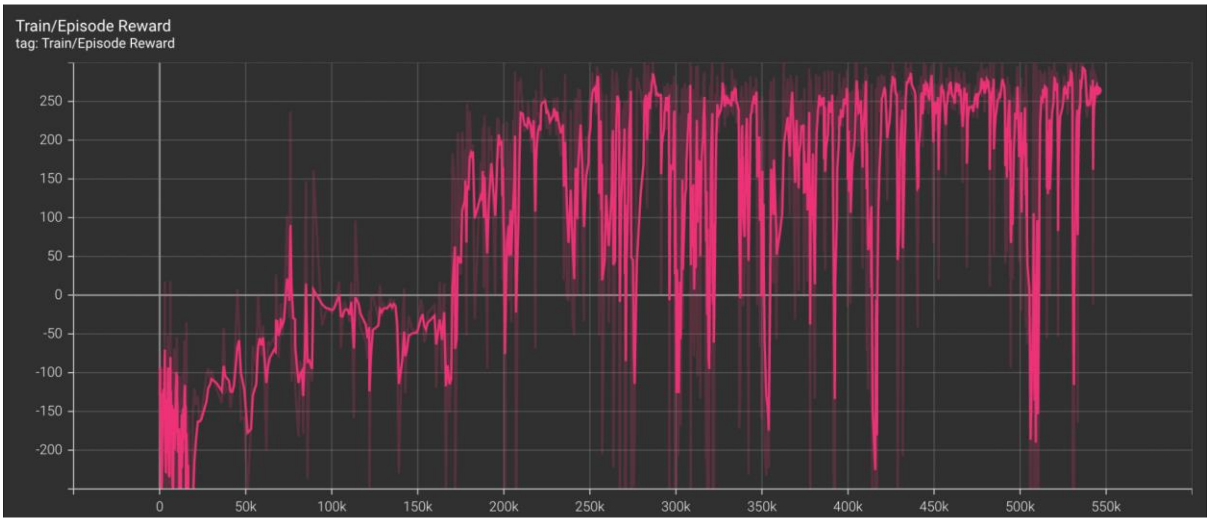
■A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2

DQN:

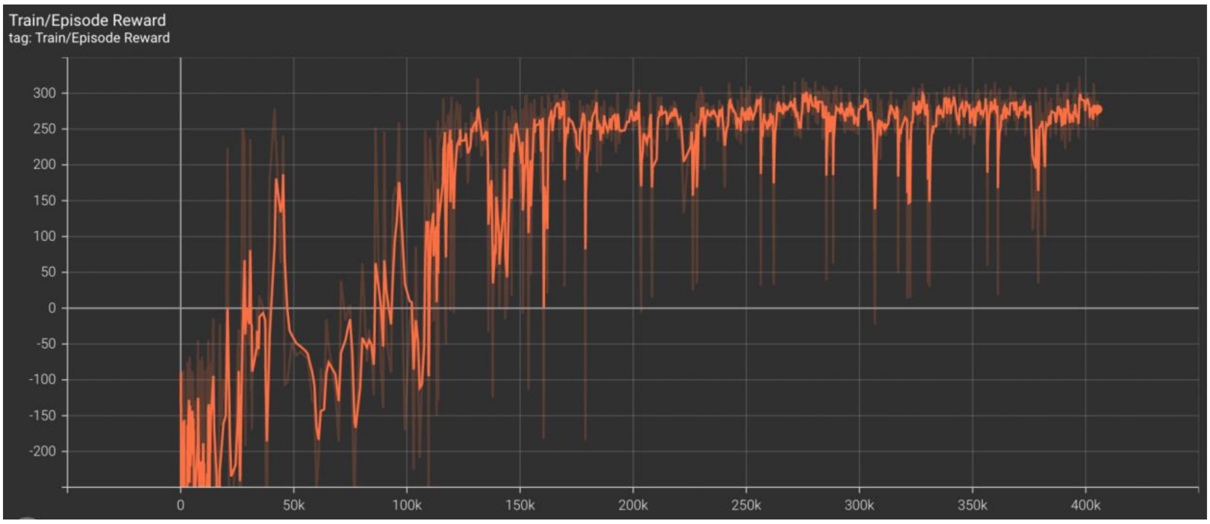
episode=1500



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

DDPG:

episode=1500



■ Describe your major implementation of both algorithms in detail.

