lab13 - report

April 22, 2021

1 Lab13 - st121413

my DQN and memory

[ ]:
```
import torch
import torch
import torch.nn as nn
import torch.nn.functional as F
import queue
import numpy as np

class Memory():
    def __init__(self, buffer_size):
        self.memory = queue.Queue()
        self.buffer_size = buffer_size
    def get_memory(self):
        return list(self.memory.queue)

    def get_memory_random(self):
        index = np.random.randint(self.memory.qsize(), size=1)
        return self.memory.queue[index[0]]

    def add_memory(self, s_t, a_t, r_t, s_t1):
        temp = (s_t, a_t, r_t, s_t1)
        if(self.memory.qsize() > self.buffer_size):
            self.memory.get()
        self.memory.put(temp)
        return True
    def reset(self):
        self.memory = queue.Queue()


class DQN(nn.Module):
    def __init__(self, number_action):
        super(DQN, self).__init__()
        # we would have just 27 inputs and 9 outputs. Two fully connected␣
↪layers of 10 units each would give us 10x28+10x11+9x11=489 parameters
        self.fc = nn.Linear(in_features=27, out_features=10)
```

```
        self.fc2 = nn.Linear(in_features=10, out_features=10)
        self.fc3 = nn.Linear(in_features=10, out_features=number_action)

    def forward(self, state):
        # state = torch.tensor(state.reshape(-1).astype(float),␣
↪requires_grad=True).float()
        # print(state)
        # # state = torch.from_numpy(state, requires_grad=True)
        # # state = state.reshape(-1)
        # state.requires_grad_(True)

        # print(state.shape, type(state.float()),state)
        out = self.fc(state)
        out = self.fc2(out)
        out = self.fc3(out)
        return out
```

The modified q_learning function

```
[ ]: def q_learning(env, gamma, n_episode, alpha, player):
         """
         Obtain the optimal policy with off-policy Q-learning method
         @param env: OpenAI Gym environment
         @param gamma: discount factor
         @param n_episode: number of episodes
         @return: the optimal Q-function, and the optimal policy
         """
         n_action = 9
         buffer = 3
         memory = Memory(buffer)
         model = DQN(number_action=n_action)
         device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
         # optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
         optimizer = torch.optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99,␣
↪eps=1e-08, weight_decay=0, momentum=0, centered=False)
         criterion = torch.nn.MSELoss()
         model.to(device)
         model.train()

         Q = defaultdict(lambda: torch.zeros(n_action))
         for episode in range(n_episode):
             ep_loss = 0
             if episode % 10000 == 9999:
                 print(f"{episode} has loss {ep_loss}")
                 # print("episode: ", episode + 1)
             state = env.reset()
             memory.reset()
```

```
        state_o = state
        state = hash(tuple(state.reshape(-1)))
        is_done = False
        with torch.set_grad_enabled(True):
            while not is_done:
                optimizer.zero_grad()
                if env.to_play() == player:
                    available_action = env.legal_actions()
                    action = epsilon_greedy_policy(state, Q, available_action)
                    next_state, reward, is_done = env.step(action)
                    next_state_o = next_state
                    next_state = hash(tuple(next_state.reshape(-1)))
                    td_delta = reward + gamma * torch.max(Q[next_state]) -␣
↪Q[state][action]
                    Q[state][action] += alpha * td_delta
                else:
                    action = env.expert_agent()
                    next_state, reward, is_done = env.step(action)
                    next_state_o = next_state
                    next_state = hash(tuple(next_state.reshape(-1)))

                    if is_done:
                        reward = -reward
                        td_delta = reward + gamma * torch.max(Q[next_state]) -␣
↪Q[state][action]
                        Q[state][action] += alpha * td_delta

                length_episode[episode] += 1
                total_reward_episode[episode] += reward
                memory.add_memory(state_o, action, reward, next_state_o)

                e = memory.get_memory_random()
                y = torch.as_tensor(e[2]).float()
                if is_done == False:
                    data = torch.as_tensor(e[3].reshape(-1).astype(float)).
↪float()
                    data.requires_grad = True
                    actions = torch.nn.functional.softmax( model(data.
↪to(device)), dim=0)
                    # print("outputs:", outputs.shape)
                    # print("output:", torch.argmax(outputs) )
                    y = e[2] + gamma * Q[hash(tuple(e[3].reshape(-1)))][torch.
↪argmax(actions)]
                # print("y:" , y)

                # a = torch.argmax(torch.nn.functional.
↪softmax(model(e[0]),dim=0))
```

```
            # print(a.view(1,-1),"=|||||=",y)
            y.requires_grad = True
            # print(y.requires_grad)
            loss = criterion(Q[hash(tuple(e[0].reshape(-1)))][e[1]], y.
↪view(1,-1))
            ep_loss += loss
            # loss.requres_grad = True
            loss.backward()
            optimizer.step()
            if(is_done):
                break
            state = next_state


    policy = {}
    for state, actions in Q.items():
        policy[state] = torch.argmax(actions).item()
    return Q, policy
```

I tried but I think I messed up at calculating the y_i. Therefore, my loss is always 0 or 1. At this moment (5AM of Friday), I decided to submit what I have.