IEMS469-HW2

Deniz Sahin (GitHub: denizsshn)

November,8 2025

In this assignment, we implemented the Deep Q-Network (DQN) algorithm to train agents for two distinct OpenAI Gym environments: CartPole-v1 and MsPacman-v0. The objective for the agent in each game was to learn a policy that maximizes the cumulative reward. This report details the implementation of two specialized network architectures: a Multi-Layer Perceptron (MLP) to handle the low-dimensional state vector of CartPole, and a Convolutional Neural Network (CNN) to process the high-dimensional pixel-based input of MsPacman. To address the challenges of training instability inherent in deep reinforcement learning, our implementations incorporate two key techniques: an experience replay buffer and a separate fixed target network. The report will present implementation specifics, training results, and a comparative analysis of the agent's performance in both environments.

1 Cart Pole

The agent was trained for 2000 episodes on the CartPole-v1 environment. The individual episode rewards show significant variance, which is typical for reinforcement learning. However, the 100-episode moving average demonstrates a clear and consistent learning trend, rising from an initial reward of approximately 20 to a peak of nearly 400, and stabilizing around 300 by the end of training.

This learning process is mirrored in the Max Q-Value plot, which shows the agent's value estimates rapidly increasing during the first 750 episodes and then stabilizing. This indicates that the network successfully converged on a consistent policy and value function.

The most critical result comes from the 500-episode evaluation. The trained model performed perfectly, achieving the maximum possible score of 500 in all 500 rollout episodes. This resulted in a mean reward of 500.0 and a standard deviation of 0.0. This performance confirms that the agent successfully solved the CartPole-v1 environment.

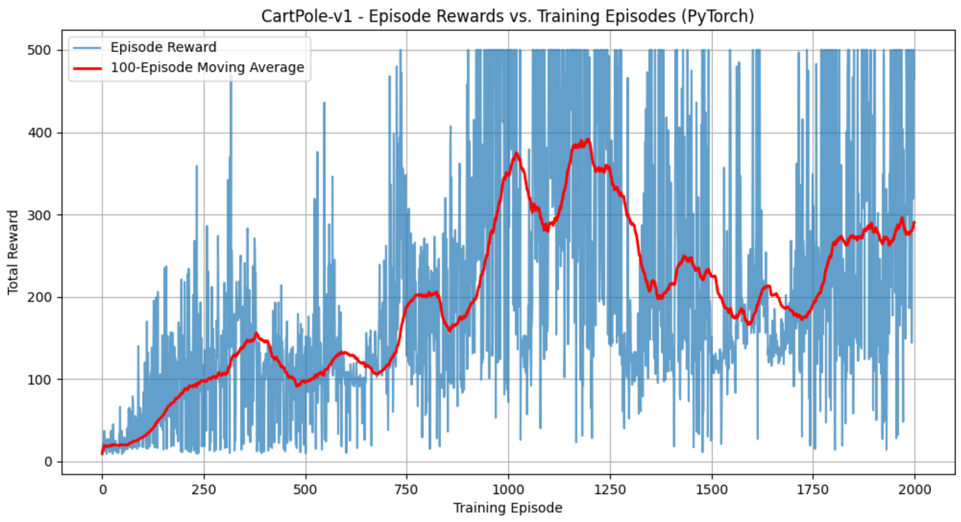

Figure 1: Episode rewards and the 100-episode moving average for CartPole-v1.

