SAPIENZA
UNIVERSITÀ DI ROMA

Machine Learning Project
Taxi Q-Learning

Artificial Intelligence & Machine Learning

Facoltà di ingegneria dell'informazione, informatica e statistica
MSc. in Engineering in Computer Science

Made by:
Angelo Trifelli, 1920939

Submitted to:
Prof. Fabio Patrizi

Academic Year 2023/2024

Contents

1 Introduction

This Machine Learning project consists of an implementation of a Reinforcement Learning agent based on Q-Learning. The environment that has been selected is the **Taxi** environment provided by **Gymnasium** (such environment is part of the *Toy Text* family). In particular, we will see three possible implementations of the agent: the first one is a **tabular q-learning** approach while the others are based on the use of a **Deep Q-Network (DQN)**.

The project has been developed entirely with the **Python** programming language and, in what follows, we will describe the problem addressed, the solutions adopted and the implementations of such solutions with the corresponding Python code.

2 Environment Description

The Taxi problem involves navigating in a *5x5* grid world, which contains **four** special locations designated as pick-up and drop-off locations (they're identified with the colors *Red, Green, Yellow* and *Blue*). The goal of the Taxi is to pick-up the passenger and drop him off at one of the four designated locations

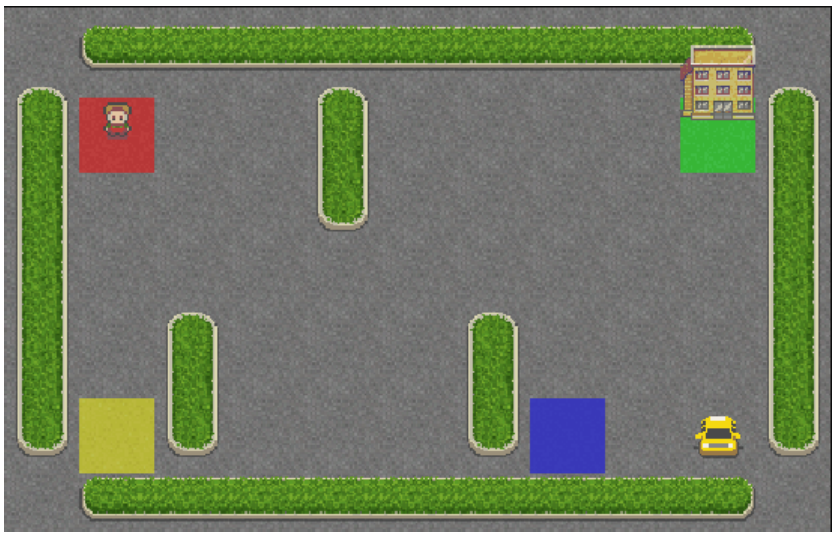

Figure 1: Environment map

Action Space

The action space is discrete and composed by **6** deterministic actions. Every action is identified by a corresponding integer number.

- 0: Move south (down)
- 1: Move north (up)
- 2: Move east (right)
- 3: Move west (left)
- 4: Pickup passenger
- 5: Drop off passenger

Some particular actions are considered to be **illegal** depending on the current state of the environment (for instance trying to pick up a passenger in a location that is not designated for pick-up or trying to move into a wall). Illegal actions will not change the state of the environment

Observation Space

Since there are **25** taxi positions, **5** possible locations of the passenger (including the case when the passenger is in the taxi) and **4** destination locations we have in total **500** discrete states.

Rewards

We have three possible rewards that we can receive:

- -1 : default reward; returned when a step is performed (expect when another reward is triggered).
- +20: delivering the passenger
- -10: executing "pickup" or "drop-off" actions illegally.

We must precise that the environment offers by default only **200** steps and the episode will end not only when the taxi drops off the passenger, but also when the maximum number of steps is reached. This means that we get a total reward of **-200** (assuming no illegal actions) if the agent is unable to successfully drop off the passenger.

Starting State

The Taxi starts off at a random square and the passenger at one of the designated special locations. The destination location is also randomly chosen among the remaining three special locations.

