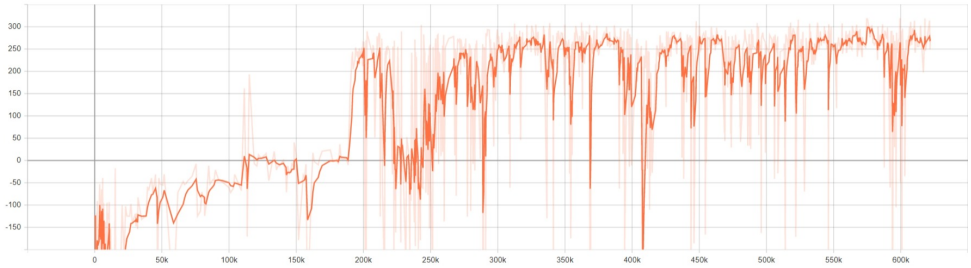
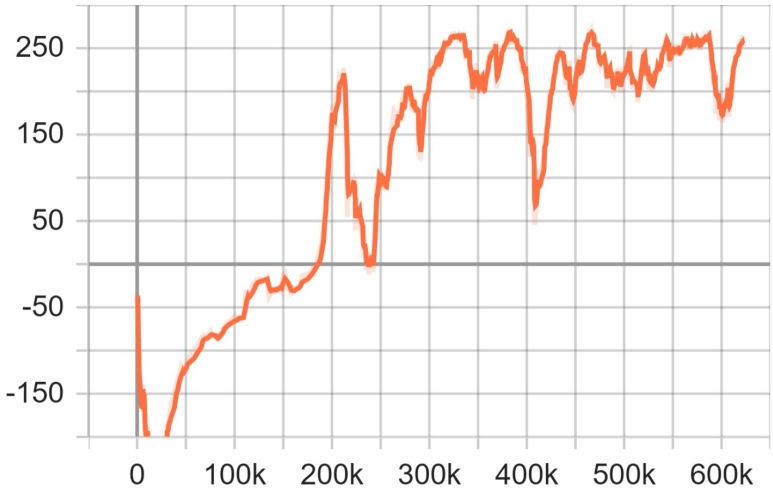
A Tensorboard Plot Shows Episode Rewards of at Least 800 Training Episodes in LunarLander-v2

The result of discrete version **LunarLander-v2** is from ***DQN***.
Here are the **episodic rewards**. The smoothing coefficient = 0.7.



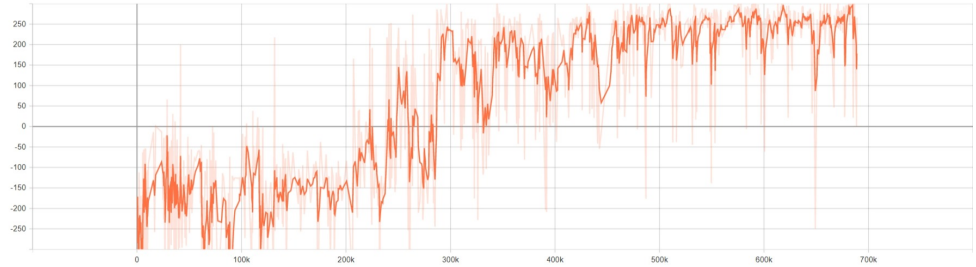
After around **300k** steps, the episodic rewards are at around 250 on average. The game episode terminates in positive but bad rewards, even negative rewards from time to time though, indicating that there might be too many redundant engine firings or some bad actions leading to unavoidable crushes.
The **EWMA rewards** are as follows (for reference):



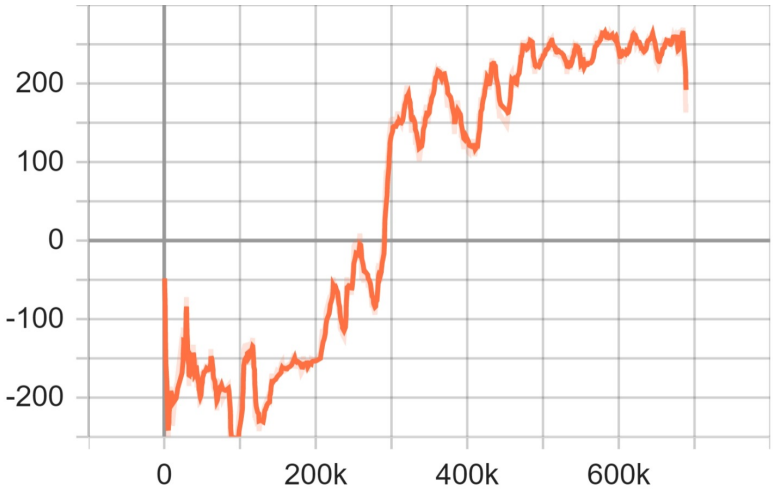
A Tensorboard Plot Shows Episode Rewards of at Least 800 Training Episodes

in LunarLanderContinuous-v2

The result of continuous version **LunarLanderContinuous-v2** is from ***DDPG***.
Here are the **episodic rewards**. The smoothing coefficient = 0.7.

