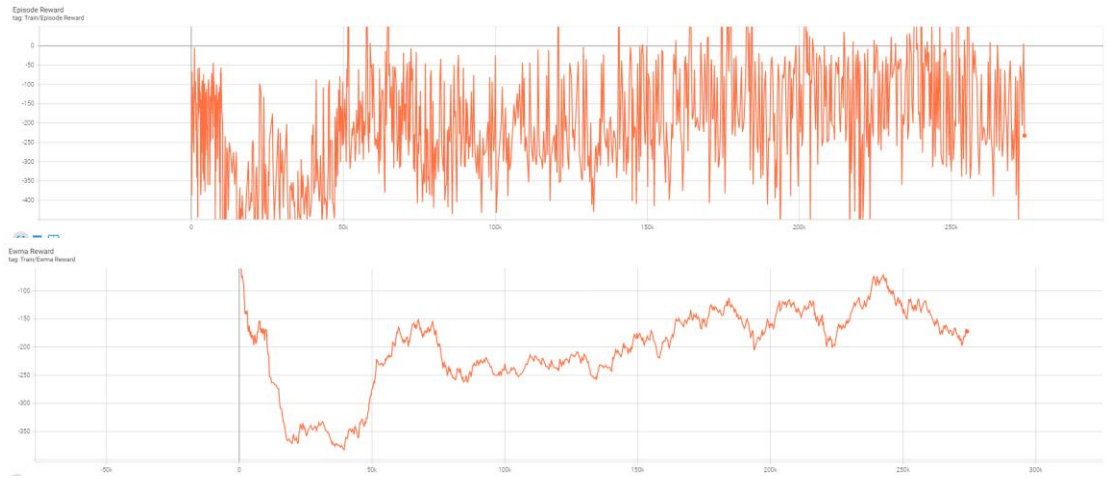
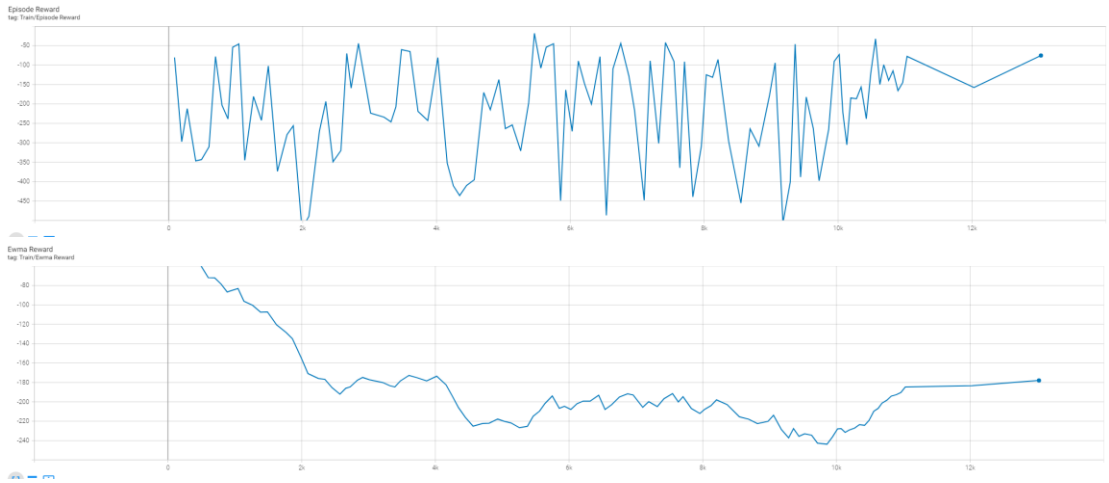
Report:

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



- **A tensorboard plot shows 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-v ##
        self.sequence = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(inplace=True),

