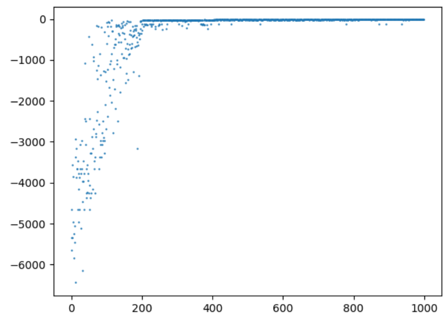
AI 3603 HW2

By: WuShuwen (521030910087)

HW#: 2

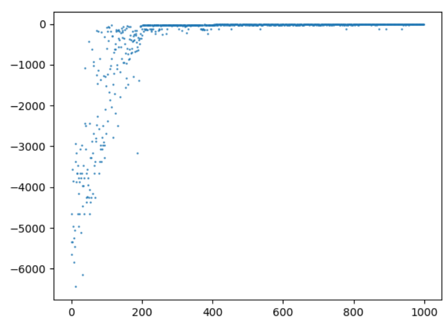
November 13, 2023

I. TASK 1: REINFORCEMENT LEARNING IN CLIFF-WALKING ENVIRONMENT



(a) Episode Reward of Sarsa

FIG. 1



(a) Episode Reward of Q-learning

FIG. 2

A. Basic algorithm

I implemented both agents with there Q-value fields represented by numpy marices. The state transfer equations of Sarsa and Q-learning are as follows respetively:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma Q(s', a'_{\epsilon-greedy}) \tag{1}$$

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma max_{a'} Q(s', a'_{\epsilon-greedy}) \tag{2}$$

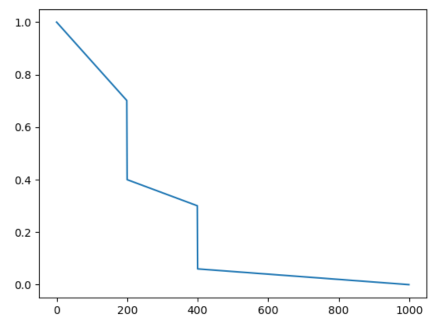
B. Epsilon Decay Scheme

Then I implemented an

$$\epsilon \tag{3}$$

-decay strategy as follow, so that it converges quite fast.

$$\epsilon = \begin{cases} 1 - 0.0015 \times episode & episode < 200 \\ 0.5 - 0.005 \times episode & 200 \leq episode < 400 \\ 0.1 - 0.0001 \times episode & otherwise \end{cases} \quad (4)$$



(a) epsilon-decay function

FIG. 3

C. Results and Analysis

As shown in the videos, sarsa and q-learning converges at different routes. Sarsa agent chooses to get as far away to the cliff as possible and walks along the edge of the map, while Q-learning agent chooses to walk the shortest path, which is really close to the cliff. There are reasons for the differences: For Q-learning, it updates Q(s, a) with the max value of Q(s , ai). Although some directions' Q value of a grid may be low, maxai (Q(s , ai)) can still be large. In our example, although the Q value for each grid on this path is small in the downward direction, the agent is still likely to choose move right due to the seductive reward for moving directly to the target point. For SARSA, this is because grid (1, 1) has a smaller Q value on the downward direction, which will gradually reduce the Q value of the grid (0, 1) in its right direction. And when the agent arrive (0, 1), it's more likely to choose to move upward.

