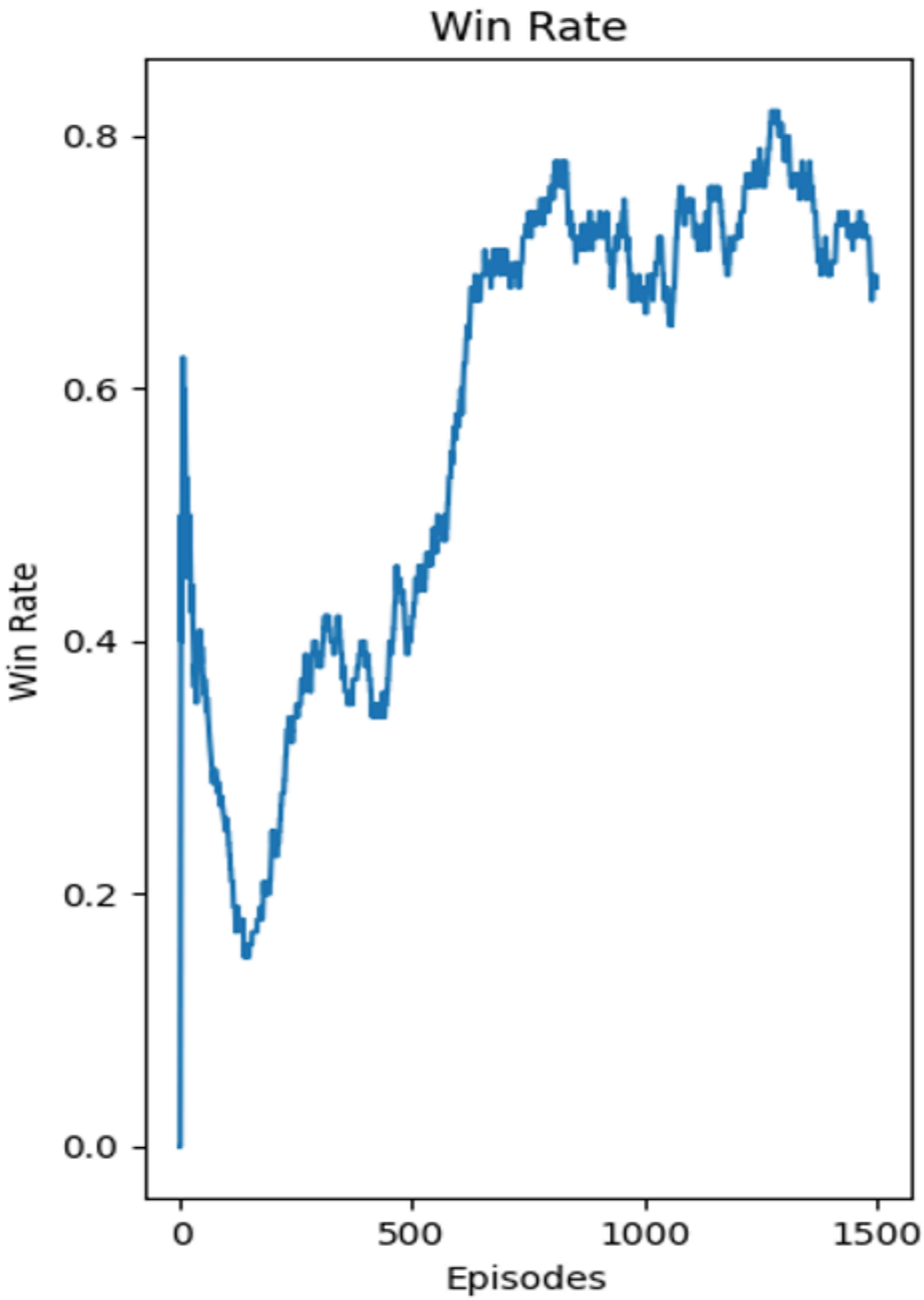
The training loss decreases exponentially indicating effective learning. Some spikes were observed in the initial few training steps due to exploration. As the exploration rate decayed, the loss stabilized, demonstrating that the agent was learning to make consistent predictions for Q-values.

b) Exploration Rate(epsilon)



