7 -DQN

August 13, 2025

[1]:
```python
import random
import gymnasium as gym
import numpy as np
import collections
from tqdm import tqdm
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
import rl_utils
```

[2]:
```python
class ReplayBuffer:
    '''     '''
    def __init__(self, capacity):
        self.buffer = collections.deque(maxlen=capacity)  #  ,

    def add(self, state, action, reward, next_state, done):  #    buffer
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):  # buffer   , batch_size
        transitions = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*transitions)
        return np.array(state), action, reward, np.array(next_state), done

    def size(self):  #  buffer
        return len(self.buffer)
```

[3]:
```python
class Qnet(torch.nn.Module):
    '''     Q  '''
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(Qnet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))  #    ReLU
        return self.fc2(x)
```

```
[4]: class DQN:
         ''' DQN   '''
         def __init__(self, state_dim, hidden_dim, action_dim, learning_rate, gamma,
                      epsilon, target_update, device):
             self.action_dim = action_dim
             self.q_net = Qnet(state_dim, hidden_dim,
                               self.action_dim).to(device)  # Q
             #
             self.target_q_net = Qnet(state_dim, hidden_dim,
                                      self.action_dim).to(device)
             #  Adam
             self.optimizer = torch.optim.Adam(self.q_net.parameters(),
                                               lr=learning_rate)
             self.gamma = gamma  #
             self.epsilon = epsilon  # epsilon-
             self.target_update = target_update  #
             self.count = 0  #   ,
             self.device = device

         def take_action(self, state):  # epsilon-
             if np.random.random() < self.epsilon:
                 action = np.random.randint(self.action_dim)
             else:
                 state = torch.tensor([state], dtype=torch.float).to(self.device)
                 action = self.q_net(state).argmax().item()
             return action

         def update(self, transition_dict):
             states = torch.tensor(transition_dict['states'],
                                   dtype=torch.float).to(self.device)
             actions = torch.tensor(transition_dict['actions']).view(-1, 1).to(
                 self.device)
             rewards = torch.tensor(transition_dict['rewards'],
                                    dtype=torch.float).view(-1, 1).to(self.device)
             next_states = torch.tensor(transition_dict['next_states'],
                                        dtype=torch.float).to(self.device)
             dones = torch.tensor(transition_dict['dones'],
                                  dtype=torch.float).view(-1, 1).to(self.device)

             q_values = self.q_net(states).gather(1, actions)  # Q
             #     Q
             max_next_q_values = self.target_q_net(next_states).max(1)[0].view(
                 -1, 1)
             q_targets = rewards + self.gamma * max_next_q_values * (1 - dones
                                                                     )  # TD
             dqn_loss = torch.mean(F.mse_loss(q_values, q_targets))  #
             self.optimizer.zero_grad()  # PyTorch     ,      0
```

```
        dqn_loss.backward()  #
        self.optimizer.step()

        if self.count % self.target_update == 0:
            self.target_q_net.load_state_dict(
                self.q_net.state_dict())  #
        self.count += 1
```

[6]:
```
#
lr = 2e-3
num_episodes = 500
hidden_dim = 128
gamma = 0.98
epsilon = 0.01
target_update = 10
buffer_size = 10000
minimal_size = 500
batch_size = 64
device = torch.device("cuda") if torch.cuda.is_available() else torch.
 ↪device("cpu")

#   1     CartPole-v1   seed
env_name = 'CartPole-v1'
#   2      seed         reset
env = gym.make(env_name)

#
random.seed(0)
np.random.seed(0)
torch.manual_seed(0)

#   3   reset
state, _ = env.reset(seed=0)  # Gymnasium   reset    (state, info)

#
replay_buffer = ReplayBuffer(buffer_size)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = DQN(state_dim, hidden_dim, action_dim, lr, gamma, epsilon,
            target_update, device)

#
return_list = []
for i in range(10):
    with tqdm(total=int(num_episodes / 10), desc='Iteration %d' % i) as pbar:
        for i_episode in range(int(num_episodes / 10)):
            episode_return = 0
```

```
            #        state, _ = env.reset()
            state, _ = env.reset()
            done = False
            while not done:
                action = agent.take_action(state)
                #  Gymnasium  step  (next_state, reward, terminated, truncated, info)
                next_state, reward, terminated, truncated, _ = env.step(action)
                #  terminated  truncated  done
                done = terminated or truncated
                replay_buffer.add(state, action, reward, next_state, done)
                state = next_state
                episode_return += reward

                if replay_buffer.size() > minimal_size:
                    b_s, b_a, b_r, b_ns, b_d = replay_buffer.sample(batch_size)
                    transition_dict = {
                        'states': b_s,
                        'actions': b_a,
                        'next_states': b_ns,
                        'rewards': b_r,
                        'dones': b_d
                    }
                    agent.update(transition_dict)
            return_list.append(episode_return)
            if (i_episode + 1) % 10 == 0:
                pbar.set_postfix({
                    'episode': '%d' % (num_episodes / 10 * i + i_episode + 1),
                    'return': '%.3f' % np.mean(return_list[-10:])
                })
            pbar.update(1)

# Iteration 0: 100%|     | 50/50 [00:00<00:00, 764.86it/s, episode=50,
# return=9.300]
# Iteration 1: 100%|     | 50/50 [00:04<00:00, 10.66it/s, episode=100,
# return=12.300]
# Iteration 2: 100%|     | 50/50 [00:24<00:00,  2.05it/s, episode=150,
# return=123.000]
# Iteration 3: 100%|     | 50/50 [01:25<00:00,  1.71s/it, episode=200,
# return=153.600]
# Iteration 4: 100%|     | 50/50 [01:30<00:00,  1.80s/it, episode=250,
# return=180.500]
# Iteration 5: 100%|     | 50/50 [01:24<00:00,  1.68s/it, episode=300,
# return=185.000]
# Iteration 6: 100%|     | 50/50 [01:32<00:00,  1.85s/it, episode=350,
# return=193.900]
# Iteration 7: 100%|     | 50/50 [01:31<00:00,  1.84s/it, episode=400,
```