D. Code

```
# -*- coding:utf-8 -*-
import math, os, time, sys
import numpy as np
import gym
##### START CODING HERE #####
# This code block is optional. You can import other libraries or define your utility functions if necessary.

##### END CODING HERE #####

# ------------------------------------------------------------------------------------------------------- #

class SarsaAgent(object):
    ##### START CODING HERE #####
    def __init__(self, all_actions):
        """initialize the agent. Maybe more function inputs are needed."""
        self.all_actions = all_actions
        self.epsilon = 1.0
        self.gamma = 0.95
        self.lr = 0.1
```

```
        self.q = np.zeros([12,4,4])

    def choose_action(self, observation):
        x = observation % 12
        y = int((observation - x) / 12)
        """choose action with epsilon-greedy algorithm."""
        if np.random.uniform() < self.epsilon:
            action = np.random.choice(self.all_actions)
        else:
            action = np.argmax(self.q[x][y])
            #print(action)
        return action

    def learn(self,r,observation,action,observation_n,action_n):
        """learn from experience"""
        x = observation % 12
        y = int((observation - x) / 12)
        x_n = observation_n % 12
        y_n = int((observation_n - x_n) / 12)
        self.q[x][y][action] += self.lr * (r + self.gamma * self.q[x_n][y_n][action_n] - self.q[x][y][action])
        return True

    def epsilon_decay(self, episode):
        if episode < 200:
            self.epsilon = 1 - 0.0015 * episode
        elif episode < 400:
            self.epsilon = 0.5 - 0.0005 * episode
        else:
            self.epsilon = 0.1 - 0.0001 * episode
    ##### END CODING HERE #####


class QLearningAgent(object):
    ##### START CODING HERE #####
    def __init__(self, all_actions):
        """initialize the agent. Maybe more function inputs are needed."""
        self.all_actions = all_actions
        self.epsilon = 1.0
        self.gamma = 0.95
        self.lr = 0.1
        self.q = np.zeros([12,4,4])

    def choose_action(self, observation):
        x = observation % 12
        y = int((observation - x) / 12)
        """choose action with epsilon-greedy algorithm."""
        if np.random.random() < self.epsilon:
            action = np.random.choice(self.all_actions)
        else:
            action =  np.argmax(self.q[x][y])

        return action

    def learn(self,r,observation,action,observation_n):
        """learn from experience"""
        x = observation % 12
        y = int((observation - x) / 12)
        x_n = observation_n % 12
        y_n = int((observation_n - x_n) / 12)
        self.q[x][y][action] += self.lr * (r + self.gamma * np.max(self.q[x_n][y_n]) - self.q[x][y][action])
        return False

    def epsilon_decay(self, episode):
        if episode < 200:
            self.epsilon = 1 - 0.0015 * episode
        elif episode < 400:
            self.epsilon = 0.5 - 0.0005 * episode
        else:
            self.epsilon = 0.1 - 0.0001 * episode
    ##### END CODING HERE #####
```

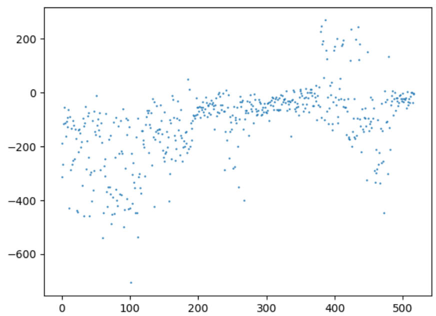
II. TASK 2: DEEP REINFORCEMENT LEARNING

A. Preparation: Install Cuda to accelerate

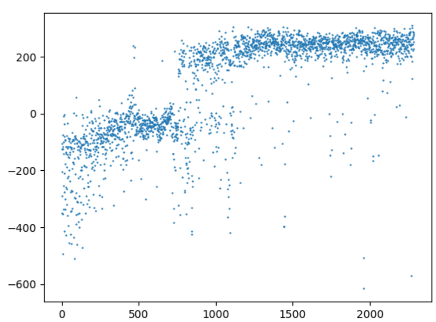
First, I installed cuda to the virtual environment, so that the training is process accelerated and I can train the model for more timesteps, which can be costly.

B. Hyper-parameters Tuning

Then I tuned the hyper-parameters as follows: learning rate = 3e-3, gamma = 0.99, start-e = 1, end-e = 0.01, to make the dqn converge. By training for 1000000 steps, I can see that it converges at about 1000 episodes, but the original number of steps,300000, is just not enough to go through so much episodes. Thanks to cuda, it seems not too costly to train so many steps, although it can certainly been improved to converge faster. A decay strategy similar to that implemented in Task1 can be helpful.



