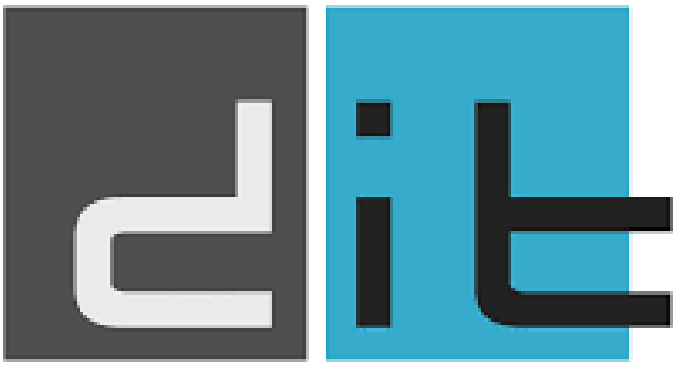

Reinforcement Learning Assignment
Winter Semester 2024-2025

Kostoulas Evangelos (7115112300017)
Kotsopoulos Dimitris (7115132300006)
Triantafillou Thanasis (7115132300010)

2 March 2025

Part 1: Selected Environment

Chosen Environment: MountainCarContinuous-v0

The **MountainCarContinuous-v0** environment is a deterministic Markov Decision Process (MDP) where an underpowered car must navigate a sinusoidal valley to reach a goal position at the top of the right hill. The agent must strategically apply continuous forces to achieve this goal, leveraging momentum effectively.

Action Space

- **Type:** Continuous, represented as a scalar value in the range $[-1.0, 1.0]$.
- **Description:** The action determines the directional force applied to the car:
  - -1.0: Maximum force to the left.
  - 1.0: Maximum force to the right.
- **Power Scaling and Clipping:** The applied force is first clipped to the range $[-1.0, 1.0]$ to ensure it stays within valid limits. It is then scaled by a constant factor of 0.0015, meaning the actual effect on the car's acceleration is relatively small. This controlled scaling prevents abrupt changes in velocity, ensuring smoother motion and making the environment dynamics more stable.

Observation Space

- **Type:** Continuous, 2-dimensional vector.
- **Shape:** $(2,)$, where each element represents a key aspect of the car's state.
- **Features:**
  - **Position** (x):
    - **Range:** $[-1.2, 0.6]$ meters.
    - Represents the car's horizontal location along the track. The left boundary (-1.2) corresponds to the leftmost valley, while the right boundary (0.6) is beyond the goal located at 0.45.
  - **Velocity** (v):
    - **Range:** $[-0.07, 0.07]$ meters per timestep.
    - Represents the rate of change of the car's position. Positive values indicate movement to the right, while negative values indicate movement to the left.
    - The velocity is influenced by both the applied force and the slope of the terrain, making it essential for building momentum to reach the goal.

Reward Signal

The environment balances energy efficiency and goal achievement using the following reward function:

$$\text{reward} = -0.1 \cdot (\text{action}^2) + 100 \cdot \mathbf{1}(\text{position} \geq 0.45)$$

$$\mathbf{1}(\text{position} \geq 0.45) = \begin{cases} 1, & \text{if position} \geq 0.45 \\ 0, & \text{otherwise} \end{cases}$$

- **Energy Efficiency Penalty:** A negative reward proportional to the square of the applied action discourages unnecessary force. This encourages the agent to find an optimal strategy that minimizes energy usage while still achieving the goal.
- **Goal Achievement Bonus:** A large positive reward of 100 is granted when the car reaches or surpasses the goal position ($x \geq 0.45$). This reward motivates the agent to reach the top of the hill as efficiently as possible.
- **Learning Strategy and Adaptation:** The reward structure encourages the agent to adopt an energy-efficient strategy rather than relying on brute force. Since excessive force is penalized, the agent must learn to build and sustain momentum by strategically utilizing the terrain's slopes. This interaction between control and environment dynamics directly influences how the agent adapts its policy to achieve the goal optimally.

Starting State

- **Initial Position:** The car's starting position is randomly initialized within the range $[-0.6, -0.4]$ meters, ensuring variation in initial conditions.
- **Initial Velocity:** The car always starts with zero velocity ($v = 0$), meaning it has no initial momentum and must build it through strategic force application.
- **Implications for Learning:** Since the initial position varies across episodes, the agent must develop a generalizable policy that works across different starting points.

Dynamics and Termination

- **Velocity Update:** The car's velocity is updated based on the applied action, a small power constant, and the effect of gravity. The update rule is:

  $$\text{velocity}_{t+1} = \text{velocity}_t + (\text{action} \cdot \text{power}) - 0.0025 \cdot \cos(3 \cdot \text{position}_t)$$

  where:

  - action $\cdot$ power applies the force selected by the agent, influencing acceleration.
  - The term $-0.0025 \cdot \cos(3 \cdot \text{position}_t)$ represents the gravitational effect, which varies depending on the car's position on the hill.

- **Position Update:** The car's position is updated based on its new velocity:

$$\text{position}_{t+1} = \text{position}_t + \text{velocity}_{t+1}$$

  This means the position is directly affected by the accumulated velocity, emphasizing the importance of momentum in reaching the goal.

- **Termination Conditions:** The episode ends when either of the following conditions is met:

  1. **Success:** The car reaches or surpasses the goal position ($x \geq 0.45$), indicating that it has successfully climbed the right hill.
  2. **Truncation:** The maximum timestep limit (999) is reached, preventing excessively long episodes.

  These conditions ensure that the agent must learn an efficient strategy to reach the goal within a limited number of steps.

Gymnasium Environment Setup

Gymnasium provides a simple and standardized API for interacting with reinforcement learning environments. It simplifies the process of defining, testing, and training agents by offering a structured framework for environment interaction. Below is a minimal setup for using the `MountainCarContinuous-v0` environment, where an agent must learn to navigate a valley using continuous control.

Initialization

To begin, the environment must be initialized using the `gym.make` function. This creates an instance of the environment where the agent can interact and learn. The `render_mode`

Listing 1: Initializing the Environment

```
import gymnasium as gym
env = gym.make("MountainCarContinuous-v0", render_mode="rgb_array",
    goal_velocity=0.1)
```

parameter allows visualization of the environment, while `goal_velocity` sets a specific velocity requirement for successfully completing the task.

Reset Environment

Before starting an episode, the environment must be reset. This ensures that the car is placed at a random starting position within a predefined range, and all necessary variables are initialized. The optional `seed` ensures reproducibility, while the `options` parameter

Listing 2: Resetting the Environment

```
state, _ = env.reset(seed=123, options={"low": -0.7, "high": -0.5})
```

customizes the starting position range.

Sample and Apply Action

At each timestep, the agent selects an action to apply. The action space in this environment is continuous, meaning the agent must choose a value within the range $[-1.0, 1.0]$. Below, we demonstrate how to sample a random action and apply it to the environment. The

Listing 3: Sampling and Applying an Action

```
action = env.action_space.sample()
next_state, reward, done, truncated, info = env.step(action)
```

`step` function takes an action as input and returns:

- `next_state`: The updated state of the environment.
- `reward`: The immediate reward for the chosen action.
- `done`: A boolean flag indicating whether the episode has ended.
- `truncated`: A flag that signals whether the maximum timestep limit has been reached.
- `info`: Additional debugging information.

This minimal setup allows the agent to interact with the environment, making it the foundation for training reinforcement learning models.

Customization for Algorithm Implementation

The flexibility of Gymnasium's API enabled us to adapt the `MountainCarContinuous-v0` environment to our reinforcement learning framework. By leveraging environment wrappers, we adjusted observations, rewards, and action processing to align with our algorithm's requirements. These modifications ensured a seamless integration between the environment and our learning models, allowing for efficient experimentation and evaluation.

Gymnasium provides various wrapper classes to facilitate environment customization:

- **Action Wrappers**: Modify or preprocess actions before passing them to the environment. More details can be found in the official documentation.
- **Observation Wrappers**: Transform or preprocess observations before presenting them to the agent. See the documentation for additional information.
- **Reward Wrappers**: Adjust or reshape rewards to suit the learning objectives. Refer to the Gymnasium API for details.

By utilizing these wrappers, we enhanced the training efficiency and adaptability of our algorithm while maintaining compatibility with Gymnasium's standardized reinforcement learning interface.

Part 2: Implementation of DQN Algorithm

This section provides a detailed overview of the implementation of the Deep Q-Learning (DQN) algorithm to train an agent in the `MountainCarContinuous-v0` environment. The implementation follows the canonical DQN flow, enhanced with experience replay and a target network for improved stability.

Class-Based DQN Implementation

The Deep Q-Network (DQN) was implemented using a modular, object-oriented approach to enhance maintainability and flexibility. The key components of the implementation include:

- **Replay Memory**: A fixed-capacity buffer that stores past transitions and enables mini-batch sampling to stabilize training.
- **Q-Network**: A fully connected neural network that approximates Q-values for discrete actions based on observed states.
- **DQN Agent**: Responsible for action selection, experience replay, policy updates, and target network synchronization.
- **DQN Trainer**: Manages the training and evaluation pipeline, including interaction with the environment, training loops, and model checkpointing.
- **Custom Environment Wrappers**: Modify the environment's behavior to better suit the requirements of the DQN algorithm.

Environment Wrappers

Since the `MountainCarContinuous-v0` environment has a continuous action space, several modifications were necessary to adapt it for discrete control using DQN. To achieve this, we implemented a set of environment wrappers that preprocess observations, reshape rewards, and discretize the action space.

1. Reward Wrapper

The `reward_wrapper` implements the standard reward function used in the chosen environment. Although the default reward structure remains intact, the wrapper was primarily introduced to allow modifications to the reward function for experimental purposes. Specifically, it facilitates adjustments aimed at accelerating convergence by testing different reward shaping strategies. This flexibility enables the evaluation of how modified incentives influence the agent's learning efficiency and policy optimization.

2. Step Wrapper

The `step_wrapper` acts as an intermediary between the agent and the environment, modifying the observations and rewards before they are passed to the learning model. It integrates the `observation_wrapper` and `reward_wrapper`, ensuring that each interaction with the environment aligns with the agent's learning objectives.

3. Observation Wrapper

The `observation_wrapper` normalizes observations using min-max scaling to ensure they fall within a standardized range. This transformation improves stability during training by preventing large numerical variations that could impact gradient updates.

4. Discretized Action Wrapper

Since DQN requires a discrete action space, the `DiscretizedActionWrapper` transforms the environment's continuous action space into a finite set of discrete actions. This is accomplished by:

- Dividing the original action space into a fixed number of bins.
- Mapping each discrete action index to a corresponding continuous action value.
- Ensuring compatibility with DQN's discrete action selection process.

By discretizing the action space, the agent can interact with the environment using standard Q-learning techniques while still leveraging the advantages of a continuous control problem.

Training and Evaluation Pipeline [2]

1. Environment and Agent Initialization

The Gymnasium environment is initialized with custom wrappers applied to modify observations, rewards, and action spaces. The agent is instantiated with:

- **Replay Memory**: A buffer for storing past transitions to enable experience replay.
- **Q-Networks**: Two neural networks are initialized:
  - The **main network** (`main_network`) is used for action selection.
  - The **target network** (`target_network`) stabilizes training by providing fixed Q-value targets.
- **Xavier Initialization**: Applied to all network layers to ensure stable gradient updates.

2. Hyperparameters

The training process begins with a set of initially chosen hyperparameters:

- **Learning rate** (α): 0.01
- **Discount factor** (γ): 0.99

- **Replay memory capacity**: $250,000$
- **Batch size**: 128
- **Epsilon-greedy exploration parameters**:
  - Maximum epsilon: 0.999
  - Minimum epsilon: 0.01
  - Epsilon decay: 0.998
- **Target network update frequency**: Every 50 steps.
- **Gradient clipping**: Maximum norm of 5.
- **Random Seed**: A fixed seed (`seed=0`) was applied across NumPy, Python, and PyTorch to ensure reproducibility in environment initialization, experience replay, and model training.

3. Training Procedure

The agent learns by interacting with the environment and updating its policy using experience replay and Q-learning updates. DQN introduces several improvements over standard Q-learning, such as experience replay and the use of a target network, to enhance stability and efficiency.

1. **Environment Reset and Initial State:** The environment is reset, and the agent starts in a randomly assigned initial state. This ensures that training explores different initial conditions, promoting generalization.
2. **Action Selection:** At each step, the agent selects an action using an **epsilon-greedy policy**, which balances exploration and exploitation:

$$a_t = \begin{cases} \text{Random action} & \text{with probability } \epsilon, \\ \arg\max_a Q(s_t, a; \theta) & \text{otherwise.} \end{cases}$$

   - **Exploration**: Selecting random actions allows the agent to discover new strategies and avoid local optima.
   - **Exploitation**: Choosing the action with the highest Q-value enables the agent to leverage learned knowledge.
   - **Epsilon Decay**: The exploration rate (ϵ) gradually decreases over time, ensuring that the agent explores initially but exploits more as training progresses.
3. **Experience Replay and Memory Storage:** After executing the selected action, the environment returns the next state, reward, and termination flag. The transition (s_t, a_t, r_t, s_{t+1}) is stored in the replay memory.
   - **Breaking Correlations**: Unlike standard Q-learning, where updates are performed sequentially, experience replay allows sampling of past transitions to reduce correlation between consecutive experiences.
   - **Efficient Learning**: The replay buffer enables the reuse of past experiences, improving sample efficiency and stability.

4. **Q-Value Target Computation:** The Q-value target is computed using the Bellman equation:

$$y_j = r_j + \gamma \max_{a'} Q_{\text{target}}(s_{j+1}, a'; \theta^-).$$

   - **Temporal Difference Learning**: The target incorporates future rewards using the discount factor γ, ensuring long-term reward maximization.
   - **Target Network Usage**: The Q-value target is computed using the `target_network`, which remains fixed for a period of time to improve training stability and prevent feedback loops.

5. **Network Optimization:** The main network's parameters are updated by minimizing the mean squared error (MSE) loss:

$$\text{Loss} = \frac{1}{N} \sum_j \left(y_j - Q(s_j, a_j; \theta)\right)^2.$$

   - **Gradient Descent Optimization**: The loss function measures the difference between predicted Q-values and target Q-values. The optimizer updates network weights to minimize this difference.
   - **Gradient Clipping**: To prevent exploding gradients, updates are constrained to a maximum norm.

6. **Target Network Update:** Every fixed number of steps, the target network is updated by copying parameters from the main network.

   - **Stabilizing Training**: Without a target network, continuously updating Q-values using the same network can lead to instability. The target network provides a more stable reference for training.
   - **Periodic Updates**: Instead of updating at every step, the target network is updated at fixed intervals (e.g., every 50 steps), reducing variance in learning.

4. Evaluation and Testing

After training, the agent's performance is assessed using stored model checkpoints:

- The trained model is tested on multiple episodes to evaluate generalization.
- Episode rewards and success rates are logged to analyze agent performance.
- The learned policy is visualized to assess its effectiveness in solving the task.

5. Results and Visualization

During training, the following performance metrics were tracked:

- **Rewards**: Episode rewards were recorded and smoothed using a Simple Moving Average (SMA) with a window size of 50 episodes.
- **Network Loss**: The loss per training step was monitored to ensure convergence.

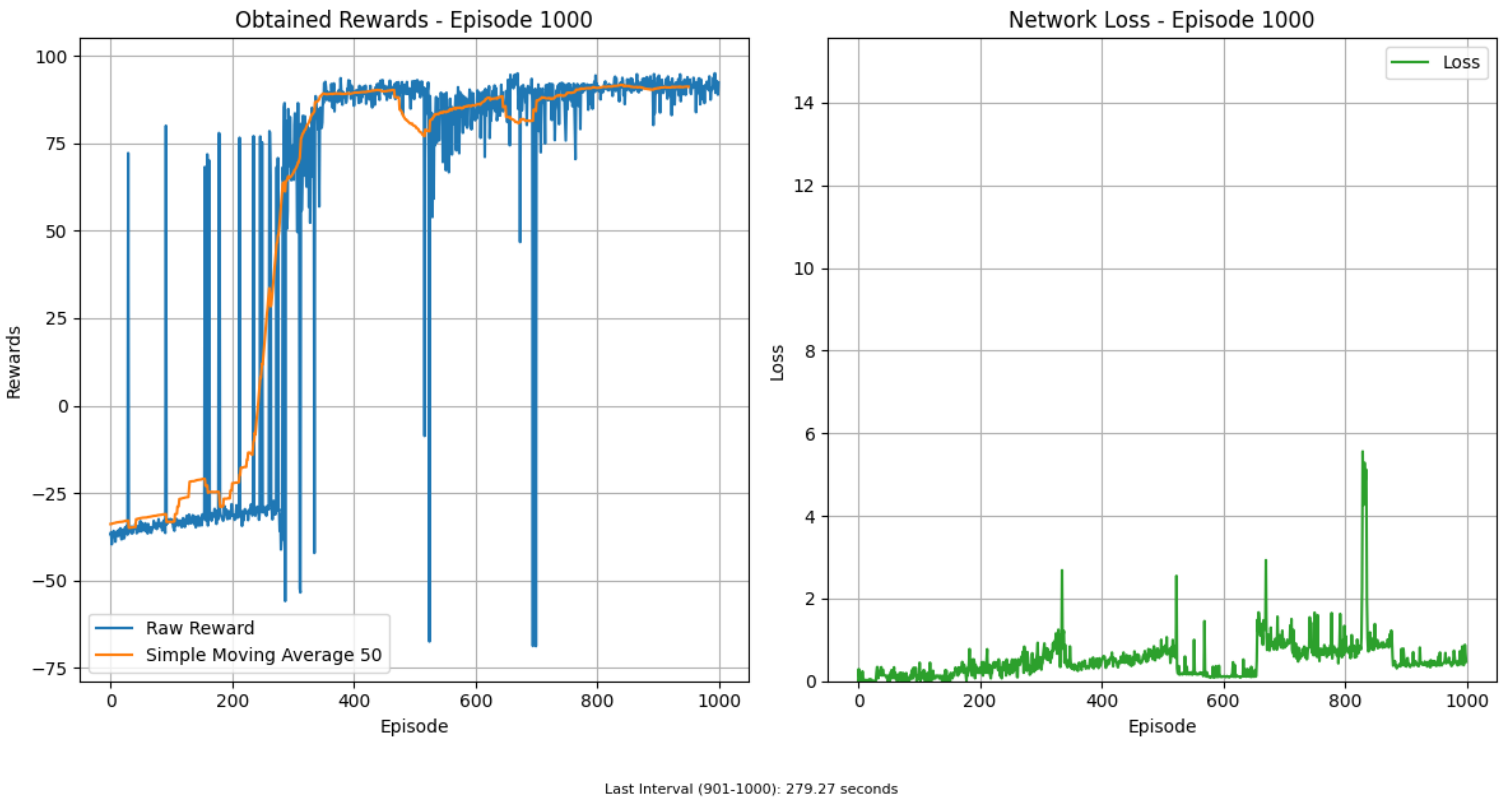

Figure 1: Training performance of DQN on MountainCarContinuous: Episode rewards (left) and network loss (right).

