Moon Landing using Deep Q-Learning

Khang Pham (A20464881)

March 22, 2025

Course: MMAE 500

Professor: Scott Dawson

Contents

1 Introduction **2**

2 Methods **2**

2.1 Q-Learning . . . 2

2.1.1 Initialize action-state table . . . 3

2.1.2 Choose an action . . . 3

2.1.3 Update Q-value . . . 3

2.2 Deep Q-Learning . . . 4

3 Environment description **5**

4 Results **6**

4.1 Episode rewards . . . 7

4.2 Episode duration . . . 7

5 Discussion **8**

A Project code **11**

Abstract

This paper discussed the implementation of Deep Q-Learning, a model-free reinforcement learning algorithm on solving continuous state space environments. In contrast to other Data-Driven Modeling method, reinforcement learning create its own data set to learn from with out supervision from experts. To aid application of Deep Q-Learning, other techniques such as epsilon-greedy and soft-updated target network are also introduced. The details on these methods are discussed in section 2. The environment is highly controlled for demonstration purposes; however, the techniques is expected to able to solve higher dimension problems. As a result of the project, the agent successfully learned the environment and found the optimal solution to the problem i.e. land the spaceship fast and accurately. Some other potential strategies to apply Deep Q-Learning are also discussed in sections 2 and 5. The code for this project can be found in the appendix **??**

1 Introduction

Traditional control theory techniques aim to describe the systems based on determined rules and correlations between variables. These techniques, with correct calibration, often present accurate results with robust computational potentials. However, with complex systems, sometimes it is difficult to derive governed equations and control them. This project aims to experiment with using reinforcement learning to solve the high-dimension problem: landing a space shuttle.

Q-learning is the original studied method. However, understanding the limitation of discrete state and action space, Deep Q-Learning is implemented. Although reinforcement learning is an unstable method with accuracy often lower than that of control theory techniques, the benefit of model-free control is still appealing. Without the need to study and model the systems in detail, it is possible to solve much more complicated problems, including uncertainty problems.

2 Methods

The result discussed in section 4 is achieved by Deep Q-Learning. This method combines Q-Learning (sec. 2.1) and neural networks (sec. 2.2. Details on why the neural network was introduced to improve the method are also discussed in the following section.

2.1 Q-Learning

Q-learning is a model-free reinforcement learning algorithm to evaluate the value at a particular state [1]. The algorithm assumes the Markov property, meaning the correlation between the current action and future state depends exclusively on the present state and not on any of the previous states [2].

Q-Learning algorithm can be described in three steps:

1. Initialize the action-state table.
2. Choose an action based on the current state.
3. Update Q-value.

2.1.1 Initialize action-state table

The action-state table lists out all action-state pairs with a Q-value assigned to each pair. This step presents a limitation of the Q-Learning algorithm which is the dimensions of possible states and actions are both required to be finite. It is suitable for cases where the state space and action space are both discrete or can be safely converted to discrete. A classic example is balancing an inverted pendulum on a cart on a 2D plane. The angle of the pole can be represented as, instead of a continuous space $[0, 2\pi)$, a discrete space [0, 2π, `step_number`]. This method has been proven to work with some trade-off of computational speed [3]. While it is possible to increase `step_number` to a large number and make the discrete space equivalent to continuous space in the computational world, the speed of the algorithm is traded off at an exponential rate.

2.1.2 Choose an action

With all action-state pairs listed out, the agent can interact with the environment. The process will start with the agent observing the current state of the environment; and based on its observation, an action is chosen to maximize the future reward. Initially, the agent would take action randomly; however, after each iteration and the Q-value gets updated (sec. 2.1.3), the agent will learn to take action more intentionally.

To avoid getting stuck in a local minimum, there has to be a method introduced to balance the exploration and exploitation rate of the agent. A simple method to do this is the epsilon-greedy algorithm [4]. In brief, epsilon ϵ is initialized as 1, and the agent would 100% choose action randomly (exploration). After each episode, ϵ will decay by a predetermined rate. At each episode, the agent would have ϵ chance to explore and $1 - \epsilon$ chance to exploit the environment.

