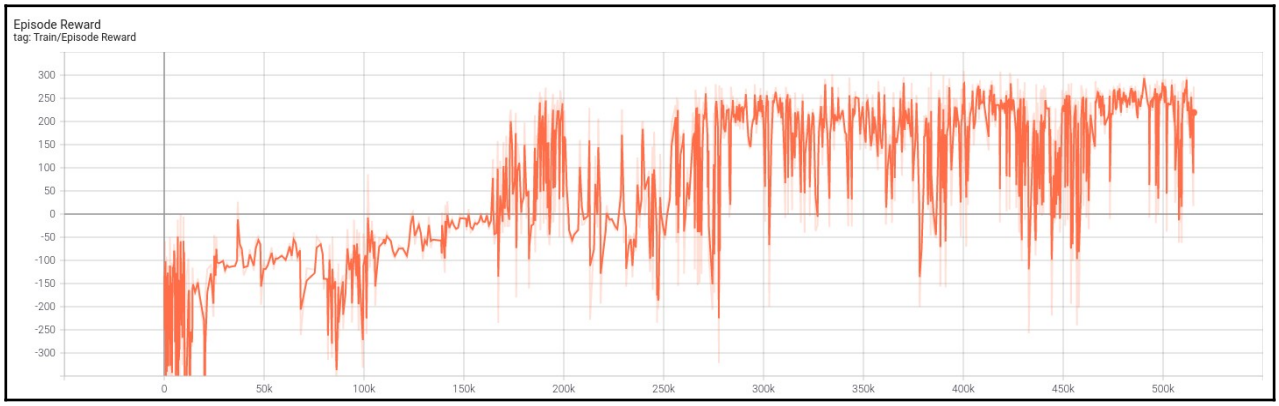
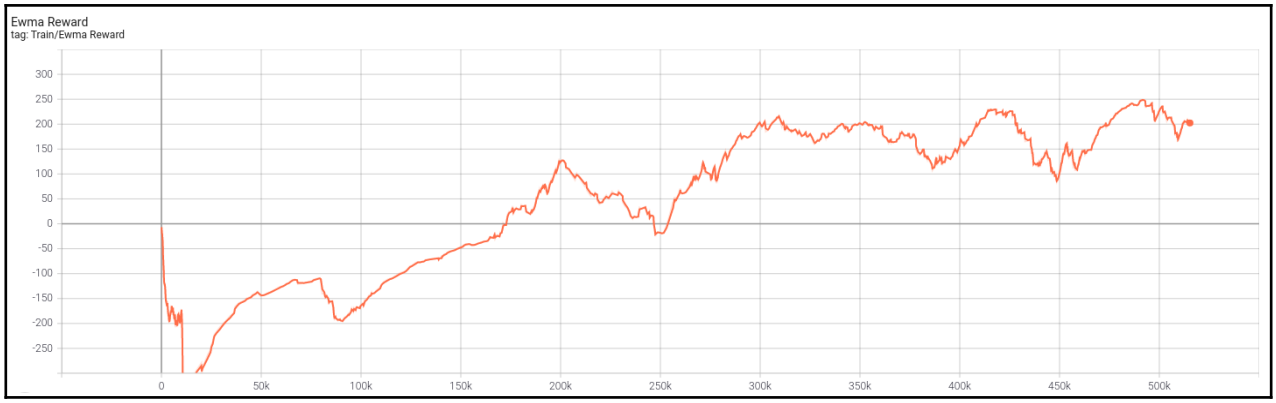
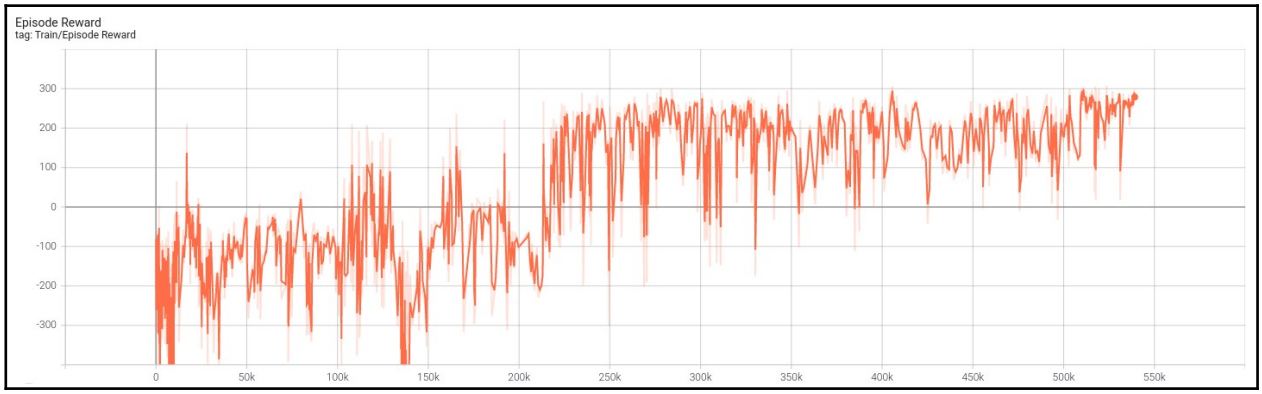
Lab 6 Report
309551064 張凱翔

