REINFORCEMENT LEARNING

Mastering MountainCar-v0: A Comprehensive Exploration of Reinforcement Learning Algorithms

Pavlos Ntais

Contents

1 Mountain Car

The **MountainCar-v0** environment is a classic reinforcement learning problem where the goal is to drive a car up a steep hill.

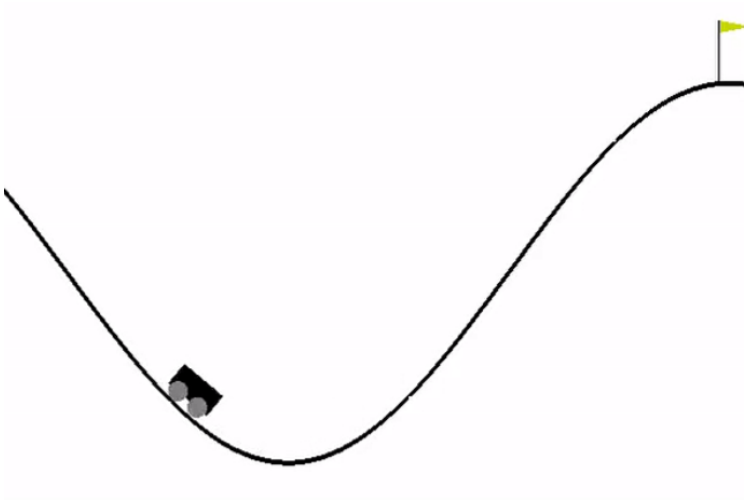


Figure 1: Visualization of the **MountainCar-v0** environment.

The environment is described by the following spaces and reward signals:

1.1 Observation Space

The observation space is represented by the `[position, velocity]` vector, where:

- **Position** (`x`): The position of the car along the x-axis, with a range of $[-1.2, 0.6]$ meters.
- **Velocity** (`v`): The velocity of the car, with a range of $[-0.07, 0.07]$ meters/second.

1.2 Action Space

The action space is discrete and consists of the following deterministic actions:

- **0**: Accelerate to the left.
- **1**: Do not accelerate.
- **2**: Accelerate to the right.

1.3 Reward Signals

- A reward of -1 is given for each time step until the car reaches the goal.
- The goal is achieved when the car's position is greater than or equal to 0.5.

The episodic nature of the environment ensures that the agent is penalized for taking more steps to reach the goal, encouraging **faster** solutions.

2 Reinforcement Learning

In Reinforcement Learning, we have an agent who exists in an environment and takes actions. When agents take actions, the environment sends back observations (in the form of states). We also get a reward back from the interaction; it's feedback to measure the success of the agent's actions.

Reward

A crucial concept in Reinforcement Learning is the *discounted total rewards*. The goal of the agent is to maximize its cumulative reward over time, where future rewards are discounted by a factor γ (gamma). This discount factor ensures that immediate rewards are prioritized over distant future rewards. The discounted total reward at time t can be expressed as:

$$R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

- R_t: **Total** discounted reward at time t,
- r_t: Reward received at time t,
- γ: Discount factor, where $0 < \gamma < 1$.

Q-function

The *Q-function* estimates the expected cumulative reward of taking an action a in a given state and following an optimal policy thereafter. These estimates are known as *Q-Values*. A higher Q-value for a state-action pair (s, a) indicates that action a is expected to yield better long-term results at state s.

Q-values are learned iteratively through a process called *Q-Learning*, a model-free, off-policy algorithm. The Q-Learning update rule adjusts the current Q-value estimate by incorporating the observed reward and maximum Q-value of the next state s' across **all** possible actions a'. This can be expressed mathematically as:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

- $Q(s, a)$: Current estimate of the Q-value for state s and action a,
- α: Learning rate, controlling the step size of updates,
- r: Reward received after taking action a in state s,
- γ: Discount factor, where $0 < \gamma < 1$, which prioritizes immediate rewards over future rewards,
- s': The state resulting from taking action a in state s,
- $\max_{a'} Q(s', a')$: The maximum Q-value for all actions a' in the next state s'.

Policy

Policy π defines the agent's behavior by mapping each state s to an action a. The goal is to find the optimal policy π^* that maximizes the expected cumulative reward.
Using the Q-function, the optimal policy can be derived by selecting the action a that maximizes the Q-value for a given state s. Mathematically, the optimal policy is defined as:

$$\pi^*(s) = \arg\max_a Q(s, a)$$

- $\pi^*(s)$: The optimal action for state s,
- $Q(s, a)$: The Q-value for taking action a in state s.

This approach ensures that the agent takes the action with the highest expected long-term reward in each state, leading to an optimal decision-making strategy.

3 Deep Q-Network (DQN)

3.1 Overview

Deep Q-Networks (DQNs)[1] combine traditional Q-Learning with deep neural networks to address the scalability issues of traditional Q-Learning in high-dimensional state and action spaces. In such environments, the number of possible state-action pairs (s, a) is too large to maintain a separate estimate for each pair. To overcome this, DQN uses a neural network parameterized by θ to approximate the Q-values, denoted as $Q(s, a; \theta)$.
Instead of directly updating individual Q-values, DQN optimizes the **parameters** θ of the network by minimizing the loss function defined as:

$$L(s, a|\theta_i) = \Big(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)\Big)^2$$

- $Q(s, a; \theta_i)$: The Q-value predicted by the network for state s and action a,
- r: The reward received,
- γ: Discount factor, where $0 < \gamma < 1$,
- $\max_{a'} Q(s', a'; \theta_i)$: The maximum Q-value for the next state s' over all actions a'.

The network parameters are updated using gradient descent as follows:

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta L(\theta_i)$$

where α is the learning rate.

