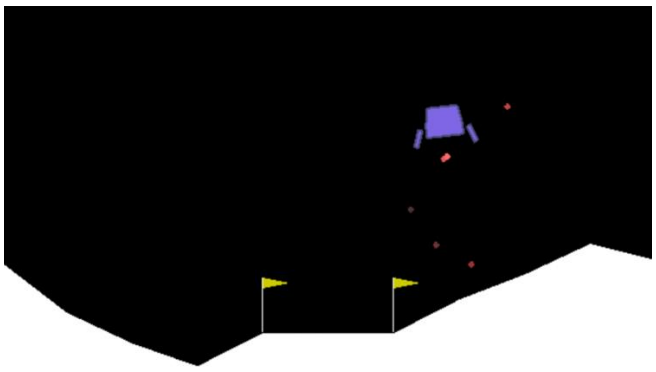
Summer 2022 Deep Learning

Report of Lab #6

Student name: 劉子齊 Jonathan
Student ID: 311605004

Part 1: Introduction

In this lab, our goal was to implement 2 deep reinforcement algorithms, which are the Deep Q-Learning Network and Deep Deterministic Policy Gradient, to get the highest score as we could in LunarLauncher-v2 and LunarLauncherContinuous-v2 correspondingly. The following figure is how LunarLauncher-v2 looked like.



In LunarLauncher-v2, there are 8 possible observations, which are the horizontal coordinate, the vertical coordinate, the horizontal speed, the vertical speed, the angle, the angle speed, the contact of left leg to the moon, and the contact of the right leg to the moon. We can take 4 kinds of actions on the spaceship, which are do nothing, fire left engine, fire main engine, and fire right, based on the 8 possible observations we may obtain.

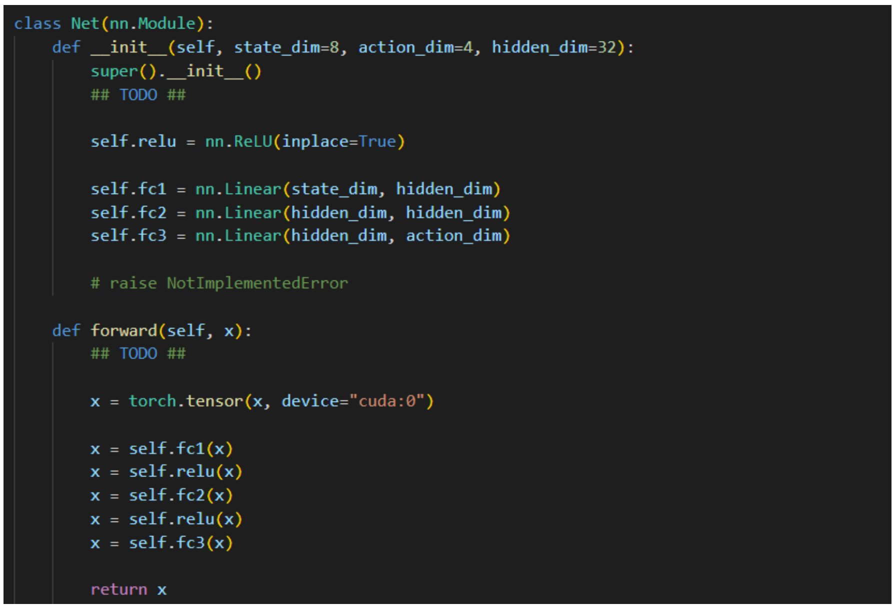
In LunarLauncherContinuous-v2, the 8 possible observations we can get are the same. However, the actions are different. The main engine can be controlled by a continuous value, when the value is between -1 to 0, the main engine is closed, and when the value is between 0 to +1, the main engine will throttle from 50% power to 100% power. The left and right engine basically work in the same way, but just on and off for their power, which when the value is between −1 to −0.5, the left engine will be fired, when the value is between +0.5 to +1, the right engine will be fired, otherwise, both engines are off.

For the Deep Q−Learning Network algorithm, we will work on the LunarLauncher−v2, and the output actions will be discrete actions. For the Deep Deterministic Policy Gradient, we will work on the LunarLauncherContinuous−v2, and the output actions will be continuous actions.

