EE619 Project #2

20233508 DoHyoung Lee

1. Explanation of networks and how they work

   A. DQN

DQN agent receives environment information and returns Q value, expected reward of action in specific state. Then, backpropagation occurred after calculating the gap of (actual reward) + (gamma * maximum Q value of action-operated state) and Q value. My DQN is consisting of (2, 4, 8, 4) fully-connected layers through ReLU function and there is a dropout layer before the output layer. The action is selected by epsilon greedy method, while epsilon starts from 0.5 and decreases *0.9 times after each 5000 actions.

   B. REINFORCE

REINFORCE algorithm is based on the monte carlo method. After one episode is finished, the gradient of log of policy probability multiplied by return is added to parameters. In my model, the gradient of log of policy probability can be calculated by putting softmax function after the output layer. Then, we can use GD method through optimizer since calculated gradient is policy gradient.

2. Difference about DQN and REINFORCE

DQN and REINFORCE are basically similar, but Q learning is type of value iteration and REINFORCE is based on policy gradient: directly optimization in the action space. DQN updates its weight by estimating the best action of the next state and REINFORCE calculated return by the agents' trajectory.

3. Description of the Robot-Gridworld including elements' role

The gridworld, which size is 20*20, has a few of terminals, bomb, food and treasure. Each of them returns -10, 50, 100 reward and episode ends. When agent make transition, -0.1 reward is supplied from environment.

4. Results

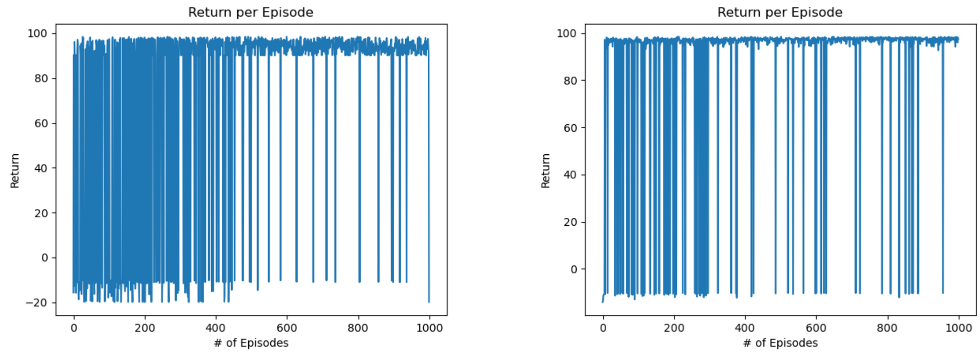

Figure 1 DQN with env1

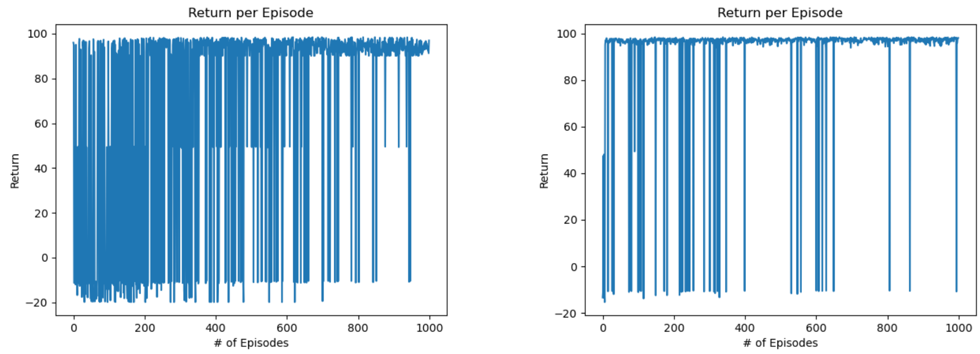

Figure 2 DQN with env2

DQN agent learns slowly, but the rate at which it finds the treasure gradually increases. Also, DQN is less influenced by the initial policy than REINFORCE. I guess this is because the number of step in each episode is not defined in DQN, so the agent can reach to the treasure through the epsilon greedy action selection.

