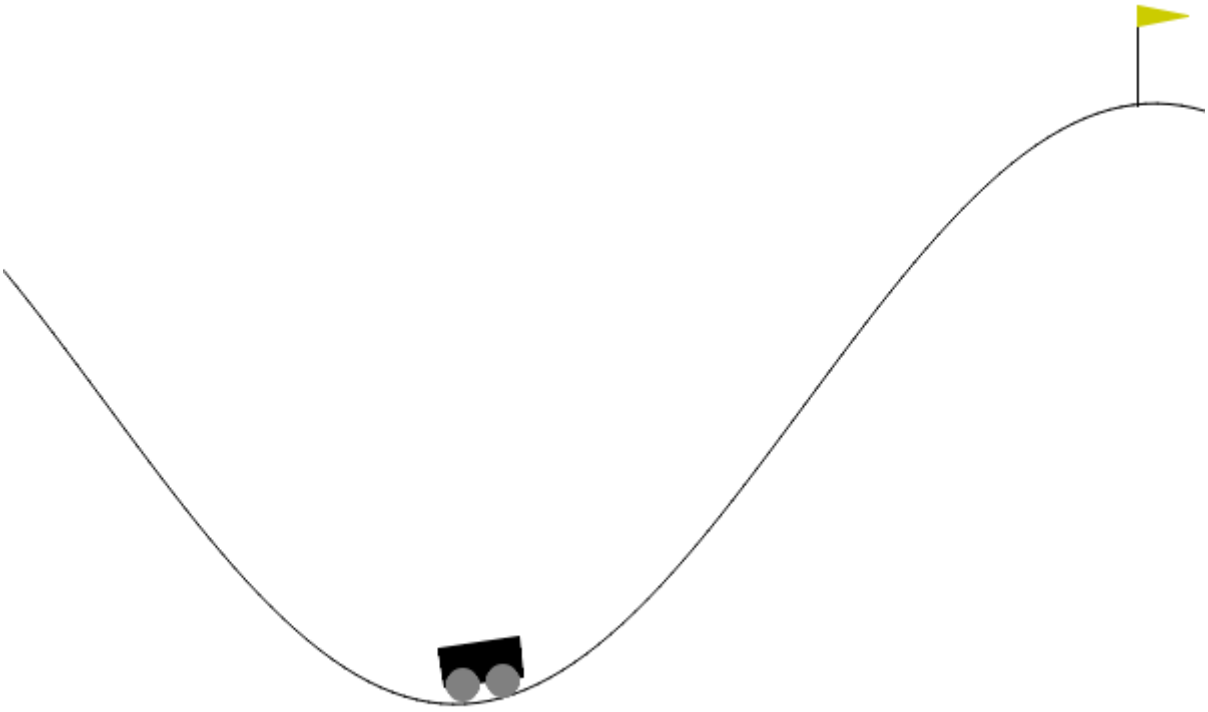
Reinforcement Learning Report

111550182

Task1: MountainCar (Classic-control)

Environment Description

For the first task, I selected the **MountainCar-v0** from classic-control. This environment involves a car situated between two hills, where the goal is to drive the car up the right hill to reach the flag. The challenge lies in the fact that the car's engine is not strong enough to drive directly up the hill, requiring the agent to build momentum by moving back and forth. The state space consists of two continuous variables: position and velocity, while the action space includes three discrete actions: push left, push right, or do nothing.



Algorithms Used

Two reinforcement learning algorithms were implemented and compared:

1. **Q-learning**: A model-free off-policy algorithm that updates the Q-values using the maximum future reward.
2. **SARSA**: A model-free on-policy algorithm that updates the Q-values based on the action actually taken by the agent.

Both algorithms utilized a discretized state space, dividing the continuous state variables into bins for simplicity.