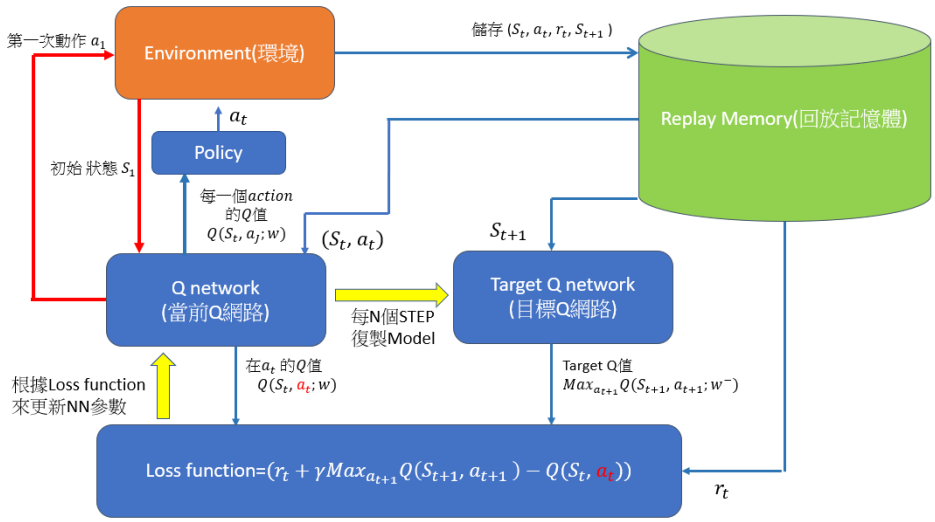


After around **450k** steps, the episodic rewards are at around 250 on average. The game episode terminates in positive but bad rewards, even negative rewards from time to time though, indicating that there might be too many redundant engine firings or some bad actions leading to unavoidable crushes.
However, one can see that the *negative-rewarded* episodes in the **DDPG** training are much fewer than those in the **DQN** training.
The **EWMA rewards** are as follows (for reference):



Describe Your Major Implementation of Both Algorithms in Detail

DQN



(image source: **https://medium.com/雞雞與兔兔的工程世界/機器學習-ml-note-reinforcement-learning-強化學習-dqn-實作atari-game-7f9185f833b0**

(https://medium.com/%E9%9B%9E%E9%9B%9E%E8%88%87%E5%85%94%E5%85%94%E7%9A%84%E5%B7%A5%E7%A8%8B%E4%B8%96%E7%95%8C/%E6%A9%9F%E5%99%A8%E5%AD%B8%E7%BF%92-ml-note-reinforcement-learning-%E5%BC%B7%E5%8C%96%E5%AD%B8%E7%BF%92-dqn-%E5%AF%A6%E4%BD%9Catari-game-7f9185f833b0))

To implement the ***DQN*** algorithms, one must construct 2 neural networks called **behavior network**(Q network) and **target network**(target Q network). These 2 networks have exactly the same architectures.

1. Network Architectures

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=300): # originally 32
        super().__init__()
        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):
        output = self.relu(self.layer1(x))
        output = self.relu(self.layer2(output))
        output = self.layer3(output)
        return output
```

```
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())
        ......
```

The inputs of both **behavior** and **target** networks are the **states** returned by the ***environment*** (the game LunarLander here). The states are composed of **8** observation values:

1. Horizontal Coordinate
2. Vertical Coordinate
3. Horizontal Speed
4. Vertical Speed
5. Angle
6. Angle Speed
7. If the first leg has contact
8. If the second leg has contact

And The outputs of both networks are the predicted **Q-values** $Q(s, a)$ of each **action** a under this **state** s. There are totally **4** actions available:

1. **0** = No operation
2. **1** = Fire Left Engine
3. **2** = Fire Main Engine
4. **3** = Fire Right Engine

Thus, the input dimensionality `state_dim` is **8** and the output dimensionality `action_dim` is **4** for both networks. So, the number of neurons in the output layer `layer3` is also **4**.

There are **2** hidden layers in my implementations, both have **300** hidden units and use ***ReLU*** as the activation function.

> Originally, the default setting is 32 hidden units for both the layers. However, it's a little too few, which leads to a mediocre result. So, I change it.

2. ϵ-greedy Policy for Action Selection

