DLP Homework 6

電控碩 0860077 王國倫

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

This is train 1200 episodes for DDQN in LunarLander-v2.

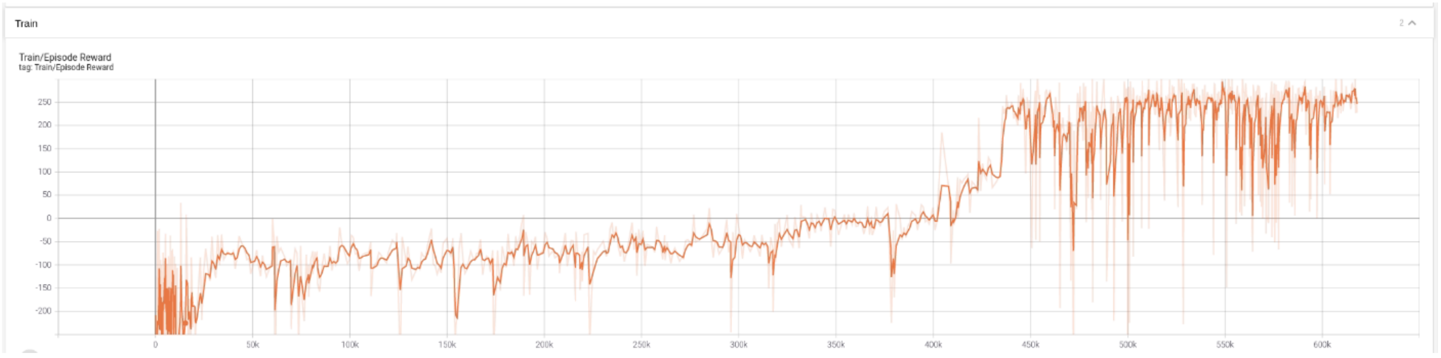


Fig.1 DDQN result

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

This is train 1200 episodes for DDPG in LunarLanderContinuous-v2.

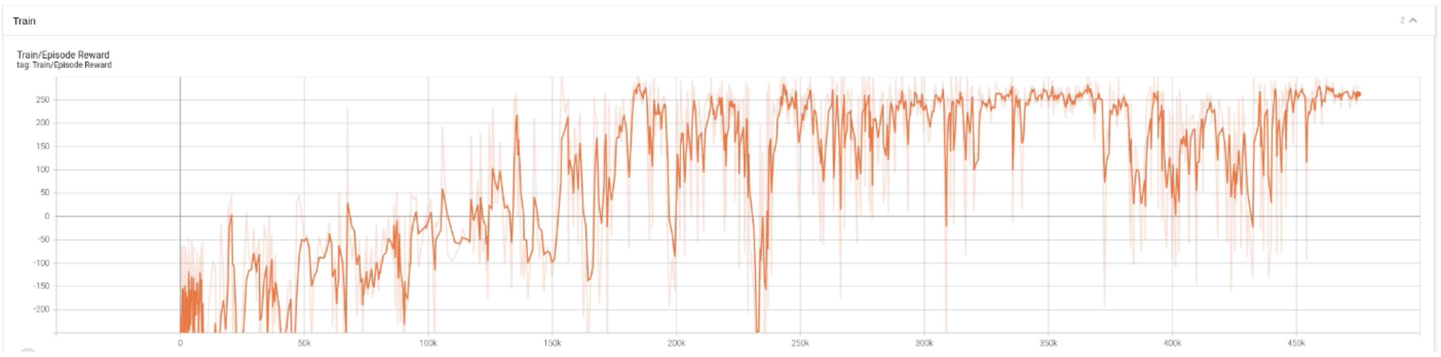


Fig.2 DDPG result

3. Describe your major implementation of both algorithms in detail

DQN

Q-table

In the DQN, I create a network to estimate Q-table, input observations from the environment, namely horizontal coordinate, vertical coordinate, horizontal speed, vertical speed, angle, angle speed, if first leg has contact and if second leg has contact. Predict four actions, including 0 (No-op), 1 (Fire left engine), 2 (Fire main engine) and 3 (Fire right engine). So the Network can simply implement by fully connected layer, and its architecture show below code.

```
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
        super().__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)
        self.relu = nn.ReLU()

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

Epsilon-Greedy

Exploration allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefit. Exploitation chooses the greedy action to get the most reward by exploiting the agent's current action-value estimates.
Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly.

$$\begin{cases} \underset{a}{argmax} Q(s,a) & P(1-\epsilon) \\ random & P(\epsilon) \end{cases}$$

Each action

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

    if random.random() < epsilon:
        return action_space.sample()
    else:
        with torch.no_grad():
            return torch.argmax(self._behavior_net(torch.from_numpy(state)\
            .view(1, -1).to(self.device)), dim=1).item()
```

Update network

Behavior Network

In the behavior Network, sample a minibatch from the replay memory, then use (s,a,r,s',doen) to calculate Q and Q next value, Q value is based on current state and action, Q next value is max value from next state and target network, take TD learning method to calculate MSE loss and optimize.

```
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)

        q_value = self._behavior_net(state).gather(dim=1,index=action.long())
        with torch.no_grad():
            q_next = torch.max(self._target_net(next_state), 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()
```

Target Network

In the target network, it will update with a fixed period times.

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

Testing

Finally, test trained agent to play the game and record reward, then calculate its mean and output this value.

```
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 = []
    for n_episode, seed in enumerate(seeds):
        total_reward = 0
        env.seed(seed)
        state = env.reset()

        for t in itertools.count(start=1):

            env.render()

            #select action
            action = agent.select_action(state, epsilon, action_space)

            #excute action
            next_state, reward, done, _ = env.step(action)

            state = next_state
            total_reward += reward

            if done:
                writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                print("total reward : {0:.2f}".format(total_reward))
                rewards.append(total_reward)
                break

    print('Average Reward', np.mean(rewards))
    env.close()
```

DDPG

For the continuous case, the DQN is hard to handle this problem, beacause the Q learning is value-based method, the objective function show in Fig.3, and this action is continuous, DQN can not learn Q value in each state, the value-based way is difficult handle continuous problem, but the policy-based can solve this problem, because its objcetive function(Fig.4), directly maximize the expectation of the sum of rewards under the current policy, without involving calculation of a specific action, so the policy-based method can handle continuous actions.

$$\min_{a}(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w))^2$$

Fig.3 Value-based objective function

$$\max_{\pi_\theta} \sum_{s \in S} d_{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(s, a) r$$

Fig.4 Policy-based objective function

The DDPG combined value-based and policy-based method(actor+critic) show Fig.5 , and it is model-free and off-policy algorithm for learning continous actions, compare with DQN, the DQN only handle low dimension and discrete action space.