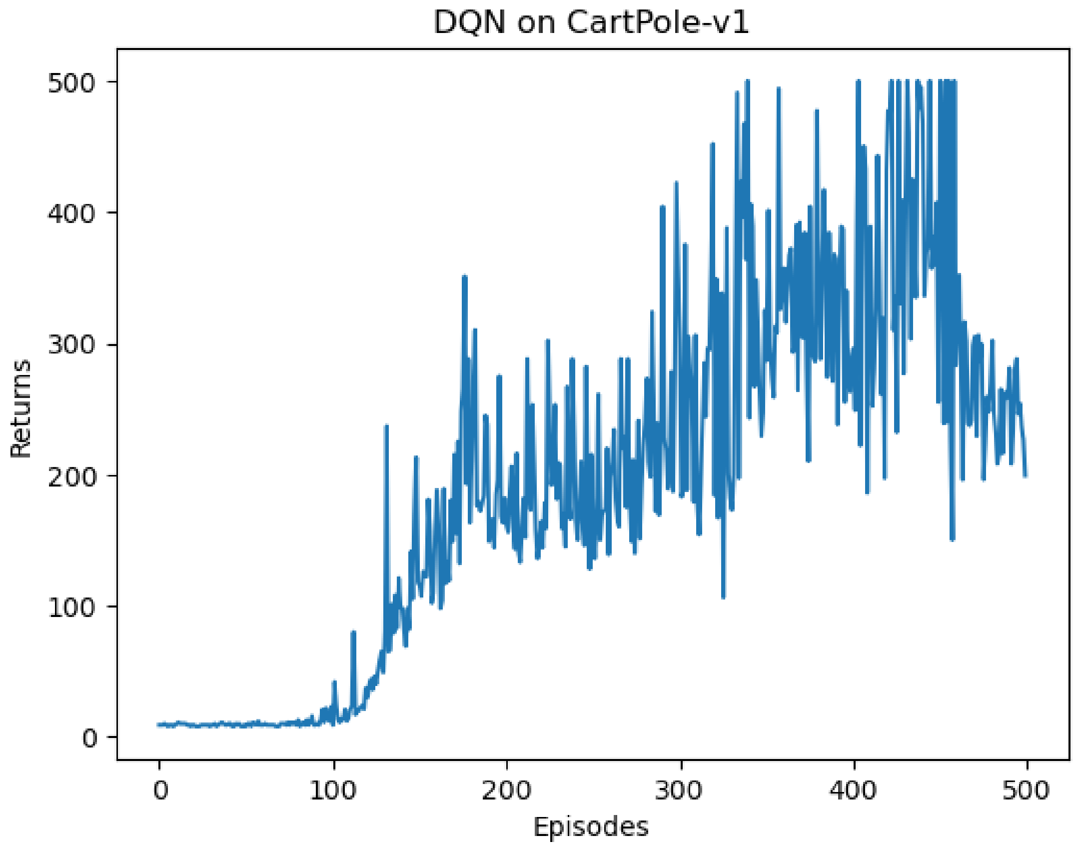
```
# return=196.600]
# Iteration 8: 100%|      | 50/50 [01:33<00:00,  1.88s/it, episode=450,
# return=193.800]
# Iteration 9: 100%|      | 50/50 [01:34<00:00,  1.88s/it, episode=500,
# return=200.000]
```