```
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    if random.random() < epsilon: # explore
        return action_space.sample() # choose the max one
    else: # exploit
        '''pick the one with the largest Q-value'''
        with torch.no_grad():
            s_in = torch.from_numpy(state).view(1, -1).to(self.device)
            return self._behavior_net(s_in).max(dim = 1)[1].item() # get the index
```

For action selection, we applied the ϵ***-greedy policy***. This is a method to balance the **exploration** and the **exploitation**. $\epsilon \in [0, 1]$.
Let a^* is the action with the **largest Q-value** given the state s, namely

$$a^* = \arg\max_{a \in \mathcal{A}} Q(s, a)$$

Then by the policy, we will **randomly** select the action based on the following probability:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{4} + (1 - \epsilon), & for\ a^* \\ \frac{\epsilon}{4}, & for\ the\ other\ 3\ actions \end{cases}$$

Typically, the value of ϵ is set to be small, so the policy **mostly** chooses to **exploit**, but sometimes try to **explore** randomly. Also, we have a ***warm-up*** at the beginning of the training, around **10000** episodes. In the warm-up session, the models can freely explore without any worries (no updates yet), so $\epsilon = 1$ for **exploration**. After that, ϵ will decay to **0.01** for mostly **exploitation** and sometimes **exploration**.

```
def train(args, env, agent, writer):
    ......
    for episode in range(args.episode):
        ......
        for t in itertools.count(start=1):
            # select action
            if total_steps < args.warmup: # 10000iter
                action = action_space.sample()
            else:
                action = agent.select_action(state, epsilon, action_space)
                epsilon = max(epsilon * args.eps_decay, args.eps_min)
                # eps_decay = 0.995, eps_min = 0.01
            ......
```

3. Update the Behavior Network

```
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_value = self._behavior_net(state).gather(dim = 1, index = action.long())
    with torch.no_grad():
       q_next = self._target_net(next_state).max(dim = 1)[0].view(-1, 1)
       q_target = reward + 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()
```

The update of the **behavior network** is based on the steps below:

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

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

For every time step, firstly we **sample** a *transition* $(\phi_j, a_j, r_j, \phi_{j+1})$ from the ***replay memory*** D, denoted `self._memory` in my code.

1. ϕ_j stands for the **current state**, denoted `state` above
2. a_j stands for the **action taken**, denoted `action` above
3. r_j stands for the **reward** from the action, denoted `reward` above
4. ϕ_{j+1} stands for the **next state** after the action, denoted `next_state` above

Then, the next few steps are typically **TD-learning**. The states `state` will be passed to the **behavior network** to predict the **Q-values of the 4 actions** given these states. The maximum ones will be selected and their Q-values will be stored in `q_value`. It is $Q(\phi_j, a_j; \theta)$ in the above figure, where θ is the parameters of the **behavior network**.

Then, we will feed the states at the next time steps, `next_state` to the **target network** (***not behavior network!!!***) to get the largest Q-values of the 4 actions, and store in `q_next`. This value is $\hat{Q}(\phi_{j+1}, a'; \theta^-)$ in the above figure, where a' is the next action given the next state, and θ^- is the parameters of the **target network**.

Next, together with the **discount factor** γ (denoted `gamma`, set to be **0.99**), we can compute the **target Q-value**, y_j, as:

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

The variable `done` indicates whether the episode terminates at the next step. If `1` then ends.
Finally, the **loss** for the TD-learning is just the **MSE loss** between the **target Q-value** and the **Q-value given by the behavior network**. To update the **behavior network** θ, we just perform a **gradient descent** step based on this loss, which is the last step in the figure above.
Note that `nn.utils.clip_grad_norm_()` is used here for **gradient clipping**, mitigating the problem of ***gradient explosion***.