3 Solutions Adopted

As we have said, the solutions adopted can be divided into two categories:

- **Tabular Q-Learning**
- **Deep Q-Learning**

In what follows we will analyze both categories and how they can be implemented

3.1 Tabular Q-Learning

Tabular Q-Learning is a model-free reinforcement learning algorithm in which the core idea is to estimate the so called **Q-Values**, which represent the *expected utility* of taking a certain action in a given state. All the Q-Values are stored in a **Q-Table**: a matrix where each entry corresponds to a specific state-action pair *(s, a)*.

Such table is updated iteratively as the agent interacts with the environment and the update rule is given by the **Bellman Equation**:

$$Q(s,a) = r(s,a) + \gamma \max_{a' \in A} Q(s', a')$$

where:
- s: current state of the environment
- a: selected action
- *r(s,a)*: reward obtained by performing the action a in the current state s
- γ: discount factor
- *s'*: next state of the environment, obtained after having performed a
- *a'*: next action to execute in *s'*

However, we must precise that the choice of the action a can be performed with two different strategies:

- **Exploration**: pick a random action
- **Exploitation**: pick the best possible action i.e the action that gives us the highest Q-Value.

The solution that has been implemented will follow an ***ϵ-greedy* strategy**: we will initialize a parameter ϵ with value **1.0** and then choose a random action with probability ϵ or the best action with probability $1 - \epsilon$. Such parameter is called *exploration rate* and will also be subject to an *exponential* decay: we will give priority to exploration during the first episodes and then we will prioritize exploitation in the final phase.

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot e^{-(episode/\epsilon_{decay})}$$

In our case, since we have **500** reachable states and **6** possible actions, the Q-Table will have a dimension of *500 x 6 = 3000*. The update rule has also been modified with the addition of a learning rate α

The training has been performed with a discount factor value $\gamma = 0.99$, a learning rate $\alpha = 0.1$, an initial exploration rate $\epsilon_{max} = 1$, a minimum exploration rate $\epsilon_{min} = 0.01$, an exploration rate decay $\epsilon_{decay} = 400$ and a total number of **3000** training episodes. As we can see from the chart in figure 2 the tabular q-learning approach proved to be a successful solution for this particular environment. In fact, the total reward collected during each episode kept increasing, eventually converging to positive values.

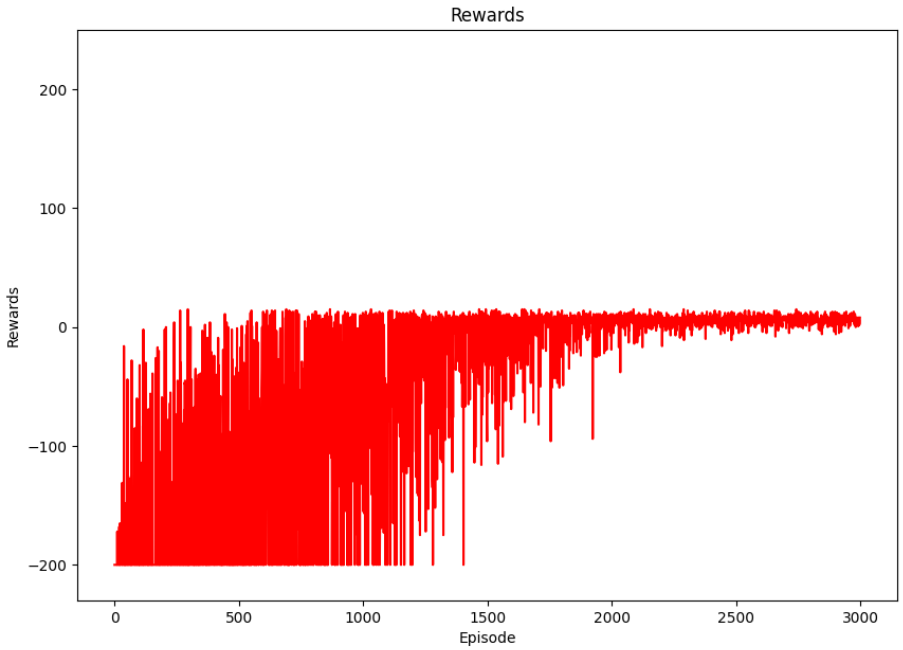

Figure 2: Episode vs total reward

3.2 Deep Q-Learning

The basic idea of **Deep Q-Learning** (**DQN**) is to remove the need to store the Q-Values into a table but instead combine the Q-Learning algorithm with an **artificial neural network**. Such network will take as input the current state of the environment and its output will be an estimation of all the Q-Values related to the input state (which correspond to an entire row of the Q-Table). We will keep using the same strategy for the action selection and the formula that will be used to update the Q-Values remains the same.

