Deep Learning and Practice

Lab6 - Deep Q-Network and

Deep Deterministic Policy Gradient

310605009
吳公耀

1. Report (80%)

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

  助教推薦的參數但減少capacity成100000並且跑2000episodes

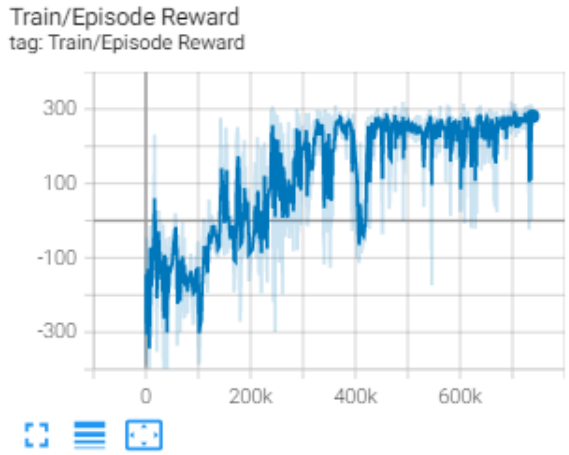

```
Start Testing DQN
total reward 246.67299245789118
total reward 230.74960396176024
total reward 277.66354114256717
total reward 272.1992212376143
total reward 248.6484589520876
total reward 271.8114874952922
total reward 236.76183414615107
total reward 220.49122993887312
total reward 257.1001150843582
total reward 294.996981224951
Average Reward 255.7095465641546
```

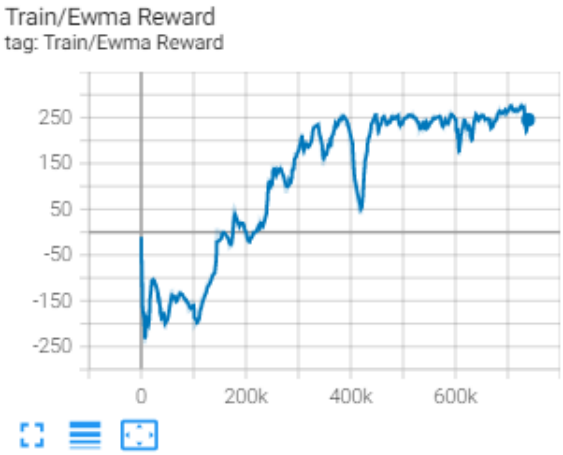
Average Reward 255.709

有嘗試過加大capacity但是效果沒有提升，也有將learning rate變大，效果都沒有比較好，也沒有建議的參數穩定，偶爾會也突然變小的值。相較其他方法都需要較大的episodes才能有比較好且穩定的結果。

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

助教推薦的參數並且跑2000episodes






```
Start Testing DDPG
total reward:  249.53064350586732
total reward:  284.6194996468789
total reward:  284.7317892448615
total reward:  277.2256274582439
total reward:  311.9917798402911
total reward:  272.7406124244343
total reward:  304.85600279250195
total reward:  297.0496652775788
total reward:  315.8809095689063
total reward:  299.8702671578461
Average Reward 289.849679691741
```

Average Reward 289.849

DDPG的結果，相比DQN需地的時間比較長但是很快就訓練起來了，效果也比DQN來的好，且沒有那麼的動盪。

- **Describe your major implementation of both algorithms in detail**

這次助教也有提供sample code所以主要說明自己實作的部分

DQN

創建DQN模型的部分，輸入state，輸出每個action的value，按照助教提供的網路架構實作(8,32,32,4)，每個layer的neural數按照助教提供的參數來實作。

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=64 ):
        super().__init__()
        ## TODO ##

        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim * 2)
        self.fc3 = nn.Linear(hidden_dim * 2, action_dim)