```
def update(self, total_steps):
    if total_steps % self.freq == 0: #4 steps
        self._update_behavior_network(self.gamma) #0.99
    ......
```

Finally, the **behavior network** is updated **every** 4 **steps** in the default settings.

4. Update the Target Network

The target network is used to help calculate the target Q-value. Infrequent updates keep it relatively stable.

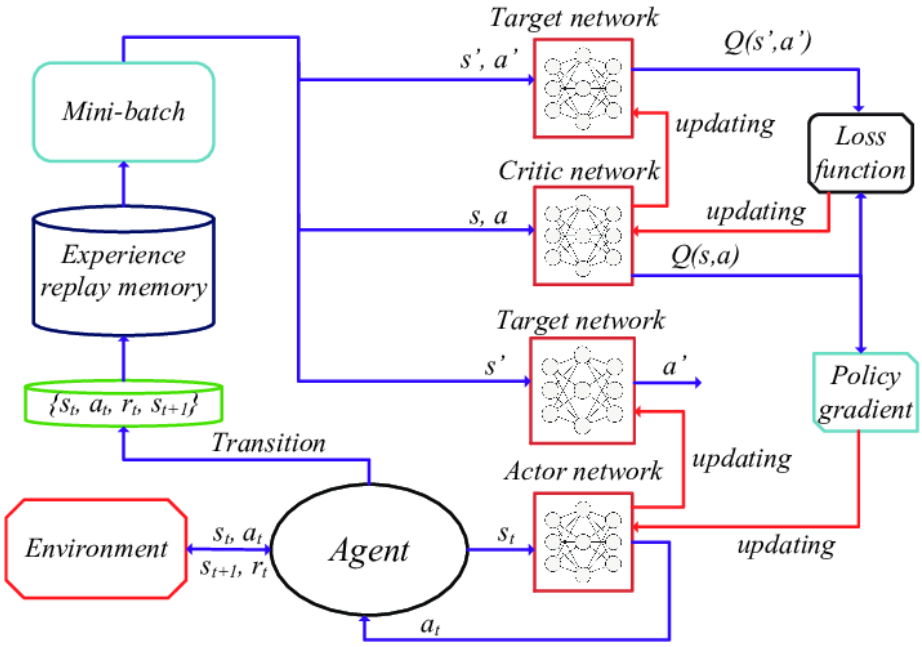
```
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

In DQN algorithm, we simply **copy** the parameters of the **behavior network** to the **target network** for updating every now and then.

```
def update(self, total_steps):
    ......
    if total_steps % self.target_freq == 0: #1000 steps
        self._update_target_network()
```

The **target network** is updated **every** 1000 **steps** in the default settings.

DDPG



(image source: **https://www.researchgate.net/figure/Structure-of-DDPG-algorithm-a-psd-p-and-critic-network-with-function-Qs-ad-Q_fig2_343005538**

(https://www.researchgate.net/figure/Structure-of-DDPG-algorithm-a-psd-p-and-critic-network-with-function-Qs-ad-Q_fig2_343005538).)

1. Network Architectures

In ***DDPG*** algorithm, there are 2 different types of networks: the **actor** and the **critic**. Moreover, Both of these 2 have the **behavior** networks and the **target** networks, respectively. Again, a behavior network and a target network in a pair have the same structure.

```
class DDPG:
    def __init__(self, args):
        '''actor和critic各自都有一組behavior網路和target網路'''
        # 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())
```

The Actor

```
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.relu(self.layer1(x))
        output = self.relu(self.layer2(output))
        output = self.tanh(self.layer3(output))
        return output
```

The inputs of both behavior θ^{μ} and target $\theta^{\mu'}$ networks of the **actor** are the **states** returned by the ***environment*** (the game LunarLander here). The states are composed of **8** observation values:

1. Horizontal Coordinate
2. Vertical Coordinate
3. Horizontal Speed
4. Vertical Speed
5. Angle
6. Angle Speed
7. If the first leg has contact
8. If the second leg has contact

And The outputs of both networks are the predicted **Q-value** $\mu(s|\theta^{\mu} \ or \ \theta^{\mu'})$ of the **expected action** a under this **state** s. The action space is **continuous** here, so we use the **expected value** μ. The action space is **2-dimensional**, indicating the action of the *main* engine and those of *the other engines*.

1. **Main Engine**: $[-1, 0]$ turn off, $[0, +1]$ throttle from 50% ~ 100% power.
2. **Other Engines**: $[-1, -\frac{1}{2}]$ fire left engine, $[-\frac{1}{2}, \frac{1}{2}]$ turn off, $[\frac{1}{2}, 1]$ fire right engine

Thus, the input dimensionality `state_dim` is **8** and the output dimensionality `action_dim` is **2** for both behavior and target networks of **the actor**.

There are **2** hidden layers in my implementations, which have **400** and **300** hidden units, respectively. Both of them use ***ReLU*** as the activation function.

The number of neurons in the output layer `layer3` is **2**. The activation function in the final layer is ***hyperbolic tangent***.

The Critic

```
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, 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)
```

Both the behavior θ^Q and target $\theta^{Q'}$ networks of the **critic** take the action a and the state s as the inputs, and output the **predicted Q-value** $Q(s, a|\theta^Q \ or \ \theta^{Q'})$.
The network structure of the **critic** consists of 2 parts. The **critic-head** part concatenates the **state** tensor and the **action** tensor, and pass them to the first hidden layer with **400** units (default settings in the sample code). The first hidden layer uses ***ReLU*** as the activation function. The second part consists of another hidden layer with **300** units (again, default settings) and the output layer. The second hidden layer here uses ***ReLU*** as the activation function as well. As for the output layer, since we are predicting the **Q-value** which is just a scalar, there will be only **1** neuron in the final layer.

