NCTU DLP Lab8-Report
DQN and DDPG

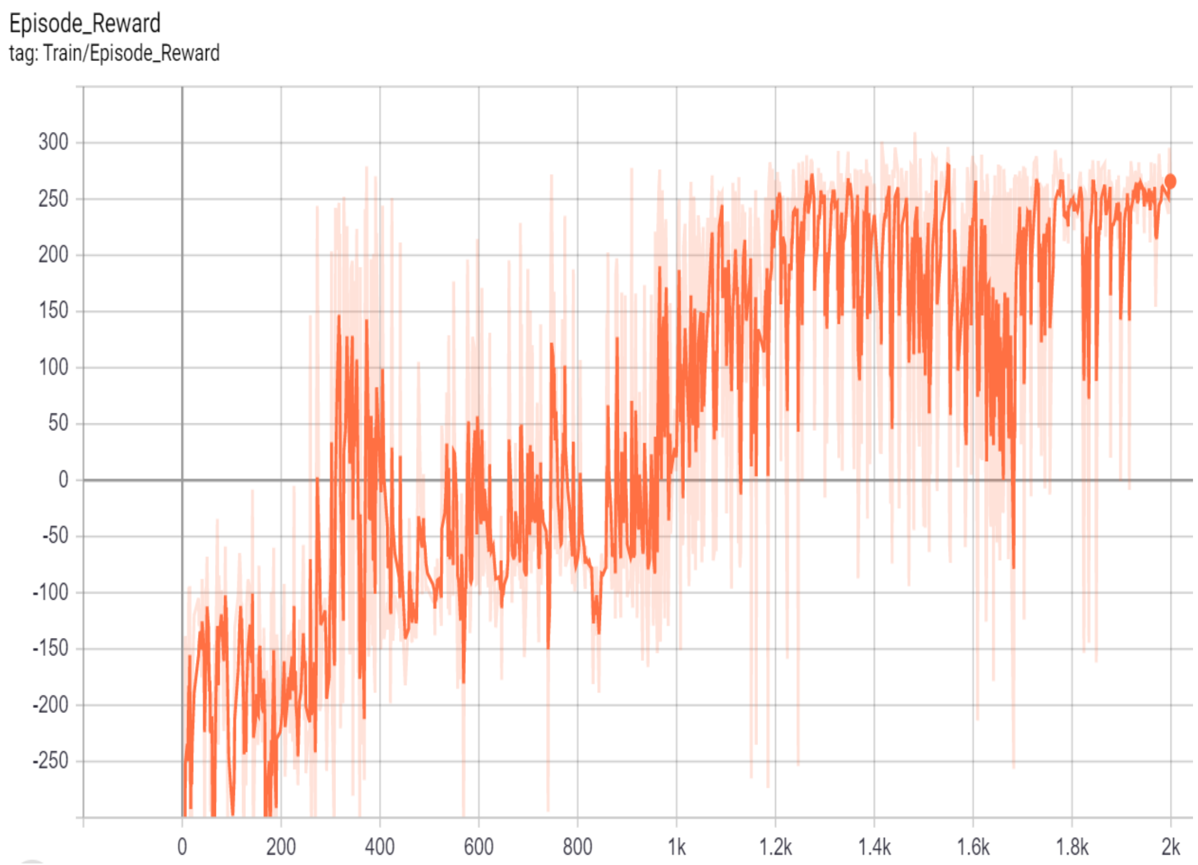
廖家鴻 0786009
2020/6/15

Outline

- Report (80%)
  - A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLander-v2 (5%)
  - A tensorboard plot shows episode rewards of at least 800 training episodes in LunarLanderContinuous-v2 (5%)
  - Describe your major implementation of both algorithms in detail. (20%)
  - Describe differences between your implementation and algorithms. (10%)
  - Describe your implementation and the gradient of actor updating. (10%)
  - Describe your implementation and the gradient of critic updating. (10%)
  - Explain effects of the discount factor. (5%)
  - Explain benefits of epsilon-greedy in comparison to greedy action selection. (5%)
  - Explain the necessity of the target network. (5%)
  - Explain the effect of replay buffer size in case of too large or too small. (5%)
- Performance (20%)
  - [LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30
  - [LunarLanderContinuous-v2] Average reward of 10 testing episodes: Average ÷ 30
- Report Bonus (20%)
  - Implement and experiment on Double-DQN (10%)

Report

Tensorboard Episode Rewards Plot in LunarLander-v2



Tensorboard Episode Rewards Plot in LunarLanderContinuous-v2

Implementation Details1– DQN -- 1

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super(Net, self).__init__()
        ## TODO ##
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim*2)
        self.fc3 = nn.Linear(hidden_dim*2, action_dim)