        #raise NotImplementedError


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

        return x
        #raise NotImplementedError
```

Select action function，使用的策略 是 epsilon greedy function，選擇在某一個state的情況下，要執行哪個動作，通常選擇最好的action但是為了避免忽略其他不錯的action所以設定一個epsilon並讓他隨機取random值如果大於epsilon從behavior network中選擇最好的action來動作，否則隨機選擇一個動作進行，而 epsilon會隨step越多而下降，使得從behavior network中選擇最好的action的機率提升。

```
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.from_numpy(state).float().unsqueeze(0).cuda()
        with torch.no_grad():
            actions_value = self._behavior_net.forward(state)
        action = np.argmax(actions_value.cpu().data.numpy())

    return action
```

update behavior network function用來更新behavior network，先從以往的step隨機抽state出來進行訓練，有了舊的state就先讓behavior看這個 state輸出的以往做的action的 Q value，然後用target network找 下個state最大的 Q’ value，把這個Q’ value乘上 gamma後加上reward，當作 target Q，那如果做了這個 action遊戲結束，所以就只會剩下 reward。接著用MSE loss計算target Q跟value Q計算他們的差然後backward回去更新weight，這部分不會更新到 target network。

```
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 = ?
    # with torch.no_grad():
    #     q_next = ?
    #     q_target = ?
    # criterion = ?
    # loss = criterion(q_value, q_target)

    q_value = self._behavior_net(state).gather(1, action.long())

    with torch.no_grad():
        q_next = self._target_net(next_state)  # Not Backpropagate
        q_target = reward + gamma * q_next.max(1)[0].view(self.batch_size, 1)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
    #raise NotImplementedError
    # optimize
    self._optimizer.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
    self._optimizer.step()
```

Optimizer使用助教建議的Adam。

```
class DQN:
    def __init__(self, args):
        self._behavior_net = Net().to(args.device)
        self._target_net = Net().to(args.device)
        # initialize target network
        self._target_net.load_state_dict(self._behavior_net.state_dict())
        ## TODO ##
        # self._optimizer = ?
        self._optimizer = torch.optim.Adam(self._behavior_net.parameters(), lr=args.lr)

        #raise NotImplementedError
        # memory
        self._memory = ReplayMemory(capacity=args.capacity)

        ## config ##
        self.device = args.device
        self.batch_size = args.batch_size
        self.gamma = args.gamma
        self.freq = args.freq
        self.target_freq = args.target_freq
```

Update target network，這就是更新 target network，每100個episode把behavior network的參數複製到target network上。

```
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())
    #raise NotImplementedError
```

ReplayMemory class用來儲存以往的step的class，有儲存上限，用deque來儲存 ，超過上限的state會從最舊的開始刪除。

Test function，以前面訓練好的network來進行，因為是test所以不使用隨機選擇action，用select action funciton來選擇這個state要進行什麼 action，然後執行得到新state，一直重複直到遊戲結遊戲結束。

```
def test(args, env, agent, writer):
    print('Start Testing DQN')
    action_space = env.action_space
    epsilon = args.test_epsilon
    seeds = (args.seed + i for i in range(10))
    rewards = []
    for n_episode, seed in enumerate(seeds):
        total_reward = 0
        env.seed(seed)
        state = env.reset()
        ## TODO ##
        # ...
        while True:
            action = agent.select_action(state, epsilon, action_space)
            epsilon = max(epsilon * args.eps_decay, args.eps_min)# selection
            next_state, reward, done, _ = env.step(action)# execute
            agent.append(state, action, reward, next_state, done) # transition

            state = next_state
            total_reward += reward
            # total_steps += 1
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                rewards.append(total_reward)
                print('total reward: ', total_reward)
                break
        #         ...
        #raise NotImplementedError
    print('Average Reward', np.mean(rewards))
    env.close()
```

DDPG

ReplayMemory的sample function，主要就是從儲存舊state的地方 隨機抽取出state用於訓練。

```
class ReplayMemory:
    __slots__ = ['buffer']

