Deep Learning and Practice: DQN and DDPG

0756079 陳冠聞

June 5, 2019

1 Introduction

Reinforcement Learning is about the learning from interaction with environment. Value-based method such as Q-learning is to learn the expected return for all possible states, and select the action that maximize the expected return. However, if we use tabular form to represent the value function, it is hard to store and learn a huge table for large MDPs, and the tabular approach can not generalize to unseen states, which makes the learning process inefficient. Thus, using **value function approximation** is a more appropriate choice for large MDPs. As the recent development of deep learning model, it is common to use neural network to approximate the value function.

In this lab, I will implement two famous deep reinforcement learning models. One is **Deep Q Network** (DQN), and it will be trained with the discrete-action game *CartPole-v0*. Another is **Deep Deterministic Policy Gradient** (DDPG), and it will be trained on the continuous-action game *Pendulum-v0*.

2 DQN

2.1 Neural Network as a Function Approximator

Deep Q Network (DQN) is a value-based method to estimated the action value (Q-value) for a state action pair (s, a) given a policy π. As mentioned above, the traditional tabular form to record the Q value is inefficient for

large MDPs. Thus, neural network is used as a function approximator in DQN

$$\hat{Q}(s, a|\theta) \approx Q(s, a)$$

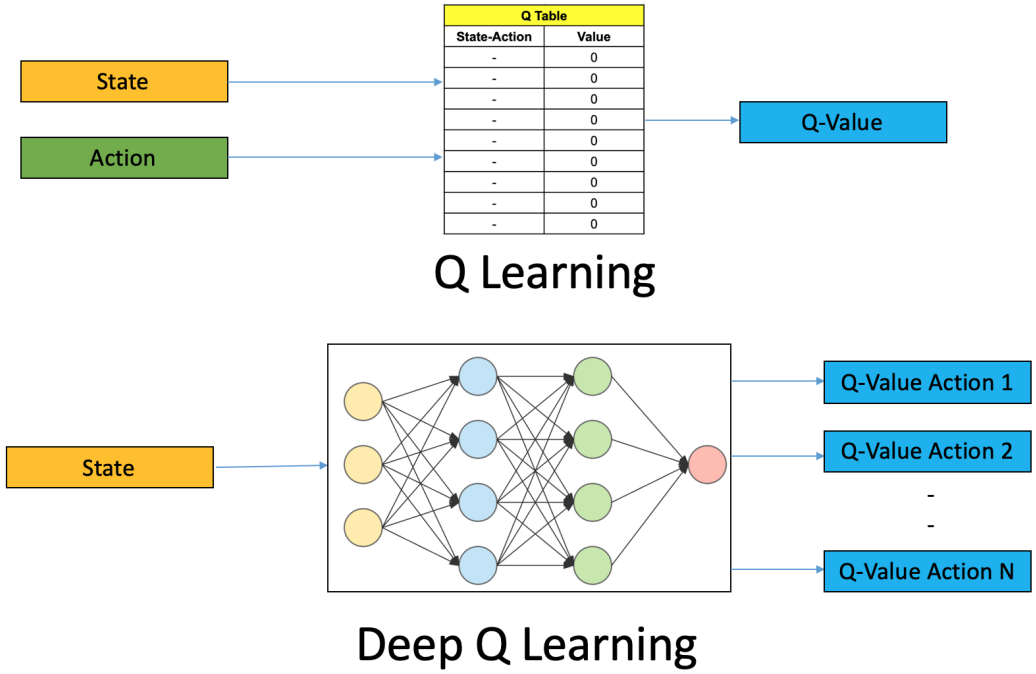

Figure 1: Q-learning v.s. DQN

To train the neural network, we can use the Q-learning target to compute the gradient, and update the model parameter using the SGD

$$\begin{aligned} target &= R(s, a, s^{'}) + \gamma \max_{a} Q(s^{'}, a^{'}|w) \\ L(w) &= \mathbb{E}[(target - Q(s, a|w))^2] \\ \frac{\partial L(w)}{\partial w} &= \mathbb{E}[(target - Q(s, a|w)) \frac{\partial Q(s, a|w)}{\partial w}] \end{aligned}$$

Unfortunately, this approach will face two problems. And DQN proposes two approach to solve these two problems.

2.2 Experience Replay

One problem is that if we train the network with online learning, the training data is not independent to each other in an episode because the state and action in an episode is huge effected by the current policy. This will violate the **i.i.d assumption** of machine learning theory, and might cause the model to overfitting the data.