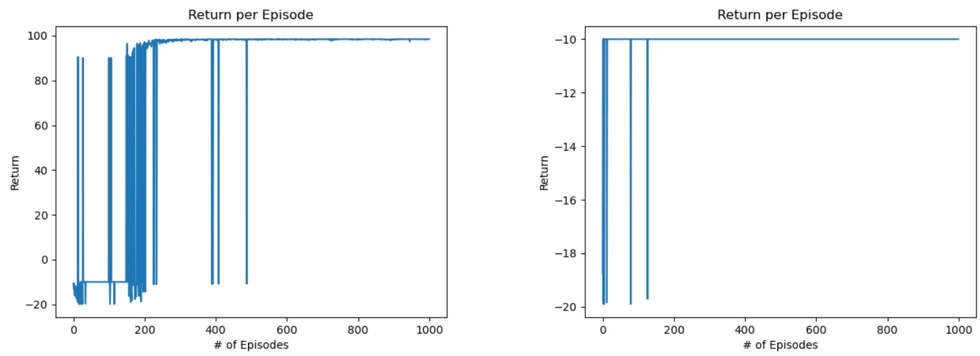

Figure 3 REINFORCE with env1

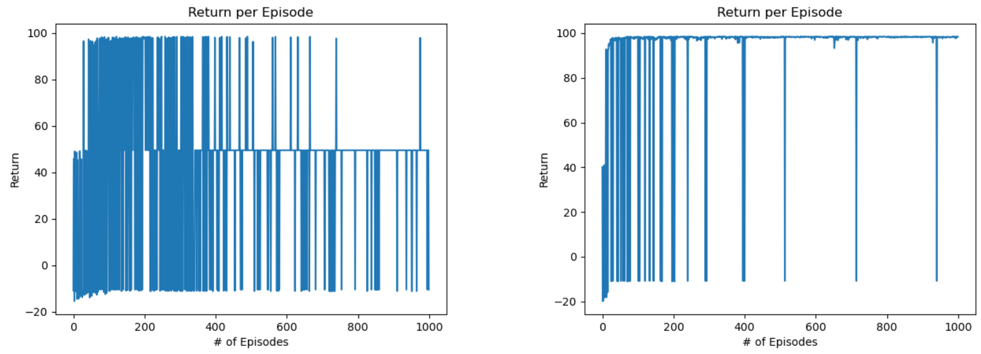

Figure 4 REINFORCE with env2

The REINFORCE algorithm in my model was very sensitive to initialization. In figure 3, if the agent was able to reach the treasure in the first few steps, this triggered a policy that allowed the agent to reach the treasure. However, if the initial policy did not allow the agent to find the treasure, it did not find the optimal policy. This trend can also be seen in Figure 4. I guess this tendency is caused by the limited steps of REINFORCE algorithm. If some action which is mandatory to visit both bomb and the treasure, however when agent is learned as the action is for bomb, the action will be blocked since the action is trained as the pathway of bomb. Then, the agent have to visit the treasure with low probability of choosing this action, but 1000 step limit is not sufficient.

5. Difference of env1 and env2

There main difference between env1 and env2. Env2 supplies one food state, which the agent is able to reach with the minimum number of transitions and returns half of the reward of treasure. Since this food supplies local minima, and it makes both DQN and REINFORCE hard to find treasure because of the maximum action probability is learned to arrive at the food.

6. Performance difference between DQN and REINFORCE

As explained at chapter 4, REINFORCE was very sensitive to initialization, hence, DQN agent was able to find the treasure even though it spends lots of steps depending on the initialization. However, once a policy is formed to reach the treasure, it converges faster to optimal policy. I guess it is because REINFORCE updates all the trajectory of all (state, action) tuple in each episode.