    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, *transition):
        # (state, action, reward, next_state, done)
        self.buffer.append(tuple(map(tuple, transition)))

    def sample(self, batch_size, device):
        '''sample a batch of transition tensors'''
        ## TODO ##
        trans_sample = random.sample(self.buffer, batch_size)

        return (torch.tensor(x, dtype=torch.float, device=device) for x in zip( *trans_sample ))
```

按照助教提供來實作actor network的架構(8,400,300,2)，中間activation function是 relu，最後面輸出則是 tanh。

```
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 ##

        action_x = F.relu(self.fc1(x))
        action_x = F.relu(self.fc2(action_x))
        action_x = torch.tanh(self.fc3(action_x))
        return action_x
```

critic network，輸入state action，輸出Q value 。

```
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)
```

Optimizer一樣也是使用助教建議的使用Adam。

```
class DDPG:
    def __init__(self, args):
        # behavior network
        self._actor_net = ActorNet().to(args.device)
        self._critic_net = CriticNet().to(args.device)
        # target network
        self._target_actor_net = ActorNet().to(args.device)
        self._target_critic_net = CriticNet().to(args.device)
        # initialize target network
        self._target_actor_net.load_state_dict(self._actor_net.state_dict())
        self._target_critic_net.load_state_dict(self._critic_net.state_dict())
        ## TODO ##
        self._actor_opt = torch.optim.Adam(self._actor_net.parameters(), lr=args.lra)
        self._critic_opt = torch.optim.Adam(self._critic_net.parameters(), lr=args.lrc)
        # self._actor_opt = ?
        # self._critic_opt = ?
        # raise NotImplementedError
        # action noise
        self._action_noise = GaussianNoise(dim=2)
        # memory
        self._memory = ReplayMemory(capacity=args.capacity)

        ## config ##
        self.device = args.device
        self.batch_size = args.batch_size
        self.tau = args.tau
        self.gamma = args.gamma
```

Select action function從目前的 state，輸入到actor network中， 輸出預測目前action的值，這部分會加上 gaussian noise，而在 test時就不會加上noise。

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.from_numpy(state).float().cuda()

    sel_act = self._actor_net(state).cpu().data.numpy()
    sel_act = self._action_noise.sample() + sel_act
    return sel act
```

Update behavior network，用 target network來更新behavior network， 一樣從舊的state來sample找出出state進行訓練 ，先用目前的state拿到 Q value，然後把下一個state輸入到target actor network後得到action值，在輸入到target critic network中得到 Q’ value，在把這個值乘上gamma，加上reward就是target Q。如果 next state結束了就會只剩reward當作target，loss function是使用MSE Loss，上面的是critic loss，而actor loss則是 critic network更新之後再用 state跑一次，把state跟action輸入到更新後的 critic network中，得到的值取負號當作 gradient來更新。

```
q_value = critic_net(state, action)


with torch.no_grad():
    a_next = target_actor_net(next_state)
    q_next = target_critic_net(next_state, a_next)
    q_target = reward + (self.gamma * q_next * (1 - done))

criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)
# raise NotImplementedError
# 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()
# raise NotImplementedError
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

test 的部份，就依照目前的state去選擇action進行，這部分就不使用 noise，把action執行得到新的state， 累加reward，直到遊戲結束。

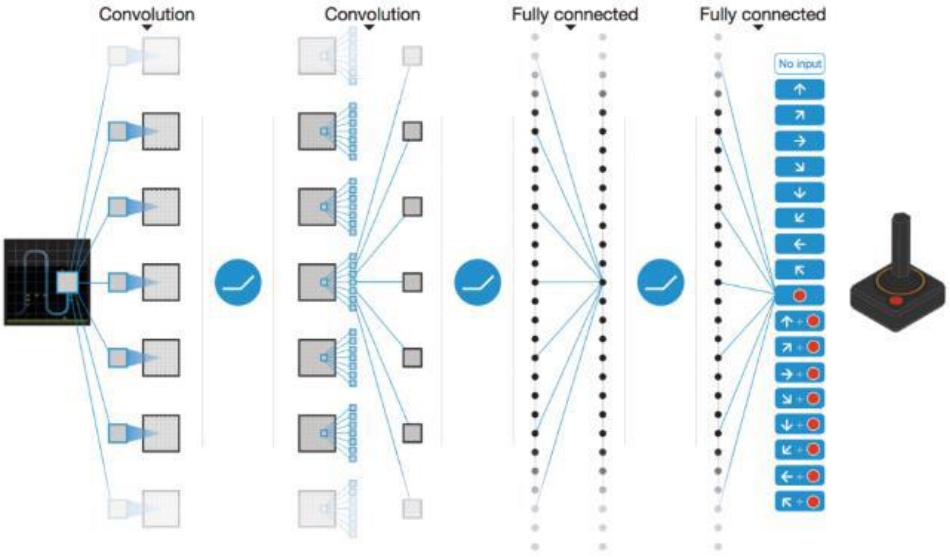
```
def test(args, env, agent, writer):
    print('Start Testing DDPG')
    seeds = (args.seed + i for i in range(10))
    rewards = []
    for n_episode, seed in enumerate(seeds):
        total_reward = 0
        env.seed(seed)
        state = env.reset()
        ## TODO ##############################
        for _ in range(1000):
            action = agent.select_action(state, False)
            state, reward, done, _ = env.step(action)
            total_reward += reward
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                break