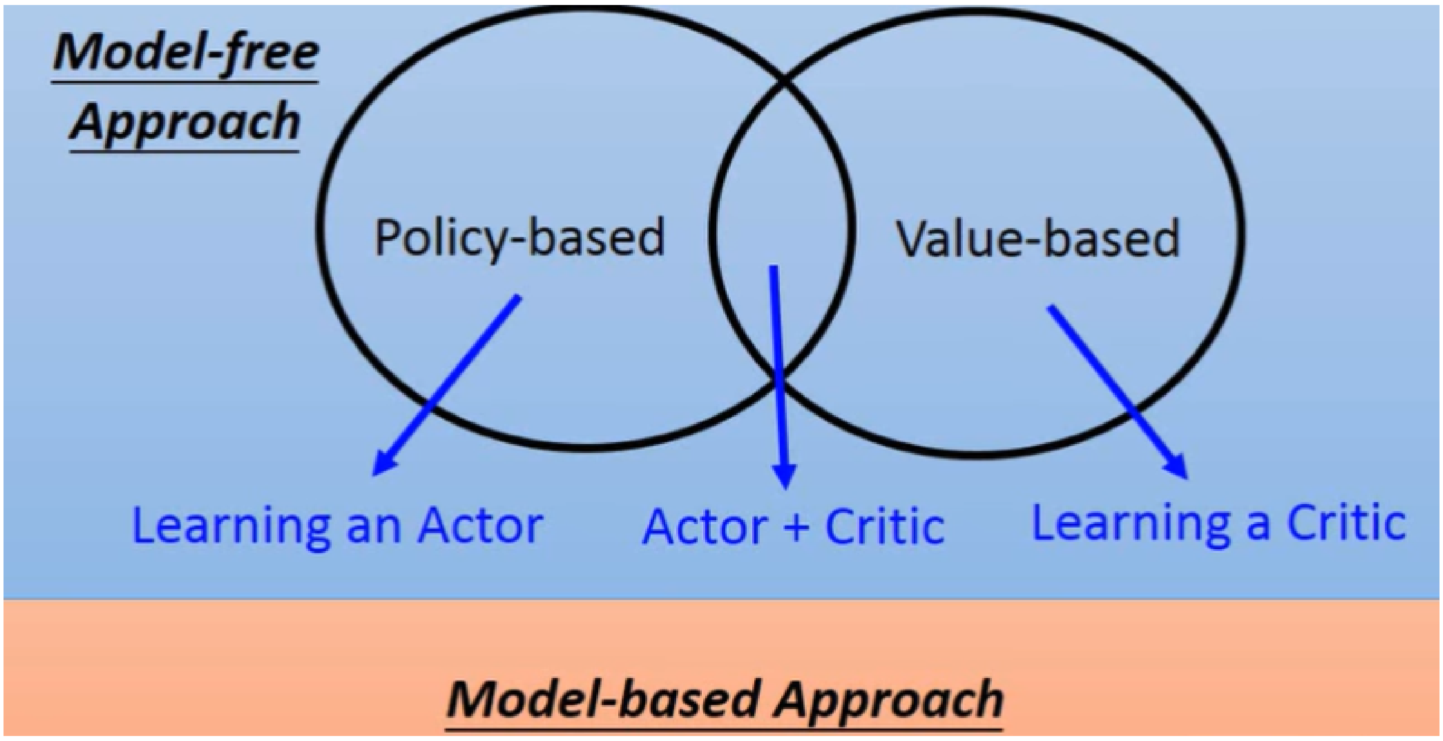


Fig.5 Reinforcement learning approaches

Actor-Critic

In the DDPG, it will have actor and critic, actor in charge of output action, and critic will evaluate this action return a value.
The ActorNet will input state, same as LunarLander-v2, output two action, including main engine(1~-1) and left-right-engine(1~-1).
The CriticNet will estimate Q(s,a), and output a value.

```
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim[0])
        self.fc2 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.fc3 = nn.Linear(hidden_dim[1], action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        out = self.tanh(self.fc3(x))
        return out


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)
```

Select action

In the training, the select action will add Gaussian noise import its randomness, make sure have fully explore the environment.

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

Update network

Critic

In CriticNet, sample minibatch from replay memory, use (s,a,r,s',done) to calculate Q and Q next value, Q value based on state and action, and got next action by used target actor network, then input next state and output action to targer critic network, calculate Q next, finally, use TD learning method to calculate MSE loss and optimize.

Actor

In the ActorNet, we hope its Q(s,a) more higher more better, then defined its loss is expectation of Q(s,a).

Update the actor policy using the sampled gradient:

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

Fig.6 Update actor network formula

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

        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

        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()
```

TargetNet

According to Fig.7, we use soft update method to ensure that the parameters can be updated slowly, and achieve the effect of DQN regularly copying parameters.

Update the target networks:

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

Fig.7 Update target network formula