DQN:

DQN是value based algorithm。

首先要先建個behavior network和target network下去計算action的value。

```
class DQN:
    def __init__(self, args):
        self._behavior_net = Net().to(args.device)
        self._target_net = Net().to(args.device)
        # initialize target network
        self._target_net.load_state_dict(self._behavior_net.state_dict())
        ## TODO ##
        self._optimizer = torch.optim.Adam(self._behavior_net.parameters(), lr=args.lr)
        #raise NotImplementedError
        # memory
        self._memory = ReplayMemory(capacity=args.capacity)
```

接著是action的選擇，由於是value based，所以這邊先去計算每個action的value後，再做epsilon greedy，也就是有epsilon的機率隨便選一種action，1-epsilon的機率選value最大的機率。

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    state = torch.from_numpy(state).to(self.device)
    action_values = self._behavior_net(state)

    # epsilon-greedy
    if random.random() < epsilon:
        action = random.choice(np.arange(action_values.shape[-1])).item()
    else:
        action = torch.argmax(action_values).item()

    return action
```

接著是update behavior net的部分，就照著演算法下去改。

具體來講就是reward+q_target(next_state)=q_target(state)，然後和q_behavior(state)下去做mse。

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

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

```
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_behavior = self._behavior_net(state)
    q_value = torch.Tensor(self.batch_size, 1).to(self.device)
    for i in range(self.batch_size):
        q_value[i] = q_behavior[i][int(action[i].item())]
    with torch.no_grad():
        q_next = self._target_net(next_state).max(axis=1)[0].view(-1, 1)
        q_target = reward + self.gamma * q_next * (1 - done)
    criterion = nn.MSELoss()
    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()
```

接著是update target net，也就是每經過一定次數的step後，將behavior net的參數直接更新到target net上。

Every C steps reset $\hat{Q} = Q$

```
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())
```

至於network和hyperparameter魔改的部分，留到後面說明。

DDPG:

DQN是method based algorithm，也就是policy based和valued based的混合版。

首先要先建個critic network和actor network下去計算value和action。

然後這兩個還要在分別建個target network，總共共4個network。

```
class DDPG:
    def __init__(self, args):
        # behavior network
        self._actor_net = ActorNet().to(args.device)
        self._critic_net = CriticNet().to(args.device)
        # target network
        self._target_actor_net = ActorNet().to(args.device)
        self._target_critic_net = CriticNet().to(args.device)
        # initialize target network
        self._target_actor_net.load_state_dict(self._actor_net.state_dict())
        self._target_critic_net.load_state_dict(self._critic_net.state_dict())
        ## 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)
```

至於動作的選擇，就是靠actor network來決定。

然後為了能做到探索的效果，所以就在output加上noise。

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.from_numpy(state).to(self.device)
    action =[]
    actor_output = self._actor_net(state)
    action.append(actor_output[0].item())
    action.append(actor_output[1].item())

    if noise == True:
        action += self._action_noise.sample()

    return action
```

然後critic 和 actor的update，留到後面做說明。

接著講一下如何update target network，跟dqn不同的是，ddpg是做soft update，也就是每次更新時，是照著原target network和local network的parameter下去做加權平均。其中tau是一個很小的數字，代表更新時，target network不會有很大的變動。

```
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_(behavior.data * tau + target.data * (1.0 - tau))
```

至於network和hyperparameter魔改的部分，留到後面說明。

■ Describe differences between your implementation and algorithms.

