Pac-Man Game Playing AI

Group 9 Members

Satyam Patel - 8975977
Yakub Yekini - 8939553
Harsh Rajkumar Yadav - 8977229

GitHub Repository

https://github.com/Harshahir01/pac-man-game/tree/main

Introduction

Project Proposal

The initial proposal for our project aimed to develop a Pac-Man game where an AI agent, specifically a Deep Q-Network (DQN), learns to play the game by avoiding enemies and collecting rewards. The game environment would be created using Pygame, and the DQN agent would be implemented using TensorFlow/Keras.

Feedback and Changes

Based on the feedback received, we made several adjustments to enhance our project:

1. **Enhanced Game Environment**: We included multiple enemies to increase complexity.
2. **Visual Score Display**: A visual counter was added to display the number of rewards collected by Pac-Man.
3. **Optimized Agent Training**: Refined the reward structure to improve the agent's learning process.

Project Goals

- Develop a functional Pac-Man game using Pygame where Pac-Man can move and interact with enemies and rewards.
- Implement a DQN agent that learns to play the game using reinforcement learning.
- Display the score in the game interface.

Implementation and Progress

Game Environment

We successfully created the Pac-Man game environment using Pygame. The game includes:

- **Grid-Based Board**: The game board is designed with walls, paths, and rewards placed strategically.
- **Pac-Man Control**: Pac-Man is controlled by the DQN agent, moving through the maze.
- **Enemies**: Three enemies move randomly within the grid, posing a challenge to Pac-Man.
- **Visual Score Counter**: A score counter that displays the number of rewards collected by Pac-Man is prominently shown on the game interface.

DQN Agent

The DQN agent was implemented using TensorFlow/Keras. Key aspects of the implementation include:

- **State Representation:** The game state is represented as a flattened array, including the positions of Pac-Man, enemies, walls, and rewards. This comprehensive state representation allows the agent to make informed decisions.
- **Action Space**: Pac-Man can move up, down, left, or right, providing a simple yet effective action space for the agent.
- **Reward Structure**: Rewards are assigned for collecting pellets, while penalties are given for colliding with enemies. This structure encourages the agent to maximize its score by collecting rewards and avoiding penalties.
- **Training**: The agent is trained using the DQN algorithm, where it learns to maximize the cumulative reward over time.

Current Results

Game Interface: The game runs, but it experiences significant lag and frame drops, leading to a less smooth user experience.

Score Display: The score counter accurately reflects the number of rewards collected by Pac-Man, providing real-time feedback on the agent's performance.

Challenges and Resolutions

Challenges Faced

State Representation: Ensuring the state array accurately reflects the game environment was challenging.

- **Resolution**: Implemented thorough boundary checks and extensive debugging to validate the state array, ensuring accurate representation of the game state.

Training Stability: The agent initially exhibited unstable training with fluctuating performance.

- **Resolution**: Tuned hyperparameters such as learning rate, discount factor, and replay buffer size to stabilize the training process and improve performance.

Performance Issues: The game experiences significant lag and frame drops during execution.

- **Resolution**: Identified potential performance bottlenecks in the code. Next steps include optimizing the Pygame loop and reducing computational overhead in the DQN agent's training process.

Open Ticket / Open Question

Optimal Hyperparameters: Further tuning of hyperparameters could enhance the agent's performance, making it more efficient and effective.

Advanced Enemy Behavior: Exploring more sophisticated enemy movements, such as tracking Pac-Man, could add complexity and realism to the game.

Future Work

Next Steps

Fine-tuning: Continue refining the DQN agent's hyperparameters to achieve better performance and stability.

Enhanced Game Dynamics: Introduce power-ups and more complex enemy behaviors to make the game more engaging and challenging.

Performance Optimization: Optimize the Pygame loop and the DQN training process to reduce lag and frame drops, ensuring a smoother game experience.

Extensive Testing: Conduct thorough testing to ensure the stability and robustness of the agent, identifying and addressing any remaining issues.

Game Update to Full Version: Plan and implement a full version of the game, incorporating all improvements and additional features for a complete and polished final product.