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

左圖前兩個TODO，是在建立behavior和target兩個net的共同結構。

```python
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.lr = args.lr
        self._optimizer = optim.Adam(self._behavior_net.parameters(), lr=self.lr)
```

左圖為第三個TODO，是在選擇優化器，在這裡我選擇Adam optimizer。

Implementation Details1– DQN -- 2

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    rnd = random.random()
    if rnd < epsilon:
        return np.random.randint(action_space.n)
    else:
        state = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
        # set the network into evaluation mode
        self._behavior_net.eval()
        with torch.no_grad():
            action_values = self._behavior_net(state)
        # Back to training mode
        self._behavior_net.train()
        action = np.argmax(action_values.cpu().data.numpy())
        return action
```

左圖為第四個TODO，主要在執行epsilon-greedy based的action selection，讓整個DQN可以既exploitation又exploration。

注意這裡rnd > epsilon時的動作選擇，必須是用behavior net而不是target net，並在最後return出屬於最大Q值的action。

```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 ##
    action_values = self._target_net(next_state).detach()
    max_action_values = action_values.max(1)[0].unsqueeze(1)
    q_target = reward + (gamma * max_action_values * (1 - done))
    q_next = self._behavior_net(state).gather(1, action.long())
    loss = F.mse_loss(q_next, q_target)

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

左圖為第五個TODO，主要在update behavior net。先用target net輸出的最大Q值乘以discount factor後加上reward來算出q_target，再用behavior net來輸出對應replay buffer中sample出來的(state, action)之q_next。然後算q_target和q_next的mse_loss，再進行backward optimization。

(注意此時target net的參數需凍結不update)

Implementation Details1– DQN -- 3

```python
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    for source_parameters, target_parameters in zip(self._behavior_net.parameters(), self._target_net.parameters()):
        target_parameters.data.copy_(self.TAU * source_parameters.data + (1.0 - self.TAU) * target_parameters.data)
```

```python
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 = []
    total_steps = 0
    for n_episode, seed in enumerate(seeds):
        total_reward = 0
        env.seed(seed)
        state = env.reset()
        ## TODO ##
        for t in itertools.count(start=1):
            # select action
            action = agent.select_action(state, epsilon, action_space)
            # execute action
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward
            total_steps += 1
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                break
        rewards.append(total_reward)
    print('Average Reward', np.mean(rewards))
    print(rewards)
    env.close()
```

上圖為第六個TODO，主要用來更新target net的參數。這裡我採用soft replace的方法，也就是target net的參數只用TAU值的比例是用behavior net的參數來update

左圖為第七個TODO，主要在執行testing，所以把training時的store transition和model update拿掉。

Implementation Details1– DQN -- 4

```python
## 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=10000, type=int)
parser.add_argument('--episode', default=2000, type=int)
parser.add_argument('--capacity', default=50000, type=int)
parser.add_argument('--batch_size', default=128, type=int)
parser.add_argument('--lr', default=.0005, type=float)
parser.add_argument('--eps_decay', default=.9999, 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=100, type=int)
parser.add_argument('--TAU', default=.01, 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)
parser.add_argument('--test_epsilon', default=.001, type=float)
args = parser.parse_args()
```

左圖是這次DQN training時最佳的參數。

Implementation Details2 – DDPG -- 1

```
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))
```

左圖為第一個TODO，主要是從replay buffer中random sample出batch experience data for off-policy training

下圖為第二&三個TODO，為ActorNet的code，net結構是參考spec中的actor結構圖實作而成

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super(ActorNet, self).__init__()
        ## TODO ##
        self.l1 = nn.Linear(state_dim, 400)
        self.l1.weight.data.normal_(0,0.1) # initialization
        self.l2 = nn.Linear(400, 300)
        self.l2.weight.data.normal_(0,0.1) # initialization
        self.l3 = nn.Linear(300, action_dim)
        self.l3.weight.data.normal_(0,0.1) # initialization
        # self.max_action = max_action
    def forward(self, x):
        ## TODO ##
        a = F.relu(self.l1(x))
        a = F.relu(self.l2(a))
        a = torch.tanh(self.l3(a))# * self.max_action
        return a
```

下圖並非TODO，是我將sample code中的CriticNet依照spec中的critic結構改寫實作而成

```
class CriticNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        h1, h2 = hidden_dim
        self.fcs1 = nn.Linear(state_dim,h1)
        self.fcs1.weight.data.normal_(0,0.1) # initialization
        self.fcs2 = nn.Linear(h1,h2)
        self.fcs2.weight.data.normal_(0,0.1) # initialization
        self.fca = nn.Linear(action_dim,h2)
        self.fca.weight.data.normal_(0,0.1) # initialization
        self.out = nn.Linear(h2,2)
        self.out.weight.data.normal_(0, 0.1)  # initialization
    def forward(self,s,a):
        x = self.fcs1(s)
        x = self.fcs2(x)
        y = self.fca(a)
        net = F.relu(x+y)
        actions_value = self.out(net)
        return actions_value
```

Implementation Details2 – DDPG -- 2