(a) Episodic Reward of 300000 timesteps



(b) Episodic Reward of 1000000 timesteps 3

FIG. 4

C. Code

```
# -*- coding:utf-8 -*-
```

```python
import argparse
import os
import random
import time

import gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from stable_baselines3.common.buffers import ReplayBuffer
from torch.utils.tensorboard import SummaryWriter
import matplotlib.pyplot as plt

def parse_args():
    """parse arguments. You can add other arguments if needed."""
    parser = argparse.ArgumentParser()
    parser.add_argument("--exp-name", type=str, default=os.path.basename(__file__).rstrip(".py"),
        help="the name of this experiment")
    parser.add_argument("--seed", type=int, default=42,
        help="seed of the experiment")
    parser.add_argument("--total-timesteps", type=int, default=1000000,
        help="total timesteps of the experiments")
    parser.add_argument("--learning-rate", type=float, default=3e-3,
        help="the learning rate of the optimizer")
    parser.add_argument("--buffer-size", type=int, default=30000,
        help="the replay memory buffer size")
    parser.add_argument("--gamma", type=float, default=0.99,
        help="the discount factor gamma")
    parser.add_argument("--target-network-frequency", type=int, default=500,
        help="the timesteps it takes to update the target network")
    parser.add_argument("--batch-size", type=int, default=128,
        help="the batch size of sample from the reply memory")
    parser.add_argument("--start-e", type=float, default=1,
        help="the starting epsilon for exploration")
    parser.add_argument("--end-e", type=float, default=0.01,
        help="the ending epsilon for exploration")
    parser.add_argument("--exploration-fraction", type=float, default=0.1,
        help="the fraction of `total-timesteps` it takes from start-e to go end-e")
    parser.add_argument("--learning-starts", type=int, default=10000,
        help="timestep to start learning")
    parser.add_argument("--train-frequency", type=int, default=10,
        help="the frequency of training")
    args = parser.parse_args()
    args.env_id = "LunarLander-v2"
    return args

def make_env(env_id, seed):
    """construct the gym environment"""
    env = gym.make(env_id)
    env = gym.wrappers.RecordEpisodeStatistics(env)
    env.seed(seed)
    env.action_space.seed(seed)
    env.observation_space.seed(seed)
    return env

class QNetwork(nn.Module):
    """comments: Neural network model for the Q-network."""
    def __init__(self, env):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(np.array(env.observation_space.shape).prod(), 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, env.action_space.n),
        )

    def forward(self, x):
        return self.network(x)

def linear_schedule(start_e: float, end_e: float, duration: int, t: int):
    """comments: Linear schedule for exploration parameter epsilon."""
    slope = (end_e - start_e) / duration
    return max(slope * t + start_e, end_e)

if __name__ == "__main__":

    """parse the arguments"""
    args = parse_args()
    run_name = f"{args.env_id}__{args.exp_name}__{args.seed}__{int(time.time())}"

    """we utilize tensorboard yo log the training process"""
    writer = SummaryWriter(f"runs/{run_name}")
    writer.add_text(
        "hyperparameters",
        "|param|value|\n|-|-|\n%s" % ("\n".join([f"|{key}|{value}|" for key, value in vars(args).items()])),
    )

    """comments: Set the random seeds for reproducibility."""
    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.backends.cudnn.deterministic = True
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    """comments: Create the environment."""
    envs = make_env(args.env_id, args.seed)

    """comments: Initialize the Q-network, optimizer, and target network."""
    q_network = QNetwork(envs).to(device)
```

```
    optimizer = optim.Adam(q_network.parameters(), lr=args.learning_rate)
    target_network = QNetwork(envs).to(device)
    target_network.load_state_dict(q_network.state_dict())

    """comments: Create the replay buffer."""
    rb = ReplayBuffer(
        args.buffer_size,
        envs.observation_space,
        envs.action_space,
        device,
        handle_timeout_termination=False,
    )

    r_l = []
    """comments: Reset the environment."""
    obs = envs.reset()
    for global_step in range(args.total_timesteps):

        """comments: Compute epsilon for exploration"""
        epsilon = linear_schedule(args.start_e, args.end_e, args.exploration_fraction * args.total_timesteps,
            global_step)

        """comments: Select an action based on epsilon-greedy policy"""
        if random.random() < epsilon:
            actions = envs.action_space.sample()
        else:
            q_values = q_network(torch.Tensor(obs).to(device))
            actions = torch.argmax(q_values, dim=0).cpu().numpy()

        """comments: Take a step in the environment"""
        next_obs, rewards, dones, infos = envs.step(actions)
        #if global_step > 290000: envs.render() # close render during training

        if dones:
            print(f"global_step={global_step}, episodic_return={infos['episode']['r']}")
            r_l.append(infos['episode']['r'])
            writer.add_scalar("charts/episodic_return", infos["episode"]["r"], global_step)
            writer.add_scalar("charts/episodic_length", infos["episode"]["l"], global_step)

        """comments: Add the transition to the replay buffer"""
        rb.add(obs, next_obs, actions, rewards, dones, infos)

        """comments: Update observation based on episode termination"""
        obs = next_obs if not dones else envs.reset()

        if global_step > args.learning_starts and global_step % args.train_frequency == 0:

            """comments: Sample a batch of transitions from the replay buffer"""
            data = rb.sample(args.batch_size)

            """comments: Compute the TD target for Q-network update"""
            with torch.no_grad():
                target_max, _ = target_network(data.next_observations).max(dim=1)
                td_target = data.rewards.flatten() + args.gamma * target_max * (1 - data.dones.flatten())
            old_val = q_network(data.observations).gather(1, data.actions).squeeze()
            loss = F.mse_loss(td_target, old_val)

            """comments: Log loss and Q-values"""
            if global_step % 100 == 0:
                writer.add_scalar("losses/td_loss", loss, global_step)
                writer.add_scalar("losses/q_values", old_val.mean().item(), global_step)

            """comments: Perform gradient descent step on the Q-network"""
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            """comments: Update the target network periodically"""
            if global_step % args.target_network_frequency == 0:
                target_network.load_state_dict(q_network.state_dict())

    """close the env and tensorboard logger"""
    x = [i for i in range(len(r_l))]
    plt.scatter(x,r_l,s = 0.5)
    plt.show()

    envs.close()
    writer.close()
```