Report Machine Learning

Francesco Tinessa

July 5, 2024

1 Background

Reinforcement learning is a powerful notion within machine learning where agents learn to make decisions by interacting with an environment to maximize some notion of cumulative reward. This approach is widely used for solving complex decision-making problems across various domains, including robotics, gaming, and autonomous driving. The **LunarLander-v2** environment, provided by **OpenAI's Gym** toolkit, it's a common way to deal with RL problems. It simulates the task of guiding a spacecraft to land safely on the moon's surface, requiring precise control of the spacecraft's engines to achieve a smooth landing on a designated landing pad. The state space is continuous and consists of eight dimensions: the position (x, y), velocity (vx, vy), angle, angular velocity, and two Boolean values indicating whether each leg is in contact with the ground. The action space is discrete, with four possible actions: do nothing, fire the left orientation engine, fire the main engine, and fire the right orientation engine.

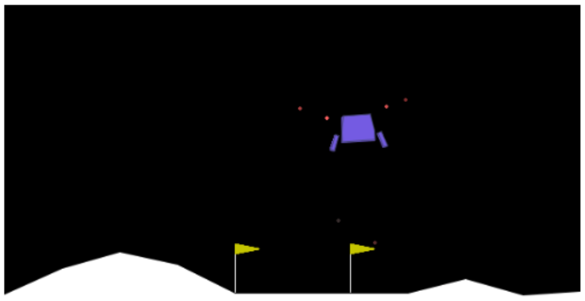

Figure 1: Lunar Lander

2 Requirements

The core objective of this study is to develop and compare the performance of two fundamental RL algorithms, ***Deep Q-Learning (DQN)*** and *Tabular Q-Learning*, in the LunarLander-v2 environment. Both algorithms aim to teach an agent how to effectively maneuver the spacecraft, optimizing landing trajectories while adhering to constraints such as minimizing fuel usage and ensuring a safe touchdown. Implement and compare two reinforcement learning algorithms: Deep Q-Learning (DQN) and Tabular Q-Learning.

- **Deep Q-Learning (DQN)**: This method uses a neural network to estimate the expected rewards for different actions in various states. It's like learning from experience and making decisions based on what has worked well before.
- **Tabular Q-Learning**: Unlike DQN, this approach stores Q-values (expected rewards) in a table. To apply it to the LunarLander-v2 environment, we need to discretize the continuous state space (like dividing it into grids) to manage this table effectively.

Evaluate how well each algorithm performs in terms of achieving high scores (which indicate successful landings) and how they balance between exploring new strategies and exploiting what they've learned. Let's get into deep and breakdown the problem. Here there are all useful libs that must be imported:

```
import torch as T
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torch.nn.functional as F
```

3 Solution: Deep Q-Network

Let's delve into the Deep Q-Network (DQN) and Agent classes, explaining their roles and functionalities in the context of the provided code. The core of the DQN is a neural network (DeepQNetwork) designed to approximate the Q-function, which estimates the expected future rewards for each action given a state. Here's a breakdown of the neural network used:

```
class DeepQNetwork(nn.Module):
def __init__(self, lr, input_dims, n_actions, fc1_dims=128):
    super(DeepQNetwork, self).__init__()
    self.input_dims = input_dims
    self.n_actions = n_actions
    self.fc1_dims = fc1_dims

    # Define the fully connected layers
    self.fc1 = nn.Linear(*self.input_dims, self.fc1_dims)
    self.fc2 = nn.Linear(self.fc1_dims, self.n_actions)

    # Set up optimizer and loss function
    self.optimizer = optim.Adam(self.parameters(), lr=lr)
    self.loss = nn.MSELoss()

    # Determine whether to use CPU or GPU (if available)
    self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
    self.to(self.device)

def forward(self, state):
    # Define the forward pass through the network with ReLU activation
    x = F.relu(self.fc1(state))
    actions = self.fc2(x)

    return actions
```

- **Neural Network Structure**: The DeepQNetwork class defines a neural network with one hidden layer (fc1) and one output layer (fc2). The hidden layer uses the ReLU activation function to introduce non-linearity, crucial for learning complex relationships between states and actions.
- **Optimizer and Loss**: The network uses the Adam optimizer (optim.Adam) for training, which adapts learning rates for each parameter. The Mean Squared Error (MSE) loss (nn.MSELoss) is employed to compute the error between predicted and target Q-values.