7. Raw code

A. main_dqn.py

```python
from env1 import Robot_Gridworld
from DQN import DeepQLearning
import matplotlib.pyplot as plt
import pdb
import numpy as np


gamma = 0.99
step = 0

def update():
    global step
    returns = []
    for episode in range(1000):

        state = env.reset()

        step_count = 0
        return_value = 0

        while True:

            env.render() # different with self.update

            action = dqn.choose_action(state)

            next_state, reward, terminal = env.step(action)

            return_value = reward + (gamma * return_value)
            step_count += 1
            dqn.store_transition(state, action, reward, next_state)

            if (step > 200) and (step % 5 == 0):
                dqn.learn()
            #### Begin learning after accumulating certain amount of memory #####
            state = next_state

            if terminal:

                print(" {} End. Total steps : {}\n".format(episode + 1, step_count))
                break

            step += 1
    ####### To Do ########
```

```
    # Plot average returns per episode
        returns.append(return_value)
    plt.figure
    plt.plot(range(1000), returns)
    plt.xlabel("# of Episodes")
    plt.ylabel("Return")
    plt.title("Return per Episode")
    plt.show()
    returns = []

    print('Game over.\n')
    env.destroy()


if __name__ == "__main__":

    env = Robot_Gridworld()

    dqn = DeepQLearning(env.n_actions, env.n_features,
                        learning_rate=0.01,
                        discount_factor=0.9,
                        e_greedy = 0.05,
                        replace_target_iter=50,
                        memory_size=3000,
                        batch_size=32)


    env.after(100, update) #Basic module in tkinter
    env.mainloop() #Basic module in tkinter
```

B. main_rl.py

```
from env1 import Robot_Gridworld
import matplotlib.pyplot as plt
import pdb
from Reinforce import Reinforce
import numpy as np
import torch

gamma = 0.99
returns = []

def update():
    global returns
    step = 0

    for episode in range(1000):

        state = env.reset()

        step_count = 0
        return_value = 0
        Reinforce.saved_rewards = []
        Reinforce.saved_log_probs = []

        while True:

            env.render()

            action, probability = Reinforce.choose_action(state)
            next_state, reward, terminal = env.step(action)

            return_value = reward + (gamma * return_value)
            step_count += 1
            Reinforce.saved_rewards.append(reward)
            #### Begin learning after accumulating certain amount of memory #####
            state = next_state
            Reinforce.saved_log_probs.append(torch.log(probability))
            if terminal:

                print(" {} End. Total steps : {}\n".format(episode + 1, step_count))
                break

            if step_count > 1000:
                break

            step += 1
        returns.append(return_value)
```

```python
        Reinforce.learn() #Reinforce.saved_rewards, Reinforce.saved_log_probs)


    ####### To Do ########
    # Plot average returns per episode
    plt.figure
    plt.plot(range(1000), returns)
    plt.xlabel("# of Episodes")
    plt.ylabel("Return")
    plt.title("Return per Episode")
    plt.show()
    returns = []

    print('Game over.\n')
    env.destroy()


if __name__ == "__main__":

    env = Robot_Gridworld()



    Reinforce = Reinforce(env.n_actions, env.n_features,
                          learning_rate=0.01,
                          discount_factor=0.9,
                          eps=0.1)


    env.after(100, update) #Basic module in tkinter
    env.mainloop() #Basic module in tkinter
```

