

**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*

[*Table of Contents* 2](#_Toc153225945)

[1. Used code with comments 3](#_Toc153225946)

[2. Code explanation 7](#_Toc153225947)

[3. Tuning hyperparameters 8](#_Toc153225948)

[4. Result Comparison 10](#_Toc153225949)

[ **Initial reward statistics:** 10](#_Toc153225950)

[ **The reward statistics after tweaking some settings:** 10](#_Toc153225951)

[5. Conclusion 11](#_Toc153225952)

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()

1. **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.

1. **Tuning hyperparameters**
2. 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.*
3. 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.*
4. 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.*
5. 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.*
6. 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.*
7. 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.*
8. 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.*
9. 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.*
10. 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:

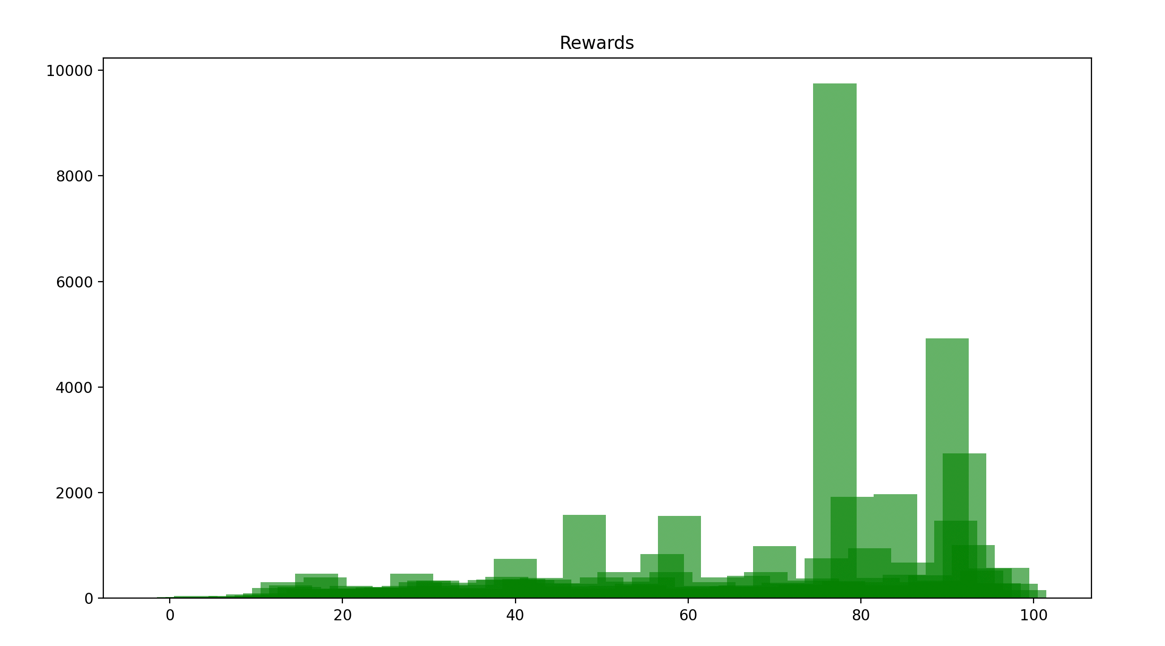
learning\_rate = 0.02  
num\_episodes = 100  
gamma = 1  
hidden\_layer = 64  
replay\_mem\_size = 50000  
batch\_size = 32  
egreedy = 0.9  
egreedy\_final = 0  
egreedy\_decay = 500

----->

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

1. **Result Comparison**

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

* **A screen shot of a black background

  Description automatically generatedInitial reward statistics:**
* A graph with green rectangles

  Description automatically generated**The reward statistics after tweaking some settings:**

A screen shot of a number

Description automatically generated

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

1. **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.*