Московский государственный технический университет им. Н.Э. Баумана

Факультет «Информатика и системы управления»

Кафедра «Системы обработки информации и управления»



**Лабораторная работа №4 по дисциплине**

**«Методы машинного обучения»**

**По теме «Алгоритм Policy Iteration»**

**ИСПОЛНИТЕЛЬ:**

Позняк А.А.

Группа ИУ5-22M

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

"\_\_"\_\_\_\_\_\_\_\_\_\_\_2023 г.

Москва 2023

Оглавление

[Задание 3](#_Toc137079386)

[Текст программы 4](#_Toc137079387)

[Экранные формы 11](#_Toc137079388)

Задание

1. На основе рассмотренного на лекции примера реализуйте алгоритм Policy Iteration для любой среды обучения с подкреплением (кроме рассмотренной на лекции среды Toy Text / Frozen Lake) из библиотеки [Gym](https://www.gymlibrary.dev/) (или аналогичной библиотеки).

Текст программы

!pip install matplotlib

!pip install numpy

!pip install seaborn

!pip install tqdm

!pip install gymnasium==0.27.0

!pip install gymnasium[toy\_text]

%matplotlib inline

# Author: Till Zemann

# License: MIT License

from \_\_future\_\_ import annotations

from collections import defaultdict

import matplotlib.pyplot as plt

import numpy as np

import seaborn as sns

from matplotlib.patches import Patch

from tqdm import tqdm

import gymnasium as gym

# Let's start by creating the blackjack environment.

# Note: We are going to follow the rules from Sutton & Barto.

# Other versions of the game can be found below for you to experiment.

env = gym.make("Blackjack-v1", sab=True,render\_mode="rgb\_array")

#reset the environment to get the first observation

done = False

observation, info = env.reset()

#observation = (16,9,False)

# sample a random action from all valid actions

action = env.action\_space.sample()

# action=1

# execute the action in our environment and receive infos from the environment

observation, reward, terminated, truncated, info = env.step(action)

class BlackjackAgent:

def \_\_init\_\_(

self,

learning\_rate: float,

initial\_epsilon: float,

epsilon\_decay: float,

final\_epsilon: float,

discount\_factor: float = 0.95,

):

"""Initialize a Reinforcement Learning agent with an empty dictionary

of state-action values (q\_values), a learning rate and an epsilon.

Args:

learning\_rate: The learning rate

initial\_epsilon: The initial epsilon value

epsilon\_decay: The decay for epsilon

final\_epsilon: The final epsilon value

discount\_factor: The discount factor for computing the Q-value

"""

self.q\_values = defaultdict(lambda: np.zeros(env.action\_space.n))

self.lr = learning\_rate

self.discount\_factor = discount\_factor

self.epsilon = initial\_epsilon

self.epsilon\_decay = epsilon\_decay

self.final\_epsilon = final\_epsilon

self.training\_error = []

def get\_action(self, obs: tuple[int, int, bool]) -> int:

"""

Returns the best action with probability (1 - epsilon)

otherwise a random action with probability epsilon to ensure exploration.

"""

# with probability epsilon return a random action to explore the environment

if np.random.random() < self.epsilon:

return env.action\_space.sample()

# with probability (1 - epsilon) act greedily (exploit)

else:

return int(np.argmax(self.q\_values[obs]))

def update(

self,

obs: tuple[int, int, bool],

action: int,

reward: float,

terminated: bool,

next\_obs: tuple[int, int, bool],

):

"""Updates the Q-value of an action."""

future\_q\_value = (not terminated) \* np.max(self.q\_values[next\_obs])

temporal\_difference = (

reward + self.discount\_factor \* future\_q\_value - self.q\_values[obs][action]

)

self.q\_values[obs][action] = (

self.q\_values[obs][action] + self.lr \* temporal\_difference

)

self.training\_error.append(temporal\_difference)

def decay\_epsilon(self):

self.epsilon = max(self.final\_epsilon, self.epsilon - epsilon\_decay)

# hyperparameters

learning\_rate = 0.01

n\_episodes = 1000

start\_epsilon = 1.0

epsilon\_decay = start\_epsilon / (n\_episodes / 2) # reduce the exploration over time

final\_epsilon = 0.1

agent = BlackjackAgent(

learning\_rate=learning\_rate,

initial\_epsilon=start\_epsilon,

epsilon\_decay=epsilon\_decay,

final\_epsilon=final\_epsilon,

)

from IPython.display import clear\_output

env = gym.wrappers.RecordEpisodeStatistics(env, deque\_size=n\_episodes)

for episode in tqdm(range(n\_episodes)):

obs, info = env.reset()

done = False

clear\_output()

# play one episode

while not done:

action = agent.get\_action(obs)

next\_obs, reward, terminated, truncated, info = env.step(action)

# update the agent

agent.update(obs, action, reward, terminated, next\_obs)

frame = env.render()

plt.imshow(frame)

plt.show()

# update if the environment is done and the current obs

done = terminated or truncated

obs = next\_obs

agent.decay\_epsilon()

rolling\_length = 500

fig, axs = plt.subplots(ncols=3, figsize=(12, 5))

axs[0].set\_title("Episode rewards")

reward\_moving\_average = (

np.convolve(

np.array(env.return\_queue).flatten(), np.ones(rolling\_length), mode="valid"

)

/ rolling\_length

)

axs[0].plot(range(len(reward\_moving\_average)), reward\_moving\_average)

axs[1].set\_title("Episode lengths")

length\_moving\_average = (

np.convolve(

np.array(env.length\_queue).flatten(), np.ones(rolling\_length), mode="same"

)

/ rolling\_length

)

axs[1].plot(range(len(length\_moving\_average)), length\_moving\_average)

axs[2].set\_title("Training Error")

training\_error\_moving\_average = (

np.convolve(np.array(agent.training\_error), np.ones(rolling\_length), mode="same")

/ rolling\_length

)

axs[2].plot(range(len(training\_error\_moving\_average)), training\_error\_moving\_average)

plt.tight\_layout()

plt.show()

Экранные формы

   