Part 2: Implementation Details

Part 2−A: Implementation of Deep Q−Learning Network

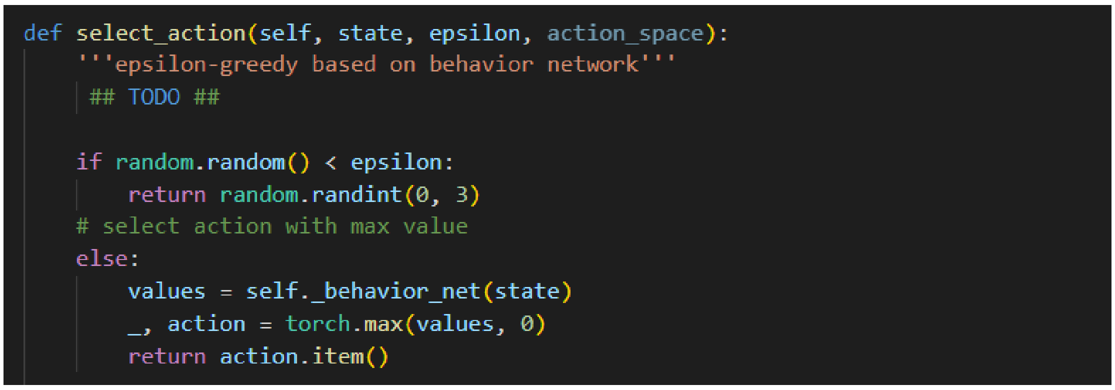
We can divide DQN into 2 parts, which are behavior network and the target network. The architecture of the two networks are basically the same. They are both formed with three fully connected layers following by an ReLU function as its activation function. The following figure is how I implemented the network architecture.



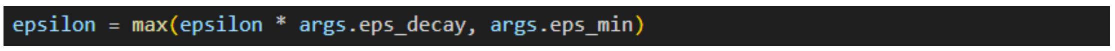
```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        ## TODO ##

        self.relu = nn.ReLU(inplace=True)

        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)

        # raise NotImplementedError

    def forward(self, x):
        ## TODO ##

        x = torch.tensor(x, device="cuda:0")

        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)

        return x
```

However, their purposes are different. Both of them are about the q value, but the behavior network gets the q value through the state and the action through each episode, and the target network estimates the target q value.

As the figure below, we can find that DQN either chose the action randomly or the one with the highest q value which obtained through the behavior network. Besides, the epsilon which was initially set as 1 would be multiplied by 0.995, which was the epsilon decay rate, until the lowest epsilon value, which was 0.1. These two processes are known as the epsilon greedy algorithm, and is one of the policies that DQN adopted.

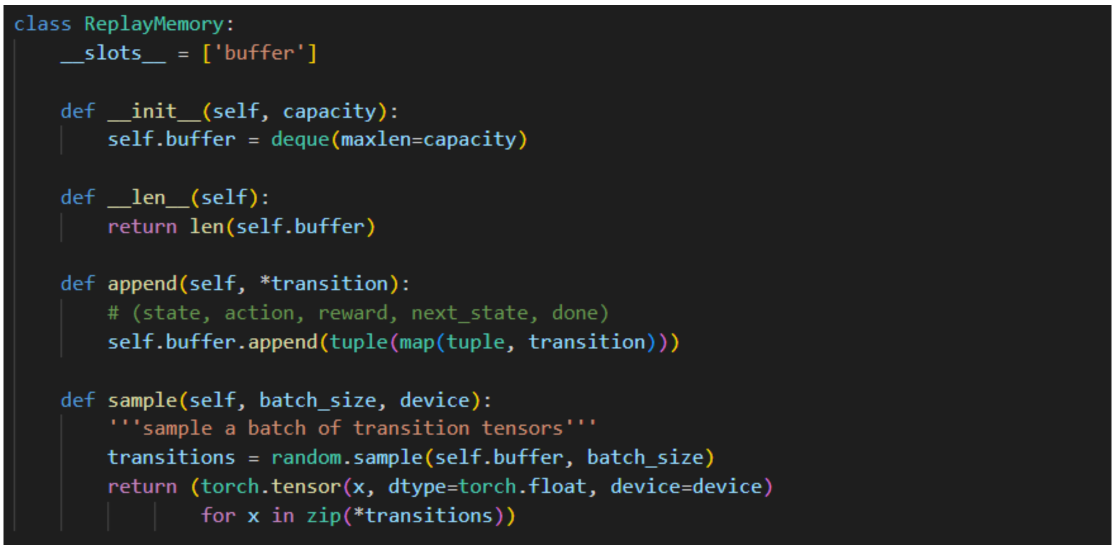


```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##

    if random.random() < epsilon:
        return random.randint(0, 3)
    # select action with max value
    else:
        values = self._behavior_net(state)
        _, action = torch.max(values, 0)
        return action.item()
```