Analysis of DQN Performance in MountainCarContinuous

The results demonstrate the effectiveness of Deep Q-Network (DQN) in learning an optimal policy for the `MountainCarContinuous-v0` environment.

Reward Progression: The left plot illustrates the agent's accumulated rewards per episode. Initially, the agent struggles to gain positive rewards due to poor policy selection and exploration. However, as training progresses, rewards increase significantly, indicating successful learning of the optimal momentum-based strategy required to reach the goal. The Simple Moving Average (SMA) smooths fluctuations and highlights the steady improvement in performance. The occasional reward drops suggest that the agent still occasionally fails to reach the goal or miscalculates force application, but overall, there is clear convergence.

Loss Behavior: The right plot shows the loss evolution during training. Initially, the loss is relatively low due to the limited learning of useful state-action mappings. As the agent explores more states and updates its Q-values, loss increases due to higher variability in the learned value function. The occasional spikes in loss could be attributed to drastic policy updates or instability in Q-value estimates, which is a known limitation of DQN. However, the overall downward trend suggests convergence toward a stable policy.

Impact of StepLR: The implementation of StepLR introduces scheduled reductions in the learning rate, allowing for a more gradual adjustment of weight updates. Early in training, a higher learning rate facilitates faster exploration and learning. As the training progresses, StepLR reduces the learning rate at predefined intervals, stabilizing updates and reducing fluctuations in Q-values. This approach helps prevent drastic policy shifts and mitigates instability in training. While the overall performance remains similar to the non-StepLR version, the controlled decay in learning rate does not provide a significant

advantage in this case. Given the comparable results, we will continue using the non-StepLR approach.

Observations: DQN faces challenges when applied to continuous control environments due to the discrete nature of action selection. The use of the `DiscretizedActionWrapper` was crucial in transforming the action space into discrete bins, making the environment compatible with Q-learning. While this approach enables learning, it also introduces some limitations, such as the inability to fully exploit fine-grained control. A potential improvement could be using a policy-based method like Deep Deterministic Policy Gradient (DDPG), which directly handles continuous action spaces.

Final Performance: By the end of training, the agent consistently achieves high rewards, indicating successful learning of the environment dynamics. The learned policy enables the agent to build sufficient momentum to overcome the valley and reach the goal efficiently.

Overall, the results highlight the capability of DQN to solve the `MountainCarContinuous-v0` problem, despite the challenges posed by discretization. While improvements could be made using alternative reinforcement learning algorithms, the current implementation successfully demonstrates the effectiveness of deep Q-learning in a continuous control setting.

Part 3: Sensitivity Analysis

1. Methodology

A sensitivity analysis was conducted to evaluate the impact of key hyperparameters on the performance of the Deep Q-Network (DQN) algorithm. The primary objective was to assess how these parameters influence learning efficiency, stability, and overall convergence. The analysis was performed by systematically varying each parameter while keeping other configurations constant to isolate its specific impact.

The primary evaluation metric was the **average episode reward**, which reflects the cumulative performance of the agent over time. Additional observations were made regarding **training stability**, measured through reward variance, and **convergence speed**, assessed based on the number of episodes required to achieve a stable reward.

2. Selected Parameters

The following hyperparameters were analyzed, as they play a crucial role in determining the efficiency and effectiveness of DQN training:

- **Learning Rate (α)**: Controls how quickly the model updates its Q-values. A well-tuned learning rate balances rapid learning with stability.
- **Exploration Rate Decay (ϵ_{decay})**: Defines how fast the agent transitions from exploratory behavior to exploitative decision-making.
- **Batch Size**: Specifies the number of experiences sampled from the replay buffer per training update, influencing gradient estimation stability.
- **Number of Discretized Action Bins**: Determines the resolution of the discretized action space, affecting the precision of control actions.
- **Hidden Layer Dimensions**: Defines the network's capacity to approximate the Q-function and capture complex patterns in state-action mappings.
- **Different Seeds**: Evaluates the impact of varying random seeds on the consistency and reproducibility of training outcomes.
- **Optimized Model with Modified Reward Function**: Examines the effect of customizing reward structures to encourage better learning behavior and more desirable policy development.

Additionally, experiments with **two hidden layers** and **sparse neural networks** were conducted, but these architectures did not yield significant improvements in performance compared to the single-layer setup.

3. Results and Observations

This sensitivity analysis highlights the impact of key hyperparameters on the training dynamics of a DQN agent. The results demonstrate how adjustments in learning rate, exploration strategy, batch size, network architecture, and reward function design influence stability, convergence speed, and overall policy performance.

- **Learning Rate (α)**: - A very low learning rate (e.g., 0.001) resulted in slow convergence, requiring significantly more training episodes to reach optimal performance. - A high learning rate (e.g., 0.1 or 0.5) led to unstable updates, with large fluctuations in rewards and failure to converge to a stable policy. - The optimal trade-off was observed at $\alpha = 0.01$, which provided consistent learning progress without excessive oscillations.

- **Exploration Rate Decay (ϵ_{decay})**: - A very slow decay (e.g., 0.999) prolonged exploration, leading to more diverse experience collection but delaying exploitation of learned policies. - A fast decay (e.g., 0.995) resulted in early exploitation, often converging to suboptimal policies due to insufficient exploration. - The best balance was achieved at $\epsilon_{\text{decay}} = 0.998$, allowing adequate exploration while transitioning to stable policy learning at the right pace.

- **Batch Size**: - Small batch sizes (e.g., 32) caused high variance in training updates, leading to instability and slower convergence. - Very large batch sizes (e.g., 256) resulted in smoother updates but increased computational time and reduced the diversity of experiences learned in each update step. - The most stable learning occurred with batch sizes between 64 and 128, where variance was minimized without excessive computational overhead.

- **Number of Discretized Action Bins**: - A low number of bins (e.g., 10) severely limited the policy's flexibility, leading to suboptimal decision-making. - Increasing bins (e.g., 50 to 100) enhanced control precision but required more training data and increased computational complexity. - A mid-range bin setting (e.g., 20 to 50) provided a good trade-off between action resolution and training feasibility.

- **Hidden Layer Dimensions**: - Networks with fewer neurons (e.g., 32) lacked the capacity to approximate complex Q-values accurately, leading to inefficient learning. - Large networks (e.g., 256) improved function approximation but introduced instability and required additional regularization techniques to prevent overfitting. - Hidden layer sizes of 64 or 128 provided the best results in terms of stability and convergence efficiency.

- **Different Seeds**: - Training results varied significantly depending on the seed value. - Some seeds led to early convergence and stable policies, while others resulted in prolonged exploration phases with higher variance. - The impact of random seed variability highlights the importance of running multiple trials to ensure robust policy evaluation.

- **Optimized Model with Modified Reward Function**: - Adjusting the reward function had a major impact on learning behavior. - A well-designed reward structure encouraged the agent to explore more meaningful policies and achieve faster convergence. - However, poorly tuned reward modifications led to overfitting specific behaviors, making the learned policy less generalizable.

Summary of Findings:

- **Learning rate and exploration decay** significantly affect stability and convergence. A balanced approach prevents both slow learning and erratic updates.
- **Batch size and hidden layer dimensions** influence training efficiency. Mid-range values provide stable learning while avoiding excessive computation costs.
- **Action discretization granularity** should match the problem complexity. Too few bins limit the model's ability to learn fine-grained actions, whereas too many increase the training burden.
- **Random seed effects** highlight the stochastic nature of training, emphasizing the need for multiple runs to evaluate policy robustness.
- **Reward function tuning** is a powerful but delicate optimization tool. Properly shaping rewards can accelerate learning, but excessive modifications can lead to unintended policy behaviors.

The overall findings of this sensitivity analysis underscore the importance of carefully tuning hyperparameters to achieve stable and efficient reinforcement learning. While deeper network architectures and alternative reward functions introduced some variability, the best performance was achieved with a well-optimized single-layer network. The results further highlight that structured sensitivity testing is essential in reinforcement learning to develop robust, high-performing policies.

4. Optimal Model Configuration

Based on the sensitivity analysis and final evaluations, the following configuration yielded the best performance:

Hyperparameter	**Value**
Learning Rate (α)	0.01
Discount Factor (γ)	0.99
Batch Size	128
Target Update Frequency	50 steps
Max Episodes	1000
Max Steps per Episode	1000
Epsilon (ϵ) Decay	0.998
Epsilon Range	[0.999, 0.01]
Memory Capacity	250,000
Hidden Layer Dimension	128
Action Bins	20
Random Seed	1234

Table 1: Final optimized hyperparameters for DQN training.

This configuration achieved a well-balanced trade-off between learning speed, stability, and final performance. It serves as a robust baseline for reinforcement learning tasks in similar environments.

5. Visualization of Results

To better understand the effect of different hyperparameter configurations and reward modifications, we visualize the training performance of selected models. Each figure presents the obtained rewards per episode (left) and the network loss progression (right). These visualizations highlight the stability, convergence, and overall effectiveness of different configurations.

5.1 Optimized Model Performance

The optimized model, trained with a carefully tuned set of hyperparameters, achieved the best trade-off between learning efficiency and stability. The training curve demonstrates a steady increase in rewards, converging to a near-optimal policy with minimal fluctuations. The network loss follows a controlled descent, indicating stable learning behavior.

Compared to the baseline model, which exhibited slower learning and higher variance in rewards, the optimized configuration led to more stable training. The network loss in the baseline fluctuated aggressively, suggesting instability in weight updates. These improvements highlight the significance of hyperparameter tuning in reinforcement learning.

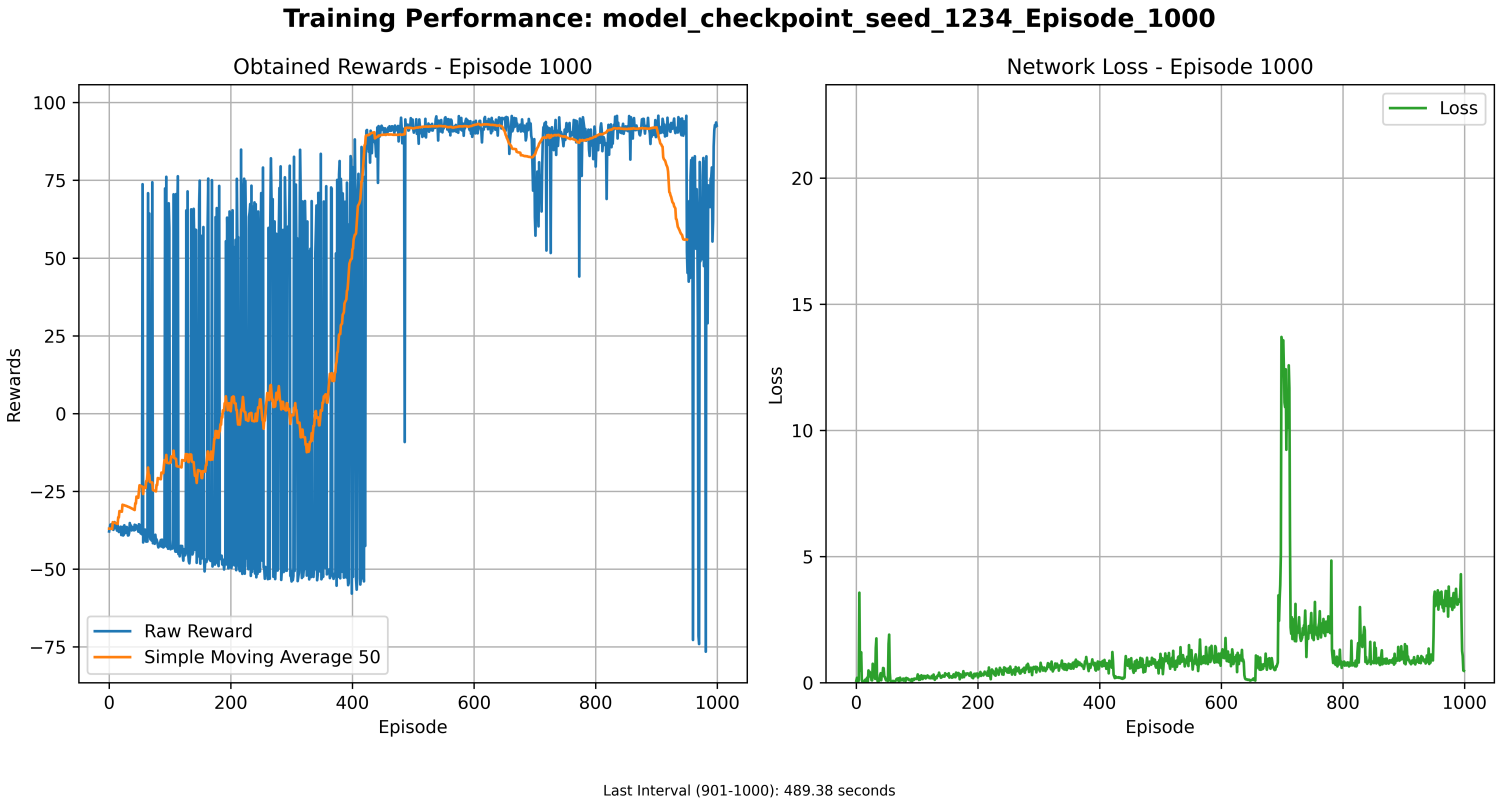

Figure 2: Training performance of the optimized DQN model: Episode rewards (left) and network loss (right).

5.2 Performance with Modified Reward Function

5.2.1 Overview of Reward Functions

To further investigate the impact of reward structure on learning dynamics, we tested multiple optimized reward functions designed to encourage more effective exploration and

faster convergence. The following reward functions were analyzed:

- **Basic Progress Incentive**: Rewards forward movement while penalizing large action magnitudes.
- **Goal-Oriented Bonus**: Provides significant rewards upon reaching the goal and favors positive velocity.
- **Progressive Scaling**: Introduces incremental bonuses as the agent nears the goal.
- **Exponential Progression**: Uses an exponential scaling factor to amplify progress rewards.
- **Balanced Velocity Scaling**: Adjusts rewards dynamically using logarithmic scaling to balance velocity incentives.
- **Cosine-Based Rewarding**: Implements oscillatory rewards to reinforce periodic movement patterns.

5.2.2 Analysis of Progressive Bonus Reward Function

Each function was evaluated using the same hyperparameters to ensure a fair comparison. The **Progressive Bonus Reward Function** was analyzed in detail, and its results are summarized below.

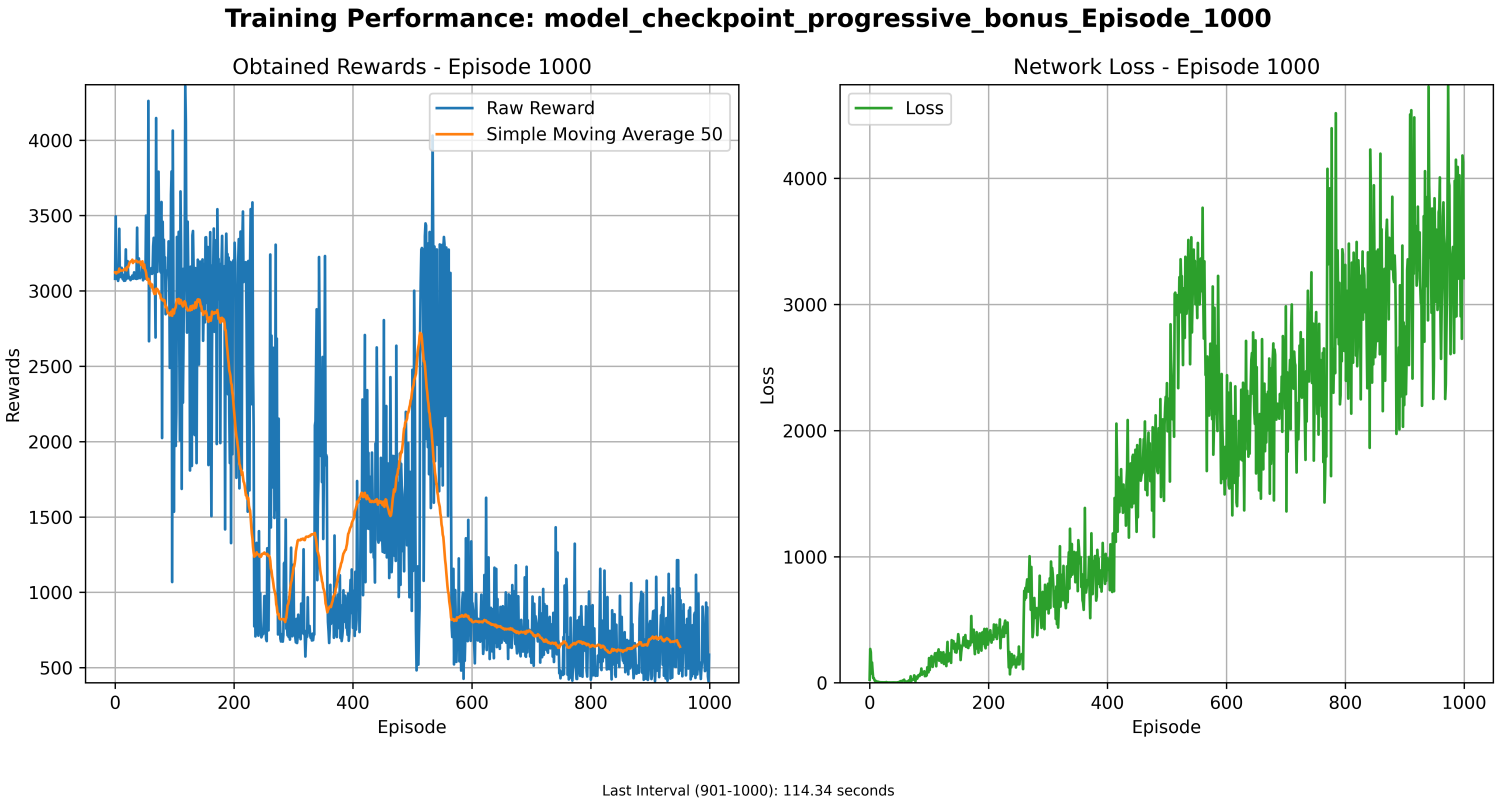

Figure 3: Training performance with the progressive bonus reward function: Episode rewards (left) and network loss (right).

