Double Deep Q-Network for Lunar Lander v2

Andrea Paletto

Alpen-Adria-Universität Klagenfurt

Lab machine learning for intelligence trasportation - SS23

Table of Contents

Introduction

Problem Description: Lunar Lander Game

The problem we are solving is the Lunar Lander Game using Double Deep Q-Network (DDQN) reinforcement learning.

Game Description:

The Lunar Lander is a classic reinforcement learning problem where the goal is to successfully land a spacecraft on the moon's surface. The game environment provides continuous observations and discrete actions.

Observation Space:

The observation space consists of eight continuous variables:

- Lunar lander's x and y coordinates
- Lunar lander's x and y velocities
- Lunar lander's angle and angular velocity
- Contact between lander's legs and the ground
- Whether or not the lander is in contact with the landing pad

Introduction (cont.)

Action Space:
The action space consists of four discrete actions:

- Do nothing
- Fire the left orientation engine
- Fire the main engine
- Fire the right orientation engine

Goal:
The goal is to learn an optimal policy to control the spacecraft and safely land it on the moon's surface, maximizing the total rewards obtained during the landing process.

System Architecture

To solve the Lunar Lander problem, we use a Double Deep Q-Network (Double DQN) as our reinforcement learning model.

- The Double DQN implementation includes two instances of the QNetwork: qNetworkLocal and qNetworkTarget. Both instances share the same architecture and are initialized with the same parameters.
- The qNetworkLocal is used for training and updating the weights, while the qNetworkTarget is used as a fixed target network for stable Q-value estimation.
- This setup helps address the overestimation bias in traditional Q-learning by decoupling the action selection and Q-value estimation steps.

Architecture of Double Deep Q-Network (Double DQN)

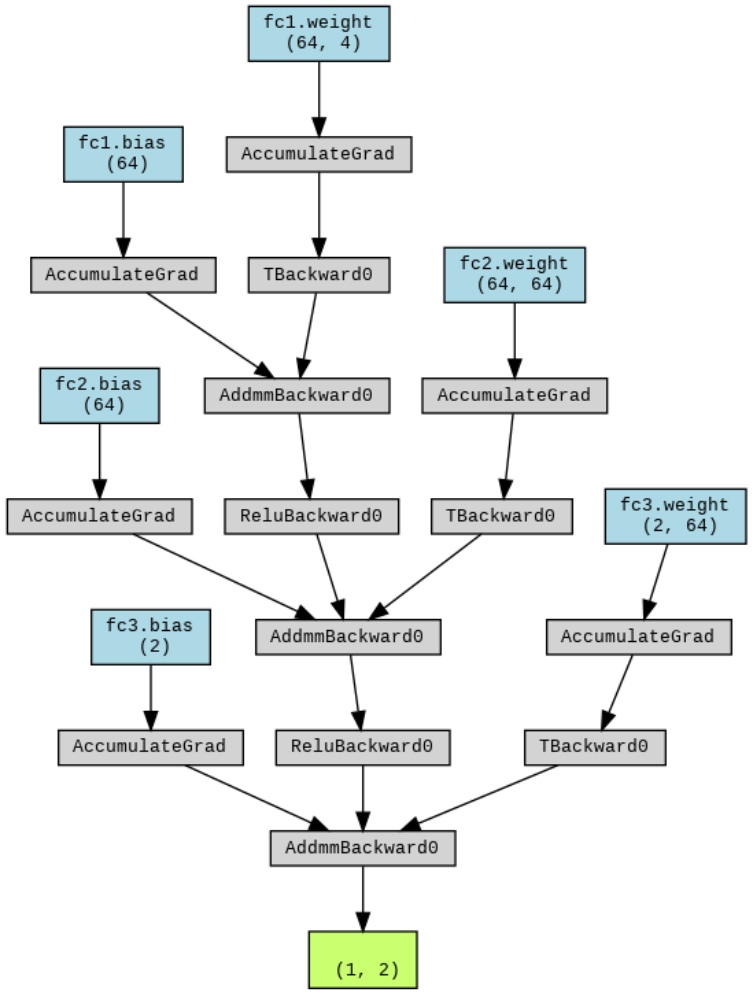

Figure: Model architecture

Adam Optimizer and MSE Loss

The Adam optimizer and mean squared error (MSE) loss were chosen for training the agent in this model. Here's a brief explanation of why they were used:

- **Adam Optimizer**: Adam is an adaptive optimization algorithm that combines the advantages of AdaGrad and RMSProp. It is well-suited for non-stationary problems with noisy or sparse gradients. The adaptive learning rate and momentum estimates help in efficient convergence and handling of different parameter scales.
- **MSE Loss**: Mean squared error (MSE) is a common loss function used for regression problems. It calculates the average squared difference between the predicted and target values. MSE is a suitable choice for this model as it penalizes larger errors more than smaller errors, which can help in achieving better convergence and learning of the Q-values.