An important data structure that has been used with this approach is the **experience**

memory: such memory is essentially a queue in which we will insert tuples of the form:

(state, action, reward, next state, done)

This data structure is essential for DQN since it will be the main datasource that will be used by the network to perform the training. This implies that some initial episodes will be dedicated for the memory population: we can't perform the network training if the memory doesn't have enough tuples. During such episodes the training isn't performed and all the actions are picked randomly (in order to compose some tuples to insert into the memory).

Everytime we need to train the network the following steps will be performed:

1. Pick a random batch of data from the experience memory
2. For each state of the environment inside the batch, use the network to predict all the related Q-Values
3. For each element of the batch use the **Bellman equation** to obtain the updated Q-Values.
4. Compare the Q-Values that has been predicted by the network with the updated Q-Values obtained from the Bellman equation, which will represent the target of the training step i.e the new values that we want to obtain from the network prediction.

We will now see three different solutions that has been implemented with the usage of a DQN:

3.2.1 First Solution

For the first solution the network has been composed as follows:

1. The input layer is an **embedding** layer that will be used to transform each integer input value (state) into a vector of size **4**. This is particular useful to improve the overall efficiency since this layer will reduce the dimensionality of the input data (avoid using a 500-dimensional one-hot vector)
2. The middle layers are composed by two fully connected layers with **25** units and the **relu** activation function.
3. The output layer is a fully connected layer with **6** units (size of the action space)

The selected optimizer is **Adam** and the selected loss function is **Huber**. The training has been performed with a batch size of **128** and a learning rate equal to **0.001** and a total number of **6000** episodes (all the other parameters are the same as the ones selected for the Tabular Q-Learning solution).

Unfortunately, this approach was not successful. In fact, as we can see from the chart below in figure 3, the training showed a huge amount of fluctuations and the network wasn't able to fully converge to positive values. The total reward indeed improved, showing that the network started to reduce the amount of illegal actions, but the result is still not satisfactory.

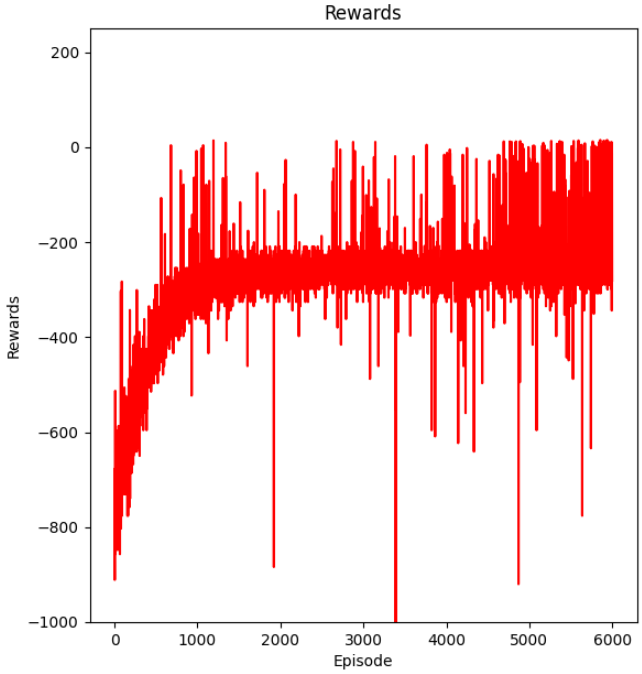

Figure 3: Episode vs total reward

3.2.2 Second solution

The second solution is based on the following key observation: since we are using a single DQN to approximate the Q-function, such network is constantly trained and also used for inference for the **bellman equation** to estimate the new Q-Values that the network should predict. We can observe from the figure 3 that during the first episodes a lot of illegal actions are selected since the total reward is strictly lower than **-200** (it's the phase where we prioritize the exploration). This means that this huge amount of illegal actions may have a negative effect on the single DQN and increase the amount of fluctuations.