            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(inplace=True),

            nn.Linear(hidden_dim, action_dim),
        )

    def forward(self, x):
        ## TODO-v ##
        x = torch.tensor(x, device="cuda")
        y = self.sequence(x)
        return y
```

For the neural network in DQN, I follow the architecture in the spec. Using 2 fully-connected layers with ReLU activation function as hidden layer.

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

I use Adam optimizer as its optimizer.

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO-v ##
    if random.random() < epsilon:
        return random.randint(0, 3)
    else:
        v = self._behavior_net(state)
        _, action = torch.max(v, 0)
        return action.item()
```

In the select_action function, I apply the epsilon greedy algorithm. Choose a random action with probability ε. Otherwise choose the best action that has the max Q-value in next state ($argmax_a Q(S, a)$).

```python
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-v ##
    q_value = self._behavior_net(state).gather(1, action.long())
    with torch.no_grad():
        q_next = self._target_net(next_state)
        q_next, idx = torch.max(q_next, 1)
        q_next = q_next.unsqueeze(1)
        q_target = reward + gamma * q_next * (1 - done)
    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)
```

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

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

Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Following the algorithm in the spec, sample a random minibatch from the memory first. Then, get the Q-value Q(S_j, a) from the behavior net. And calculate its Q-target by the target net on next state and its reward. After that use MSE as its loss function and do the back propagation on the behavior net.

```python
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO-v ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

In update_target_network function, just directly copy the parameter from behavior net to target net.

```python
for n_episode, seed in enumerate(seeds):
    total_reward = 0
    env.seed(seed)
    state = env.reset()
    with torch.no_grad():
        done = 0
        while not done:
            action = agent.select_action(state, epsilon, action_space)
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward

            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                rewards.append(total_reward)
                break
```

In each episode of the test function, reset the game environment first. And use “with torch.no_grad()” to not store the gradient information during the forward propagation. In each step, use the “select_action” function to get the best action and apply on the game environment to get the “reward”, “next state” and “done”. If the game is over, write the rewards to log file and start the next episode.

DDPG:

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO-v ##
        self.sequence = nn.Sequential(
            nn.Linear(state_dim, hidden_dim[0]),
            nn.ReLU(inplace=True),

            nn.Linear(hidden_dim[0], hidden_dim[1]),
            nn.ReLU(inplace=True),

            nn.Linear(hidden_dim[1], action_dim),
            nn.Tanh()
        )

    def forward(self, x):
        ## TODO-v ##
        x = torch.tensor(x, device="cuda")
        y = self.sequence(x)
        return y
```

For the actor neural network in DDPG, I follow the architecture in the spec. Using 2 fully-connected layers with ReLU activation function as hidden layer. In output layer, use tanh as its activation function.

```
## TODO-v ##
self._actor_opt = optim.Adam(self._actor_net.parameters(), lr=args.lra)
self._critic_opt = optim.Adam(self._critic_net.parameters(), lr=args.lrc)
```

I use Adam optimizer as their optimizer.

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO-v ##
    action = self._actor_net(state).cpu().numpy()
    if noise:
        action = action + self._action_noise.sample()
    return action
```

Get the action from the actor net, and convert it to NumPy array. After that, if it need noise, add a noise value on the action.

```
# sample a minibatch of transitions
state, action, reward, next_state, done = self._memory.sample(
    self.batch_size, self.device)

## update critic ##
# critic loss
## TODO-v ##
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
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()
```

Sample random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from R

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

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

Following the algorithm in the spec, sample a random minibatch from the memory first. Then, get the Q-value $Q(S_i, a_i)$ from the critic net Q. After that, use actor net "μ" to predict the probability of actions of next state and use that action to predict next next_state's Q-value $Q(S_{i+1}, \mu'(S_{i+1}))$. Use this information to calculate the Q-target and use MSE as its loss function and do the back propagation on the critic net.

```
## update actor ##
# actor loss
## TODO-v ##
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()
```

Update the actor policy using the sampled gradient:

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

After updating the critic net Q, we need to update the actor net “μ”. I use actor net to predict the probability of actions of current state and use the mean value of critic net as its loss function to do the back propagation.