        rewards.append(total_reward)
        print('total reward: ', total_reward)

    print('Average Reward', np.mean(rewards))
    env.close()
```

- **Describe differences between your implementation and algorithms**

DQN



Off-policy是Q-Learning的特點，DQN中也延用了這一特點。而不同的是，Q-Learning中用來計算target和預測值的Q是同一個Q，也就是說使用了相同的神經網絡。這樣帶來的一個問題就是，每次更新神經網絡的時候，target也都會更新，這樣會容易導致參數不收斂。回憶在有監督學習中，標籤label都是固定的，不會隨著參數的更新而改變。

因此DQN在原來的Q-Network的基礎上又引入了一個**target Q**網絡，即用來計算target的網絡。它和Q-Network結構一樣，初始的權重也一樣，只是Q-Network每次迭代都會更新，而target Q-Network是每隔一段時間才會更新。DQN的target是

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

DQN所做的改進：

相比於Q-Learning，DQN做的改進：一個是使用了卷積神經網絡來逼近行為值函數，一個是使用了target Q network來更新target，還有一個是使用了經驗回放Experience replay。 由於在強化學習中，我們得到的觀測數據是有序的，step by step的，用這樣的數據去更新神經網絡的參數會有問題。回憶在有監督學習中，數據之間都是獨立的。因此DQN中使用經驗回放，即用一個Memory來存儲經歷過的數據，每次更新參數的時候從Memory中抽取一部分的數據來用於更新，以此來打破數據間的關聯。

DQN算法整體流程 ：

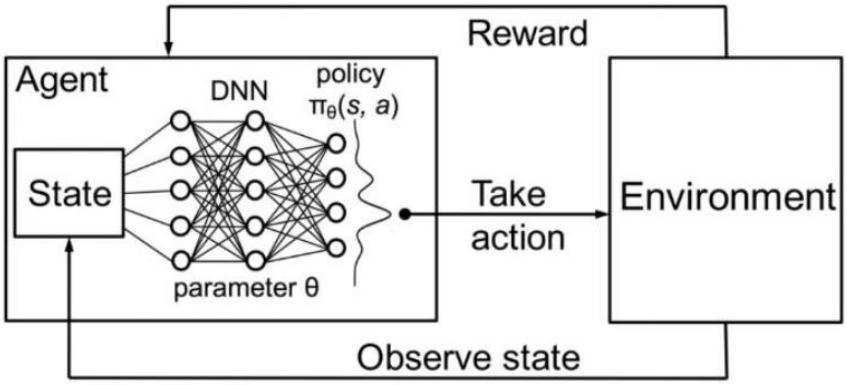
Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

DQN 在 Atari 遊戲中，輸入是 Atari 的遊戲畫面，使用卷積神經網絡（CNN）來處理 pixel 的資訊。

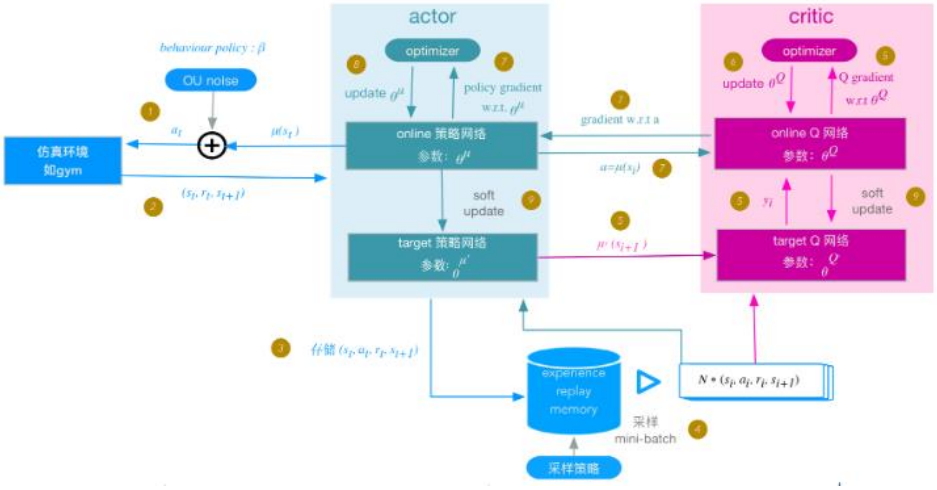
Q-Learning是傳統RL演算法，在算法中，有一個稱為 Q Function 的函數，用於根據狀態估計獎勵。稱之為 Q（s，a），其中 Q 是一個函數，它從狀態 s 和動作 a 計算預期的未來值。

在 DQN 中，使用神經網絡代替原本的 Q 值。



鑑於環境的狀態state是該網絡的圖像輸入，它會嘗試預測所有可能的操作（如 regression 問題）的預期最終報酬reward，選擇具有最大預測 Q 值的動作作為我們在環境中採取的動作action。

DDPG



DDPG 可以分為‘DEEP’和‘Deterministic Policy Gradient’，然後‘Deterministic Policy Gradient’又能被細分為‘Deterministic’和‘Policy Gradient’。