```
    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()):
            target.data.copy_(target.data * (1 - tau) + behavior.data * tau)
```

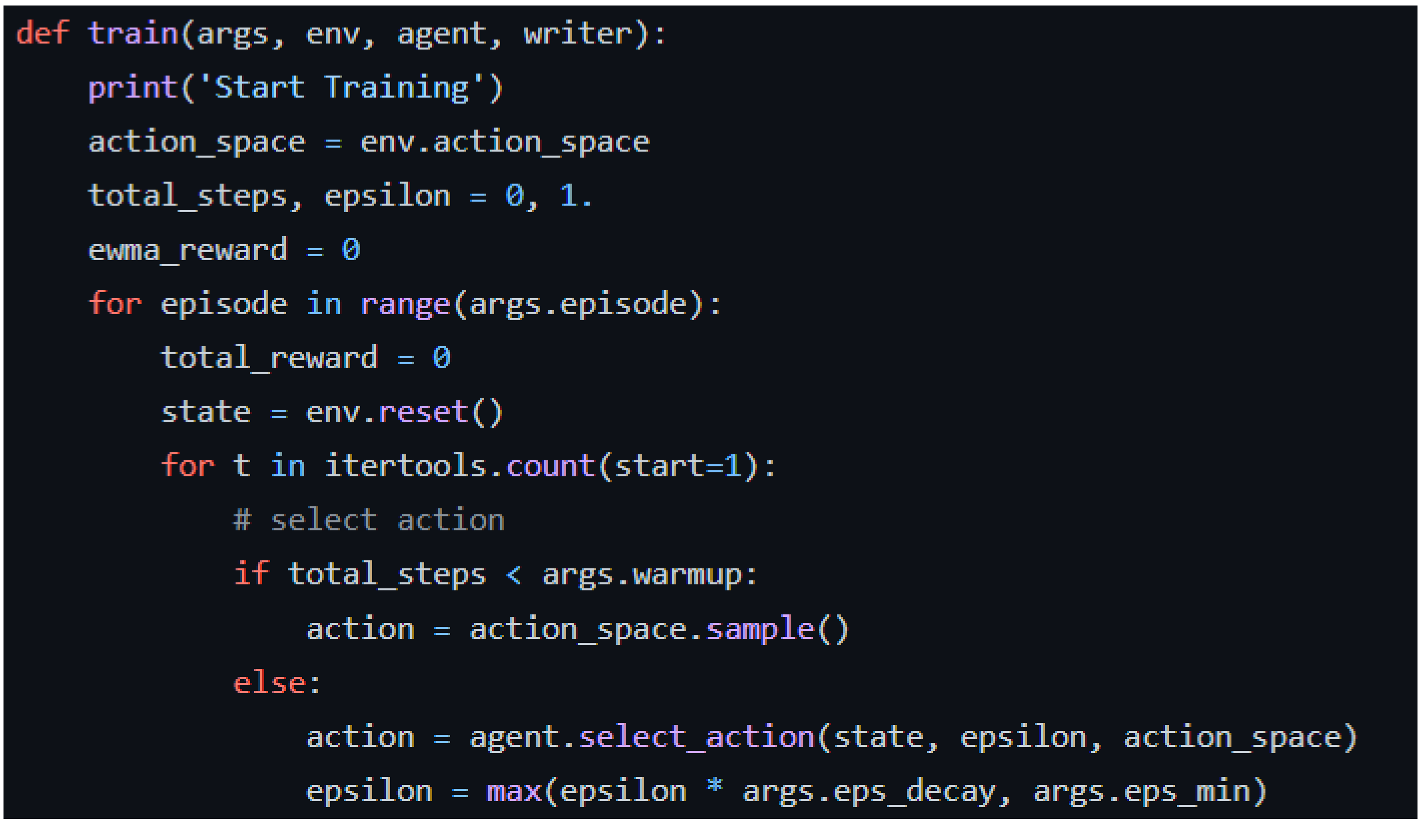
Test

Finally, test trained agent to play the game and record reward, then calculate its mean and output this value.

```
  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()

          for t in itertools.count(start=1):

              env.render()
              # select action
              action = agent.select_action(state, noise=False)
              # execute action
              next_state, reward, done, _ = env.step(action)

              state = next_state
              total_reward += reward

              if(done):
                  writer.add_scalar('Test/Episode Reward', total_reward, n_episode)
                  print("total reward : {0:.2f}".format(total_reward))
                  rewards.append(total_reward)
                  break

      print('Average Reward', np.mean(rewards))
      env.close()
```

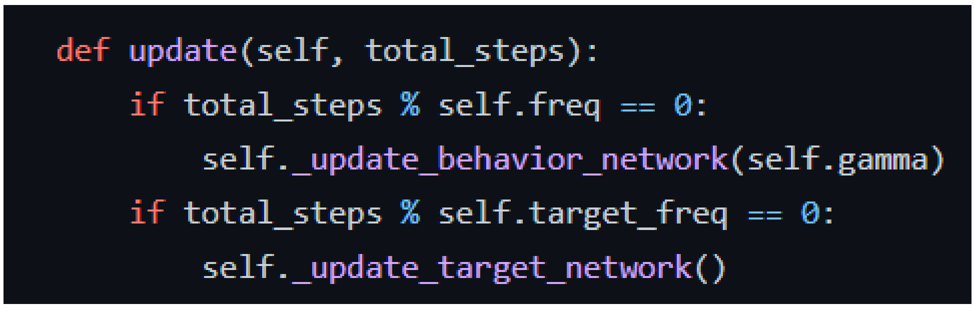
4. Describe differences between your implementation and algorithms

In the begining, the agent will random select action and interaction with game, then save current state, action, next state, reward and done to replay memory. it will repect excute untill the episode grater than warnup(Fig.8), then the network will start update and use epsilon-greedy method to select action, but the bahavior network will update for four iteration in the DQN(Fig.9), instead of update for each iteration.



```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):
            # 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)
```

Fig.8 Warnup



```python
  def update(self, total_steps):
      if total_steps % self.freq == 0:
          self._update_behavior_network(self.gamma)
      if total_steps % self.target_freq == 0:
          self._update_target_network()
```

Fig.9 Update frequency

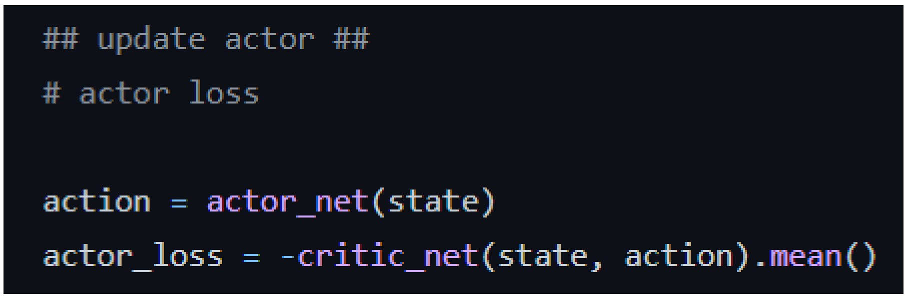
5. Describe your implementation and the gradient of actor updating

In the ActorNet, we hope its Q(s,a) more higher more better, then defined its loss is expectation of Q(s,a).

Update the actor policy using the sampled gradient:

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

Fig.10 Update actor network formula