In the program used for the project, epsilon is decayed following the following equation. ϵ_{min} is set up to ensure that the agent keeps exploring the environment at a small rate as time goes on.

$$\epsilon(t) = \epsilon_{min} + \frac{\epsilon_{max} - \epsilon_{min}}{\exp(t/\epsilon_{decay})}$$

2.1.3 Update Q-value

After the agent takes the action, the Q-value for that state-action pair needs to be calculated and saved for future iteration. The Bellman Equation is used for this step.

$$Q_{new}(s(t), a) = Q_{current}(s(t), a) + \alpha \left[(R + \gamma * Q_{max}(s(t+1), a)) - Q_{current}(s(t), t) \right]$$

where:
- $Q(s(t), a) \rightarrow Q$ of state s and action a

- $R \rightarrow$ reward when action a is taken as state s
- $\gamma \rightarrow$ discount factor

The specific values for learning rate α and discount factor γ are up to the user. However, there is always a trade-off when changing these values. With high α. the agent would learn faster but may never get to the optimal result since the step size is too large.

2.2 Deep Q-Learning

Deep Q-learning is a variation of Q-learning. To combat a Q-Learning limitation of finite state and action space, the Q-value, instead of being calculated based on state-action pairs, is determined by a neural network.

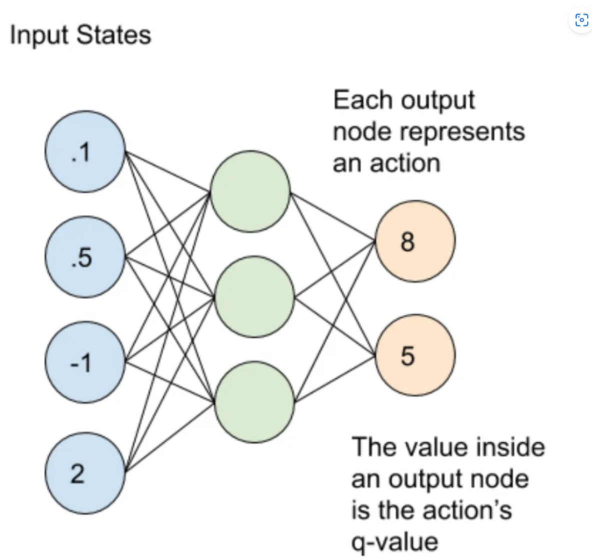

Figure 1: Deep Q Network maps states to actions and corresponding Q-values

To utilize Deep Q-Learning, instead of setting up a list of action-state pairs with randomly assigned Q-values, a neural network with random weights is set up. After each iteration, the weights of the neural network will be updated similar to Q-Learning. The complexity of the neural network can vary depending on the problem. For this project, a simple linear model consisting of two layers is implemented.

```
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()

        hidden_layer_1 = 128
        hidden_layer_2 = 128

        self.layer1 = nn.Linear(n_observations, hidden_layer_1)
        self.layer2 = nn.Linear(hidden_layer_1, hidden_layer_2)
        self.layer3 = nn.Linear(hidden_layer_2, n_actions)

```

```
        # Called with either one element to determine next action, or a batch
        # during optimization. Returns tensor([[left0exp,right0exp]...]).
        def forward(self, x):
            x = F.relu(self.layer1(x))
            x = F.relu(self.layer2(x))
            return self.layer3(x)
```

To add stability to the algorithm, a copy of the main work is created, called a target network. Instead of getting updated every episode like the main network, the target network is updated in a delayed manner. Specific rules to dictate when the target network is updated can be different depending on the program. For example, the target network can be updated after a specific number of episodes or it can be updated every episode but multiply with a scale factor. The program implemented for this project uses the second method.

```
# Soft update of the target network's weights
#                  + (1       )
target_net_state_dict = target_net.state_dict()
policy_net_state_dict = policy_net.state_dict()

    for key in policy_net_state_dict:
        target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_state_dict[key]*(1-TAU)
        target_net.load_state_dict(target_net_state_dict)
```

The target network can also be removed entirely; and expectedly, that will support faster learning at the cost of stability. However, in my attempt, the network shows no positive learning after 1000 episodes.