```
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-v ##
        target.data.copy_(tau*behavior.data + (1.0-tau)*target.data)
```

Update the target networks:

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

Use soft update to update the target network.

```
for n_episode, seed in enumerate(seeds):
    total_reward = 0
    env.seed(seed)
    state = env.reset()
    with torch.no_grad():
        done = 0
        while not done:
            action = agent.select_action(state, noise=False)
            next_state, reward, done, _ = env.step(action)

            state = next_state
            total_reward += reward

            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                rewards.append(total_reward)
                break
```

In each episode of the test function, reset the game environment first. And use “with torch.no_grad()” to not store the gradient information during the forward propagation. In each step, use the “select_action” function to get the best action(with no noise) and apply on the game environment to get the “reward”, “next state” and “done”. If the game is over, write the rewards to log file and start the next episode.

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

  No difference, I followed the algorithms to implement it.

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

```
# sample a minibatch of transitions
state, action, reward, next_state, done = self._memory.sample(
    self.batch_size, self.device)

## update critic ##
# critic loss
## TODO-v ##
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
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()
```

Sample random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from R

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

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

Following the algorithm in the spec, sample a random minibatch from the memory first. Then, get the Q-value Q(S_i, a_i) from the critic net Q. After that, use actor net "μ" to predict the action of next state and use that action to predict next next_state's Q-value Q(S_{i+1}, μ′(S_{i+1})). Use this information to calculate the Q-target and use MSE as its loss function and do the back propagation on the critic net.

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

