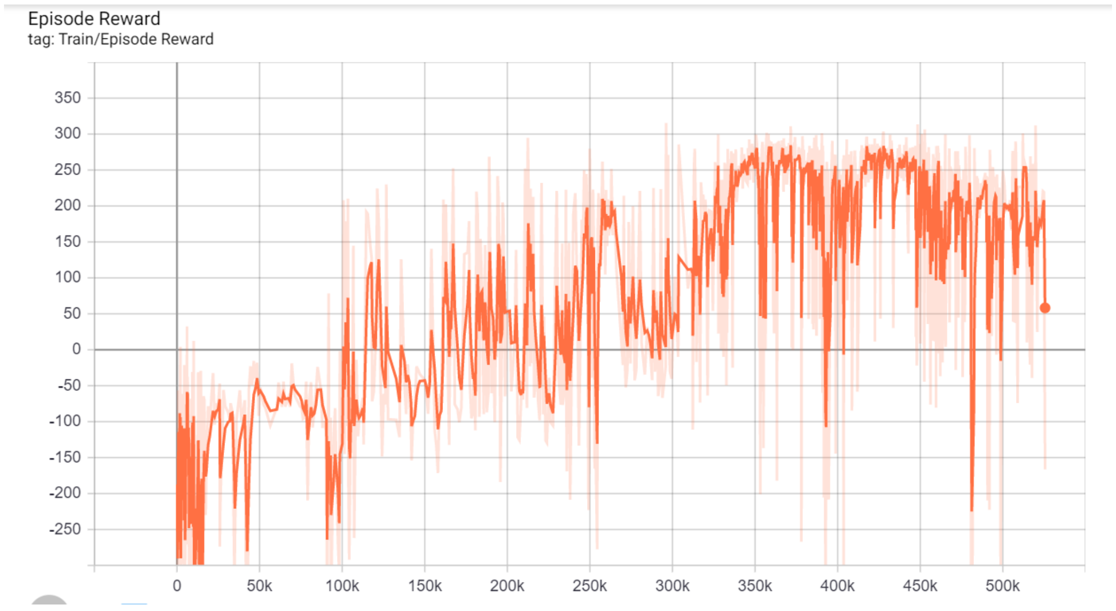
Lab8

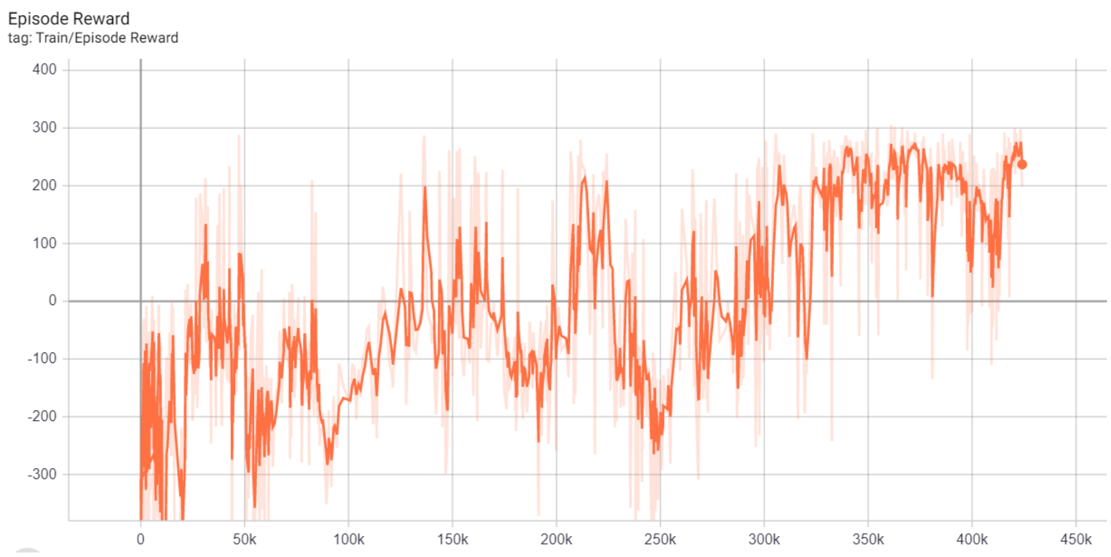
0856108 謝宗祐

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



1200 episodes

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



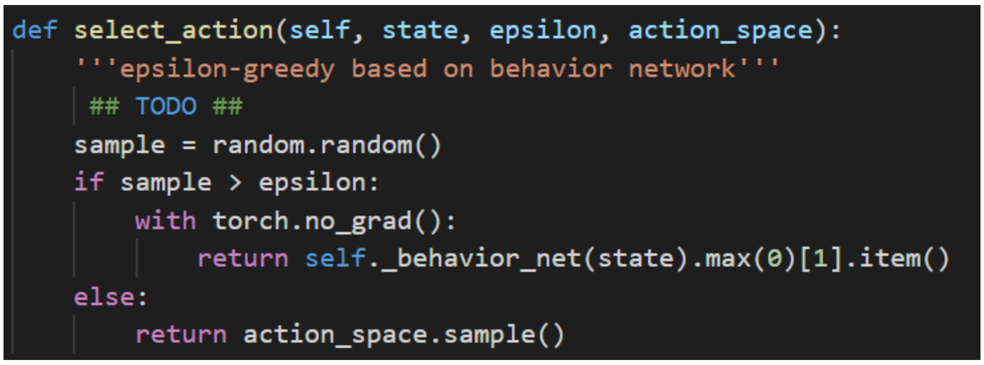
1200 episodes

- Describe your major implementation of both algorithms in detail

  Both network architectures are the same with the architecture introduced in the TA's guide, so I don't post the snapshots of the architecture. Focus on the algorithm implementation.

  1. DQN