The major modification of this solution is given by the introduction of a **second DQN** which we will refer to as the **target network**. The first DQN, which we will now call the **training network**, will still be subject to the constant training process but now the inference for the bellman equation will be performed by the target network. Additionally, now we will define a new parameter called *target update frequency*: it will be used to define when we will update the target network weights with the weights

of the training network.

Some additional modifications have been introduced:

- The DQN model has been modified: now the two middle layers are composed by 50 units each
- Now the learning rate is also subject to an exponential decay (the formula is the same as the one used for the exploration rate decay)
- For performances reasons, the number of steps have been limited to **100**

As we can see in the chart below in figure 4, now the training process proved to be more effective. The network managed to converge to values close to the 0, implying that it was able to complete with success several episodes.

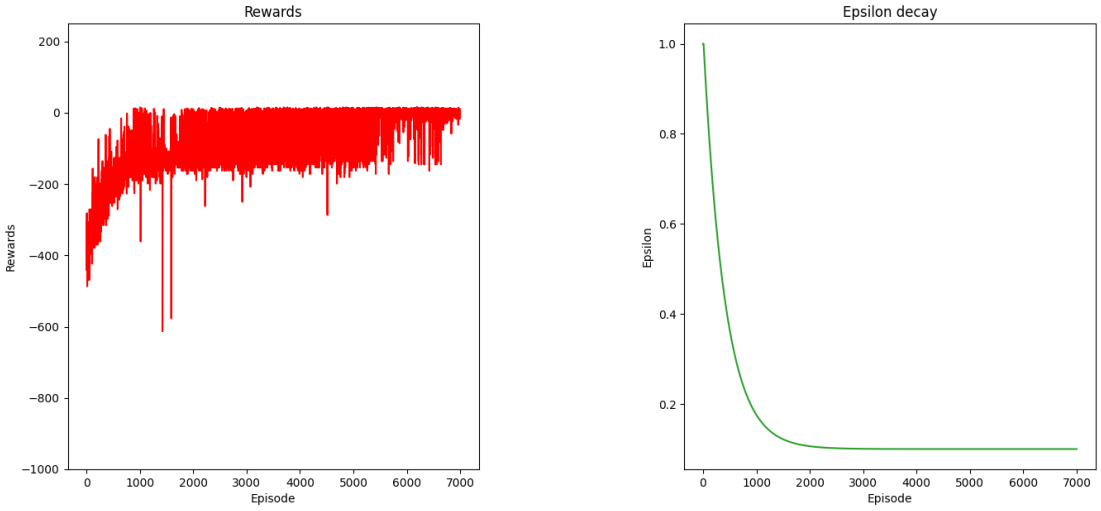

Figure 4: Episode vs total reward

4 Implementation

In what follows we will see the actual implementation of all the solutions that we have discussed.

4.1 Dependencies and Hyperparameters

```
import gymnasium as gym
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import random

from pathlib import Path
from collections import deque
from types import SimpleNamespace
from tqdm import trange
from tqdm import tqdm
```

```
tabular_q_learning_props = SimpleNamespace(
    learning_rate = 0.1,
    discount_factor = 0.99,
    exploration_rate = 1.0,
    exploration_rate_decay = 400,
    minimum_exploration_rate = 0.01,
    num_episodes = 3000,
    max_steps_per_episode = 200
)

deep_q_network_props = SimpleNamespace(
    network = SimpleNamespace(
        initial_lr = 0.001,
        minimum_lr = 0.0001,
        lr_decay = 5000,
        batch_size = 128
    ),
    env = SimpleNamespace(
        num_episodes = 7000,
        max_steps_per_episode = 100
    ),
    rl = SimpleNamespace(
        discount_factor = 0.99,
        exploration_rate = 1,
        minimum_exploration_rate = 0.1,
        exploration_rate_decay = 400,
        memory_size = 5000,
        train_start = 1000,
        target_network_update_freq = 20
    )
)
```

4.2 Tabular Q-Learning

4.2.1 Variables Setup

```
env = gym.make('Taxi-v3')

#Initialize Q-Table. It will be a table with dimension 500 x 6
q_table = np.zeros((env.observation_space.n, env.action_space.n))

#Get configuration parameters
num_episodes = tabular_q_learning_props.num_episodes
max_steps = tabular_q_learning_props.max_steps_per_episode
alpha = tabular_q_learning_props.learning_rate
gamma = tabular_q_learning_props.discount_factor
epsilon = tabular_q_learning_props.exploration_rate
epsilon_min = tabular_q_learning_props.minimum_exploration_rate
epsilon_decay = tabular_q_learning_props.exploration_rate_decay

success_list = []
total_rewards_list = []
```