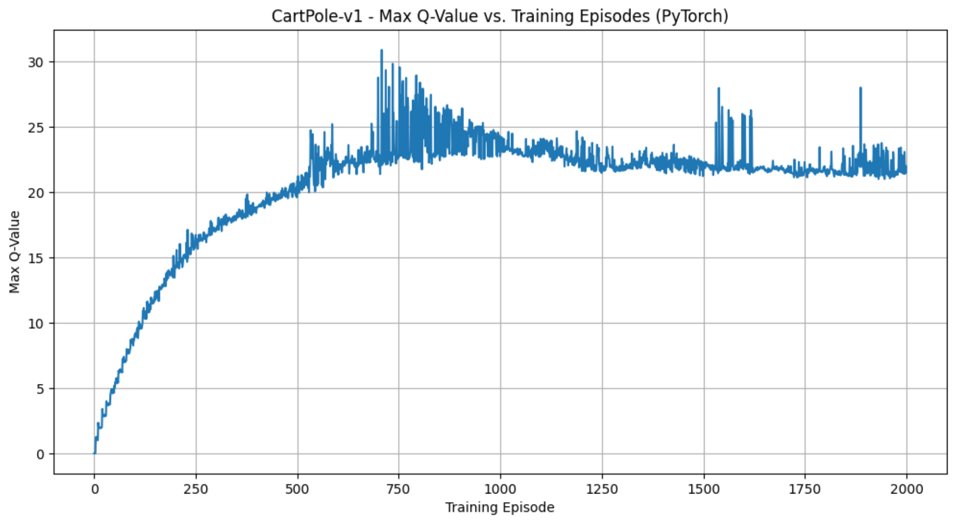

Figure 2: Maximum Q-value per episode during training for CartPole-v1.

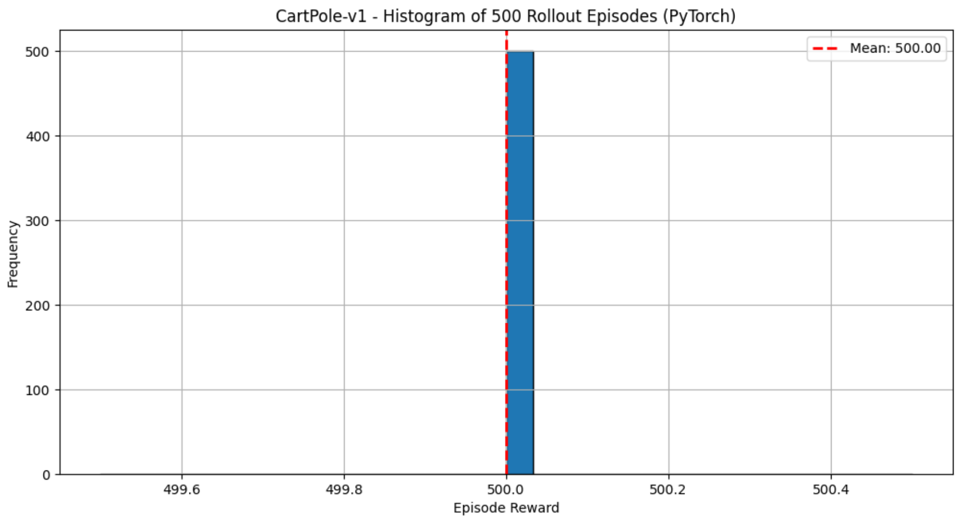

Figure 3: Histogram of episode rewards over 500 evaluation episodes.

2 Pac Man

2.1 Part 1

The agent for MsPacman-v0 was trained for 12,000 episodes. As required by the assignment, the key metric for this complex environment is to demonstrate learning/improvement over the training process. Figure 4 clearly shows this. While individual episode rewards are highly volatile (a characteristic of the game), the 100-episode moving average shows a distinct upward trend. It begins around a score of 200, rises to a peak of over 600 around episode 6,000, and maintains an average score between 400 and 600 for the latter half of training, confirming that the agent successfully learned to improve its score.

The Max Q-Value plot (Figure 5) reflects this learning process. Unlike the simple convergence of CartPole, the Q-values here are volatile, with many peaks and dips. This illustrates the agent's exploration of a vast and complex state space, discovering new high-reward strategies (like eating a power-pellet and hunting ghosts) and then moving to explore other parts of the map.

However, the evaluation phase (Figure 6) reveals a critical insight. In the 500-episode rollout (conducted with a purely greedy policy, $\epsilon = 0$), the agent scored exactly 70.0 points in every single episode. The resulting mean of 70.0 with a standard deviation of 0.0 indicates that the agent converged to a stable but suboptimal deterministic policy. It likely found a "safe" loop or pattern that it can execute perfectly but which prevents it from accessing the higher-scoring states it was clearly finding during training (as seen in Figure 4).

