МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Обучение с подкреплением»
Тема: Реализация DQN для среды CartPole-v1

Студент гр. 0310 _____ Волков К.А.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург
2025

СОДЕРЖАНИЕ

Цель работы

Реализовать алгоритм DQN для обучения агента в среде CartPole.

Задание

1. Реализовать алгоритм DQN;
2. Измените архитектуру нейросети (например, добавьте слои);
3. Попробуйте разные значения γ и ε_{decay};
4. Проведите исследование как изначальное значение ε влияет на скорость обучения.

Выполнение работы

3.1. Реализовать алгоритм DQN

Алгоритм DQN реализован с использованием библиотеки TensorFlow на языке Python.

Полный код представлен в приложении А

Ниже представлены стандартные параметры, которые использовались для обучения агентов.

```
hidden_layers=[64, 64],
gamma=0.99,
epsilon_start=1.0,
epsilon_end=0.01,
epsilon_decay=0.995,
learning_rate=0.001,
batch_size=64,
memory_size=10000,
target_update_freq=10):
```

Результаты представлены на рисунке 1.

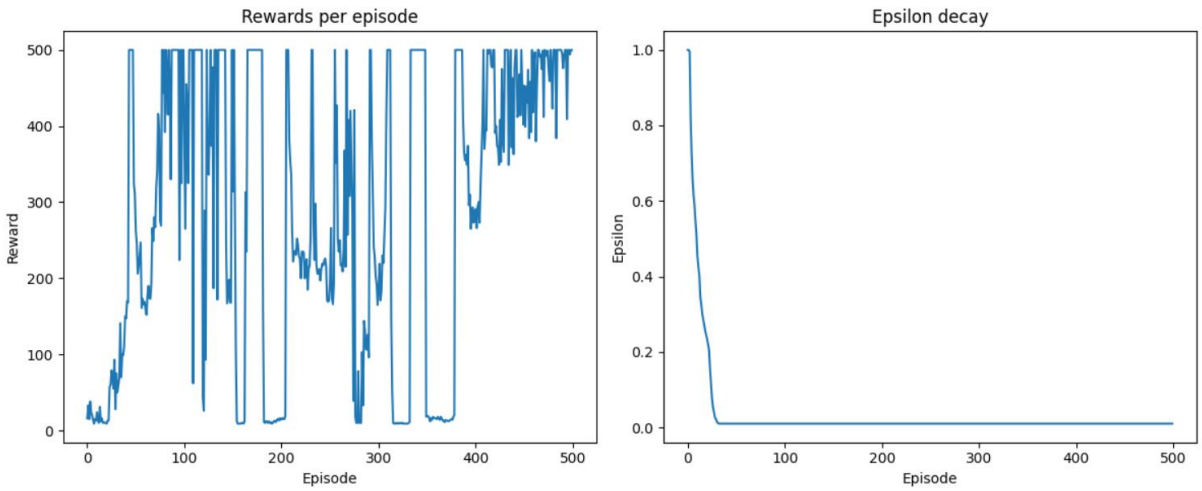

Рисунок 1 – Результаты обучения агента на стандартных параметрах

3.2. Измените архитектуру нейросети (например, добавьте слои)

Всего использовалось 4 различных набора слоев: скрытый слой на 32 нейрона, 2 скрытых слоя на 64 и 32 нейрона, 3 скрытых слоя (128, 64, 32), 4 скрытых слоя (256, 128, 64, 32)

```
architectures = [
    [32],
    [64, 32],
    [128, 64, 32
    [256, 128, 64, 32]
]
architecture_results =
experiment_architecture(env, architectures)
```

