Deep Q-Network (DQN) 2013

Jae Yun JUN KIM*

Due: Before the beginning of next lab session
Evaluation: Code, results, and explanation about the code
Remark:

- Only groups of two or three people accepted (preferably three).
- No plagiarism. If plagiarism happens, both the "lender" and the "borrower" will have a zero.
- Do thoroughly all the demanded tasks.
- Study the theory for the questions.

Source: Andrew Gordienko's Reinforcement learning: DQN with PyTorch (https://andrew-gordienko.medium.com/reinforcement-learning-dqn-w-pytorch-7c6faad3d1e)

1 Introduction

In this lab session, we would like to control an inverted pendulum attached to a cart as shown in Figure 1

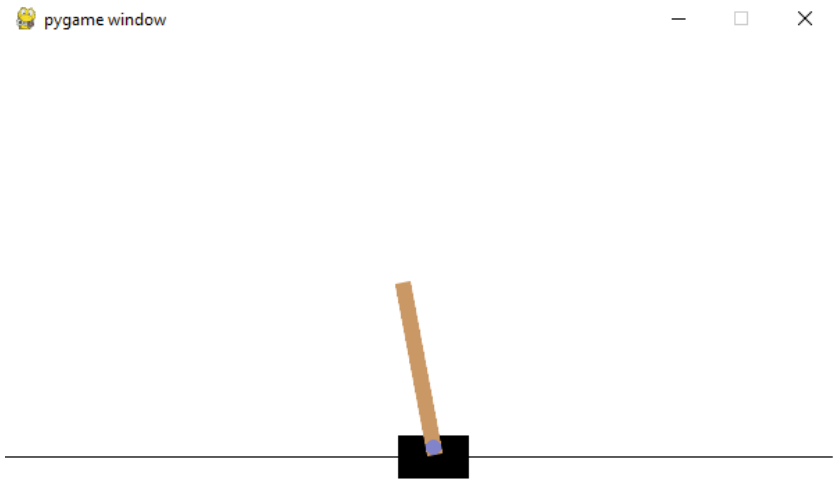

Figure 1: An inverted pendulum attached to a cart

Important note: **Use Python 3.7** for this pre-lab.

*ECE Paris Graduate School of Engineering, 10 Rue Sextius Michel 75015 Paris, France; jae-yun.jun-kim@ece.fr

2 Task 1: Installation of associated Python modules

You need to install the following modules if you have not them installed yet:

- **gym**
- **torch**
- **matplotlib**
- **gym[classic_control]**

3 Task 2: Implement DQN 2013

Code the following Python script.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import math
import gym
import random
import numpy as np
import matplotlib.pyplot as plt

# env = gym.make('CartPole-v1', render_mode = 'human')
env = gym.make('CartPole-v1')
observation_space = env.observation_space.shape[0]
action_space = env.action_space.n

EPISODES = 1000 # 10 # 100 # 1000
LEARNING_RATE = 0.0001
MEM_SIZE = 10000
BATCH_SIZE = 64
GAMMA = 0.95
EXPLORATION_MAX = 1.0
EXPLORATION_DECAY = 0.999
EXPLORATION_MIN = 0.001

FC1_DIMS = 1024
FC2_DIMS = 512
DEVICE = torch.device("cpu")

best_reward = 0
average_reward = 0
episode_number = []
average_reward_number = []

class Network(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.input_shape = env.observation_space.shape
        self.action_space = action_space

        self.fc1 = nn.Linear(*self.input_shape, FC1_DIMS)
        self.fc2 = nn.Linear(FC1_DIMS, FC2_DIMS)
        self.fc3 = nn.Linear(FC2_DIMS, self.action_space)