3 Environment description

For this project, the `LunarLander-v2` from Gymnasium of Farama Foundation is implemented.

The environment tasks a space shuttle to land on a landing pad always at coordinates $(0, 0)$. The terrain around the landing site is randomly generated. Initially, the spaceship is spawned at the top center of the viewport with a random force applied to its center of mass. To simplify the problem, there is no wind in the environment. This is accurate in the case of the moon but would need to change if the simulation is for another planet. The action space of the agent consists of four discrete actions:
- Do nothing
- Fire left orientation engine
- Fire main engine
- Fire right orientation engine.

The engine is assumed to be perfect and can only be turned fully on or off without variation.

In every frame, the observed state consists of six float values and two boolean values:
- (x, y) coordinates of the spaceship
- $v(x, y)$ linear velocity of the spaceship
- Angle of the spaceship

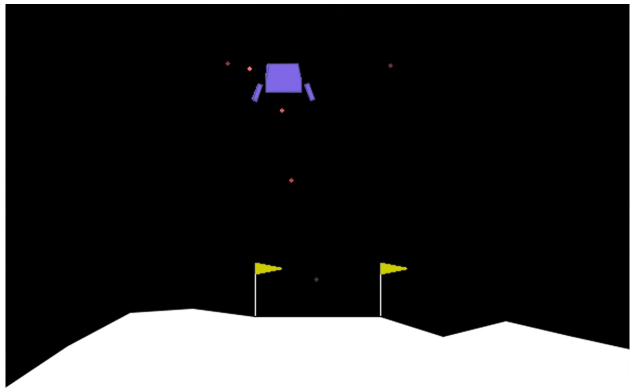

Figure 2: Example of an episode

- Angular velocity of the spaceship
- Whether the left leg is in contact with the ground (boolean).
- Whether the right leg is in contact with the ground (boolean)

After every step, a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode. For each step, the reward:
- is increased/decreased the closer/further the lander is to the landing pad.
- is increased/decreased the slower/faster the lander is moving.
- is decreased the more the lander is tilted (angle not horizontal).
- is increased by 10 points for each leg that is in contact with the ground.
- is decreased by 0.03 points each frame a side engine is firing.
- is decreased by 0.3 points each frame the main engine is firing.

The episode receives an additional reward of -100 or +100 points for crashing or landing safely respectively.

The episode ends when:
- the frame spent in the environment reaches 1000
- the spacecraft flies out of the environment window.

4 Results

Before analyzing the result, it is worthied to set expectations. The agent would not start training until its memory fills up a batch size, there should be little or no positive trends to the received rewards. Because there is punishment for each frame the engine is fired, the agent will take advantage of gravity to accelerate downward and only fire the engine last minute to correct itself. It should be noted that the landing gear is assumed to be indestructible, and there is no punishment for landing at a very high speed,

4.1 Episode rewards

The plot 3 shows that the agent successfully learns the environment. After 128 episodes, which filled up on batch, the agent starts to learn about the environment and quickly reaches 200 rewards, which is considered "solved" for this problem [5].

