CartPole-DQN

January 20, 2021

0.1 CartPole

[1]:
```
import gym
import numpy as np
import torch
import torch.nn as nn
import random
from tqdm import tqdm
import matplotlib.pyplot as plt
import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
```

[2]:
```
env = gym.make('CartPole-v0')
```

0.1.1 Method 1. DQN

[3]:
```
## Experience Set(for experience replay)

class Experience():
    def __init__(self, capacity = 20000):
        self.experience = []  ## store  s,a,r,s'
        self.capacity = capacity  ## max capacity
        self.volume = 0  ## current capacity
        self.iter = 0

    def insert(self, transition):
        if self.volume < self.capacity:
            ## insert directly
            self.experience.append(transition)
            self.volume += 1
        else:
            ## random choose a transition to cover
            self.experience[self.iter] = transition
            self.iter = (self.iter + 1) % self.capacity

    def sample(self, batch_size):
        ## random sample a batch including batch_size transitions
        return random.sample(self.experience, k = batch_size)
```

[4]:
```
experience = Experience()

for e in range(100):
    s0 = env.reset()
    is_end = False
    while not is_end:
        action = env.action_space.sample()
        s1, reward, is_end, _ = env.step(action)
        experience.insert([s0,action,reward,s1])
        s0 = s1
```

[5]:
```
class QNetwork(nn.Module):
    def __init__(self,obs_space,hidden_dim,output_dim,action_space):
        super(QNetwork,self).__init__()
        self.Network = nn.Sequential(
            nn.Linear(obs_space,hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim,hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2,hidden_dim // 4),
            nn.ReLU(),
            nn.Linear(hidden_dim // 4, output_dim),
            nn.ReLU(),
            nn.Linear(output_dim, action_space)
        )

    def forward(self,x):
        return self.Network(x)
```

[6]:
```
## Agent

class DQNAgent():
    def __init__(self, env, experience, hidden_dim, output_dim, gamma = 0.9,␣
 ↪epsilon = 0.1, decay_rate = 1, learning_rate = 1e-4):
        self.env = env
        self.action_space = env.action_space
        self.obs_space = env.observation_space.shape[0]
        self.action_len = len([i for i in range(self.action_space.n)])
        self.experience = experience

        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        self.behaviour_QNetwork = QNetwork(self.obs_space, hidden_dim,␣
 ↪output_dim, self.action_len).to(self.device)
        self.target_QNetwork = QNetwork(self.obs_space, hidden_dim, output_dim,␣
 ↪self.action_len).to(self.device)
        self.loss_fn = nn.MSELoss()
```

```
        self.optimizer = torch.optim.Adam(self.behaviour_QNetwork.parameters(),␣
↪lr = learning_rate)

        self.epsilon = epsilon
        self.decay_rate = decay_rate
        self.gamma = gamma

    def policy(self, state, epsilon = 0.1):
        if np.random.random() < epsilon:
            action = self.action_space.sample()
        else:
            score = self.behaviour_QNetwork(torch.Tensor(state).to(self.
↪device)).detach()
            action = torch.argmax(score).item()

        return action

    def learn(self, batch_size, display = False):
        s0 = self.env.reset()
        if display:
            self.env.render()
        is_end = False
        episode_reward = 0

        while not is_end:
            ## choose an action and make a step
            a0 = self.policy(s0, epsilon = self.epsilon)
            s1, reward, is_end, _ = self.env.step(a0)
            if display:
                self.env.render()
            if is_end:
                s1 = np.array([100,100,100,100])
            ## store the transition into experience
            self.experience.insert([s0,a0,reward,s1])
            ## sample minibatch from experience
            minibatch = self.experience.sample(batch_size = batch_size)
            s, a, r, s_next = [], [], [], []
            for batch in minibatch:
                s.append(batch[0])
                a.append(batch[1])
                r.append(batch[2])
                s_next.append(batch[3])
            s = torch.Tensor(s).to(self.device)
            a = torch.LongTensor(a).to(self.device).reshape(-1,1)
            r = torch.Tensor(r).to(self.device).reshape(-1,1)
            s_next = torch.Tensor(s_next).to(self.device)
```

```
            Q_target = r + self.gamma * torch.max(self.
↪target_QNetwork(s_next),1)[0].reshape(-1,1) * (s_next[:,0] != 100).
↪reshape(-1,1)
            Q_behaviour = self.behaviour_QNetwork(s).gather(1,a)
            loss = self.loss_fn(Q_target, Q_behaviour)
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
            ## iteration
            s0 = s1
            episode_reward += reward

        ## update target network
        self.target_QNetwork.load_state_dict(self.behaviour_QNetwork.
↪state_dict())
        self.epsilon *= self.decay_rate

        return episode_reward, loss.item()
```

