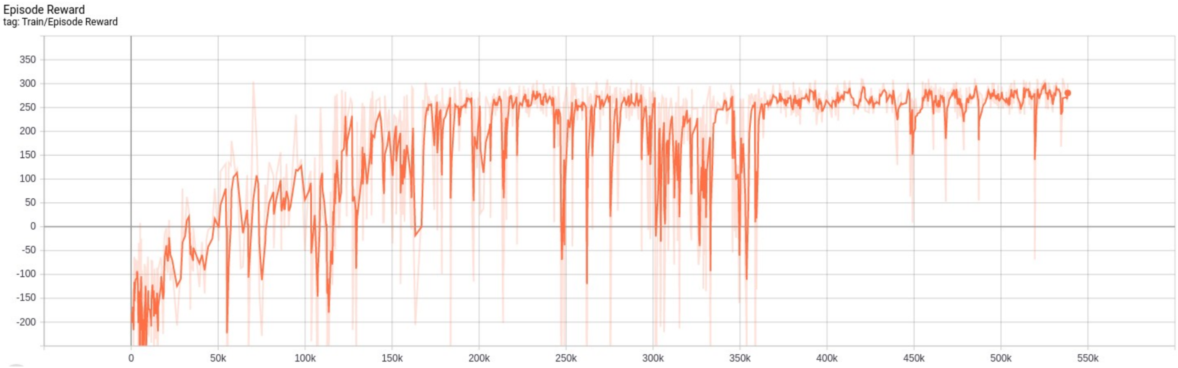
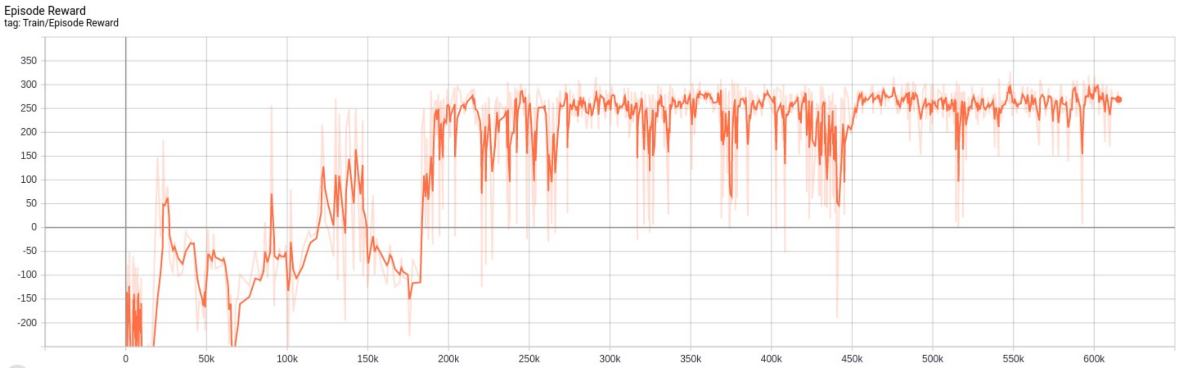
1. A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2



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

  建立一個 NN 來預測 action。我們要處理的環境為 LunarLander-v2，因此，NN 的輸入是 8-dimension observation，輸出是 4-dimension action。

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, 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.relu=nn.ReLU()

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

利用 deque 類別建立一個 buffer(經驗緩衝區)。每次在環境中執行一個 step，會將取得之 transition 加到 buffer 中。訓練時，會隨機從 buffer 中取樣 batch，這個技巧可以打破環境中連續 step 間的相關性，讓訓練資料更 independent and identically。

```python
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'''
        transitions = random.sample(self.buffer, batch_size)
        return (torch.tensor(x, dtype=torch.float, device=device)
                for x in zip(*transitions))
```

選擇行動時，有 epsilon 的機率隨機探索（隨機從 action space 中取樣），否則就是以過去的 model 來選取所有可能 action 中，Q 值最好的最佳 action。

```python
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():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return self._behavior_net(torch.from_numpy(state).view(1,-1).to(self.device)).max(dim=1)[1].item()
```