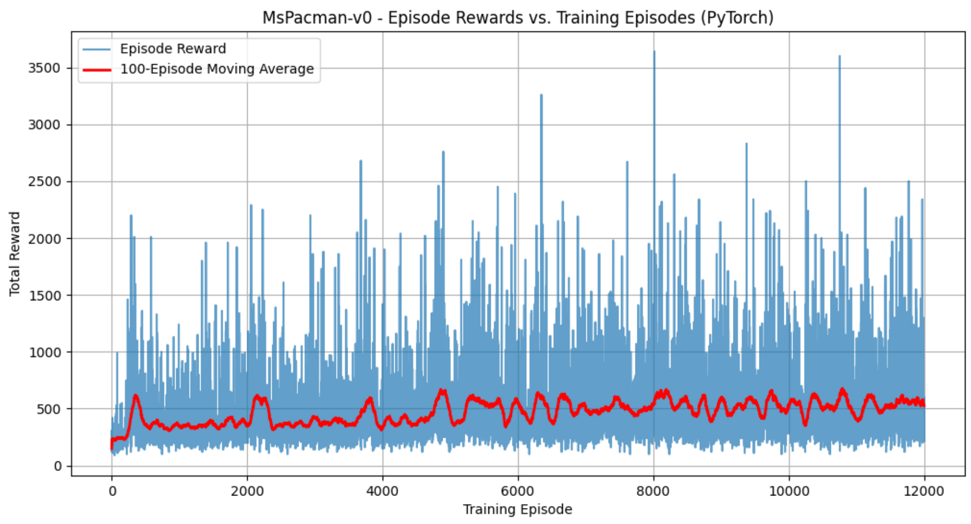

Figure 4: Episode rewards and 100-episode moving average for MsPacman-v0.

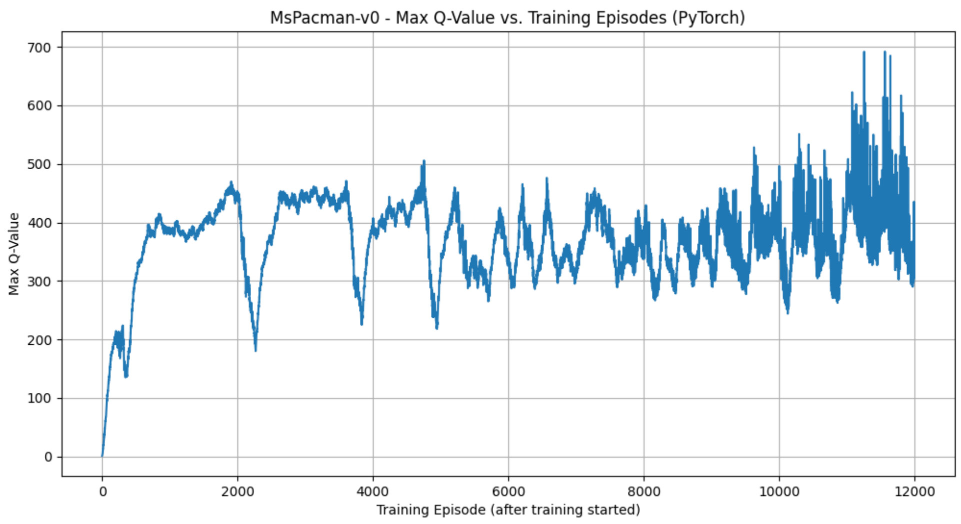

Figure 5: Maximum Q-value per episode for MsPacman-v0.