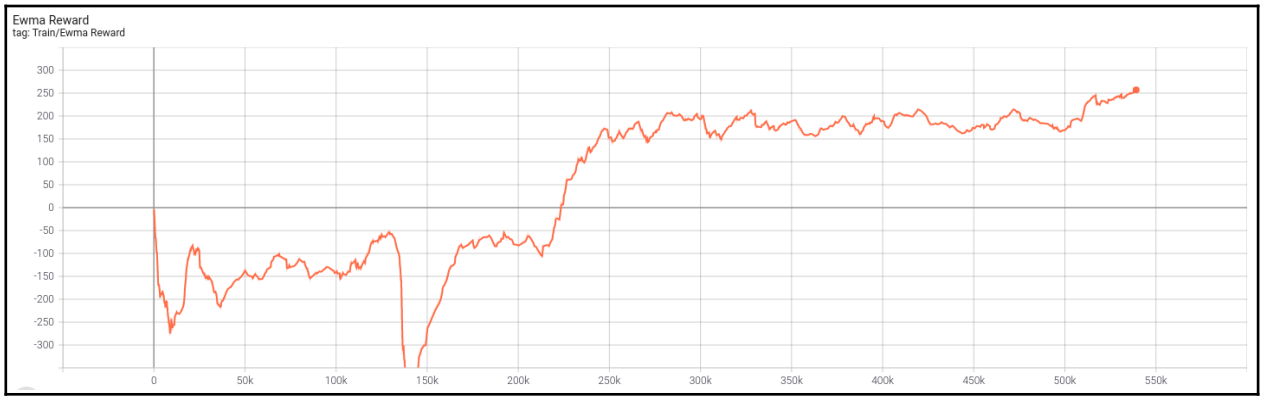
- A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2





- 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:
  - Construct the network like the spec specified which is constructed with three fully connected layers with RuLU function to the output of first two layers and the last layer outputs 4 dimension representing the expected value of four action.

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##
        self.layer1 = nn.Linear(state_dim, hidden_dim)
        self.layer2 = nn.Linear(hidden_dim, hidden_dim)
        self.layer3 = nn.Linear(hidden_dim, action_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        ## TODO ##
        output = self.layer1(x)
        output = self.relu(output)
        output = self.layer2(output)
        output = self.relu(output)
        output = self.layer3(output)
        return output
```

  - Use epsilon-greedy algorithm to select action. If the random value is smaller than epsilon, select a action randomly; otherwise, select the action which has the largest value of the output of behavior net according to current state.

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if random.random() < epsilon:
        return action_space.sample()
    else:
        return int(torch.argmax(self._behavior_net(torch.Tensor([state]).to(self.device))).detach().cpu().numpy())
```

  - Use Adam optimizer to update the behavior net every 4 steps.

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

  - And copy the parameters of behavior net to update target net every 1000 steps. The reason that should implement in this way is discussed below.

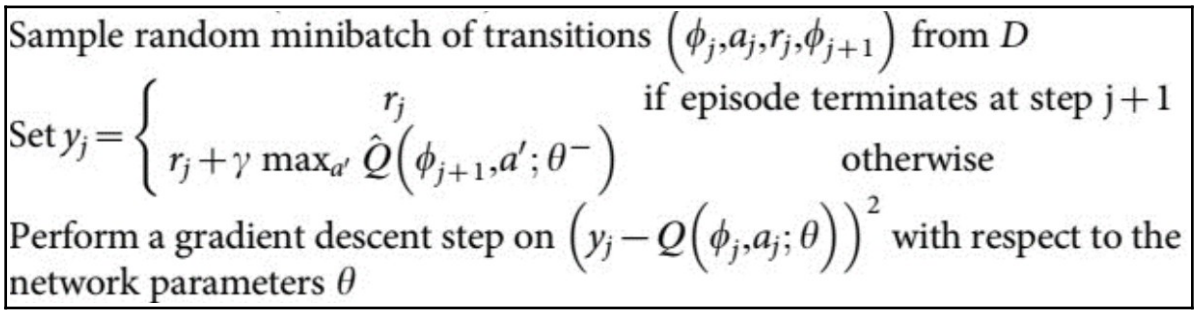
```
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

  - The testing phase is similar with the training phase without warming up and updating network. In addition, should memorize reward and calculate the mean.

