VILNIUS UNIVERSITY

ŠIAULIAI ACADEMY

BACHELOR PROGRAMME SOFTWARE ENGINEERING

Artificial Intelligence

Report on

“DQN Method” task

Student: Anna Kutova

Lecturer: Dr. Gintautas Daunys

Šiauliai, 2023

TABLE OF CONTENTS

1. Used code with comments

```
# Importing necessary libraries and modules
import torch
import torch.nn as nn
import torch.optim as optim
import gym
import random
import math
import time
import matplotlib.pyplot as plt

# Checking if GPU is available and defining device accordingly
use_cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if use_cuda else "cpu")
Tensor = torch.Tensor
LongTensor = torch.LongTensor

# Creating the CartPole environment
env = gym.make('CartPole-v0')

# Setting random seeds for reproducibility
seed_value = 23
torch.manual_seed(seed_value)
random.seed(seed_value)

# Setting hyperparameters and parameters
learning_rate = 0.02
num_episodes = 100
gamma = 1
hidden_layer = 64
replay_mem_size = 50000
batch_size = 32
egreedy = 0.9
egreedy_final = 0
egreedy_decay = 500
report_interval = 10
score_to_solve = 195

# Defining the number of inputs and outputs based on the environment
number_of_inputs = env.observation_space.shape[0]
number_of_outputs = env.action_space.n

# Function to calculate epsilon for epsilon-greedy policy
def calculate_epsilon(steps_done):
    epsilon = egreedy_final + (egreedy - egreedy_final) * math.exp(-1. *
steps_done / egreedy_decay)
    return epsilon

# Definition of the Neural Network class
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.linear1 = nn.Linear(number_of_inputs, hidden_layer)
        self.linear2 = nn.Linear(hidden_layer, number_of_outputs)
        self.activation = nn.Tanh()  # Using hyperbolic tangent as the
activation function

    def forward(self, x):
```

```
        output1 = self.linear1(x)
        output1 = self.activation(output1)
        output2 = self.linear2(output1)
        return output2

# Definition of the QNet_Agent class
class QNet_Agent(object):
    def __init__(self):
        self.nn = NeuralNetwork().to(device)
        self.loss_func = nn.MSELoss()
        self.optimizer = optim.Adam(params=self.nn.parameters(),
lr=learning_rate)

    def select_action(self, state, epsilon):
        random_for_egreedy = torch.rand(1)[0]

        if random_for_egreedy > epsilon:
            with torch.no_grad():
                state = Tensor(state).to(device)
                action_from_nn = self.nn(state)
                action = torch.max(action_from_nn, 0)[1]
                action = action.item()
        else:
            action = env.action_space.sample()

        return action

    def optimize(self):
        if len(memory) < batch_size:
            return

        state, action, new_state, reward, done =
memory.sample(batch_size)
        state = Tensor(state).to(device)
        new_state = Tensor(new_state).to(device)
        reward = Tensor(reward).to(device)
        action = LongTensor(action).to(device)
        done = Tensor(done).to(device)

        new_state_values = self.nn(new_state).detach()
        max_new_state_values = torch.max(new_state_values, 1)[0]
        target_value = reward + (1 - done) * gamma *
max_new_state_values

        predicted_value = self.nn(state).gather(1,
action.unsqueeze(1)).squeeze(1)

        loss = self.loss_func(predicted_value, target_value)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

# Definition of the ExperienceReplay class
class ExperienceReplay(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, state, action, new_state, reward, done):
```

```
        transition = (state, action, new_state, reward, done)

        if self.position >= len(self.memory):
            self.memory.append(transition)
        else:
            self.memory[self.position] = transition
        self.position = (self.position + 1) % self.capacity

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

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

# Creating an instance of the ExperienceReplay class
memory = ExperienceReplay(replay_mem_size)
# Creating an instance of the QNet_Agent class
qnet_agent = QNet_Agent()
qnet_agent.nn.eval()