2. Action Selection Policy

```
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise: # if exploration noises are applied
            actions = self._actor_net(torch.from_numpy(state)
            .view(1, -1).to(self.device)) + torch.from_numpy(self._action_noise.
            sample()).view(1, -1).to(self.device)
        else:
            actions = self._actor_net(torch.from_numpy(state).view(1, -1)
            .to(self.device))
    return actions.cpu().numpy().squeeze()
```

Here, since the action space is ***continuous***, ***exploration*** is done via adding a **normal noise** N_t in the ***DDPG*** algorithm.

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

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

In my code implementation, the exploration noise `self._action_noise` is sampled from the Gaussian $N(\mu = 0, \sigma = 0.1)$. The noise is also **2-dimensional**, the same as the action.

```
class GaussianNoise:
    def __init__(self, dim, mu=None, std=None):
        self.mu = mu if mu else np.zeros(dim) # default mean = 0
        self.std = std if std else np.ones(dim) * .1 # default std = 0.1

    def sample(self):
        return np.random.normal(self.mu, self.std)

class DDPG:
    def __init__(self, args):
        ......
        # action noise
        self._action_noise = GaussianNoise(dim=2)
        ......
```

3. Update Behavior Networks

Both the **actor** and the **critic** have their own ***behavior networks***. They have different ways to update. I will elaborate them separately in the following.

```
def _update_behavior_network(self, gamma):
        actor_net, critic_net, target_actor_net, target_critic_net = self._actor_n
        self._critic_net, self._target_actor_net, self._target_critic_net
        actor_opt, critic_opt = self._actor_opt, self._critic_opt

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

        ## update critic ##
        # critic loss
        ## TODO ##
        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 * (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()

        ## 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()
```

The behavior network for the critic

The **critic** networks take **state** s and **action** a as inputs, and return predicted **Q-value** $Q(s,a)$.

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

For every time step, firstly we **sample** a *transition* (s_j, a_j, r_j, s_{j+1}) from the ***replay memory*** R, denoted `self._memory` in my code.

1. s_j stands for the **current state**, denoted `state` above
2. a_j stands for the **action taken**, denoted `action` above
3. r_j stands for the **reward** from the action, denoted `reward` above
4. s_{j+1} stands for the **next state** after the action, denoted `next_state` above

Then, the next few steps are typically **TD-learning**. The states `state` will be passed to the **behavior network of the critic** `critic_net()` to predict the **Q-value of the**

actions given these states. The Q-values returned will be the **expectation** of the action-space since the space is continuous, and will be stored in `q_value`. It is $Q(s_i, a_i|\theta^Q)$ in the above figure, where θ^Q is the parameters of the **behavior network of the critic**.
Then, we will feed the states at the next time steps, `next_state` to the **target network of the actor** `target_actor_net()` to get the **next action** suggested, which will be stored in `a_next`. In the above figure, the next state is s_{t+1}, and `a_next` is $\mu'(s_{t+1}|\theta^{\mu'})$ where $\theta^{\mu'}$ is the parameters of the **target network of the actor**.
After that, we will use the next action suggested by the target network of the actor as well as the next state s_{t+1}, to compute the predicted **Q-value** via the **target network of the critic** `target_critic_net()`, and store in `q_next`. This value is $Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$ in the above figure, where $\mu'(s_{t+1}|\theta^{\mu'})$ is the next action, and $\theta^{Q'}$ is the parameters of the **target network of the critic**.
Next, together with the **discount factor** γ (denoted `gamma`, set to be **0.99**), we can compute the **target Q-value**, y_j, as:

$$y_j = \begin{cases} r_j, & if\ episode\ ends\ at\ step\ j+1 \\ r_j + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}), & otherwise \end{cases}$$

The variable `done` indicates whether the episode terminates at the next step. If `1` then ends.
Finally, the **loss** for the TD-learning is just the **MSE loss** between the **target Q-value** and the **Q-value given by the behavior network of the critic**. To update the **behavior network of the critic** θ^{μ}, we just perform a **gradient descent** step to minimize this loss, which is the last step in the figure above.