```
## update actor ##
# actor loss

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

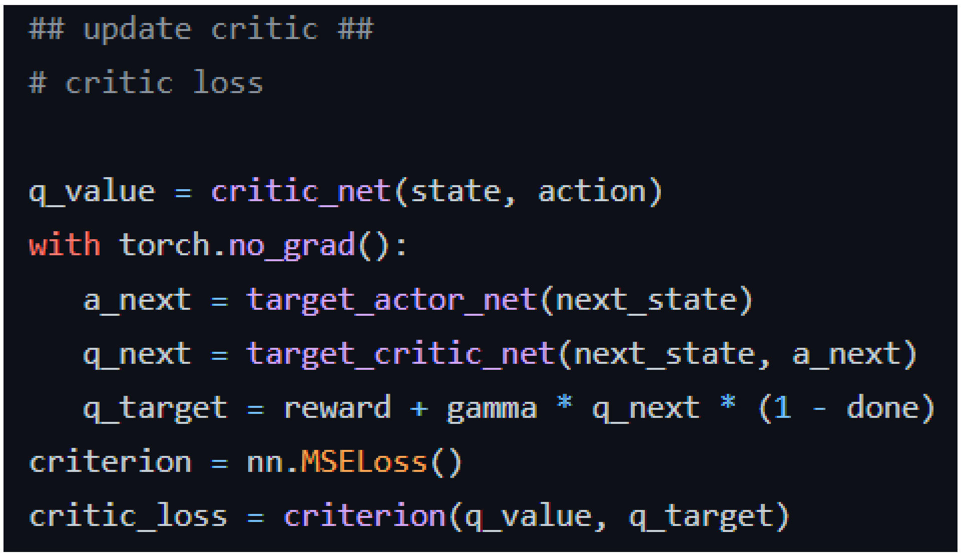
6. Describe your implementation and the gradient of critic updating

In CriticNet, sample minibatch from replay memory, use (s,a,r,s',done) to calculate Q and Q next value, Q value based on state and action, and got next action by used target actor network, then input next state and output action to targer critic network, calculate Q next, finally, use TD learning method to calculate MSE loss and optimize.

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

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

Fig.11 Update critic network formula



```
## update critic ##
# critic loss

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)
```

7. Explain effects of the discount factor

The discount factor essentially determines how much the reinforcement learning agents cares about rewards in the distant future relative to those in the immediate future. If γ=0, the agent will be completely myopic and only learn about actions that produce an immediate reward. If γ=1, the agent will evaluate each of its actions based on the sum total of all of its future rewards.

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

If we use greedy action selection, this method easy cause some actions continuous chose, and other actions no chance to select, maybe other action can get more reward, so epsilon-greedy approache balance between exploration and exploitation, let $\epsilon$ to decide agent action.

9. Explain the necessity of the target network

If without target network, the Q next value and Q value will calculate from the same network, every update will change Q value and Q next value, this situation easy cause the network training unstable, so use a fixed target network, and regular update the target network from behavior network.

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

Use replay buffer can let algorithm more convergence and improvement generalization, but if the replay buffet size is too large, the training will more stable, increase training time. On the contrary, the replay buffer size is too small, the network easy foucus on recently experience cause overfitting, so the appropriate size is important for training a good agent.

11. Implement and experiment on Double-DQN

The Double-DQN improve DQN easy is overestimating problem, the problem is DQN directly select the max Q next from target network input with next state, the Q next is different with behavior network. The Double-DQN use behavior network and next state to decide the action, then based on the selected action to choose target network Q next, although this Q next maybe not maximum, this improvement can increase stability and training time.

```
    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)

        q_value = self._behavior_net(state).gather(dim=1,index=action.long())
        with torch.no_grad():
            action_index = torch.argmax(self._behavior_net(next_state), dim=1).view(-1
            q_next = self._target_net(next_state).gather(dim=1,index=action_index.long
            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()
```

12. Extra hyperparameter tuning, e.g., Population Based Training

Adjust hyperparameter maybe can increase performance, but this hyperparameter tune is difficult, and always finished training, then adjust hyperparameter to train next model, this method namely sequential optimisation, it will waste a plenty of time, but the parallel search can effectively improve this problem, ths disadvantage is can not use mutual parameter optimization information.
The Population Based Training(Fig.12) combined sequential and mutual optimization, not only can explore in parallel, but also use other better hyperparameter to eliminate bad it.

