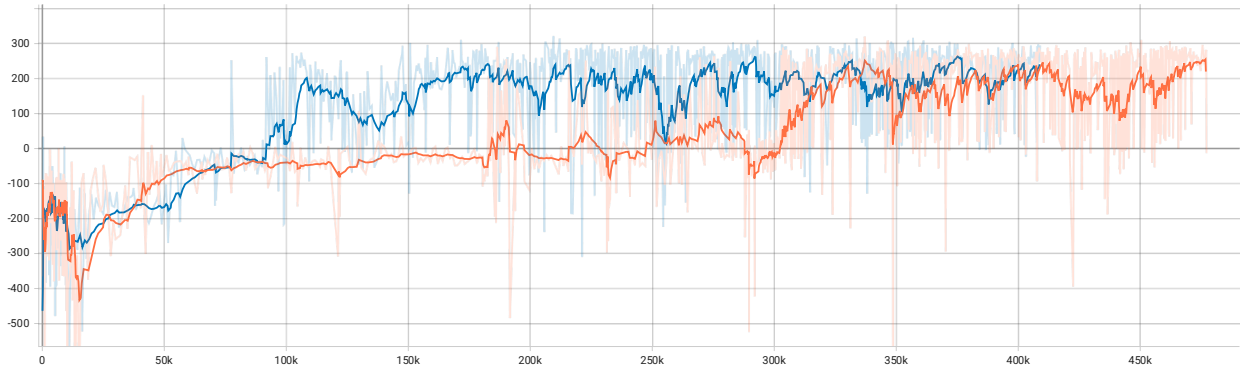
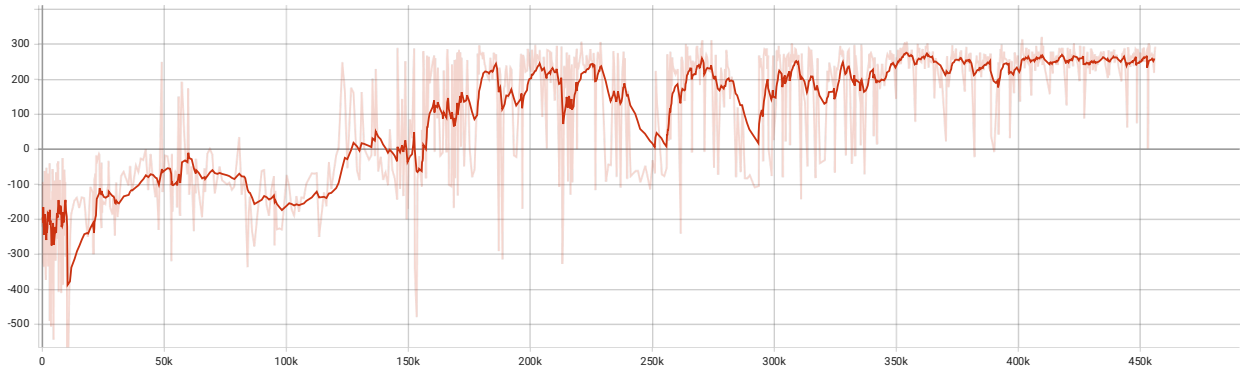
- Report:
- A tensorboard plot show episode rewards of at least 800 training episodes in LunarLander-v2:



- A tensorboard plot show episode rewards of at least 800 training episodes in LunarLanderContinuous-v2:



- Describe your major implementation of both algorithms in detail:

DQN:

```
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 = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        return x
```

在 LunarLander-v2 中，env 會給出 8 個 observation，例如(Horizonal Coordinate , Vertical Coordinate, Horizontal Speed 等等)，並不是給整個 engine

的畫面，所以我們可以用 fully-connect 來處理，而 Lander 會有 4 個動作，先經過 hidden layer，最後再給 4 dim 作為輸出。

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    with torch.no_grad():
        if random.random() > epsilon:
            state = torch.Tensor([state]).to(self.device)
            return self._behavior_net(state).max(1)[1].item()
        else:
            return action_space.sample()
```

在選擇 action 的部分，這邊使用 epsilon greedy 的策略，避免陷入 local 的最佳解，先 random 出一個 0~1 的數字，如果大於就使用 behavior_net 所給出的 action，反之則 random 給一個動作出來。

```
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.type(torch.long))
    with torch.no_grad():
        if self.ddqn:
            max_act = self._behavior_net(next_state).max(1)[1].view(-1, 1)
            q_next = self._target_net(next_state)
            q_next = q_next.gather(1, max_act.type(torch.long))
        else:
            q_next = self._target_net(next_state).max(1)[0].view(-1, 1)
        q_target = reward + gamma * q_next * (1 - done)

    criterion = nn.SmoothL1Loss()
    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()
```

在這裡我們要用 q_value 跟 q_target 來計算 loss，這邊 q_value 是給當下這個時態的 state 給 behavior_net 所給出來的值，而 q_target 則是當下這個時態的 state 所做出的 action 所做出來的 rewards，加上 gamma 乘以下一個時態的 state：next_state，預測下一個時態 target_net 所遞迴推下去的直。利用兩者之差來計算 loss。