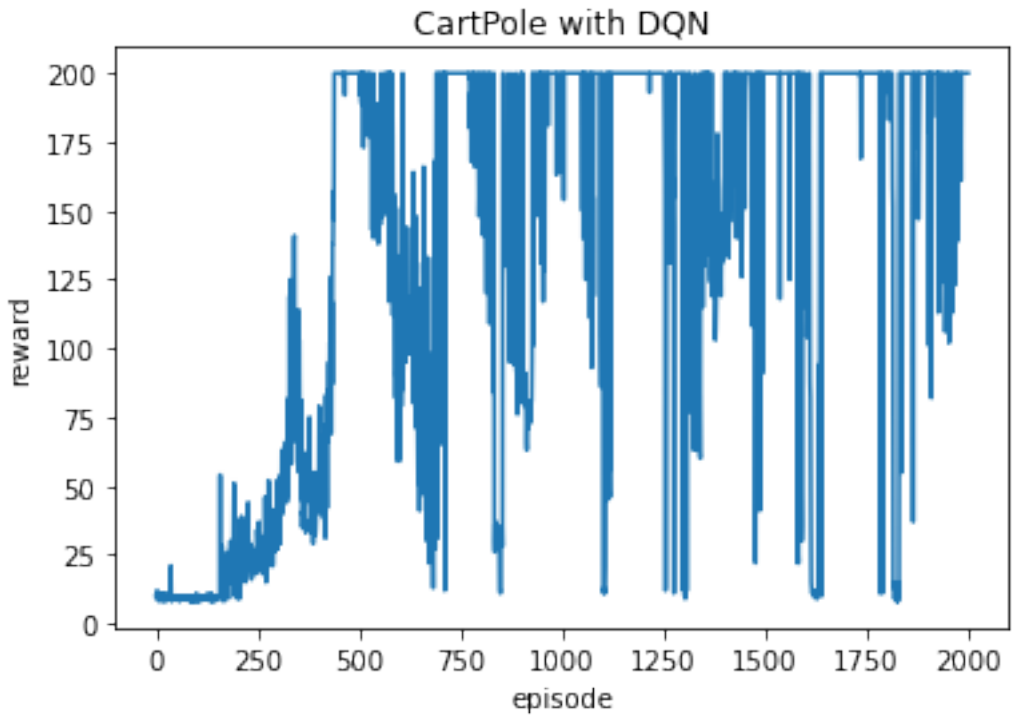
[7]:
```
## train
dqn_agent = DQNAgent(env, experience, hidden_dim = 16,output_dim = 3, gamma = 0.
↪99, epsilon = 0.1, decay_rate = 0.9, learning_rate = 1e-3)
MAX_EPISODE = 2000
dqn_episode_reward = []
dqn_loss = []
average_100_step = []

for e in tqdm(range(MAX_EPISODE)):
    reward, loss = dqn_agent.learn(batch_size = 100, display = False)
    dqn_episode_reward.append(reward)
    average_100_step.append(np.mean(dqn_episode_reward[-100:]))
    dqn_loss.append(loss)
```

```
100%|
  | 2000/2000 [22:51<00:00,  1.46it/s]
```

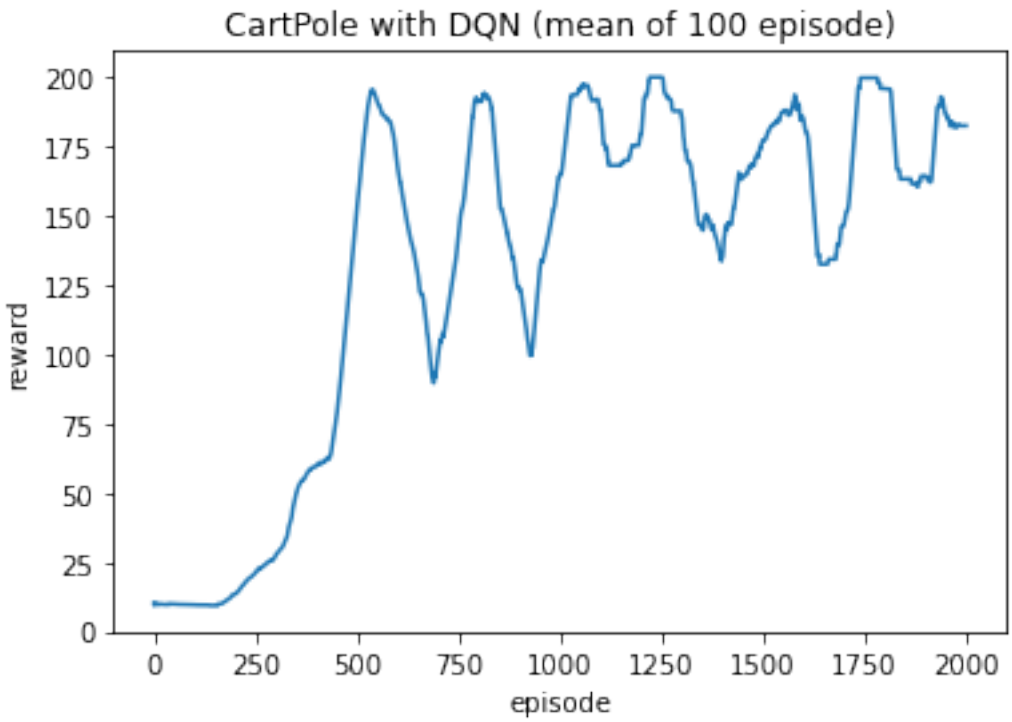
[8]:
```
# outcome of each episode
plt.plot(dqn_episode_reward)
plt.title("CartPole with DQN")
plt.xlabel("episode")
plt.ylabel("reward")
```

[8]:
```
Text(0, 0.5, 'reward')
```



[9]:
```
# outcome of episode(mean of last 100 episode)
plt.plot(average_100_step)
plt.title("CartPole with DQN (mean of 100 episode)")
plt.xlabel("episode")
plt.ylabel("reward")
```

[9]:
```
Text(0, 0.5, 'reward')
```


CartPole with DQN (mean of 100 episode)
reward
episode
0
25
50
75
100
125
150
175
200
0
250
500
750
1000
1250
1500
1750
2000