主要是改network和hyperparameter，hyperparameter留到bonus做說明。

首先是dqn的network:

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=64):
        super().__init__()
        ## TODO ##
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim)
            )

    def forward(self, x):
        ## TODO ##
        return self.net(x)
```

把原本的三層網路變成五層，然後hidden_dim從32變成64。

可以看到紅色練起來果然有比原本藍色好一些。

接下來是ddpg:

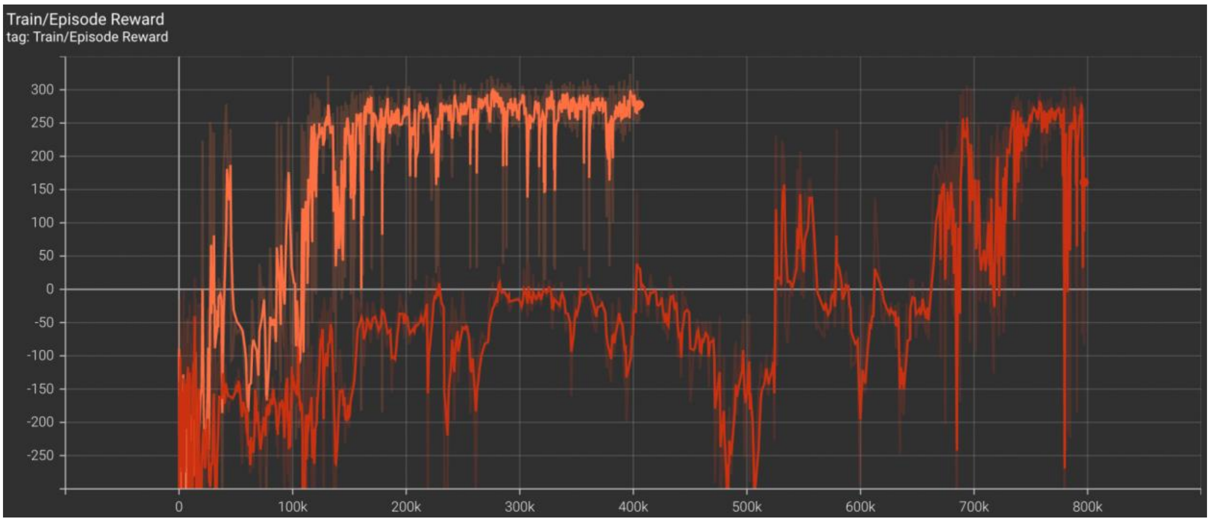
我就直接將network無腦堆個5層和4層fc，效果顯著。

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        h1, h2 =hidden_dim
        self.net = nn.Sequential(
            nn.Linear(state_dim, h1),
            nn.ReLU(),
            nn.Linear(h1, h1),
            nn.ReLU(),
            nn.Linear(h1, h1),
            nn.ReLU(),
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, action_dim)
            )
        ## TODO ##
    def forward(self, x):
        return torch.tanh(self.net(x))

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, h1),
            nn.ReLU(),
            nn.Linear(h1, h1),
            nn.ReLU(),
            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)
```

紅線是原本的network，橘線是我魔改後的，可以發現reward穩定很多。
至於hyperparameter怎麼調，留到後面談。



■ Describe your implementation and the gradient of actor updating.

藉由計算policy gradiant，去更新actor network。
policy gradiant公式如下：

Update the actor policy using the sampled gradient:

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

然後因為希望能收斂到max Q，反過來說就是希望能做成min -Q，所以要做gradiant ascent，要多加個負號。
實作如下：

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

■ Describe your implementation and the gradient of critic updating.

公式如下：
其實精神跟dqn更新behavior net幾乎一模一樣，只差在behavior net選action時是挑value最大的那個action，而ddpg是直接用actor output action。

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

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

實作如下：

```
## update critic ##
# critic loss
## TODO ##
q_value = critic_net(state, action)
with torch.no_grad():
    action_next = target_actor_net(next_state)
    q_next = target_critic_net(next_state, action_next)
    q_target = reward + gamma * q_next *(1-done)
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()
```

■ Explain effects of the discount factor.

在RL中，常會用對未來的期望值來做出action，但問題是理論上現在的action對越未來的reward應該影響會越小，好比說國小用功讀書不代表升大學學測能考好，所以加上個discount factor比較貼近符合現實。反過來說，如果discount factor接近1，那代表reward較不易因為時間而改變。

Recall that the return is the total discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T$$

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

Greedy actoi selection是照著max estimate value下去選擇action，最後不一定能帶來最好的結果，因為estimate value不見得跟value差不多。所以需要額外一點隨機探索的機會，於是epsilon greedy就誕生了，epsilon greedy會分一點機率epsilon作為隨機選擇action。

■ Explain the necessity of the target network.