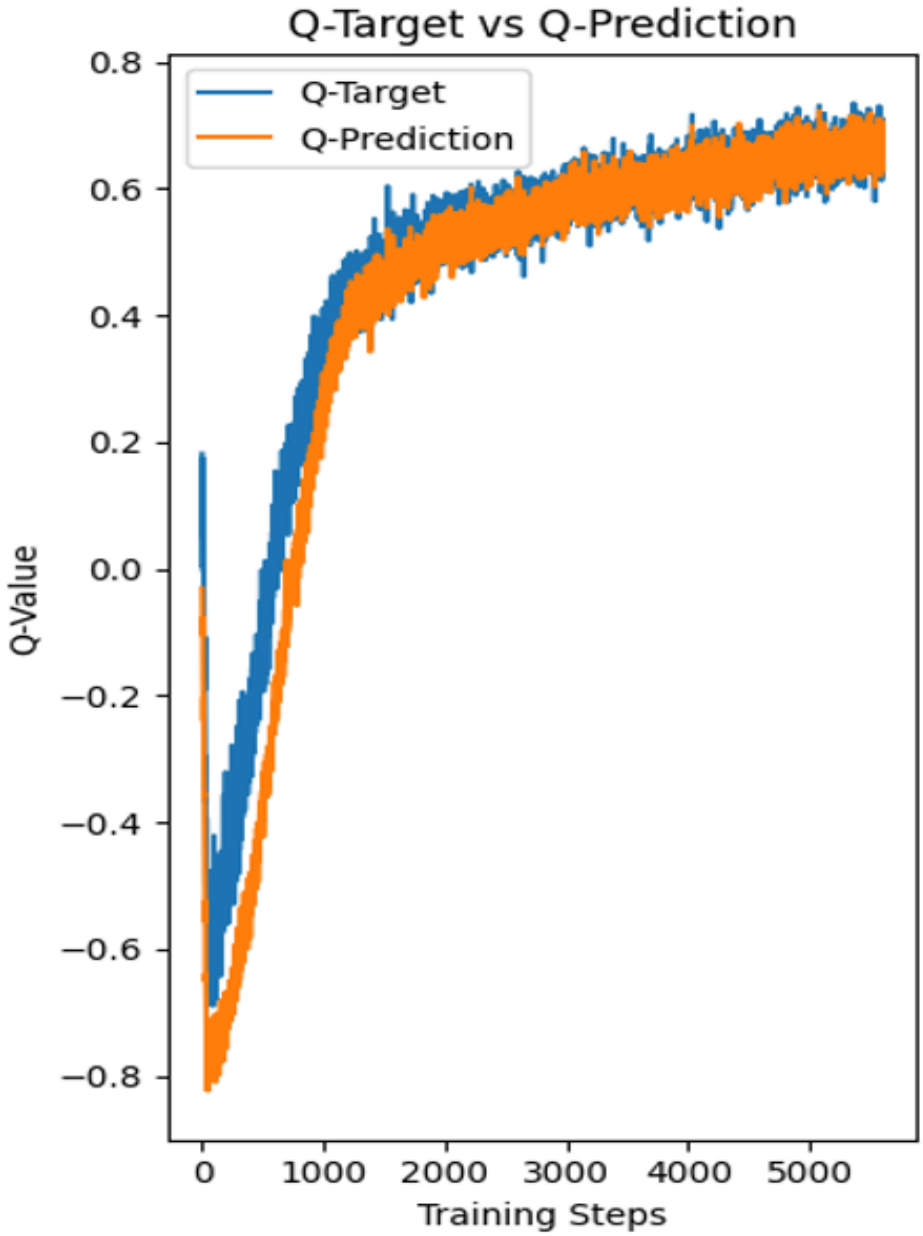
The epsilon value decreases as expected, transitioning the agent from exploration to exploitation. This decay allowed the agent to gather a diverse set of experiences early on, while later focusing on refining its learned strategies.

c) Win Rate



The win rate increases as a whole but there are spikes and drops because I am training the model at different smartnesses, so when the smartness increases the win rate drops a little and increases again. It kind of saturates at around 70%.

d) Q-target V/s Q-prediction



The Q-predicted values converge to the Q-target values as the number of training steps increase, highlighting reduced prediction error and accurate learning.

4. Discussion

After training 8-9 models these were my observations/challenges that I faced:

a) Overfitting issue got better when I implemented the adaptive difficulty mechanism (increasing difficulty dynamically).

b) The win rate increased when I heavily rewarded blocking and winning actions and penalised invalid moves.

c) Instability caused due to high epsilon value at the start of training was resolved by dividing the training into 2 phases. The agent collected experiences without training the model, which allowed the replay buffer to be populated with a diverse set of samples before learning began. This strategy reduced the likelihood of the model overfitting to early, suboptimal policies.

d) The chosen decay factor (`epsilon_decay = 0.998`) provided a balanced approach, allowing the agent sufficient time to explore various actions early on, while gradually shifting towards exploitation as the training progressed.

5. Conclusion

In this project, we implemented a Shallow Q-Network (SQN) for Tic-Tac-Toe, successfully training the agent to play the game using reinforcement learning. The use of experience replay and an adaptive difficulty mechanism helped stabilize training and improve the agent's performance over time. Manual adjustments for blocking and winning moves accelerated learning, allowing the agent to quickly grasp critical strategies.

Overall, the SQN agent achieved a win rate of around 65-70%, demonstrating its ability to learn effective strategies through self-play. The project highlighted the potential of neural networks combined with Q-learning in game environments, while also revealing areas for future improvement, such as exploring deeper networks and diverse opponent strategies.