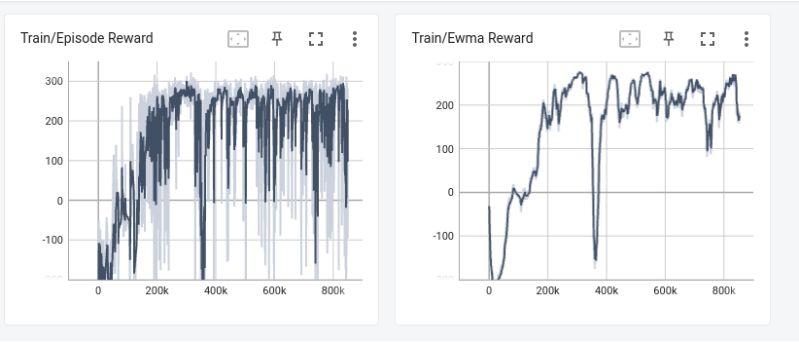
- Experimental Results
  - Screenshot of tensorboard and testing results on LunarLander-v2
    - Testing results

```
Start Testing
Episode: 0      \Length: 173    Total Reward: 273.29
Episode: 1      \Length: 169    Total Reward: 277.98
Episode: 2      \Length: 160    Total Reward: 286.49
Episode: 3      \Length: 177    Total Reward: 272.78
Episode: 4      \Length: 198    Total Reward: 245.70
Episode: 5      \Length: 167    Total Reward: 268.64
Episode: 6      \Length: 209    Total Reward: 285.75
Episode: 7      \Length: 124    Total Reward: -10.65
Episode: 8      \Length: 200    Total Reward: 279.25
Episode: 9      \Length: 214    Total Reward: 283.57
Average Reward 246.28161397437717
```

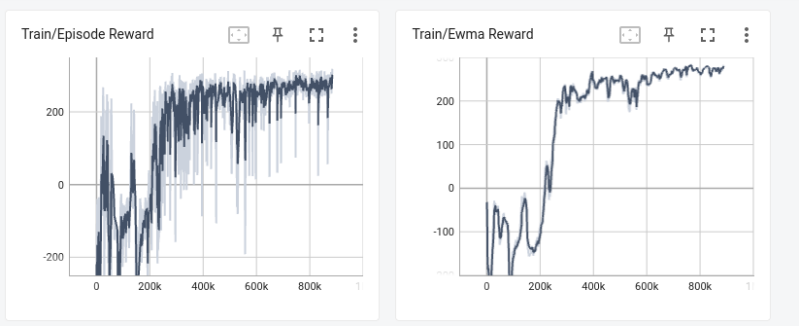
    - Tensorboard



  - Screenshot of tensorboard and testing results on LunarLanderContinuous-v2
    - Testing results

```
Start Testing
Episode: 0      Total reward: 249.87
Episode: 1      Total reward: 290.64
Episode: 2      Total reward: 279.10
Episode: 3      Total reward: 280.89
Episode: 4      Total reward: 310.42
Episode: 5      Total reward: 269.63
Episode: 6      Total reward: 308.79
Episode: 7      Total reward: 299.31
Episode: 8      Total reward: 317.13
Episode: 9      Total reward: 299.69
Average Reward 290.546378734922
```

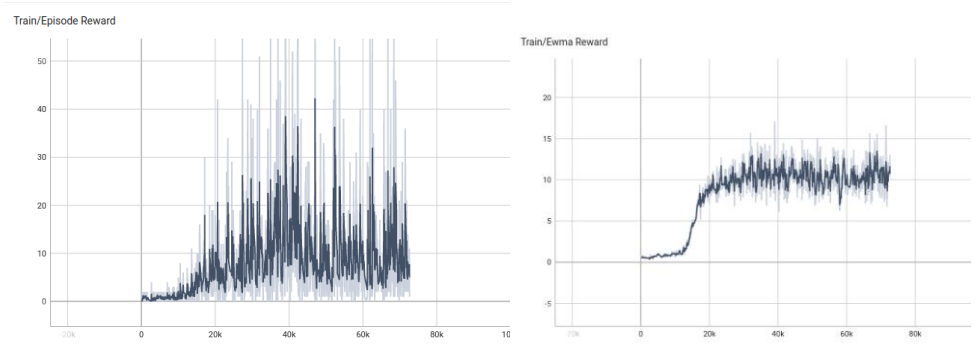
    - Tensorboard



  - Screenshot of tensorboard and testing results on BreakoutNoFrameskip-v4
    - Testing results

```
Start Testing
episode 1: 423.00
episode 2: 435.00
episode 3: 428.00
episode 4: 422.00
episode 5: 833.00
episode 6: 807.00
episode 7: 428.00
episode 8: 413.00
episode 9: 412.00
episode 10: 424.00
```

- Tensorboard



- Questions
  - Describe your major implementation of both DQN and DDPG in detail
    - DQN

    下圖是我實作的 DQN 網路架構，這個網路由 3 層 fully connected layer 組成，且每層的後面都會通過一層 ReLU。這個網路是用來估計 action-value function 的，當 input 的 state(dimension = 8)送進來之後，網路會把 state 轉變成高維(我是用 128)的 hidden representation，最後再 output 出 4 維的代表了 action space 的向量。