Expected Outcomes

Improved Agent Performance: A well-trained agent capable of efficiently navigating the game environment and maximizing the score, demonstrating advanced strategies and decision-making skills.

Smooth Game Experience: An optimized and engaging Pac-Man game with minimal lag and frame drops, providing an enjoyable experience for users.

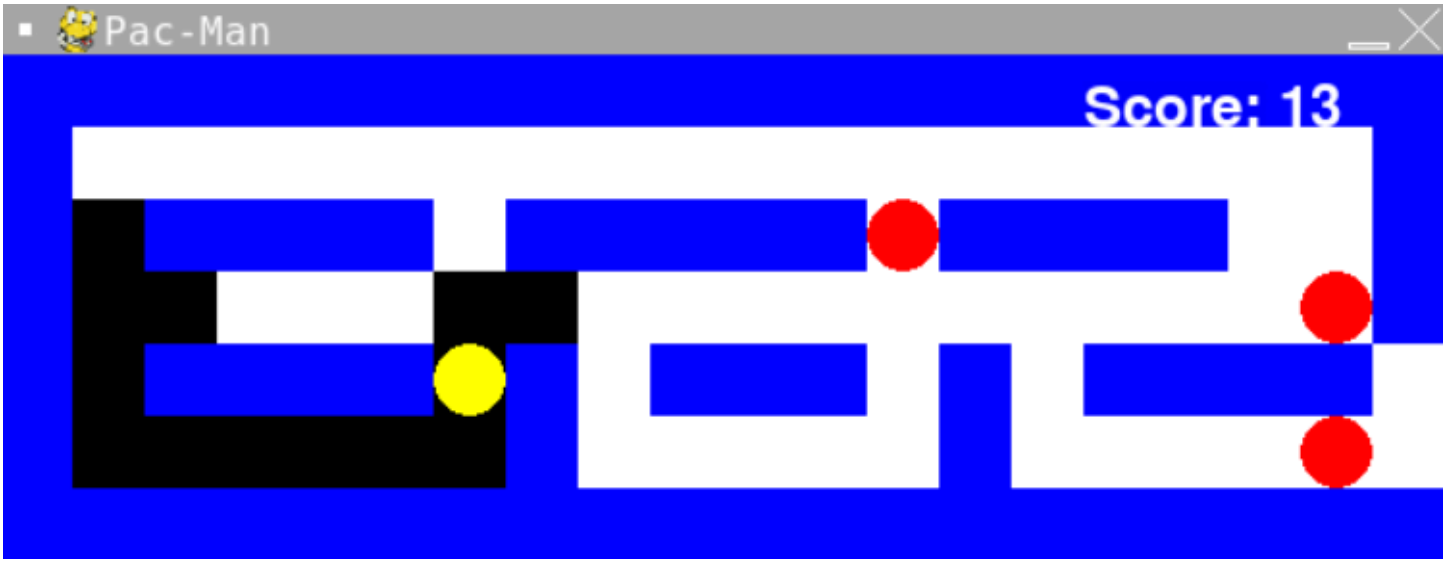
Rich Game Experience: A more dynamic and engaging Pac-Man game with advanced features, providing an enjoyable experience for users.

Proof of Work

Code

```
import pygame
import sys
import numpy as np
import random
from collections import deque
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import time

class DQNAgent:

    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95  # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse', optimizer=Adam(learning_rate=self.learning_rate))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])

    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                target = reward + self.gamma * np.amax(self.model.predict(next_state)[0])
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
```

Pac-Man Game



GitHub Repository

Harshahir01 / pac-man-game Public

Code Issues Pull requests Actions Projects Security Insights

main 1 Branch 0 Tags Go to file Code

yadavharsh2324 update 8da6c7d · yesterday 8 Commits

.gitignore	Initial commit	yesterday
.replit	Initial commit	yesterday
README.md	Create README.md	yesterday
main.py	update	yesterday
poetry.lock	Initial commit	yesterday
pyproject.toml	Initial commit	yesterday
replit.nix	Initial commit	yesterday
requirements.txt	Initial commit	yesterday

Code Explanation

The main.py file implements a Pac-Man game using Pygame and a Deep Q-Network (DQN) agent that learns to play the game. Below is a detailed explanation of the key components and how they work together:

Libraries and Imports

The script begins with importing necessary libraries:

- **pygame:** For creating the game environment.
- **sys**: To handle system-specific parameters and functions.
- **numpy**: For numerical operations.
- **random**: For generating random numbers.
- **deque**: For creating a double-ended queue, which is used for storing experiences.
- **tensorflow.keras:** For building and training the neural network model used by the DQN agent.
- **matplotlib.pyplot**: For plotting the results.