其中 DEEP 就是走向更深層次，使用一個記憶庫和兩套結構相同，但是引數更新頻率不同的神經網路有效促進學習。採用的神經網路類似於DQN，但是DDPG的神經網路形式比DQN要複雜一點。

Deterministic Policy Gradient，相比其他強化學習方法，Policy gradient 能被用來在連續動作上進行動作的篩選，而且篩選的時候是根據所學習到的動作分佈隨機進行篩選。而Deterministic 改變了輸出動作的過程，旨在連續動作上輸出一個動作值。DDPG用到的神經網路有點類似於Actor-Critic，也需要有基於策略Policy 的神經網路 和 基於價值 Value 的神經網路，但是為了體現DQN的思想，每種神經網路我們都需要再細分成倆個。Policy Gradient 這邊有估計網路和現實網路，估計網路用來輸出實時的動作，而現實網路則是用來更新價值網路系統的。再看看價值系統這邊，我們也有現實網路和估計網路，他們都在輸出這個狀態的價值，而輸入端卻有不同，狀態現實網路這邊會拿著當時actor施加的動作當做輸入。在實際運用中，DDPG的這種做法的確帶來了更有效的學習過程。

$\nabla\theta\mu J \approx 1N\Sigma i \nabla aQ(s,a|\theta Q)|s=si,a=u(si)\nabla\theta\mu\mu(s|\theta u)|si$
DDPG 的更新公式可從上面得出，關於Actor 部分，他的引數更新會涉及到Critic， 上面是關於 Actor 引數的更新，它的前半部分 grad[Q] 是從Critic 來的，闡述了Actor 的動作要怎麼移動，才能獲得更大的 Q。而後半部分 grad[u] 是從Actor來的，闡述了 Actor 要怎麼樣修改自身引數，才使得Actor 有可能做這個動作，所以倆者闡述了Actor 要朝著更有可能獲取Q的方向修改動作引數。

DDPG算法整體流程 ：

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:

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

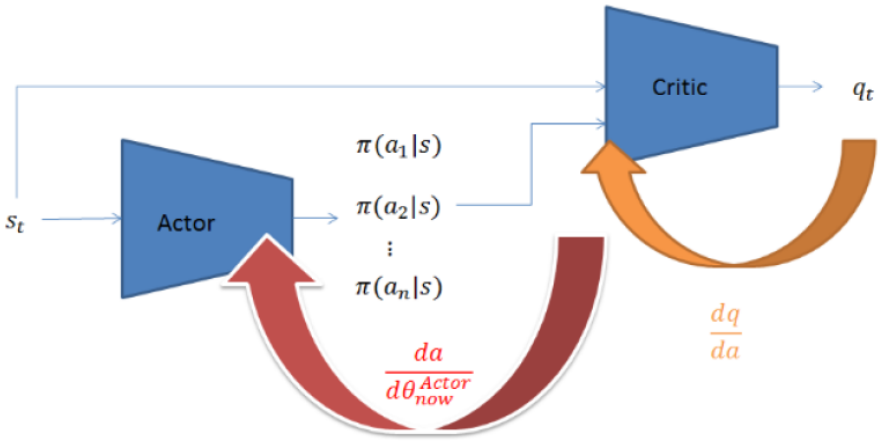
  end for
end for

完整的DDPG 虛擬碼如上(取自論文)，取代target網路的時候論文不是採取複製全部的權重而是使用逐漸的取代並且設置一個參數來決定取代的比重。
且在兩個方法實作上在training的時候會有一段warn up 的時間，這段時間不會更新網路的參數，只會隨機選擇action，並把遊戲過程存到memory裡面。

- **Describe your implementation and the gradient of actor updating**

Actor Network

Actor 更新的概念很新穎，是採取連續微分的方式計算出action應該變動的方向，設計此網路時比較特殊，首先必須終止梯度計算倒傳遞到Actor網路，再來就是將Critic網路dq/da的梯度計算出來，然後再算出Actor網路da/dparams的微分，之後將兩者微分相乘並套用到Actor網路的參數上即可完成更新網路的動作了。

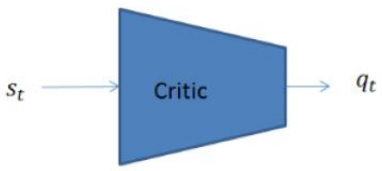