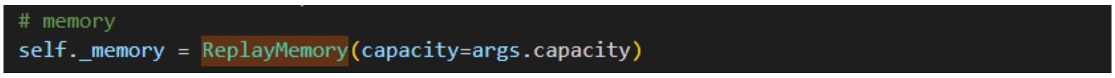


```python
epsilon = max(epsilon * args.eps_decay, args.eps_min)
```

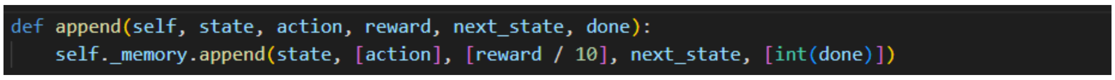
Afterwards, the chosen action would be fed into the environment then the reward, the status of the game, and the next step would be returned by the environment. This information would form a transition set and would be stored in the reply memory. Then the stored transitions in the reply memory would be sampled into several mini batch transitions by the model in the training process. The following figure is how I implemented the reply memory, and how they were made use of.



```python
class ReplayMemory:
    __slots__ = ['buffer']

    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, *transition):
        # (state, action, reward, next_state, done)
        self.buffer.append(tuple(map(tuple, transition)))

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

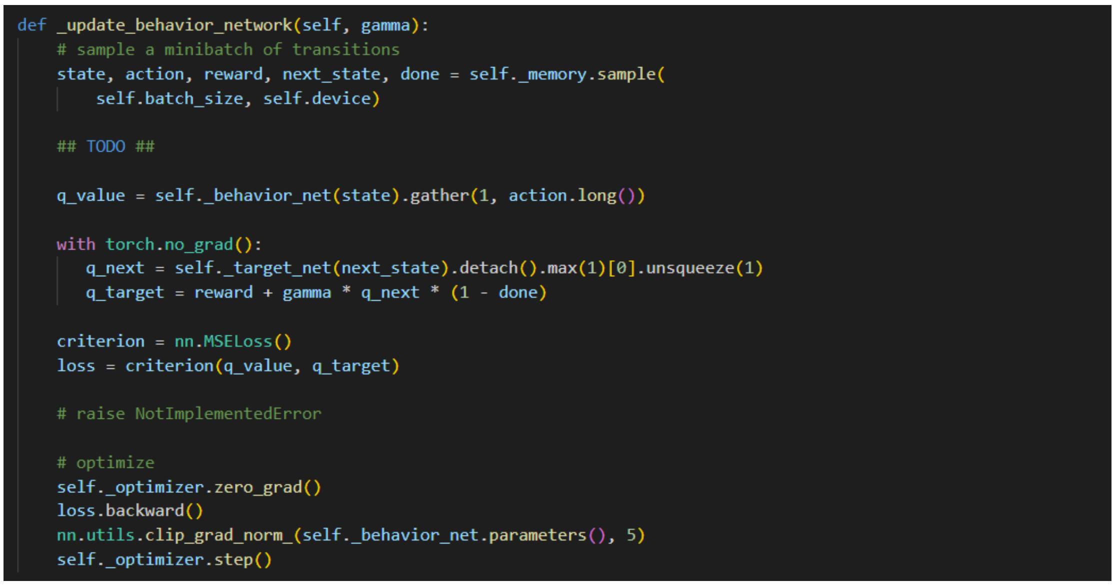