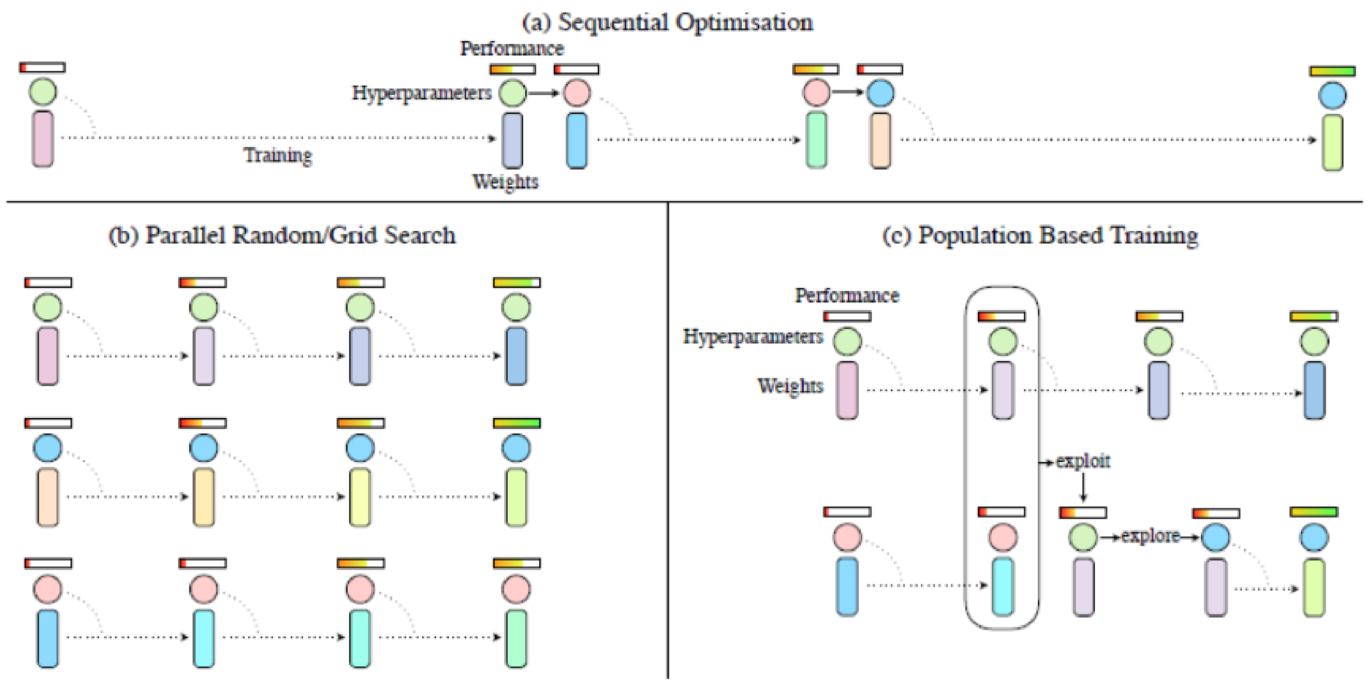
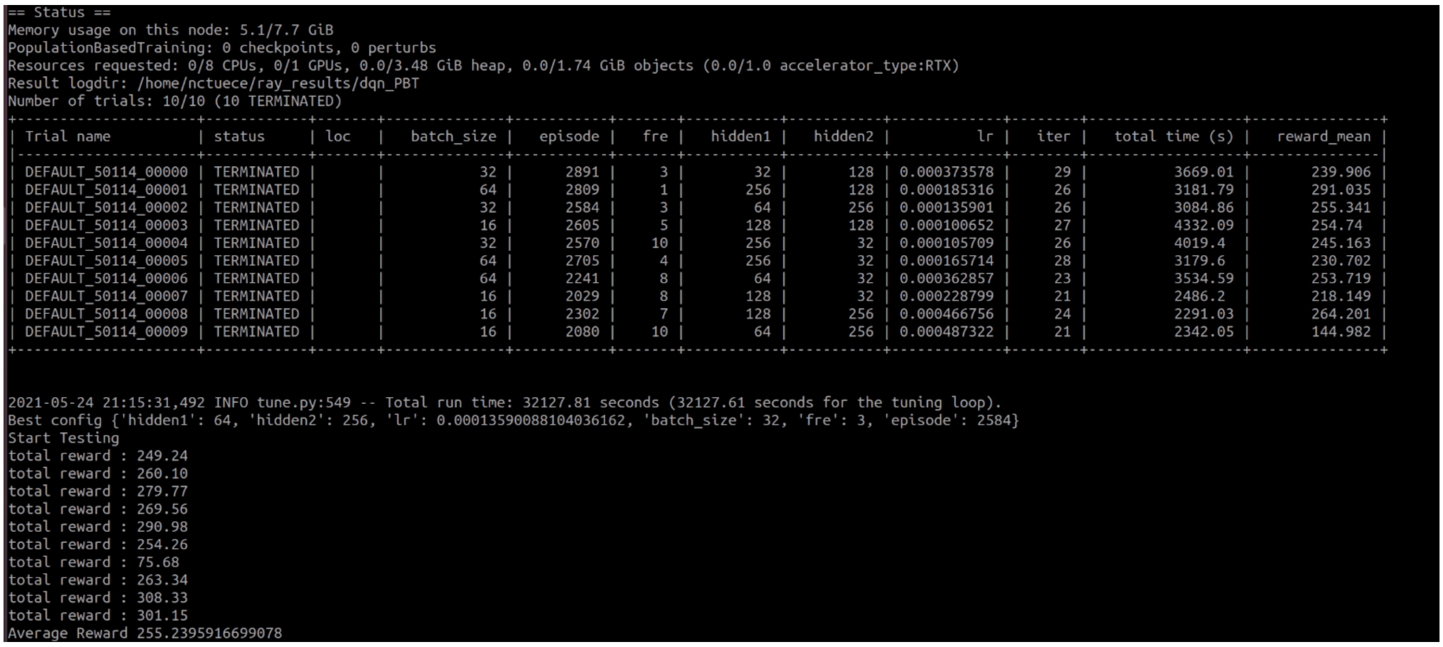


Fig.12 Optimization approaches
In this Lab, I use PBT and ray api to improve DQN, DDQN and DDPG, the result can obvious point out PBT indeed let agent can learning well, the PBT result can see Fig.13~18.



```
== Status ==
Memory usage on this node: 5.1/7.7 GiB
PopulationBasedTraining: 0 checkpoints, 0 perturbs
Resources requested: 0/8 CPUs, 0/1 GPUs, 0.0/3.48 GiB heap, 0.0/1.74 GiB objects (0.0/1.0 accelerator_type:RTX)
Result logdir: /home/nctuece/ray_results/dqn_PBT
Number of trials: 10/10 (10 TERMINATED)
+---------------------+------------+-------+--------------+-----------+-------+-----------+-----------+-------------+--------+------------------+---------------+
| Trial name          | status     | loc   |   batch_size |   episode |   fre |   hidden1 |   hidden2 |          lr |   iter |   total time (s) |   reward_mean |
|---------------------+------------+-------+--------------+-----------+-------+-----------+-----------+-------------+--------+------------------+---------------|
| DEFAULT_50114_00000 | TERMINATED |       |           32 |      2891 |     3 |        32 |       128 | 0.000373578 |     29 |          3669.01 |       239.906 |
| DEFAULT_50114_00001 | TERMINATED |       |           64 |      2809 |     1 |       256 |       128 | 0.000185316 |     26 |          3181.79 |       291.035 |
| DEFAULT_50114_00002 | TERMINATED |       |           32 |      2584 |     3 |        64 |       256 | 0.000135901 |     26 |          3084.86 |       255.341 |
| DEFAULT_50114_00003 | TERMINATED |       |           16 |      2605 |     5 |       128 |       128 | 0.000100652 |     27 |          4332.09 |       254.74  |
| DEFAULT_50114_00004 | TERMINATED |       |           32 |      2570 |    10 |       256 |        32 | 0.000105709 |     26 |          4019.4  |       245.163 |
| DEFAULT_50114_00005 | TERMINATED |       |           64 |      2705 |     4 |       256 |        32 | 0.000165714 |     28 |          3179.6  |       230.702 |
| DEFAULT_50114_00006 | TERMINATED |       |           64 |      2241 |     8 |        64 |        32 | 0.000362857 |     23 |          3534.59 |       253.719 |
| DEFAULT_50114_00007 | TERMINATED |       |           16 |      2029 |     8 |       128 |        32 | 0.000228799 |     21 |          2486.2  |       218.149 |
| DEFAULT_50114_00008 | TERMINATED |       |           16 |      2302 |     7 |       128 |       256 | 0.000466756 |     24 |          2291.03 |       264.201 |
| DEFAULT_50114_00009 | TERMINATED |       |           16 |      2080 |    10 |        64 |       256 | 0.000487322 |     21 |          2342.05 |       144.982 |
+---------------------+------------+-------+--------------+-----------+-------+-----------+-----------+-------------+--------+------------------+---------------+


2021-05-24 21:15:31,492 INFO tune.py:549 -- Total run time: 32127.81 seconds (32127.61 seconds for the tuning loop).
Best config {'hidden1': 64, 'hidden2': 256, 'lr': 0.00013590088104036162, 'batch_size': 32, 'fre': 3, 'episode': 2584}
Start Testing
total reward : 249.24
total reward : 260.10
total reward : 279.77
total reward : 269.56
total reward : 290.98
total reward : 254.26
total reward : 75.68
total reward : 263.34
total reward : 308.33
total reward : 301.15
Average Reward 255.2395916699078
```