        self.optimizer = optim.Adam(self.parameters(), lr=LEARNING_RATE)
        self.loss = nn.MSELoss()
        self.to(DEVICE)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```

```
        return x

class ReplayBuffer:
    def __init__(self):
        self.mem_count = 0

        self.states = np.zeros((MEM_SIZE, *env.observation_space.shape),dtype=np.float32)
        self.actions = np.zeros(MEM_SIZE, dtype=np.int64)
        self.rewards = np.zeros(MEM_SIZE, dtype=np.float32)
        self.states_ = np.zeros((MEM_SIZE, *env.observation_space.shape),dtype=np.float32)
        self.dones = np.zeros(MEM_SIZE, dtype=np.bool_)

    def add(self, state, action, reward, state_, done):
        mem_index = self.mem_count % MEM_SIZE

        self.states[mem_index]  = state
        self.actions[mem_index] = action
        self.rewards[mem_index] = reward
        self.states_[mem_index] = state_
        self.dones[mem_index] =  1 - done

        self.mem_count += 1

    def sample(self):
        MEM_MAX = min(self.mem_count, MEM_SIZE)
        batch_indices = np.random.choice(MEM_MAX, BATCH_SIZE, replace=True)

        states  = self.states[batch_indices]
        actions = self.actions[batch_indices]
        rewards = self.rewards[batch_indices]
        states_ = self.states_[batch_indices]
        dones   = self.dones[batch_indices]

        return states, actions, rewards, states_, dones

class DQN_Solver:
    def __init__(self):
        self.memory = ReplayBuffer()
        self.exploration_rate = EXPLORATION_MAX
        self.network = Network()

    def choose_action(self, observation):
        if random.random() < self.exploration_rate:
            return env.action_space.sample()

        state = torch.tensor(observation).float().detach()
        state = state.to(DEVICE)
        state = state.unsqueeze(0)
        q_values = self.network(state)
        return torch.argmax(q_values).item()

    def learn(self):
        if self.memory.mem_count < BATCH_SIZE:
            return

        states, actions, rewards, states_, dones = self.memory.sample()
        states = torch.tensor(states , dtype=torch.float32).to(DEVICE)
        actions = torch.tensor(actions, dtype=torch.long).to(DEVICE)
        rewards = torch.tensor(rewards, dtype=torch.float32).to(DEVICE)
        states_ = torch.tensor(states_, dtype=torch.float32).to(DEVICE)
        dones = torch.tensor(dones, dtype=torch.bool).to(DEVICE)
        batch_indices = np.arange(BATCH_SIZE, dtype=np.int64)

        q_values = self.network(states)
        next_q_values = self.network(states_)

        predicted_value_of_now = q_values[batch_indices, actions]
        predicted_value_of_future = torch.max(next_q_values, dim=1)[0]

        q_target = rewards + GAMMA * predicted_value_of_future * dones

        loss = self.network.loss(q_target, predicted_value_of_now)
        self.network.optimizer.zero_grad()
```

```
        loss.backward()
        self.network.optimizer.step()

        self.exploration_rate *= EXPLORATION_DECAY
        self.exploration_rate = max(EXPLORATION_MIN, self.exploration_rate)

    def returning_epsilon(self):
        return self.exploration_rate

agent = DQN_Solver()

for i in range(1, EPISODES):
    state = env.reset()
    #print(state[0])
    #print(observation_space)

    state = np.reshape(state[0], [1, observation_space])
    # print(state.shape)
    score = 0

    # for itera in range(1):
    while True:
        # env.render()
        action = agent.choose_action(state)
        aux = env.step(action)
        #print(aux)
        state_, reward, done, info, _ = env.step(action)
        #print(state_.shape)
        #print(observation_space)
        state_ = np.reshape(state_, [1, observation_space])
        agent.memory.add(state, action, reward, state_, done)
        agent.learn()
        state = state_
        score += reward

        if done:
            if score > best_reward:
                best_reward = score
            average_reward += score
            print("Episode {} Average Reward {} Best Reward {} Last Reward {} Epsilon {}".format(
            i, average_reward/i, best_reward, score, agent.returning_epsilon()))
            break

        episode_number.append(i)
        average_reward_number.append(average_reward/i)

plt.plot(episode_number, average_reward_number)
plt.show()
```

Verify that the code works. If not, try to solve the possible errors.

4 Task 3: Show results

If the code runs correctly, you should see that the algorithm returns the score convergence illustrated in Figure 4.

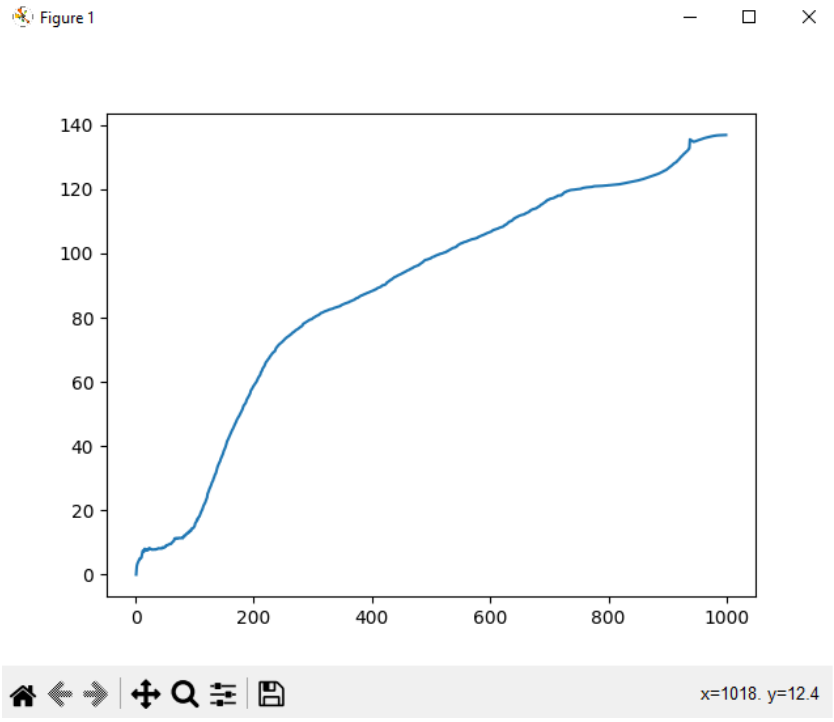

Figure 2: Score convergence

Show these results to the professor.

5 Task 4: Test the optimized model

Once you have optimized the DQN, test this optimized model by modifying the above code and by showing the corresponding animation.