The Agent class orchestrates the reinforcement learning process using the defined neural network. Here are the key functionalities:

```
    class Agent:
    def __init__(self, gamma, epsilon, lr, input_dims, batch_size, n_actions,
                 max_mem_size=100000, eps_end=0.05, eps_dec=1e-5):
        # Initialize agent parameters
        self.gamma = gamma
        self.epsilon = epsilon
        self.eps_min = eps_end
        self.eps_dec = eps_dec
        self.lr = lr
        self.action_space = [i for i in range(n_actions)]
        self.mem_size = max_mem_size
        self.batch_size = batch_size
        self.mem_cntr = 0
        self.iter_cntr = 0

        # Initialize the Q-network
        self.Q_eval = DeepQNetwork(lr, n_actions=n_actions,
                                   input_dims=input_dims)

        # Initialize memory for experience replay
        self.state_memory = np.zeros((self.mem_size, *input_dims),
                                     dtype=np.float32)
        self.new_state_memory = np.zeros((self.mem_size, *input_dims),
                                         dtype=np.float32)
        self.action_memory = np.zeros(self.mem_size, dtype=np.int32)
        self.reward_memory = np.zeros(self.mem_size, dtype=np.float32)
        self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool_)
```

```
def store_transition(self, state, action, reward, state_, terminal):
    # Store transition into memory
    index = self.mem_cntr % self.mem_size
    self.state_memory[index] = state
    self.new_state_memory[index] = state_
    self.reward_memory[index] = reward
    self.action_memory[index] = action
    self.terminal_memory[index] = terminal
    self.mem_cntr += 1

def choose_action(self, observation):
    # Epsilon-greedy action selection
    if np.random.random() > self.epsilon:
        state = T.tensor(np.array([observation])).to(self.Q_eval.device)
        actions = self.Q_eval.forward(state)
        action = T.argmax(actions).item()
    else:
        action = np.random.choice(self.action_space)

    return action

def learn(self):
    # Update Q-network parameters using experience replay
    if self.mem_cntr < self.batch_size:
        return

    self.Q_eval.optimizer.zero_grad()

    max_mem = min(self.mem_cntr, self.mem_size)
    batch = np.random.choice(max_mem, self.batch_size, replace=False)
    batch_index = np.arange(self.batch_size, dtype=np.int32)

    state_batch = T.tensor(self.state_memory[batch]).to(self.Q_eval.device)
    new_state_batch = T.tensor(self.new_state_memory[batch]).to(self.Q_eval.
        device)
    action_batch = self.action_memory[batch]
    reward_batch = T.tensor(self.reward_memory[batch]).to(self.Q_eval.device)
    terminal_batch = T.tensor(self.terminal_memory[batch]).to(self.Q_eval.device)

    q_eval = self.Q_eval.forward(state_batch)[batch_index, action_batch]
    q_next = self.Q_eval.forward(new_state_batch)
    q_next[terminal_batch] = 0.0
```

```
        q_target = reward_batch + self.gamma * T.max(q_next, dim=1)[0]

        loss = self.Q_eval.loss(q_target, q_eval).to(self.Q_eval.device)
        loss.backward()
        self.Q_eval.optimizer.step()

        self.iter_cntr += 1
        self.epsilon = self.epsilon - self.eps_dec \
            if self.epsilon > self.eps_min else self.eps_min
```

- **Initialization**: The Agent class initializes parameters such as discount factor (`gamma`), exploration rate (`epsilon`), learning rate (`lr`), and memory size (`max_mem_size`) for experience replay.
- **Experience Replay**: The agent uses arrays (`state_memory`, `action_memory`, etc.) to store transitions (`store_transition`) experienced during interaction with the environment. This facilitates efficient training by breaking correlations between consecutive samples and stabilizing learning.
- **Action Selection**: The `choose_action` method implements an epsilon-greedy strategy for action selection, balancing exploration and exploitation. It selects actions either based on the current policy (exploitation) or randomly (exploration).
- **Training (Learning)**: The `learn` method updates the Q-network parameters using batches of experiences sampled from memory. It computes Q-values for both current and next states (`q_eval`, `q_next`) and computes the MSE loss between predicted and target Q-values to update the network via backpropagation.

