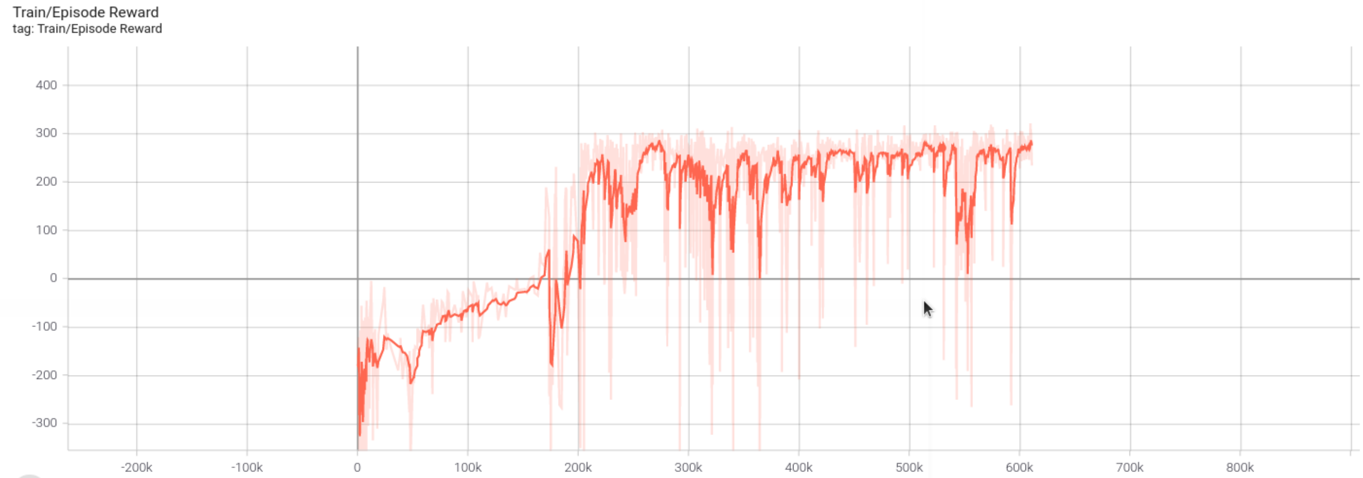
HW6 DQN-DDPG

tags: **`DL and Practice`**

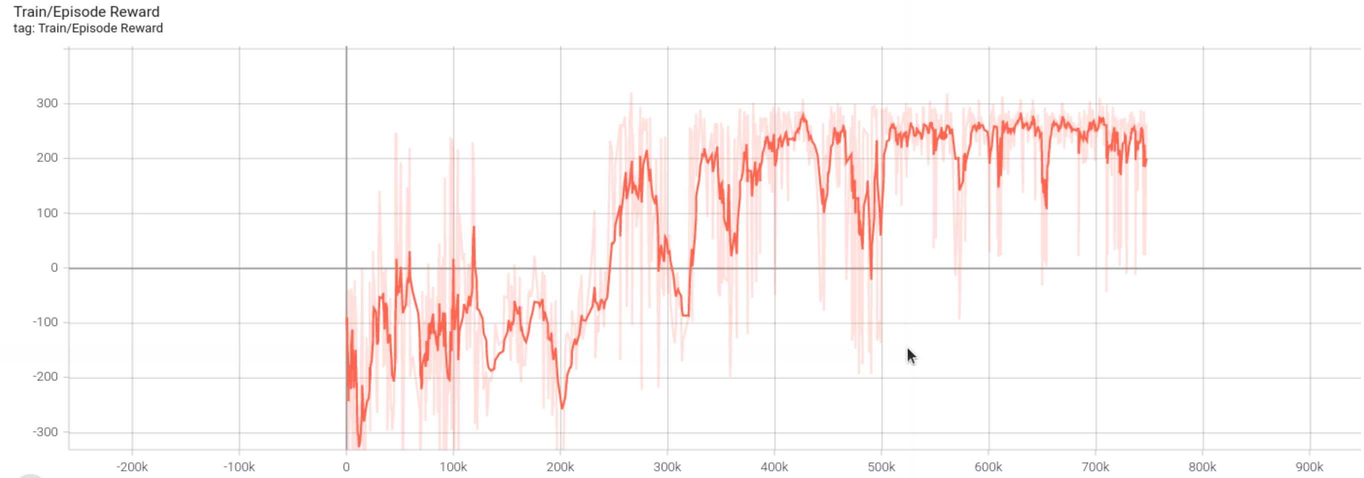
電控碩一 黃柏叡 309512074

Report

1. A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2

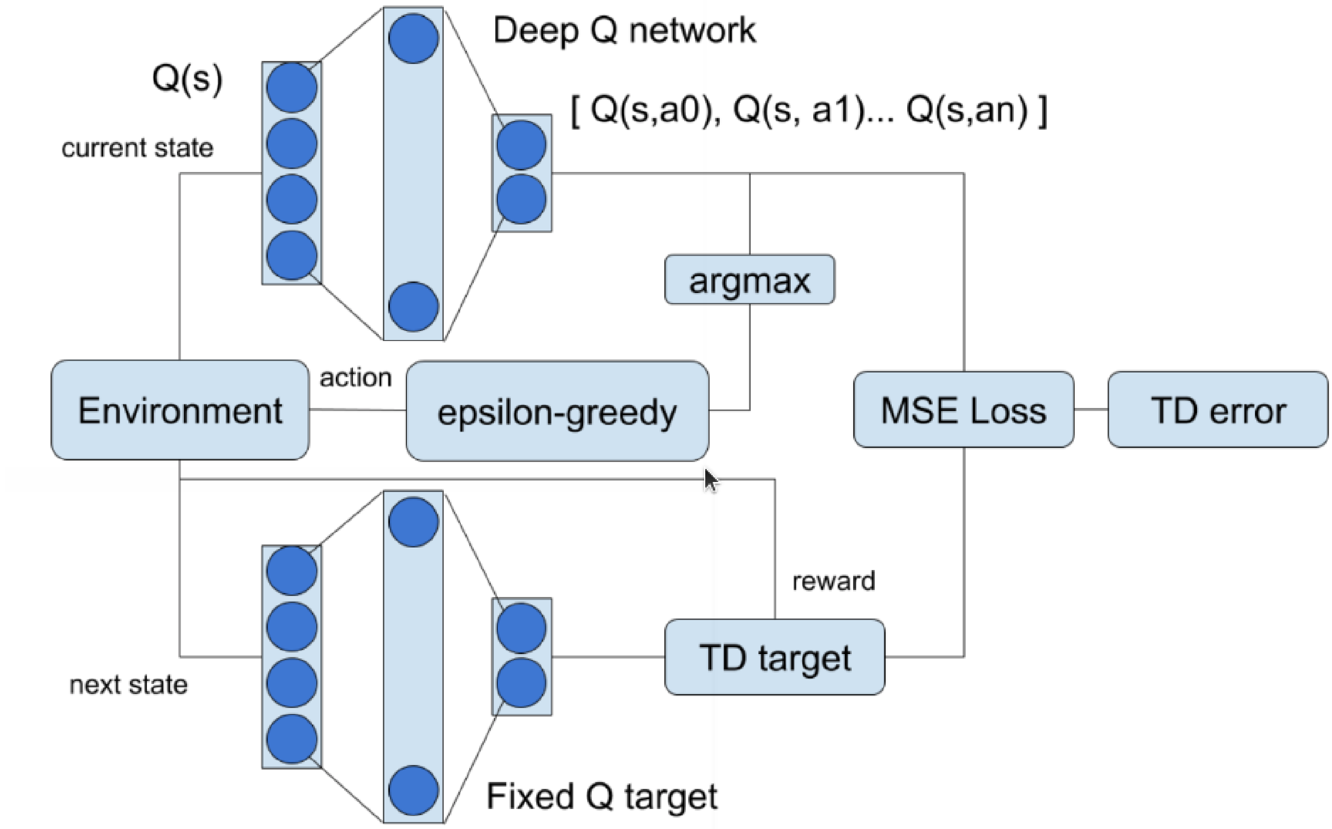


2. A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2



3. Describe your major implementation of both algorithms in detail

- DQN Structure



建立一個network來預測Q(s,a)的value, LunarLander-v2有4種action,所以最後一層為4個neuron

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=(300, 300)):
        super(Net, self).__init__()
        ## TODO ##
        self.fc1=nn.Linear(state_dim, hidden_dim[0])
        self.fc2=nn.Linear(hidden_dim[0],hidden_dim[1])
        self.fc3=nn.Linear(hidden_dim[1],action_dim)
        self.relu=nn.ReLU()

    def forward(self, x):
        ## TODO ##
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