```python
# memory
self._memory = ReplayMemory(capacity=args.capacity)
```



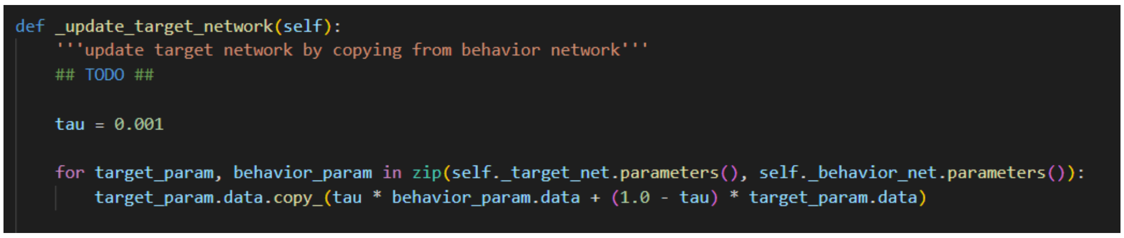
```python
def append(self, state, action, reward, next_state, done):
    self._memory.append(state, [action], [reward / 10], next_state, [int(done)])
```

With the mini batch transitions sampled, we would calculate all the q-value and get the highest one among all the sampled states by the behavior network mentioned above. Also, we would calculate the highest next q-value for the calculation of the target value through the target network. By comparing the q-value and the next q-value, we could get the loss through the MSE loss. Below is the implementation of the updating of the target network.



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

    with torch.no_grad():
        q_next = self._target_net(next_state).detach().max(1)[0].unsqueeze(1)
        q_target = reward + gamma * q_next * (1 - done)

    criterion = nn.MSELoss()
    loss = criterion(q_value, q_target)

    # raise NotImplementedError

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

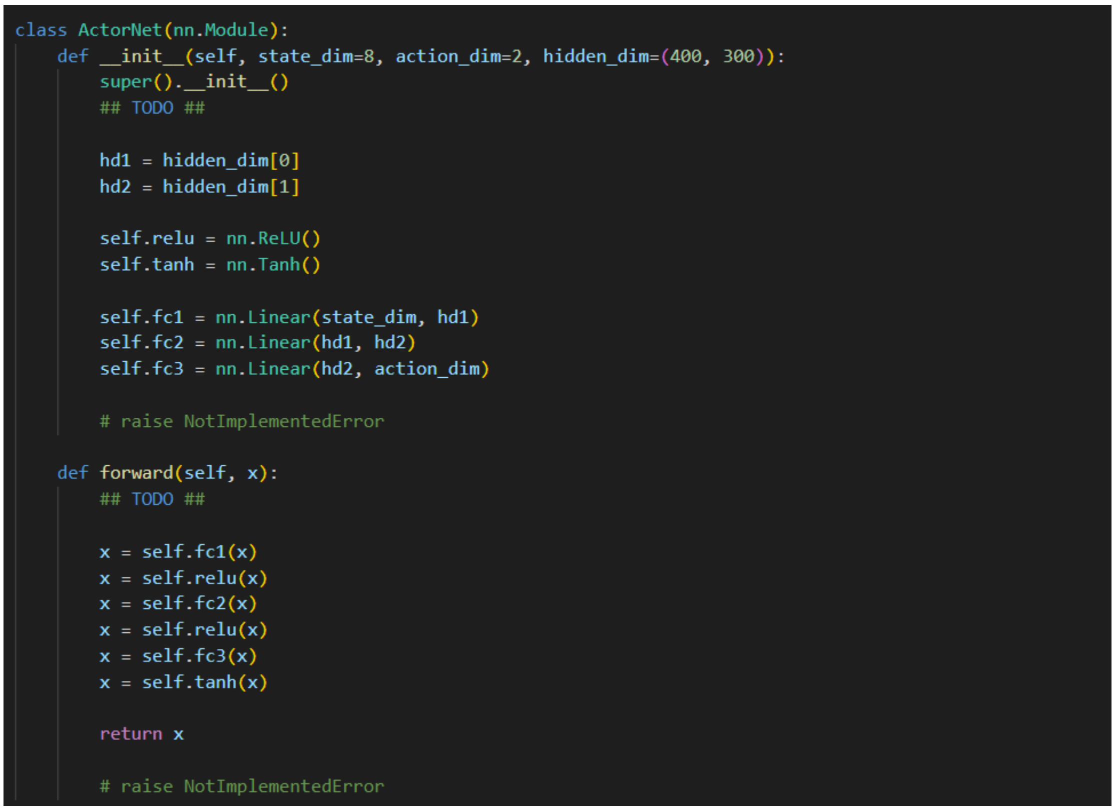
For the update of the target network, we applied the soft update. The reason why I applied soft update here is the characteristic of the soft update which brings more stability to the target network when updating. Below is the implementation of the updating of the target network.



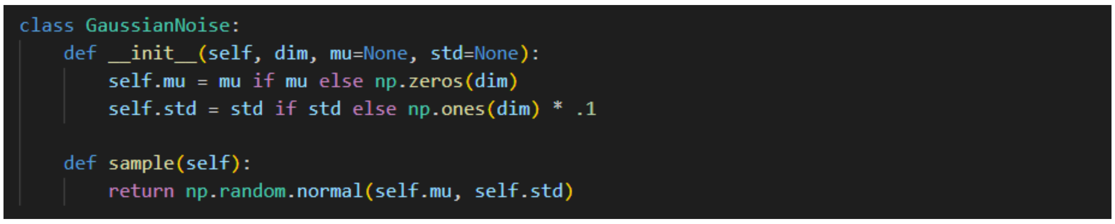
```python
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##