Experimental Setup

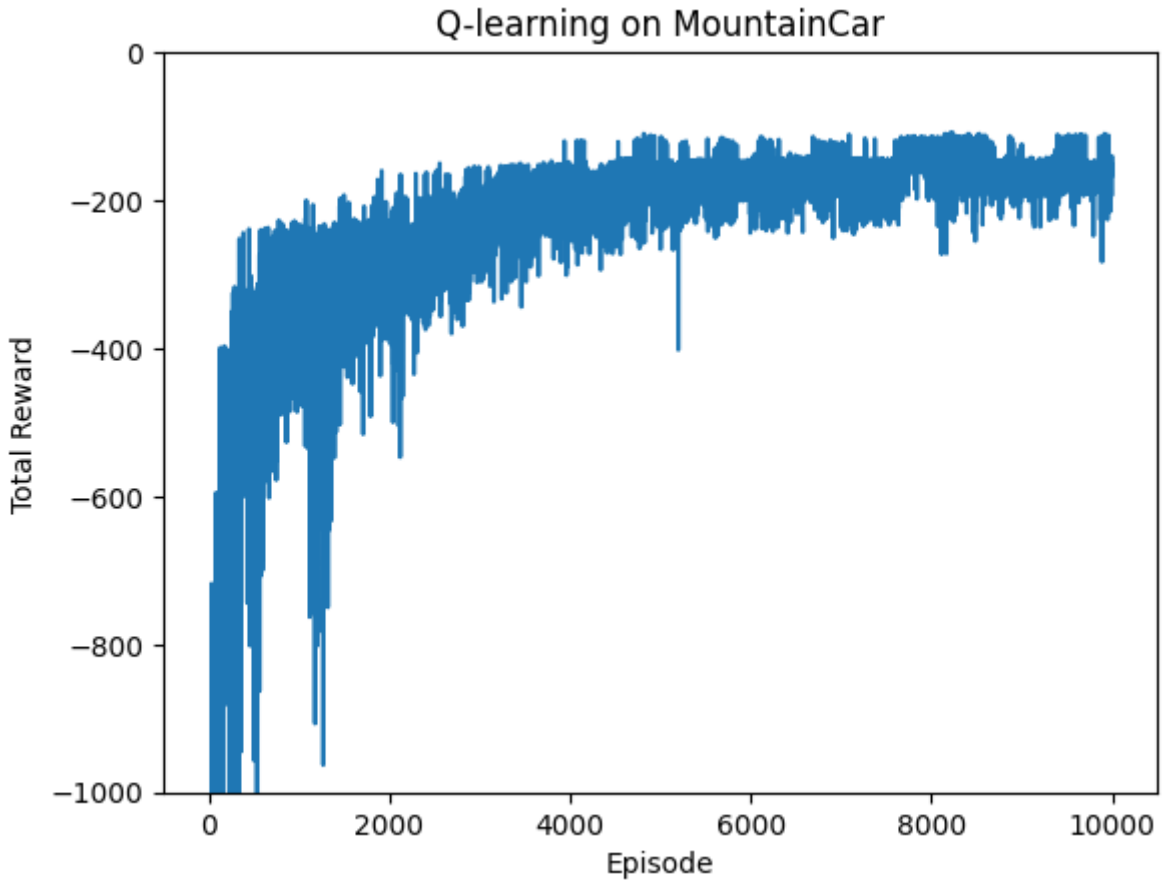
The experiments were conducted with the following fixed parameters unless otherwise specified:

- **Number of bins (n_bins)**: 40
- **Learning rate (alpha)**: 0.1
- **Discount factor (gamma)**: 0.99
- **Exploration rate (epsilon)**: 0.1
- **Episodes**: 10,000

The experiments explored the impact of varying key hyperparameters (`n_bins`, `alpha`, `gamma`, and `epsilon`) on the performance of the Q-learning algorithm. Additionally, the performance of SARSA was compared to Q-learning with fixed parameters.

Results

The goal was to achieve a reward greater than -200, which is considered a success in this environment.