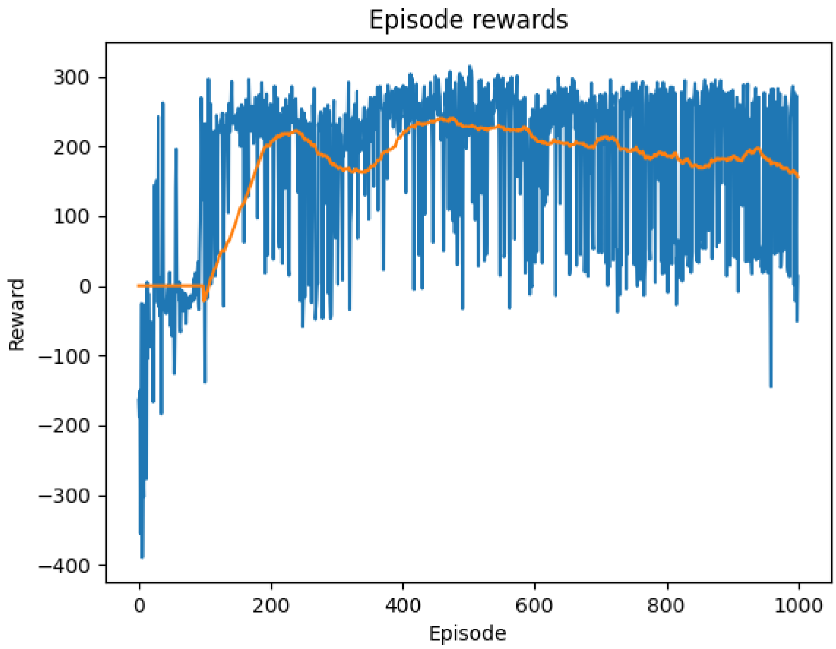

Figure 3: Rewards for 1000 episodes

After that, the reward curve went through a dip but quickly recovered later. This may be because the agent is trying to explore the environment instead of exploiting pre-existing strategies. However, because epsilon is relatively small at episode 300, it is unlikely that this would happen again if the program is executed again.

4.2 Episode duration

As mentioned above, there is a mild punishment for firing the engine, and there is no punishment for landing at a very high speed. The agent learned to land quicker as time went on (fig. 4). However, the rate of agents learning this is uncertain. In other attempts to train the neural network, the agent learns to cut down the time a lot slower. Usually, it takes about 400 episodes to see a clear drop in duration.

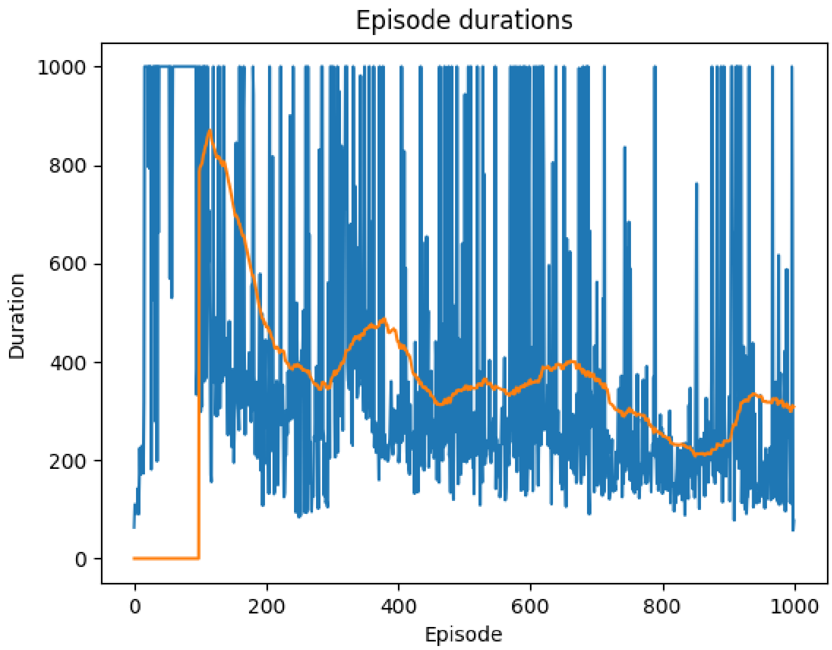

Figure 4: Durations of 1000 episodes

The speed at which the agent completes the task also comes with a trade-off. At about episode 400, the agent received the most reward. For this run to train the neural network, it seems like this is the optimal point where the agent landed the spaceship confidently and accurately. As time goes on, the episode durations keep decreasing (possibly a consequence of the rapid decrease in duration), and the episode rewards also experience a mild decrease. When the agent tries to land the spaceship too fast, its accuracy also decreases.