    tau = 0.001

    for target_param, behavior_param in zip(self._target_net.parameters(), self._behavior_net.parameters()):
        target_param.data.copy_(tau * behavior_param.data + (1.0 - tau) * target_param.data)
```

Part 2-B: Implementation of Deep Deterministic Policy Gradient

The basic structure of the DDPG is very similar to the DQN, but with slightly differences. Start with the actor network, the network will choose the action by adding the Gaussian noise to the action distribution which generated by the actor network. Below is the implementation of the actor network and the Gaussian noise function.

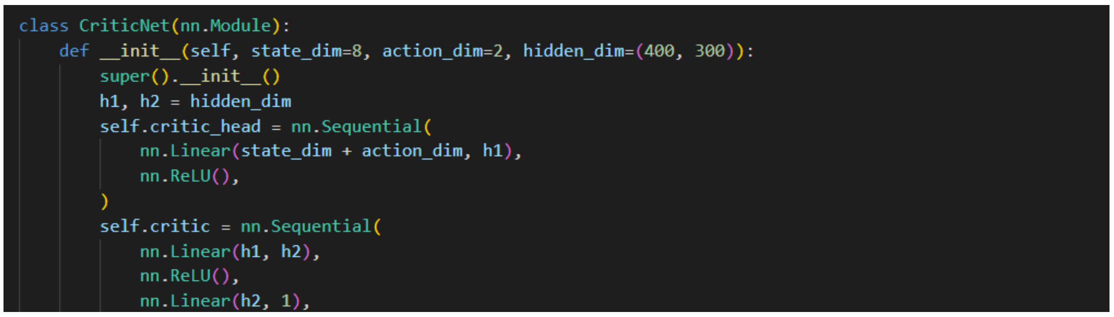


```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##

        hd1 = hidden_dim[0]
        hd2 = hidden_dim[1]

        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

        self.fc1 = nn.Linear(state_dim, hd1)
        self.fc2 = nn.Linear(hd1, hd2)
        self.fc3 = nn.Linear(hd2, action_dim)

        # raise NotImplementedError

    def forward(self, x):
        ## TODO ##

        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.tanh(x)

        return x

        # raise NotImplementedError
```



```python
class GaussianNoise:
    def __init__(self, dim, mu=None, std=None):
        self.mu = mu if mu else np.zeros(dim)
        self.std = std if std else np.ones(dim) * .1

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

For the critic network, it will estimate the q-value through the action and the state from the actor network. Besides, we will concatenate the state and the action together before feeding them to the critic network. Below is the implementation of the critic network.



```python
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)
```

