2025

Mountain Car

FINAL REPORT

SARA SOBSTYL I ALICJA BANASZEWSKA

List of contents

Introduction

This project focuses on solving the classic reinforcement learning environment **MountainCar-v0** from OpenAI's Gymnasium suite. The problem involves training an agent to control a car situated between two hills. The car must build enough momentum to reach the flag on the right hill's summit. We implement a Q-learning algorithm to train the agent, discretize the continuous state space, and analyze its learning progress through experiments.

Task Analysis

Possible Approaches and Selected Methodology

Several methods exist for solving reinforcement learning (RL) problems such as MountainCar-v0:

- Q-Learning, which is simple, interpretable and works with small state spaces
- Deep Q-Network (DQN), which handles continuous state directly and scalable, but requires mode computation and tuning
- SARSA, which can lead to safer policies, but has slower convergence compared to Q-learning

Selected Methodology: Q-Learning

We chose Q-learning because it's easy to understand and works well with tables. Since the environment uses continuous values, we split the position and velocity into bins to make them easier to handle.

Datasets

MountainCar-v0 is a simulation, so it doesn't use any external dataset. Instead, the data comes from the agent's own experience as it interacts with the environment during training.

Tools, Libraries and Frameworks

Used tools with python:

- Gymnasium – To simulate the MountainCar environment
- Numpy – For numerical operaions
- Pickle – To save and load the trained Q-table
- Matplotlib – For plotting training metrics

These libraries are easy to use, work well, and are enough to build a basic RL algorithm. Gymnasium provides a clear and standard way to run the simulation.

Software specification

Scripts and Functions

- main.py – Entry point; runs training or evaluation
- agent.py - Contains train_agent() and evaluate_agent functions
- utils.py – Contains helper functions (e.g., discretization, plotting)
- config.py – Stores constants like number of bins and file paths.

Data Structures

- Q-table: np.zeros((N_BINS, N_BINS, action_space)). A 3D NumPy array storing Q-values for all discretized state-action pairs.
- Bins: Arrays dividing the continuous position and velocity space into equal-sized intervals

User interface

- Console interface
- Renderes the environment when render=true during evaluation
- Ouputs statistics such as time to goal and steps taken

The project is easy to run and customize. The main.py file lets users train the agent by uncommenting the train_agent line, or evaluate a trained model with evaluate_agent. This makes testing and viewing results simple.

The config.py file stores key settings like the number of bins (N_BINS) and file paths for saving the model and summary plot. Users can change these values to try different setups.

User can see all generated plots in DATA folder or information in terminal from which it was ran.

Experiments

Experimental Background

Objective: To observe how well the Q-learning learns and performs over time we played with the number of episodes.

Metrics Observed

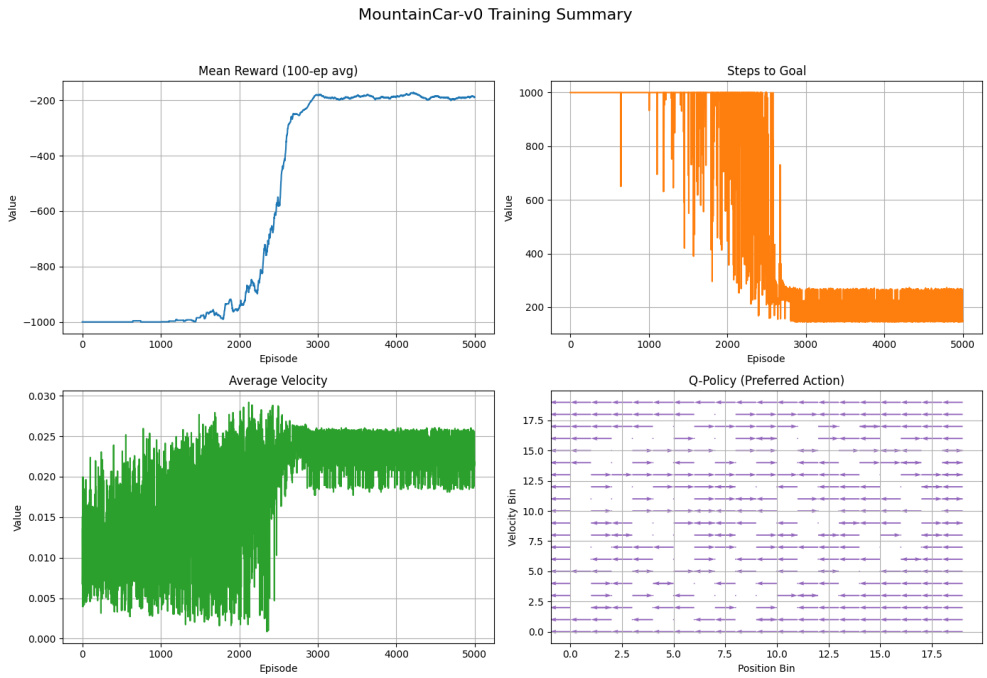
- Reward per episode
- Steps to reach the goal
- Average velocity during episodes