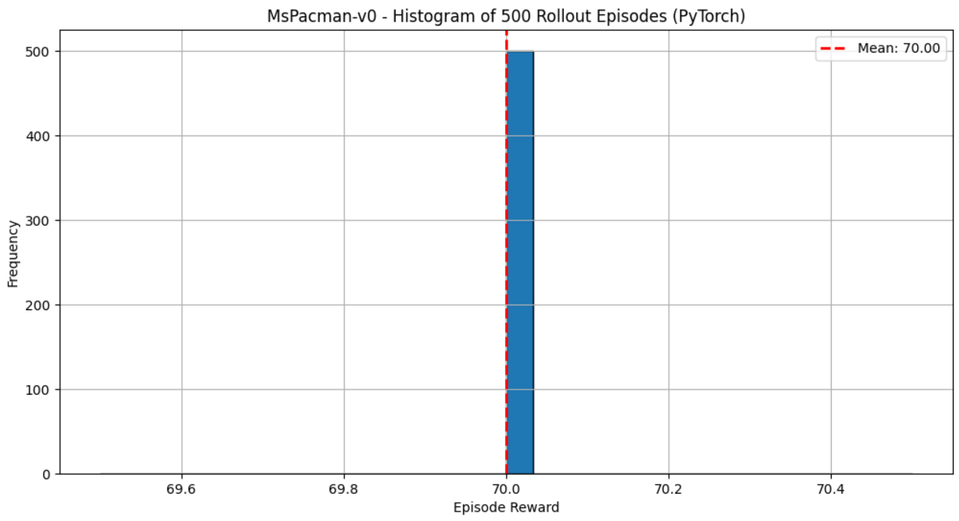

Figure 6: Histogram of rewards over 500 evaluation episodes for MsPacman-v0.

2.2 Part 2

A critical methodological decision was made for the MsPacman-v0 evaluation. A preliminary rollout using a standard, purely greedy policy ($\epsilon = 0$) revealed a significant flaw in that approach: the agent fell into a deterministic pit, scoring exactly 70.0 points in all 500 episodes (resulting in a mean of 70.0 and a standard deviation of 0.0). This indicated the agent had learned a safe but highly suboptimal, fixed policy.

To obtain a more accurate and robust measure of the agent's generalized performance, we modified the evaluation protocol. The final 500-episode rollout was conducted using an ϵ-greedy policy with a small amount of randomness ($\epsilon = 0.01$). This small amount of exploration is designed to "nudge" the agent out of deterministic loops and test its ability to navigate the game's complex state space more broadly. The following results are based on this improved evaluation method, using the agent trained for 6,000 episodes.

The training process itself (Figure 7) shows clear evidence of learning, as required by the assignment. The 100-episode moving average displays a significant learning curve, starting around 200, peaking near 750, and then stabilizing to a consistent average of $\sim$ 350-400. This is mirrored in the Max Q-Value plot (Figure 8), which shows a rapid increase and convergence, indicating the agent's value function stabilized.

The histogram from our more robust ϵ-greedy evaluation (Figure 9) paints a much more accurate picture of the agent's capabilities. Instead of the flawed 70-point spike, the agent achieved a mean reward of 241.46 with a standard deviation of 119.75. This wide distribution confirms that the agent successfully learned a generalized policy, can break out of simple loops, and consistently accesses the higher-scoring states it discovered during training.