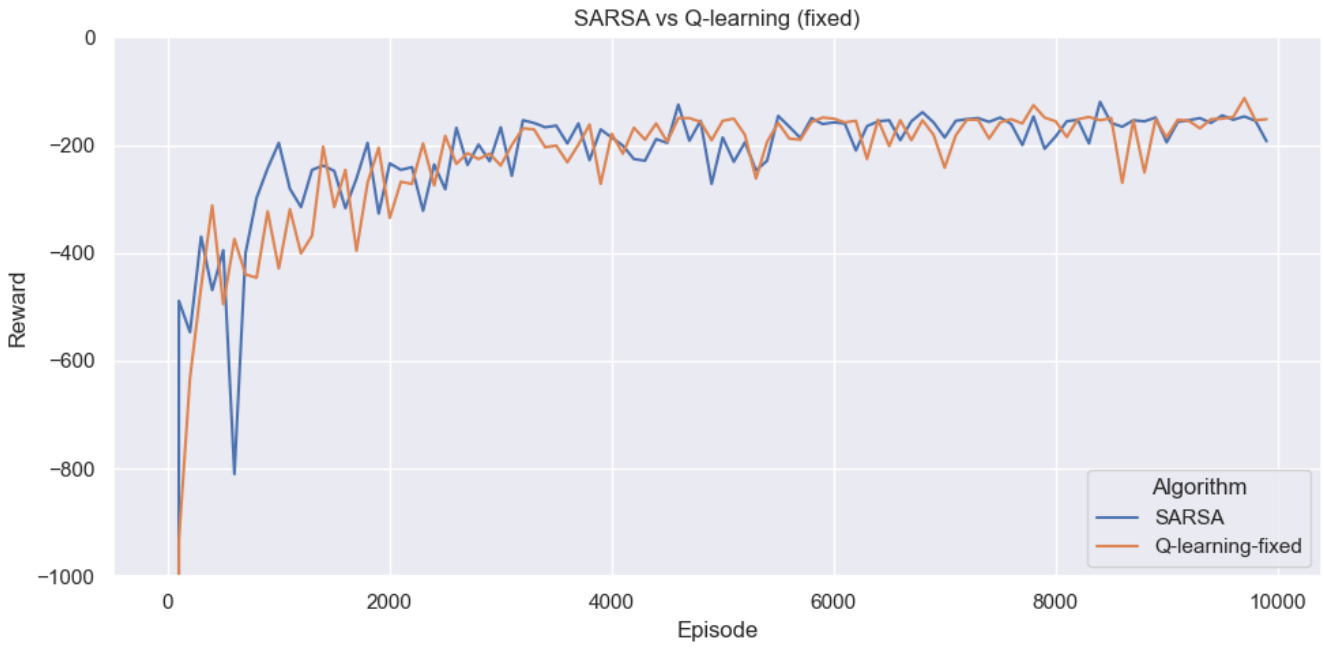


The results indicate that the Q-learning algorithm successfully learned to solve the MountainCar task, as the cumulative reward consistently exceeded -200 after sufficient training episodes. This demonstrates the agent's ability to build momentum and reach the goal effectively. The learning curve shows steady improvement, highlighting the algorithm's capability to adapt and optimize its policy over time.

1. Comparison of SARSA and Q-learning (fixed parameters)

The results showed that both algorithms converged to similar performance levels after 10,000 episodes, as seen in the figure below. However, SARSA exhibited slightly more variability in the early episodes. Although

Q-learning is generally considered to converge faster, no significant differences were observed in this case.