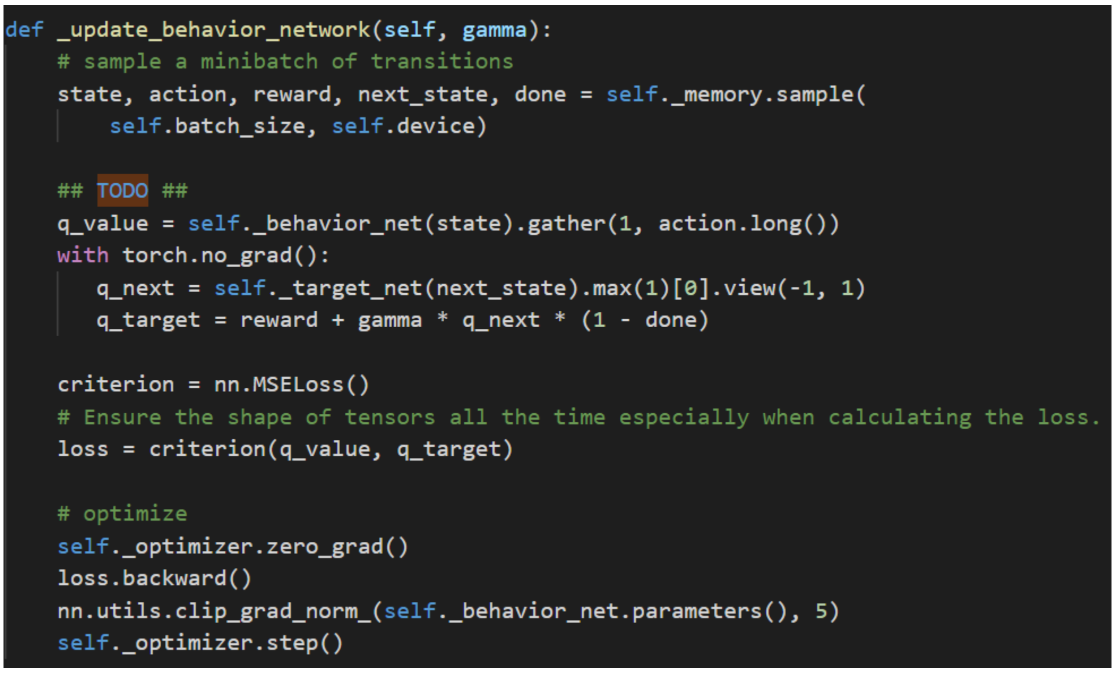
- Select action according to epsilon-greedy



```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
     ## TODO ##
    sample = random.random()
    if sample > epsilon:
        with torch.no_grad():
            return self._behavior_net(state).max(0)[1].item()
    else:
        return action_space.sample()
```

First sample a random number between 0 to 1, if sample is greater than epislon, do the greedy action selection, otherwise, select an action randomly. It means that we have the probability epislon for exploration.