Fig.13 DQN PBT

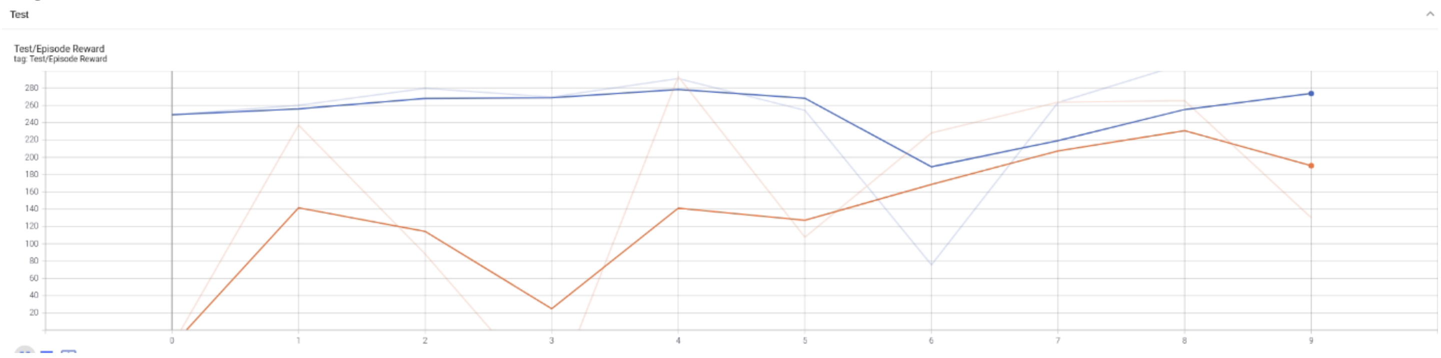
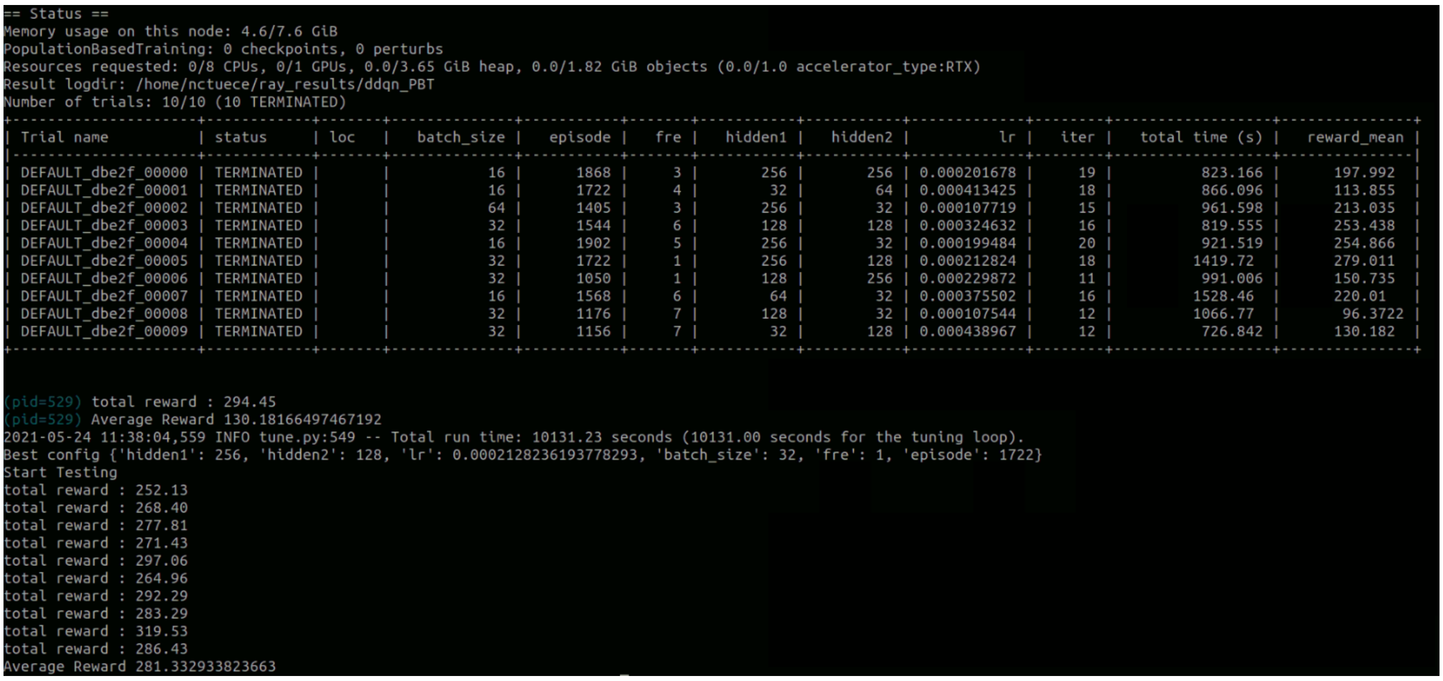