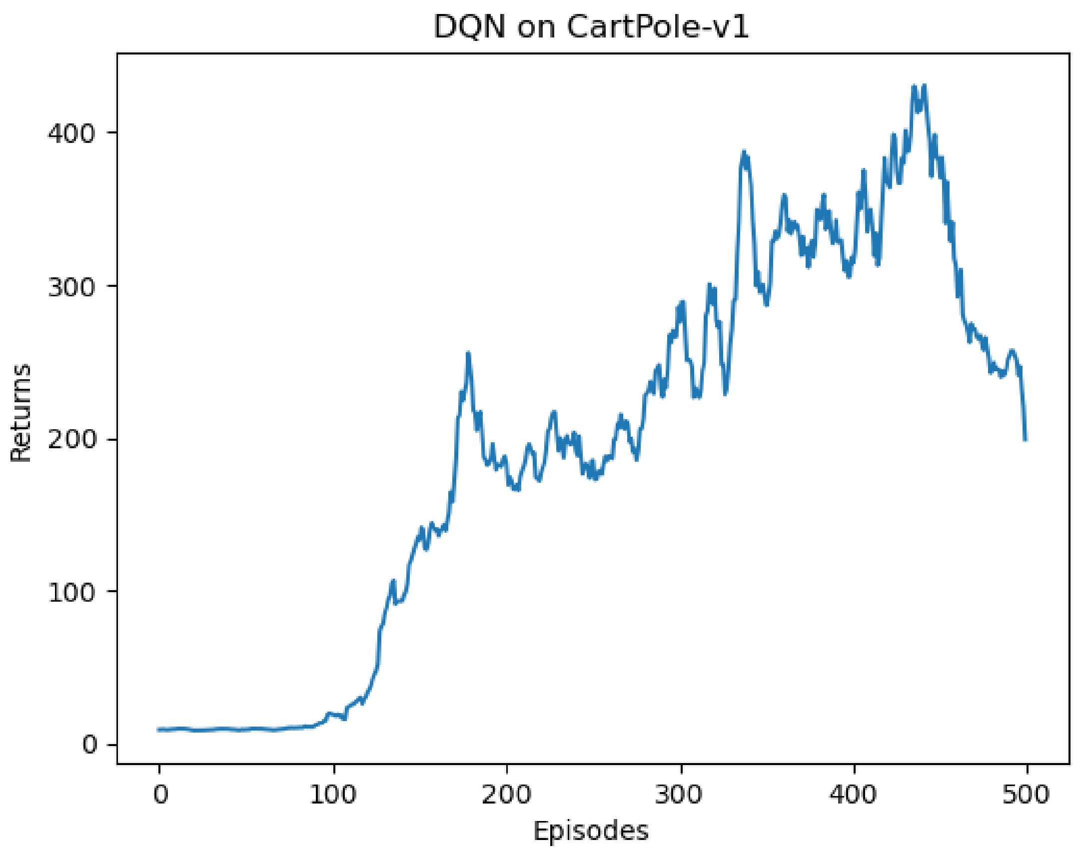
```
Iteration 0:   0%|          | 0/50 [00:00<?,
?it/s]C:\Users\Administrator\AppData\Local\Temp\ipykernel_16088\725080296.py:24:
UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow.
Please consider converting the list to a single numpy.ndarray with numpy.array()
before converting to a tensor. (Triggered internally at
C:\cb\pytorch_1000000000000\work\torch\csrc\utils\tensor_new.cpp:281.)
  state = torch.tensor([state], dtype=torch.float).to(self.device)
Iteration 0: 100%|      | 50/50 [00:00<00:00, 184.56it/s, episode=50,
return=9.000]
Iteration 1: 100%|      | 50/50 [00:00<00:00, 52.78it/s, episode=100,
return=14.500]
Iteration 2: 100%|      | 50/50 [00:04<00:00, 10.76it/s, episode=150,
return=117.500]
Iteration 3: 100%|      | 50/50 [00:14<00:00,  3.51it/s, episode=200,
return=179.300]
Iteration 4: 100%|      | 50/50 [00:14<00:00,  3.37it/s, episode=250,
return=180.400]
Iteration 5: 100%|      | 50/50 [00:17<00:00,  2.92it/s, episode=300,
return=276.500]
Iteration 6: 100%|      | 50/50 [00:22<00:00,  2.18it/s, episode=350,
return=302.100]
Iteration 7: 100%|      | 50/50 [00:25<00:00,  1.93it/s, episode=400,
return=314.800]
Iteration 8: 100%|      | 50/50 [00:29<00:00,  1.70it/s, episode=450,
return=380.700]
Iteration 9: 100%|      | 50/50 [00:21<00:00,  2.29it/s, episode=500,
return=245.100]
```

[7]:

```
episodes_list = list(range(len(return_list)))
plt.plot(episodes_list, return_list)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('DQN on {}'.format(env_name))
plt.show()

mv_return = rl_utils.moving_average(return_list, 9)
plt.plot(episodes_list, mv_return)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('DQN on {}'.format(env_name))
plt.show()
```


DQN on CartPole-v1
Returns
Episodes
0
100
200
300
400
500
0
100
200
300
400
500



[9]:
```
class ConvolutionalQnet(torch.nn.Module):
    '''    Q   '''
    def __init__(self, action_dim, in_channels=4):
        super(ConvolutionalQnet, self).__init__()
        self.conv1 = torch.nn.Conv2d(in_channels, 32, kernel_size=8, stride=4)
        self.conv2 = torch.nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = torch.nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc4 = torch.nn.Linear(7 * 7 * 64, 512)
        self.head = torch.nn.Linear(512, action_dim)

    def forward(self, x):
        x = x / 255
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.fc4(x))
        return self.head(x)
```