DQN Agent Class

The DQNAgent class defines the reinforcement learning agent that will play Pac-Man. Here's a breakdown of its components:

1. **Initialization (__init__ method)**:
   - state_size and action_size: Define the size of the state and action space.
   - Memory: A deque used to store past experiences (state, action, reward, next state, done).
   - Gamma: Discount rate for future rewards.
   - Epsilon: Exploration rate for the epsilon-greedy policy.
   - epsilon_min and epsilon_decay: Parameters to decrease epsilon over time.
   - learning_rate: Learning rate for the neural network.
   - Model: The neural network model built using the _build_model method.
2. **Building the Model (_build_model method)**:
   - Constructs a Sequential neural network with three dense layers.
   - The first two layers have 24 neurons each and use the ReLU activation function.
   - The output layer has action_size neurons with a linear activation function.
   - The model is compiled with mean squared error loss and the Adam optimizer.
3. **Memory Management**:
   - Remember method: Stores experiences in the memory deque.
4. **Action Selection (act method)**:
   - Chooses an action using an epsilon-greedy policy. With probability epsilon, a random action is selected; otherwise, the action with the highest predicted Q-value is chosen.
5. **Experience Replay (replay method)**:
   - Samples a random batch of experiences from memory.
   - Updates the neural network weights based on these experiences.
   - Adjusts epsilon to reduce exploration over time.

Game Loop and Pygame Setup

The game environment is set up using Pygame. Key components include:

- **Initialization**: Initialize Pygame, set up the display, and define game parameters such as the grid, colors, and frame rate.
- **Main Game Loop**: Runs the game, handling events, updating the game state, and rendering the screen.
- **Event Handling**: Processes user inputs (e.g., closing the window).
- **State Update**: Moves Pac-Man and enemies, checks for collisions, and updates the score.
- **Rendering**: Draws the game elements (Pac-Man, enemies, rewards) and the score on the screen.

Game Loop and Pygame Setup

Total Reward Over Time

This plot shows the total reward accumulated by the DQN agent over each episode. It helps visualize the learning progress and stability of the agent.

Epsilon Decay Over Time

This plot shows how the exploration rate (epsilon) decreases over time. It helps illustrate how the agent transitions from exploration to exploitation as training progresses.

Agent Performance: Score Over Time

This plot shows the game score achieved by the agent in each episode. It can help visualize improvements in the agent's performance.