4 Solution: Tabular Q-Learning

The provided code implements a Tabular Q-Learning Agent, which is a reinforcement learning technique aimed at learning an optimal policy for an agent interacting with an environment through discrete actions. Let's delve into the details and rationale behind each component of the implementation.

1. **Initialization**:

```
        def __init__(self, gamma, epsilon, lr, n_actions, state_bins, eps_end
            =0.05, eps_dec=5e-4):
        self.gamma = gamma
        self.epsilon = epsilon
        self.eps_min = eps_end
        self.eps_dec = eps_dec
        self.lr = lr
        self.action_space = [i for i in range(n_actions)]
        self.state_bins = state_bins
        self.state_space_size = np.prod([len(b) + 1 for b in state_bins])
        self.Q = np.zeros((self.state_space_size, n_actions))
```

- *Gamma (gamma)*: Represents the discount factor for future rewards, influencing how much weight is given to future versus immediate rewards.
- *Epsilon (epsilon)*: Controls the exploration-exploitation trade-off, where higher epsilon values encourage exploration (choosing random actions), while lower values promote exploitation (choosing actions based on learned Q-values).
- *Learning Rate (lr)*: Determines how much the agent updates its Q-values in response to new information.
- *Action Space (action_space)*: Defines all possible actions the agent can take in the environment.
- *State Bins (state_bins)*: Defines the discretization bins for each state variable, facilitating the conversion of continuous state space into a discrete state representation.
- *State Space Size (state_space_size)*: Calculated based on the number of bins per state variable, providing the total number of possible discrete states.
- *Q-Table (Q)*: Initialized as a matrix of zeros where rows represent states and columns represent actions, storing the estimated Q-values for each state-action pair.

2. **State Discretization**

```
def discretize_state(self, state):
    state_discrete = sum([np.digitize(state[i], self.state_bins[i]) * (
        len(self.state_bins[i]) ** i) for i in range(len(state))])
    return state_discrete
```

- **Purpose**: Converts continuous state variables into a discrete state representation suitable for tabular Q-learning.
- **Implementation**: Uses `np.digitize` to assign state variables into discrete bins defined by `state_bins`, and computes a unique index (`state_discrete`) for each state based on its discretized values.

3. **Action Selection**:

```
def choose_action(self, state):
    state_discrete = self.discretize_state(state)
    if np.random.random() > self.epsilon:
        action = np.argmax(self.Q[state_discrete, :])
    else:
        action = np.random.choice(self.action_space)
    return action
```

- **Purpose**: Determines which action to take based on the current state, balancing exploration and exploitation.
- **Implementation**: Uses an epsilon-greedy policy where, with probability epsilon, the agent chooses a random action to explore the environment; otherwise, it selects the action with the highest Q-value for the given state (state_discrete).

4. **Q-Value Update**

```
def learn(self, state, action, reward, state_):
    state_discrete = self.discretize_state(state)
    state_discrete_ = self.discretize_state(state_)
    max_future_q = np.max(self.Q[state_discrete_, :])
    current_q = self.Q[state_discrete, action]
    new_q = current_q + self.lr * (reward + self.gamma * max_future_q -
        current_q)
    self.Q[state_discrete, action] = new_q

    # Decay epsilon
    self.epsilon = self.epsilon - self.eps_dec if self.epsilon > self.
        eps_min else self.eps_min
```

- **Purpose**: Updates the Q-value for the state-action pair based on observed rewards and estimated future rewards.

5 Result

5.1 Deep Q Learning

Let's analyze and compare the results obtained from the two methods used for training:

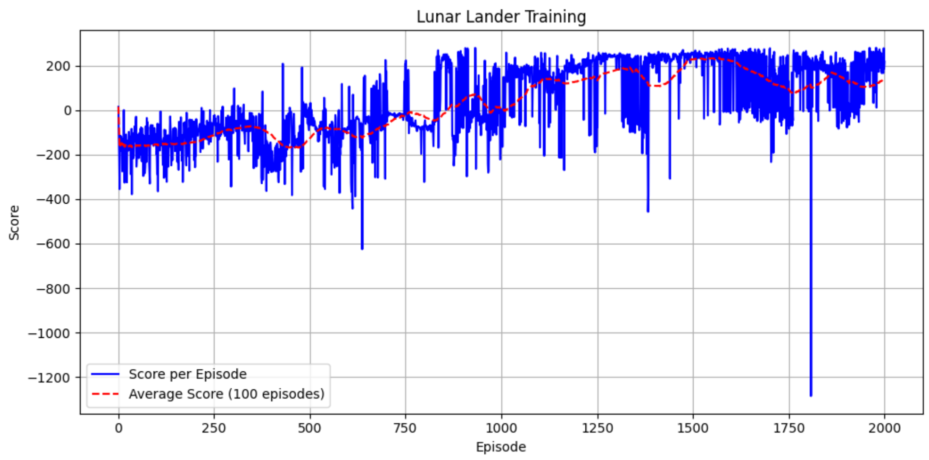

Figure 2: DQN

The first chart represents the training results over 2000 episodes. Initially, we observe that the scores obtained by the agent fluctuate significantly, with many episodes yielding negative scores. This indicates that, in the early stages of training, the agent struggles to perform the task correctly. The red dashed line, representing the average scores over 100 episodes, starts at a low value, suggesting that the agent is having difficulty achieving consistent performance.

As the training progresses, around 500 episodes, there is a noticeable improvement in the average score. This positive trend indicates that the agent is learning and enhancing its landing capabilities.

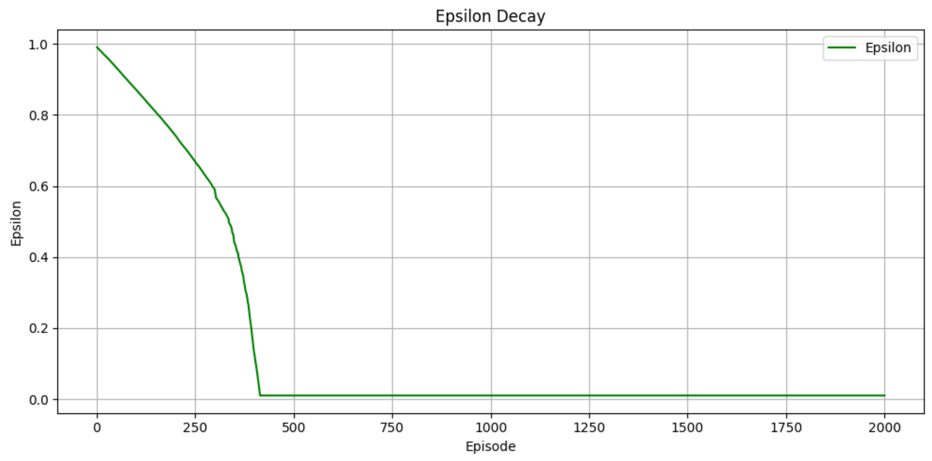

Figure 3: Decay

However, the score fluctuations remain high, demonstrating inconsistency in the agent's performance.

Approaching 1500 episodes, the average score starts to stabilize at higher values, signifying that the agent is becoming more consistently competent. Despite this, there are still occasional episodes with very low scores, although these are less frequent than at the beginning.

To better understand the agent's learning behavior, let's examine the epsilon decay plot for the first method. The epsilon value starts at 1.0, indicating complete exploration, and gradually decreases over the first 500 episodes. After 500 episodes, epsilon drops to 0.0 and remains there for the rest of the training period.

The rapid epsilon decay encourages the agent to explore a wide range of actions initially, which is beneficial for learning diverse strategies. However, by episode 500, the agent switches entirely to exploitation, relying solely on the knowledge it has acquired so far. This strategy promotes rapid learning, which is reflected in the significant performance improvement within the first 2000 episodes. However, the complete reliance on exploitation after episode 500 may contribute to the observed high variance in performance, as the agent lacks flexibility to adapt to new strategies.

5.2 Tabular Q Learning

Moving to the second chart, it shows the results of a much longer training period, extending over 50,000 episodes. Initially, I had trained this method for 5000 episodes, but it showed inconsistent results, prompting me to extend the training significantly. In the early episodes of this extended training period, the scores are highly variable and often negative, reflecting the initial difficulties the agent faces in learning the task.

During the intermediate training phase, between 10,000 and 30,000 episodes, a gradual improvement in the average score is observed. This suggests slow but steady learning. Although the score fluctuations remain significant, the range of variation seems to slightly reduce compared to the initial stages.