However, because of the discount factor γ, the calculated Q-value from experience would lose effect over time. The agent is expected to correct itself as time goes on.

5 Discussion

This project experimented with a recently developed reinforcement learning technique, Deep Q-Learning, to study an unknown environment. The implementation of the project shows promising results. The network solves the problem reasonably quickly and continues to improve even after an initial solution is found. It is possible to apply this technique to solve more complex systems even ones that are not fully understood.

For future study, some calibration to make the model more unstable may be introduced: removing the target network, using Q-Learning on continuous space, etc. These techniques are briefly mentioned in this report but were not implemented successfully in the program. A trial to apply the techniques to more complex simulations will also be considered. Simply introducing turbulence in the environment and control systems is an interesting path to study the program more in-depth.

Acknowledgement

This project is completed by following closely with PyTorch's example on Deep Q Network [6] and Gymnasium documentation [7].

Large language models ChatGPT 3.5 from Open AI [8] and Gemini 1.0 from Google [9] are used for general research purposes and fixing syntax while programming.

References

1. Watkins, C. J. C. H. & Dayan, P. Q-learning. *Machine Learning.* https://doi.org/10.1007/BF00992698 (1992).
2. Andersen, P. & Keiding, N. in *International Encyclopedia of the Social Behavioral Sciences* (eds Smelser, N. J. & Baltes, P. B.) 4946–4956 (Pergamon, Oxford, 2001). ISBN: 978-0-08-043076-8. https://www.sciencedirect.com/science/article/pii/B0080430767021033.
3. Arts, L. *Comparing Discretization Methods for Applying Q learning in Continuous State-Action Space* 2017.
4. McCaffrey, J. D. *The Epsilon-Greedy Algorithm* https://jamesmccaffrey.wordpress.com/2017/11/30/the-epsilon-greedy-algorithm/.
5. Klimov, O. *Lunar Lander* https://gymnasium.farama.org/environments/box2d/lunar_lander/#credits.
6. Paszke, A. & Towers, M. *Reinforcement Learning (DQN) Tutorial* https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.
7. Foundation, F. *Gymnasium Documentation* https://gymnasium.farama.org/.
8. OpenAI. *ChatGPT* https://openai.com/chatgpt.
9. Google. *Gemini* https://gemini.google.com/.

A Project code

```python
import gymnasium as gym
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
import csv
# import pandas as pd

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# env = gym.make("LunarLander-v2", render_mode = "human")
env = gym.make("LunarLander-v2")

# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display
plt.ion()

# use GPU if available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("torch.cuda.get_device_name(0) = ", torch.cuda.get_device_name(0))

# %%

Transition = namedtuple("Transition", ("state", "action", "next_state", "reward"))

# %%

class ReplayMemory(object):
    # create memory
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    #
    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        # return random.sample(self.memory, batch_size)
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

```python

# %%

class DQN(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()

        hidden_layer_1 = 128
        hidden_layer_2 = 128

        self.layer1 = nn.Linear(n_observations, hidden_layer_1)
        self.layer2 = nn.Linear(hidden_layer_1, hidden_layer_2)
        self.layer3 = nn.Linear(hidden_layer_2, n_actions)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)