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=128):
        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)
        self.act = nn.ReLU()
        # raise NotImplementedError

    def forward(self, x):
        ## TODO ##
        x = self.fc1(x)
        x = self.act(x)
        x = self.fc2(x)
        x = self.act(x)
        out = self.fc3(x)

        return out
```

    DQN 裡面用到的 behavior net 和 target net 都是由上述的網路架構構成。

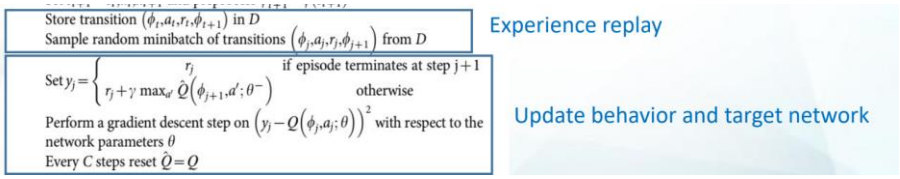
```python
class DQN:
    def __init__(self, args):
        self._behavior_net = Net().to(args.device)
        self._target_net = Net().to(args.device)
```

下圖則是在每個 state 要決定下一個 action 時的實作方法，使用的是 epsilon-greedy 的 action selection。每次選下一個 action 的時候，有 epsilon 的機率會隨機的從 action space 中挑選要做的 action，這樣子可以幫助 action 去探索一些不同的選擇，幫助遇到還沒看到過的 state。剩下 1 - epsilon 的機率則會選目前 behavior net 預測的最好的 action。

```
82      def select_action(self, state, epsilon, action_space):
83          '''epsilon-greedy based on behavior network'''
84          ## TODO ##
85          if random.random() <= epsilon:
86              ## Exploration
87              action = action_space.sample()
88          else:
89              ## Exploitation
90              with torch.no_grad():
91                  input_state = torch.from_numpy(state).to(self.device)
92                  actions_vec = self._behavior_net(input_state)
93                  action = torch.argmax(actions_vec).item()
94          return action
```

下圖則是實作了更新 behavior net 及 target net 中參數的方法，更新的方法參考了 spec 中提供的演算法來實作。首先要更新 behavior net，會先從之前已存起來的、已遇到的所有 state transition 中去 sample 出一小坨的 state transition。這些 sample 出的 state transition 會被送進 behavior net 和 target net 中，算出目前這個 state 預測的 q value(由 behavior net 預測)，還有認為是正確的 q value(由 target net 預測下一個 state 的 q value 乘上 discount factor gamma 後再加上 transition 中的 reward)。會計算預測的和被認為是正確的 q value 的 mean square error 作為 loss，然後再利用 gradient descent 的方法去更新 behavior net 中的參數。更新 target net 的參數的方法 就是每隔一段時間去 copy behavior net 中的參數值。

```
107     def _update_behavior_network(self, gamma):
108         # sample a minibatch of transitions
109         state, action, reward, next_state, done = self._memory.sample(
110             self.batch_size, self.device)
111
112         ## TODO ##
113         q_value = torch.gather(self._behavior_net(state), dim=1, index=action.long())
114         with torch.no_grad():
115             q_next = torch.max(self._target_net(next_state), dim=1)[0].unsqueeze(dim=1)
116             q_target = reward + gamma * q_next * (1 - done)
117         criterion = nn.MSELoss()
118         loss = criterion(q_value, q_target)
119
120         # raise NotImplementedError
121         # optimize
122         self._optimizer.zero_grad()
123         loss.backward()
124         nn.utils.clip_grad_norm_(self._behavior_net.parameters(), 5)
125         self._optimizer.step()
126
127     def _update_target_network(self):
128         '''update target network by copying from behavior network'''
129         ## TODO ##
130         self._target_net.load_state_dict(self._behavior_net.state_dict())
131         # raise NotImplementedError
132
```



- DDPG

我的 actor net 及 critic net 的架構如下圖所示。actor net 的架構由 3 層 fully connected layer 組成，除了最後一層的後面是通過 tanh 以外，每層的後面都會通過一層 ReLU。critic net 的架構也是由 3 層 fully connected layer 組成，不過只有前兩層的後面會通過一層 ReLU。當 input 的 state(dimension = 8)送首先送到 actor net 之後，網路會從 action space 中預測出一個要執行的 action(DDPG 的 action space 是連續的，和 action space 是離散的 DQN 不同)。由 actor net 預測出的 action 接著會和目前的 state 一起當作 input 送進 critic net 中，critic net 會去估計目前 state + action 的 action-value function 的值。

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__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.act_relu = nn.ReLU()
        self.act_tanh = nn.Tanh()
        # raise NotImplementedError

    def forward(self, x):
        ## TODO ##

        x = self.fc1(x)
        x = self.act_relu(x)
        x = self.fc2(x)
        x = self.act_relu(x)
        x = self.fc3(x)
        out = self.act_tanh(x)

        return out
```

```python
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, 1),
        )

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

選擇下一個要執行的 action 如上述，是由 actor net output 出的結果決定，不過為了讓 agent 可以去探索更多的選擇，actor net output 出的結果會視情況加上一個 gaussian noise。