Результат обучения на каждом наборе представлен на рисунке 2

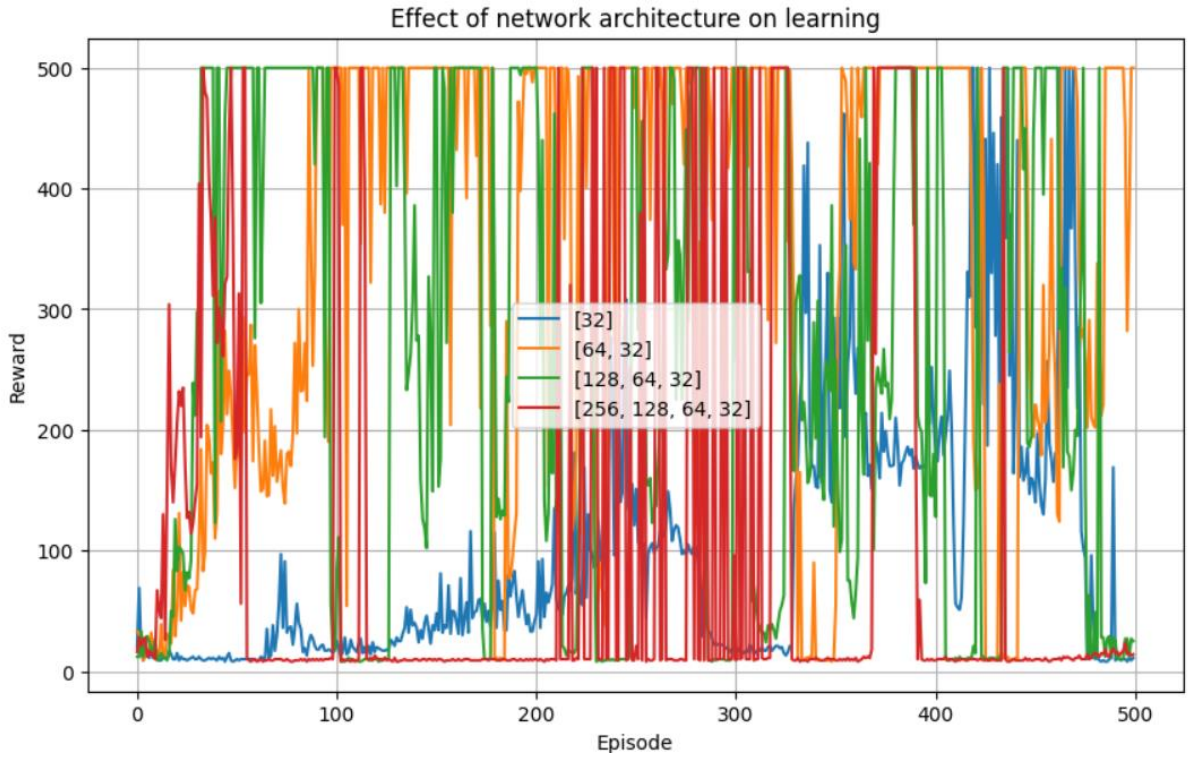

Рисунок 2 – Результат обучения на наборе слоев

3.3. Попробуйте разные значения γ и ε_{decay}

В качестве значений для γ выступал набор значений:

```
gammas = [0.8, 0.95, 0.99]
```

В качестве значений для ε_{decay} выступал набор значений:

```
epsilons = [0.99, 0.995, 0.999]
```

На рисунке 3 представлены графики с разными значениями γ и ε_{decay}.