```
## update actor ##
# actor loss
## TODO-v ##
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()
```

Update the actor policy using the sampled gradient:

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

After updating the critic net Q, we need to update the actor net "μ". I use actor net to predict the probability of actions of current state and use the mean value of critic net as its loss function to do the back propagation.

- **Explain effects of the discount factor**

The discount factor describes how our model pay attention to future rewards. If the factor is close to 0, then the model would pay more attention on short rewards, otherwise, it would pay more attention on future rewards.

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

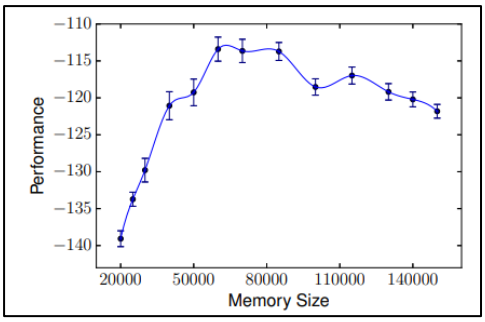
Using epsilon-greedy makes the agent be able to explore some not explored yet states/actions. Otherwise, using greedy action selection may cause the agent always explore certain states/actions. Some best states/actions may be in the not explored states.

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

We can regard the model as a regression problem. Input is state and action. The goal is to fit the network output $Q(s_t, a_t)$ to target $r_t + Q(s_{t+1}, \pi(s_{t+1}))$. Without using the target network, the training target would change after every iteration. This cause the model hard to converge.

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

From the paper "*The Effects of Memory Replay in Reinforcement Learning*", I found that both too large or too small memory would slow down learning.



This is the experiment figure in that paper. It shows that too large or too small memory would cause the performance bad.

Report Bonus

- ## Implement and experiment on Double-DQN:

I followed professor's slides to implementation DDQN. Decouple the selection from the evaluation.

$$Y_t^{Q} = r_{t+1} + \gamma \max_a Q(S_{t+1}, a|\theta^-)$$

$$Y_t^{DoubleQ} = r_{t+1} + \gamma Q\left(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a|\theta)|\theta^-\right)$$

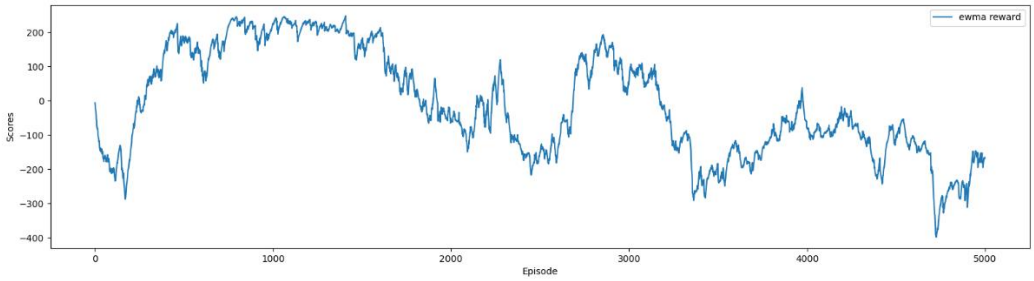
```
## TODO-v ##
q_value = self._behavior_net(state).gather(1, action.long())
with torch.no_grad():
    q_next = self._target_net(next_state)
    q_next, idx = torch.max(q_next, 1)
    q_next = q_next.unsqueeze(1)
    q_target = reward + gamma * q_next * (1 - done)
criterion = nn.MSELoss()
loss = criterion(q_value, q_target)
```

▲ **DQN implementation**

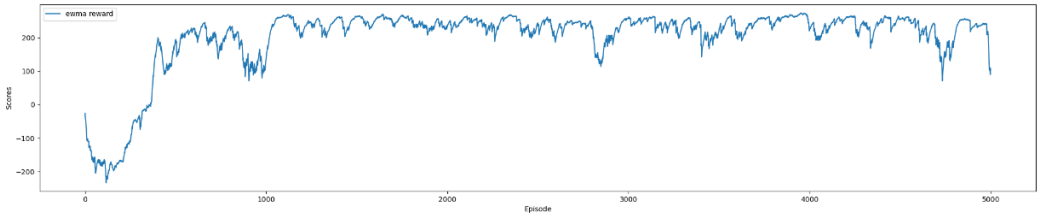
```
## TODO-v ##
q_value = self._behavior_net(state).gather(1, action.long())
with torch.no_grad():
    q_next_value = self._behavior_net(next_state)
    next_action = torch.max(q_next_value, 1)[1].unsqueeze(1)
    q_next = self._target_net(next_state).gather(1, next_action)
    q_target = reward + gamma * q_next * (1 - done)
criterion = nn.MSELoss()
loss = criterion(q_value, q_target)
```

▲ **DDQN implementation**

In the implementation of DDQN, I only modify how it calculate Q-target in "update_behavior_network" function. In DQN, the Q-target is the largest predicted Q-value of next_state's action by target network. In DDQN, we need to calculate the largest Q-value of next_state's action by behavior network. Then, use this action and next_state on target network to get the predicted Q-value as its Q-target.



▲ **Ewma reward on DQN training in 5000 episodes**



▲ **Ewma reward on DDQN training in 5000 episodes**

From the above 2 figures, we can see that both models have similar converge speed at the beginning of training. But DQN's performance would decrease dramatically in about 1500th episodes. In contrast, DDQN's performance is smooth. Even though it would still have some bad play, DDQN would fix that in a small number of episodes. Unlike DQN, DQN would fall into that bad play and follow that bad rule in upcoming episodes.

Performance:

- **[LunarLander-v2]**

```
python ddqn.py --test_only --render
2022-05-22 16:00:25.846648: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dynamic
library cudart64_110.dll
Start Testing
241.79478371058542
281.4197860919139
270.8899996430052
272.85031820825725
236.06789625353437
257.07709266450036
16.63595599581869
298.35352867091297
322.84298766155626
221.91790867978244
Average Reward 241.98502575798676
```

- **[LunarLanderContinuous-v2]**

```
python ddpg.py --test_only --render
2022-05-20 10:38:59.831226: I tensorflow/stream_executor/platform/default/dso_loader.cc:49] Successfully opened dynamic
library cudart64_110.dll
Start Testing
240.67189461137403
282.1262396763293
274.5691052936192
269.37231887993596
300.6717528011284
270.65671800164023
170.24658258694106
296.0608392655791
250.38649883039662
253.8877281077853
Average Reward 260.86496780547293
```