To decorrelate the training data, DQN uses a **replay buffer** to store transitions, and randomly sample a mini-batch for training. Since the sampled data are not in the same episode or under the same policy, the data is more independent and closer to the i.i.d assumption.

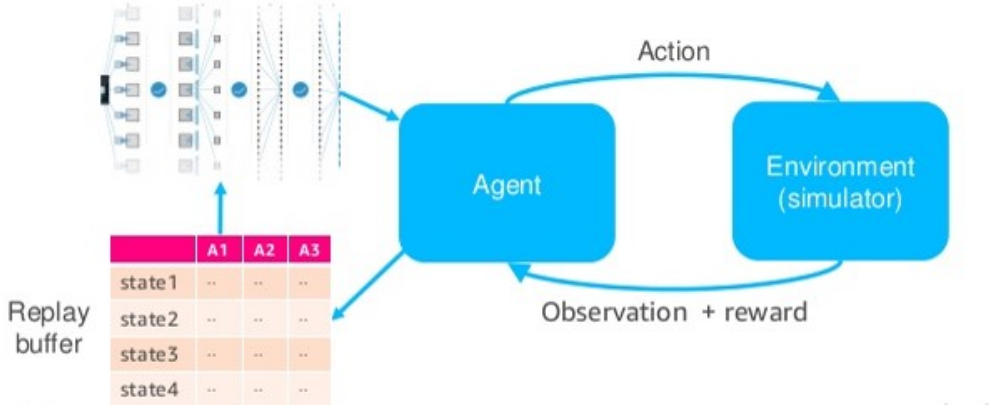

Figure 2: Replay Buffer in DQN

2.3 Fixed Q-Target

Second problem is that both target and the action are determined by the current policy. After updating the model parameters, both target and action are changed. That is, the model is **chasing a moving target**, and it might lead to instability in training.

To stabilize the training, DQN use two network, which are Q network θ and **target Q network** θ^-. The loss function become

$$L(\theta) = \mathbb{E}[(R(s, a, s^{'}) + \gamma \max_{a} Q(s^{'}, a^{'}|\theta^-) - Q(s, a|\theta))^2]$$

In the training, the target Q network is fixed and only update the Q network, which avoid the problem of chasing moving target. Then synchronize the θ^- with θ after certain number of updates (e.g. 1,000 updates).

3 DDPG

Deep Deterministic Policy Gradient (DDPG) use the **Actor Critic** approach and neural network as value function approximation. The difference between DQN and DDPG is that the former one can only output actions in discrete space, and the latter one can output actions in continuous space.

In actor critic approach, both value function and policy distribution is computed explicitly. In DDPG, both action value function $Q(s, a)$ and deterministic policy distribution $\mu(a|s)$ are approximated using neural network.

$$\hat{Q}(s, a|\theta^Q) \approx Q(s, a)$$
$$\hat{\mu}(s, a|\theta^\mu) \approx \mu(a|s)$$

where the $\hat{Q}(s, a|\theta^Q)$ is called critic network, and $\hat{\mu}(s, a|\theta^\mu)$ is called actor network.

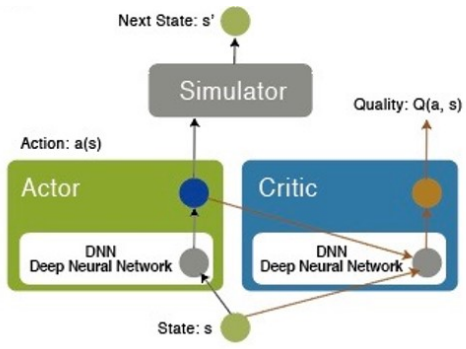

Figure 3: Actor and Critic Network in DDPG

Similar to DQN, DDPG also uses the **experience replay** to decorrelate the training data and **target network** $Q^{'}$ and $\mu^{'}$ to stabilize the training.

3.1 Update of Critic Network

The critic network is used to estimated the action value of a state. The target and loss is similar to DQN, except for the action is selected by actor network

$$target_t = R(s_t, a_t, s_{t+1}) + \gamma Q(s_{t+1}, \mu(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$$
$$L(\theta^Q) = \frac{1}{N}\sum_{t=1}^{N}[(target_t - Q(s_t, a_t|\theta^Q))^2]$$

The model parameters of critic network can be updated using SGD. For the target critic network, DDPG uses soft update (e.g. $\tau = 0.001$)