Training DQNs can be challenging due to instability and divergence, as the same network generates both the target Q-values and the predictions. To address these issues, DQN employs two key techniques:

1. **Experience Replay:** Experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a replay memory $\mathcal{D}$. During training, batches of experiences are sampled uniformly from $\mathcal{D}$, breaking the temporal correlation between consecutive updates and improving sample efficiency.
2. **Target Networks:** A separate target network $\hat{Q}$ is used to generate the target Q-values. The parameters of the target network are updated less frequently, decoupling the targets from the updates of the main network and improving stability.

To be exact, the following is the algorithm used in the Deep Q-Learning process:

Algorithm 1 Deep Q-Network (DQN)

Require: MDP (S, A, P, R, γ), replay memory $\mathcal{M}$, number of iterations T, minibatch size n, exploration probability $\epsilon \in (0, 1)$, a family of deep Q-networks $Q_\theta : S \times A \to \mathbb{R}$, an integer T_{target} for updating the target network, and a sequence of stepsizes $\{\alpha_t\}_{t \geq 0}$

Initialize the replay memory $\mathcal{M}$ to be empty.
Initialize the Q-network with random weights θ.
Initialize the weights of the target network with $\theta^* = \theta$.
Initialize the initial state S_0.
for $t = 0, 1, \ldots, T$ **do**
    With probability ϵ, choose A_t uniformly at random from A, and with probability $1 - \epsilon$, choose A_t such that $Q_\theta(S_t, A_t) = \max_{a \in A} Q_\theta(S_t, a)$.
    Execute A_t and observe reward R_t and the next state S_{t+1}.
    Store transition (S_t, A_t, R_t, S_{t+1}) in $\mathcal{M}$.
    Sample random minibatch of transitions $\{(s_i, a_i, r_i, s_i')\}_{i \in [n]}$ from $\mathcal{M}$.
    for each $i \in [n]$ **do**
        Compute the target $Y_i = r_i + \gamma \max_{a \in A} Q_{\theta^*}(s_i', a)$.
    end for
    Update the Q-network: Perform a gradient descent step

$$\theta \leftarrow \theta - \alpha_t \cdot \frac{1}{n} \sum_{i \in [n]} [Y_i - Q_\theta(s_i, a_i)] \cdot \nabla_\theta Q_\theta(s_i, a_i).$$

    Update the target network: Update $\theta^* \leftarrow \theta$ every T_{target} steps.
end for
Define policy π as the greedy policy with respect to Q_θ.

Ensure: Action-value function Q_θ and policy π.

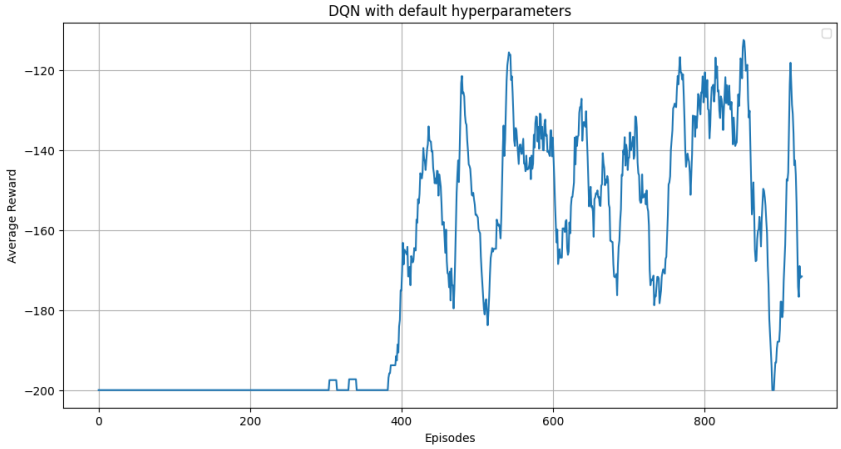
3.2 Results

The following results were obtained using the Deep Q-Network (DQN) algorithm. The training was performed with the following hyperparameters:

Table 1: Hyperparameters for DQN Training

Hyperparameter	**Value**
Learning Rate	1×10^{-3}
Discount Factor (γ)	0.99
Initial Epsilon (ϵ_{start})	1.0
Epsilon Decay Rate	0.995
Minimum Epsilon (ϵ_{end})	0.05
Batch Size	128
Replay Buffer Size	10,000
Target Network Update Frequency	Every 10 steps
Number of Episodes	930

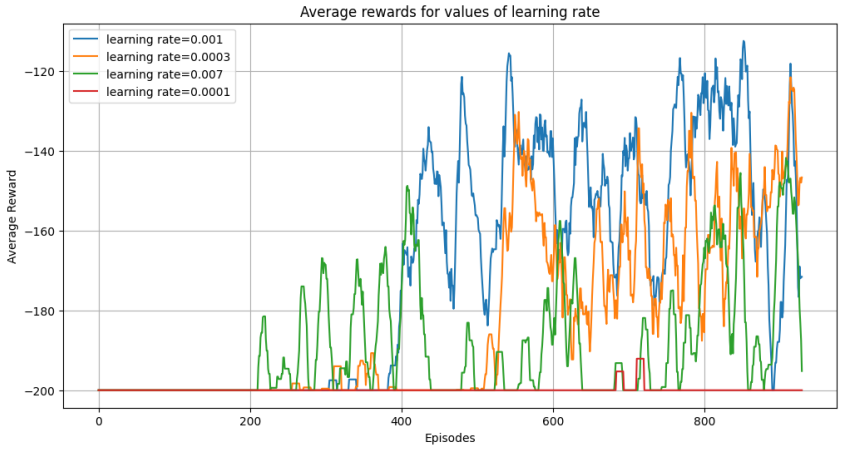
Below is the plot of the results, showing the performance over the training episodes:



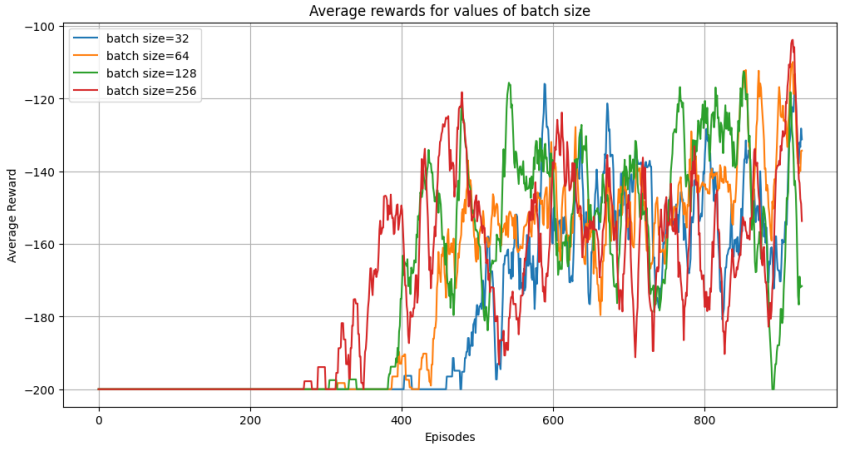
The episodic reward increases over time, indicating improved decision-making by the agent as the training progresses. The effect of epsilon decay is also visible as the exploration rate decreases, leading to more exploitation of the learned policy.

3.3 Sensitivity Study

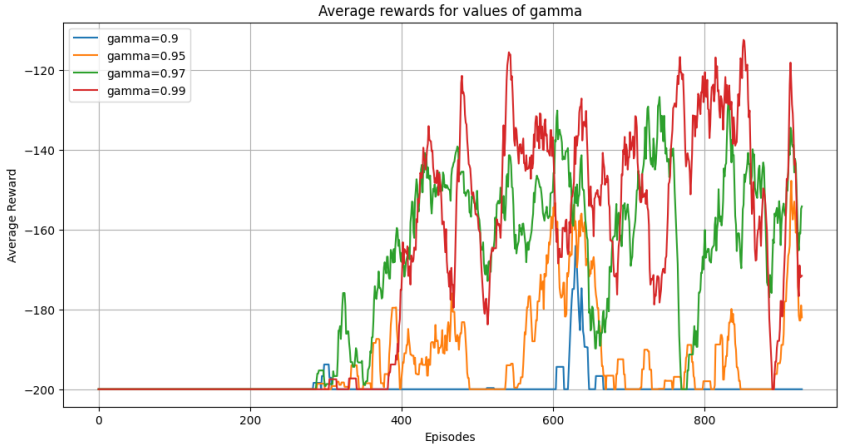
3.3.1 Learning Rate



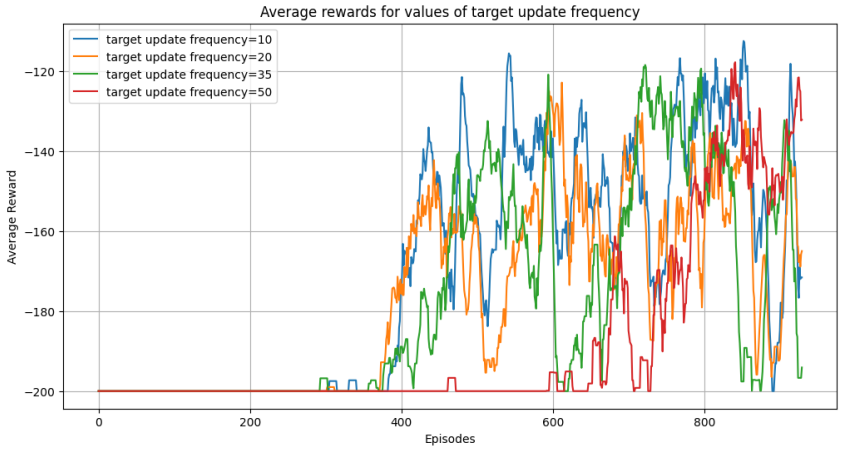
3.3.2 Batch Size



3.3.3 Gamma



3.3.4 Target Update Frequency



4 Noisy Networks

4.1 Overview

Noisy networks for exploration[2] is an **advanced** reinforcement learning technique designed to enhance exploration during the learning process of DQNs by adding trainable noise to the network's parameters. Traditional approaches, such as epsilon-greedy exploration, often struggle to balance exploration and exploitation effectively; this technique addresses this challenge, enabling parameter-driven exploration that adapts to the environment.

In a standard neural network, the output of a fully connected (linear) layer is computed as:

$$y = Wx + b, \tag{1}$$

where W and b are the weight matrix and bias vector, respectively, and x is the input.

In a noisy linear layer, the weights and biases are **perturbed** with trainable noise parameters:

$$y = (W + \sigma_W \odot \epsilon_W)x + b + \sigma_b \odot \epsilon_b, \tag{2}$$

where:

- σ_W and σ_b are learnable parameters controlling the scale of noise for weights and biases.
- ϵ_W and ϵ_b are random noise variables, typically sampled from a standard Gaussian distribution.
- $\odot$ denotes element-wise multiplication.

To reduce the number of random variables and improve computational efficiency, *factorized Gaussian noise* is used. The noise for each element of the weight matrix is computed as:

$$\epsilon_{ij} = f(\epsilon_i)f(\epsilon_j), \tag{3}$$

where ϵ_i and ϵ_j are independent Gaussian noise variables, and $f(x) = \text{sign}(x)\sqrt{|x|}$ ensures the distribution remains centered with unit variance.