```python
    def select_action(self, state, noise=True):
        '''based on the behavior (actor) network and exploration noise'''
        ## TODO ##
        if noise:
            gaussian_noise = self._action_noise.sample()
        else:
            gaussian_noise = np.zeros(self._action_noise.sample().shape)
        gaussian_noise = torch.from_numpy(gaussian_noise).to(self.device)

        with torch.no_grad():
            input_state = torch.from_numpy(state).to(self.device)
            action = self._actor_net(input_state) + gaussian_noise

        return action.cpu().numpy()
```

首先 update critic 的參數值的做法和 DQN 很類似，會透過從之前已存起來的、已遇到的所有 state transition 中去 sample 出一小坨的 state transition，然後用這個裡面的 state 放到 actor net，然後 actor net

output 再丟到 critic net 去 output 出預測的 Q value。然後用 target critic net 預測的下一個 state 的 Q value 乘上 discount factor gamma 後加上 reward 作為認為的真正的 Q value。這兩個 Q value 一樣計算 mean square error，然後用 gradient descent 的方法去更新 critic net 的參數。Actor net 的更新則是透過由 critic net 建議的方向來更新，因為 actor net 的目標是選擇能 maximize critic net 的 q value 值的 action。因此，再計算 actor net 的 loss 時，可以先透過 actor net 預測出這些 state transition 目前 state 要做的 action 後，透過用 critic net 去計算 value 值後取平均作為 loss(在加負號就可以變 minimize)。然後一樣使用 gradient descent 的方法去更新 actor net 的參數。

```python
def _update_behavior_network(self, gamma):
    actor_net, critic_net, target_actor_net, target_critic_net = self._actor_net, self._critic_net, self._target_actor_net, self._target_critic_net
    actor_opt, critic_opt = self._actor_opt, self._critic_opt

    # 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 = 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 + 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_loss = ?
    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()
```

更新 target network 的參數做法和 DQN 直接整個替換參數的做法不同，DDPG 有一個 hyperparameter tau 決定新參數值是由多少比例的原本參數值和要更新的參數值混合而成。

```python
def _update_target_network(target_net, net, tau):
    '''update target network by _soft_ copying from behavior network'''
    for target, behavior in zip(target_net.parameters(), net.parameters()):
        ## TODO ##
        with torch.no_grad():
            target.copy_(tau * behavior + (1 - tau) * target)
        # raise NotImplementedError
```

- Explain effects of the discount factor

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots$$
$$= R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \cdots)$$

上圖的 gamma 就是 discount factor (< 1)。discount factor 決定了 value 的計算中，有多重視未來的 reward。小的 discount factor 會讓 agent 更 focus on 當下立即能得到的 reward；大的 discount factor 則會讓 agent 更考慮 long-tern 能得到的 reward。

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

比起只會一直選擇最好 action 的 greedy action selection，epsilon-greedy 可以平衡 RL 中重要的 exploration(探索新的、可能更好的未知選擇)和 exploitation(選擇已知的好選擇)。每次，agent 都有 epsilon 的機率去探索新的、可能更好的未知選擇，同時也有 1-epsilon 的機率選擇已知的好選擇，這樣他就不會因為一味選擇已做過的好選擇而錯失了可能更好的未知選擇。

• Explain the necessity of the target network

在訓練 DQN 和 DDPG 的時候，state 的 Q value 是由神經網路去預測的。在這樣的情況下，只要每次更新了神經網路的參數後，就很容易讓每次就算是同樣的 input state 和 action 都預測出不同的 q value，讓 agent 的決策過程變得很不穩定。所以加上一個較長期定期更新的較穩定的 target net 可以幫助減少這個問題。

• Describe the tricks you used in Breakout and their effects, and how they differ from those used in LunarLander.

和 LunarLander 每個 state 就是當下那個時刻的 observation 不同，在 Breakout 的時候，每個 state 是使用現在這個時刻的 observation + 前三個時刻的 observation = 總共 4 張圖集結起來作為一個 state。因為 agent 要學習接球來打掉磚塊，所以 agent 可能會要學習預測球往哪個方向移動，把四個 frame 集結起來作為 state 的做法，可以幫助 agent 看出球是要往哪個方向移動。

Reference:

https://medium.com/%E9%9B%9E%E9%9B%9E%E8%88%87%E5%85%94%E5%85%94%E7%9A%84%E5%B7%A5%E7%A8%8B%E4%B8%96%E7%95%8C/%E6%A9%9F%E5%99%A8%E5%AD%B8%E7%BF%92-ml-note-reinforcement-learning-%E5%BC%B7%E5%8C%96%E5%AD%B8%E7%BF%92-dqn-%E5%AF%A6%E4%BD%9Catari-game-7f9185f833b0

https://www.cupoy.com/qa/club/ai_tw/0000016D6BA22D97000000016375706F795F72656C656173654B5741535354434C5542/0000017C0D81C45D0000002B6375706F795F72656C656173655155455354