# Setting up the CartPole environment for rendering
env = gym.make('CartPole-v1', render_mode='human')

# Running a loop for a few episodes to visualize the agent's behavior
for episode in range(10):
    state, _ = env.reset()
    time.sleep(1.)
    for step in range(10000):
        env.render()
        time.sleep(0.02)
        action = qnet_agent.select_action(state, 0)
        new_state, reward, done, _, _ = env.step(action)
        state = new_state
        if done:
            print('Finished after steps:', step)
            break

# Closing the CartPole environment
env.close()
env.env.close()

# Resetting the CartPole environment for training
env = gym.make('CartPole-v0')

# Setting the QNet_Agent to training mode
qnet_agent.nn.train()

# Lists to store the number of steps and frames for each episode
steps_total = []
frames_total = 0
solved_after = 0
solved = False

# Timing the training process
start_time = time.time()

# Main training loop
for i_episode in range(num_episodes):
    state, _ = env.reset()
    step = 0
```

```
    # Inner loop for each episode
    while True:
        step += 1
        frames_total += 1
        epsilon = calculate_epsilon(frames_total)
        action = qnet_agent.select_action(state, epsilon)
        new_state, reward, done, _, _ = env.step(action)
        memory.push(state, action, new_state, reward, done)
        qnet_agent.optimize()
        state = new_state

        # Check if the episode is done
        if done:
            steps_total.append(step)

            # Calculate the mean reward over the last 100 episodes
            mean_reward_100 = sum(steps_total[-100:]) / 100

            # Check if the environment is considered solved
            if mean_reward_100 > score_to_solve and not solved:
                print("SOLVED! After %i episodes " % i_episode)
                solved_after = i_episode
                solved = True

            # Printing the training progress at regular intervals
            if i_episode % report_interval == 0:
                print("\n*** Episode %i *** \
                      \nAv.reward: [last %i]: %.2f, [last 100]: %.2f,
[all]: %.2f \
                      \nepsilon: %.2f, frames_total: %i"
                      %
                      (i_episode,
                       report_interval,
                       sum(steps_total[-report_interval:]) /
report_interval,
                       mean_reward_100,
                       sum(steps_total) / len(steps_total),
                       epsilon,
                       frames_total
                       )
                      )

                elapsed_time = time.time() - start_time
                print("Elapsed time: ", time.strftime("%H:%M:%S",
time.gmtime(elapsed_time)))

            break

# Saving the trained model's state dictionary to a file
state_dict = qnet_agent.nn.state_dict()
torch.save(state_dict, 'dqn_er.pth')

# Displaying the average reward statistics
print("\n\n\n\nAverage reward: %.2f" % (sum(steps_total) /
num_episodes))
print("Average reward (last 100 episodes): %.2f" % (sum(steps_total[-
100:]) / 100))
if solved:
    print("Solved after %i episodes" % solved_after)

# Plotting the rewards over episodes
```

```
plt.figure(figsize=(12, 5))
plt.title("Rewards")
plt.bar(torch.arange(len(steps_total)), steps_total, alpha=0.6,
color='green', width=5)
plt.show()

# Closing the CartPole environment
env.close()
env.env.close()

# Loading the trained model's state dictionary from the file
state_dict = torch.load('dqn_er.pth')
qnet_agent.nn.load_state_dict(state_dict)
qnet_agent.nn.eval()

# Setting up the CartPole environment for rendering
env = gym.make('CartPole-v1', render_mode='human')

# Running a loop for a few episodes to visualize the trained agent's
behavior
for episode in range(10):
    state, _ = env.reset()
    time.sleep(1.)
    for step in range(10000):
        env.render()
        time.sleep(0.02)
        action = qnet_agent.select_action(state, 0)
        new_state, reward, done, _, _ = env.step(action)
        state = new_state
        if done:
            print('Finished after steps:', step)
            break

# Closing the CartPole environment
env.close()
env.env.close()
```