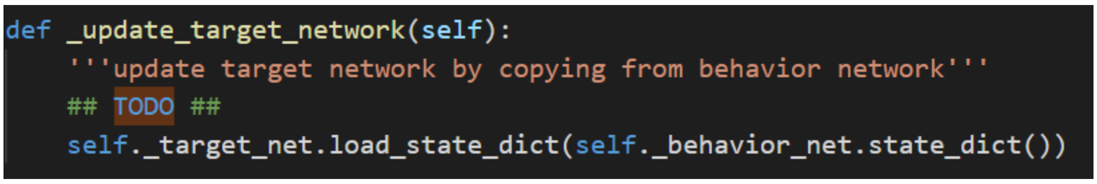
- Update behavior network and calculate loss function



```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).max(1)[0].view(-1, 1)
        q_target = reward + gamma * q_next * (1 - done)

    criterion = nn.MSELoss()
    # Ensure the shape of tensors all the time especially when calculating the loss.
    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()
```

Just follow the algorithm provided by TA to write down the code. I think it’s clear and no need to explain it, and choose MSELoss as the loss function to do the backpropagation.

- Update target network

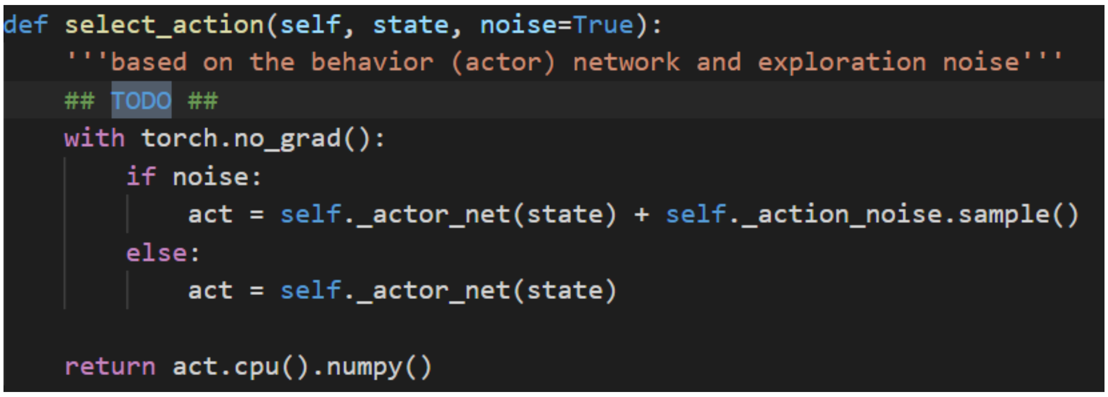


```python
def _update_target_network(self):
    '''update target network by copying from behavior network'''
    ## TODO ##
    self._target_net.load_state_dict(self._behavior_net.state_dict())
```

Simply copy the behavior network to target network.

2. DDPG

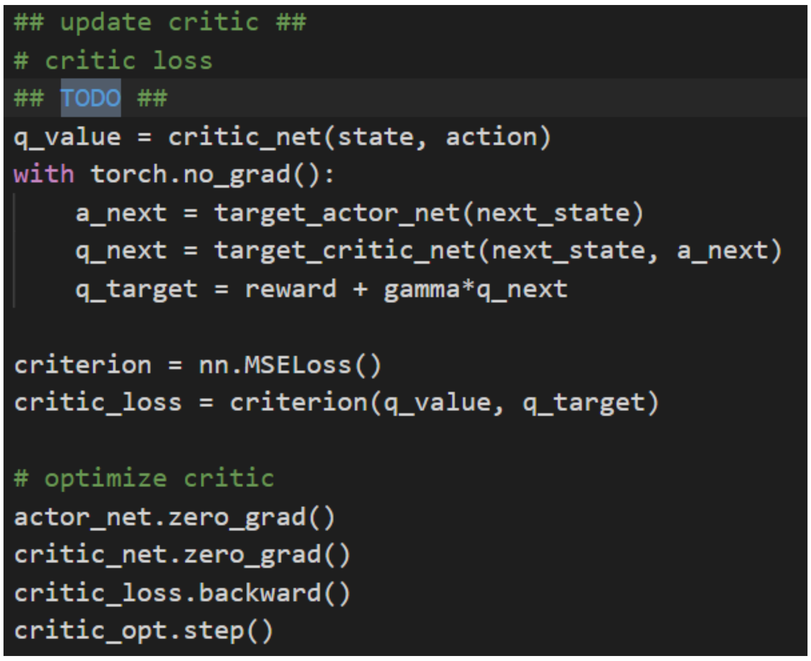
- Select action according to the actor and the exploration noise



```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    with torch.no_grad():
        if noise:
            act = self._actor_net(state) + self._action_noise.sample()
        else:
            act = self._actor_net(state)

    return act.cpu().numpy()
```

Add the exploration noise to the action in the training phase. When testing DDPG, action selection need NOT include the noise.