The behavior network for the actor

As for the **behavior network of the actor** `actor_net()`, it is updated using the ***sampled gradient***

Update the actor policy using the sampled gradient:

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

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

So first, we use the **behavior network for the actor** to determine the **actions** given **current states**. These states and actions are then fed to the **behavior network for the critic** to predict the **Q-values** and averaged over the whole mini-batch.
We want to maximize this average Q-value. Because we need to conduct the *backpropagation*, we convert it to be the **losses** via *flipping the sign*.
Finally, the **behavior networks** are updated **every steps after the warmup session** in the default settings.

```
def update(self):
    # update the behavior networks
    self._update_behavior_network(self.gamma) #0.99
    ......
```

4. Update Target Networks

```
@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_((1 - tau) * target.data + tau * behavior.data)
```

Both the **target networks** of the **actor** and the **critic** are updated as follows:
Update the target networks:

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

Here, we apply a '***soft***' update. That is, we set a small value τ, which is denoted `tau` and default to **0.005** in my code implementation. And then, the parameters of the **target networks** $\theta^{Q'}, \theta^{\mu'}$ are updated ***mostly*** $(1-\tau)$ from the previous state of itself $\theta^{Q'}, \theta^{\mu'}$ and ***only a little*** (τ) from the current state of the corresponding behavior networks θ^{Q}, θ^{μ}.
The **target networks** are updated **every steps after the warmup session** in the default settings as well.

```
def update(self):
    ......
    # update the target networks
    self._update_target_network(self._target_actor_net, self._actor_net,
                                self.tau)
    self._update_target_network(self._target_critic_net, self._critic_net,
                                self.tau)
```

Describe Differences Between Your Implementation of Both Algorithms in Detail.

1. In the ***DQN***, unlike what is written in the algorithm, I update the **behavior network** every **4** steps. (In the original algorithm, it is updated every steps).
2. In my implementation of both the ***DQN*** and ***DDPG***, I set up a **warm up** period, which is around **10000 episodes** at the **beginning** of the training. In this period, we will let the agent ***freely play and explore actions***, and store these transitions in the replay buffer for later use.
   We **will not update** models' parameters in the warm up period.
   In the theorectical algorithms, this step is omitted. However, this is needed in practice.
3. **The difference between DQN and DDPG**:
   DQN is a **value-based** RL method. The networks output the predicted Q-values for **each actions**. It is very useful in discrete action space, but almost ***infeasible*** if the action space is **continuous**, since we cannot have infinite number of neurons in the output layer.
   On the other hand, ***DDPG*** is a **policy-based** RL method. It uses 2 different networks: the **actor** and the **critic**. This allows them to select an action directly by learning a distribution. Thus, ***DDPG*** succeeds in **continuous** or **discrete but countably infinite** action space.

Describe Your Implementation and the Gradient of Actor Updating

(This part was elaborated in the introduction of major implementations. The below part is basically the same thing.)

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

So first, we use the **behavior network for the actor** `actor_net()` to determine the **actions** given **current states**. These states and actions are then fed to the **behavior network for the critic** `critic_net()` to predict the **Q-values** and averaged over the whole mini-batch. *We want to maximize this average Q-value*. Because we need to conduct the *backpropagation*, we convert it to be the **losses** via *flipping the sign*.
Let the θ^{μ}, θ^{Q} be the network parameters of the behavior network for the **actor** and the **critic**, respectively. Let s be the current state, then the action a is selected by the **actor**.

$$a = \mu(s|\theta^{\mu})$$

As we said earlier, we define the loss function to be the negative of Q-value predicted by the **critic**. The average operation is taken when these inputs are presented in mini-batches.

$$L = -Q(s, a|\theta^{Q})$$

The gradient is:

$$\begin{aligned}\nabla L_{\theta^{\mu}} &= \nabla L_a \nabla a_{\mu(s|\theta^{\mu})} \nabla \mu(s|\theta^{\mu})_{\theta^{\mu}} \\ &= \nabla L_a \times 1 \times \nabla \mu(s|\theta^{\mu})_{\theta^{\mu}} \\ &= -\frac{\nabla Q(s, a|\theta^{Q})}{\mu(s|\theta^{\mu})} \frac{\mu(s|\theta^{\mu})}{\theta_{\mu}}\end{aligned}$$

Describe Your Implementation and the Gradient of Critic Updating

(This part was elaborated in the introduction of major implementations. The below part is basically the same thing.)