4.2.2 Training

```
for episode in tqdm(range(num_episodes), desc="Executing"):
    state, info = env.reset()   #Start the environment and obtain the initial state
    completed = False
    current_step = 0
    total_reward = 0

    while not completed and current_step < max_steps:
        action_mask = info.get('action_mask')
        valid_actions = np.where(action_mask == 1)[0]

        if np.random.uniform(0, 1) < epsilon:
            action = np.random.choice(valid_actions)  #Exploration: pick a random action
        else:
            q_values_valid_actions = q_table[state, valid_actions]
            action = valid_actions[np.argmax(q_values_valid_actions)]  #Exploitation: choose the action with the highest Q-Value

        next_state, reward, done, truncated, new_info = env.step(action)     #Execute the action
        total_reward += reward

        #Update the Q-Value
        best_next_action = np.argmax(q_table[next_state, :])
        q_table[state, action] = q_table[state, action] + alpha * (reward + gamma * q_table[next_state, best_next_action] - q_table[state, action])

        state = next_state
        info = new_info
        current_step += 1

        if done or truncated:
            if not truncated:
                success_list.append(episode)
            completed = True

    epsilon = epsilon_min + (tabular_q_learning_props.exploration_rate - epsilon_min) * np.exp(-episode / epsilon_decay)
    total_rewards_list.append(total_reward)
```

4.2.3 Evaluation

```
new_env = gym.make('Taxi-v3')

for episode in range(10):
    print(f"Episode number: {episode + 1}")
    state, info = new_env.reset()

    for _ in tqdm(range(200)):
        action_mask = info.get('action_mask')
        valid_actions = np.where(action_mask == 1)[0]

        q_values_valid_actions = q_table[state, valid_actions]
        action = valid_actions[np.argmax(q_values_valid_actions)]

        next_state, reward, terminated, truncated, new_info = new_env.step(action)

        state = next_state
        info = new_info

        if terminated or truncated:
            if truncated:
                print(f"Insuccess for episode {episode + 1}")
            else:
                print(f"Success for episode {episode + 1}")

            break

new_env.close()
```

4.3 Deep Q-Learning - First Solution

4.3.1 Model Definition

```
class RLModel(nn.Module):

    def __init__(self, input_size, output_size):
        super(RLModel, self).__init__()

        self.embedding = nn.Embedding(input_size, 4)
        self.fc1 = nn.Linear(4, 25)
        self.fc2 = nn.Linear(25, 25)
        self.output = nn.Linear(25, output_size)

    def forward(self, x):
        x = self.embedding(x)
        x = nn.functional.relu(self.fc1(x))
        x = nn.functional.relu(self.fc2(x))
        x = self.output(x)
        return x
```

4.3.2 Functions

```
def choose_action(model, state: int, exploration_rate: float):
    if np.random.uniform(0, 1) < exploration_rate:
        return env.action_space.sample()        # Exploration: pick a random action

    with torch.no_grad():
        q_values = model(torch.tensor([state], device=torch.device('cpu')))     #Exploitation: pick the action with the highest q_value
        return q_values.max(1)[1].item()
```

```
def plot_chart(total_reward_list, epsilon_list):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))
    plt.subplots_adjust(wspace=0.5)

    plt.title("Training result")

    ax1.set_xlabel('Episode')
    ax1.set_ylabel('Rewards')
    ax1.set_ylim(-1000, 250)
    ax1.set_title("Rewards")
    ax1.plot(total_reward_list, color="red")

    ax2.set_xlabel('Episode')
    ax2.set_ylabel('Epsilon')
    ax2.set_title('Epsilon decay')
    ax2.plot(epsilon_list, color="C2")

    plt.show()
```

```python
def train_model(model, optimizer, loss_fn, memory):
    if len(memory) < deep_q_network_props.network.batch_size:
        return

    batch = random.sample(memory, deep_q_network_props.network.batch_size)
    states, actions, rewards, next_states, dones = zip(*batch)

    state_batch = torch.tensor(states, device=torch.device('cpu'))
    action_batch = torch.tensor(actions, device=torch.device('cpu'), dtype=torch.long)
    reward_batch = torch.tensor(rewards, device=torch.device('cpu'))
    next_state_batch = torch.tensor(next_states, device=torch.device('cpu'))
    done_batch = torch.tensor(dones, device=torch.device('cpu'), dtype=torch.bool)

    # Compute current q_values prediction
    current_q_values = model(state_batch).gather(1, action_batch.unsqueeze(1))

    # Compute expected q_values that the network should predict
    expected_q_values = reward_batch + (torch.logical_not(done_batch) * deep_q_network_props.rl.discount_factor * model(next_state_batch).max(1)[0])

    loss = loss_fn(current_q_values, expected_q_values.unsqueeze(1))

    optimizer.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step()
```