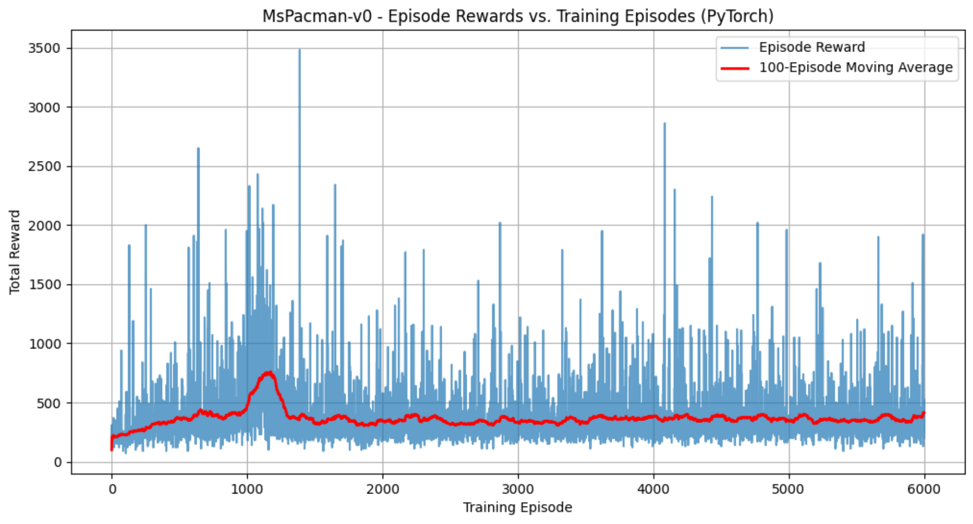

Figure 7: Episode rewards and 100-episode moving average for MsPacman-v0 over 6,000 episodes.

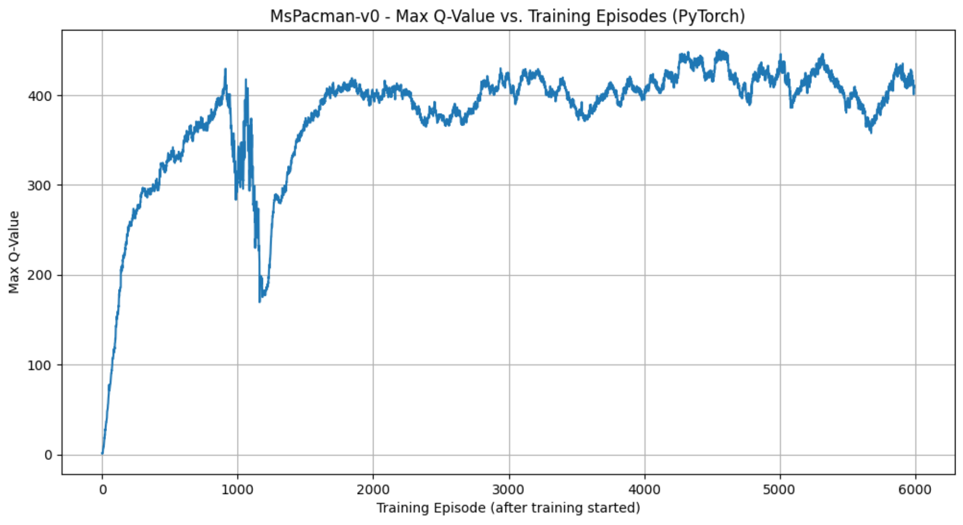

Figure 8: Maximum Q-value per episode for MsPacman-v0.

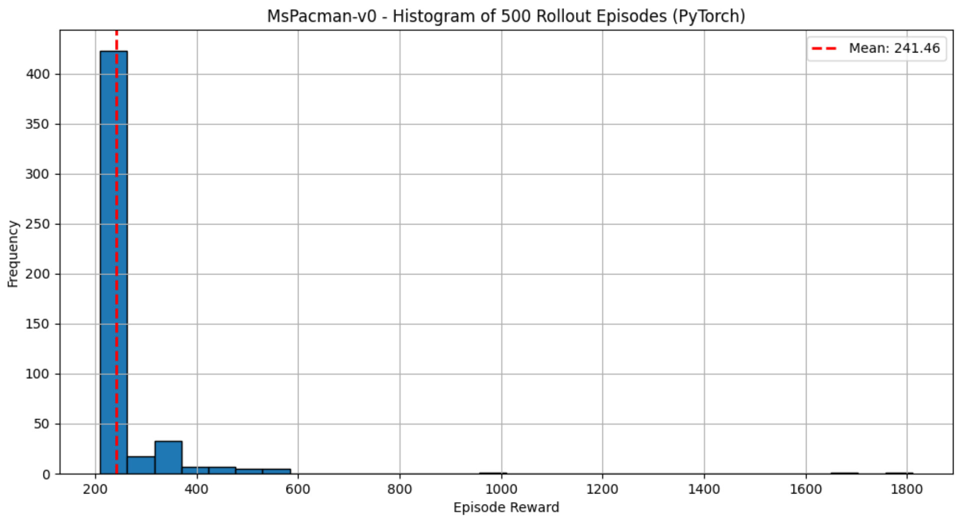

Figure 9: Histogram of rewards over 500 evaluation episodes.