訓練過程中，有兩個網路需要更新，我們正在使用的 behavior_network 和 target_network。

更新時先從 buffer 中取得 batch 資料。接著將 state 傳給 behavior_network，來得到選擇 action 時用的特定 q_value。然後將 next_state 傳給 target_network，計算相同 action 維度（dim=1）中的最大 Q 值 q_next，並計算 Bellman 近似值 q_target，最後計算 Loss 並做反向傳播來更新 behavior_network。

會以固定的頻率將 behavior_network 同步給 target_network，這個技巧可以讓訓練較穩定。

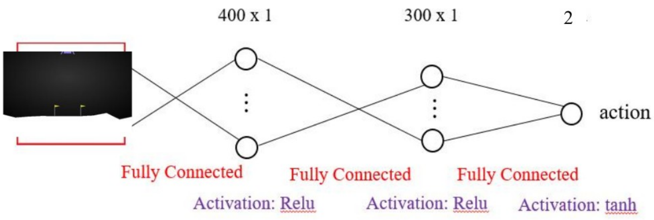
訓練過程中就是讓 agent 和環境互動，從每個 step 取得 trasition 並放進 buffer，並以特定頻率 update。直到 episode 結束計算 reward。

```python
def update(self, total_steps):
    if total_steps % self.freq == 0:
        self._update_behavior_network(self.gamma)
    if total_steps % self.target_freq == 0:
        self._update_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 = 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)

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

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())
```

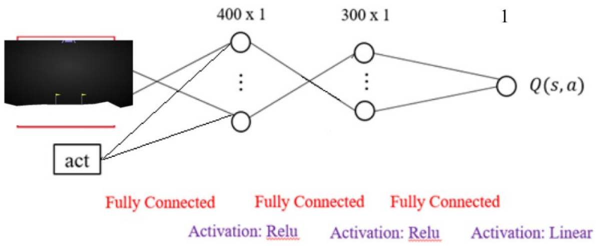
```python
def train(args, env, agent, writer):
    print('Start Training')
    action_space = env.action_space
    total_steps, epsilon = 0, 1.
    ewma_reward = 0
    for episode in range(args.episode):
        total_reward = 0
        state = env.reset()
        epsilon = max(epsilon * args.eps_decay, args.eps_min)
        for t in itertools.count(start=1):  # play an episode
            # select action
            if total_steps < args.warmup:
                action = action_space.sample()
            else:
                action = agent.select_action(state, epsilon, action_space)
            # execute action
            next_state, reward, done, _ = env.step(action)

            # store transition
            agent.append(state, action, reward, next_state, done)

            # update
            if total_steps >= args.warmup:
                agent.update(total_steps)

            state = next_state
            total_reward += reward
            total_steps += 1
            if done:
                ewma_reward = 0.05 * total_reward + (1 - 0.05) * ewma_reward
                writer.add_scalar('Train/Episode Reward', total_reward, total_steps)
                writer.add_scalar('Train/Ewma Reward', ewma_reward, total_steps)
                print(f'Step: {total_steps}\tEpisode: {episode}\tLength: {t:3d}\tTotal
                break
    env.close()
```

- DDPG
  DDPG 模型由兩個獨立的網路組成，Actor 和 Critic，其架構和程式實做如圖。Critic 有兩個獨立的輸入 observation 和 action，會 concatenate 在一起，最後只有一個輸出。

  - Actor



  - Critic




```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.relu=nn.ReLU()
        self.tanh=nn.Tanh()

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


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)
```

在選擇 action 時，在 actor 回傳 action 給環境之前，先將雜訊加到 action 中，藉以達成探索的目的。

```python
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))+\
                    torch.from_numpy(self._action_noise.sample()).view(1,-1).to(self.device)
        else:
            re = self._actor_net(torch.from_numpy(state).view(1,-1).to(self.device))
    return re.cpu().numpy().squeeze()
```

```python
class GaussianNoise:
    def __init__(self, dim, mu=None, std=None):
        self.mu = mu if mu else np.zeros(dim)
        self.std = std if std else np.ones(dim) * .1

    def sample(self):
        return np.random.normal(self.mu, self.std)
```

Critic 更新與 DQN 類似。
Actor 更新的部份，我們將 Actor 的輸出傳給 Critic，然後將 Critic 的負輸出當成 Loss，反向傳播完成網路的更新。

```python
def _update_behavior_network(self, gamma):
    actor_net, critic_net, target_actor_net, target_critic_net = self._actor
    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 = 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)

    # bp
    actor_net.zero_grad()
    critic_net.zero_grad()
    critic_loss.backward()
    critic_opt.step()

    ## 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()
```