實作的actor network用以往的經驗state來進行學習用過去的經驗來update，在update 完critic之後，直接用來update actor，把目前的state送進actor network，得到目前actor network的action distribution，這跟過去經驗的不相同，將這個action與目前的state作critic，直接將這個value的平均取負數作為loss，主要是因為用critic的value預測結果來判定actor要update多少，把critic value最大化。

- **Describe your implementation and the gradient of critic updating**

Critic network

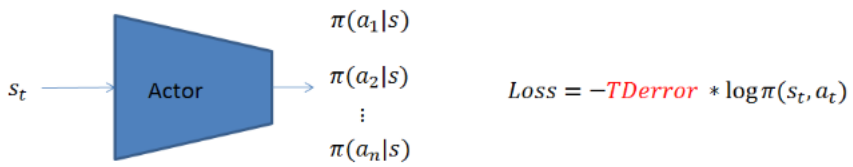
Critic網路是來自於Q-learning，使用TD error當成網路的loss function，訓練的方式很簡單，當收集到一組資訊集的時候(st, at ,s(t+1), rt) 分別把st與 st +1送進網路後可以得到相對應的qt與q(t+1)，將網路輸出的數值與rt代入下列公式中算出loss即可一步一步慢慢的更新網路參數使得TD error越來越小。



TD error

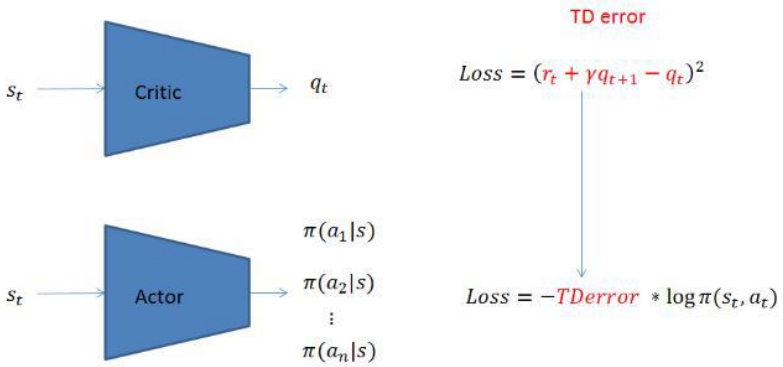
$$Loss = (r_t + \gamma q_{t+1} - q_t)^2$$

Actor 網路更新演算法和之前所敘述的policy gradient更新方式一樣，只是Advantage function換成了Critic網路所提供的TD error。



Actor Critic network

因此，完整的Actor Critic網路表示如下，第一步先用Critic網路算出TD error，第二步再使用TD error對Actor網路進行參數更新



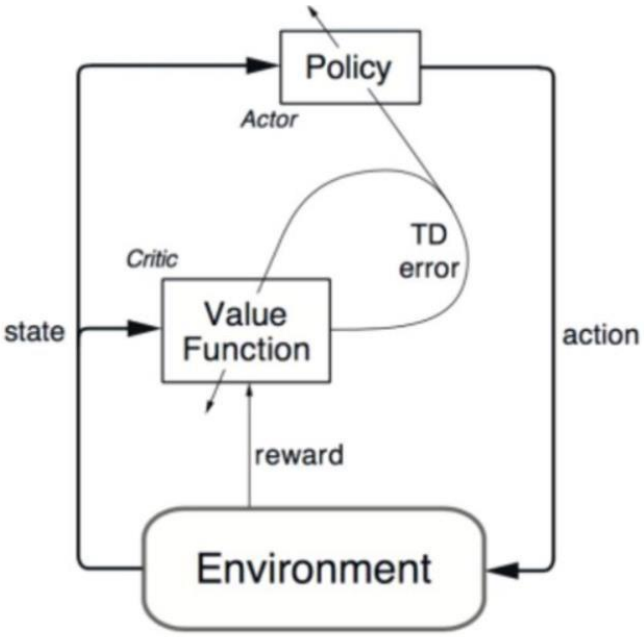
Actor Critic 為類似於Policy Gradient 和 Q-Learning 等以值為基礎的演算法的組合。

a. 其中Actor 類似於Policy Gradient，以狀態s為輸入，神經網路輸出動作actions，並從在這些連續動作中按照一定的概率選取合適的動作action。

b. Critic 類似於 Q-Learning 等以值為基礎的演算法，由於在Actor模組中選擇了合適的動作action，通過與環境互動可得到新的狀態s_，獎勵r，將狀態 s_作為神經網路的輸入，得到v_，而原來的狀態s通過神經網路輸出後得到v。

c. 通過公式 td error=r γv− −v 得到狀態之間的差 td error，最後通過狀態s，動作action，以及誤差 td error 更新Actor網路的引數，實現單步更新。

d. 將s_ 狀態賦予給 s 狀態。



這部分利用以往的經驗state來進行學習，用target network來更新behavior network，先用目前的state還有action輸入到critic network中得到Q value，然後把下一個state輸入到target actor network內，得到action值，在輸出到target critic network中，得到Q' value，在把這個值乘上gamma，加上reward就是target Q，然後如果next state結束了就會只剩reward當作target，而yi就是target Q，另一個就是Q value，用TD learning的方式來更新，把兩個相減計算MSE error，就可以得到之間的gradient