Observations:

- The initial high reward values are due to the reward function's structure, which incentivizes movement toward the goal.
- As training progresses, the agent reaches the goal faster, leading to lower cumulative rewards per episode.
- The function rewards **position and velocity**, but since reaching the goal in fewer steps results in a lower total reward, later episodes exhibit lower reward values.
- The number of steps per episode decreases over training, reflecting improved efficiency in reaching the goal.

Loss Analysis:

- The loss increases over time, indicating potential instability in value function approximation.
- The rising loss suggests the agent is experiencing rapid policy changes, possibly due to aggressive reward scaling.
- Despite the loss increase, the model successfully learns an efficient policy by minimizing the number of steps required to reach the goal.

Summary of Results:

- The **Progressive Bonus Reward Function** encourages efficient goal-reaching behavior but results in lower cumulative rewards as episodes become shorter.
- The increasing loss suggests potential instability, which may require additional regularization or learning rate adjustments.
- Future improvements could involve **adjusting reward scaling** to maintain a more stable reward signal while preserving the incentive for quick goal attainment.

6. Conclusion

This sensitivity analysis highlights the crucial role of hyperparameter selection, network architecture, and reward function design in optimizing the training dynamics of a DQN agent. Fine-tuning parameters such as the learning rate, batch size, and exploration decay proved essential for achieving a balance between learning efficiency and stability, while action space discretization and network structure influenced the model's ability to generalize effectively. The analysis of different reward functions demonstrated that well-designed incentives accelerate learning and policy refinement, whereas poorly tuned rewards can introduce instability. Additionally, the effect of varying random seeds emphasized the inherent stochasticity of training, reinforcing the need for multiple runs to ensure robustness and reproducibility. Overall, the findings underscore the importance of structured sensitivity testing in reinforcement learning to develop stable, high-performing models.

Part 4: Advanced DQN Improvements

Implemented Improvements

Deep Q-Networks (DQN) have been significantly improved over time with various extensions that address its limitations, such as instability, slow learning, and inefficient exploration. In this section, we explore several advanced techniques that enhance the learning process of DQN. These modifications aim to improve convergence speed, reduce overestimation errors, and make exploration more effective. The implemented techniques are:

- **Prioritized Experience Replay (PER)**: Focuses training on more meaningful experiences.
- **Dueling DQN Architecture**: Separates state value estimation from action advantages for better decision-making.
- **Double DQN (DDQN)**: Reduces the overestimation bias in Q-value updates.
- **Noisy Networks for Exploration**: Introduces parameterized noise into the network for more efficient exploration.
- **Multi-Step Learning**: Uses longer-term rewards for more effective training.
- **Distributional RL (Categorical DQN - C51)**: Predicts reward distributions instead of single values.
- **Rainbow DQN**: Combines multiple techniques into a single powerful model.

Class Modifications: To implement these advanced techniques, modifications were applied through **class inheritance** in the following DQN components:

- `DQNAgent` – Extended to support new Q-learning strategies.
- `ReplayBuffer` – Modified to implement experience prioritization and multi-step learning.
- `QNetwork` – Redesigned to integrate dueling architecture, noisy layers, and distributional learning.

1. Prioritized Experience Replay [7]

Standard DQN treats all experiences equally when sampling from the replay buffer. However, some experiences are more valuable than others for learning. **Prioritized Experience Replay (PER)** modifies this by giving higher priority to experiences that contribute more to learning progress. This results in faster convergence and improved sample efficiency.

Key Concepts:

- **Priority Assignment:** Experiences are prioritized based on Temporal Difference (TD) errors:
$$p_i = |\delta_i| \tag{1}$$
where the TD error is computed as:
$$\delta_i = r + \gamma \max_{a'} Q_{\text{target}}(s', a') - Q(s, a) \tag{2}$$

- **Sampling Probability:** Transitions are sampled with probability proportional to their priority:
$$p(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3}$$
where α controls the level of prioritization (e.g., 0.4–0.7).

- **Importance Sampling (IS) Weights:** To counteract the bias introduced by prioritized sampling, importance weights are used:
$$w_i = \left(\frac{1}{N \cdot p(i)} \right)^\beta \tag{4}$$
where β is annealed from 0.4 to 1.0 over training.

- **Priority Updates:** After each learning step, new TD errors are computed, and priorities are updated accordingly:
$$p_i = |\delta_i| \tag{5}$$

Benefits of PER:

- Faster convergence by focusing on the most valuable transitions.
- Improved sample efficiency by reducing redundant, uninformative experiences.
- Better handling of sparse rewards by ensuring rare but important experiences are replayed.

Implementation Details:

- **Replay Buffer Class:** The standard replay buffer is replaced with `ReplayBufferPER`, which implements a priority-based sampling mechanism using a sum-tree structure.
- **DQN Agent Modification:** The agent integrates PER by computing TD errors, applying importance-sampling corrections, and dynamically updating priorities.
- **Training Adjustments:** The importance-sampling factor β is annealed over training:
$$\beta = \min(1.0, \beta_{\text{start}} + \frac{\text{frames}}{\beta_{\text{frames}}}) \tag{6}$$
where β_{frames} defines the total steps for full annealing.

Computational Considerations:

- Efficient data structures (sum-trees) ensure sampling and updating priorities occur in $\mathcal{O}(\log N)$ time.

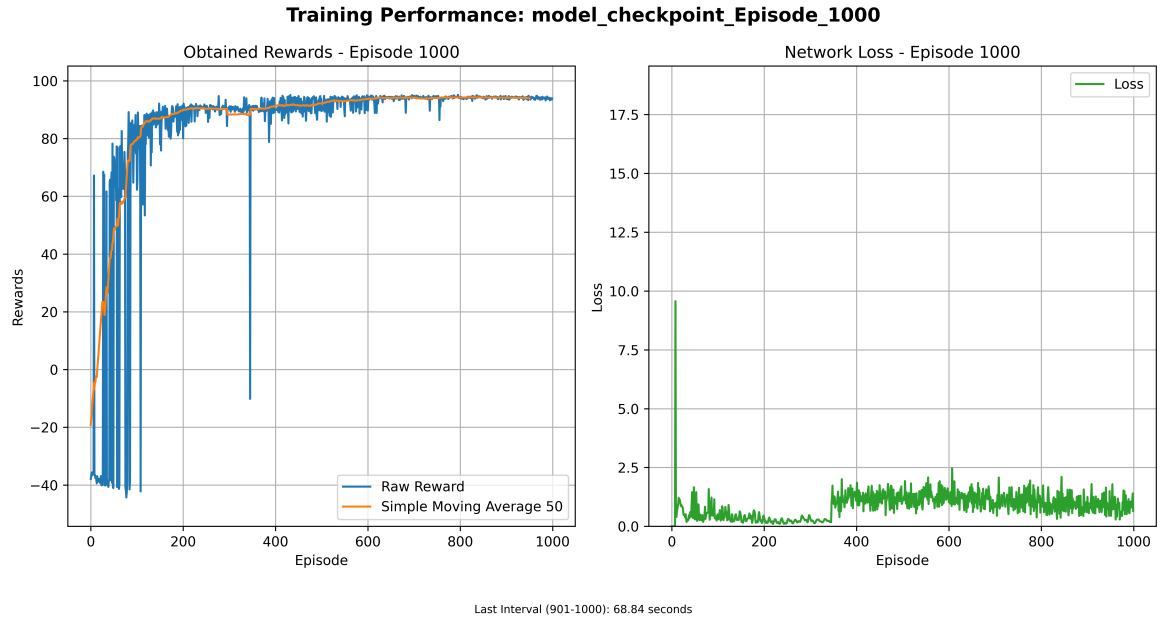

Figure 4: Training performance: rewards and loss over episodes

- Hyperparameters (α, β, ϵ) need careful tuning to balance learning efficiency and stability.
- Large-scale implementations benefit from GPU-accelerated priority updates.

The following figure illustrates the training performance of a DQN agent using Prioritized Experience Replay.

PER significantly improves deep reinforcement learning by ensuring that the agent learns from its most valuable experiences first, accelerating learning efficiency and performance.

2. Double DQN [8]

A well-known problem with standard DQN is its tendency to overestimate Q-values, leading to unstable training. **Double DQN** addresses this issue by **decoupling action selection from value evaluation**, leading to more stable learning and improved policy performance.

Key Concepts:

- **Overestimation Bias in Standard DQN:**

$$y_j = r_j + \gamma \max_{a'} Q_{\text{target}}(s_{j+1}, a'; \theta^-) \tag{7}$$

  Since the same network selects and evaluates the action, this often leads to overestimated Q-values.

- **Double DQN Target Computation:**

$$y_j = r_j + \gamma Q_{\text{target}}(s_{j+1}, \arg\max_{a'} Q_{\text{main}}(s_{j+1}, a'; \theta); \theta^-) \tag{8}$$

  where:

  - The **main network** selects the best action: $\arg\max_{a'} Q_{\text{main}}(s_{j+1}, a'; \theta)$.
  - The **target network** evaluates the action's Q-value: $Q_{\text{target}}(s_{j+1}, a'; \theta^-)$.

- **Decoupling Action Selection and Evaluation:**
  - The main network chooses the optimal action.
  - The target network estimates its value, leading to more accurate Q-value estimation.

Benefits of Double DQN:

- **Reduced Overestimation Bias:** More accurate Q-value estimates.
- **Improved Stability:** Training curves tend to be smoother.
- **Better Generalization:** Performs better in environments with stochastic rewards.

Implementation Considerations:

- The **target network** should still be periodically updated ($\theta^- \leftarrow \theta$).
- Works well with **standard DQN hyperparameters** without additional tuning.

a) Updating the `DQN_Agent` Class

Modify the `learn` method by updating the target Q-value calculation:

- The **main network** selects the best action.
- The **target network** evaluates the chosen action's value.

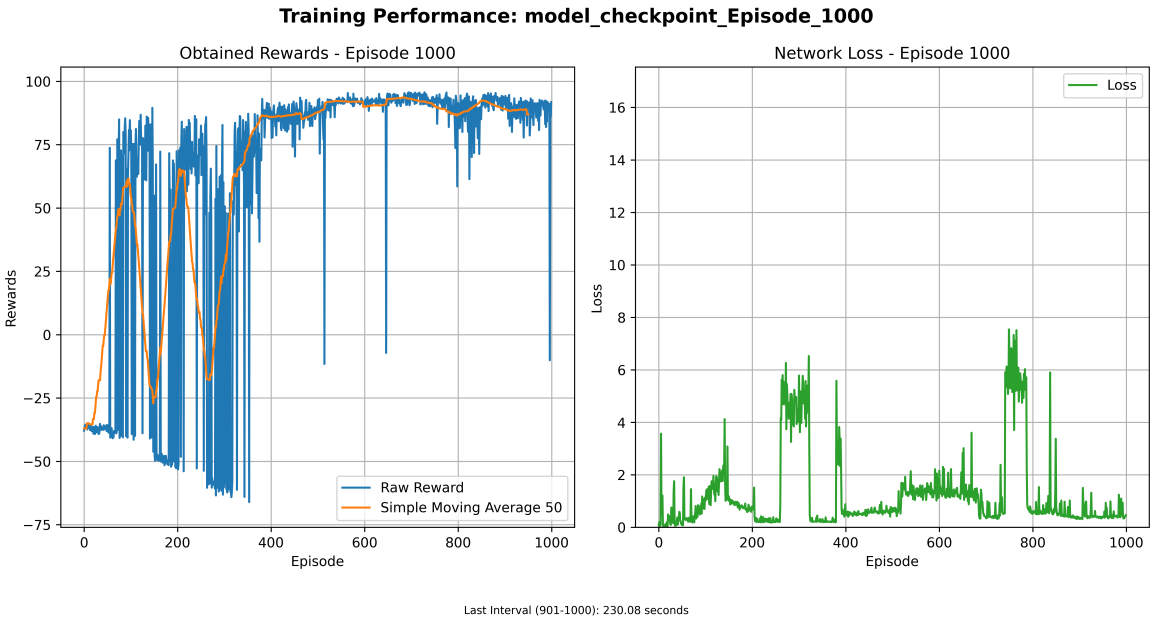

Figure 5: Training performance: rewards and loss over episodes

Why Might Double DQN Not Always Outperform Standard DQN?

- **Improved Stability Does Not Always Mean Higher Rewards:**
  - Standard DQN's overestimation can sometimes lead to faster exploration and higher initial rewards.

  - Double DQN's conservative updates prevent overestimation but may slow down initial learning.

- **Performance Depends on Environment Complexity:**
  - In simpler environments, Double DQN's benefits are minimal.
  - In environments with noisy rewards, its stability advantage becomes more noticeable.

- **Hyperparameter Sensitivity:**
  - Double DQN may require careful tuning of learning rates and exploration strategies to show improvements.
  - Without tuning, its cautious updates may slow learning without significantly improving asymptotic performance.

Despite these factors, **Double DQN** remains a valuable improvement in Deep Reinforcement Learning, especially in environments where standard DQN suffers from overestimation-related instability.

3. Dueling DQN Architecture [9]

Traditional DQNs compute a Q-value for every possible action, making them inefficient in large state-action spaces. **Dueling DQN** improves upon this by separating the estimation of state values and action-specific advantages, allowing the agent to better differentiate between states where actions matter and those where they don't.

Implementation Approach:

- Instead of directly estimating Q-values, the model first estimates how valuable it is to be in a given state, independent of actions.
- A second network estimates the advantage of each possible action relative to the state's value, helping the agent focus on action selection when necessary.
- The two estimates are combined to compute the final Q-values, ensuring better generalization and more robust learning, especially in environments with redundant actions.

Class Modifications:

- `DuelingQNetwork` – A modified Q-network class inheriting from `QNetwork`, adding separate value and advantage streams.

4. Noisy Networks for Exploration [10]

Exploration is critical in reinforcement learning, but standard DQN relies on **epsilon-greedy**, which decays over time and may limit exploration too soon. **Noisy DQN** replaces this with **parameterized noise**, allowing the agent to explore more effectively while adapting exploration based on uncertainty.

Implementation Strategy:

- Noisy layers introduce randomness in the Q-network's parameters, forcing the agent to explore different actions naturally without requiring an explicit exploration schedule.
- Unlike epsilon-greedy, where randomness decreases over time, noisy networks adjust dynamically, maintaining useful exploration even in later training stages.
- The noise is state-dependent, ensuring the agent explores more in uncertain situations and exploits knowledge where confidence is higher.

Class Modifications:

- `NoisyLinear` – A new class for replacing standard linear layers with noisy linear layers, enabling adaptive exploration.

5. Multi-Step Learning [1]

Traditional Q-learning relies on a **single-step reward** update, which can lead to slow learning. **Multi-step learning** extends this by considering rewards over multiple time steps, making learning faster and more effective in environments with delayed rewards.

Learning Approach:

- Instead of updating the Q-value using only the immediate reward, the agent accumulates multiple future rewards to create a more informative training signal.
- This approach helps in environments where rewards are delayed, allowing the agent to recognize long-term benefits instead of relying on short-term gains.
- By capturing longer-term dependencies in decision-making, multi-step learning improves sample efficiency and enhances learning speed.

Class Modifications:

- `ReplayBufferMultiStep` – A new replay buffer extending `ReplayBuffer` to store multi-step returns, enabling better credit assignment.

6. Distributional RL (Categorical DQN - C51) [11]

Standard DQN predicts a single expected Q-value for each action, which can be limiting in uncertain environments. **Distributional RL** improves upon this by predicting an entire probability distribution over possible rewards, giving the agent a deeper understanding of risk and uncertainty.

Conceptual Framework:

- Instead of estimating one Q-value per action, the model learns a **distribution of rewards**, capturing the variability in outcomes.
- This allows the agent to differentiate between actions that have the same expected return but different levels of risk, leading to more informed decision-making.
- Helps in environments with stochastic transitions or noisy rewards, as it considers both optimistic and pessimistic outcomes when choosing actions.

Class Modifications:

- `CategoricalDQN` – A modified DQN implementation that predicts reward distributions instead of scalar Q-values, improving robustness in dynamic environments.

7. Rainbow DQN [16]

Each of the previous improvements addresses a specific limitation of standard DQN. **Rainbow DQN** integrates all of them into a single, more powerful algorithm, benefiting from their combined strengths.