---

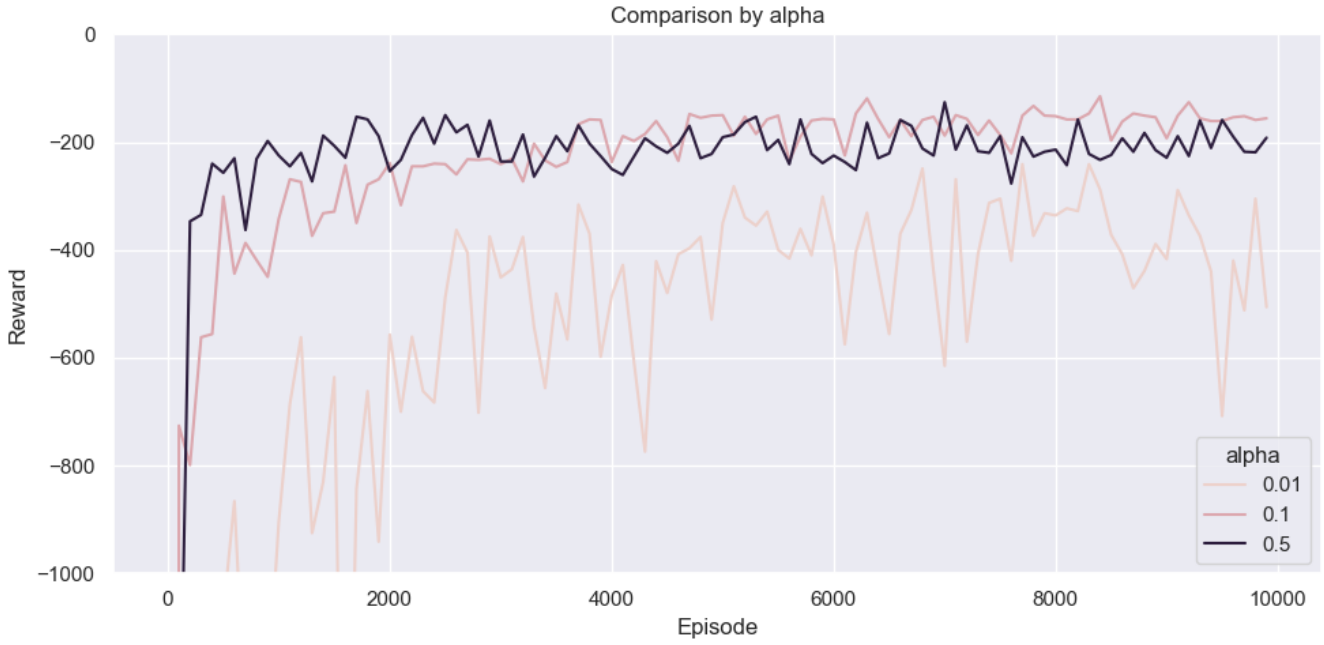
2. Effect of Hyperparameters on Q-learning

Number of bins (`n_bins`)

Increasing the number of bins improved performance, as finer discretization allowed for more accurate state representation. However, diminishing returns were observed beyond 40 bins.

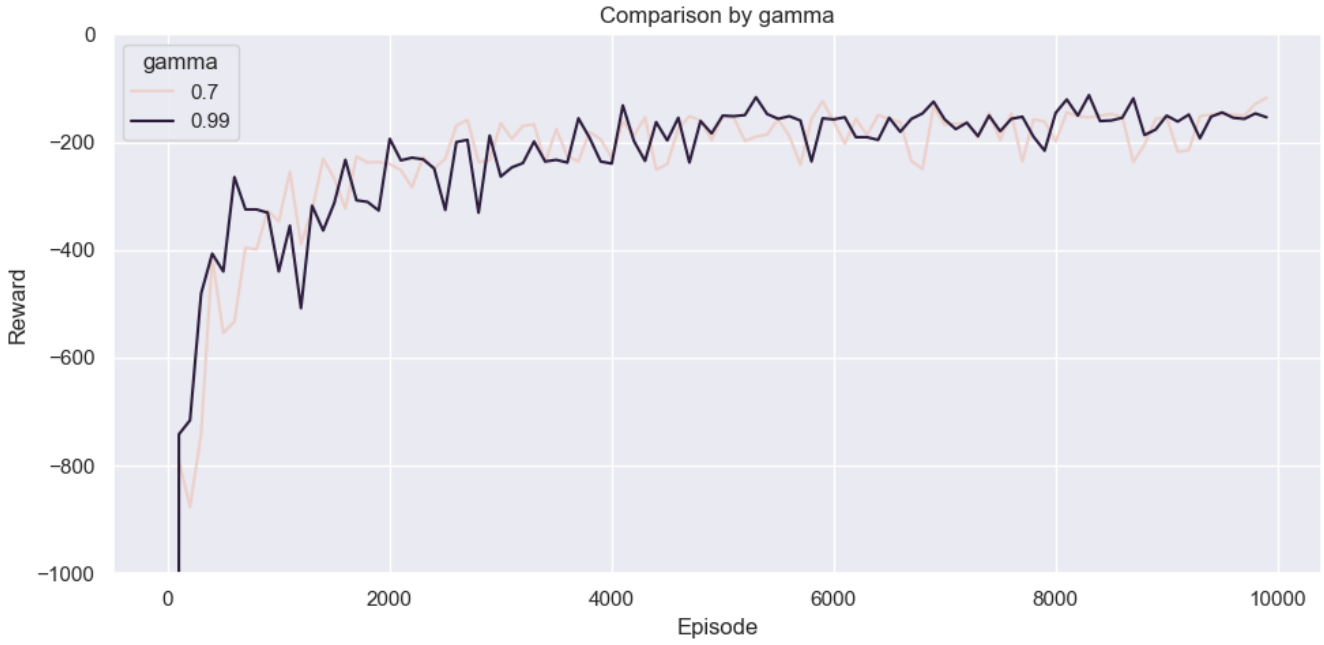
Learning rate (`alpha`)

A moderate learning rate (0.1) provided the best results. A very low learning rate (0.01) led to slow convergence, while a high learning rate (0.5) caused instability.