```
for n_episode, seed in enumerate(seeds):
    total_reward = 0
    env.seed(seed)
    state = env.reset()
    ## TODO ##
    for t in itertools.count(start=1):
        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)
            break
    rewards.append(total_reward)
```

- First, randomly sample from the replay buffer, then use these data to update network. To update the behavior net, should calculate the Mean Square Error loss of the predicted value and target value. The predicted value is the value of the output of behavior net which index is the same as action and all I do is concatenate these values. And the target value is calculated by reward + r * max_a' Q(s', a') where max_a' Q(s', a') is the max value of the output of target net.

```
## TODO ##
q_value = self._behavior_net(state)
q_value_each_action = q_value[0][int(action[0][0])].view(1, 1)
for i in range(1, action.size(0)):
    q_value_each_action = torch.cat((q_value_each_action, q_value[i][int(action[i][0])].view(1, 1)), 0)

with torch.no_grad():
    q_next = torch.max(self._target_net(next_state), 1).values.unsqueeze(1)
    q_target = reward + gamma * q_next * (1 - done)
criterion = nn.MSELoss()
loss = criterion(q_value_each_action, q_target)
```



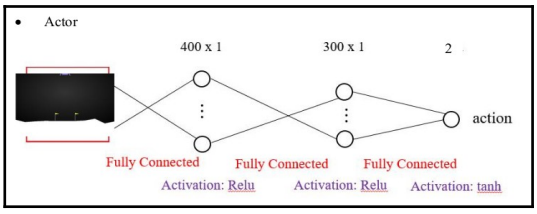
- DDPG:

- randomly sample data of a batch size from the replay buffer

```
def sample(self, batch_size, device):
    '''sample a batch of transition tensors'''
    ## TODO ##
    transitions = random.sample(self.buffer, batch_size)
    return (torch.tensor(x, dtype=torch.float, device=device)
            for x in zip(*transitions))
```

- Construct the network like the spec specified which is constructed with three fully connected layers with RuLU function to the output of first two layers and Tanh function to the output the last layer.

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.layer1 = nn.Linear(state_dim, hidden_dim[0])
        self.layer2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.layer3 = nn.Linear(hidden_dim[1], action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        ## TODO ##
        output = self.layer1(x)
        output = self.relu(output)
        output = self.layer2(output)
        output = self.relu(output)
        output = self.layer3(output)
        output = self.tanh(output)

        return output
```



- Use Adam optimizer to update the actor net and critic net

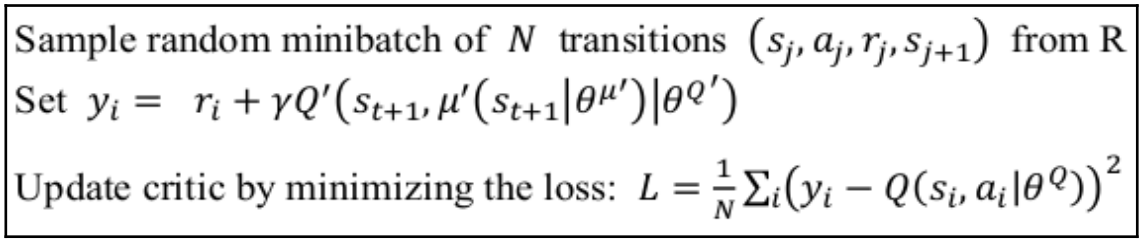
```
## TODO ##
self._actor_opt = torch.optim.Adam(self._actor_net.parameters(), lr=args.lra)
self._critic_opt = torch.optim.Adam(self._critic_net.parameters(), lr=args.lrc)
```

- select action by adding the output of actor net according to current state and some noise as exploration

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    return self._actor_net(torch.Tensor([state]).to(self.device)).detach().cpu().numpy()[0] \
        + int(noise) * self._action_noise.sample()
```

- First, randomly sample from the replay buffer, then use these data to update network. To update the critic net, should calculate the Mean Square Error loss of the predicted value and target value. The predicted value is the value of the output of critic net. And the target value is calculated by reward + r * max_a' Q(s', a') where max_a' Q(s', a') is the output of target critic net with action specified by target actor net according to next state.