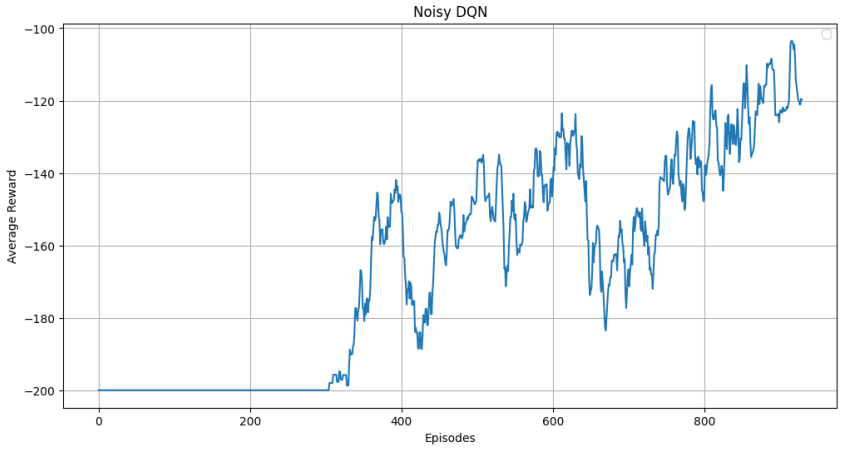
4.2 Results

The following results were obtained using the **Noisy** DQN algorithm. The training was performed with the following hyperparameters:

Table 2: Hyperparameters for Noisy DQN Training (same as DQN)

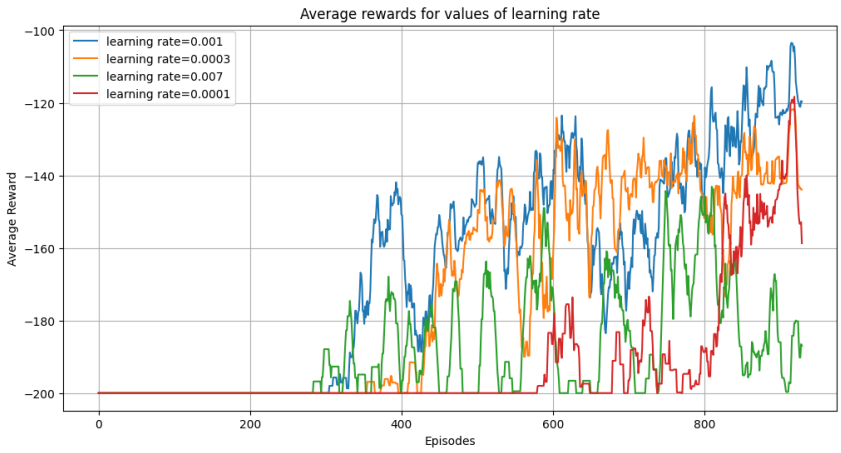
Hyperparameter	**Value**
Learning Rate	1×10^{-3}
Discount Factor (γ)	0.99
Initial Epsilon (ϵ_{start})	1.0
Epsilon Decay Rate	0.995
Minimum Epsilon (ϵ_{end})	0.05
Batch Size	128
Replay Buffer Size	10,000
Target Network Update Frequency	Every 10 steps
Number of Episodes	930

Below is the plot of the results, showing the performance over the training episodes:

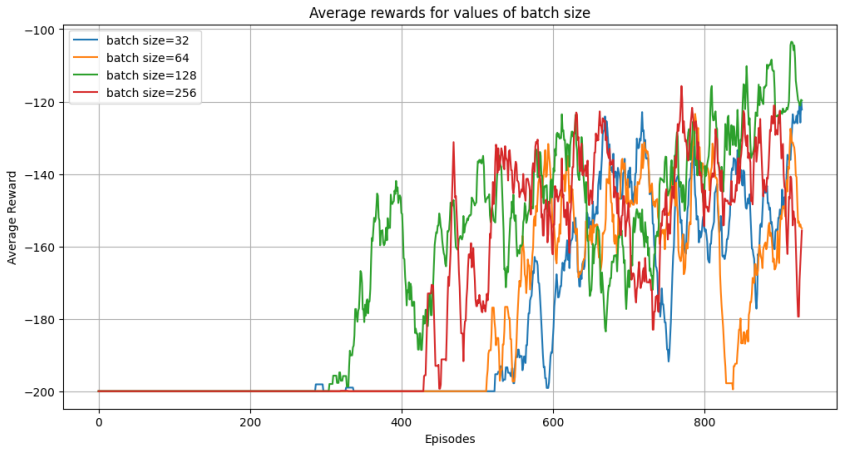


4.3 Sensitivity Study

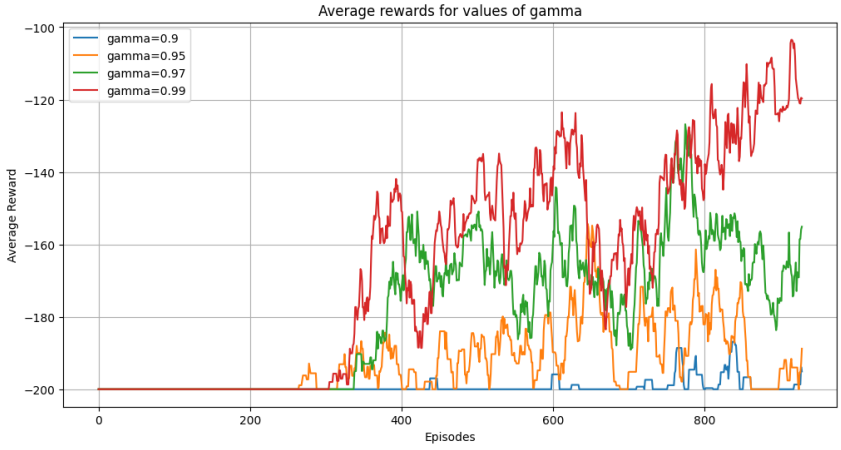
4.3.1 Learning Rate



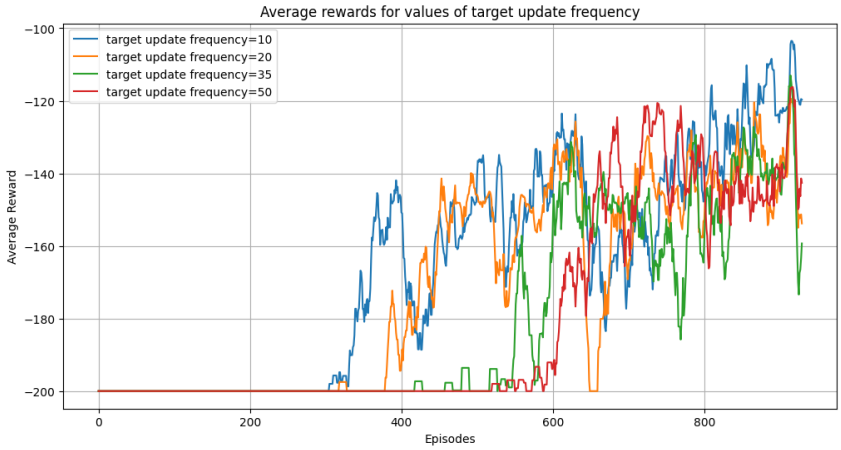
4.3.2 Batch Size



4.3.3 Gamma



4.3.4 Target Update Frequency



5 Deep Recurrent Q-Network (DRQN)

5.1 Overview

Deep **Recurrent** Q-Networks (DRQNs)[3] extend the capabilities of DQNs by incorporating recurrent layers, enabling them to tackle Partially Observable Markov Decision Processes (POMDPs). Unlike DQNs, which rely on stacked frames to infer the state of the environment, DRQNs use Long Short-Term Memory (LSTM) units to integrate information across time, effectively capturing temporal dependencies and dealing with incomplete observations.

DRQN extends DQN by replacing the first fully connected post-convolutional layer with an LSTM layer. This architecture enables DRQN to learn policies that are robust to missing or incomplete observations.

DRQN modifies the training process to accommodate recurrent updates. Two primary update strategies are used:

- **Bootstrapped Sequential Updates:** Training is performed sequentially over entire episodes, preserving the hidden state across timesteps. This allows the LSTM to capture long-term dependencies but introduces correlations that may reduce training stability.
- **Bootstrapped Random Updates:** Training starts at random points within episodes and proceeds for a short sequence of timesteps, and the LSTM's hidden state is reset at the beginning of each update. This method improves training stability but sacrifices long-term memory retention.

By using these strategies, DRQN learns to approximate the underlying system state more effectively than traditional DQNs, making it well-suited for environments with partial observability.

For this implementation, we will be using the **Bootstrapped Random Updates** method due to the relatively short-term dependencies of the MountainCar-v0 environment.

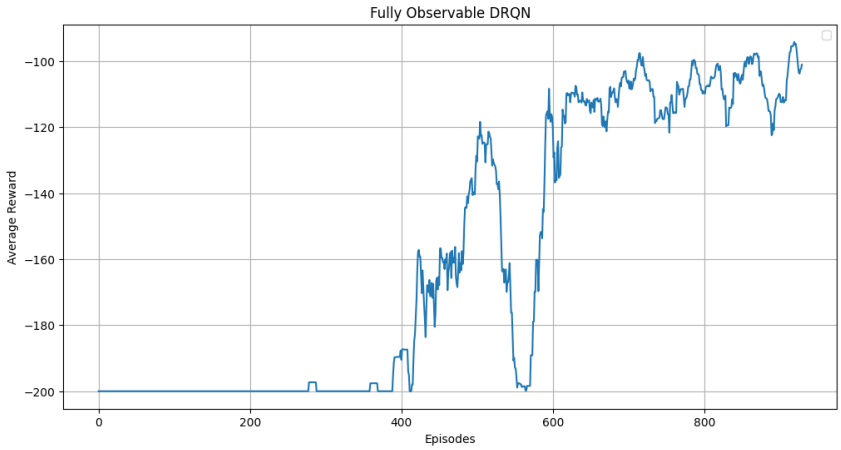
5.2 Results

The following results were obtained using the Deep Q-Network (DQN) algorithm. The training was performed with the following hyperparameters:

Table 3: Hyperparameters for DRQN Training

Hyperparameter	**Value**
Learning Rate	1×10^{-3}
Batch Size	32
Replay Buffer Size	100,000
Minimum Episode Number	32
Number of Episodes	930
Target Network Update Frequency	Every 10 steps
Initial Epsilon (ϵ_{start})	0.5
Minimum Epsilon (ϵ_{end})	0.01
Epsilon Decay Rate	0.995
Tau	0.05
Max Steps per Episode	200
Random Update	True
Lookup Step	30
Max Episode Number	100
Max Episode Length	200

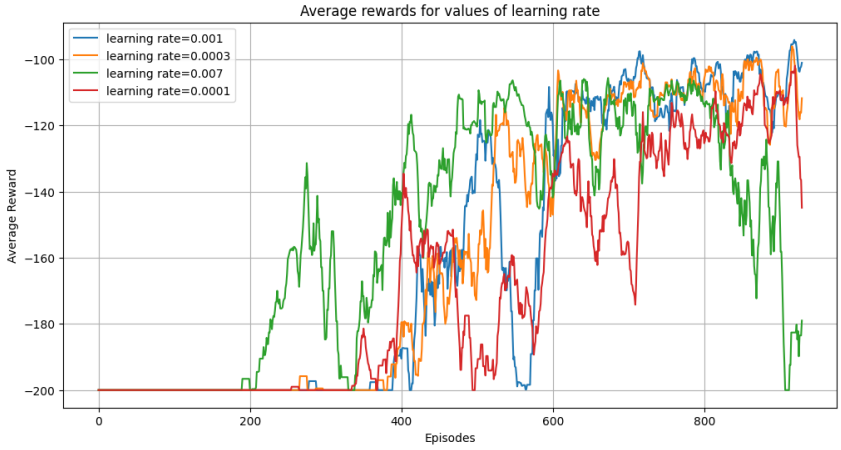
Below is the plot of the results, showing the performance over the training episodes:



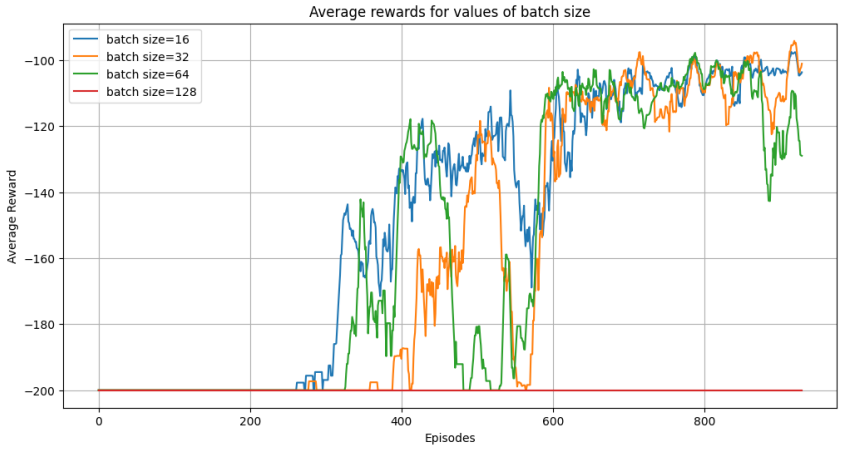
We can see the DRQN agent is performing better than the DQN agent, as rewards are higher and more consistent in the later phase of the training.