C. DQN.py

```
import numpy as np
import torch
import pdb
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms as T
from collections import deque

np.random.seed(1)

class DeepQLearning:
    def __init__(
            self,
            n_actions,
            n_features,
            learning_rate,
            discount_factor,
            e_greedy,
            replace_target_iter,
            memory_size,
            batch_size
        ):
        ############ To DO #################
        # Initialize variables
        self.n_actions = n_actions
        self.n_features = n_features
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.e_greedy = e_greedy
        self.eps = 0.5
        self.replace_target_iter = replace_target_iter
        self.memory_size = memory_size
        self.batch_size = batch_size

        self.memory = deque()
        self.construct_network()
        self.target_network = self.model
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr =
self.learning_rate)
        self.numiter = 0


    def construct_network(self):
```

```python
        ############# To Do ##############
        self.model = nn.Sequential(
            nn.Linear(self.n_features, 4),
            nn.ReLU(),
            nn.Linear(4, 8),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(8, self.n_actions))

    def store_transition(self, s, a, r, next_s):
        ############ To Do #############
        e = (s, a, r, next_s)
        self.memory.append(e)

    def epsilon_decay(self):
        self.eps = max(self.eps*0.9, self.e_greedy)

    def choose_action(self, state):
        ############# To Do ##############
        state = torch.tensor(state, dtype=torch.float32)

        qvalues = self.model(state).detach().numpy()

        if state[0] + 0.9 ==0 :
            qvalues[2] = float("-inf")
        if state[0] + 0.0 ==0 :
            qvalues[3] = float("-inf")
        if state[1] - 0.9 ==0 :
            qvalues[1] = float("-inf")
        if state[1] + 0.0 ==0 :
            qvalues[0] = float("-inf")

        best_action = np.argmax(qvalues)
        probability_array = []
        for index in range(len(qvalues)):
            if index==int(best_action):
                probability_array.append(1 - self.eps + (self.eps / len(qvalues)))
            else:
                probability_array.append(self.eps / len(qvalues))

        self.numiter +=1
        if self.numiter % 5000==0:
            self.epsilon_decay()
        action = np.random.choice([0, 1, 2, 3], 1, p=probability_array).item()
        return action

    def learn(self):
        ############# To Do ##############
```

```
        samplenum =
np.random.choice(len(self.memory),min(len(self.memory),self.batch_size), replace=False)
        batch = [self.memory[n]for n in samplenum]
        for element in batch:
            state, action, reward, next_s = element
            state = torch.tensor(state, dtype=torch.float32)
            next_s = torch.tensor(next_s, dtype=torch.float32)
            loss = (reward + max(self.target_network(next_s)) -
max(self.model(state)))**2

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

        if self.numiter % self.replace_target_iter == 0:
            self.target_network.load_state_dict(self.model.state_dict())
```

D. Reinforce.py

```
import numpy as np
import torch
import pdb
import random
from torch.distributions import Categorical
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim


np.random.seed(1)

class DNN(nn.Module) :
    def __init__(self, n_actions, n_features) :
        super(DNN, self).__init__()
        self.n_actions = n_actions
        self.n_features = n_features
        self.model = nn.Sequential(
            nn.Linear(self.n_features, 4),
            nn.ReLU(),
            nn.Linear(4, 8),
            nn.ReLU(),

            nn.Linear(8, n_actions),
            )
        self._init_weights()

    def _init_weights(self):

        for module in self.modules() :
            if isinstance(module, nn.Linear):
                nn.init.kaiming_normal_(module.weight)
                if module.bias is not None:
                    module.bias.data.zero_()


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



class Reinforce:
    def __init__(self, n_actions, n_features, learning_rate, discount_factor, eps):

        ############ To DO #################
        # Initialize variables
```

```python
        super(Reinforce, self).__init__()
        self.n_actions = n_actions
        self.n_features = n_features
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.eps = eps
        self.construct_network()
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)


    def construct_network(self):
        ############ To DO #################
        self.model = DNN(self.n_actions, self.n_features)


    def choose_action(self, state):
        ############ To DO #################
        state = torch.tensor(state, dtype=torch.float32)
        probability_array = F.softmax(self.model(state))
        action = torch.multinomial(probability_array, 1).item()
        return action, probability_array[action]

    def learn(self):
        ############ To DO #################
        returns = []
        discounted_reward = 0
        for reward in reversed(self.saved_rewards):
            discounted_reward = reward + discounted_reward * self.discount_factor
            returns.insert(0, discounted_reward)
        returns = torch.tensor(returns)

        log_probs = torch.stack(self.saved_log_probs)
        loss = -torch.mean(returns * log_probs)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```