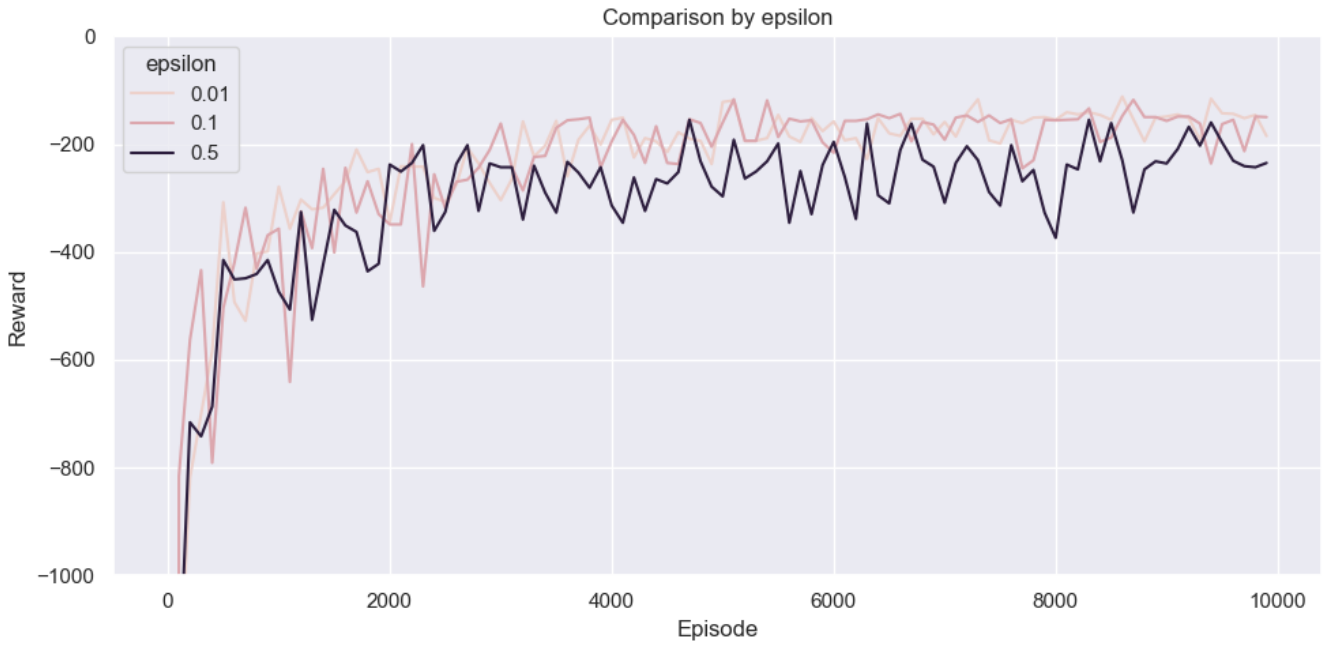
Discount factor (`gamma`)

A higher discount factor (0.99) resulted in better performance, as it prioritized long-term rewards. This highlights the importance of considering long-term rewards rather than focusing solely on immediate gains when climbing the hill. Such a strategy enables the agent to build the necessary momentum to achieve the goal effectively.



Exploration rate (`epsilon`)

A lower exploration rate (0.01) led to faster convergence but risked suboptimal policies, while a higher rate (0.5) delayed convergence.



Discussion

From these experiments, it was observed that:

- Both SARSA and Q-learning are effective for solving the MountainCar task, with Q-learning being slightly more stable.
- Hyperparameter tuning significantly impacts the performance of Q-learning. Proper selection of `n_bins`, `alpha`, `gamma`, and `epsilon` is crucial for achieving optimal results.