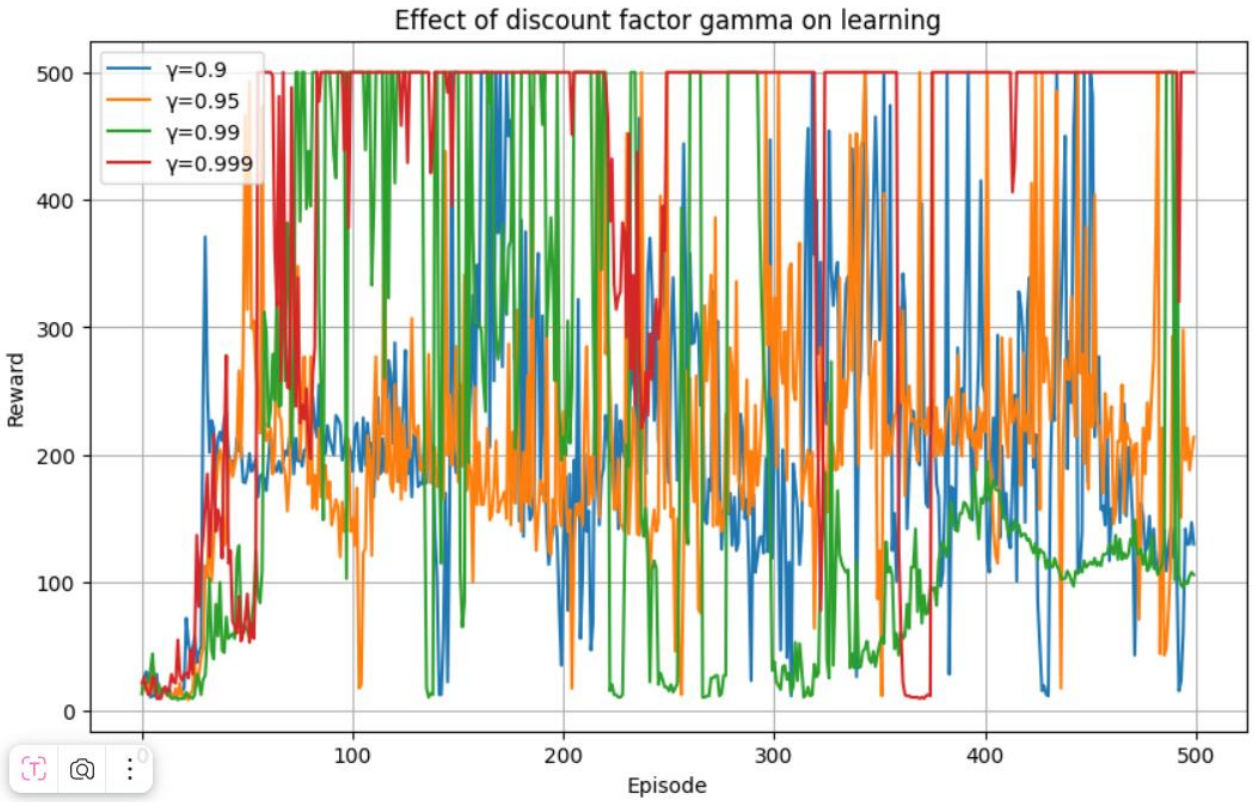

Рисунок 3 – Результаты от разных значений γ и ε_{decay}

3.4. Проведите исследование как изначальное значение ε влияет на скорость обучения

Параметры для исследования:

$$\varepsilon = \{0.8,\ 0.6,\ 0.4,\ 0.2,\ 0.1,\ 0.05,\ 0.01\ \}$$

На рисунке 4 представлены результаты эксперимента.