4.3.3 Variables Setup

```python
env = gym.make('Taxi-v3')
state_size = env.observation_space.n        # 500 states
action_size = env.action_space.n            # 6 actions

model = RLModel(state_size, action_size).to(torch.device('cpu'))
optimizer = torch.optim.Adam(model.parameters(), lr=deep_q_network_props.network.initial_lr)
loss_fn = torch.nn.functional.smooth_l1_loss

memory = deque(maxlen=deep_q_network_props.rl.memory_size)
epsilon = deep_q_network_props.rl.exploration_rate
minimum_epsilon = deep_q_network_props.rl.minimum_exploration_rate
epsilon_decay = deep_q_network_props.rl.exploration_rate_decay

total_reward_list = []
epsilon_list = []
successful_episodes = []
```

4.3.4 Training

```
progress_bar = trange(0, deep_q_network_props.env.num_episodes)

for episode in progress_bar:
    state, _ = env.reset()
    total_reward = 0

    if len(memory) >= deep_q_network_props.rl.train_start:
        epsilon = minimum_epsilon + (deep_q_network_props.rl.exploration_rate - minimum_epsilon) * np.exp(-episode / epsilon_decay)

    for current_step in range(0, 200):
        action = choose_action(model, state, epsilon)

        next_state, reward, done, truncated, _ = env.step(action)

        if len(memory) > deep_q_network_props.rl.memory_size:
            memory.popleft()

        memory.append([*[state, action, reward, next_state, done]])

        if len(memory) >= deep_q_network_props.rl.train_start:
            train_model(model, optimizer, loss_fn, memory)

        state = next_state
        total_reward += reward

        if done or truncated:
            if not truncated:
                successful_episodes.append(episode)

            total_reward_list.append(total_reward)
            epsilon_list.append(epsilon)

            progress_bar.set_postfix({
                "reward": total_reward,
                "epsilon": epsilon
            })
            break
```

4.3.5 Evaluation

```python
def choose_action(model, state, info):
    action_mask = info.get('action_mask')

    with torch.no_grad():
        if np.random.uniform(0, 1) < 0.3:
            valid_actions = np.where(action_mask == 1)[0]
            return np.random.choice(valid_actions)
        else:
            predicted = model(torch.tensor([state], device=torch.device('cpu')))
            q_values = predicted.cpu().numpy()[0]
            q_values[action_mask == 0] = -float('inf')
            action = q_values.argmax()
            return action
```

```python
env = gym.make("Taxi-v3")
state_size = env.observation_space.n
action_size = env.action_space.n

for episode in range(10):
    print(f"Episode number: {episode + 1}")
    state, info = env.reset()
    total_reward = 0

    progress_bar = trange(0, 200, initial=0, total=200)

    for _ in progress_bar:
        action = choose_action(model, state, info)

        print(f"Selected action: {action}")

        next_state, reward, done, truncated, new_info = env.step(action)
        total_reward += reward

        state = next_state
        info = new_info

        if done or truncated:
            print(f"Total reward {total_reward}")
            if truncated:
                print(f"Insuccess for episode {episode + 1}")
            else:
                print(f"Success for episode {episode + 1}")

            break

env.close()
```