- **Explain effects of the discount factor**

主要是來決定說未來時間點的數值對目前的影響有多大，從以下Q-learning的公式中可得知，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。

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

- **Explain benefits of epsilon-greedy in comparison to greedy action selection**

在選擇的時候會有一定機率是隨機選，因為在訓練初期的action效果通常非常的差，使得agent難以進入獲得reward的狀態。例如：火箭剛開始總是選擇向右移，並很快地就結束這個episode，這樣我們就只有向右移的sample，而無法訓練出可獲得更好reward的權重。所以epsilon-greedy可以藉由給定一個 epsilon的機率值，讓agent有機會「隨機地」action以探索新的可能action，而非原本的總是選擇最大利益的action(greedy)。epsilon通常會從1開始，表示從完全隨機的action開始，並在訓練過程中遞減，表示逐漸降低action的隨機性，有越來越高的機率以原先的Q-learning equation來選擇action。

- **Explain the necessity of the target network.**

因為DQN是以neural network來取代table，所以這變成一個預測output Q(s, a)的regression問題，可是因為Q-learning中Q值的遞迴關係，加上Q值一直在變動，而導致模型訓練難以穩定，因此，我們再製作一個target network，作為另一個一段時間才更新一次的Q network，更新時就直接把實際在訓練的Q network權重複製給target network即可，加上target network的做法能使模型訓練更加穩定。

- **Explain the effect of replay buffer size in case of too large or too small**

Replay buffer是儲存以往經驗的地方，會先從舊的經驗開始刪除，buffer size如果太大，會使需要的記憶體空間過大、訓練時間過長且會存有一些很舊的經驗，而且buffer中較近期的資料會被早期的資料稀釋加上早期的經驗太舊了沒有什麼參考價值，較難從新的資料中學習，但如果buffer size過小，則會使隨機採樣後的資料相關性過低，因為buffer中大多是近期剛加入的資料，而導致神經網路的訓練不穩定。

2. Performance (20%)

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

DQN: 255.709

255.70/30 = 8.52