Plots generated

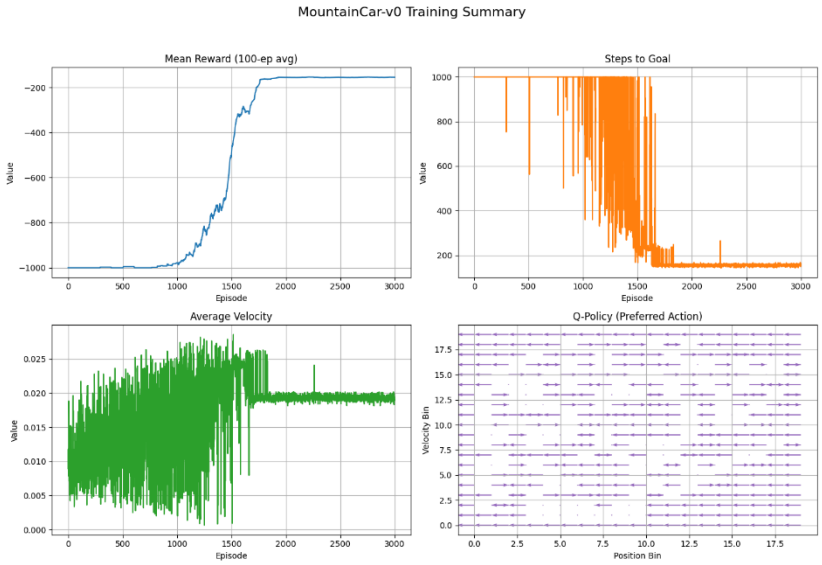
- Learning curve (reward vs episode)
- Steps per episode
- Velocity trend
- Q-value heatmap

Our graphs

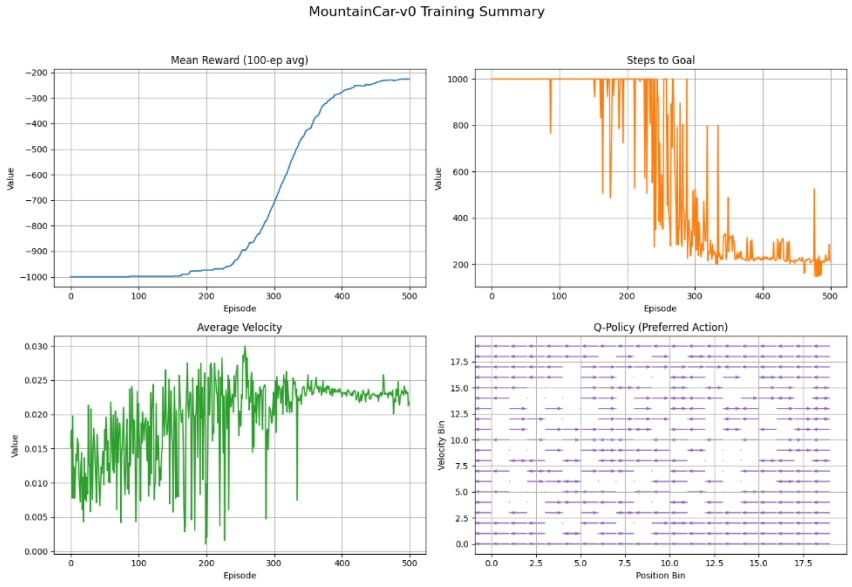
Graphs after learning with 5000:



Graphs after learning with 3000:



Graphs after learning with 500:



With 5000 episodes, it's easier to see a clear and steady improvement as the agent learns. With only 500 episodes, the steps to goal and average velocity are more random, showing more variability. The mean reward indicates that learning is still happening at 500 episodes, but it's less stable—while with 5000 episodes, the results form a smoother, more consistent trend.

Our evaluation

Evaluation for trained with 5000 episodes:

```
--- Evaluation for 3 episode(s) ---
Max velocity reached: 0.022401994094252586
Average max velocity: 0.022317858412861824
Min steps to goal: 145.0
Avg steps to goal: 169.33333333333334
Avg time to goal (s): 5.702544450759888
Min time to goal (s): 4.879469633102417
```

Evaluation for trained with 3000 episodes:

```
--- Evaluation for 3 episode(s) ---
Max velocity reached: 0.02247893251478672
Average max velocity: 0.02195403290291627
Min steps to goal: 159.0
Avg steps to goal: 179.33333333333334
Avg time to goal (s): 6.006640911102295
Min time to goal (s): 5.316523551940918
```

Evaluation for trained with 500 episodes:

```
--- Evaluation for 3 episode(s) ---
Max velocity reached: 0.024010105058550835
Average max velocity: 0.02214537685116132
Min steps to goal: 249.0
Avg steps to goal: 309.0
Avg time to goal (s): 10.340151389439901
Min time to goal (s): 8.33625054359436
```

The agent trained for 5000 episodes achieves the best time and the fewest steps to reach the goal. In contrast, the agent trained for only 500 episodes takes the longest time and the most steps.

Results and Analysis

The agent initially performs poorly due to random exploration but gradually improves as ε decreases. Over time:

- The reward becomes less negative (goal is reached faster)
- The number of steps to goal decreases
- The Q-value matrix stabilizes, showing stronger preferences for certain actions

Summary

Conclusions

This project gave us a deeper understanding of how reinforcement learning works in practice. By building a Q-learning agent for the MountainCar-v0 environment, we saw how an agent can learn from trial and error and improve its behavior over time. We learned how to handle continuous input by discretizing it and how important parameters like learning rate and exploration are to training success. Most importantly, we gained hands-on experience with the process of designing, training, and evaluating a learning agent, which helped us understand the core ideas behind Q-learning and agent-based learning much better.

Future work

- Compare performance with other algorithms (SARSA, DQN).
- Add live visualization of Q-table updates.
- Apply similar methods to more complex environments like Acrobot or CartPole.

References

- OpenAI Gym documentation: https://gymnasium.farama.org
- Documentation of used library

Project files:

https://github.com/ss19190/hill-climb-racing-knockoff.git