在episode中，選擇value最大的Q(s, ai)的ai或有一定機率(epsilon)隨機選擇action

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    if random.random() < epsilon:  # explore
        return action_space.sample()
    else: # exploit
        with torch.no_grad():
            return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.devic
```

update behavior network是由replay memory中sample一些遊戲的過程，(state, action, reward, next_state, done)做td-learning，在對q_value跟q_target(reward+gamma*max Q'(s',a'))做MSELoss

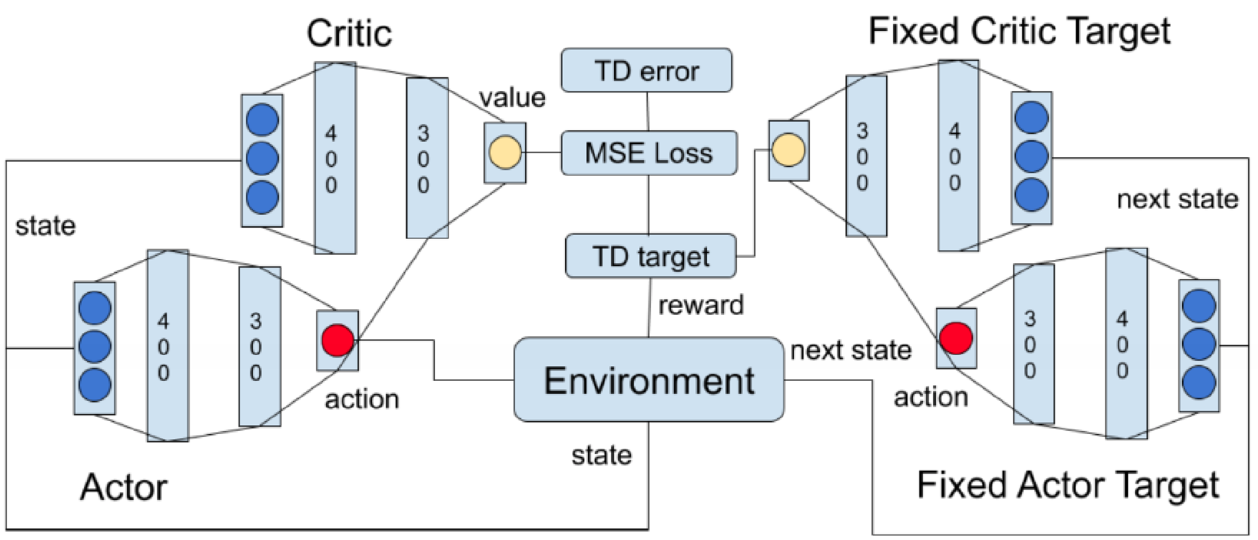
```
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)
    ## TODO ##
    q_value = self._behavior_net(state).gather(dim=1,index=action.long())
    with torch.no_grad():
        q_next = self._target_net(next_state).max(dim=1)[0].view(-1,1)
        q_target = reward + gamma*q_next*(1-done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)

    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

最後每隔一段時間(設1000個iterations)，用behavior network取代target network

```
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

- DDPG structure



建立一個可以依據目前state決定要執行哪個action的Actor Network，有2個action，最後一層為2個neuron

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(300, 300)):
        super(ActorNet, self).__init__()
        ## TODO ##
        self.fc1=nn.Linear(state_dim,hidden_dim[0])
        self.fc2=nn.Linear(hidden_dim[0],hidden_dim[1])
        self.fc3=nn.Linear(hidden_dim[1],action_dim)
        self.relu=nn.ReLU()
        self.tanh=nn.Tanh()

    def forward(self, x):
        ## TODO ##
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.tanh(self.fc3(x))
        return x
```

建立一個可以預估Q(s,a)的Critic Network

```
class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(300, 300)):
        super(CriticNet, self).__init__()
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, hidden_dim[0]),
            nn.ReLU(),
        )
        self.critic = nn.Sequential(
            nn.Linear(hidden_dim[0], hidden_dim[1]),
            nn.ReLU(),
            nn.Linear(hidden_dim[1], 1),
        )

    def forward(self, x, action):
        x = self.critic_head(torch.cat([x, action], dim=1))
        return self.critic(x)
```

episode中，由Actor Network選擇action

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:
            re = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))+t
        else:
            re = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))
    return re.cpu().numpy().squeeze()
```

在episode中，要更新behavior network的actor以及critic，以及target network的actor以及critic。利用target network產生的q_target與behavior network產生的q_value做MSE loss

```
# sample a minibatch of transitions
state, action, reward, next_state, done = self._memory.sample(
    self.batch_size, self.device)

## update critic ##
# critic loss
## TODO ##
q_value = self._critic_net(state,action)
with torch.no_grad():
   a_next = self._target_actor_net(next_state)
   q_next = self._target_critic_net(next_state,a_next)
   q_target = reward + gamma*q_next*(1-done)
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)

# optimize critic
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()
```

使用behavior network的Actor Network與Critic Network可以求出Q(s,a)，我們想更新Actor Network來使輸出的Q(s,a)越大越好，因此定義Loss Value為mean(-Q(s,a))，在通過backpropagation更新

```
## update actor ##
# actor loss
## TODO ##
action = self._actor_net(state)
actor_loss = -self._critic_net(state,action).mean()