These choices were made based on their proven effectiveness in deep Q-networks (DQNs) and reinforcement learning tasks.

Model Hyperparameters

- Learning Rate: `LR` (5e-4)
- Replay Buffer Size: `BUFFER_SIZE` (1e5)
- Minibatch Size: `BATCH_SIZE` (64)
- Update Frequency: `UPDATE_EVERY` (4)
- Discount Factor: `GAMMA` (0.99)
- Soft Update Parameter: `TAU` (1e-3)

Performance Evaluation

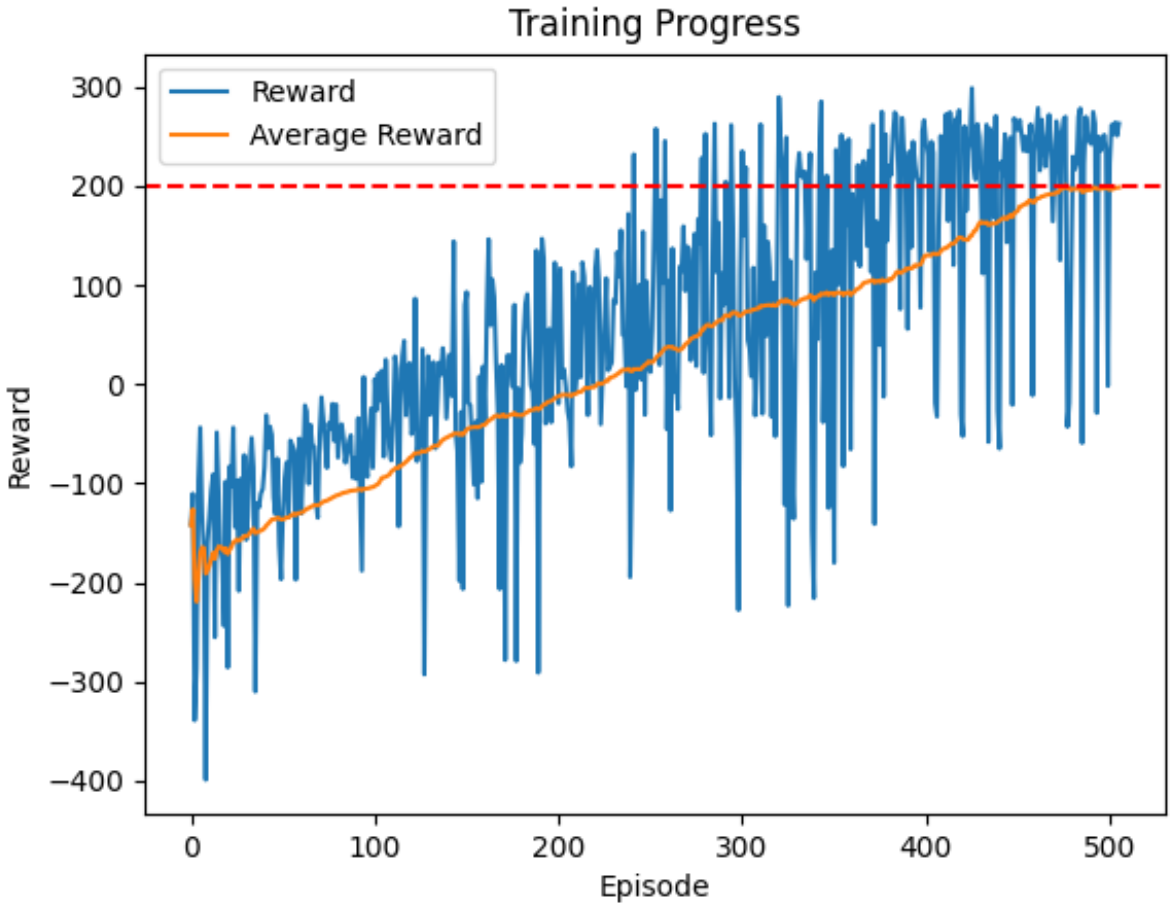

Figure: Reward, Average reward per episode

Epsilon decay

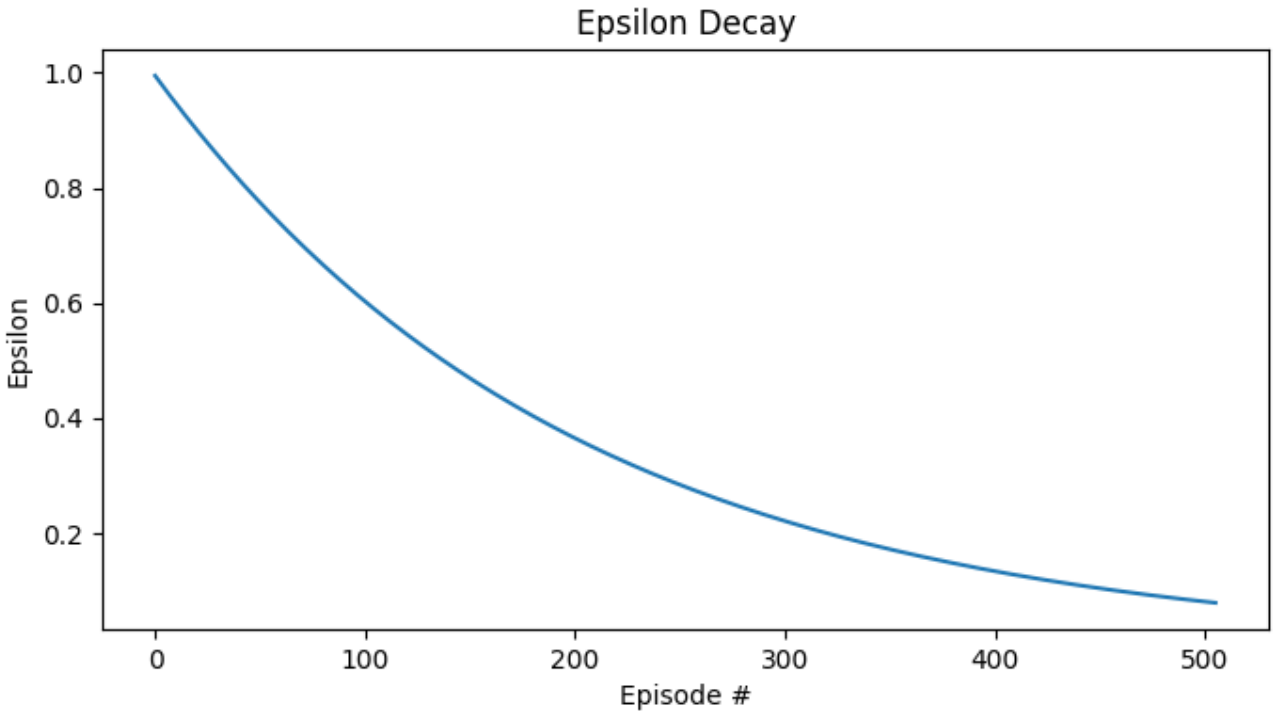

Figure: Epsilon decay graph

Model Outputs

- Show outputs of model (Make Movie of playing 5 episodes of the game)

```
Episode 1: Score = 221.36603011211184
Episode 2: Score = 243.80841738575063
Episode 3: Score = 223.50826523481692
Episode 4: Score = 232.70373184758 22
Episode 5: Score = 253.61783716164396

Average Score over 5 episodes: 235.0008563483811
```

Figure: Reward of 5 episodie test

Improvement Suggestions

- Implement **n-step training** to improve sample efficiency and stability n-step training allows us to update the Q-values using multiple consecutive transitions instead of individual transitions. This technique enhances sample efficiency and stability by considering the long-term effects of actions and reducing the variance in the updates.
- **Experiment with different hyperparameter settings** to find optimal values. Hyperparameters, such as learning rate, discount factor, batch size, and network architecture, significantly impact the performance of the DQN. By conducting experiments and fine-tuning these hyperparameters, we can discover optimal settings that enhance learning efficiency and overall performance.

Suggestion part 2

- **Multi-Step Learning**: Instead of updating the Q-values using only a single experience, multi-step learning combines multiple consecutive experiences to estimate the Q-values. This approach helps the agent to bootstrap its learning more efficiently by considering longer-term dependencies.
- **Gradient Clipping**: To stabilize the training process, gradient clipping can be applied during the backpropagation step to limit the magnitude of gradients. This prevents large gradients from causing unstable updates and diverging training.
- **Prioritized Experience Replay**: Instead of uniformly sampling experiences from the replay buffer, prioritized experience replay assigns priorities to experiences based on their TD errors. Experiences with higher TD errors, indicating higher learning potential, are sampled more frequently, allowing the agent to focus on important experiences.

Live Demo

- Play one episode of the game with your model