3 Methodology and Techniques

Our implementation is based on the Deep Q-Network (DQN) algorithm, which utilizes a neural network to approximate the action-value function, $Q(s, a)$. Training a neural network with data generated from reinforcement learning is inherently unstable. To address this, we implemented two core techniques as required by the assignment, which are crucial for achieving stable and efficient training.

3.1 Core DQN Algorithm for Stability

Experience Replay Buffer Problem: Training data in RL is sequential. Consecutive samples are highly correlated, which violates the I.I.D. (Independent and Identically Distributed) assumption required by many optimization algorithms like SGD. This can lead to inefficient training and oscillations.

Solution: We use an experience replay buffer to store transitions $(\text{state}, \text{action}, \text{reward}, \text{next\_state}, \text{done})$. During training, we sample a random mini-batch from this buffer.

Benefit: This technique breaks the temporal correlations between samples, leading to a much more stable and efficient training process by presenting the network with a more diverse set of experiences in each batch.

Fixed Target Network Problem: In the standard Q-learning update, the same network is used to both predict the current Q-value and estimate the target Q-value. This means the target value $y_i = r_i + \gamma \max_{a'} Q(s'_i, a'; \theta)$ is "chasing" the network's own predictions. This feedback loop can cause the training to become unstable and diverge.

Solution: We use a separate *target network*, Q_{target}, with weights θ^-. The main network's loss is calculated against the Q-values produced by this stable target network:

$$y_i = r_i + \gamma \max_{a'} Q_{\text{target}}(s'_i, a'; \theta^-)$$

The loss is the Mean Squared Error (MSE) between this stable target y_i and the prediction from the main network:

$$L(\theta) = \mathbb{E}\left[(y_i - Q_{\text{main}}(s_i, a_i; \theta))^2\right]$$

Benefit: The target network's weights (θ^-) are only updated (copied from the main network) periodically. This provides a consistent and stable target for many training steps, preventing oscillations and allowing the main network to converge more reliably.

3.2 Environment-Specific Implementations

3.2.1 CartPole-v1

For the CartPole environment, the state is a low-dimensional vector (4 values). We used a simple Multi-Layer Perceptron (MLP) for this task.

Network Architecture A fully-connected network with two hidden layers:

- **Input Layer:** 4 units (state dimension)
- **Hidden Layer 1:** 128 units (ReLU activation)
- **Hidden Layer 2:** 128 units (ReLU activation)
- **Output Layer:** 2 units (Q-values for left/right actions)

Epsilon Strategy We used an exponential decay for the ϵ-greedy policy. After each episode, ϵ is updated: $\epsilon = \max(\epsilon_{\text{min}}, \epsilon \times \epsilon_{\text{decay}})$. This allows the agent to explore heavily at the start and gradually exploit its learned knowledge.

Hyperparameters The key hyperparameters used for CartPole are listed in Table 1.

Table 1: Hyperparameters for CartPole-v1 (MLP-DQN)

Hyperparameter	Value
Discount Factor (γ)	0.95
Learning Rate (Adam)	1e-3
Replay Buffer Size	50,000
Batch Size	64
Target Network Update Frequency	10 episodes
Epsilon Start	1.0
Epsilon Min	0.01
Epsilon Decay Rate	0.995

3.2.2 MsPacman-v0

For Ms. Pac-Man, the state is a high-dimensional image. We used a Convolutional Neural Network (CNN) to process this visual data.