Integration of Enhancements:

- Uses **Prioritized Experience Replay** to focus learning on the most useful transitions, improving sample efficiency.
- Incorporates **Dueling Architecture** to separate state values from action advantages, enabling more robust generalization.
- Applies **Double DQN** to reduce overestimation bias, stabilizing training.
- Uses **Noisy Networks** for adaptive exploration, eliminating the need for epsilon-greedy strategies.
- Employs **Multi-Step Learning** for more efficient credit assignment, speeding up training convergence.
- Integrates **Distributional RL** to predict entire reward distributions, allowing better risk-aware decision-making.

Class Modifications:

- `RainbowDQN` – A complete integration of all enhancements into a single DQN variant, leveraging the advantages of each technique.

Key Benefits:

- **State-of-the-Art Performance** – Combines the best of all techniques.
- **More Stable and Efficient Learning** – Reduced bias, better exploration, and faster convergence.

8. Experiment

The above enhancements were tested and evaluated with slight parameter tuning. While some improvements led to better performance, others did not always result in convergence.

One key observation was that certain enhancements, such as **Duelling DQN** and **Noisy Networks**, did not consistently improve training stability. This suggests that parameter tuning plays a significant role in determining the effectiveness of these techniques.

Furthermore, the chosen environment, **MountainCarContinuous**, may not be ideal for applying all of these enhancements. The environment is known for its sparse rewards and high sensitivity to exploration strategies, making it challenging for value-based methods like DQN to learn effectively. Algorithms that focus on policy optimization, such as **Proximal Policy Optimization (PPO)** or **Soft Actor-Critic (SAC)**, might be better suited for this specific task.

Despite these challenges, the experiments provided valuable insights into the strengths and limitations of different DQN enhancements, emphasizing the importance of **environment suitability and hyperparameter tuning** in reinforcement learning.

Part 5: Neural Network Structure Exploration

This chapter explores the implementation and experimentation of Long Short-Term Memory (LSTM), GRU, and Transformer architectures within a Deep Q-Network (DQN) framework.

1. RNNs Integration into DQN

1.1 Background: RNNs and LSTMs

Recurrent Neural Networks (RNNs) are designed to process sequential data by maintaining a hidden state that evolves over time, thereby capturing temporal dependencies. However, traditional RNNs suffer from the vanishing gradient problem, making it difficult to learn long-term dependencies. LSTMs mitigate this issue by incorporating gating mechanisms that regulate information flow and preserve important signals over longer sequences.

1.2 Rationale for Using LSTMs in DQN

In the context of DQNs, incorporating LSTMs offers some potential benefits:

1. **Handling Partially Observable Environments (POMDPs):**
   The paper *Deep Recurrent Q-Learning for Partially Observable MDPs* [13] was one of the first to integrate RNNs—specifically LSTMs—into a DQN framework. In their work, the authors addressed the challenge posed by environments like Atari Pong, where a single, flickering frame lacks crucial information (such as the ball's velocity and direction), thereby rendering the problem a POMDP. By incorporating recurrency, the network can aggregate information over time, effectively bridging the gap between incomplete observations and the true underlying state.

   While handling Partially Observable Markov Decision Processes (POMDPs) is the primary reason for implementing Recurrent Neural Network (RNN) architectures in Deep Q-Networks (DQNs), other reasons that such integration could be beneficial include:

2. **Temporal Context Accumulation:**
   LSTMs excel at integrating sequential observations, enabling the network to retain and infer hidden aspects of the state that may not be evident in a single observation. This temporal context accumulation is vital for understanding dynamics over time.

3. **Handling Observation Quality Variations:**
   Even though recurrency might not always enhance performance during the learning phase, it allows the network to adapt more effectively when the quality of observations

varies. This flexibility can be crucial in environments where the data is noisy or inconsistent.

4. **Delayed Reward Association:**
   In many reinforcement learning scenarios, rewards are delayed. LSTMs are particularly well-suited to associating actions with rewards that occur several time steps later, thereby facilitating more effective policy learning.

1.3 Implementation Challenges and Modifications

1.3.1 Sequential Dependency vs. Random Sampling

Challenge: Standard DQN training relies on random sampling from a replay memory to ensure independent and identically distributed (i.i.d.) samples. However, LSTMs require sequences of data to capture temporal dependencies effectively.

Solution: To address this, we introduced a new method, `sample_sequences`, which extracts contiguous blocks of transitions. This approach enables the network to learn from sequential data while preserving a degree of randomness in the training process.

1.3.2 Handling Hidden States

Challenge: Managing the LSTM's hidden state is critical. With standard DQN, each observation is treated independently, but LSTMs require a decision on whether to reinitialize the hidden state at each update or maintain continuity across updates.

Solution: We experimented with two approaches:

1. The first approach adhered to the standard DQN methodology by reinitializing the LSTM's hidden state at every update, ensuring consistency with random sampling but potentially limiting long-term dependency capture.

2. In our second approach, we modified the update policy to perform updates once per episode, allowing the hidden state to persist longer and potentially capture extended temporal dependencies—though this comes with a trade-off deviating from the DQN's core ideas.

1.3.3 Computational Efficiency

Challenge: Backpropagation through numerous time steps increases computational demands significantly.

Solution: To balance the need for temporal context against computational constraints, the network was unrolled for a fixed sequence length. This method helps manage the computational load; however, it is important to note that RNN-based architectures, when combined with the DQN's standard update policy, are generally much slower compared to the other network architectures we explored.

1. Vanilla Update Policy for DQN-LSTM Implementation

- **Replay Memory:**
  - The original implementation sampled individual transitions from memory. To better serve the sequential nature of LSTMs, we introduced a new `sample_sequences` method that extracts contiguous sequences of transitions.

- **DQN Network Architecture:**
  - The conventional feed-forward network was enhanced by inserting an LSTM layer before the fully connected layers. This LSTM processes inputs with the shape (`batch_size, sequence_length, input_dim`) and utilizes the final output of the sequence to predict Q-values.
- **DQN Agent Adjustments:**
  - When selecting actions or performing updates, the agent now reshapes states into sequences (even when the sequence length is one) to meet the LSTM's input requirements.

2. Modified Update Policy for DQN-LSTM Implementation

This alternative approach was designed to leverage the recurrent architecture's strength in modeling temporal sequences by adopting an episodic learning strategy.

- **Episodic Learning:**
  - The learning method, `learn_episode`, processes an entire episode's transitions in sequence rather than sampling mini-batches. This change capitalizes on the LSTM's ability to capture long-term dependencies within a complete episode.
- **Simplified Replay Memory:**
  - The replay memory was streamlined by removing the need for random sampling methods, thereby aligning it with the episodic update strategy.
- **Episodic Training Loop:**
  - The training loop was adapted to process full episodes—updating both the primary network and the target network based on complete sequences instead of periodic mini-batch updates—thus harmonizing with the episodic learning approach.

1.4 Experimental Results: LSTM-Based DQN

We conducted a series of experiments using both the vanilla and modified update policies. The observations described below are derived from the accompanying plots:

1. **Vanilla DQN-LSTM Implementation**

   - **Observations:**
     - The agent quickly converged to a "safe" strategy, remaining stationary throughout episodes.
   - **Analysis:**
     - The loss function dropped rapidly and remained consistently low.
     - Cumulative rewards increased slowly due to ϵ's progressive decay but remained below zero.
   - **Possible Causes:**

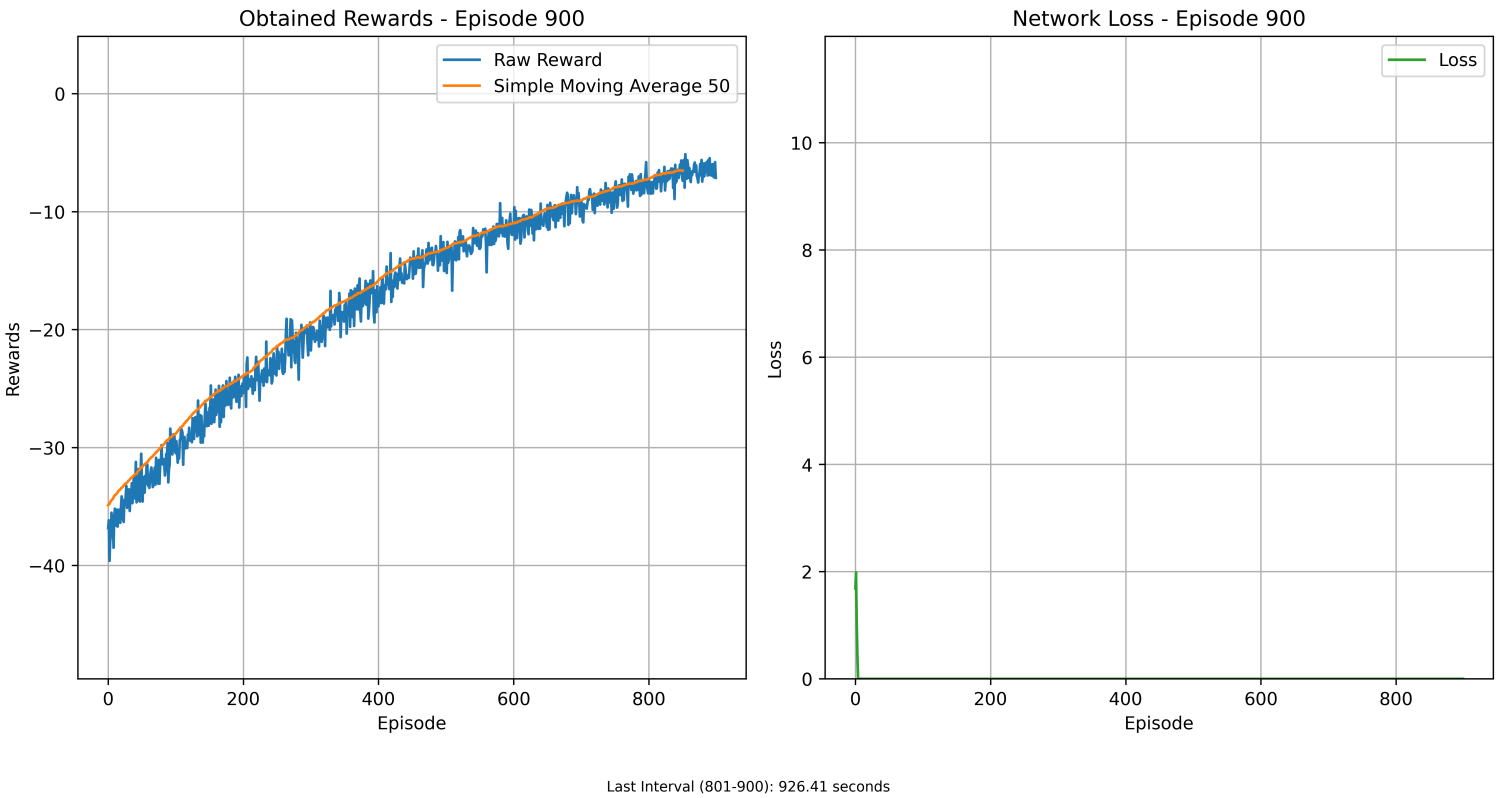

Figure 6: Training performance of Vanilla DQN-LSTM

- Initial reward design.
- Reinitialization of the LSTM hidden state at every update.
- Limited number of hidden state dimensions (selected to manage computational costs).

2. **Vanilla DQN-LSTM with Modified Reward Function**

- **Observation:**
  - To encourage exploration, we implemented a modified reward function. Although the new approach produced more promising results than our initial attempt, the agent still failed to consistently navigate the car to the finish line.

3. **GRU-Based Implementation**

- **Observation:**
  - In an effort to reduce the network's architectural complexity, we replaced the LSTM with a GRU. However, the experimental outcomes were identical to those of the LSTM-based approach, leading us to discontinue further exploration of this variant.

4. **Modified LSTM-DQN with Episodic Learning**

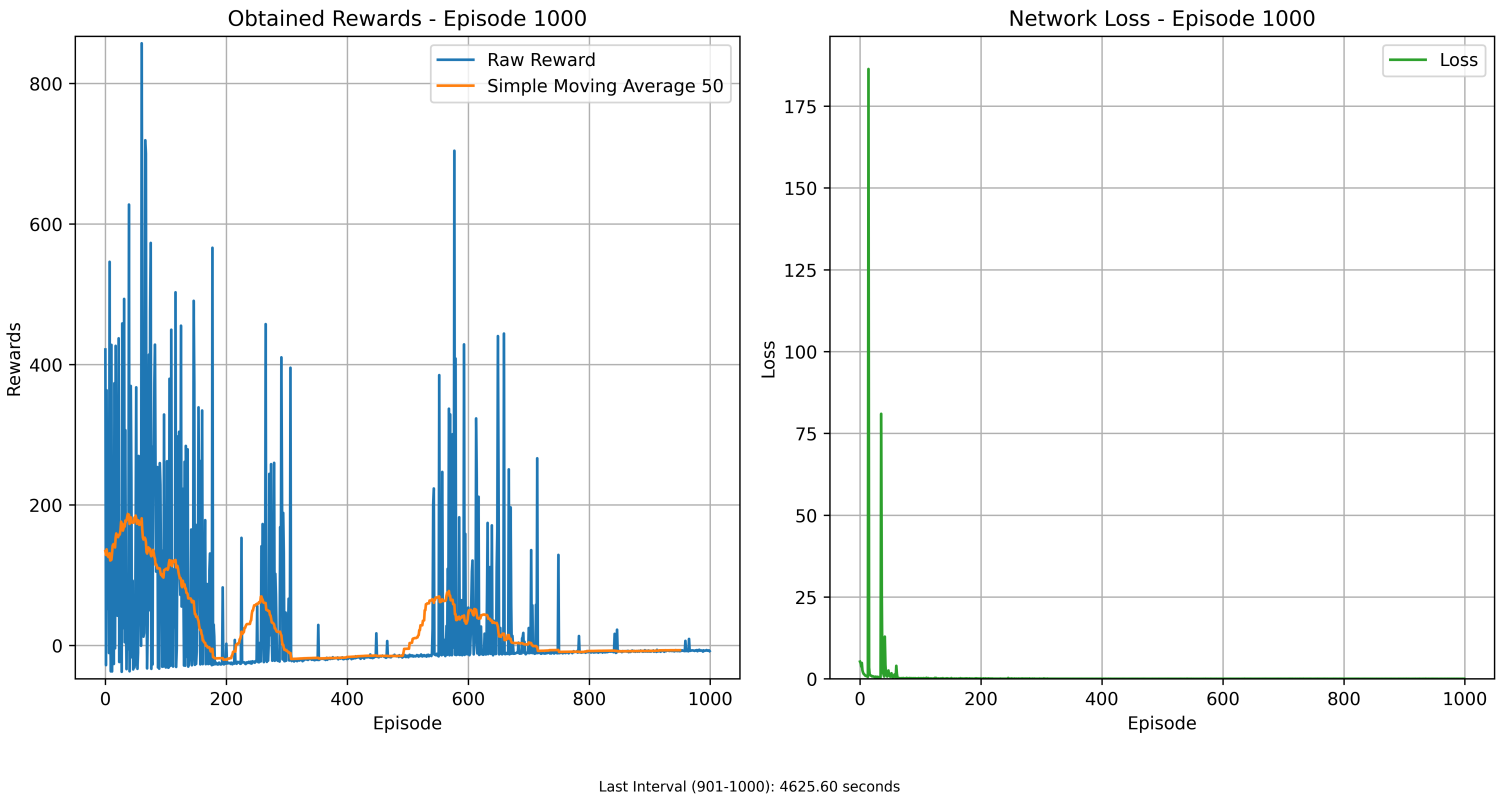

Figure 7: Training performance of Vanilla DQN-LSTM with Modified Reward Function.

- **Modifications:**
  - The hidden state update policy was revised to process entire episodes sequentially (episodic learning), rather than relying on mini-batches.
  - The replay memory was simplified by removing random sampling methods.
- **Observations:**
  - Training accelerated considerably, which allowed us to increase the number of LSTM hidden units from 4 to 20.
  - Despite these enhancements, the agent still failed to effectively navigate to the finish line.

5. **Modified LSTM-DQN with Episodic Learning and Modified Reward Function**

- **Observation:**
  - When applying the modified reward function in combination with the episodic learning strategy, the agent learned to reliably navigate the car to the finish line. This approach outperformed the vanilla DQN using the same reward system, with minimal additional computational overhead.
- **Further Investigation Required:**
  - There are indications that the agent might exploit the new reward system by frequently visiting areas near the finish line to accumulate rewards before completing the episode.
  - A deeper analysis of the loss function's behavior is warranted.