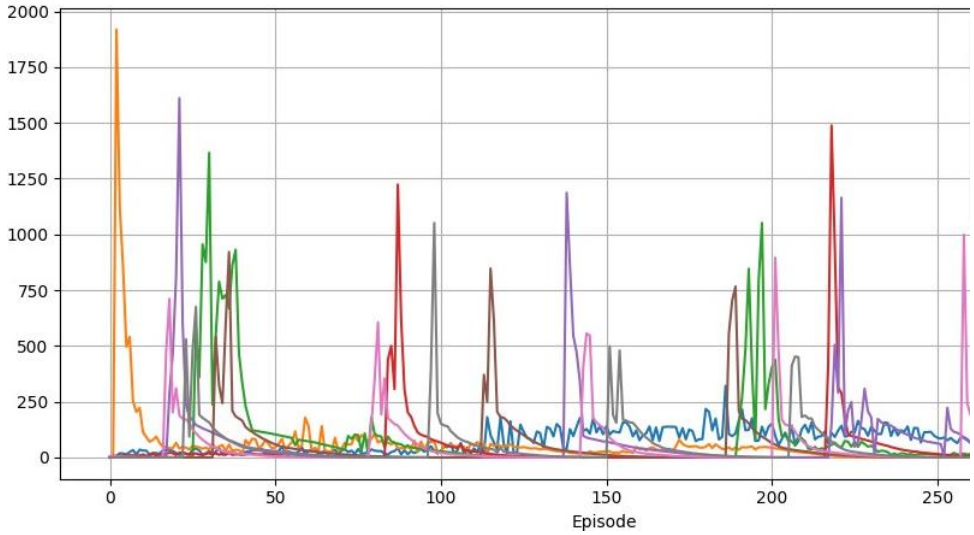

Рисунок 4 – Результаты эксперимента по изменению параметра ε

Выводы

В ходе выполнения лабораторной работы был реализован алгоритм DQN и исследовано влияние различных параметров и архитектур нейросети на процесс обучения агента в среде CartPole.

Таким образом правильный подбор гипермараметров и параметров для обучения очень важны для получения более точных моделей.

ПРИЛОЖЕНИЕ А

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import gymnasium as gym
from collections import deque
import random
import matplotlib.pyplot as plt
from tqdm import tqdm

class DQN(nn.Module):
def __init__(self, state_size, action_size, hidden_layers=[64,
64]):
super(DQN, self).__init__()
self.layers = nn.ModuleList()

self.layers.append(nn.Linear(state_size, hidden_layers[0]))

for i in range(1, len(hidden_layers)):
self.layers.append(nn.Linear(hidden_layers[i-1], hidden_layers[i]))

self.output_layer = nn.Linear(hidden_layers[-1], action_size)

def forward(self, x):
for layer in self.layers:
x = torch.relu(layer(x))
return self.output_layer(x)

class DQNAgent:
def __init__(self, state_size, action_size,
hidden_layers=[64, 64],
gamma=0.99,
epsilon_start=1.0,
epsilon_end=0.01,
epsilon_decay=0.995,
learning_rate=0.001,
batch_size=64,
memory_size=10000,
target_update_freq=10):

self.state_size = state_size
self.action_size = action_size
self.memory = deque(maxlen=memory_size)
self.gamma = gamma
```

```
self.epsilon = epsilon_start
self.epsilon_start = epsilon_start
self.epsilon_end = epsilon_end
self.epsilon_decay = epsilon_decay
self.batch_size = batch_size
self.target_update_freq = target_update_freq
self.steps = 0

self.policy_net = DQN(state_size, action_size, hidden_layers)
self.target_net = DQN(state_size, action_size, hidden_layers)
self.target_net.load_state_dict(self.policy_net.state_dict())
self.target_net.eval()

self.optimizer = optim.Adam(self.policy_net.parameters(),
lr=learning_rate)
self.loss_fn = nn.MSELoss()

def remember(self, state, action, reward, next_state, done):
self.memory.append((state, action, reward, next_state, done))

def act(self, state):
if random.random() < self.epsilon:
return random.randint(0, self.action_size - 1)
with torch.no_grad():
state = torch.FloatTensor(state).unsqueeze(0)
q_values = self.policy_net(state)
return q_values.argmax().item()

def replay(self):
if len(self.memory) < self.batch_size:
return

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

states = torch.FloatTensor(np.array(states))
actions = torch.LongTensor(actions)
rewards = torch.FloatTensor(rewards)
next_states = torch.FloatTensor(np.array(next_states))
dones = torch.FloatTensor(dones)

current_q = self.policy_net(states).gather(1, actions.unsqueeze(1))

next_q = self.target_net(next_states).max(1)[0].detach()
expected_q = rewards + (1 - dones) * self.gamma * next_q