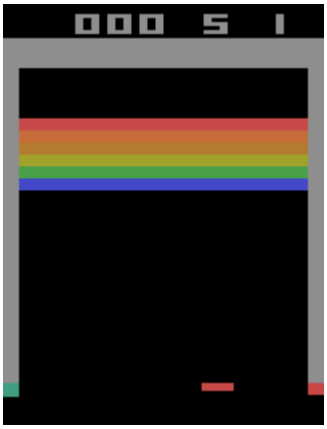
Remaining Questions

- How would the performance of these algorithms change with function approximation methods (e.g., neural networks) instead of discretization?
- Would alternative exploration strategies, such as epsilon decay improve performance?

Task2: Atari- Breakout

Environment Description

For the second task, I selected the **Breakout** environment from the Atari. In this environment, the agent controls a paddle at the bottom of the screen and aims to hit a ball to break bricks at the top. The game ends when all bricks are broken or the ball falls below the paddle. The action space includes discrete actions such as moving the paddle left, right, or staying still.



Algorithm Used

The algorithm implemented for this task is the **Deep Q-Network (DQN)**, as some described in the paper *"Playing Atari with Deep Reinforcement Learning"*(2013). This approach combines Q-learning with deep neural networks to approximate the Q-value function for high-dimensional state spaces like Atari games.

Key Features of the Algorithm

1. **Experience Replay**: A replay buffer stores past experiences (state, action, reward, next state, done), which are sampled randomly during training. This breaks the temporal correlation between consecutive experiences and improves sample efficiency.
2. **Target Network**: A separate target network is used to compute the target Q-values, which stabilizes training by reducing oscillations.
3. **Frame Stacking**: Four consecutive frames are stacked to provide temporal context, enabling the agent to infer the ball's velocity and direction.
4. **Reward Clipping**: Rewards are clipped to [-1, 1] to stabilize training and prevent large gradients.
5. **Epsilon-Greedy Exploration**: The agent balances exploration and exploitation by selecting random actions with a decaying probability (`epsilon`).

Implementation Details

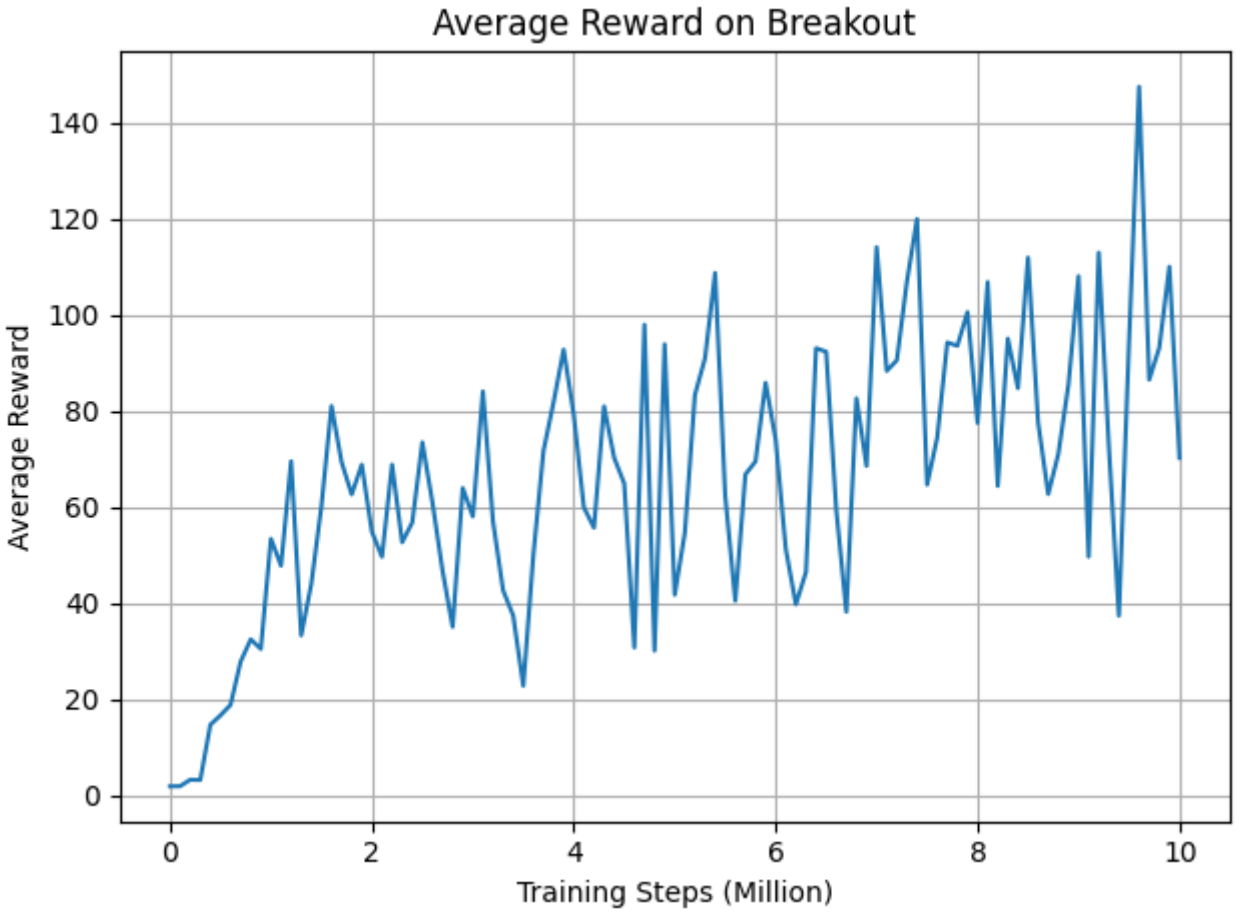
The implementation closely follows the approach outlined in the paper:

- **Neural Network Architecture**: The DQN consists of convolutional layers followed by fully connected layers. The input is a stack of four grayscale frames, and the output is the Q-value for each possible action.
    - **Convolutional Layers**:
        - First layer: 16 filters of size 8x8 with a stride of 4, followed by ReLU activation.

        - Second layer: 32 filters of size 4x4 with a stride of 2, followed by ReLU activation.
    - **Fully Connected Layers**:
        - One hidden layer with 256 units and ReLU activation.
        - Output layer with one unit per action (e.g., 4 actions for Breakout).
- **Hyperparameters**:
    - Batch size: 32
    - Replay buffer size: 100,000
    - Discount factor (`gamma`): 0.99
    - Learning rate: 1.25e-4
    - Epsilon decay: From 1.0 to 0.01 over 500,000 steps
    - Target network update frequency: Every 10,000 steps
- **Training**: The agent was trained for 10 million steps, with the Q-network updated every 4 steps and the target network updated periodically.

Results

The training process was evaluated by plotting the average reward over time. The figure below shows the agent's performance improving steadily as training progresses. The agent achieves an average reward of over 100 after 10 million steps, demonstrating its ability to learn an effective policy for the Breakout environment.



Discussion

The experiments with the Breakout environment provided several key insights into the effectiveness and limitations of the Deep Q-Network (DQN) algorithm:

Key Learnings

1. **Effectiveness of DQN Components**:
    - The use of **Experience Replay** and a **Target Network** significantly stabilized training, allowing the agent to learn effective policies directly from raw pixel inputs.
    - **Frame Stacking** enabled the agent to capture temporal dependencies, such as the ball's velocity and direction, which are critical for success in Breakout.
    - **Reward Clipping** ensured stable learning by normalizing the reward scale.
2. **Learning Dynamics**:
    - The agent demonstrated steady improvement, achieving an average reward of over 100 after 10 million steps. However, the learning curve exhibited fluctuations, highlighting the challenges of balancing exploration and exploitation in high-dimensional environments.
3. **Time-Intensive Training**:
    - One major limitation observed was the significant time required for training. The DQN algorithm required 10 million steps to achieve optimal performance, which is computationally expensive. This highlights the need for more sample-efficient methods.

Remaining Questions

- How would the performance change with different hyperparameter settings, such as learning rate, epsilon decay schedule, or replay buffer size?
- Could alternative architectures, such as dueling DQN or double DQN, further improve performance?

References

1. V. Mnih, et al. (2013). *Playing Atari with Deep Reinforcement Learning*. arXiv:1312.5602
2. OpenAI Gym
3. Atari environment usage
4. Deep Q-learning Code Reference

Video Link

A video demonstrating the agent's performance.

Mountain Car

Breakout

Github Link

here