```
Start Testing DQN
total reward 246.67299245789118
total reward 230.74960396176024
total reward 277.66354114256717
total reward 272.1992212376143
total reward 248.6484589520876
total reward 271.8114874952922
total reward 236.76183414615107
total reward 220.49122993887312
total reward 257.1001150843582
total reward 294.996981224951
Average Reward 255.7095465641546
```

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

DDPF:289.84

289.84/30 = 9.66

```
Start Testing DDPG
total reward:  249.53064350586732
total reward:  284.6194996468789
total reward:  284.7317892448615
total reward:  277.2256274582439
total reward:  311.9917798402911
total reward:  272.7406124244343
total reward:  304.85600279250195
total reward:  297.0496652775788
total reward:  315.8809095689063
total reward:  299.8702671578461
Average Reward 289.849679691741
```

3. Report Bonus (25%)

- **Implement and experiment on Double-DQN (10%)**

以 Double-DQN 的概念， 嘗試使用另一個 Q function 來一起計算 Q value，其餘部分與 DQN 相同。

```
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)


    q_value = self._behavior_net(state).gather(1, action.long())

    with torch.no_grad():
        q_next = self._behavior_net(next_state)
        q_target_next = self._target_net(next_state)
        s_value = q_target_next.gather(1,torch.max(q_next,1)[1].unsqueeze(1))
        q_target = reward + gamma * s_value*(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()
```

實驗:


```
total reward:  251.77873056550857
total reward:  282.2328054488355
total reward:  276.93895798205915
total reward:  278.00562387121124
total reward:  309.697521467938
total reward:  274.6963091304573
total reward:  -5.591819139367487
total reward:  268.94285162845205
total reward:  273.91304510932974
total reward:  52.27276292301681
Average Reward 226.28867889874405
```

相較其他兩個方法，Double-DQN 訓練的時候較穩定上升。

- **Implement and experiment on TD3 (Twin-Delayed DDPG) (10%)**
- **Extra hyperparameter tuning, e.g., Population Based Training. (5%)**