5.3 Sensitivity Study

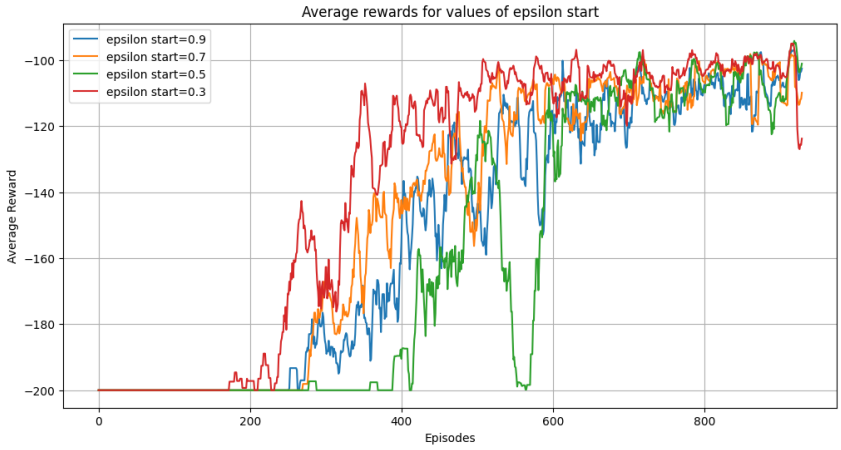
5.3.1 Learning Rate



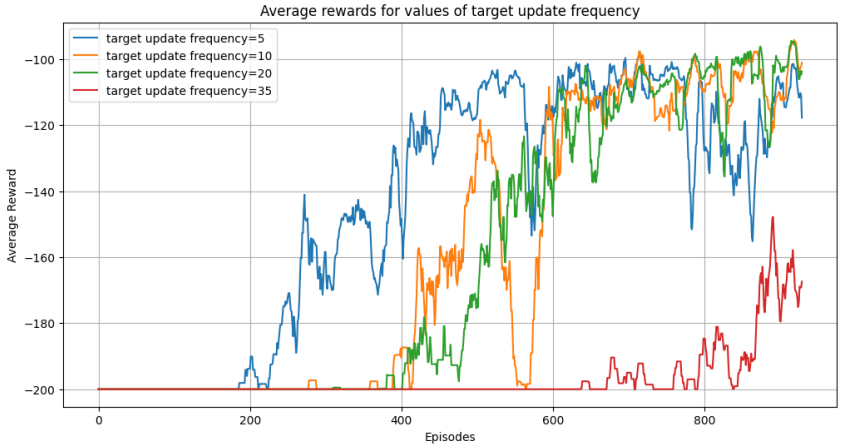
5.3.2 Batch Size



5.3.3 Epsilon Start



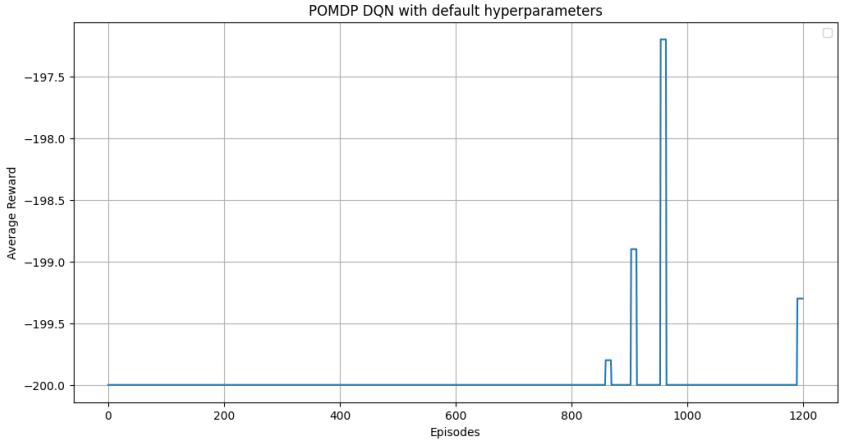
5.3.4 Target Update Frequency



5.4 Partially Observable (POMDP) MountainCar

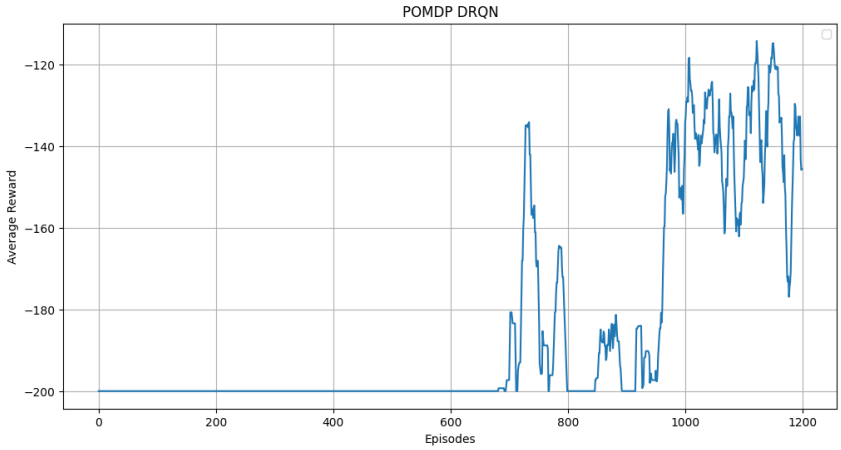
To simulate partial observability, we modify the environment so that the agent receives only the position x as its observation, with the velocity v hidden. The resulting problem is a Partially Observable Markov Decision Process (POMDP), where the agent must infer the missing velocity information from a history of observations.

We first test the POMDP environment on the classical DQN agent. Here are the results:



We can clearly see that the DQN agent failed to converge to an acceptable solution and cannot learn a robust policy.

Now we will test the **DRQN** agent on the same environment. Here are the results:



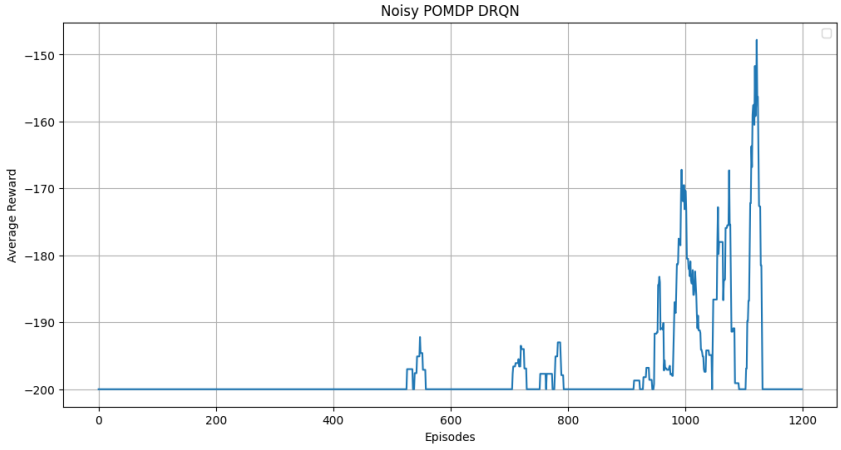
The results now are more satisfactory, DRQN leverages its recurrent structure to maintain a memory of past observations, thereby compensating for the missing velocity information, making the agent much more robust.

Noisy POMDP Environment

To further challenge the agent, we introduce noise into the observed position. More specifically, the observation x_{obs} is given by

$$x_{\text{obs}} = x + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{4}$$

where σ^2 is the variance of the Gaussian noise. This modification not only renders the environment partially observable but also corrupts the available information, making it even more challenging for the agent to learn a robust policy.



Under these noisy conditions, the classical DRQN's performance deteriorates further, but is still able to make progress in learning the environment.

6 Proximal Policy Optimization (PPO)

6.1 Overview

In policy gradient methods, we aim to update the policy parameters θ to maximize the expected return. Proximal Policy Optimization (PPO)[4] is a policy gradient method that introduces a surrogate objective function that limits the size of the policy update to stay within a "trust region." One common formulation of the PPO objective is the clipped surrogate objective:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min\Big(r_t(\theta)\, \hat{A}_t,\ \text{clip}\big(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon\big)\, \hat{A}_t \Big) \right], \tag{5}$$

where the probability ratio is defined as

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \tag{6}$$

and:

- $\pi_\theta(a_t \mid s_t)$ is the probability of taking action a_t in state s_t under the current policy,
- $\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ is the probability under the previous policy,
- $\hat{A}_t$ is an estimator of the advantage function at time step t, and
- ϵ is a hyperparameter that sets the threshold for clipping.

The clipping in (5) ensures that the update does not change the policy too much (i.e., it remains within a trust region), which helps maintain stability during training.

Often, additional terms are included in the overall loss to account for value function approximation and to encourage exploration. A common full objective is:

$$L(\theta) = \hat{\mathbb{E}}_t \left[L^{\text{CLIP}}(\theta) - c_1 \left(V_\theta(s_t) - V_t^{\text{target}} \right)^2 + c_2\, S\big[\pi_\theta\big](s_t) \right], \tag{7}$$

where:

- $V_\theta(s_t)$ is the value function approximation,
- V_t^{target} is the target value (which may be obtained via bootstrapping),
- $S\big[\pi_\theta\big](s_t)$ denotes the entropy of the policy at state s_t, encouraging exploration, and
- c_1, c_2 are coefficients that balance the contributions of the value loss and the entropy bonus, respectively.

This complete objective function allows PPO to simultaneously update the policy and the value function while keeping the policy update within a safe (proximal) region.