```
## 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)
```

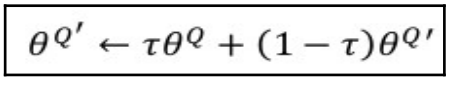


- First, get the action from actor net according to current state. Then the loss is the negative mean of the output of critic net with action specified according to currate state, because the target is that we want the output of critic net as large as possible.

```
## TODO ##
action = self._actor_net(state)
actor_loss = -self._critic_net(state, action).mean()
```

- update the network by soft copy the parameter calculated according to the formula.

```
@staticmethod
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 ##
        target.data.copy_(tau * behavior.data + (1 - tau) * target.data)
```



- The testing phase is similar with the training phase without warming up and updating network. In addition, should memorize reward and calculate the mean.

```
## TODO ##
for t in itertools.count(start=1):
    action = agent.select_action(state)
    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)
        break
rewards.append(total_reward)
```

- Describe differences between your implementation and algorithms

  There are two differences between my implementation and algorithms. First, there is a warm up phase in the implementation to store the process in replay buffer and would not update in this phase. Second, network does not update every steps , instead, it updates every 4 steps in my implementation.

- Describe your implementation and the gradient of actor updating

  First, get the action from actor net according to current state. Then the loss is the negative mean of the output of critic net with action specified according to currate state, because the target is that we want the output of critic net as large as possible. (Reference)

```
## TODO ##
action = self._actor_net(state)
actor_loss = -self._critic_net(state, action).mean()
```

- Describe your implementation and the gradient of critic updating

  First, randomly sample from the replay buffer, then use these data to update network. To update the critic net, should calculate the Mean Square Error loss of the predicted value and target value. The predicted value is the value of the output of critic net. And the target value is calculated by reward + r * max_a' Q(s', a') where max_a' Q(s', a') is the output of target critic net with action specified according to next state.