```
    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())
```

整個 DQN 有兩個網路一個是現在這一時態的網路和下一個時態延遲的網路，而這邊是單純是時態往後移，把在時間 t 的 behavoir_net 交給時間 t+1 的 target，就等於單純平移過去。

```
def test(args, env, agent, writer):
    print('Start Testing')
    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()
        for t in itertools.count(start=1):
            if args.render:
                env.render()
            action = agent.select_action(state, epsilon, action_space)
            next_state, reward, done, info = env.step(action)

            state = next_state
            total_reward += reward
            if done:
                writer.add_scalar("Test/Episode Reward", total_reward, n_episode)
                print(f"Episode: {n_episode:d} Reward: {total_reward:.4f}")
                rewards.append(total_reward)
                break
```

這邊就是單純的 test，每次給出來的 action 交給 env，直到整個 episode 結束。重複執行。

DDPG:

```
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)

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

DDPG 在網路架構與 DQN 最大的不同就是 DDPG 有 ActorNet 和 CriticNet，而 ActorNet 的部分則與 DQN 一樣，相較於離散的問題，連續的網路 hidden_dim 比較大。

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        state = torch.tensor(state).to(self.device)
        action = self._actor_net(state)
        if noise:
            action +=
                torch.tensor(self._action_noise.sample()).to(self.device)
        return torch.clamp(action,-1,1).detach().cpu().numpy()
```

在這邊 select_action 的時候，除了把當下的 state 交給網路外，在訓練的時候在給出的 action vector 加上 noise 可以幫助 network 更好的探索，否則可能會陷入網路自身的 loacl 最佳解。在 test 時則不需要加上 noise

```
q_value = critic_net(state,action)
with torch.no_grad():
    a_next = target_actor_net(state)
    q_next = target_critic_net(next_state,a_next)
    q_target = reward + gamma * q_next * (1 - done)

criterion = nn.SmoothL1Loss()
critic_loss = criterion(q_value, q_target)
```

在 DDPG 中，q_value 的值是要由 critic_net 來做評估的，action_next 則如 DQN 一樣是延遲的 target_actor_net 給予 state 所出來的，action_next 和下一個時態的 state 再交給延遲的 target_critic_net 所計算出，最後在用出來的 q_next 用 Q learning 的公式和 DQN 一樣算出 q_target，再用兩者算出 loss。

```
action = actor_net(state)
actor_loss = -torch.mean(critic_net(state, action))
```

Action 的部分則是單純的交給 criticNet 做評斷，將出來的數值做平均。

```
def test(args, env, agent, writer):
    print('Start Testing')
    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()
        for t in itertools.count(start=1):
            if args.render:
                env.render()
            action = agent.select_action(state, noise=False)
            next_state, reward, done, info = env.step(action)

            state = next_state
            total_reward += reward
            if done:
                writer.add_scalar("Test/Episode Reward", total_reward, n_episode)
                print(f"Episode: {n_episode:d} Reward: {total_reward:.4f}")
                rewards.append(total_reward)
                break
```

Test 的部分上基本上跟 DQN 一樣，唯一的不同是，select_action 有 noise 的選項，在測試時需要關掉。

```
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())
```

與 DQN 完全一致，將當前的網路傳給 target(延遲)網路

- Describe differences between your implementation and algorithms:

DQN:

在 DQN 的演算法是在每一個時態都更新兩個不同的網路，但其實可以利用 freq 參數來改變不同的更新頻率。另外在一開始 model 什麼都不會和不清楚 state、action、rewards 的關係時可以利用 warmup 的參數，產生隨機的動作供 model 做參考，這也是為什麼在訓練時，前面幾個 episode 會特別快的原因。

DDPG:

給予 actor_loss 負的 critic 所給出得值可以令 actor 做小化 loss 就如同 actor 需要去學習如何才是最好的 action。

- Describe your implementation and the gradient of actor updating:

```
action = actor_net(state)
actor_loss = -torch.mean(critic_net(state, action))
```

如先前所述，actor_loss 是由 critic_net 做　　評斷，就如同 GAN 中的 discriminator，為 genernator 做評斷一樣，actor 和 critic 的關係就如同一個時根據經驗做出回饋而一個旁觀者依照當前的 state 和做出的 action 所得到的 reward 做出評價，這就是 critic 所做的事情。

- Describe your implementation and the gradient of critic updating:

```
q_value = critic_net(state,action)
with torch.no_grad():
    a_next = target_actor_net(state)
    q_next = target_critic_net(next_state,a_next)
    q_target = reward + gamma * q_next * (1 - done)

criterion = nn.SmoothL1Loss()
critic_loss = criterion(q_value, q_target)
```

如同上所述，critic_loss 的計算就如同傳統 Q-learning 和 DQN 一樣，需要算出 q_target，q_target 是下一個時態所預估出來的值，我們想要逼近的，要利用 q_next 乘上 discount factor 加上這次 action 應該要得到的 reward，而 q_next 則是需要下一個時態的 state 和 action 所利用 target_critic_net 所算出，利用還尚未更新的延遲網路做評估。當中所需的 action_next 是利用 target_actor_net 所算出，一樣也是延遲網路。最後再利用兩個不同時態的 q_value 做 loss 的計算。

- Explain effects of the discount factor:

可以給不同階段的 rewards 不同的權重，換句話說給越後面的權重越低，較注重越較近的狀態。

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

如果沒有使用 epsilon-greedy 的話，network 會一直給予最 greedy 的 action，這樣子就很容易陷入到一個 local 最佳的誤區，但如果使用 epsilon-greedy 的話就會給予亂走 action 的機會，這樣相較之下就可以嘗試更多不同的動作得到不同的反饋，多多探索可能整體會比較好。

- Explain the necessity of the target network:

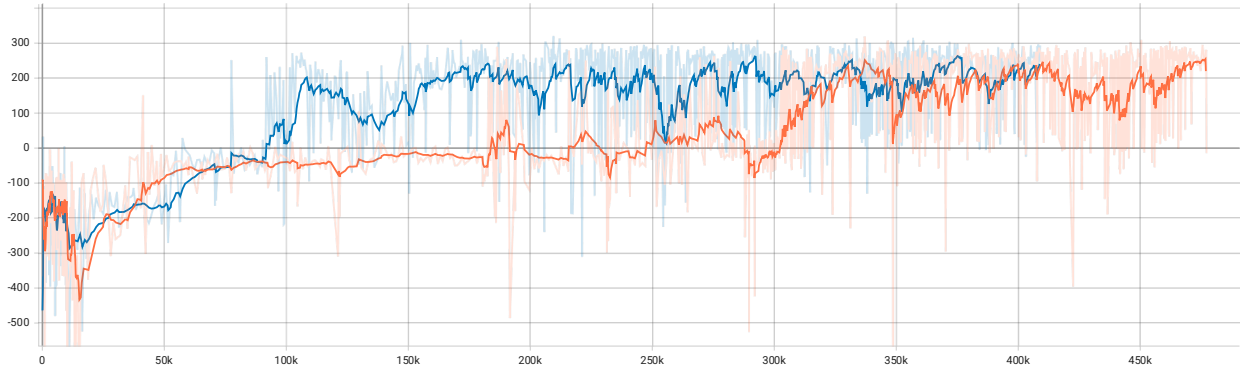
如果只有單一網路，用一個網路的參數做預測及更新權重，會造成以下問題，如果用一個網路做預測，在當你更新參數的時候，你更新之後的網路預測會做改動，這樣下來，整體就不會穩定，數值會一直變動。這時候如果有一個網路做延遲一個時態的話，相較之下就會穩定很多。

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

Buffer size 會影響模型的訓練速度和結果的成效，buffer 太大的話抽出來的 memory 可能會與當下的 state 不大相關，而且需要久久一次才能使用到離現在較近的記憶。Buffer 太小則記憶的目的就不顯著，且可能會造成 overfitting。

- Report Bonus:
- Implement and experiment on Double-DQN:

在原生的 DQN 的 training 時的 reward 經常會 overestimate，有點高估所得到的 reward 的現象，如下圖，藍色的部分是 DQN、紅色則是 DDQN，在前三分之二的地方可以觀察到，藍色是明顯高於紅色的，原因在於當在算 q_value 時是使用舊的參數做運算，舊的參數尚未更新至新的時態，所以會比正確的 q_value 有高估的情況。造成整體結果的過於樂觀。



而 DDQN 則是用來解決這 overestimate 的問題，想法就是利用 DQN 原本就現有的 behavior_net 和 target_net 做雙層的 prediction，原本 q_next 只需要一個 target_net 做決策，DDQN 的 q_next 則需要先求用當前時態的 behavior_net 決定出 action 再給 q_next 共同做出參考的依據。

- Extra hyperparameter tuning, e.g., Population Based Training:

沒做。

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

```
Start Testing
Episode: 0 Reward: 258.8433
Episode: 1 Reward: 288.8952
Episode: 2 Reward: 232.2303
Episode: 3 Reward: 274.6371
Episode: 4 Reward: 214.5564
Episode: 5 Reward: 244.7160
Episode: 6 Reward: 284.9407
Episode: 7 Reward: 302.3238
Episode: 8 Reward: 171.8627
Episode: 9 Reward: 226.0598
Average Reward 249.90653072303297
```

249.9065/30=8.330

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

```
Start Testing
Episode: 0 Reward: 252.5789
Episode: 1 Reward: 276.3934
Episode: 2 Reward: 279.0972
Episode: 3 Reward: 272.5573
Episode: 4 Reward: 294.8504
Episode: 5 Reward: 270.8280
Episode: 6 Reward: 298.0273
Episode: 7 Reward: 285.6820
Episode: 8 Reward: 309.8313
Episode: 9 Reward: 258.1826
Average Reward 279.80283351630 87
```

279.8028/30=9.326