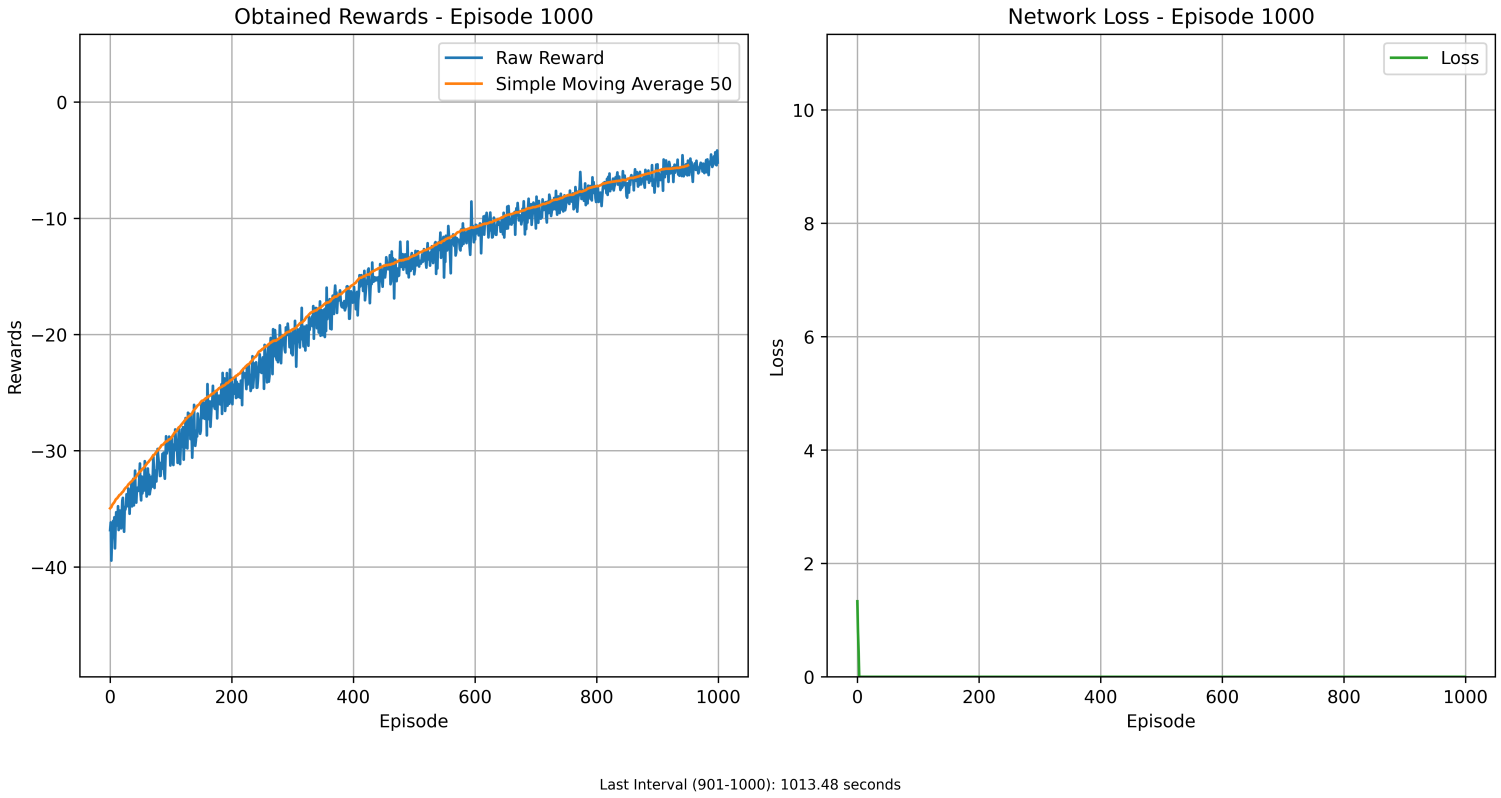

Figure 8: Training performance of GRU-Based Implementation.

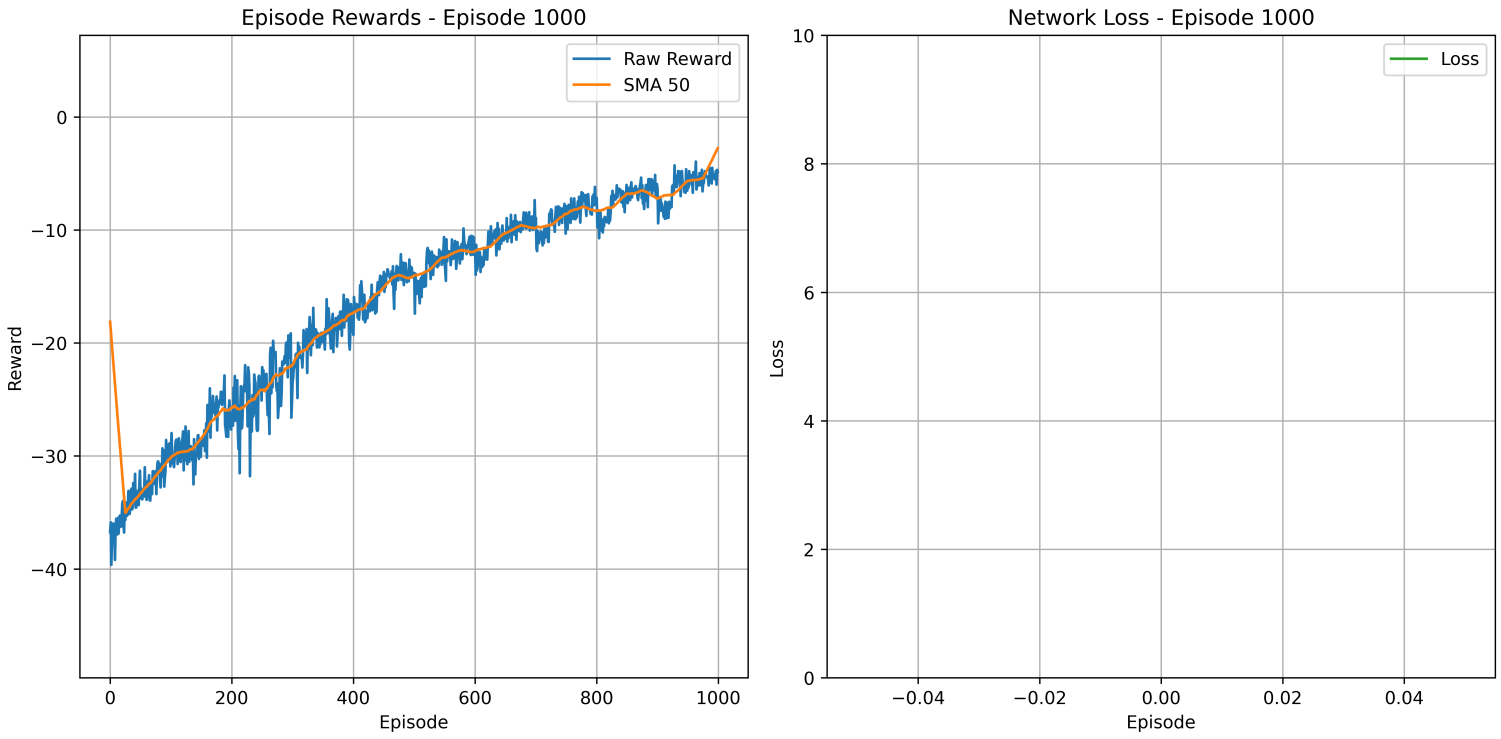

Figure 9: Training performance of Modified LSTM-DQN with Episodic Learning.

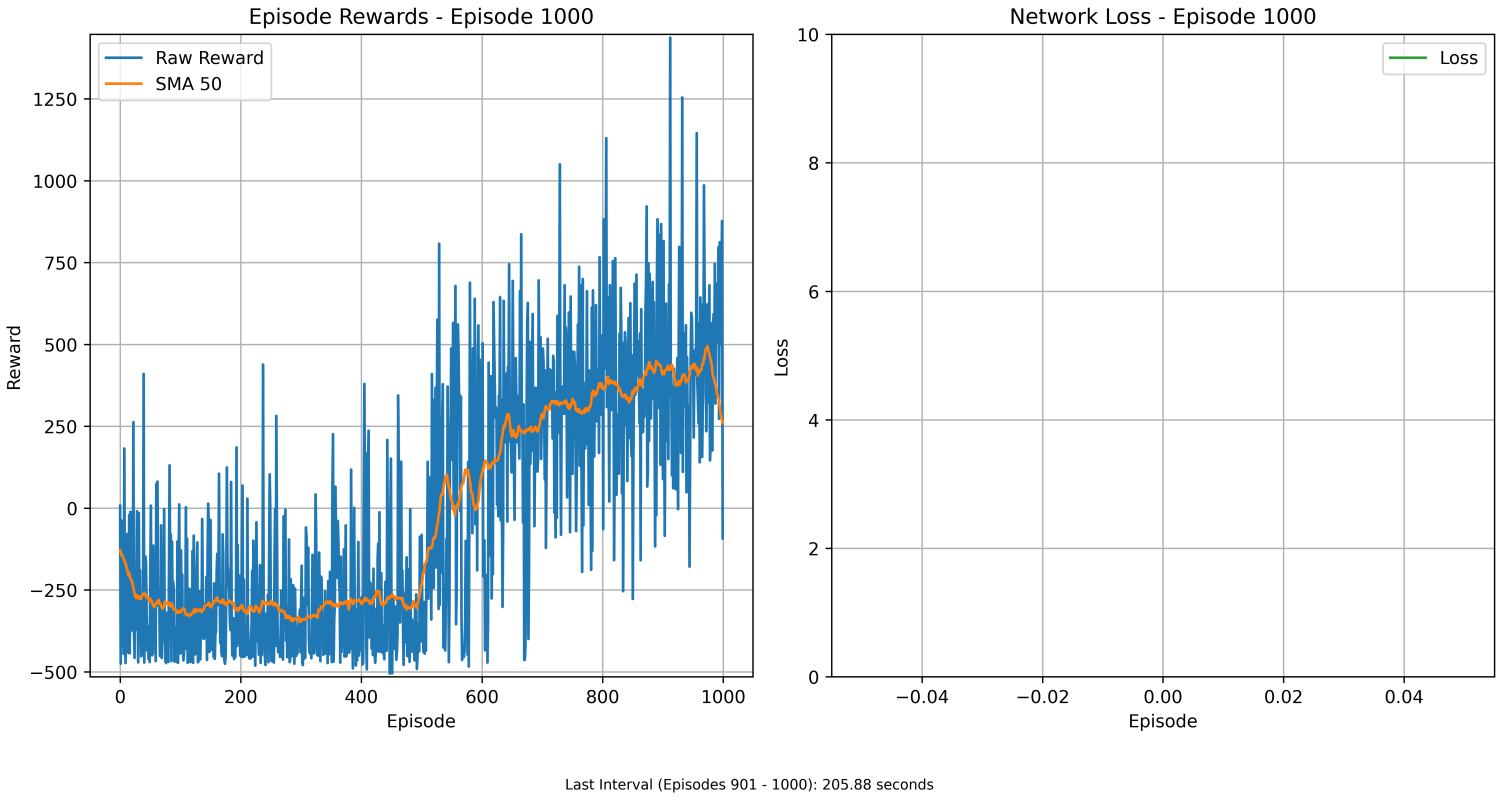

Figure 10: Training performance of Modified LSTM-DQN with Episodic Learning and Modified Reward Function.

2. Transformer Integration into DQN

2.1 Background: Transformers in Reinforcement Learning

Introduced by Vaswani et al. (2017) [17], Transformer architectures leverage self-attention mechanisms to capture relationships between all elements in a sequence simultaneously. This approach has several advantages over traditional recurrent or convolutional layers:

- **Efficiency and Parallelization:** Self-attention allows for parallel computation, reducing training times.
- **Richer Representations:** By encoding contextual relationships, Transformers can generate more informative representations of each state.

As a result, Transformers have been steadily replacing RNNs in reinforcement learning because of the richer, non-sequential contextual representations and enabling parallel processing. This shift is particularly advantageous in RL, where understanding complex spatial–temporal patterns is essential for effective decision-making.

2.2 Implementation Modifications

For the Transformer-based DQN, the following changes were made:

- **Embedding Layer:** Converts the 2D state into a higher-dimensional space appropriate for self-attention.
- **Transformer Encoder:** Processes the embedded input with a configurable number of attention heads and layers (in our case, 4 heads and 1 layer) to learn intra-sequence relationships.
- **Sequence Dimension:** The network treats each state as a sequence of length one, maintaining compatibility with transformer inputs.
- **Output Layer:** The final output of the transformer is mapped to Q-values, preserving the original action space.

Overall, the integration of the Transformer architecture was significantly easier and presented significantly fewer challenges compared to the LSTM approach.

2.3 Experimental Results: Transformer-Based DQN

1. **Vanilla Transformer Implementation**

    - **Observations:**
        - Experiments with the vanilla Transformer architecture yielded results similar to the LSTM-based implementation.
        - Notably, this approach incurred significantly lower computational costs and achieved better reward outcomes.
        - However, the agent predominantly adopted a safe policy, remaining stationary rather than moving toward the goal.

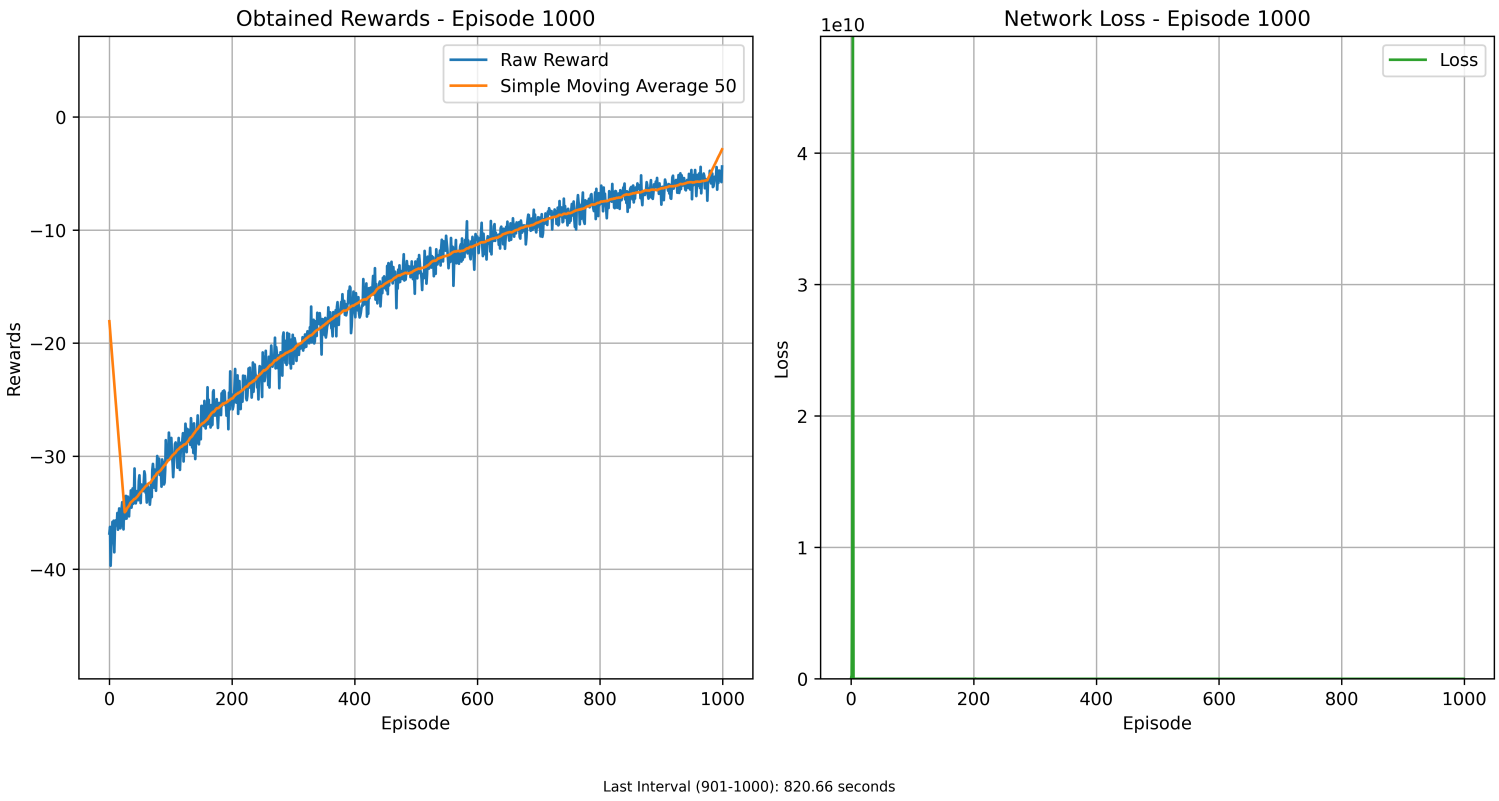

Figure 11: Training performance of Vanilla Transformer Implementation.

- **Hypothesis:**
  The inherent complexity of the Transformer may be excessive for the simplicity of the environment under study.

2. **Alternating Between DQN and Transformer**

- **Implementation:**
  To address the complexity issue, we implemented a hybrid approach:
  - In the first 200 episodes, the network operated as a vanilla DQN to facilitate exploration and enable the agent to locate the finish line.
  - After this initial phase, the Transformer was integrated into the network.
- **Observations:**
  - While this strategy enabled the agent to reliably find the finish line, it did not offer a systemic advantage over a fully vanilla DQN.
  - The additional computational overhead was not justified by a significant performance gain.