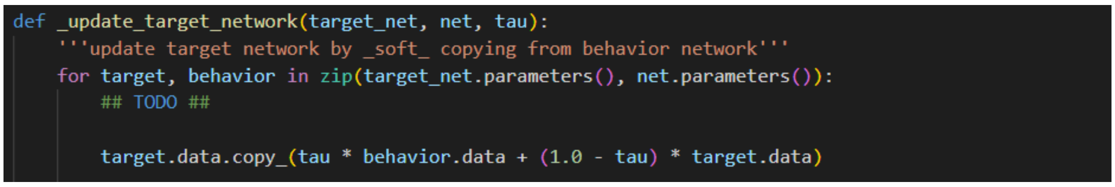
The whole training and updating process are actually quite similar to the ones of the DQN. We will first get the q-value through the states sampled from the reply memory by the behavior critic network. Then we will get the next q-value through feeding the next action provided by the target actor network and the next state into the target critic network. Finally, we will get the MSE loss and update the behavior critic network. The implementation of the updating of the behavior critic network is in the following figure.

```
def _update_behavior_network(self, gamma):
    actor_net, critic_net  = self._actor_net, self._critic_net
    target_actor_net, target_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)

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

Besides, there is also a super important element I the DDPG, which is the actor loss. The actor loss, which is actually the negative summation of the of all the q-values, can help to update the network by maximizing the expecting value of the q-value, below is the implementation of the actor loss.

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

Finally, for the update of the target network, we also applied the soft update. As I mentioned above, to bring more stability to the target network when updating, we adopted soft update here, and below is the implementation.



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

Part 2-C: Further Explanations

Part 2-C-1: Effects of the discount factor

In the 2 parts above, we can both find gamma in the behavior network updating function. Gamma here is the discount factor, which determines whether the model will depend on the future more or the past. When gamma is 1, it means that we will take the next q-value into the target value. When gamma is 0, we will simple update the model parameters by the reward.