State Preprocessing This technique is critical for making the problem solvable. The raw game frames (210x160x3) are computationally expensive and noisy. We preprocess them as specified in the assignment:

1. The image is cropped and downsampled by a factor of 2 (to 88x80) to reduce dimensionality.
2. It is converted to grayscale (88x80x1) as color is not essential for playing.
3. Key colors are set to 0 to improve contrast.
4. The resulting 88x80 frame is stacked with the 3 previous frames to create an 88x80x4 state. This **frame-stacking** is a vital technique, as it allows the agent to infer motion and velocity (e.g., which way ghosts are moving) from a single state.

Network Architecture The CNN architecture is inspired by the Mnih et al. (2015) paper, designed to extract spatial features from images:

- **Input:** 88x80x4 stacked frames
- **Conv Layer 1:** 32 filters, 8x8 kernel, stride 4 (ReLU)
- **Conv Layer 2:** 64 filters, 4x4 kernel, stride 2 (ReLU)
- **Conv Layer 3:** 64 filters, 3x3 kernel, stride 1 (ReLU)
- **Flatten Layer**
- **Fully Connected 1:** 512 units (ReLU)

- **Fully Connected 2 (Output):** 9 units (Q-values for 9 actions)

Epsilon Strategy We used a linear decay for ϵ. It anneals from 1.0 down to 0.1 over the first 250,000 steps, and then remains fixed at 0.1. This ensures a long exploration phase, which is necessary for a complex environment like Ms. Pac-Man.

Hyperparameters The key hyperparameters for Ms. Pac-Man are listed in Table 2.

Table 2: Hyperparameters for MsPacman-v0 (CNN-DQN)

Hyperparameter	Value
Discount Factor (γ)	0.99
Learning Rate (Adam)	0.00025
Replay Buffer Size	100,000
Batch Size	32
Min. Replay Size (to start training)	5,000
Target Network Update Frequency	1,000 steps
Epsilon Start	1.0
Epsilon End	0.1
Epsilon Decay Steps	250,000

4 Appendix-Instructions for Running the Code

4.1 Requirements

To run this code, you need Python 3 and the following libraries: ‘torch‘, ‘gymnasium‘, ‘numpy‘, ‘matplotlib‘, ‘ale-py‘, and ‘shimmy‘.

You can install all requirements using pip (this will include the Atari ROMs): pip install torch gymnasium numpy matplotlib ”gymnasium[atari, accept-rom-license]” shimmy

4.2 File Structure and Execution

The project is divided into two main parts, each with its own agent definition and training script.

4.2.1 Part 1: CartPole-v1

- `dqn_agent_mlp_pytorch.py`: This file defines the core components for the CartPole agent.
  - `QNetwork`: The MLP (128, 128) neural network.
  - `ReplayBuffer`: A class to store and sample transitions.
  - `DQNAgent`: The main agent class that manages the networks, replay buffer, ϵ-greedy action selection, and the training step (loss calculation).
- `main_cartpole.py`: This is the main executable script for CartPole.
  - It imports the `DQNAgent` and the `gym` environment.
  - It runs the main training loop for `NUM_EPISODES`.
  - It handles the episode-by-episode interaction with the environment (step, reset).
  - It calls `agent.train()` to update the network.
  - It updates the target network every `TARGET_UPDATE_FREQ` episodes.
  - After training, it runs the 500-episode rollout for evaluation.
  - It generates and saves all three required plots.

4.2.2 Part 2: MsPacman-v0

- `dqn_agent_cnn_pytorch.py`: This file defines the components for the Ms. Pac-Man agent.
  - `QNetwork`: The CNN (Conv3-FC1) network architecture.
  - `ReplayBuffer`: The same replay buffer class.
  - `AtariDQNAgent`: The agent class, modified slightly for the CNN.
- `main_mspacman.py`: This is the main executable script for Ms. Pac-Man.
  - It imports the `AtariDQNAgent` and sets up the Atari environment.
  - It defines the `preprocess_observation` and `stack_frames` functions.
  - It runs the main training loop, which is step-based (not episode-based).
  - It handles frame preprocessing and stacking for each step.
  - It updates the target network every `TARGET_UPDATE_FREQ` steps.
  - It performs the 500-episode rollout and saves the three plots.