```python
class DDPG:
    def __init__(self, args, max_action):
        # 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 = optim.Adam(self._actor_net.parameters(), lr=args.lra)
        self._critic_opt = optim.Adam(self._critic_net.parameters(), lr=args.lrc)
```

左圖為第四個TODO，我將actor和critic的優化器選為Adam optimizer

```python
def select_action(self, state, noise_inp=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.FloatTensor(state).to(self.device)
    next_action = self._actor_net(state).detach()
    if noise_inp == True:
        self._action_noise = GaussianNoise(dim=2, mu=None, std=self.Gau_var)
        noise = self._action_noise.sample()
        noise = torch.FloatTensor(noise).to(self.device)
        noise_action = next_action + noise
        noise_a_next = noise_action.clamp(-self.max_action, self.max_action)
        self.Gau_var *= self.Gau_decay
        return noise_a_next.cpu().data.numpy()
    else:
        return next_action.cpu().data.numpy()
```

左圖為第五個TODO。有別於DQN是discrete actions可以用epsilon-greedy來選擇動作，DDPG的連續型的action space，若想一樣保持exploitation又exploration，則要採用Ornstein-Uhlenbeck的方法，而左圖中noise_inp==True的部分，就是該隨機流程的實作。

Implementation Details2 – DDPG -- 3

```python
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)
```

左圖為第8個TODO，主要是target net的update。這裡採用soft replace的方法來update。

```python
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()
        total_steps = 0
        ## TODO ##
        for t in itertools.count(start=1):
            # select action
            action = agent.select_action(state, noise_inp=False)
            # execute action
            next_state, reward, done, _ = env.step(action)
            state = next_state
            total_reward += reward
            total_steps += 1
            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                break
        rewards.append(total_reward)
    print('Average Reward', np.mean(rewards))
    print(rewards)
    env.close()
```

左圖為第9個TODO，主要是執行trained DDPG model的testing。注意這裡的動作選擇就不用加noise了，所以noise_inp設為False。

註：第6&7個TODO將在後面的Actor Critic的updating處說明。

Implementation Details2 – DDPG -- 4

```python
## arguments ##
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('-d', '--device', default='cuda')
parser.add_argument('-m', '--model', default='ddpg.pth')
parser.add_argument('--logdir', default='log/ddpg')
# train
parser.add_argument('--warmup', default=10000, type=int)
parser.add_argument('--episode', default=2000, type=int)
parser.add_argument('--batch_size', default=64, type=int)
parser.add_argument('--capacity', default=100000, 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=.001, type=float)
parser.add_argument('--Gau_var', default=3, type=float)
parser.add_argument('--Gau_decay', default=0.99995, 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()
```

左圖為我這次DDPG training的最佳參數。

Differences between implementation and Algorithm- DQN

Algorithm – Deep Q-learning with experience replay:

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

```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()
        for t in itertools.count(start=1):
        # for t in range(300): #itertools.count(start=1):
            # select action
            if total_steps < args.warmup:
                action = action_space.sample()
            else:
                action = agent.select_action(state, epsilon, action_space)
                epsilon = max(epsilon * args.eps_decay, args.eps_min)
            # execute action
            next_state, reward, done, _ = env.step(action)
            # store transition
            agent.append(state, action, reward, next_state, done)
            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, episode)
                writer.add_scalar('Train/Ewma Reward', ewma_reward, episode)
                print('Step: {}\tEpisode: {}\tLength: {:3d}\tTotal reward: {:.2f}\tEwma reward: {:.2f}\tEpsilon: {:.3f}'
                      .format(total_steps, episode, t, total_reward, ewma_reward, epsilon))
                if ewma_reward > 220:
                    model_fn = 'save_dqn/DQN_eps' +str(episode) +'reward' +str(np.round(ewma_reward,2)) +'.pth'
                    agent.save(model_fn)
                break
    env.close()
```

有三個不同之處：

1. 在total_step>=warmup之後，動作選擇才進入epsilon-greedy方式，並且model也是此時開始update。
2. 在前的TODO有提到，在探索時的epsilon值有實作成隨著steps增加而decay，到後面的training時就是exploitation dominate。
3. 我在target_net的update實作中，採用soft replace，並非完全copy behavior_net來reset target_net的參數。

Differences between implementation and Algorithm- DDPG

Algorithm – DDPG algorithm:

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer R

for $episode = 1, M$ **do**

Initialize a random process N for action exploration

Receive initial observation state s_1

for $t = 1, T$ **do**

Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise

Execute action a_t and observe reward r_t and observe new state s_{t+1}

Store transition (s_t, a_t, r_t, s_{t+1}) in R

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

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

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

Update the actor policy using the sampled gradient:

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

Update the target networks:

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

end for

end for