3. **Transformer with Modified Reward Function**

- **Experiment:**
  In this experiment, we applied a modified reward function to the Transformer-based DQN.
- **Observations:**

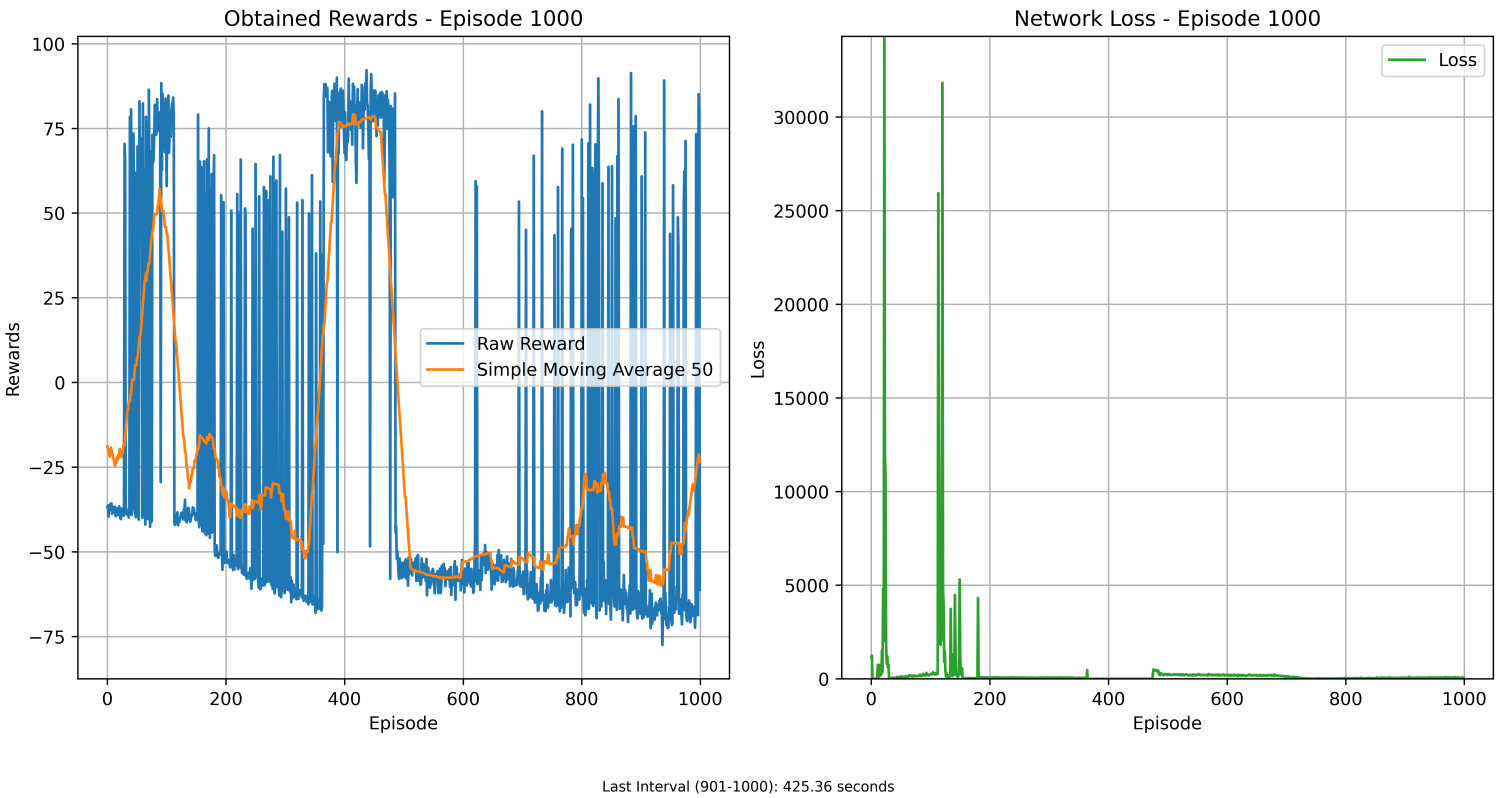

Figure 12: Training performance of Alternating Between DQN and Transformer.

  - This modification resulted in superior reward performance compared to both the vanilla DQN and the modified LSTM approach.

- **Concerns:**
  - The potential for the agent to exploit the reward system remains an issue.
  - Further analysis of the loss function's behavior is required.

4. **Transformer in a Noisy Environment**

- **Experiment:**
  We introduced environmental noise—a scenario where Transformers typically excel due to their robustness in POMDP settings.
- **Observations:**
  - Under these conditions, the agent demonstrated notably successful performance.
  - The agent learned to navigate to the finish line relatively quickly (around the 400th epoch).
  - The agent continued to do so consistently, albeit without complete stability.

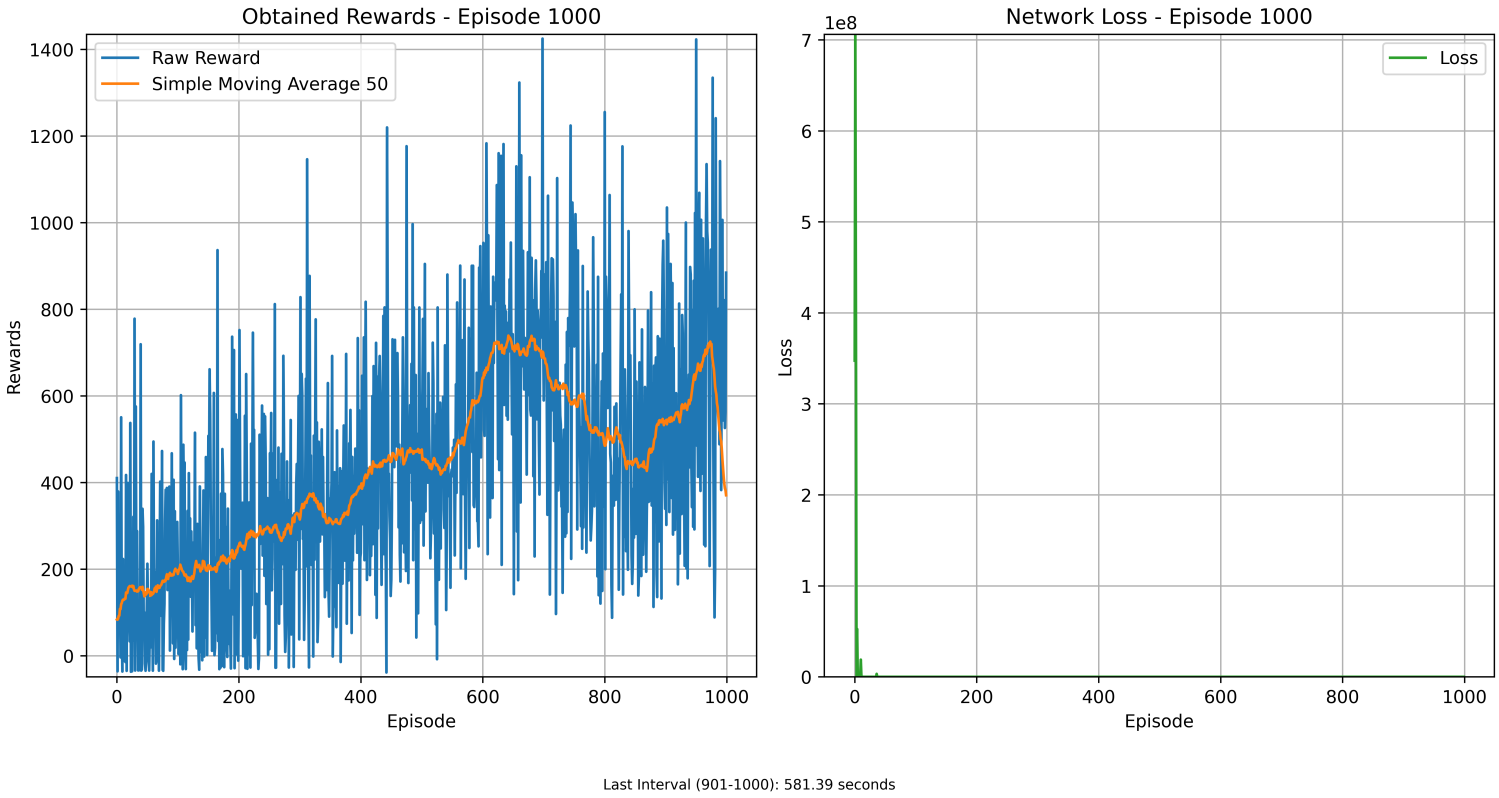

Figure 13: Training performance of Transformer with Modified Reward Function.

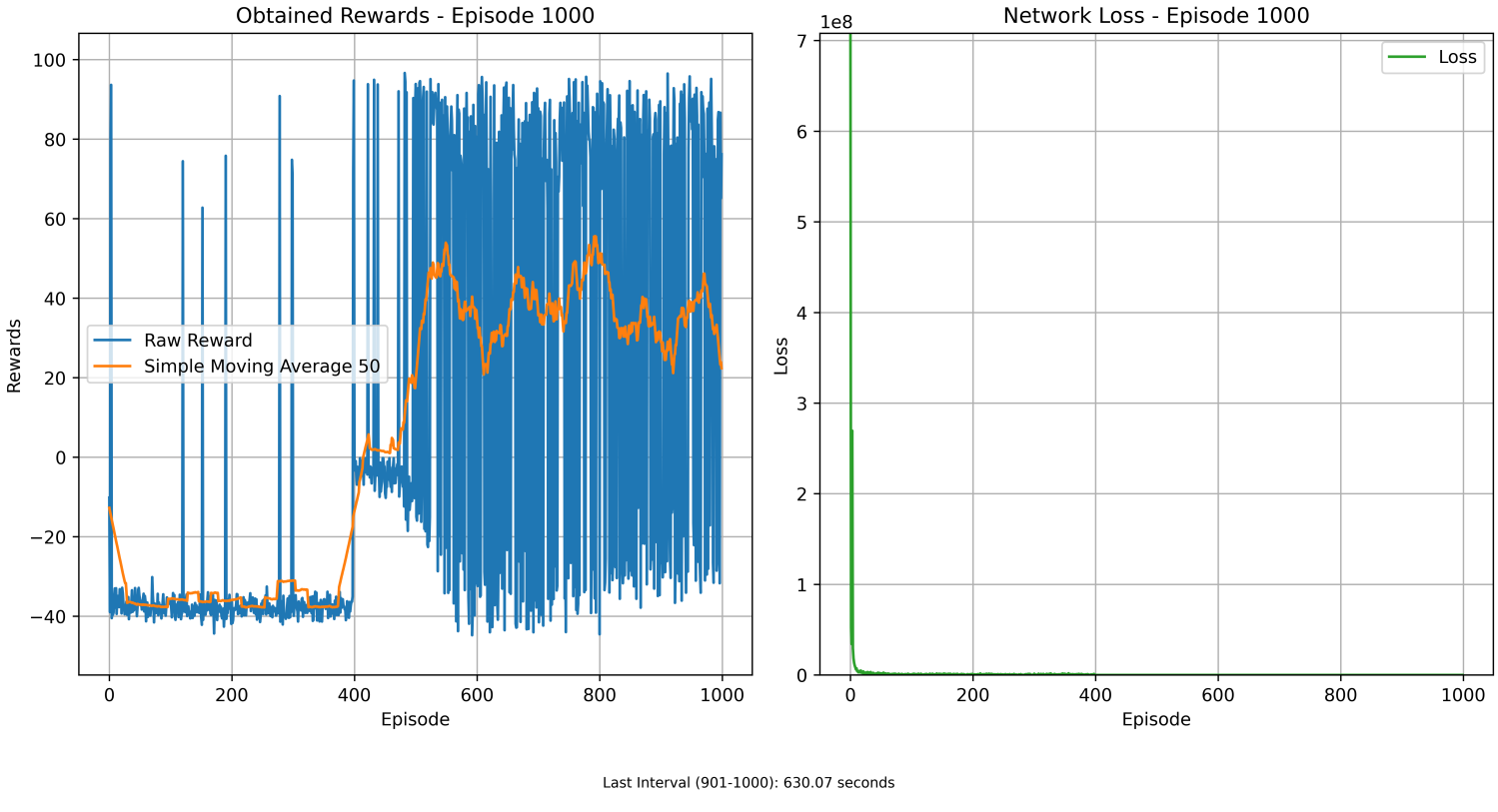

Figure 14: Training performance of Transformer in a Noisy Environment.

Conclusion

Our findings suggest that for simple environments, the complexity of RNNs and Transformer architectures may not provide significant performance benefits over a vanilla DQN—an observation encapsulated by the adages *"less is more"* and *"if it ain't broke, don't fix it."*

However, as implied by the bibliography, in more complex or noisy environments, where essential information is sparse or observations are incomplete, architectures based on RNNs and Transformers can play a pivotal role in enhancing agent performance.

Part 6: PPO and A2C Algorithm Comparison

In this chapter, we explore the implementations of Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C) algorithms in reinforcement learning. These algorithms are widely used in policy optimization tasks, each with unique strengths and trade-offs. The following sections provide a detailed analysis of their training processes, performance comparisons, and hyperparameter tuning.

Part 1: Proximal Policy Optimization (PPO)

a. Model Training and Analysis

Proximal Policy Optimization (PPO) is a policy gradient method that improves traditional actor-critic methods by "clipping" policy updates. Instead of allowing the new policy to change drastically from the old one in a single update, PPO restricts the change to a small, safe range. This helps keep learning stable and prevents the agent from taking leaps in policy space. Thus, PPO is a more stable policy gradient algorithm.

We initialize the PPO algorithm with the Stable-Baselines3 default hyperparameters, and we use the `MlpPolicy`, which utilizes two multi-layer perceptron (MLP) networks—one to parameterize policy updates and one for the value updates.

Policy and Value Networks

Policy Network (Actor)

The policy network determines the actions the agent should take based on the current state and updates the policy. (Continuous Action Space)

$$\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)^2) \tag{9}$$

Value Network (Critic)

The value network calculates the value function, which represents the expected cumulative reward from a given state. It helps compute the advantage function A_t, which evaluates how good an action is compared to the expected value.

$$V_\phi(s) = f_\phi(s) \tag{10}$$

Advantage Function and Temporal Difference (TD) Error

The advantage function is approximated using the TD error:

$$A_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{11}$$

The TD error quantifies the difference between the predicted value and the observed value. PPO uses this advantage function within the clipped surrogate objective to stabilize policy updates and prevent excessively large changes.

Generalized Advantage Estimation (GAE)

To improve stability, PPO employs Generalized Advantage Estimation (GAE), which computes the advantage function by weighting TD errors across multiple future steps:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \tag{12}$$

$$\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t). \tag{13}$$

GAE helps the algorithm stabilize training and improve efficiency.

Neural Network Architecture

Both MLPs have the following default structure [5]:

- 2 hidden layers with 64 neurons each.
- Tanh activation function for hidden layers.
- Adam optimizer.

Policy and Value Network Updates in PPO

- **Actor:** Updates the policy to maximize the clipped surrogate objective, ensuring stable updates.
- **Critic:** Minimizes the value loss by reducing the squared difference between the targeted value and the observed returns.

b. Total Time Steps Analysis

These are the results when the agent trains and interacts with the environment for two different timestep values: 40k and 100k.

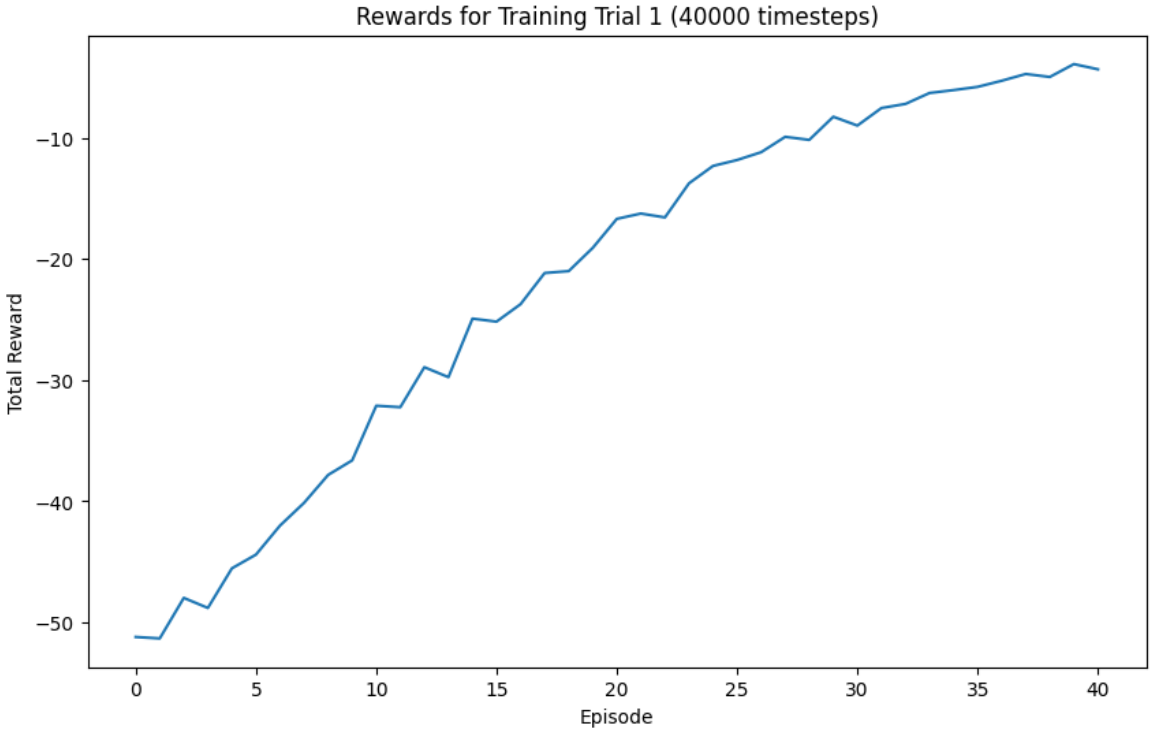

Figure 15: Rewards for Training Trial 1 (40,000 timesteps)

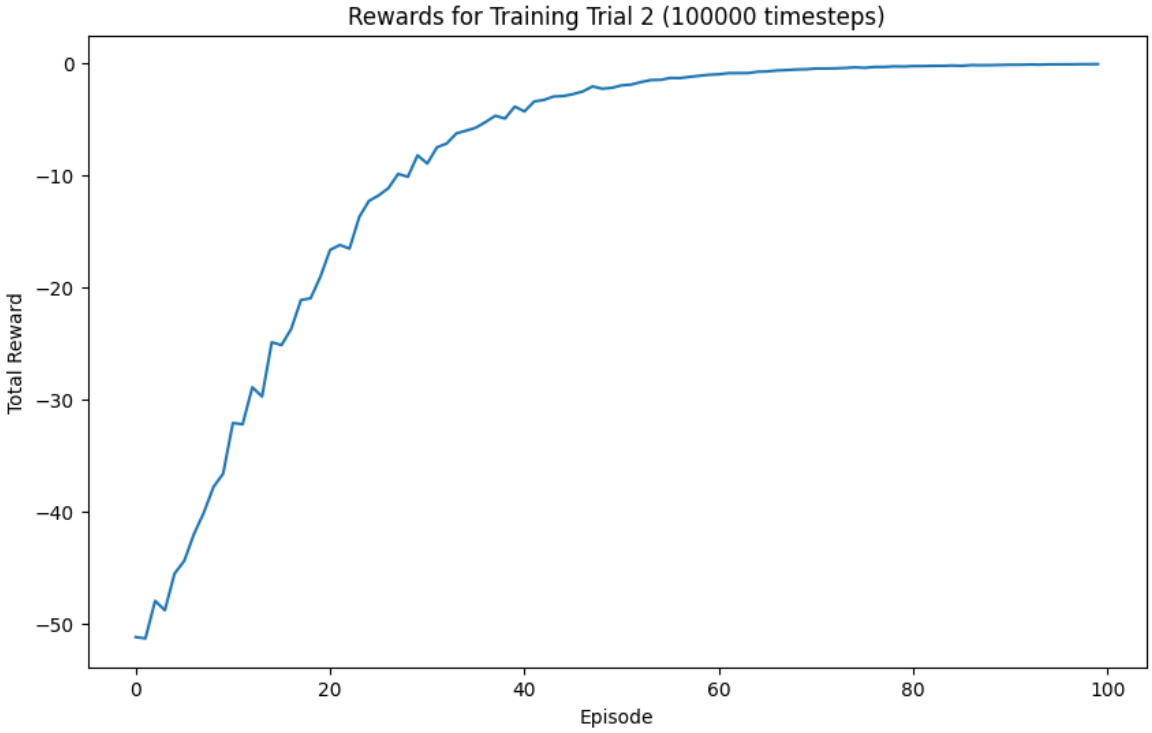

Figure 16: Rewards for Training Trial 2 (100,000 timesteps)

Training Reward Summary

Training Trial	Timesteps	Training Reward Mean
Trial 1	40,000	-21.84
Trial 2	100,000	-9.50

Table 2: Training reward comparison for different timesteps

Observations

These graphs represent the total rewards per episode over the training period. The x-axis shows the episode number, and the y-axis shows the cumulative reward for each episode. At first, we see that the algorithm is improving, as the rewards increase from -50 toward 0. However, for both 40k and 100k total timesteps, the rewards stabilize around 0 and do not exceed this value, showing that the agent has plateaued and is not learning the optimal policy. This happens because it tries to maximize the reward only with small moves back and forth, making it stuck at 0, never reaching the goal of $+100$ reward. This possibly happens because the agent does not explore enough and reaches the reward sparsely.

c. State-Dependent Exploration Analysis

The SDE parameter controls whether the state-dependent exploration mechanism is used during training.