```
## 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)
```

- Explain effects of the discount factor

  λ is the discount factor in the following value function. The discount factor gets smaller as the k grows. The effects of discount factor is that the future time step which is closer to current time step affects more the further one.

$$G_t = R_{t+1} + \lambda R_{t+2} + \ldots = \sum_{k=0}^{\infty} \lambda^k R_{t+k+1}$$

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

  Greedy action selection always select the action with the max output of network according to the current state. However, this selection is based on the previous knowledge and there might be some action that were not taken before but they are the best action. On the other hand, epsilon-greedy selection allows to select a action randomly to discover its environment which is called exploration. Initially the agent doesn't know the outcomes of possible actions. Hence, sufficient initial exploration is required. When the agent explores, it can improve its current knowledge and gain better rewards in the long run.
- Explain the necessity of the target network

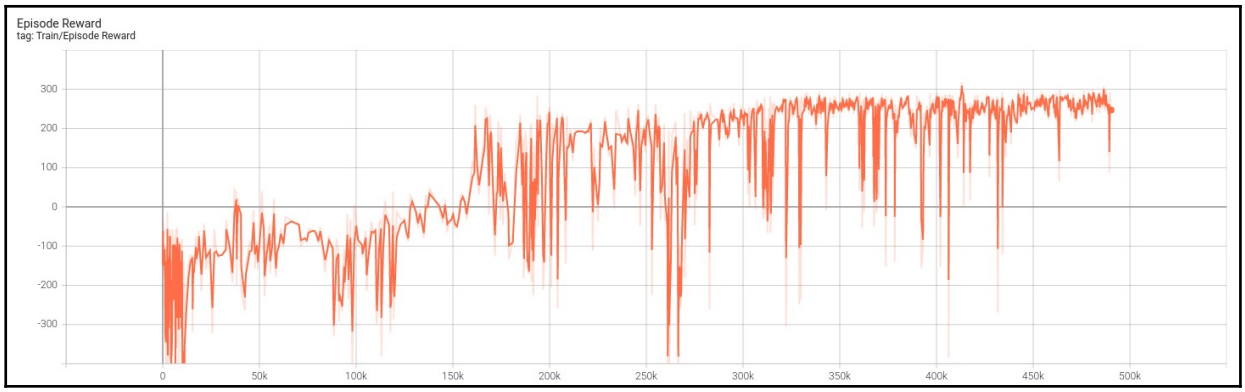
  Take a look at the Q function: Q(s, a) = reward + r * max_a' Q(s', a').
  If there is only one network, when updating the Q(s, a), the target Q(s', a') is changing too, and the result is that the training process will not be stable. Therefore, it is necessary to have two network, behavior network and target network, and let the target network update after fixed episodes.
- Explain the effect of replay buffer size in case of too large or too small

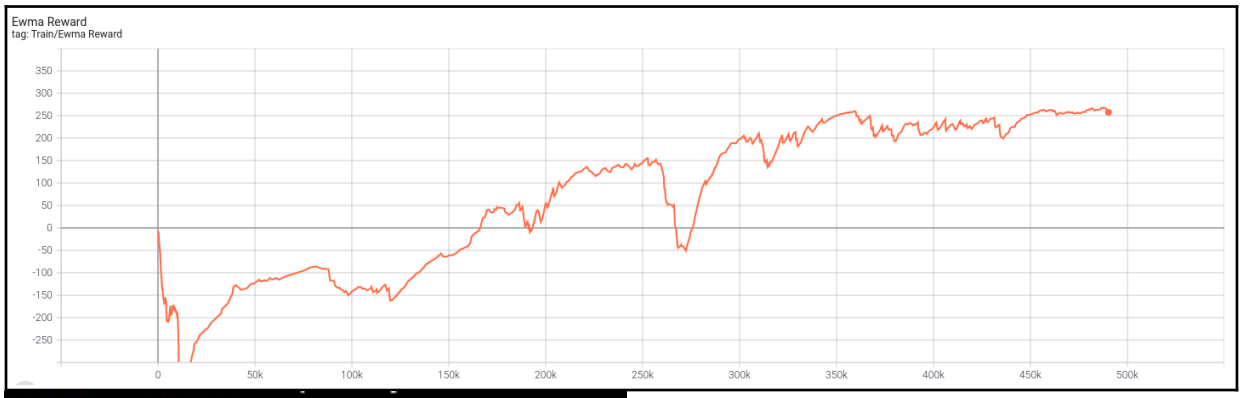
  The training data of Reinforcement Learning is better to be independent. However, there is temporality between training data. Therefore, building a replay buffer and randomly sampling training data from it can eliminate dependency to a certain degree. The larger the replay buffer, the less likely will sample correlated elements, hence the more stable the training of the NN will be. However, a large replay buffer also requires a lot of memory and it might slow training. On the other hand, the small the replay buffer, the more likely will sample correlated elements, hence the less stable the training of the NN will be.

- Implement and experiment on Double-DQN

  The main difference between DQN and Double-DQN is the way it derives target value. The way DQN deriving target value is using the max value of the output of target net according to next state and calculate with the formula. However, Double-DQN first gets the index of value of the output of behavior net according to next state. Then, use these indexes as the index of each output of target net according to next state and concatenate all of them. Finally, calculate with the formula to get target value.

```
with torch.no_grad():
    max_index = torch.max(self._behavior_net(next_state), 1).indices
    q_next = self._target_net(next_state)
    q_next_each_action = q_next[0][int(max_index[0])].view(1, 1)
    for i in range(1, max_index.size(0)):
        q_next_each_action = torch.cat((q_next_each_action, q_next[i][int(max_index[i])].view(1, 1)), 0)

    q_target = reward + gamma * q_next_each_action * (1 - done)
```






```
Start Testing
Average Reward 270.30944858381685
```

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

  - DQN: 272.344969

```
kai@kai-System-Product-Name:~/Desktop/NCTU-Deep_Learning_and_Practice-2020-Spring/Lab6$ python3 dqn.py --test_only
2021-05-21 16:59:10.769649: W tensorflow/stream_executor/platform/default/dso_loader.cc:59] Could not load dynamic library 'libcudart.so.10.1'
 dlerror: libcudart.so.10.1: cannot open shared object file: No such file or directory
2021-05-21 16:59:10.769683: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up o
 your machine.
Start Testing
Average Reward 272.34496912899056
```

  - DDPG: 276.57019

```
kai@kai-System-Product-Name:~/Desktop/NCTU-Deep_Learning_and_Practice-2020-Spring/Lab6$ python3 ddpg.py --test_only
2021-05-21 17:11:53.857423: W tensorflow/stream_executor/platform/default/dso_loader.cc:59] Could not load dynamic library 'libcudart.so.10.1'
; dlerror: libcudart.so.10.1: cannot open shared object file: No such file or directory
2021-05-21 17:11:53.857457: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up o
n your machine.
Start Testing
Average Reward 276.5701944176626
```