Fig.14 DQN and DQN PBT test result



```
== Status ==
Memory usage on this node: 4.6/7.6 GiB
PopulationBasedTraining: 0 checkpoints, 0 perturbs
Resources requested: 0/8 CPUs, 0/1 GPUs, 0.0/3.65 GiB heap, 0.0/1.82 GiB objects (0.0/1.0 accelerator_type:RTX)
Result logdir: /home/nctuece/ray_results/ddqn_PBT
Number of trials: 10/10 (10 TERMINATED)
+---------------------+------------+-------+--------------+-----------+-------+-----------+-----------+-------------+--------+------------------+---------------+
| Trial name          | status     | loc   |   batch_size |   episode |   fre |   hidden1 |   hidden2 |          lr |   iter |   total time (s) |   reward_mean |
|---------------------+------------+-------+--------------+-----------+-------+-----------+-----------+-------------+--------+------------------+---------------|
| DEFAULT_dbe2f_00000 | TERMINATED |       |           16 |      1868 |     3 |       256 |       256 | 0.000201678 |     19 |          823.166 |      197.992  |
| DEFAULT_dbe2f_00001 | TERMINATED |       |           16 |      1722 |     4 |        32 |        64 | 0.000413425 |     18 |          866.096 |      113.855  |
| DEFAULT_dbe2f_00002 | TERMINATED |       |           64 |      1405 |     3 |       256 |        32 | 0.000107719 |     15 |          961.598 |      213.035  |
| DEFAULT_dbe2f_00003 | TERMINATED |       |           32 |      1544 |     6 |       128 |       128 | 0.000324632 |     16 |          819.555 |      253.438  |
| DEFAULT_dbe2f_00004 | TERMINATED |       |           16 |      1902 |     5 |       256 |        32 | 0.000199484 |     20 |          921.519 |      254.866  |
| DEFAULT_dbe2f_00005 | TERMINATED |       |           32 |      1722 |     1 |       256 |       128 | 0.000212824 |     18 |         1419.72  |      279.011  |
| DEFAULT_dbe2f_00006 | TERMINATED |       |           32 |      1050 |     1 |       128 |       256 | 0.000229872 |     11 |          991.006 |      150.735  |
| DEFAULT_dbe2f_00007 | TERMINATED |       |           16 |      1568 |     6 |        64 |        32 | 0.000375502 |     16 |         1528.46  |      220.01   |
| DEFAULT_dbe2f_00008 | TERMINATED |       |           32 |      1176 |     7 |       128 |        32 | 0.000107544 |     12 |         1066.77  |       96.3722 |
| DEFAULT_dbe2f_00009 | TERMINATED |       |           32 |      1156 |     7 |        32 |       128 | 0.000438967 |     12 |          726.842 |      130.182  |
+---------------------+------------+-------+--------------+-----------+-------+-----------+-----------+-------------+--------+------------------+---------------+


(pid=529) total reward : 294.45
(pid=529) Average Reward 130.18166497467192
2021-05-24 11:38:04,559 INFO tune.py:549 -- Total run time: 10131.23 seconds (10131.00 seconds for the tuning loop).
Best config {'hidden1': 256, 'hidden2': 128, 'lr': 0.0002128236193778293, 'batch_size': 32, 'fre': 1, 'episode': 1722}
Start Testing
total reward : 252.13
total reward : 268.40
total reward : 277.81
total reward : 271.43
total reward : 297.06
total reward : 264.96
total reward : 292.29
total reward : 283.29
total reward : 319.53
total reward : 286.43
Average Reward 281.332933823663
```

Fig.15 DDQN PBT result

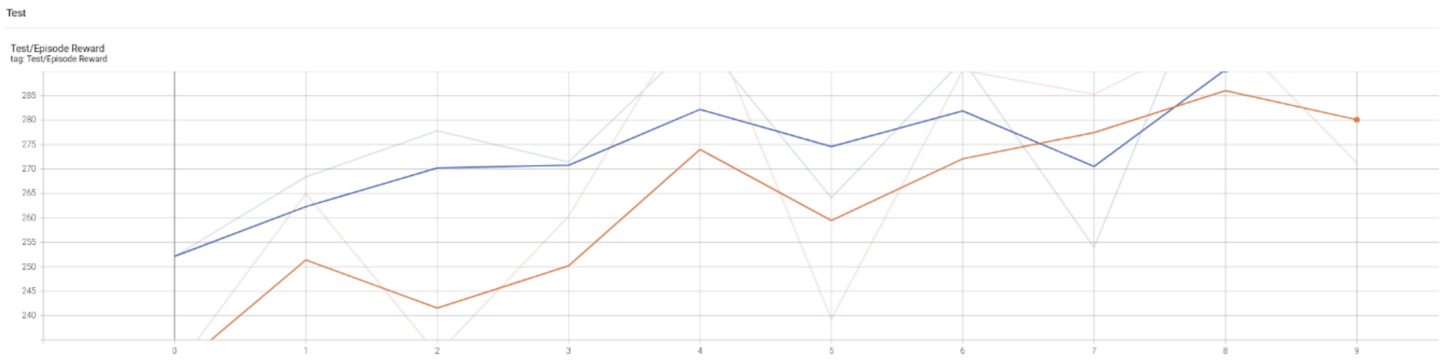
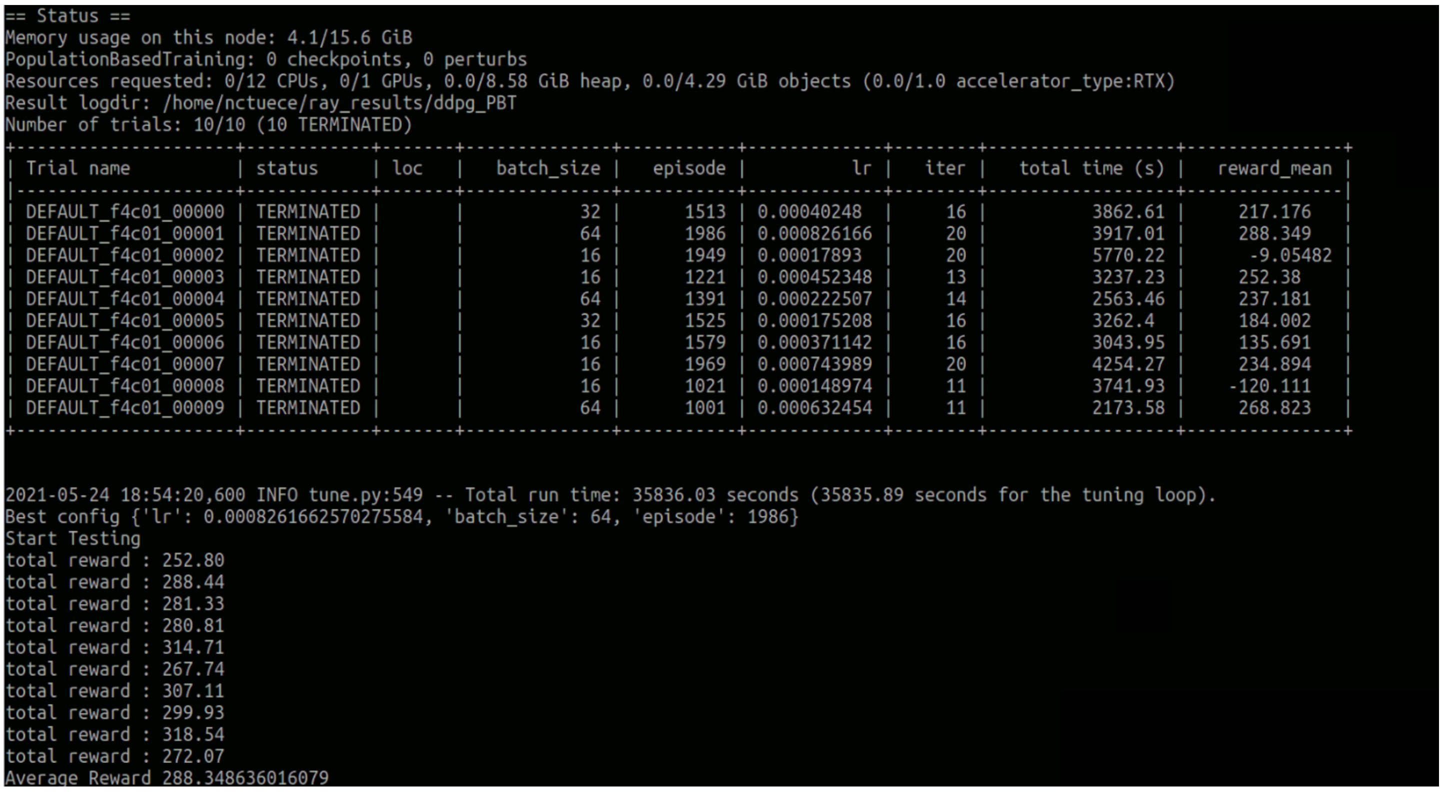