```python
def train(args, env, agent, writer):
    print('Start Training')
    total_steps = 0
    ewma_reward = 0
    for episode in range(args.episode):
        total_reward = 0
        state = env.reset()
        for t in itertools.count(start=1):
        # for t in range(300): #itertools.count(start=1):
            # select action
            if total_steps < args.warmup:
                action = env.action_space.sample()
            else:
                action = agent.select_action(state)
            # execute action
            next_state, reward, done, _ = env.step(action)
            # store transition
            agent.append(state, action, reward, next_state, done)
            if total_steps >= args.warmup:
                agent.update()

            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, episode)
                writer.add_scalar('Train/Ewma Reward', ewma_reward, episode)
                print(
                    'Step: {}\tEpisode: {}\tLength: {:3d}\tTotal reward: {:.2f}\tEwma reward: {:.2f}'
                    .format(total_steps, episode, t, total_reward, ewma_reward))
                if ewma_reward > 270:
                    model_fn = 'save_ddpg/DDPG_eps' +str(episode) +'reward' +str(np.round(ewma_reward,2)) +'.pth'
                    agent.save(model_fn)
                break
    env.close()
```

有三個不同之處：

1. 在total_step>=warmup之後，動作選擇才進入Ornstein-Uhlenbeck方式，並且model也是此時開始update。
2. 在前的TODO有提到，在探索時的gaussian noise的variance有實作成隨著steps增加而decay，到後面的training時就是exploitation dominate。
3. 這裡actor的update並非coding成左圖algorithm的sampled gradient ascent。而是將actor_net的output接到critic_net後的output取Q值的負的期望值，然後再對actor_net進行gradient descent。

Implementation of the Gradient of Actor Updating - DDPG

– Actor updates policy in direction suggested by critic (DDPG):

$$\nabla_\theta J(\mu_\theta) \approx \mathbb{E}_\mu[\nabla_\theta Q(s_t, \mu(s_t|\theta)|\omega)]$$

$$= \mathbb{E}_\mu\left[\nabla_a Q(s_t, a|\omega)\Big|_{a=\mu(s_t|\theta)} \nabla_\theta \mu(s_t|\theta)\right]$$

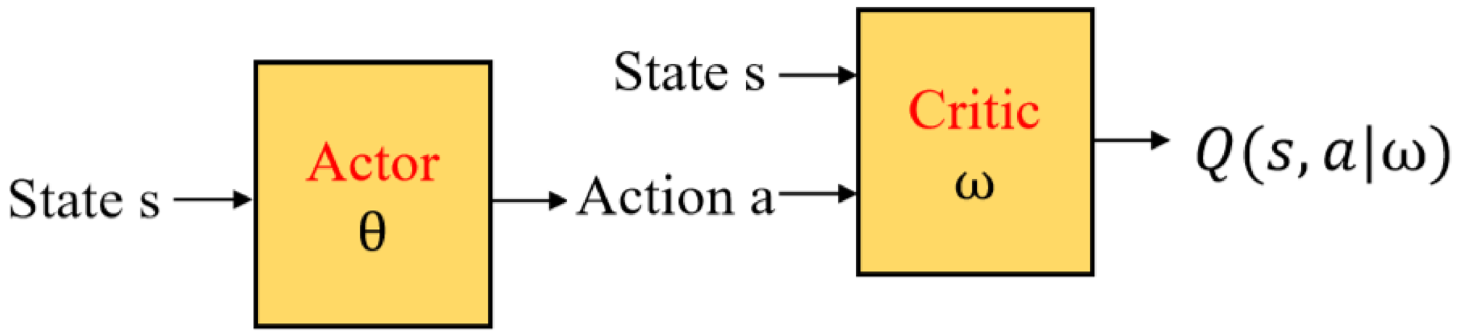
上圖說明原本用來update actor net的參數所用的gradient formula，然後理論上update actor是採gradient ascent，不過實作時有調整做法來達到一樣的效果。

右圖是第6個TODO，code中說明我實際實作時的loss計算與actor_net的update。

Coding的概念如下圖所示，將actor輸出的action接到critic的input，然後再將critic output出來的Q值取負的期望值，然後只對actor_net進行gradient descent optimization。

```
##### update actor #####
# actor loss
## TODO ##
a_act = actor_net(state)
q_act = critic_net(state, a_act)
actor_loss = -torch.mean(q_act)

### optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```



Implementation of the Gradient of Critic Updating - DDPG

- Critic estimates value of current action by Q-learning
  - Gradient:

$$\nabla_\omega L_Q(s_t, a_t|\omega) = \Big(\big(r_{t+1} + \gamma Q(s_{t+1}, \mu(s_{t+1}|\theta)|\omega)\big) - Q(s_t, a_t|\omega)\Big)\nabla_\omega Q(s_t, a_t|\omega)$$