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

## update critic ##
# critic loss
## TODO ##
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 * (1 - done)
criterion = nn.MSELoss()
critic_loss = criterion(q_value, q_target)
```

The **critic** networks take **state** s and **action** a as inputs, and return predicted **Q-value** $Q(s,a)$.

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: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

For every time step, firstly we **sample** a *transition* (s_j, a_j, r_j, s_{j+1}) from the ***replay memory*** R, denoted `self._memory` in my code.

1. s_j stands for the **current state**, denoted `state` above
2. a_j stands for the **action taken**, denoted `action` above
3. r_j stands for the **reward** from the action, denoted `reward` above
4. s_{j+1} stands for the **next state** after the action, denoted `next_state` above

Then, the next few steps are typically **TD-learning**. The states `state` will be passed to the **behavior network of the critic** `critic_net()` to predict the **Q-value of the actions** given these states. The Q-values returned will be the **expectation** of the action-space since the space is continuous, and will be stored in `q_value`. It is $Q(s_i, a_i|\theta^Q)$ in the above figure, where θ^Q is the parameters of the **behavior network of the critic**.

Then, we will feed the states at the next time steps, `next_state` to the **target network of the actor** `target_actor_net()` to get the **next action** suggested,

which will be stored in `a_next`. In the above figure, the next state is s_{t+1}, and `a_next` is $\mu'(s_{t+1}|\theta^{\mu'})$ where $\theta^{\mu'}$ is the parameters of the **target network of the actor**.
After that, we will use the next action suggested by the target network of the actor as well as the next state s_{t+1}, to compute the predicted **Q-value** via the **target network of the critic** `target_critic_net()`, and store in `q_next`. This value is $Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'})$ in the above figure, where $\mu'(s_{t+1}|\theta^{\mu'})$ is the next action, and $\theta^{Q'}$ is the parameters of the **target network of the critic**.
Next, together with the **discount factor** γ (denoted `gamma`, set to be **0.99**), we can compute the **target Q-value**, y_j, as:

$$y_j = \begin{cases} r_j, & if\ episode\ ends\ at\ step\ j+1 \\ r_j + \gamma Q'(s_{t+1}, \mu'(s_{t+1}|\theta^{\mu'})|\theta^{Q'}), & otherwise \end{cases}$$

The variable `done` indicates whether the episode terminates at the next step. If `1` then ends.
Finally, the **loss** for the TD-learning is just the **Mean Square Error loss** between the **target Q-value** and the **Q-value given by the behavior network of the critic**. To update the **behavior network of the critic** θ^{μ}, we just perform a **gradient descent** step to minimize this loss, which is the last step in the figure above.
The loss is

$$L = \frac{1}{N} \sum_i \left(Q_i^{target} - Q(s_i, a_i|\theta_Q) \right)^2$$

The gradient is thus ∇L_{θ^Q}.

Explain Effects of the Discount Factor

Let G_t be the **return**, i.e. the total discounted reward from time-step t with the **discount factor** γ

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

The value of receiving reward R is diminishing, meaning that the return G_t **values immediate reward above delayed reward**.

Explain Benefits of Epsilon-Greedy in Comparison to Greedy Action Selection

Instead of choosing the maximized option as in greedy action selection, the ϵ**-greedy strategy** balance the **exploration** and the **exploitation**. Most of the time, with a greater probability, the strategy still chooses the maximized option. Nevertheless, there are still chances that at times the strategy will randomly choose other actions, to **explore those unknown action which might elicit better return** G_t.

Explain the Necessity of the Target Network

Target Networks are used to give out the predicted **Q-values**. Since the target networks are updated ***every once a while***, their output values are **more stable** than those of the behavior networks. So, typically we use the **behavior networks** to generate **actions** given states, and predict the **Q-values** of these actions and states by the **target networks**, for **more stable training processes**.

Explain the Effect of Replay Buffer Size in Case of too Large or too Small

If the Replay buffer size is **too large**, the training processes are likely to be more ***stable*** because the agent can *"memorize"* more transitions. However, the cons are **longer training time**.
On the other hand, if the replay buffer size is **too small**, the agent can **only focus on the transitions from recent episodes**, which might lead to the ***overfitting*** problem.

BONUS: DDQN

DDQN stands for **double DQN**, meaning that **2** networks are used to determine the **Q-value** for the *next* action under DQN framework.
Recall that in **DQN**, the **TD-learning** steps are as follows: The states `state` will be passed to the **behavior network** to predict the **Q-values of the 4 actions** given these states. The **maximum** ones will be selected by the **behavior network**. These **Q-values for the next step** predicted by the **target network** will be used to compute **Q-target** *directly*.

```
q_value = self._behavior_net(state).gather(dim = 1, index = action.long())
with torch.no_grad():
    q_next = self._target_net(next_state).max(dim = 1)[0].view(-1, 1)
    q_target = reward + gamma * q_next * (1 - done)
```

However, in ***DDQN***, it is not the case. The first step is the same: passing the states to the **behavior network** and predicting the **Q-values of the 4 actions** given these states.
The difference starts from the following step: Instead of directly using the Q-values for the next step predicted by the target network, ***DDQN*** selects the **maximum** actions by the **behavior network**, but the **Q-values for the next step**, which will be used to compute **Q-target**, are derived from the **target network**.

