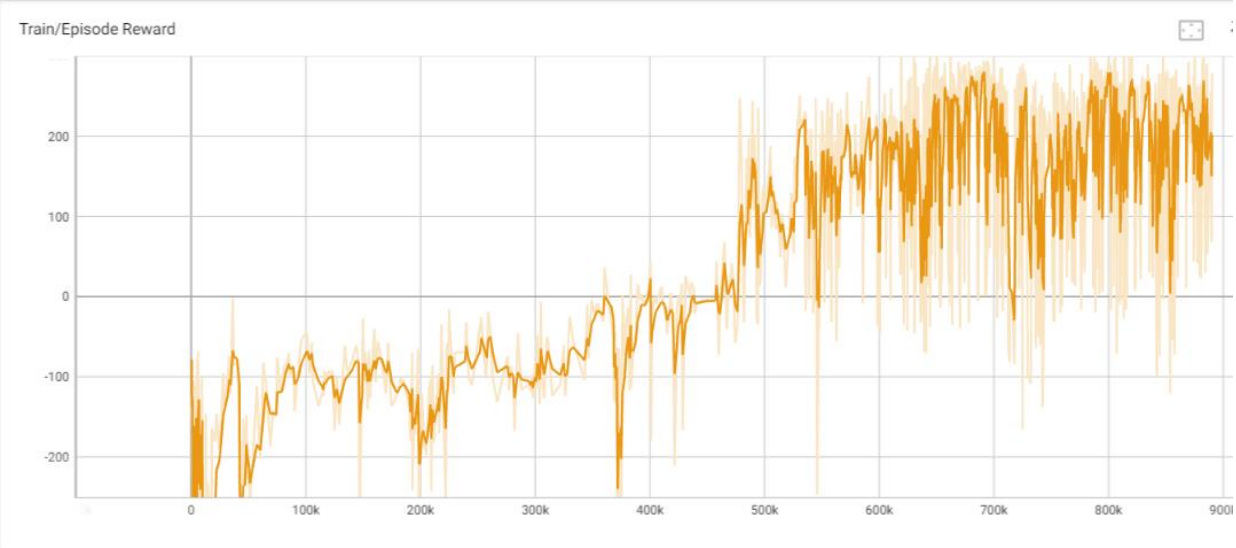
Deep Learning and Practice Lab6

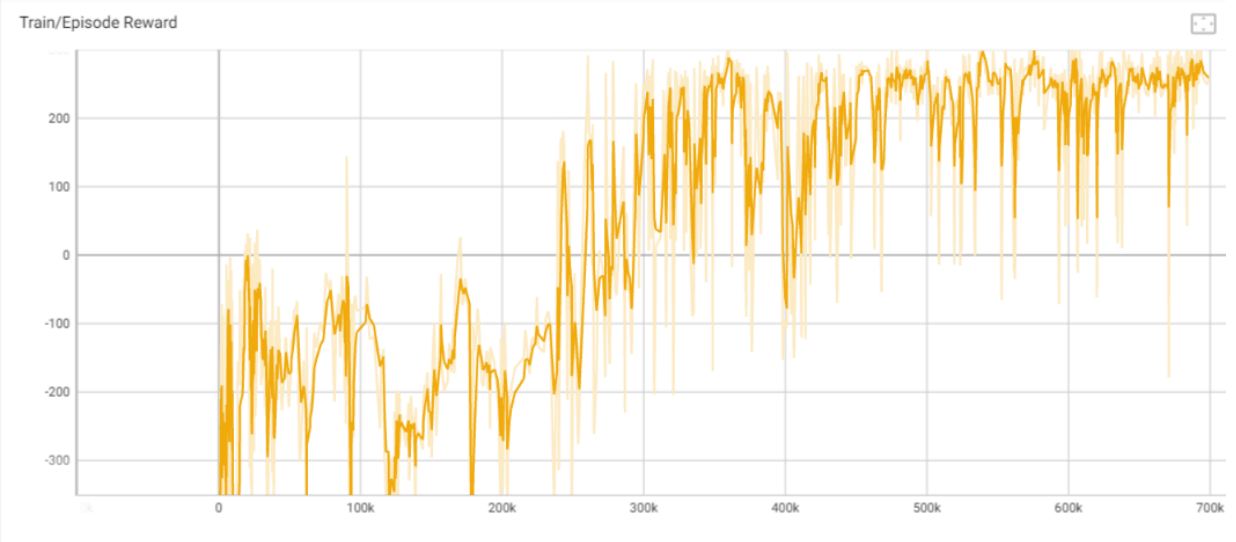
311605011

黃品振

1. tensor board plot (at least 800 training episodes in LunarLander-v2)