上圖說明critic update方式採跟Q-learning很相似的做法

下圖是第7個TODO，說明critic的update方式，其實跟DQN很像。先用target_actor_net輸出next action，再把這action與next_state輸入target_critic_net，然後該output再與gamma和reward做乘加來得到q_target。至於q_next這個預測值則由屬behavior的critic_net產生，然後再與q_target取MSE_loss，而最後只update critic_net

```
##### update critic #####
# critic loss
## TODO ##
a_next = target_actor_net(next_state).detach()
q_target = reward + (1-done) * self.gamma * target_critic_net(next_state, a_next).detach()
q_next = critic_net(state, action)
loss_fn = nn.MSELoss()
critic_loss = loss_fn(q_next, q_target)

### optimize critic
actor_net.zero_grad()
critic_net.zero_grad()
critic_loss.backward()
critic_opt.step()
```

Explain the effects of the discount factor

- Target Q (A real number):
  - $Y_t^Q = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'|\theta)$
- Loss Function:
  - $L_Q(s_t, a_t|\theta) = \left(Y_t^Q - Q(s_t, a_t|\theta)\right)^2$

左圖為DQN的target Q和loss function。

從Y_t^Q式子中，發現這是屬於1-step的TD target，而在此時的γ值，只是決定maxQ值在target的比例，對training是有影響，但沒有n-step來的大。

N-step return如下圖，discount factor γ∈ (0，1]，通過公式可以看出，隨著時間愈遠的γ次方愈大，就代表對整個累積回報$G_t^{(n)}$值的影響越小，所以時間近的狀態value function影響更大，從而對決策結果影響更大。