In the later stages, between 30,000 and 50,000 episodes, the average score improves more noticeably, indicating that the agent continues to learn and refine its skills. However, the variability in scores persists, though the agent avoids the extremely low scores seen in the first method.

The epsilon decay plot for the second method reveals that the epsilon value starts at 1.0 and drops almost immediately to a low value, remaining close to zero for the entire duration of the 50,000 episodes.

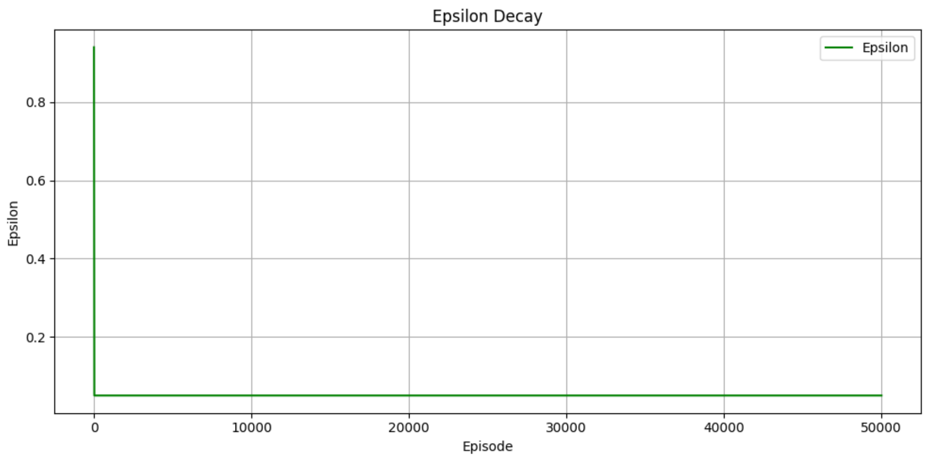

Figure 4: Decay

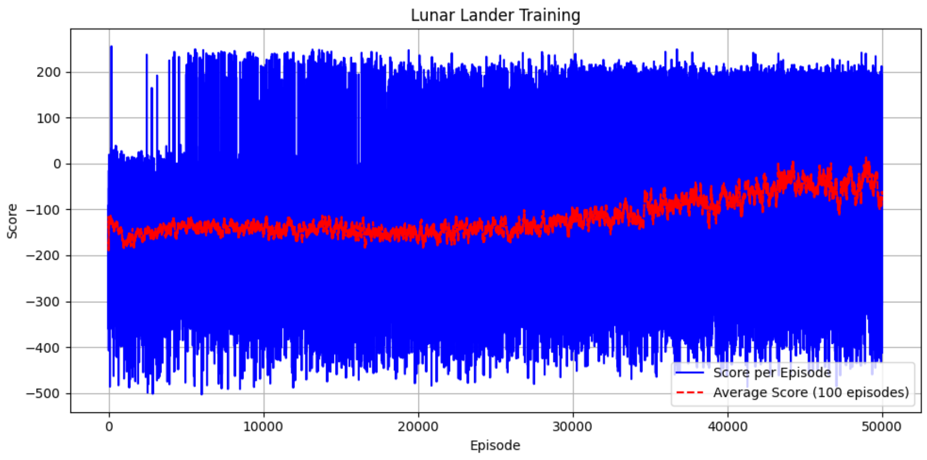

Figure 5: Tabular Q

This approach minimizes exploration right from the start and emphasizes exploitation throughout the training period.

The immediate drop to a low epsilon value restricts exploration, resulting in a slower learning process. The agent's performance improves gradually over a much longer period (50,000 episodes). The low exploration rate promotes consistency, reducing the likelihood of extremely poor performance episodes, but also limits the agent's ability to discover potentially better strategies.

5.3 Comparison

Analyzing the two methods as a whole, some significant differences emerge. The first method demonstrates a faster learning capability, with noticeable improvements within the first 2000 episodes. This could be particularly useful in contexts requiring rapid adaptability. However, this rapidity is accompanied by high score variability, with occasional episodes of catastrophic performance.

On the other hand, the second method, while showing slower learning, ensures greater stability in the long term. The agent avoids extremely low scores and shows a consistent, though very gradual, progression. This might be preferable in situations where performance consistency and safety are more critical than the speed of learning.