如果沒有target network，這就代表每次更新時，都是用同個網路來estimate value，會造成收斂的不穩定，所以要用一個target network先固定部分的value。然後如果update target network是直接copy的話，那最好update target network的時間長一點比較好。

■ Explain the effect of replay buffer size in case of too large or too small.

buffer size太大的話，會浪費空間，training的時間也會過長。反之buffer size太小的話，sample太少，可能會overfitting。

● Report Bonus

■ Implement and experiment on Double-DQN

DDQN和DQN的差別只差在update behavior net。

因為DQN的behavior net選了覺得value最大的動作，然後又對其動作後的state取behavior net的對應value，下去對q_next做update，但實際上這樣的算法會造成overestimate。

為了不造成overestimate，所以就讓behavior net選了覺得value最大的動作，然後又對其動作後的state取target net的對應value。

這樣就不會有overestimate的問題。

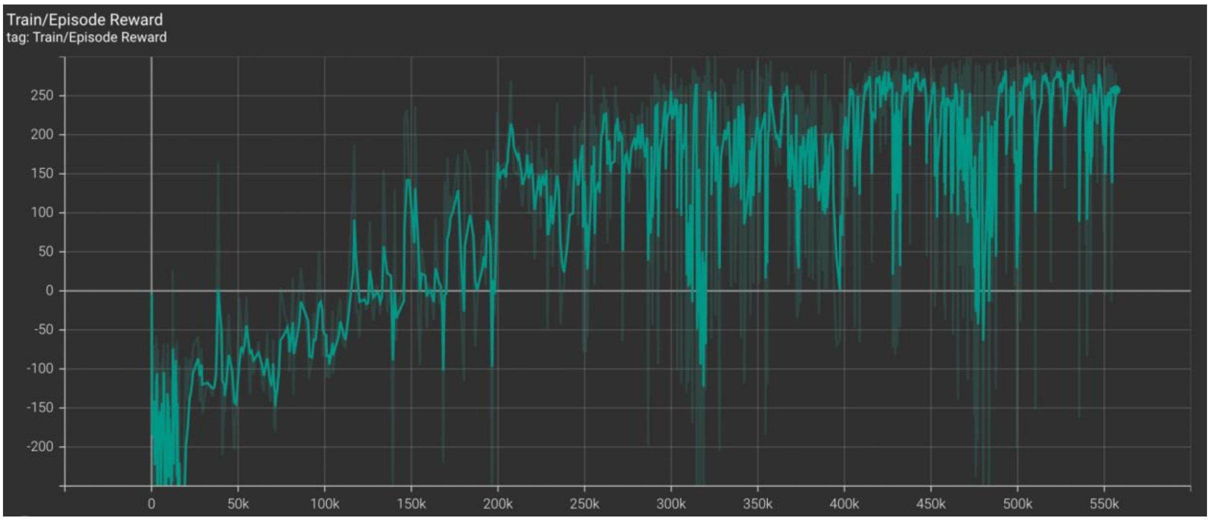
```python
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(
        self.batch_size, self.device)

    ## TODO ##
    q_behavior = self._behavior_net(state)
    q_value = torch.Tensor(self.batch_size, 1).to(self.device)
    for i in range(self.batch_size):
        q_value[i] = q_behavior[i][int(action[i].item())]
    with torch.no_grad():
        index = self._behavior_net(next_state).argmax(axis=1).view(-1, 1)
        q_next_state_target = self._target_net(next_state)
        q_next = q_next_state_target.gather(1,index)
        q_target = reward + self.gamma * q_next * (1 - done)
    criterion = nn.MSELoss()
    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()
```