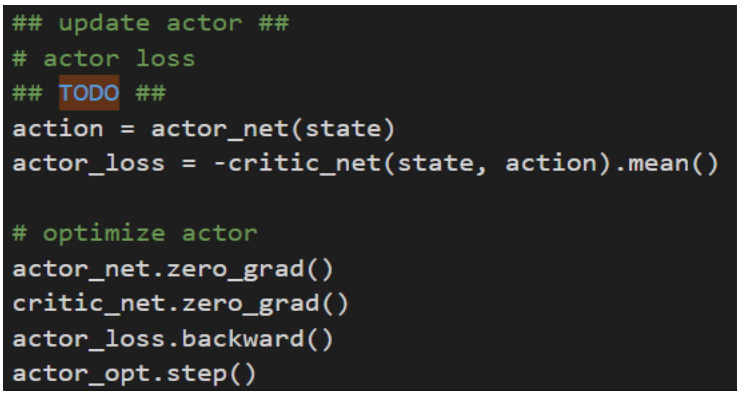
- Update critic



```python
## 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

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()
```

Follow the algorithm provied by TA to update the critic network. I think it’s clear and no need to explain it, too. And choose the MSELoss as the loss function.

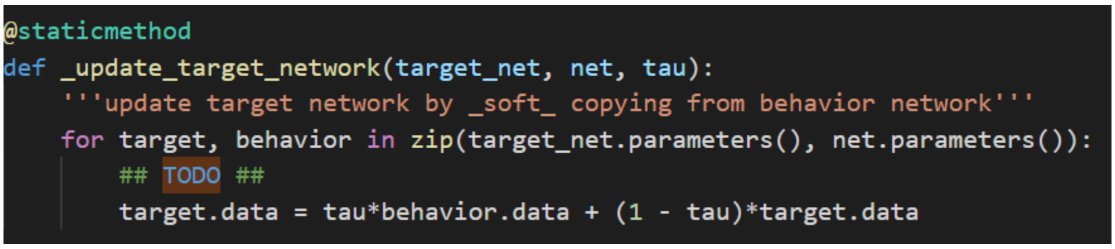
- Update actor



```
## 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()
```

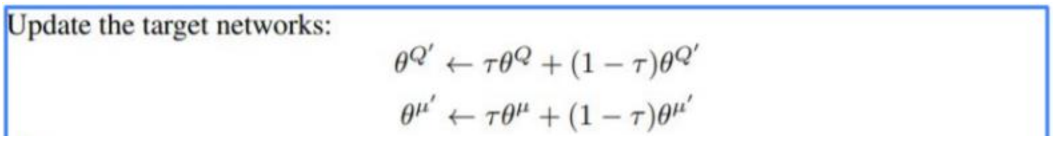
Different from the algorithm, and I will explain it in the next question.

- Update target network softly



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

Update the target network softly according to the following formula.



Update the target networks:

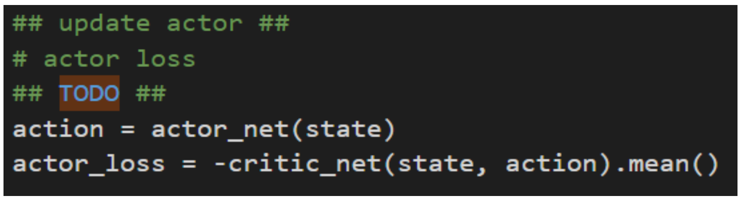
$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

■ Describe differences between your implementation and algorithms

1. DQN

   From the algorithm, it seems that we have to update the behavior network for every iterations, but my implementation only updates the behavior network for args.freq iterations.

2. DDPG



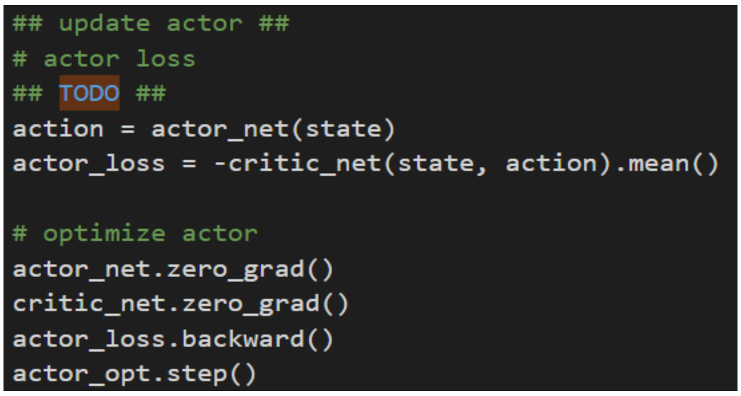
```
## update actor ##
# actor loss
## TODO ##
action = actor_net(state)
actor_loss = -critic_net(state, action).mean()
```

   Set the actor loss as the negative value of the critic. In order to minimize the loss, the actor needs to maximize the value of the critic, which means the actor has to choose the better action(Q increases) after the updating.