# %%

# BATCH_SIZE is the number of transitions sampled from the replay buffer
# GAMMA is the discount factor as mentioned in the previous section
# EPS_START is the starting value of epsilon
# EPS_END is the final value of epsilon
# EPS_DECAY controls the rate of exponential decay of epsilon, higher
    means a slower decay
# TAU is the update rate of the target network
# LR is the learning rate of the ``AdamW`` optimizer
BATCH_SIZE = 500
GAMMA = 0.99
EPS_START = 1.0
EPS_END = 0.05
EPS_DECAY = 1000
TAU = 0.005
LR = 1e-4

# Get number of actions from gym action space
n_actions = env.action_space.n
# Get the number of state observations
state, info = env.reset()
n_observations = len(state)

policy_net = DQN(n_observations, n_actions).to(device)
target_net = DQN(n_observations, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())

optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)
memory = ReplayMemory(10000)


```

```python
steps_done = 0

# %%

def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY) # decaying epsilon exponentially
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return the largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1).indices.view(1, 1)
    else:
        return torch.tensor([[env.action_space.sample()]], device=device, dtype=torch.long)

# %%

episode_durations = []
episode_rewards = []

def plot_episode_data(episode_data, data_name:str, instant:int, show_result=False):
    """
    data_name should be capitalize
    """
    plt.figure(instant)
    data_t = torch.tensor(episode_data, dtype=torch.float)
    if show_result:
        plt.title("Episode " + data_name)
    else:
        plt.clf()
        plt.title('Training... (Episode', data_name + ")")
    plt.xlabel('Episode')
    plt.ylabel(data_name)
    plt.plot(data_t.numpy())

    # Take 100 episode averages and plot them too
    if len(data_t) >= 100:
        means = data_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    # pause a bit so that plots are updated
    # can remove if num_episodes is too large
    plt.pause(0.001)

    # # use when writting code in ipynb
```

```python
    # if is_ipython:
    #     if not show_result:
    #         display.display(plt.gcf())
    #         display.clear_output(wait=True)
    #     else:
    #         display.display(plt.gcf())

# %%

# set up file to record training data
file_name = "training_data.csv"

def record_csv(episodes, rewards, durations, file_name=file_name):
    # Open the file in write mode
    with open(file_name, mode='w', newline='') as file:
        writer = csv.writer(file)

        # header
        writer.writerow(['Episode', 'Reward', 'Duration'])

        # record training data
        for episode_val, reward_val, duration_val in zip(episodes, rewards, durations):
            writer.writerow([episode_val.item(), reward_val.item(), duration_val])

# %%

def optimize_model():
    # start training when memory is equal to BATCH_SIZE
    if len(memory) < BATCH_SIZE:
        return

    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
```

```
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1).values
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).max(1).values
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    # In-place gradient clipping
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()

# %%

if torch.cuda.is_available():
    num_episodes = 1000
else:
    num_episodes = 50

for i_episode in range(num_episodes):
    # Initialize the environment and get its state
    state, info = env.reset()
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)

    episode_reward = 0

    for t in count():
        action = select_action(state)
        observation, reward, terminated, truncated, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)
        done = terminated or truncated
```

```python

        if terminated:
            next_state = None
        else:
            next_state = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(0)

        # Store the transition in memory
        memory.push(state, action, next_state, reward)
        episode_reward += reward

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()

        # Soft update of the target network's weights
        #                  + (1        )
        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_state_dict[key]*(1-TAU)
        target_net.load_state_dict(target_net_state_dict)

        if done:
            episode_durations.append(t + 1)
            episode_rewards.append(episode_reward)
            # plot_durations()
            # plot_rewards()
            print("Episode =", i_episode, "\tReward=%.2f" % episode_reward, "\tDuration=%.2f" % (t+1))
            break

print('Complete')
# plot_durations(show_result=True)
# plot_rewards(show_result=True)
plot_episode_data(episode_rewards, "Reward", 1, True)
plot_episode_data(episode_durations, "Duration", 2, True)
record_csv(episodes=torch.arange(num_episodes), rewards=episode_rewards, durations=episode_durations)
plt.ioff()
plt.show()
```