2. Code explanation

In this code is demonstrated the implementation of the Deep Q-Network (DQN) algorithm to master the CartPole-v1 environment in OpenAI Gym.

Beginning with the necessary imports for deep learning, reinforcement learning, and visualization, the code dynamically checks for GPU availability and adapts device settings accordingly.

The CartPole environment is initialized, and random seeds are strategically established to ensure reproducibility. Key hyperparameters, including learning rate, episode count, and exploration rate, are meticulously defined.

The neural network architecture, implemented using PyTorch, features two fully connected layers with a hyperbolic tangent activation function. Orchestrating the Q-network, loss function, and optimization, the QNet_Agent class plays a pivotal role.

Action selection adheres to an epsilon-greedy policy, balancing exploration and exploitation. The ExperienceReplay class takes center stage in storing and selectively sampling experiences to optimize training efficiency.

The training loop systematically navigates through episodes, orchestrating interactions with the environment, accumulating experiences, and refining the Q-network. Regular intervals witness the portrayal of training progress, with the script preserving the trained model upon successful environment mastery.

Visualizing the average rewards across episodes contributes to a comprehensive understanding of training performance. Subsequently, the script reloads the trained model for assessment, inviting a visual journey into the game's proficiency across a curated set of test episodes.

In essence, the script encapsulates a holistic demonstration of the DQN methodology, shedding light on pivotal components such as experience replay, neural network architecture, and the delicate balance between exploration and exploitation within the CartPole-v1 gaming context.

3. Tuning hyperparameters

1. learning_rate: *This parameter controls the step size taken during optimization. A higher learning rate allows the model to learn faster, but if it's too high, the model might overshoot the optimal weights. Conversely, a lower learning rate may lead to slow convergence or getting stuck in a suboptimal solution.*

2. num_episodes: *The number of episodes determines how many times the model will interact with the environment for training. If the model is not converging, it may be needed to increase the number of episodes.*

3. gamma: *Gamma is the discount factor used in the Bellman equation, controlling the importance of future rewards. A higher gamma prioritizes long-term rewards, while a lower gamma prioritizes short-term rewards.*

4. hidden_layer: *This parameter controls the number of neurons in the model's hidden layer. If the model is not capturing the complexity of the problem, it may be needed to increase the number of neuron, on the other hand, if it's overfitting, it may be needed to decrease this number.*

5. replay_mem_size: *Replay memory size determines the size of the buffer used to store past experiences. If the model is not learning from past experiences, it may be needed to increase this size, on the other side, if it's overfitting, it can be needed to decrease it.*

6. batch_size: *Batch size determines the number of experiences used to update the model at each iteration. If the model is not learning from small batches, consider increasing this size. If it's overfitting, consider decreasing it.*

7. egreedy: *This parameter determines the probability of taking a random action in the epsilon-greedy policy. If the model is not exploring enough, consider increasing the egreedy value. If it's over-exploring and not exploiting enough, consider decreasing it.*

8. egreedy_final: *Egreedy final determines the minimum value of egreedy. If the model is not exploring enough at the end of training, consider decreasing this value, on the other hand, if it's over-exploring at the end, consider increasing it.*

9. egreedy_decay: *Egreedy decay determines the rate at which egreedy decreases over time. If the model is not exploring enough early in training, consider decreasing this value. If it's over-exploring early in training, consider increasing it.*

Here is, how I changed these parameters:

```
learning_rate = 0.02
num_episodes = 100
gamma = 1
hidden_layer = 64
replay_mem_size =
50000
batch_size = 32
egreedy = 0.9
egreedy_final = 0
egreedy_decay = 500
```

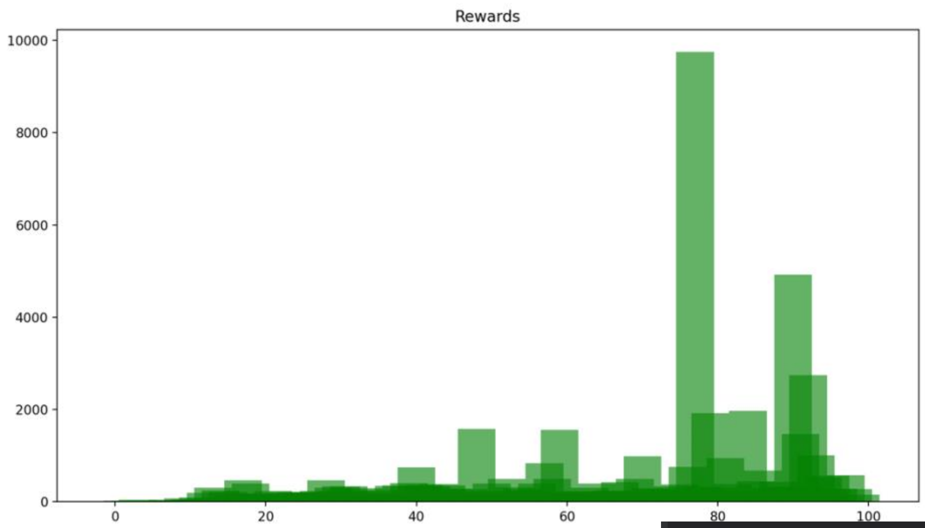
```
----->
```

```
learning_rate = 0.03
num_episodes = 50
gamma = 1
hidden_layer = 64
replay_mem_size =
75000
batch_size = 64
egreedy = 1
egreedy_final = 0
egreedy_decay = 250
```