```
q_value = self._behavior_net(state).gather(dim = 1, index = action.long())
with torch.no_grad():
    action_from_behavior = self._behavior_net(next_state).max(dim = 1)[1]
    .view(-1, 1)
    q_next = self._target_net(next_state).gather(dim = 1,
    index = action_from_behavior.long())
    q_target = reward + gamma * q_next * (1 - done)
```

In short, in **DDQN**, the **Q-values for the next step** which are used to calculate Q-target are ***not*** necessary the maximum values given by the **behavior network**, since it's given by the **target network** indirectly. The behavior network gives out the best action it thinks, but the Q-values for the next step come from the target network. It's a 2-step processes.

The ***benefit*** of **DDQN** compared to **DQN**, is that **DDQN** alleviates the **overestimation** problem existed in DQN. That is, DDQN won't be too optimistic.

Performances

DQN

```
Start Testing
Total Reward 1: 261.96
Total Reward 2: 294.68
Total Reward 3: 289.57
Total Reward 4: 287.40
Total Reward 5: 251.93
Total Reward 6: 281.87
Total Reward 7: 286.37
Total Reward 8: 283.36
Total Reward 9: 302.25
Total Reward 10: 269.67
Average Reward 280.9082116589326
```

DDPG

```
Start Testing
Total Reward 1: 305.19
Total Reward 2: 266.56
Total Reward 3: 294.37
Total Reward 4: 285.67
Total Reward 5: 312.65
Total Reward 6: 300.44
Total Reward 7: 278.61
Total Reward 8: 267.41
Total Reward 9: 267.80
Total Reward 10: 265.26
Average Reward 284.3952165589927
```

Both of the models achieved very good performances **(> 280)**.

The **DDPG** with continuous version is slightly better than the **DQN** with discrete version. (**284.40>280.91**)

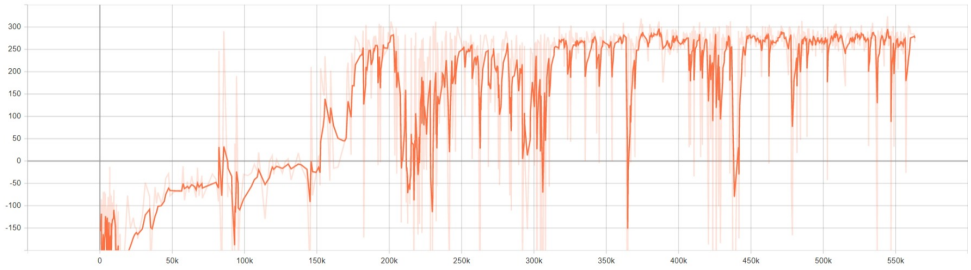
DDQN

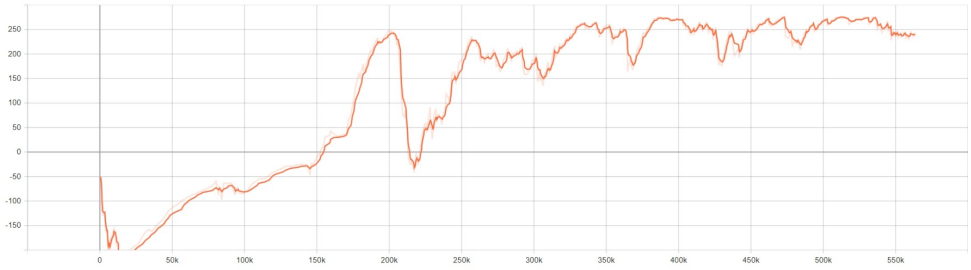
```
Start Testing
Total Reward 1: 287.36
Total Reward 2: 296.40
Total Reward 3: 297.65
Total Reward 4: 276.18
Total Reward 5: 268.15
Total Reward 6: 255.28
Total Reward 7: 295.73
Total Reward 8: 279.40
Total Reward 9: 302.86
Total Reward 10: 273.52
Average Reward 283.25207689709407
```

One can see that the average testing reward of **DDQN** is **283.25**, which is better than **DQN** (**280.91**) but slightly worse than **DDPG** (**284.40**). This shows that **DDQN does improve the performance** from DQN.

The **episodic rewards** and the **EWMA rewards** for the **DDQN** are as follows.

Both cuves are similar as those of **DQN**.


300
250
200
150
100
50
0
-50
-100
-150
0
50k
100k
150k
200k
250k
300k
350k
400k
450k
500k
550k


250
200
150
100
50
0
-50
-100
-150
0
50k
100k
150k
200k
250k
300k
350k
400k
450k
500k
550k