Part 2-C-2: Benefits of epsilon-greedy in comparison to greedy action selection

Epsilon-greedy controls the strategy of action selection by the epsilon, which makes the model to get a probability of "epsilon" to find actions with higher rewards. Besides, it also makes the model to have a probability of "1-epsilon" to get the highest reward. However, the greedy action selection only makes decisions on the actions randomly, which means we can never be sure that we will get the highest reward from the action.

Part 2-C-3: Necessity of the target network

With the target network, the model can calculate the target by a network with better stability. We will not change the parameters in the target network in the same frequency as the behavior network, but with way lower frequency, and this brings the stability.

Part 2-C-4: Effect of replay buffer size in case of too large or too small

As I mentioned above, replay buffers are used to save transitions and the model will take those as its experiences. Hence, if the replay buffer size is too small, the model can only depend on the little experiences, which can make the model to learn faster, but may also lead to overfit. On the other side, if the relay buffer size is too large, the model will learn very slow, and may lead to underfit.

Part 3: Experimental Results & Discussions

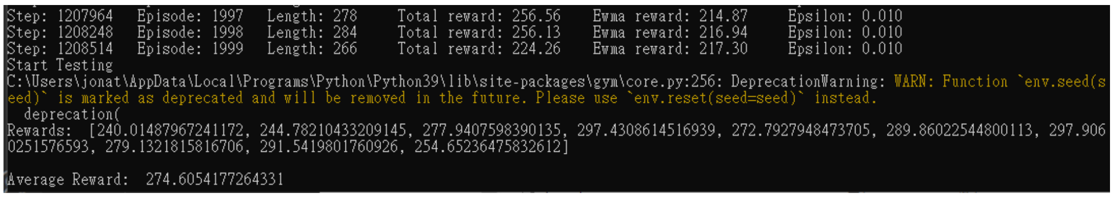
Part 3-A: Best Results

Part 3-A-1: DQN

Below is the capture, the curves of the best result I got for the DQN, which was with the average score of “274.6054”. The hyperparameters I adopted were as the following:

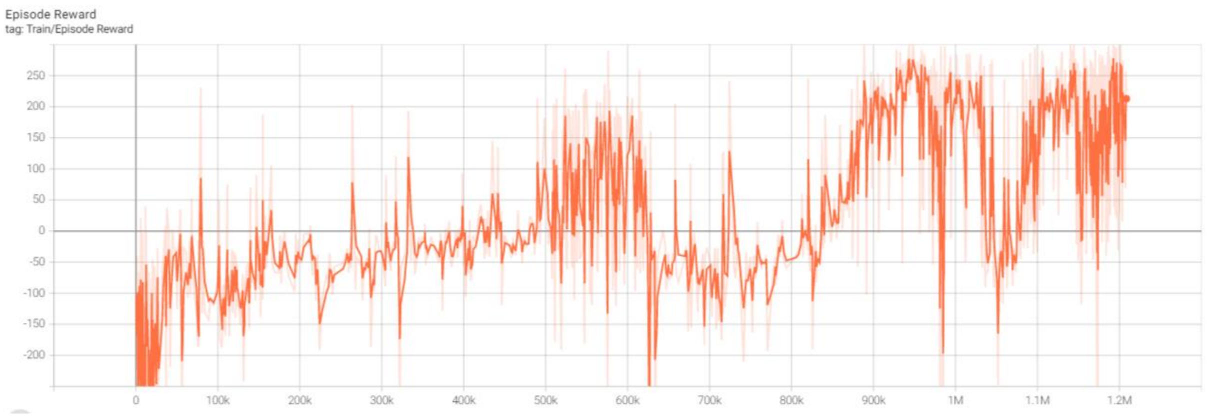
- Episode: 2000
- Learning Rate: 0.0005
- Batch Size: 128
- Capacity: 10000
- Optimizer: Adam
- Eps Decay: 0.995
- Eps Minimum: 0.01
- Gama: 0.99
- Frequency: 4
- Target Frequency: 4
- Test Epsilon: 0.001

- Screenshot of the Result of DQN

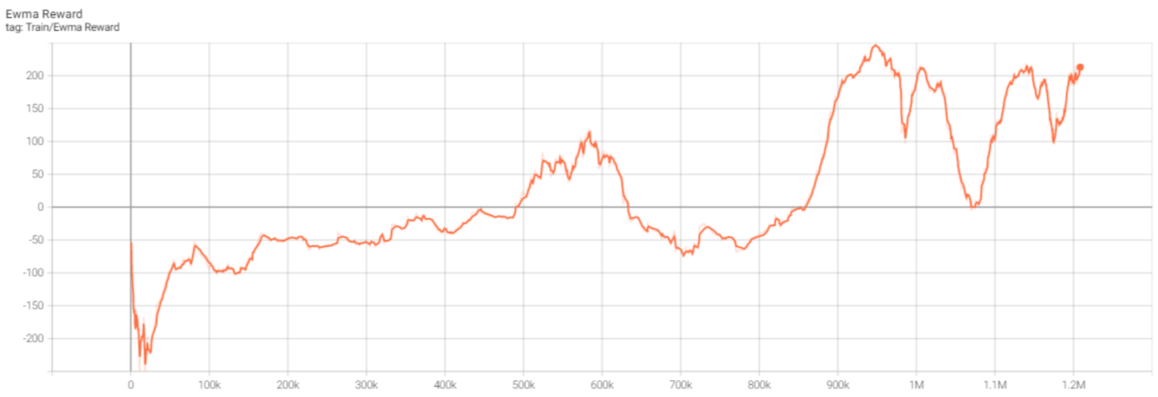