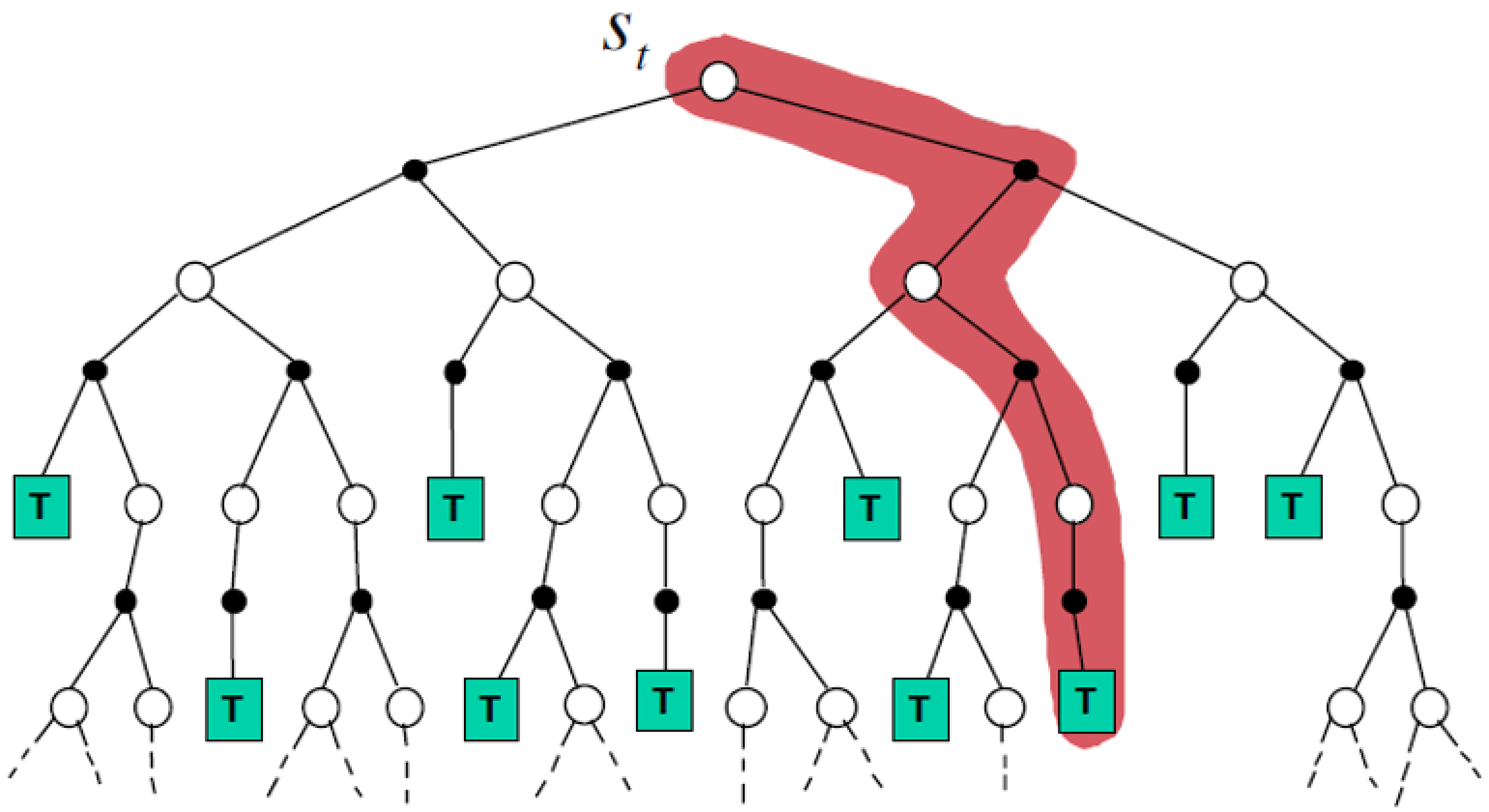
Define the n-step return

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

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

epsilon-greedy的最主要好處是可以讓agent保持既exploitation又exploration。

若讓policy一直都是greedy action selection的話，如下圖，會讓agent傾向走某些specific的trajectory，而對於其它的trajectories都沒去探索到，如此train出來的RL model會不夠smart，在實際testing時容易失敗。



Explain the necessity of the target network

- 多一個target net主要想想穩定model behaior。希望每次訓練時，target值不要動來動去，如此便可穩定訓練過程。

- 若都是用同一個behavior network，因為update它的頻率高，則model output出來的Q值跳動會很大，很可能一時間偏某個action，下一時間馬上又偏向不同的action，所以用update頻率相對低很多的target net就可以避免掉這個現象。

Explain the effect of replay buffer size

- 先說明replay buffer的作用：深度學習的training data最好為獨立同分佈的data，然而RL的data是有時序性的、代表data前後是有關聯的，這樣可能會造成模型無法正常訓練，所以建立了一個buffer來儲存trajectory data，並且利用Random batch sample的方式來進行training，這樣就不會有關聯性的問題。舉例：雖然像2048這類遊戲是可以照原trajectory學得起來，但遇到像pacman這類上下左右走來走去的，若不從經驗裡random學的話，很容易overfitting到某一區的某些動作，就學不到optimal policy。
- too large：我推測這樣的random sample會選到太舊的經驗，而通常太舊經驗多是model還不夠smart時所選出來的bad action policy，所以buffer太大的話，model萬一學到太多舊的不佳的經驗data，就很有可能train不起來。也許buffer大讓model學習後的bias小，但卻可能讓variance變太大。
- too small：buffer太小即存儲下來給model訓練的經驗data就相對少，如此容易造成model在learning的時候會有bias。就算在某次training的score不錯，但因為有bias，很可能在不同environment的testing時失敗。

Performance

[LunarLander-v2] Average reward of 10 testing episodes

下圖10個testing episodes的結果，並且DQN在此的Average Reward~=275.09

```
Average Reward 275.08983918591707
[254.52285566464948, 280.4913389870097, 256.61373966610023, 268.3173771227329,
290.0037787613699, 267.65676622380147, 300.010230514732, 286.4078931999634,
296.14639583750784, 250.72801588130352]
```

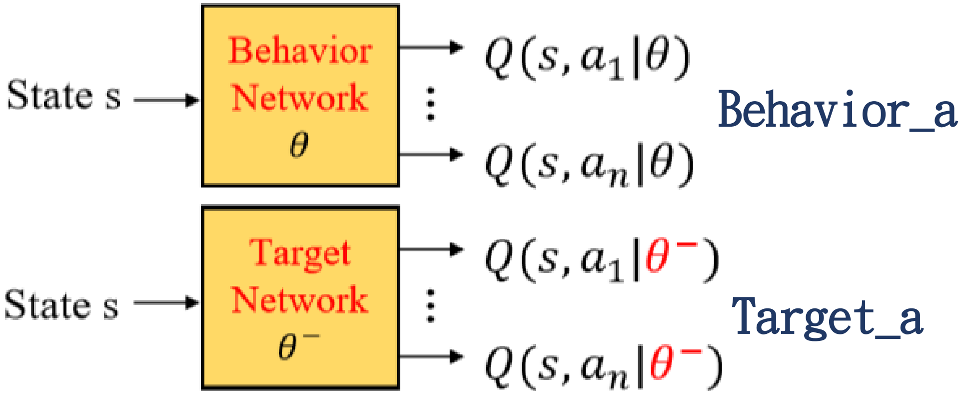
[LunarLanderContinuous-v2] Average reward of 10 testing episodes

下圖10個testing episodes的結果，並且DQN在此的Average Reward~=290.33

```
Start Testing
Average Reward 290.3325701931041
[275.88316603793317, 299.9577524514271, 279.36971610717876, 298.6305462903296,
274.3213609631681, 287.4906914116312, 310.4247854817372, 303.58665027094423,
304.8802445705487, 268.78078834614314]
```

Report Bonus

Implement and experiment on Double-DQN -- 1



- Prevent over-optimistic value estimates on DQN.
- Decouple the selection from the evaluation.