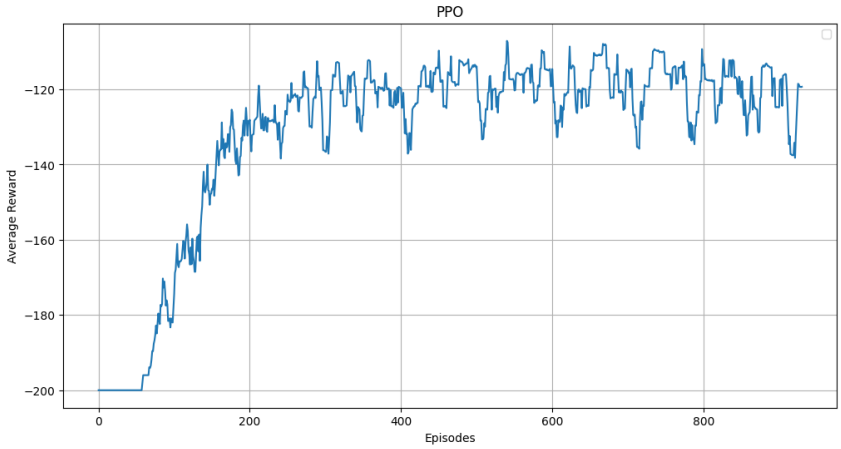
6.2 Results

The following results were obtained using the PPO algorithm. In our implementation, we use the version provided by `stable-baselines3`. The training was performed with the following hyperparameters:

Table 4: Hyperparameters for PPO Training

Hyperparameter	**Value**
Policy	MlpPolicy
Learning Rate	3×10^{-4}
Number of Steps (n_steps)	1024
Batch Size	32
GAE Lambda (λ)	0.9
Discount Factor (γ)	0.99
Number of Epochs	20
Clipping Range	0.2

Below is the plot of the results, showing the performance over the training episodes:



7 Advantage Actor Critic (A2C)

7.1 Overview

The Advantage Actor-Critic (A2C) [5] algorithm is a synchronous variant of the popular actor-critic methods. It leverages both policy gradient and value function estimation to reduce the variance of policy updates by using an advantage function as a baseline. In A2C, the actor is responsible for selecting actions, while the critic estimates the value of states, and the advantage function is used to inform the policy updates.

A2C consists of two main components:

- **Actor:** Parameterized by θ, it defines the policy $\pi_\theta(a \mid s)$ used to select actions.
- **Critic:** Parameterized by w, it estimates the state-value function $V_w(s)$, which serves as a baseline for advantage computation.

Loss Functions and Updates

Discounted Return

For each time step t, the discounted return R_t is computed by bootstrapping from the critic:

$$R_t = \sum_{k=0}^{n-1} \gamma^k \, r_{t+k} + \gamma^n \, V_w(s_{t+n}),$$

where $\gamma \in [0, 1]$ is the discount factor and n is the number of steps in the rollout.

Advantage Function

The advantage function quantifies how much better the taken action a_t is compared to the expected value of state s_t:

$$A(s_t, a_t) = R_t - V_w(s_t).$$

Policy Loss

The actor (policy) is updated using the policy gradient. The policy loss is:

$$L_{\text{policy}} = -\log \pi_\theta(a_t \mid s_t) \, A(s_t, a_t).$$

This loss encourages increasing the probability of actions that yield a positive advantage.

Value Loss

The critic is trained to accurately estimate the return. Its loss is the mean squared error between the predicted value and the discounted return:

$$L_{\text{value}} = \big(V_w(s_t) - R_t\big)^2.$$

Entropy Regularization

An entropy bonus is added to encourage exploration by penalizing overly deterministic policies:

$$H(\pi_\theta(\cdot|s_t)) = -\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t).$$

Overall Loss

The total loss function combines the policy loss, value loss, and the entropy regularization term:

$$L = L_{\text{policy}} + c_1 L_{\text{value}} - c_2 H(\pi_\theta(\cdot|s_t)),$$

where c_1 and c_2 are coefficients that balance the relative contributions of the value loss and entropy bonus.

Parameter Updates

The parameters are updated synchronously using gradients computed from the overall loss:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \left[-\log \pi_\theta(a_t|s_t) A(s_t, a_t) \right], \tag{8}$$

$$w \leftarrow w - \beta \nabla_w \left(V_w(s_t) - R_t \right)^2, \tag{9}$$

where α and β are the learning rates for the actor and critic, respectively.

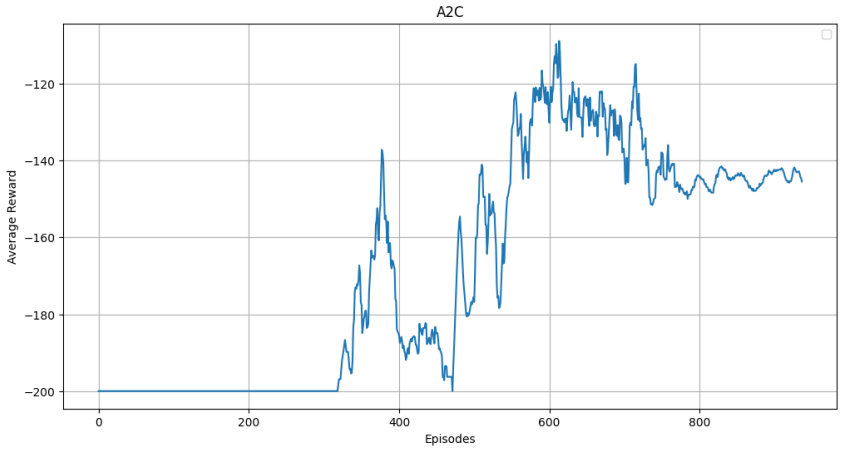
7.2 Results

The following results were obtained using the A2C algorithm. In our implementation, we use the version provided by `stable-baselines3`. The training was performed with the following hyperparameters:

Table 5: Hyperparameters for A2C Training

Hyperparameter	**Value**
Policy	MlpPolicy
Learning Rate	5×10^{-4}
Number of Steps (`n_steps`)	5
Discount Factor (γ)	0.99
GAE Lambda (λ)	1.0

Below is the plot of the results, showing the performance over the training episodes:


A2C
Average Reward
−120
−140
−160
−180
−200
0
200
400
600
800
Episodes

References

[1] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep q-learning," https://arxiv.org/pdf/1901.00137, 2020.

[2] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy networks for exploration," https://arxiv.org/pdf/1706.10295, 2019.

[3] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," https://arxiv.org/pdf/1507.06527, 2015.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," https://arxiv.org/pdf/1707.06347, 2015.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," https://arxiv.org/pdf/1602.01783, 2015.