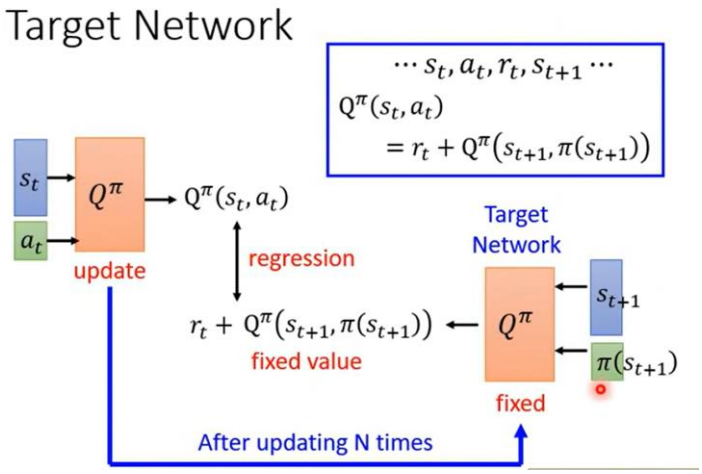
2. A tensor board plot (at least 800 training episodes in LunarLanderContinuous-v2)



3. DQN(including how I implement DQN algorithm)

DQN 的概念是利用神經網路去估算出在一個 state 各種 action 的 value，去有效的解決在 Q-learning 中多種狀態難以建造以及更新 Q-table 的問題(計算量大，占記憶體空間)。

這次的環境在只有 4 種動作的情況下，我們將 state 輸入，就只會有 4 種不同動作對應得到的 value。我們會有兩個神經網路需要訓練，如下圖所示:



需要訓練的網路分別為上圖的兩個 Qpi(behavior、target)，我們會先訓練左邊 Qpi(behavior)去讓輸出逼近右邊的 target 的輸出加上 rt
(即為 loss function)，在訓練了一定程度後，再將右邊的網路替換成左邊的，不斷循環。

$$L(w) = \mathbb{E}[(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{Target} - Q(s, a, w))^2]$$

Loss function

創建網路:

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)


    def forward(self, x):
        ## TODO ##
        x = self.fc1(x)
        x = nn.ReLU()(x)
        x = self.fc2(x)
        x = nn.ReLU()(x)
        x = self.fc3(x)

        return x
```

選擇動作:

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    rnd = random.random()
    if rnd < epsilon:
        return np.random.randint(action_space.n)
    else:
        state = torch.tensor(state).float().unsqueeze(0).cuda()
        with torch.no_grad():
            action_value = self._behavior_net(state)

        action = np.argmax(action_value.cpu().data.numpy())

    return action
```

從上面可以看出，動作選擇不一定總會是值選最大的的那個，因為在訓練的一開始需要多探索一些可能性來增加資料的各種可能性。到了之後才會逐漸傾向於選擇值最大的動作。

選擇 Optimizer:

```
    ## TODO ##
    # self._optimizer = ?
    self._optimizer = torch.optim.Adam(self._behavior_net.parameters(), lr=args.lr)
```

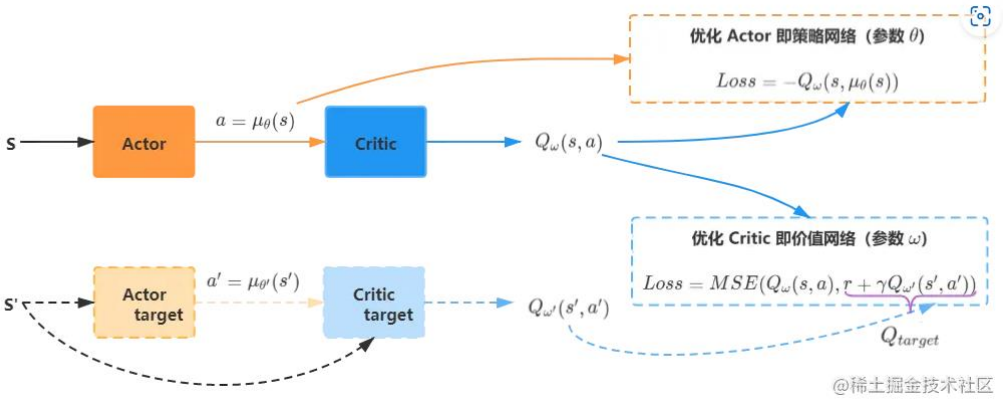
update:

```
q_value = self._behavior_net(state).gather(1, action.long())
with torch.no_grad():
    q_next = self._target_net(next_state)
    q_target = reward + gamma * q_next.max(1)[0].view(self.batch_size, 1)
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()
```

4. DDPG(Including difference between DQN and DDPG、DDPG algorithm、How I update actor and critic)

DDPG 與 DQN 的不同點是在 DDPG 可以用於處理連續動作，且相較於 DQN 多了一個 actor，目標是使 Q 值最大，詳細流程圖如下圖所示:



建立 Actor 網路:

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        hidden_dim_1, hidden_dim_2 = hidden_dim
        self.fc1 = nn.Linear(state_dim, hidden_dim_1)
        self.fc2 = nn.Linear(hidden_dim_1, hidden_dim_2)
        self.fc3 = nn.Linear(hidden_dim_2, action_dim)

        # raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        x = self.fc1(x)
        x = nn.ReLU()(x)
        x = self.fc2(x)
        x = nn.ReLU()(x)
        x = self.fc3(x)
        x = nn.Tanh()(x)

        return x
```

建立 Critic 網路:

```
class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.critic_head = nn.Sequential(
            nn.Linear(state_dim + action_dim, h1),
            nn.ReLU(),
        )
        self.critic = nn.Sequential(
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, action_dim),
        )

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

選擇 action:

```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.tensor(state).float().cuda()
    action = self._actor_net(state).cpu().data.numpy()
    action = self._action_noise.sample() + action
    return action
    # raise NotImplementedError
```

Update:

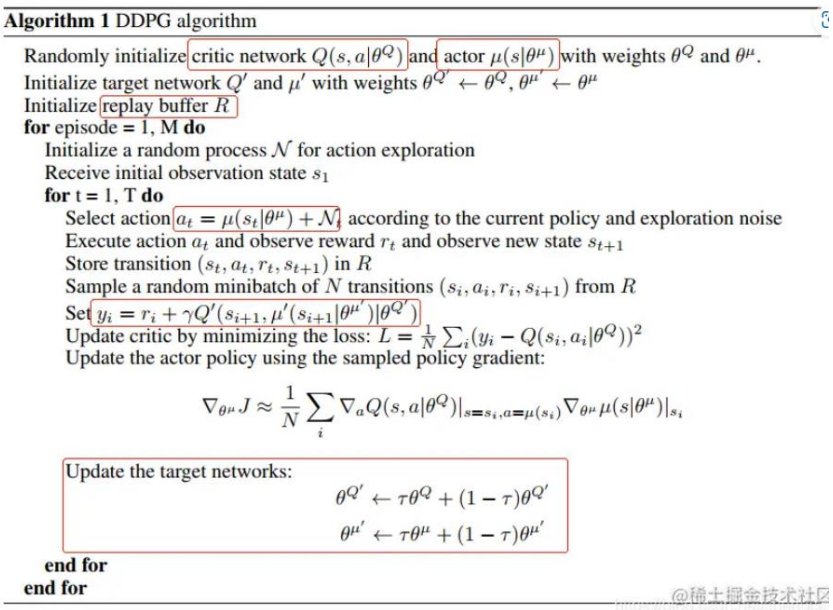
```python
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)

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

## update actor ##
# actor loss
## TODO ##
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()

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

網路的架構以及更新可以由下面 pseudo code 表示:



Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

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

  end for
end for

訓練的流程其實跟 DQN 差不多，最大的差別在於 DDPG 會同時計算 action

以及 Q-value，利用 Q-value 去計算 TD-error 去更新 critic network，並將 Q-value 取負號當成 Loss 去更新 actor network，這裡的概念有點像是讓自輸出的期望值更高，在做 Gradient absent。除此之外，DDPG 中也會使用 DQN 中的 target 以及 behavior 技巧，使整體訓練更穩定，在訓練一定程度後，將 behavior 網路中的權重直接複製到 target 裡面。

5. Explain effects of the discount factor:

Discount factor，通常的代號為 γ ，從 Q-learning 公式中可得知，discount factor (γ)可控制 agent 較重視眼前的 reward 或是從歷史資料計算出的長期利益，若 γ 越小，則 agent 越短視近利，只重視眼前 reward；若 γ 越大，則 agent 越重視長期利益。

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

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

在一開始為了讓資料蒐集能有比較多可能性，會使用隨機採樣的方式，在一定的次數之後，就會傾向於使用能得到最大值的動作。

7. Explain the necessity of the target network

由於在 DQN 中是用 Neural Network 來取代 Q-table，而 Q-learning 中的 Q 值遞迴關係會導致 Q 值一值在變動，導致訓練困難，因此會先設立一個 Target Network 去使結果穩定，因為這個網路會是固定的，一直到 behavior Network 訓練到一定程度，再將參數複製過去，就可以使模型預測更穩定。

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

如果 Buffer size 太大，會導致舊有的資料去影響到訓練到已經比較好的結過。但是如果太小又會使資料的相依性太高，導致可能 over-fitting。

9. Experimental results:

DQN:

```
(pytorch) D:\NCTU\110summer\deeplearn
Start Testing
Average Reward 213.03837774733157
```

DDPG

```
Start Testing
total_reward:  250.4868184609601
total_reward:  271.37436561601123
total_reward:  278.46620134262776
total_reward:  276.1502975971613
total_reward:  266.5954655705442
total_reward:  265.4558285575851
total_reward:  263.116537591602
total_reward:  272.61315105744063
total_reward:  290.6625497421003
total_reward:  265.35620597329864
Average Reward 270.027742150933
```