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

$$\Downarrow$$

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

在DQN的training中，用來計算Q_target的maxQ值對應的action常常與behavior的action不同，也就是Target_a ≠ Behavior_a。而DDQN則是改成令Target_a = Behavior_a時的target_net的Q值來計算Q_target。講更細一點，DDQN先從Behavior_net輸出的Q值中找出對應其maxQ值的Behavior_a，再來從target_net輸出的Q值中找出對應Target_a = Behavior_a的Q值來計算Q_target，而非過於樂觀的直接用target_net的maxQ。

根據以上的概念，DDQN與DQN的實作主要差別如下code：

```
q_next = self._behavior_net(state).gather(1, action.long())

q_next_action_val = self._behavior_net(next_state)
action_values = self._target_net(next_state).detach()
next_q_target = action_values.gather(1, q_next_action_val.max(1)[1].unsqueeze(1))
q_target = reward + (gamma * next_q_target * (1 - done))

loss = F.mse_loss(q_next, q_target)
```

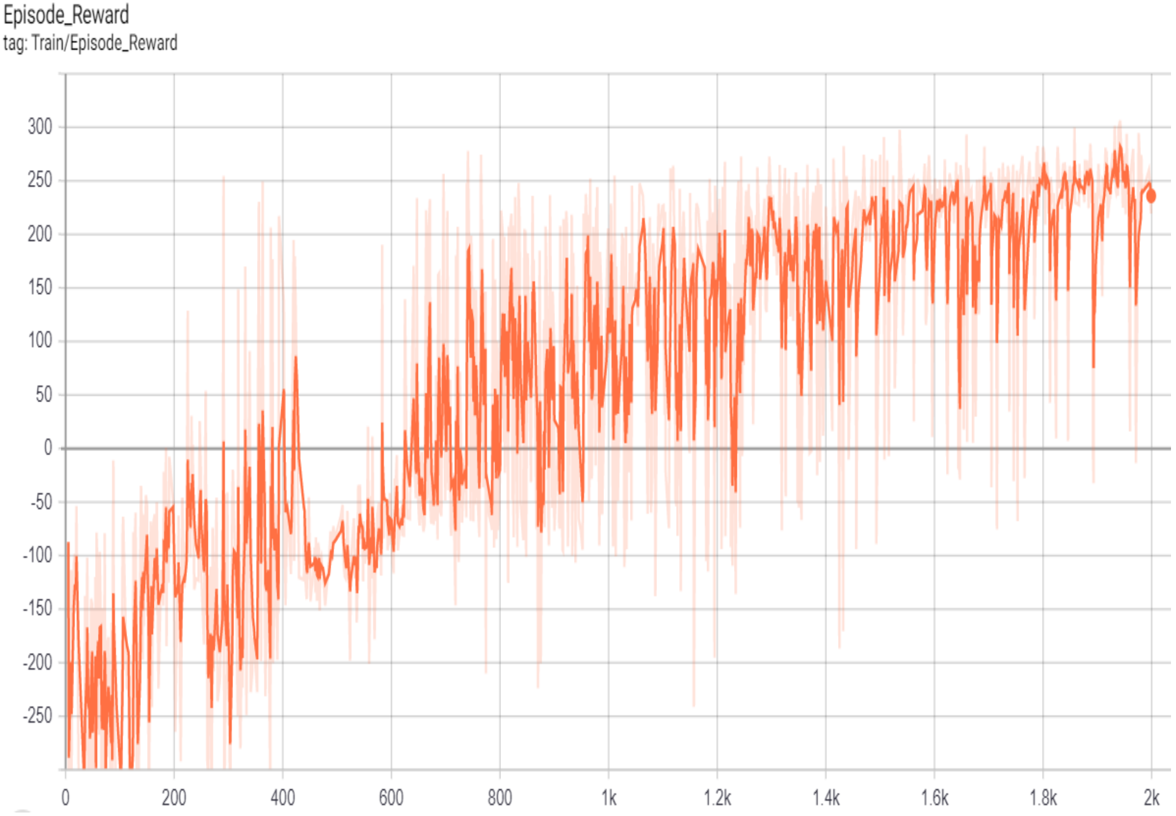
Implement and experiment on Double-DQN -- 2