3. Both

   There is a procedure called warmup in my implementation. This procedure happens at the beginning of the training phase, and it is to play the game with random action selection. However, it won't update the network, just used to collect some experiences to the replay buffer.

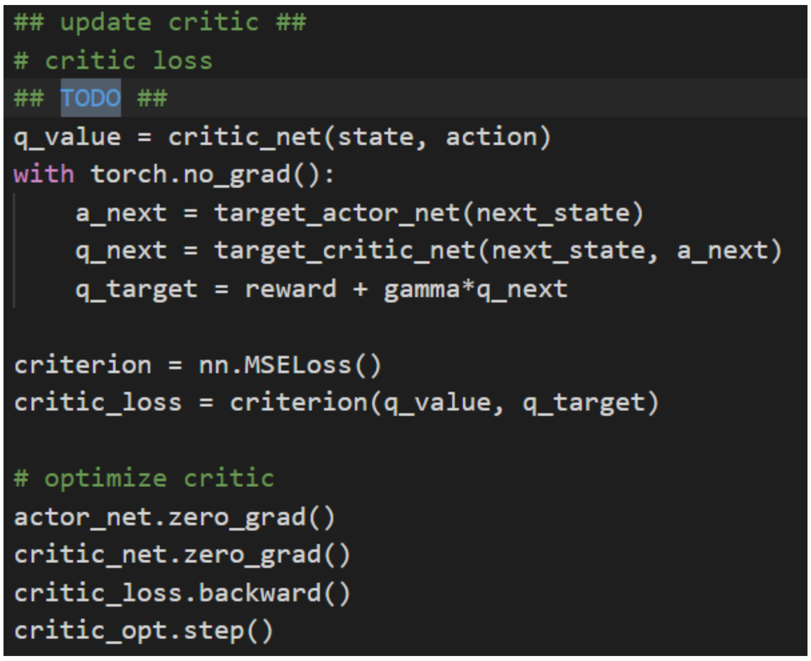
- Describe your implementation and the gradient of actor updating



```
## 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()
```

Introduced in the previous question. Fix the critic and update the actor.

- Describe your implementation and the gradient of critic updating



```
## 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

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()
```

Fix the actor and update the critic. Use the MSELoss of the behavior value and the target value to update the critic network.

- Explain effects of the discount factor
  Discount factor can reduce the effect of the future reward.

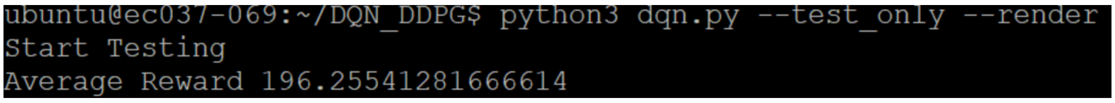
- Explain benefits of epsilon-greedy in comparison to greedy action selection
  Greedy action selection only selects the ‘’best’’ action, so there may be some states that we cannot reach. Epsilon-greedy has some probabilities to do the

random action, so it’s good for exploration and may do a better action.

- Explain the necessity of the target network
  If we only have one network, It’s so unstable because when updating the network, not only the Q(s, a) is changed, but the target Q(s’,a’) is changed. However, with the target network only updating periodically, the network becomes more stable and converges faster.

- Explain the effect of replay buffer size in case of too large or too small
  The size of the replay buffer can heavily hurt the speed of learning and quality of the resulting agent. If the replay buffer is too small, the replay buffer serves little to no purpose. If the replay buffer is too big, the batched samples are uncorrelated, but the agent will learn from the newest experience a long time after.

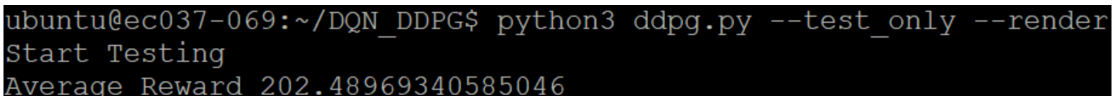
- [LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30



```
ubuntu@ec037-069:~/DQN_DDPG$ python3 dqn.py --test_only --render
Start Testing
Average Reward 196.25541281666614
```

  196.2554/30 = 6.54

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



```
ubuntu@ec037-069:~/DQN_DDPG$ python3 ddpg.py --test_only --render
Start Testing
Average Reward 202.48969340585046
```

  202.4896/30 = 6.75