然後DDQN我沒有魔改network，所以結果就比較普通。

1500 episode:




```
Start Testing
Episode: 1   Total reward: 248.6319476776429
Episode: 2   Total reward: 294.84786082986335
Episode: 3   Total reward: 260.0752677158988
Episode: 4   Total reward: 283.3119505782211
Episode: 5   Total reward: 32.10904117560628
Episode: 6   Total reward: 268.4304717471266
Episode: 7   Total reward: 66.17109590514451
Episode: 8   Total reward: 307.85191382588965
Episode: 9   Total reward: 43.35713270955932
Episode: 10   Total reward: 285.2801858585734
Average Reward 209.00668680235262
```

■ Extra hyperparameter tuning

DQN:

除了調網路，我還有調一些參數。我覺得影響最大的是將target-freq調成1000這件事，沒調這個前是完全train不起來的。此外我把warmup和episode train調長一點，防止有時候收斂太慢的問題。

```python
def main():
    ## arguments ##
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('-d', '--device', default='cuda')
    parser.add_argument('-m', '--model', default='dqn.pth')
    parser.add_argument('--logdir', default='log/dqn')
    # train
    parser.add_argument('--warmup', default=15000, type=int)
    parser.add_argument('--episode', default=1500, type=int)
    parser.add_argument('--capacity', default=10000, type=int)
    parser.add_argument('--batch_size', default=256, type=int)
    parser.add_argument('--lr', default=.0005, type=float)
    parser.add_argument('--eps_decay', default=.999, type=float)
    parser.add_argument('--eps_min', default=.01, type=float)
    parser.add_argument('--gamma', default=.99, type=float)
    parser.add_argument('--freq', default=4, type=int)
    parser.add_argument('--target_freq', default=1000, type=int)
    # test
    parser.add_argument('--test_only', action='store_true')
    parser.add_argument('--render', action='store_true')
    parser.add_argument('--seed', default=20200519, type=int)
    parser.add_argument('--test_epsilon', default=.001, type=float)
    args = parser.parse_args()
```

DDPG:

我把warmup和episode train調長一點，防止有時候收斂太慢的問題。

原本想對lr做調整，後來發現lr=1e-3剛剛好。

```python
## arguments ##
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('-d', '--device', default='cuda')
parser.add_argument('-m', '--model', default='ddpg2-version2.pth')
parser.add_argument('--logdir', default='log/ddpg2-version2')
# train
parser.add_argument('--warmup', default=15000, type=int)
parser.add_argument('--episode', default=1500, type=int)
parser.add_argument('--batch_size', default=64, type=int)
parser.add_argument('--capacity', default=500000, type=int)
parser.add_argument('--lra', default=1e-3, type=float)
parser.add_argument('--lrc', default=1e-3, type=float)
parser.add_argument('--gamma', default=.99, type=float)
parser.add_argument('--tau', default=.005, type=float)
# test
parser.add_argument('--test_only', action='store_true')
parser.add_argument('--render', action='store_true')
parser.add_argument('--seed', default=20200519, type=int)
args = parser.parse_args()
```

- Performance

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

DQN:

```
Start Testing
Episode: 1   Total reward: 247.21507268096178
Episode: 2   Total reward: 280.30582134493943
Episode: 3   Total reward: 276.1912791100215
Episode: 4   Total reward: 272.64707298131896
Episode: 5   Total reward: 283.94337557618394
Episode: 6   Total reward: 273.0335383261325
Episode: 7   Total reward: 303.9709742571548
Episode: 8   Total reward: 290.2412148686906
Episode: 9   Total reward: 304.09256604843773
Episode: 10   Total reward: 284.56696219597336
Average Reward 281.62078773898145
```

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

DDPG:

```
Start Testing
Episode: 1   Total reward: 245.21532035568015
Episode: 2   Total reward: 282.1427679863917
Episode: 3   Total reward: 272.015185424042
Episode: 4   Total reward: 276.5198699559628
Episode: 5   Total reward: 306.08095228866625
Episode: 6   Total reward: 265.57063866476517
Episode: 7   Total reward: 301.05849619032074
Episode: 8   Total reward: 294.60197173277487
Episode: 9   Total reward: 310.0719659453488
Episode: 10   Total reward: 289.9587059933458
Average Reward 284.32358745372983
```