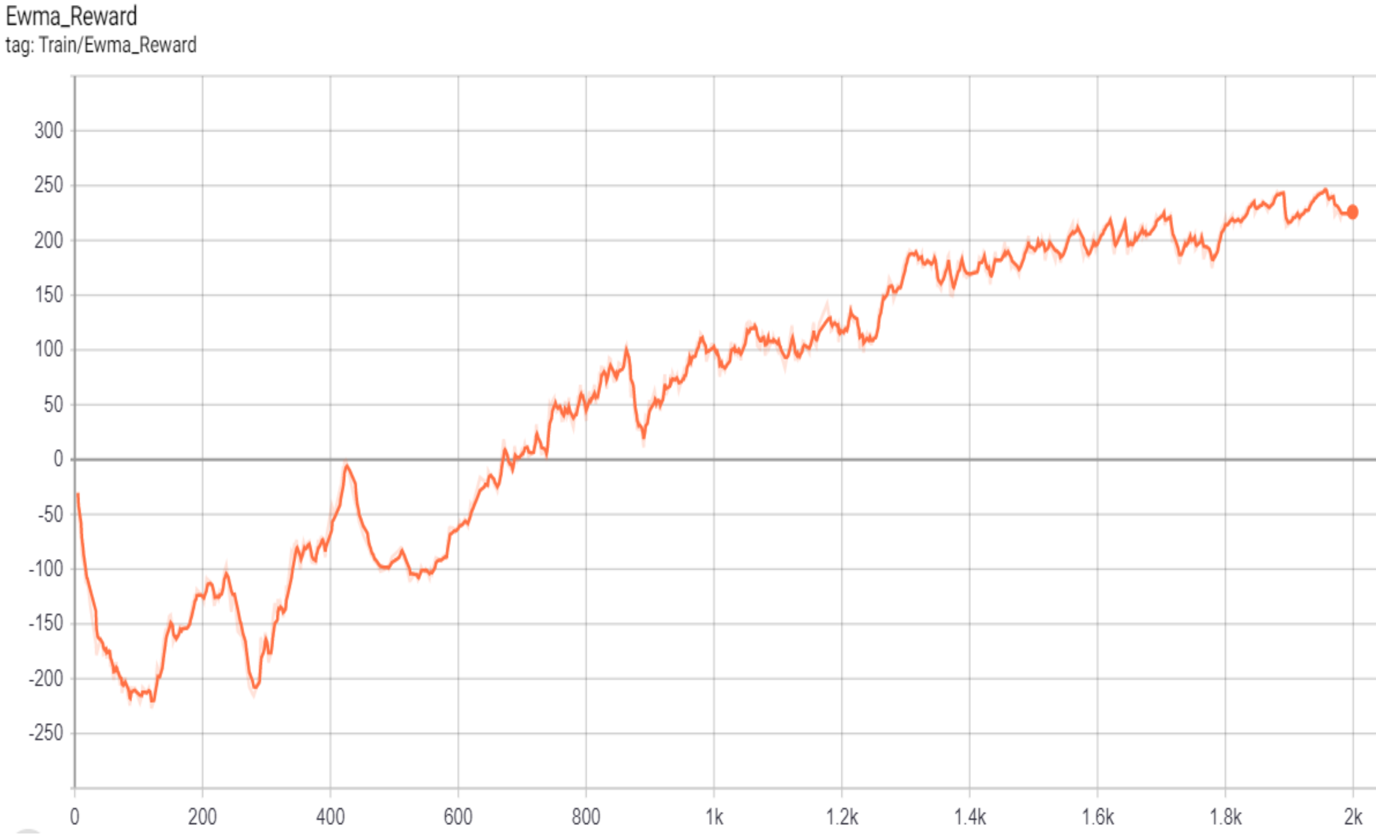
```python
## arguments ##
parser = argparse.ArgumentParser(description=__doc__)
parser.add_argument('-d', '--device', default='cuda')
parser.add_argument('-m', '--model', default='ddqn.pth')
parser.add_argument('--logdir', default='log/ddqn')
# train
parser.add_argument('--warmup', default=10000, type=int)
parser.add_argument('--episode', default=2000, type=int)
parser.add_argument('--capacity', default=50000, type=int)
parser.add_argument('--batch_size', default=128, type=int)
parser.add_argument('--lr', default=.0005, type=float)
parser.add_argument('--eps_decay', default=.99995, 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=100, type=int)

parser.add_argument('--TAU', default=.01, 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)
parser.add_argument('--test_epsilon', default=.001, type=float)
args = parser.parse_args()
```

左圖即為實作DDQN的參數，唯一與DQN不同的地方是我將eps_decay從0.9999改成0.99995，也就是讓exploration的時間再長一些。

Implement and experiment on Double-DQN -- 3






```
Start Testing
Average Reward 272.5608218021873
[241.81493741291874, 291.6041285411246, 251.07504000289157, 262.28866032306496,
279.0975552993562, 306.15055827596314, 267.63594241802883, 284.8556019810321,
295.7045759860239, 245.38121778146885]
```

來不及再試更多的參數，若以跟DQN近乎一樣的參數來比較的話，DDQN的平均reward=272，只比DQN的275差一點點。

The End