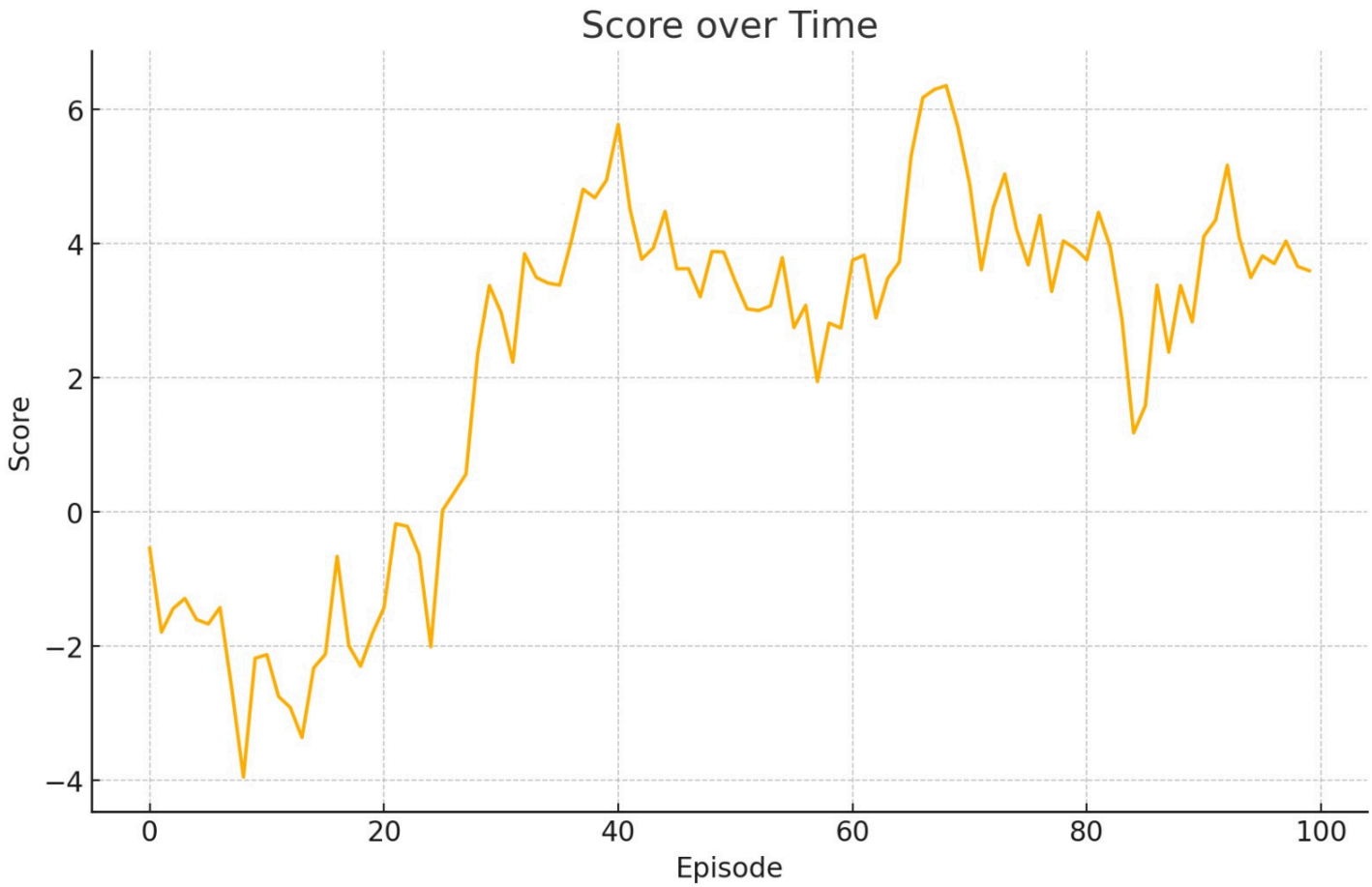
Reward Over Time

This plot shows the total reward accumulated by the DQN agent over each episode. The reward is a combination of positive rewards for collecting pellets and negative penalties for colliding with enemies.