```
Step: 1207964   Episode: 1997   Length: 278     Total reward: 256.56    Ewma reward: 214.87     Epsilon: 0.010
Step: 1208248   Episode: 1998   Length: 284     Total reward: 256.13    Ewma reward: 216.94     Epsilon: 0.010
Step: 1208514   Episode: 1999   Length: 266     Total reward: 224.26    Ewma reward: 217.30     Epsilon: 0.010
Start Testing
C:\Users\jonat\AppData\Local\Programs\Python\Python39\lib\site-packages\gym\core.py:256: DeprecationWarning: WARN: Function `env.seed(s
eed)` is marked as deprecated and will be removed in the future. Please use `env.reset(seed=seed)` instead.
  deprecation(
Rewards:  [240.01487967241172, 244.78210433209145, 277.9407598390135, 297.4308614516939, 272.7927948473705, 289.86022544800113, 297.906
0251576593, 279.1321815816706, 291.5419801760926, 254.65236475832612]

Average Reward:  274.6054177264331
```

- Episode Reward Curve of DQN



- Episode Reward Curve of DQN

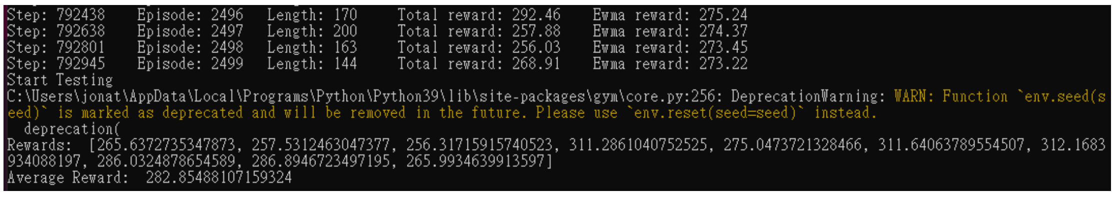


Part 3-A-2: DDPG

Below is the capture, the curves of the best result I got for the DDPG, which was with the average score of "282.8549". The hyperparameters I adopted were as the following:

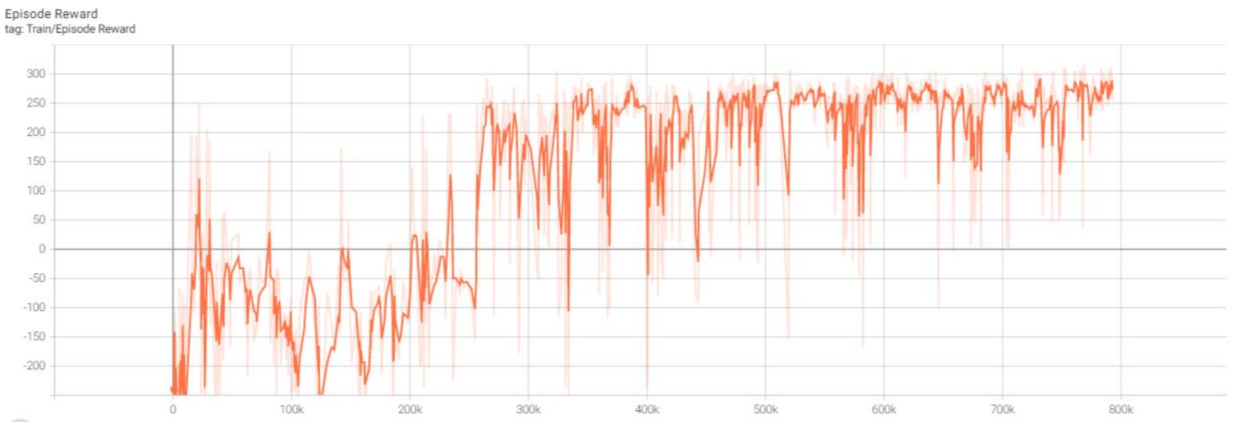
- Episode: 2500
- Learning Rate of Actor: 0.001
- Learning Rate of Critic: 0.001
- Batch Size: 64
- Capacity: 500000
- Optimizer: Adam
- Gama: 0.99
- Frequency: 4
- Target Frequency: 4
- Test Epsilon: 0.001
- Early Stop When AVG Reward = 300

- Screenshot of the Result of DDPG

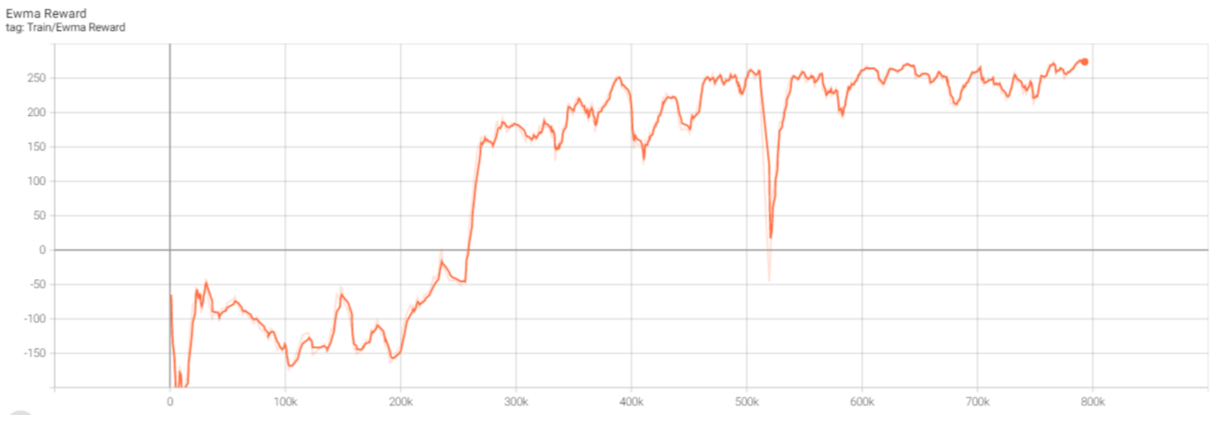


```
Step: 792438    Episode: 2496   Length: 170     Total reward: 292.46    Ewma reward: 275.24
Step: 792638    Episode: 2497   Length: 200     Total reward: 257.88    Ewma reward: 274.37
Step: 792801    Episode: 2498   Length: 163     Total reward: 256.03    Ewma reward: 273.45
Step: 792945    Episode: 2499   Length: 144     Total reward: 268.91    Ewma reward: 273.22
Start Testing
C:\Users\jonat\AppData\Local\Programs\Python\Python39\lib\site-packages\gym\core.py:256: DeprecationWarning: WARN: Function `env.seed(s
eed)` is marked as deprecated and will be removed in the future. Please use `env.reset(seed=seed)` instead.
  deprecation(
Rewards:  [265.6372735347873, 257.5312463047377, 256.31715915740523, 311.2861040752525, 275.0473721328466, 311.64063789554507, 312.1683
934088197, 286.0324878654589, 286.8946723497195, 265.9934639913597]
Average Reward:  282.85488107159324
```

➢ Episode Reward Curve of DDPG



➢ Episode Reward Curve of DDPG


Ewma Reward
tag: Train/Ewma Reward
100k
200k
300k
400k
500k
600k
700k
800k