4. Result Comparison

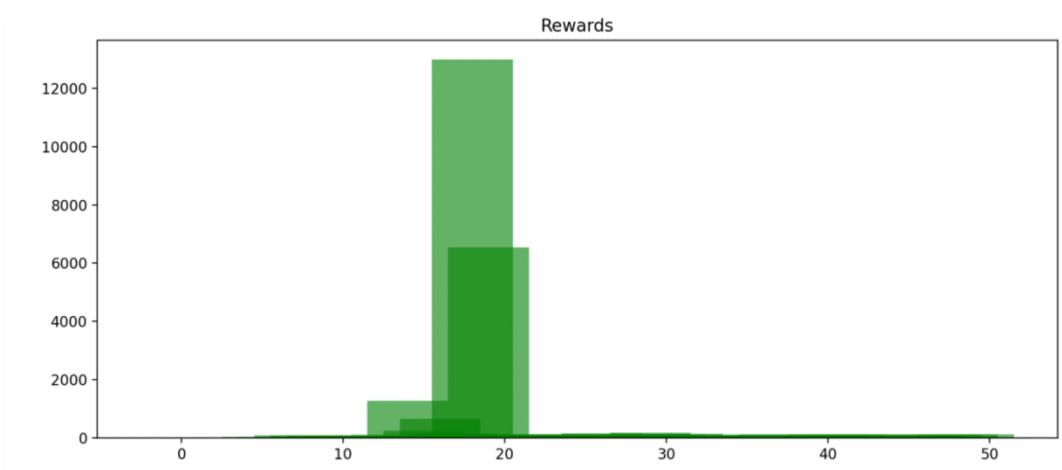
Here is a comparison of the results obtained from the initial model versus the results obtained after I adjusted certain parameters:

- **Initial reward statistics:**




```
Average reward: 547.69
Average reward (last 100 episodes): 547.69
Solved after 65 episodes
```

- **The reward statistics after tweaking some settings:**




```
Average reward: 523.32
Average reward (last 100 episodes): 261.66
Solved after 19 episodes
```

The game was successfully completed after only 19 episodes, as opposed to the original version, where it took 65 episodes.

5. Conclusion

To sum up, in the pursuit of implementing the Q-Learning algorithm for training an agent in the CartPole-v1 game from OpenAI Gym, this assignment has proven to be a valuable exploration of reinforcement learning principles. Delving into the intricacies of reward signals, I gained a nuanced understanding of their pivotal role in shaping the agent's behavior. The delicate balance between exploration and exploitation emerged as a critical factor in crafting an effective learning strategy.

Harnessing PyTorch as the primary tool for implementation provided not only a robust and flexible framework but also enabled efficient training of the Q-Learning model.

The tangible outcome of these efforts is evident in the achieved results. The fine-tuning of hyperparameters resulted in a significant reduction in the number of episodes required to solve the CartPole-v1 game. These enhancements underscore the practical implications of parameter optimization in reinforcement learning applications.

As a comprehensive exploration of reinforcement learning, this assignment has equipped me with a solid foundation applicable across diverse machine learning settings. The skills and insights garnered from this lab will undoubtedly serve as valuable assets in future endeavors within the realm of machine learning.