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

3.2 Update of Actor Network

The actor network is used to select a deterministic action for a given state. The training objective is to maximize the expected Q value

$$J(\theta) = \mathbb{E}[Q(s, a)|s = s_t, a = u(s_t|\theta^\mu)]$$

The gradient of objective function with respect to actor network is

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s, a)\nabla_{\theta^\mu}\mu(s_t|\theta^\mu)$$

Since the learning is off-policy with batch of transition experience, we compute the sample mean gradient and do gradient ascent to update the actor network parameters

$$\nabla_{\theta^u} J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}[\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)}\nabla_{\theta^\mu}\mu(s_t|\theta^u)|_{s_i}]$$

For the target actor network, DDPG also uses soft update (e.g. $\tau = 0.001$)

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

4 Implementation

4.1 Network structure and loss function

4.1.1 DQN

For DQN, I use two fully connected layers for Q network and Q^- network. The dimension of fc layer are $(d_s, 32)$ and $(32, d_a)$ respectively, where d_s is the dimension of state and d_a is the dimension of action ($d_s = 4$ and $d_a = 2$ in CartPole-v0). The loss function for DQN is the Mean Square Loss between the target Q value and the predicted Q value (mentioned in DQN section).

```
class DQN(nn.Module):
    def __init__(self, env, replayMemory):
        super(DQN, self).__init__()
        self.action_dim = env.action_space.n
        self.state_dim =  env.observation_space.shape[0]
        self.model = nn.Sequential(nn.Linear(self.state_dim, 32),
                                   nn.ReLU(),
                                   nn.Linear(32, self.action_dim))
```

Figure 4: Code Snippet for DQN model architecture

```
expected_state_action_values = (next_state_values * discount) + reward_batch
criterion = nn.MSELoss()
loss = criterion(torch.squeeze(state_action_values), expected_state_action_values)
```

Figure 5: Loss Computation of DQN

4.1.2 DDPG

For DDPG, I use the DDPG framework implemented in `https://github.com/ShangtongZhang/DeepRL`, and modify the network architecture and training parameters. The loss function of critic network is Mean Square Loss between the target Q value and the predicted Q value. For the actor network, the loss is the gradient of $Q(s, \mu(s))$ multiplied by the gradient of $\mu(s)$.

```
Critic(
  (main): Sequential(
    (0): Linear(in_features=3, out_features=512, bias=True)
    (1): Linear(in_features=513, out_features=256, bias=True)
    (2): Linear(in_features=256, out_features=1, bias=True)
  )
)
Actor(
  (main): Sequential(
    (0): Linear(in_features=3, out_features=512, bias=True)
    (1): Linear(in_features=512, out_features=256, bias=True)
    (2): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

Figure 6: Network Architecture of DDPG

```
# Loss of critic network
q = self.network.critic(states, actions)
q_next = self.target_network.critic(states_next, actions_next)
critic_loss = (q - (reward + discount*q_next)).pow(2).mul(0.5).sum(-1).mean()

# Loss of actor network
action = self.network.actor(states)
policy_loss = -self.network.critic(states.detach(), action).mean()
```

Figure 7: Loss Computation of DDPG

4.2 Training process of deep Q-learning

For the training of DQN, the agent take action using epsilon-greedy approach, and save the transition $(s, a, s^{'}, r)$ to the replay buffer. In addition, sample a batch of data in replay buffer, and compute the MSE error loss between the target Q value and the predicted Q value then do back-propagation to update the parameters of Q network. The target Q network is updated per 50 step by copying the weight of Q network.

```
def train(dqn, env, optimizer, STEP, num_episodes):
    for episode in range(1, num_episodes+1):
        state = env.reset()
        state = torch.from_numpy(state.reshape((-1,4))).float()
        for t in range(1, STEP+1):
            action = dqn.egreedy_action(state)
            next_state,reward,done,_ = env.step(int(action[0,0].data.item()))
            next_state = torch.from_numpy(next_state.reshape((-1,4))).float()
            total_reward += reward
            reward = Tensor([reward])
            final = LongTensor([done])
            dqn.push(state,action,next_state,reward,final)
            state = next_state
            loss = dqn.loss()

            # Backward
            if loss is not None:
                optimizer.zero_grad()
                loss.backward()
                for param in dqn.model.parameters():
                    param.grad.data.clamp_(-1, 1)
                optimizer.step()
                total_loss += loss.data.item()
            if steps_done % TARGETQ_UPDATE == 0:
                dqn.updateTargetModel()
            if done:
                break
```

Figure 8: Code Snippet for Training DQN

4.3 Epsilon-greedy action select method

To explore more states, I adopt epsilon-greedy action select approach. At the beginning of training, the probability to randomly explore is set to 1.0, and multiply it by 0.995 for each step the agent takes until the probability is lower than 0.01.