另外，DDPG 中 target_network 和 buffer 的使用技巧也和 DQN 類似。

4. Describe differences between your implementation and algorithms.
每個 episode 的前幾個 step，只會隨機探索（隨機從 action space 中取樣）並將 transition 存進 buffer，不會使用到 network。設計原理類似 epsilon-greedy ，先增加 buffer 中更完整的 transition 資訊。

5. Describe your implementation and the gradient of actor updating.

Update the actor policy using the sampled gradient:

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

將 Actor 的輸出傳給 Critic，然後將 Critic 的負輸出當成 Loss，反向傳播完成網路的更新。目的是最小化 Critic 回傳的負值。
程式碼請見第 3 題。

6. Describe your implementation and the gradient of critic updating.

Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i\left(y_i - Q(s_i,a_i|\theta^Q)\right)^2$

利用 target_network 得到的 Q(s’,a’)，和 behavior_network 得到的 Q(s,a)，計算 root-mean-square 誤差作為 loss，然後用反向傳播來更新網路。
程式碼請見第 3 題。

7. Explain effects of the discount factor.
在做 Bellman 近似值更新的時候，下一個 step 的值會乘上 gamma，0<gamma<1。意思是即時的 reward 影響最大，而其他長期 reward 的影響會隨持間而打折扣。

$$\text{Set } y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$$

$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

8. Explain benefits of epsilon-greedy in comparison to greedy action selection.
在一開始訓練時，Q 可能還不是很好，造成 agent 在某些 state 下，可能一直被困在錯誤行動中而不去嘗試新的 action。這時需要多一些隨機探索，建立更完整的 transition 資訊，讓他有機會了解哪些 action 會得到什麼。隨著不斷訓練，隨機行動就會變得很沒效率，不應該浪費時間反覆嘗試已經試過且知道結果的 action。需要改成使用 Q 來決定 action。epsilon-greedy 就是這兩種情形的混合解法。一開始 epsilon 很大，在逐漸用 eps_decay 調降。

9. Explain the necessity of the target network.
使用 Bellman 做 Q 值近似時，它以 Q(s’,a’)提供我們 Q(s,a)，然而因為數據非完全 independent and identically，當我們更新網路參數時，也會間接改變了 Q(s’,a’)，讓訓練變得很不穩定。target_network 是一種訓練技巧，建立一個網路的副本專門用來計算 Q(s’,a’)，再定期和主網路 behavior_network 做同步。

10. Explain the effect of replay buffer size in case of too large or too small.
會影響訓練速度。如果 buffer 太大，訓練會更穩定，但太多不新鮮的資料，需要更長時間才能收斂。但若 buffer 太小，裡面的資料都會非常接近，非常不獨立，訓練結果容易 overfit 。

11. Double DQN
在 Deep Reinforcement Learning with Double Q-Learning 這篇論文中指出，原本的 DQN 傾向於高估 Q 值。

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

它修改 Bellman 更新如下：

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t') .$$

程式實作上就是修改取得 q_target 的部份。改成用 behavior_network 取得 Q 最大的 action（原本的 DQN 是用 target_network 取得），並採取 action，但是與此 action 相應的 q_target 卻是來自 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 = 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()
```

12. Result

- DQN

```
Start Testing
total reward: 252.36
total reward: 288.51
total reward: 280.78
total reward: 281.46
total reward: 305.52
total reward: 270.42
total reward: 310.58
total reward: 298.61
total reward: 319.59
total reward: -42.81
Average Reward 256.50204080522815
```

- DDPG

```
Start Testing
total reward: 250.87
total reward: 278.69
total reward: 272.41
total reward: 275.83
total reward: 286.07
total reward: 251.83
total reward: 281.49
total reward: 291.15
total reward: 303.39
total reward: 257.09
Average Reward 274.88237229918013
```

- Double DQN

```
Start Testing
total reward: 243.83
total reward: 280.23
total reward: 273.66
total reward: 263.54
total reward: 223.24
total reward: 266.06
total reward: 280.52
total reward: 274.09
total reward: 301.54
total reward: 299.87
Average Reward 270.6587131123694
```