- When set to **False**, the default exploration strategy is used (independent Gaussian noise). This approach is computationally cheaper and sufficient for standard tasks.
- When set to **True**, SDE replaces the standard noise in the policy with state-dependent noise. This allows for more structured and adaptive exploration based on the state, making it more suitable for continuous action spaces. However, it is computationally expensive since the noise is sampled once at the start of a rollout.

To evaluate this hyperparameter, we conduct experiments using **100k total timesteps**.

Mathematical Formulation

The policy network samples an action a from a Gaussian distribution with a mean $\mu_\theta(s)$ and a state-dependent variance σ_s^2:

$$\pi_\theta(a \mid s) = \mathcal{N}(\mu_\theta(s), \sigma_s^2) \tag{14}$$

The action at time step t is defined as:

$$a_t = \mu(s_t, \theta_\mu) + \epsilon(s_t, \theta_\epsilon) \tag{15}$$

where $\epsilon(s_t, \theta_\epsilon)$ represents the state-dependent noise, modeled as:

$$\epsilon(s_t, \theta_\epsilon) = \theta_\epsilon \cdot s_t \tag{16}$$

The noise parameter θ_ϵ is sampled from a normal distribution:

$$\theta_\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{17}$$

State-dependent exploration allows the agent to explore in a way that depends on the state itself, which can be beneficial for environments where exploration needs to be adaptive rather than purely random.

Training Reward Summary with SDE

Training Trial	SDE	Training Reward Mean
Trial 1	False	-9.50
Trial 2	True	60.28

Table 3: Training reward comparison with and without State-Dependent Exploration (SDE)

Observations

The results show the impact of enabling State-Dependent Exploration (SDE) during training.

- Without SDE, the training process remains stable, but the agent does not explore sufficiently. As a result, it struggles to learn the optimal policy and fails to improve significantly.

- With SDE enabled, the agent explores the environment more effectively, leading to a noticeable improvement in performance. The training reward increases over time, and the agent eventually stabilizes at a higher reward level.

These findings indicate that enabling SDE enhances exploration and improves learning efficiency, but it may introduce additional computational overhead.

d. Entropy Coefficient Analysis

The entropy coefficient c_2 controls the weight of the entropy term in the PPO loss function. Entropy measures the randomness in the agent's action distribution. Adding an entropy bonus encourages exploration by preventing the policy from becoming too deterministic early in training.

For testing this hyperparameter, we use **100k total timesteps**.

Full Objective Function

The PPO loss function with entropy regularization is given by [3]:

$$L_t^{\mathsf{CLIP+VF+S}}(\theta) = \widehat{\mathbb{E}}_t \left[L_t^{\mathsf{CLIP}}(\theta) - c_1 L_t^{\mathsf{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{18}$$

where:

$$L_t^{\mathsf{CLIP}}(\theta) = \widehat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\widehat{A}_t, \mathsf{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\widehat{A}_t \right) \right] \tag{19}$$

$$L_t^{\mathsf{VF}}(\theta) = \left(V_\theta(s_t) - V_t^{\mathsf{target}} \right)^2 \tag{20}$$

$$S[\pi_\theta](s_t) = \frac{1}{2} \log(2\pi e \sigma^2) \tag{21}$$

Training Reward Summary with Different Entropy Coefficients

Training Trial	Entropy Coefficient	Training Reward Mean
Trial 1	0.001	-29.69
Trial 2	0.01	-15.97
Trial 3	0.1	24.51

Table 4: Training reward comparison with different entropy coefficients

Observations

The results demonstrate how different entropy coefficients affect the agent's performance.

- For a low entropy coefficient, the agent struggles to explore effectively. The policy becomes too deterministic early in training, leading to poor performance.

- When the entropy coefficient is increased, the agent explores more than in the lower entropy setting, but the improvement is not significant.

- For a high entropy coefficient, the agent shows a substantial increase in performance. However, excessive exploration may still prevent the agent from converging to the optimal policy.

These results indicate that tuning the entropy coefficient is crucial—too little entropy limits exploration, while too much may reduce stability.

e. Learning Rate Analysis

The learning rate is a hyperparameter that controls the step size of the gradient updates for both the policy and value networks, which share the same optimizer. By default, SB3's PPO uses the Adam optimizer with a learning rate of 3×10^{-4}, which balances convergence speed and stability for most tasks.

For testing this hyperparameter, we use **100k total timesteps**.

Training Reward Summary with Different Learning Rates

Training Trial	Learning Rate	Training Reward Mean
Trial 1	3×10^{-5}	-43.18
Trial 2	0.0003	-15.97
Trial 3	0.003	81.77

Table 5: Training reward comparison with different learning rates

Observations

The results demonstrate how different learning rates affect the agent's ability to learn an optimal policy.

- With a smaller learning rate, the algorithm struggles to converge or stabilize, leading to a negative training reward mean. The small step size slows down learning, preventing the agent from effectively optimizing its policy.

- When the learning rate is increased, the agent improves its exploration and learns a better policy. However, the reward mean remains negative, suggesting that learning is still slow and suboptimal.

- At a high learning rate, the agent shows significantly better performance. The optimizer makes larger parameter updates, enabling the agent to learn the optimal policy faster and terminate each episode sooner. This results in a higher reward mean and a faster convergence process.

These findings highlight the importance of tuning the learning rate—too small, and learning is slow and ineffective; too large, and instability may arise, but in this case, it accelerates convergence to an optimal policy.

f. Maximum Gradient Clipping Analysis

During training, gradients are computed for each parameter in the networks, and if their total norm exceeds the threshold (*max_grad_norm*), they are scaled down proportionally to match the threshold. This prevents large updates, stabilizing training and avoiding exploding gradients.

A smaller Maximum Gradient Clipping restricts updates more aggressively, improving stability but slowing convergence, while a larger one allows faster convergence but risks instability if the gradients become too large.

Gradient Clipping Formulation

The gradients for all parameters are represented as:

$$g = [g_1, g_2, \ldots, g_n] \tag{22}$$

The norm of the gradients is calculated as:

$$\|g\|_2 = \sqrt{\sum_{i=1}^{n} g_i^2} \tag{23}$$

If:

$$\|g\|_2 > \text{max\_grad\_norm} \tag{24}$$

Then, the gradients are scaled as:

$$g_i' = g_i \cdot \frac{\text{max\_grad\_norm}}{\|g\|_2} \tag{25}$$

This ensures that the new gradient norm satisfies:

$$\|g'\|_2 = \text{max\_grad\_norm} \tag{26}$$

Training Reward Summary with Different Gradient Clipping Values

Training Trial	Maximum Gradient Clipping	Training Reward Mean
Trial 1	0.4	65.20
Trial 2	0.5	81.77
Trial 3	0.6	80.01
Trial 4	0.65	90.93
Trial 5	0.7	28.70

Table 6: Training reward comparison with different maximum gradient clipping values

Observations

The results illustrate how different gradient clipping values influence training performance.

- With a **small clipping value**, training remains unstable, as the updates are too constrained, leading to slower convergence and suboptimal performance.

- Around the **default range (0.5-0.6)**, the algorithm achieves more stability and consistently converges with a training reward mean close to 80.

- With a **higher clipping value (0.65)**, the algorithm converges faster, accumulating higher rewards, reaching close to a 90 training reward mean.

- When the **clipping value exceeds 0.65**, the algorithm becomes increasingly unstable, causing excessive fluctuations and preventing it from reaching an optimal convergence point.

These results suggest that while increasing gradient clipping can accelerate convergence, excessive values lead to instability, making it important to find a balanced threshold.

g. Seeding

Seeding controls the initialization of randomness in training processes, ensuring reproducibility by fixing the random number generators for neural network weights, environment dynamics, and exploration noise.

To analyze its impact, the algorithm is tested with different seeds.

Training Reward Summary with Different Seeds

Training Trial	Seed	Training Reward Mean
Trial 1	0	22.64
Trial 2	5	1.49
Trial 3	10	64.27
Trial 4	25	1.73
Trial 5	42	80.01
Trial 6	50	-22.17

Table 7: Training reward comparison with different seeds

Observations

The results demonstrate significant variability in performance across different seeds:

- Some seeds, such as **10 and 42**, result in high training rewards, suggesting that certain initializations favor better policy learning.

- Other seeds, such as **5, 25, and 50**, lead to poor or even negative training rewards, highlighting the sensitivity of the training process to initialization.

- The inconsistency in results indicates that different seeds can cause widely varying learning behaviors, with some trials failing to converge or showing unstable reward progressions.

Based on these results, **seed 42** is selected as the optimal choice for this project due to its strong performance, while **seed 50** is identified as the worst-performing seed.

h. Conclusion

In the MountainCar Continuous environment, the agent only receives a positive reward upon reaching the goal. With the PPO algorithm, this results in a **sparse reward problem**. PPO is an on-policy algorithm that performs a policy gradient update after each episode. Since reaching the goal by random actions is a rare event, the agent struggles to receive consistent learning signals. Even when it reaches the goal by chance, a single policy gradient update is often insufficient to maintain progress, causing the agent to get stuck without further learning signals.

To address this issue, I experimented with hyperparameters that promote **aggressive exploration** and **faster learning**:

- A **higher entropy coefficient** (`ent_coef=0.01`) encourages more stochastic policies and broader exploration.

- **State-dependent exploration** (`use_sde=True`) adapts exploration noise dynamically based on the state.

- A **higher learning rate** (`3e-3`) and **larger gradient updates** (`max_grad_norm=0.65`) allow for quicker adaptation to sparse rewards.

These modifications increase variance and training fluctuations, helping the agent explore more effectively and mitigate the sparse reward problem. As a result, the algorithm is able to **converge more reliably**. Additionally, the experiments highlight the importance of **seeding**, as different random initializations can significantly impact the model's training dynamics.

By fine-tuning these hyperparameters, PPO can better handle sparse rewards and improve its learning efficiency in challenging environments like MountainCar Continuous.

2. Synchronous Advantage Actor-Critic (A2C)

a. Model Training and Analysis

Synchronous Advantage Actor-Critic (A2C) is an actor-critic method where an actor selects actions and a critic estimates the value of each state, updating the policy based on the computed advantage (the difference between the actual return and the expected return). By synchronizing updates across multiple concurrent environments, A2C achieves more stable and reproducible training compared to asynchronous methods like A3C. However, while its simplicity makes it easier to implement, A2C can be more sensitive to large updates than algorithms like PPO, which incorporate clipping in policy updates.

For this implementation, I initialize the A2C algorithm using the **Stable-Baselines3** default hyperparameters and utilize the `MlpPolicy`, which consists of two multi-layer perceptron (MLP) forward networks—one to parameterize policy updates and one for value estimation. All experiments are conducted using **four concurrent environments** in the `MountainCarContinuous` environment.

Policy Network (Actor)

The policy network determines the actions the agent should take based on the current state and updates the policy using the advantage function.

$$\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma^2) \tag{27}$$

Since the action space is continuous, the policy is modeled as a Gaussian distribution with mean $\mu_\theta(s)$ and fixed variance σ^2.

Value Network (Critic)

The value network computes the state value function, which represents the expected cumulative reward from a given state. It helps compute the advantage function A_t, which measures how much better an action is compared to the expected value.

$$V_\phi(s) = f_\phi(s) \tag{28}$$

Advantage Function with Temporal Difference (TD) Error

The advantage function is computed using the TD error:

$$\delta_t^{(n)} = \left(\sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}) \right) - V(s_t) \tag{29}$$

$$A_t = \delta_t^{(n)} \tag{30}$$

The TD error quantifies the difference between the predicted value and the observed return. A2C directly uses this advantage function to update the policy.

Generalized Advantage Estimation (GAE)

A2C optionally supports Generalized Advantage Estimation (GAE) for more stable advantage computation when the parameter $\lambda < 1$:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{31}$$

$$\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t). \tag{32}$$

GAE applies a weighting factor $\gamma\lambda$ across multiple time steps, balancing bias and variance in the advantage estimation, which stabilizes training.

Default MLP Structure

The MLP architecture used in A2C consists of [6]:

- Two hidden layers with 64 neurons each.
- **ReLU** activation function for hidden layers.
- **RMSProp** optimizer.

The Policy and Value Networks in A2C

A2C optimizes both the policy (actor) and value (critic) networks as follows:

1. The **Actor** updates the policy to maximize the expected advantage A_t by performing gradient ascent.
2. The **Critic** minimizes the TD error by adjusting $V_\phi(s)$ to better align with observed rewards.

b. Total Time Steps Analysis

These are the results of how the agent trains and interacts with the environment for three different timestep values: **40k, 100k, and 200k**.

Training Reward Summary for Different Timesteps

Training Trial	Timesteps	Training Reward Mean
Trial 1	40,000	-29.23
Trial 2	100,000	-13.73
Trial 3	200,000	-6.93

Table 8: Training reward comparison for different timesteps

Observations

Similar to the **PPO** algorithm, the **A2C** algorithm demonstrates improvement as rewards increase from **-50 to 0**. However, across all trials, the rewards stabilize around **0** and do not surpass this value, indicating that the agent has plateaued and is not learning the optimal policy.

This occurs because the agent attempts to **maximize the reward** using only minor back-and-forth movements, which causes it to become stuck at **0** rather than reaching the goal of **+100** reward. This behavior mirrors that observed with the PPO algorithm.

c. State-Dependent Exploration Analysis

The **SDE** parameter controls whether the state-dependent exploration mechanism is used during training.

- When set to **False**, the default exploration strategy is used (independent Gaussian noise). This approach is computationally cheaper and sufficient for standard tasks.
- When set to **True**, **SDE** replaces the standard noise in the policy with **state-dependent noise**, allowing for more structured and adaptive exploration based on the state. This is particularly beneficial for continuous action spaces but comes with higher computational cost. The noise is sampled once at the start of a rollout.

For testing this hyperparameter, I use **100k total timesteps**.

Mathematical Formulation

The policy network samples an action a from a Gaussian distribution with a mean $\mu_\theta(s)$ and state-dependent variance σ_s^2:

$$\pi_\theta(a \mid s) = \mathcal{N}(\mu_\theta(s), \sigma_s^2) \tag{33}$$

The action at time step t is defined as:

$$a_t = \mu(s_t, \theta_\mu) + \epsilon(s_t, \theta_\epsilon) \tag{34}$$

where $\epsilon(s_t, \theta_\epsilon)$ represents the state-dependent noise, modeled as:

$$\epsilon(s_t, \theta_\epsilon) = \theta_\epsilon \cdot s_t \tag{35}$$

The noise parameter θ_ϵ is sampled from a normal distribution:

$$\theta_\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{36}$$

Training Reward Summary with SDE

Training Trial	SDE	Training Reward Mean
Trial 1	False	-13.73
Trial 2	True	50.63

Table 9: Training reward comparison with and without State-Dependent Exploration (SDE)

Observations

Similar to **PPO**, we observe that without the **SDE** mechanism, training remains stable, but the agent does not explore sufficiently. As a result, it fails to learn the optimal policy and stabilizes near **0**.

With **SDE** enabled, the agent explores the environment more effectively, leading to improved performance. However, the training process is not entirely stable, with the training reward mean fluctuating around **50**. While the agent benefits from increased exploration, it still faces challenges in achieving a stable learning phase.

d. Entropy Coefficient Analysis

The entropy coefficient c_2 controls the weight of the entropy term in the **A2C** loss function. Entropy measures the randomness in the agent's action distribution. Adding an entropy bonus encourages the agent to explore more by preventing the policy from becoming overly deterministic early in training.