#bp
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

4. Describe differences between your implementation and algorithms

- DQN是一種基於值函數的方法，基於值函數的方法難以應對的是大的動作空間，特別是連續動作情況。因為網絡難以有這麼多輸出，且難以在這麼多輸出之中搜索最大的Q值。而DDPG是基於Actor-Critic方法，在動作輸出方面採用一個網絡來擬合策略函數，直接輸出動作，可以應對連續動作的輸出及大的動作空間。
- DQN不會每個iteration都更新behavior network，而是每隔一段時間(設4個iteration)才會更新一次

5. Describe your implementation and the gradient of actor updating

$$L = -Q(s, a|\theta_Q), \; a = u(s|\theta_u)$$

$$\begin{aligned} \frac{\nabla L}{\nabla \theta_u} &= -\frac{\nabla Q(s,a|\theta_Q)}{\nabla a} \frac{\nabla a}{\nabla u(s|\theta_u)} \frac{\nabla u(s|\theta_u)}{\nabla \theta_u} \\ &= -\frac{\nabla Q(s,a|\theta_Q)}{\nabla u(s|\theta_u)} \frac{\nabla u(s|\theta_u)}{\nabla \theta_u} \end{aligned}$$

利用Behavior Network的Actor Network u與Critic Network Q可以求出Q(s,a),我們想更新Actor network u來使輸出的Q(s,a)越大越好，因此定義loss function為mean(-Q(s,u(s)))

```
## update actor ##
# actor loss
## TODO ##
action = self._actor_net(state)
actor_loss = -self._critic_net(state,action).mean()

#bp
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

6. Describe your implementation and the gradient of critic updating

利用Target Network產生的Q~target~與Behavior network產生的Q(s,a)做MSE loss來更新Q Network(critic network)

```
# sample a minibatch of transitions
state, action, reward, next_state, done = self._memory.sample(
    self.batch_size, self.device)

## update critic ##
# critic loss
## TODO ##
q_value = self._critic_net(state,action)
with torch.no_grad():
   a_next = self._target_actor_net(next_state)
   q_next = self._target_critic_net(next_state,a_next)
   q_target = reward + gamma*q_next*(1-done)
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)

# optimize critic
```

```
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()
```

7. Explain effects of the discount factor

$$G_t = R_{t+1} + \lambda R_{t+2} + \ldots = \sum_{k=0}^{\infty} \lambda^k R_{t+k+1}$$

λ是discount factor，越未來所給的reward影響越來越小，當下的reward影響是最大的

8. Explain benefits of epsilon-greedy in comparison to greedy action selection

必須在explore與exploit之間取得平衡，因此在greedy action selection的基礎上，偶而必須選擇其他action來探索雖然是未知但可能是最佳的action

9. Explain the necessity of the target network

target network可以使training更穩定，因為產生Q_target的Target Network(更新較慢)每隔一段時間(設1000個iterations)才會被behavior network更新

10. Explain the effect of replay buffer size in case of too large or too small

如果replay buffer size越大，training會越穩定，但會降低training速度。如果replay buffer size越小，training會著重在最近的episode的狀況，容易造成overfitting problem

Report Bonus

1. Implement and experiment on Double-DQN

DDQN跟DQN的差異不大，只差在update behavior network時是如何決定Q_target的，DDQN在決定Q_target時，不是直接取max Q'(s,ai)，而是用Q(s,ai)中的最大值的index作為Q'(s,ai)的index

```
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(self.batch_size, sel
    ## TODO ##
    q_value = self._behavior_net(state).gather(dim=1,index=action.long())
    with torch.no_grad():
        action_index=self._behavior_net(next_state).max(dim=1)[1].view(-1,1)
        q_next = self._target_net(next_state).gather(dim=1,index=action_index.long())
        q_target = reward + gamma*q_next*(1-done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)

    # bp
    self._optimizer.zero_grad()
```

```
        loss.backward()
        nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
        self._optimizer.step()
```

DDQN: 2000個episode

```
Start Testing
total reward: 251.00
total reward: 282.76
total reward: 281.78
total reward: 276.48
total reward: 176.59
total reward: 271.91
total reward: 179.98
total reward: 291.18
total reward: 304.77
total reward: 301.63
Average Reward 261.80844044817496
```

2. Extra hyperparameter tuning, e.g., Population Based Training

Performance

1. [LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30

DQN: 2000個episode

```
Start Testing
total reward: 247.75
total reward: 287.79
total reward: 278.21
total reward: 262.65
total reward: 312.55
total reward: 271.76
total reward: 310.95
total reward: 291.24
total reward: 315.63
total reward: 302.00
Average Reward 288.05429527968823
```

2. [LunarLanderContinuous-v2] Average reward of 10 testing episodes: Average ÷ 30

DDPG: 2000個episode

```
Start Testing
total reward: 252.48
total reward: 287.00
total reward: 267.22
total reward: 285.18
total reward: 275.31
total reward: 202.65
total reward: 302.47
total reward: 283.34
total reward: 312.92
total reward: 239.79
Average Reward 270.83436389363
```