Fig.16 DDQN and DDQN PBT test result



```
== Status ==
Memory usage on this node: 4.1/15.6 GiB
PopulationBasedTraining: 0 checkpoints, 0 perturbs
Resources requested: 0/12 CPUs, 0/1 GPUs, 0.0/8.58 GiB heap, 0.0/4.29 GiB objects (0.0/1.0 accelerator_type:RTX)
Result logdir: /home/nctuece/ray_results/ddpg_PBT
Number of trials: 10/10 (10 TERMINATED)
+-------------------+------------+-------+--------------+-----------+-------------+--------+------------------+--------------+
| Trial name        | status     | loc   |   batch_size |   episode |          lr |   iter |   total time (s) |  reward_mean |
|-------------------+------------+-------+--------------+-----------+-------------+--------+------------------+--------------|
| DEFAULT_f4c01_00000 | TERMINATED |       |           32 |      1513 | 0.00040248  |     16 |          3862.61 |    217.176   |
| DEFAULT_f4c01_00001 | TERMINATED |       |           64 |      1986 | 0.000826166 |     20 |          3917.01 |    288.349   |
| DEFAULT_f4c01_00002 | TERMINATED |       |           16 |      1949 | 0.00017893  |     20 |          5770.22 |     -9.05482 |
| DEFAULT_f4c01_00003 | TERMINATED |       |           16 |      1221 | 0.000452348 |     13 |          3237.23 |    252.38    |
| DEFAULT_f4c01_00004 | TERMINATED |       |           64 |      1391 | 0.000222507 |     14 |          2563.46 |    237.181   |
| DEFAULT_f4c01_00005 | TERMINATED |       |           32 |      1525 | 0.000175208 |     16 |          3262.4  |    184.002   |
| DEFAULT_f4c01_00006 | TERMINATED |       |           16 |      1579 | 0.000371142 |     16 |          3043.95 |    135.691   |
| DEFAULT_f4c01_00007 | TERMINATED |       |           16 |      1969 | 0.000743989 |     20 |          4254.27 |    234.894   |
| DEFAULT_f4c01_00008 | TERMINATED |       |           16 |      1021 | 0.000148974 |     11 |          3741.93 |   -120.111   |
| DEFAULT_f4c01_00009 | TERMINATED |       |           64 |      1001 | 0.000632454 |     11 |          2173.58 |    268.823   |
+-------------------+------------+-------+--------------+-----------+-------------+--------+------------------+--------------+


2021-05-24 18:54:20,600 INFO tune.py:549 -- Total run time: 35836.03 seconds (35835.89 seconds for the tuning loop).
Best config {'lr': 0.0008261662570275584, 'batch_size': 64, 'episode': 1986}
Start Testing
total reward : 252.80
total reward : 288.44
total reward : 281.33
total reward : 280.81
total reward : 314.71
total reward : 267.74
total reward : 307.11
total reward : 299.93
total reward : 318.54
total reward : 272.07
Average Reward 288.348636016079
```

Fig.17 DDPG PBT result

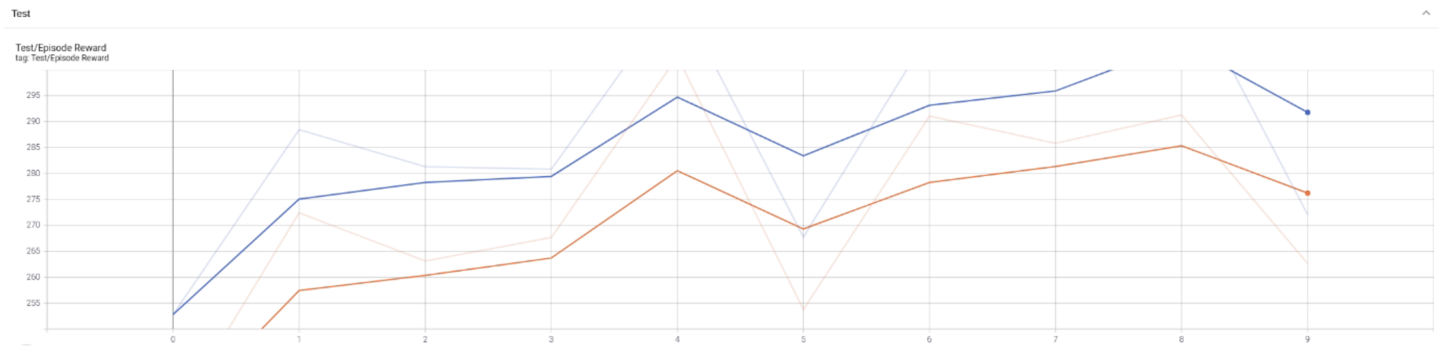


FIg.18 DDPG and DDPG PBT test result

13. Performance

LunarLander-v2

The result can see DDQN PBT result(Fig.15), the Average Reward is **281.33**.

LunarLanderContinuous-v2

The result can see DDQN PBT result(Fig.17), the Average Reward is **288.34**.

tags: `DLP2021`