```
def egreedy_action(self,state):
    global steps_done
    if self.epsilon >= EPS_END:
        self.epsilon *= EPS_DECAY
    steps_done += 1
    if random.random() > self.epsilon:
        return self.action(state)
    else:
        return LongTensor([[random.randrange(self.action_dim)]])
```

Figure 9: Code Snippet for Epsilon-greedy action selection

4.4 How to calculate the gradients

For the gradient computation in DQN, please refer Section 2 and Section 4.11. For the gradient computation in DDPG, please refer Section 3 and Section 4.12.

4.5 How the code works

Training procedure of DQN and DDPG can be described as follow. First, initialize the agent and the game environment. Then the agent observe state information from game, and take action accordingly in each step. The transition composed of (state, action, next state, reward) is recorded and saved in replay buffer. If current buffer have enough transition records, the agent will sample a mini batch and use them to train the network. The target network will be updated after certain number of steps (DQN) or using soft update (DDPG). A game episode is finished if the game is over (e.g. pole is too far) or the maximal steps is reach (e.g. max step = 200).
The detail training processing for DQN can refer to Section 4.2, and the code snippet for training DDPG can refer to Figure 10 (modified from framework).

```
def train(agent):
    config = agent.config
    agent_name = agent.__class__.__name__
    t0 = time.time()
    while True:
        if config.save_interval and not agent.total_steps % config.save_interval:
            agent.save('data/%s-%s-%d' % (agent_name, config.tag, agent.total_steps))
        if config.eval_interval and not agent.total_steps % config.eval_interval:
            agent.eval_episodes()
        if config.max_steps and agent.total_steps >= config.max_steps:
            agent.close()
            break
        agent.step()
        agent.switch_task()
```

Figure 10: Code Snippet for DDPG training

5 Result

5.1 DQN with CartPole-v0

In this lab, I train the DQN on the *CartPole-v0* for 1,000 episodes in total. The size of replay buffer is set to 5,000, and the discounting rate γ is set to 0.95. I use Adam optimizer with learning rate 0.0005 as well as batch size 128. The target Q network is updated using Q network every 50 step.

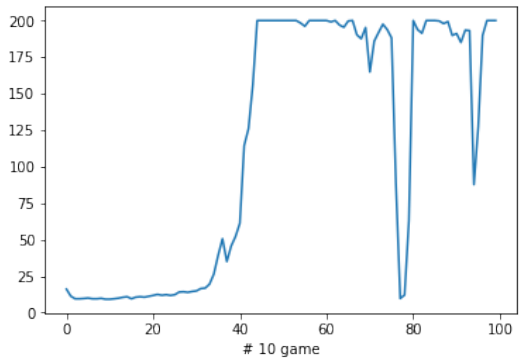

Figure 11: Returns Trend of DQN

After training, I run another 100 test episode to evaluate the agent. And the average return is 200 (max return the agent can get since the game is terminated after 200 step), which means the agent is able to balance the pole entire game.

5.2 DDPG with Pendulum

In this lab, I train the DDPG on the *CartPole-v0* for 10,000 episodes in total. The size of replay buffer is set to 10,000, and the discounting rate γ is set to 0.95. Adam optimizer with batch size 64 is used for both critic network and actor network, and the learning rate is set to 0.001 for critic network as well as 0.0001 for actor network. The mix rate τ for soft update target network is set to 0.001 for both critic and actor network.

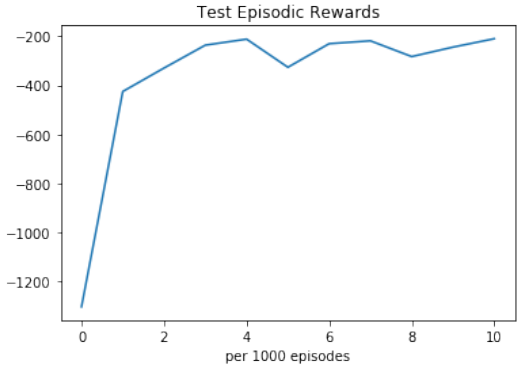

Figure 12: Returns Trend of DDPG

After training, I run another 100 test episode to evaluate the agent, and the average return is -210.32.

5.3 Overall Result

Table 1: Return Of DQN / DDPG

	Test Return
CartPole-v0 (DQN)	200.0
Pendulum (DDPG)	-210.32