For testing this hyperparameter, I use **100k total timesteps**.

Full Objective Function

The A2C loss function incorporating entropy regularization is given by [4]:

$$L_t^{\mathsf{A2C}}(\theta, \phi) = \widehat{\mathbb{E}}_t \left[L_t^{\mathsf{PG}}(\theta) - c_1 L_t^{\mathsf{VF}}(\phi) + c_2 S[\pi_\theta](s_t) \right] \tag{37}$$

where:

$$L_t^{\mathsf{PG}}(\theta) = -\widehat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \widehat{A}_t \right] \tag{38}$$

$$L_t^{\mathsf{VF}}(\phi) = \widehat{\mathbb{E}}_t \left[\left(V_\phi(s_t) - V_t^{\mathsf{target}} \right)^2 \right] \tag{39}$$

$$S[\pi_\theta](s_t) = \frac{1}{2} \log(2\pi e \sigma^2) \tag{40}$$

Training Reward Summary with Different Entropy Coefficients

Training Trial	Entropy Coefficient	Training Reward Mean
Trial 1	0.001	46.24
Trial 2	0.01	55.42
Trial 3	0.1	55.16

Table 10: Training reward comparison with different entropy coefficients

Observations

Using **higher entropy coefficients** (0.01 and 0.1) in the three trials led to higher average rewards than the lowest entropy coefficient (0.001).

- **Trial 1** had a lower mean reward compared to **Trials 2 and 3**, which reached a mean reward of about **55**. - While all trials exhibited fluctuations and occasional drops into negative rewards, the trials with **higher entropy coefficients** rebounded more quickly and maintained higher peaks. - These results suggest that increasing entropy—thus encouraging **more exploration**—may help the policy escape **local minima** in the environment.

This highlights the importance of tuning the entropy coefficient to balance exploration and stability for improved learning performance.

e. Learning Rate Analysis

The learning rate is a hyperparameter that controls the step size of the gradient updates for both the policy and value networks, which share the same optimizer. By default, **A2C** uses the **RMSProp** optimizer with a learning rate of 7×10^{-4}.

For testing this hyperparameter, I use **100k total timesteps**.

Training Reward Summary with Different Learning Rates

Training Trial	Learning Rate	Training Reward Mean
Trial 1	7×10^{-6}	19.59
Trial 2	7×10^{-5}	54.72
Trial 3	0.0007	55.16

Table 11: Training reward comparison with different learning rates

Observations

- With the **smallest learning rate** (7×10^{-6}), the model struggled to improve, achieving an average reward of approximately **19.59**. - The **higher learning rates** (7×10^{-5} and 7×10^{-4}) both resulted in better performance, reaching average rewards close to **55**. - This suggests that a **slightly faster learning rate** helped the agent converge to a more effective policy. - The training reward progression shows some **volatility early in training**, but the model generally stabilizes to higher reward levels when using larger learning rates.

These results demonstrate that even **small changes in learning rate** can lead to **significant differences** in final performance.

f. GAE λ Analysis

In **A2C**, the **GAE** λ parameter controls how much weight is placed on higher-order returns when computing the advantage estimate. **Generalized Advantage Estimation (GAE)** combines multiple-step returns in a way that balances bias and variance:

- A **smaller** λ places more emphasis on **shorter horizons**, reducing bias but increasing variance. - A **larger** λ gives more weight to **longer horizons**, increasing bias but reducing variance. - When $\lambda = 1$, the algorithm follows the **default multi-step advantage** approach, relying entirely on multi-step returns rather than decaying their weight.

Training Reward Summary with Different GAE λ Values

Training Trial	GAE λ	Training Reward Mean
Trial 1	0.5	64.51
Trial 2	0.65	62.40
Trial 3	0.8	57.01
Trial 4	0.9	61.48
Trial 5	1.0	54.36

Table 12: Training reward comparison with different GAE λ values

Observations

- The results show that $\lambda = 0.5$ achieved the highest mean return (**64.51**), while $\lambda = 1.0$ resulted in the lowest mean reward (**54.36**). - This suggests that in this environment, placing **more weight on shorter returns (lower λ)** may lead to better performance. - The training reward progression indicates that **all trials experienced some volatility**, occasionally dropping to lower or negative returns. - However, by the end of training, most trials **recovered to prior reward levels**, reinforcing the idea that **higher variance** (smaller λ) is beneficial for advantage estimation in this environment.

These findings highlight the importance of tuning λ to balance bias and variance, ensuring stable and efficient learning.

g. Seeding

Here, I test the algorithm with different seeds to showcase the importance of **seeding**. Seeding controls the initialization of randomness in training processes, ensuring **reproducibility** by fixing the random number generators for neural network weights, environment dynamics, and exploration noise.

Training Reward Summary with Different Seeds

Training Trial	Seed	Training Reward Mean
Trial 1	0	61.19
Trial 2	5	55.88
Trial 3	10	52.71
Trial 4	25	51.41
Trial 5	42	54.74
Trial 6	50	54.53

Table 13: Training reward comparison with different seeds

Observations

- The training results indicate **less variability** compared to **PPO**, with **Trial 1 (seed = 0)** achieving the highest mean reward of **61.19**. - The remaining trials (**seeds in the range 5-50**) resulted in mean rewards between **50-55**, showing **lower but not significantly different** performance. - This suggests that the **A2C algorithm exhibits low sensitivity to initialization**, meaning the choice of seed does not drastically impact training outcomes. - The average rewards range from **51 to 61**, indicating that while the agent consistently

learns a successful policy, the final performance can vary by about **10 points** depending on the seed.

For the scope of this project, **seed 0** is selected as the **optimal choice**, while **seed 25** is identified as the **worst-performing seed**.

h. Conclusion

In the **MountainCar Continuous** environment, the agent only receives a positive reward upon reaching the goal. With the **A2C** algorithm, this results in a **sparse reward problem** (similar to **PPO**). **A2C** is an **on-policy** algorithm that performs a policy gradient update after each episode. Since reaching the goal by random actions is a rare event, the agent struggles to receive consistent learning signals. Even when it reaches the goal by chance, a single policy gradient update is often insufficient to maintain progress, causing the agent to get stuck without further learning signals.

The experiments show that both **A2C** and **PPO** tend to **plateau at suboptimal reward levels** when exploration is limited, with agents often converging around **0** despite occasional improvement. However, using mechanisms such as **State-Dependent Exploration (SDE)** and **higher entropy coefficients** improves exploration, allowing the policy to **escape local minima** and achieve higher mean rewards.

With optimal configurations: - Mean rewards reached up to **55** with enhanced exploration. - The highest performance of **64.51** was achieved using a **lower** λ for advantage estimation. - Slight increases in **learning rate** improved performance. - **Seed variability** played a minor role, affecting mean rewards by roughly **10 points**.

These findings emphasize the importance of **balancing exploration and exploitation** in reinforcement learning and highlight the **sensitivity of performance to hyperparameter tuning**, which must be carefully optimized for stable and efficient learning.

PPO and A2C Optimal Models

I will use the hyperparameters tuned based on the above experiments. For the rest of the hyperparameters, I will use the default values.

PPO Hyperparameters

```python
# Create and train the PPO model
env = gym.make("MountainCarContinuous-v0", render_mode="rgb_array")
logger_callback.episode_rewards = []


model = PPO(
    "MlpPolicy",
    env,
    learning_rate=3e-3,                                    # Change the learning rate
    n_steps=2048,                                          # Steps per rollout
    batch_size=64,                                         # Minibatch size
    n_epochs=10,                                           # Optimization epochs per update
    gamma=0.99,                                            # Discount factor
    clip_range=0.2,                                        # Clipping range
    ent_coef=0.01,                                         # Entropy coefficient
    vf_coef=0.5,                                           # Value function coefficient
    max_grad_norm=0.65,                                    # Gradient clipping
    gae_lambda=0.95,                                       # GAE lambda
    use_sde=True,                                          # Use generalized State Dependent Exploration (
    seed=42,                                               # Seeding
    verbose=0,                                             # Verbosity level
    device='cpu'                                           # Device
)

# Train the model with the callback
model.learn(total_timesteps=40_000, callback=logger_callback)
```

Figure 17: Optimized PPO Hyperparameters

A2C Hyperparameters

```python
# Create and train the A2C model in parallel enviroments
num_envs = 4
env = DummyVecEnv([
    lambda: Monitor(gym.make("MountainCarContinuous-v0", render_mode="rgb_array"))
    for _ in range(num_envs)
])
logger_callback.episode_rewards = []

model = A2C(
    "MlpPolicy",
    env,
    learning_rate=7e-4,                                    # Change the learning rate
    n_steps=5,                                             # Steps per rollout
    gae_lambda=0.5,                                        # GAE lambda
    ent_coef=0.1,                                          # Entropy coefficient
    vf_coef=0.5,                                           # Value function coefficient
    max_grad_norm=0.7,                                     # Gradient clipping
    use_rms_prop=True,                                     # Rmsprop or Adam
    use_sde=True,                                          # Use generalized State Dependent Exploration (gSDE)
    seed=0,                                                # Seeding
    verbose=0,                                             # Verbosity level
    device='cpu'                                           # Device
)

# Train the model with the callback
model.learn(total_timesteps=100_000, callback=logger_callback)
```

Figure 18: Optimized A2C Hyperparameters

PPO Result

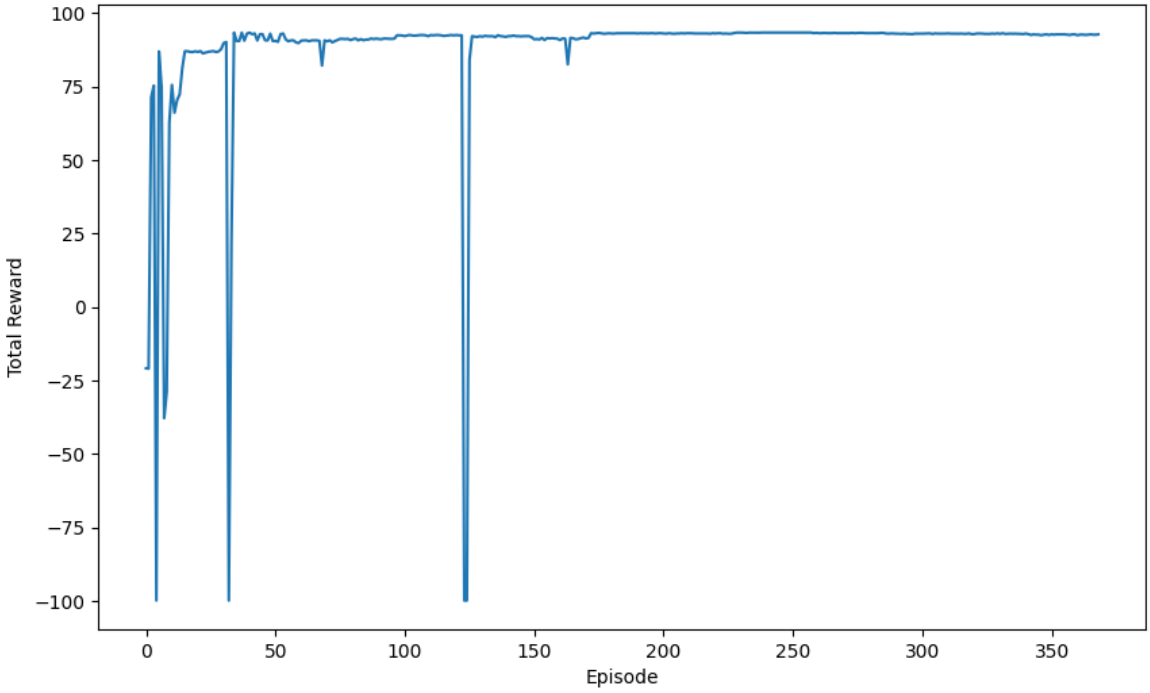

Figure 19: PPO Training Reward Progression

A2C Result

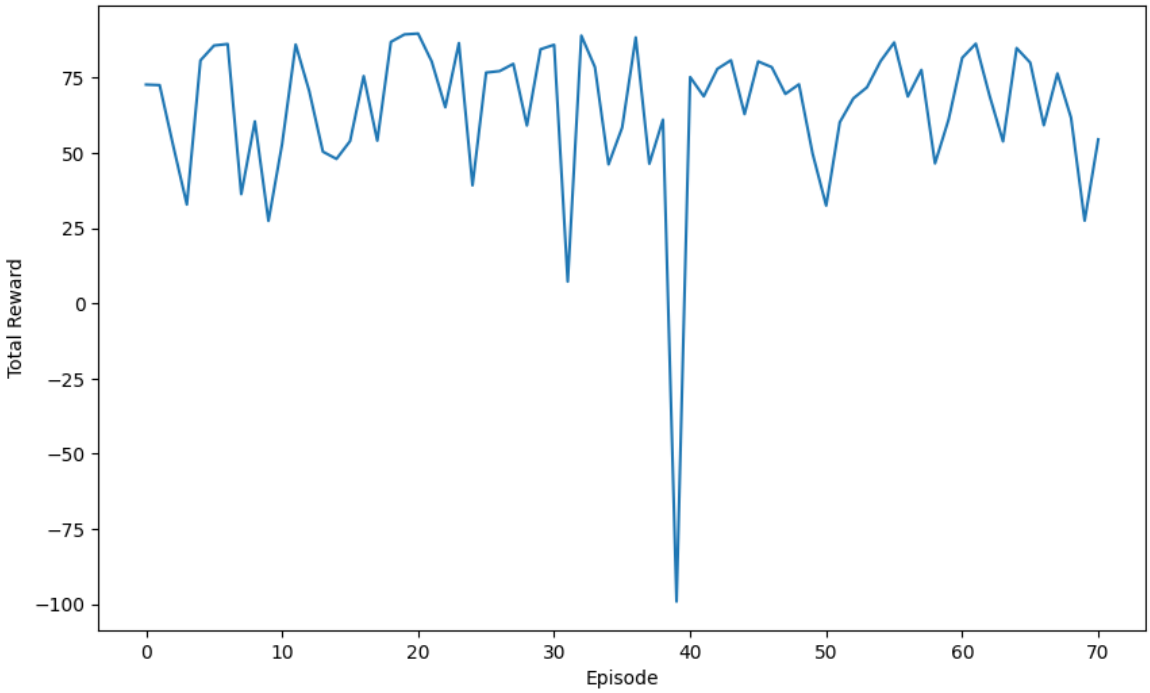

Figure 20: A2C Training Reward Progression

Bibliography

1. Foundational Reinforcement Learning Works

[1] Sutton, R. S., and Barto, A. G. (1998). **Reinforcement Learning: An Introduction (2nd Edition)**. MIT Press. Available online: http://incompleteideas.net/book/the-book-2nd.html.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). **Human-level control through deep reinforcement learning**. Nature, 518, 529-533.

2. Policy-Based RL Algorithms (PPO & A2C)

[3] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). **Proximal Policy Optimization Algorithms**. arXiv preprint arXiv:1707.06347. Available online: https://arxiv.org/abs/1707.06347.

[4] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... and Kavukcuoglu, K. (2016). **Asynchronous Methods for Deep Reinforcement Learning**. In International Conference on Machine Learning (ICML). Available online: https://arxiv.org/abs/1602.01783.

[5] Stable-Baselines3 Documentation. **PPO Implementation Details**. Available online: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html.

[6] Stable-Baselines3 Documentation. **A2C Implementation Details**. Available online: https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html.

3. Advanced Deep Q-Network (DQN) Techniques

[7] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). **Prioritized Experience Replay**. arXiv preprint arXiv:1511.05952. Available online: https://arxiv.org/abs/1511.05952.

[8] van Hasselt, H., Guez, A., and Silver, D. (2015). **Deep Reinforcement Learning with Double Q-learning**. arXiv preprint arXiv:1509.06461. Available online: https://arxiv.org/abs/1509.06461.

[9] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2015). **Dueling Network Architectures for Deep Reinforcement Learning**. arXiv preprint arXiv:1511.06581. Available online: https://arxiv.org/abs/1511.06581.

[10] Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2017). **Noisy Networks for Exploration**. arXiv preprint arXiv:1706.10295. Available online: https://arxiv.org/abs/1706.10295.

[11] Bellemare, M. G., Dabney, W., and Munos, R. (2017). **A Distributional Perspective on Reinforcement Learning**. arXiv preprint arXiv:1707.06887. Available online: https://arxiv.org/abs/1707.06887.

[12] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2017). **Rainbow: Combining Improvements in Deep Reinforcement Learning**. arXiv preprint arXiv:1710.02298. Available online: https://arxiv.org/abs/1710.02298.

4. Recurrent and Memory-Based RL Methods

[13] Hausknecht, M., and Stone, P. (2015). **Deep Recurrent Q-Learning for Partially Observable MDPs**. Proceedings of the 2015 AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents.

[14] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2017). **Deep Reinforcement Learning That Matters**. arXiv preprint arXiv:1709.06560. Available online: https://arxiv.org/abs/1709.06560.

5. Multi-Step Learning in RL

[15] Sutton, R. S., and Barto, A. G. (2018). **Reinforcement Learning: An Introduction (2nd Edition)**. MIT Press. (Chapter 7: Multi-Step Bootstrapping). Available online: http://incompleteideas.net/book/the-book-2nd.html.

[16] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2017). **Rainbow: Combining Improvements in Deep Reinforcement Learning**. arXiv preprint arXiv:1710.02298. Available online: https://arxiv.org/abs/1710.02298.

6. Transformers and Attention in RL

[17] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). **Attention Is All You Need**. Advances in Neural Information Processing Systems (NeurIPS). Available online: https://arxiv.org/abs/1706.03762.