4.4 Deep Q-Learning - Second solution

4.4.1 Model Definition

```python
class RLModel(nn.Module):

    def __init__(self, input_size, output_size):
        super(RLModel, self).__init__()

        self.emb = nn.Embedding(input_size, 4)
        self.l1 = nn.Linear(4, 50)
        self.l2 = nn.Linear(50, 50)
        self.l3 = nn.Linear(50, output_size)

    def forward(self, x):
        x = self.emb(x)
        x = nn.functional.relu(self.l1(x))
        x = nn.functional.relu(self.l2(x))
        x = self.l3(x)
        return x
```

4.4.2 Functions

The functions *choose_action* and *plot_chart* are equal to the ones used in the first solution

```python
def train_model(model, target_model, optimizer, loss_fn, memory):
    if len(memory) < deep_q_network_props.network.batch_size:
        return

    batch = random.sample(memory, deep_q_network_props.network.batch_size)
    states, actions, rewards, next_states, dones = zip(*batch)

    state_batch = torch.tensor(states, device=torch.device('cpu'))
    action_batch = torch.tensor(actions, device=torch.device('cpu'), dtype=torch.long)
    reward_batch = torch.tensor(rewards, device=torch.device('cpu'))
    next_state_batch = torch.tensor(next_states, device=torch.device('cpu'))
    done_batch = torch.tensor(dones, device=torch.device('cpu'), dtype=torch.bool)

     # Compute predicted Q values
    predicted_q_value = model(state_batch).gather(1, action_batch.unsqueeze(1))

    # Use Bellman equation to compute new expected values
    expected_q_values = reward_batch + (torch.logical_not(done_batch) * target_model(next_state_batch).max(1)[0] * 0.99)

    # Compute loss
    loss = loss_fn(predicted_q_value, expected_q_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in model.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

```
def decay_learning_rate(optimizer, episode):
  minimum_lr = deep_q_network_props.network.minimum_lr
  new_lr = minimum_lr + (deep_q_network_props.network.initial_lr - minimum_lr) * np.exp(- episode / deep_q_network_props.network.lr_decay)
  for elem in optimizer.param_groups:
      elem['lr'] = new_lr
```

4.4.3 Variables Setup

```
env = gym.make('Taxi-v3')
state_size = env.observation_space.n        # 500 states
action_size = env.action_space.n            # 6 actions

model = RLModel(state_size, action_size).to(torch.device('cpu'))
optimizer = torch.optim.Adam(model.parameters(), lr=deep_q_network_props.network.initial_lr)
loss_fn = torch.nn.functional.smooth_l1_loss

target_model = RLModel(state_size, action_size).to(torch.device('cpu'))
target_model.load_state_dict(model.state_dict())
target_model.eval()


memory = deque(maxlen=deep_q_network_props.rl.memory_size)
epsilon = deep_q_network_props.rl.exploration_rate
minimum_epsilon = deep_q_network_props.rl.minimum_exploration_rate
epsilon_decay = deep_q_network_props.rl.exploration_rate_decay

total_reward_list = []
epsilon_list = []
```

4.4.4 Training

```
for episode in progress_bar:
    state, _ = env.reset()
    total_reward = 0

    if len(memory) >= deep_q_network_props.rl.train_start:
        epsilon = minimum_epsilon + (deep_q_network_props.rl.exploration_rate - minimum_epsilon) * np.exp(-episode / epsilon_decay)

    for current_step in range(0, deep_q_network_props.env.max_steps_per_episode):
        action = choose_action(model, state, epsilon)

        next_state, reward, done, truncated, _ = env.step(action)

        if len(memory) > deep_q_network_props.rl.memory_size:
            memory.popleft()

        memory.append([*[state, action, reward, next_state, done]])

        if len(memory) >= deep_q_network_props.rl.train_start:
            train_model(model, target_model, optimizer, loss_fn, memory)
            decay_learning_rate(optimizer, episode)

        state = next_state
        total_reward += reward

        if done and not truncated:
            successful_episodes.append(episode)

        done = done or (current_step == deep_q_network_props.env.max_steps_per_episode - 1)

        if done or truncated:
            total_reward_list.append(total_reward)
            epsilon_list.append(epsilon)

            progress_bar.set_postfix({
                "reward": total_reward,
                "epsilon": epsilon
            })
            break

    if episode % deep_q_network_props.rl.target_network_update_freq == 0:
        target_model.load_state_dict(model.state_dict())
```

4.4.5 Evaluation

Same as the evaluation of the first Deep Q-Learning solution