loss = self.loss_fn(current_q.squeeze(), expected_q)
```

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

self.epsilon = max(self.epsilon_end, self.epsilon *
self.epsilon_decay)
self.steps += 1

if self.steps % self.target_update_freq == 0:
self.target_net.load_state_dict(self.policy_net.state_dict())

def save(self, filename):
torch.save(self.policy_net.state_dict(), filename)

def load(self, filename):
self.policy_net.load_state_dict(torch.load(filename))
self.target_net.load_state_dict(self.policy_net.state_dict())

def train(env, agent, episodes=1000, render_every=100):
rewards_history = []
epsilons_history = []

for episode in tqdm(range(episodes)):
state, _ = env.reset()
total_reward = 0
done = False

while not done:
action = agent.act(state)
next_state, reward, terminated, truncated, _ = env.step(action)
done = terminated or truncated
agent.remember(state, action, reward, next_state, done)
state = next_state
total_reward += reward
agent.replay()

rewards_history.append(total_reward)
epsilons_history.append(agent.epsilon)

if (episode + 1) % render_every == 0:
print(f"Episode: {episode+1}, Reward: {total_reward}, Epsilon:
{agent.epsilon:.2f}")

return rewards_history, epsilons_history

def plot_results(rewards_history, epsilons_history):
plt.figure(figsize=(12, 5))
```

```
plt.subplot(1, 2, 1)
plt.plot(rewards_history)
plt.title('Rewards per episode')
plt.xlabel('Episode')
plt.ylabel('Reward')

plt.subplot(1, 2, 2)
plt.plot(epsilons_history)
plt.title('Epsilon decay')
plt.xlabel('Episode')
plt.ylabel('Epsilon')

plt.tight_layout()
plt.show()

def test(env, agent, episodes=10, render=True):
for episode in range(episodes):
state, _ = env.reset()
total_reward = 0
done = False

while not done:
if render:
env.render()
action = agent.act(state)
next_state, reward, terminated, truncated, _ = env.step(action)
done = terminated or truncated
state = next_state
total_reward += reward

print(f"Test Episode: {episode+1}, Reward: {total_reward}")

def experiment_epsilon(env, epsilon_values=[0.1, 0.5, 0.9, 1.0],
episodes=500):
results = {}

for epsilon in epsilon_values:
print(f"\nRunning experiment with epsilon_start={epsilon}")
agent = DQNAgent(
state_size=env.observation_space.shape[0],
action_size=env.action_space.n,
epsilon_start=epsilon
)
rewards, _ = train(env, agent, episodes=episodes)
results[epsilon] = rewards

plt.figure(figsize=(10, 6))
for epsilon, rewards in results.items():
```

```
plt.plot(rewards, label=f"ε={epsilon}")
plt.title('Effect of initial epsilon on learning')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend()
plt.grid()
plt.show()

return results

def experiment_architecture(env, architectures, episodes=500):
results = {}

for arch in architectures:
print(f"\nRunning experiment with architecture {arch}")
agent = DQNAgent(
state_size=env.observation_space.shape[0],
action_size=env.action_space.n,
hidden_layers=arch
)
rewards, _ = train(env, agent, episodes=episodes)
results[str(arch)] = rewards


plt.figure(figsize=(10, 6))
for arch, rewards in results.items():
plt.plot(rewards, label=arch)
plt.title('Effect of network architecture on learning')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend()
plt.grid()
plt.show()

return results

def experiment_gamma(env, gamma_values=[0.9, 0.95, 0.99, 0.999],
episodes=500):
results = {}

for gamma in gamma_values:
print(f"\nRunning experiment with gamma={gamma}")
agent = DQNAgent(
state_size=env.observation_space.shape[0],
action_size=env.action_space.n,
gamma=gamma
)
rewards, _ = train(env, agent, episodes=episodes)
```

```
results[gamma] = rewards

plt.figure(figsize=(10, 6))
for gamma, rewards in results.items():
plt.plot(rewards, label=f"γ={gamma}")
plt.title('Effect of discount factor gamma on learning')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend()
plt.grid()
plt.show()

return results

if __name__ == "__main__":
env = gym.make("CartPole-v1")

agent = DQNAgent(
state_size=env.observation_space.shape[0],
action_size=env.action_space.n
)

rewards, epsilons = train(env, agent, episodes=500)
plot_results(rewards, epsilons)

test(env, agent)

epsilon_results = experiment_epsilon(env)

architectures = [
[32],
[64, 32],
[128, 64, 32],
[256, 128, 64, 32]
]
architecture_results = experiment_architecture(env, architectures)

gamma_results = experiment_gamma(env)

env.close()
```