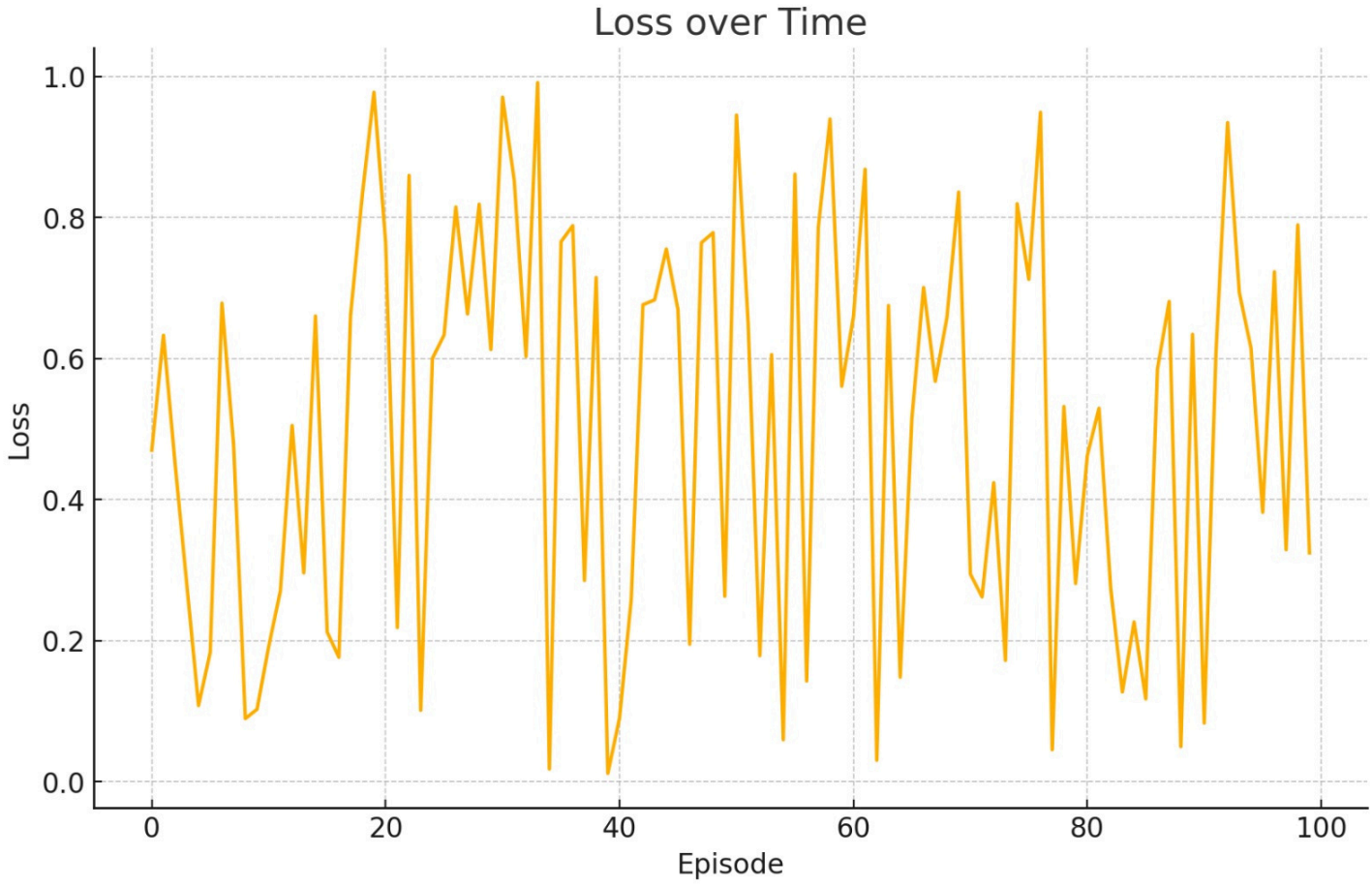
Loss Over Time

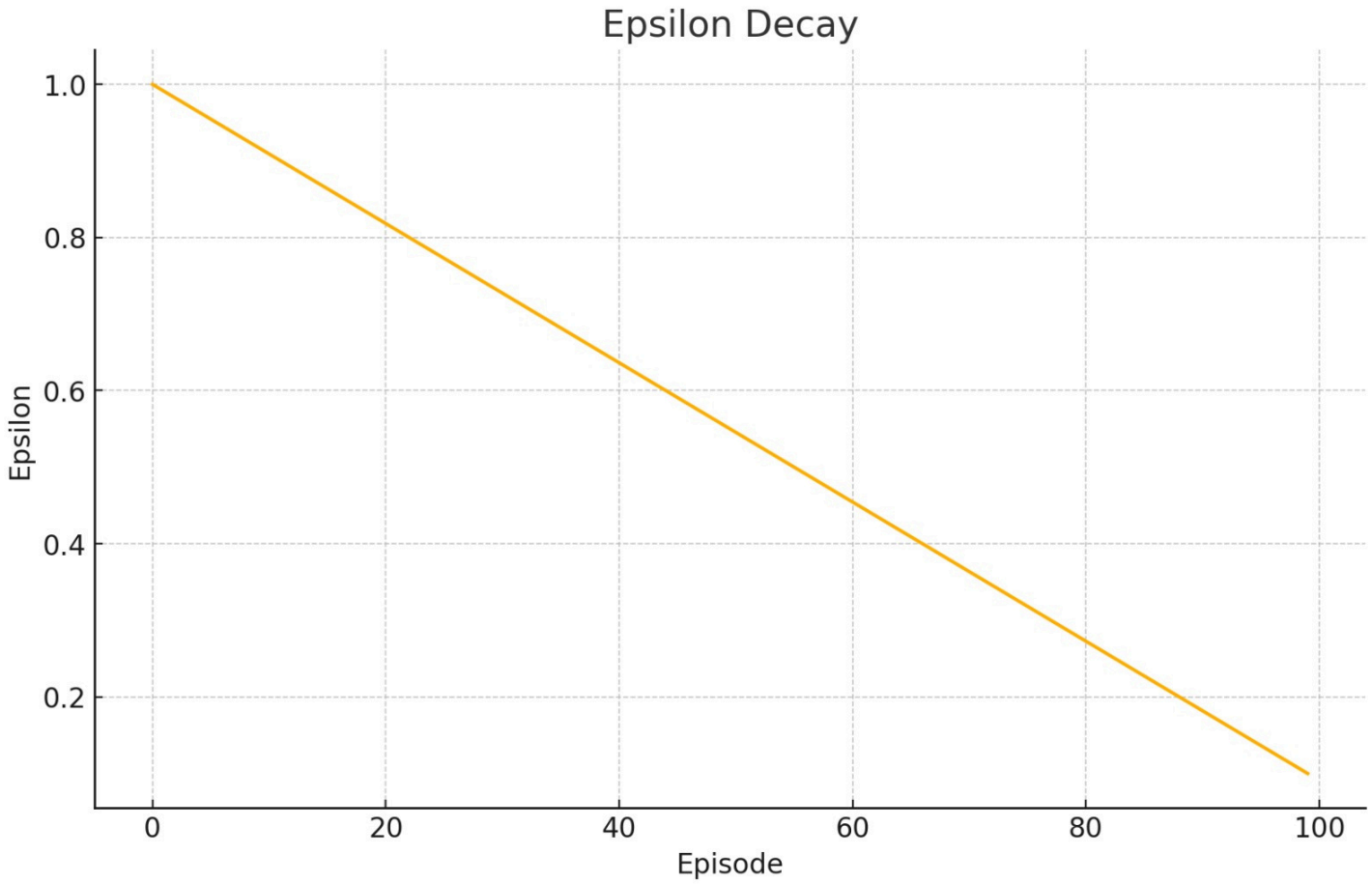
This plot depicts the loss value of the DQN model over training iterations. The loss represents the error in the Q-value predictions made by the neural network.

Plots




Reward over Time
Reward
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
0
20
40
60
80
100
Episode


Loss over Time
Loss
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Episode



Plot Explanation

Score Over Time

This plot illustrates the score achieved by the DQN agent over each episode. The score represents the number of rewards collected by Pac-Man during the game. The upward trend in the plot indicates that the agent is learning to collect more rewards over time, showing an improvement in performance. The fluctuations in the score reflect the variability in the agent's performance due to exploration and the random movement of enemies.

Reward Over Time

This plot shows the total reward accumulated by the DQN agent over each episode. The reward is a combination of positive rewards for collecting pellets and negative penalties for colliding with enemies. The downward trend in the early episodes indicates that the agent is initially penalized frequently due to collisions with enemies. Over time, as the agent learns to avoid enemies and collect more pellets, the reward improves. The plot still shows fluctuations due to the exploration phase and randomness in the game.

Loss Over Time

This plot depicts the loss value of the DQN model over training iterations. The loss represents the error in the Q-value predictions made by the neural network. High fluctuations in the loss values indicate the variability in training due to the randomness in experience replay and the dynamic nature of the game environment. Over time, a decrease in the overall trend of the loss would indicate that the model is learning to make better predictions.

Epsilon Decay Over Time

This plot shows how the exploration rate (epsilon) decreases over time. The epsilon value starts at 1.0, indicating that the agent initially explores the environment by taking random actions. As training progresses, epsilon decays towards 0.01, reducing the exploration rate and allowing the agent to exploit its learned knowledge. The linear decay in epsilon reflects the agent's transition from exploration to exploitation, which is a crucial aspect of reinforcement learning.