DLP HW6

310551053 Chieh-Ming Jiang

August 22th 2021

1. LunarLander-v2

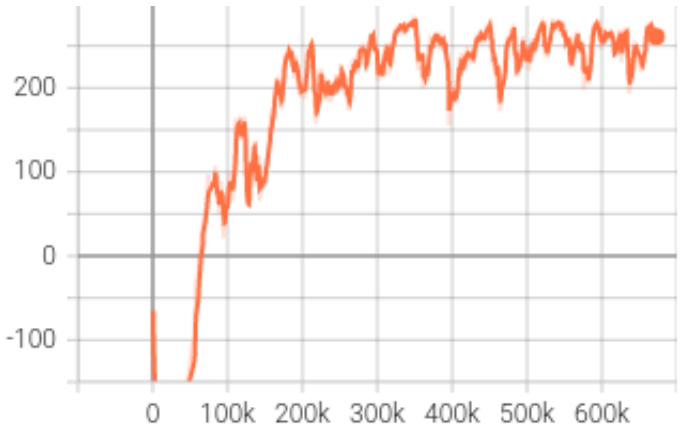

Figure 1: Tensorboard of DDQN

```
(base) ubuntu@ec037-048:~/DLP/HW6$ python3 ddqn.py --test_only --render
Net(
  (relu): ReLU()
  (fc1): Linear(in_features=8, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=512, bias=True)
  (fc4): Linear(in_features=512, out_features=512, bias=True)
  (classify): Linear(in_features=512, out_features=4, bias=True)
)
Start Testing
Average Reward 283.7527255988115
```

Figure 2: LunarLander-v2

2. LunarLanderContinuous-v2

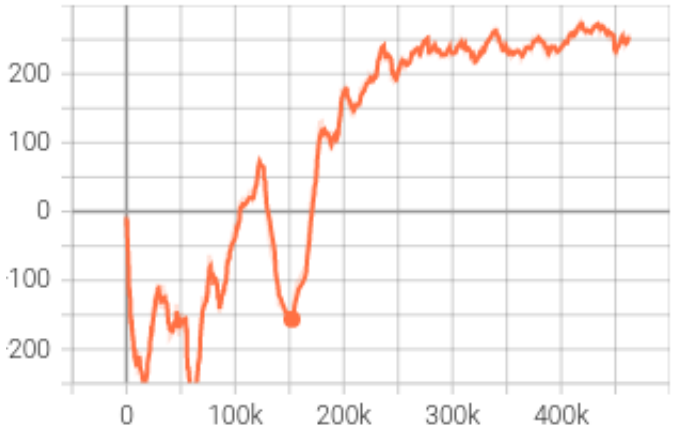

Figure 3: Tensorboard of DDPG

```
(base) ubuntu@ec037-048:~/DLP/HW6$ python3 ddpg.py --test_only --render
Start Testing
Average Reward 263.0410644490999
```

Figure 4: LunarLanderContinuous-v2

3. Implementation of both algorithms

   (a) DQN

   - Network
     The Network is composed of five fully connected layers.

```python
class Net(nn.Module):
    def __init__(self, state_dim=8, action_dim=4, hidden_dim=512):
        super().__init__()
        ## TODO ##
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, hidden_dim)
        self.classify = nn.Linear(hidden_dim, 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.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        x = self.classify(x)
        return x
```

Figure 5: DQN network

- Action
  Take action with ϵ greedy algorithm. The agent takes the move with the highest score with probability of ϵ, and takes the random action with probability of 1 - ϵ. The value of ϵ decreases when the episodes getting larger.

```python
def select_action(self, state, epsilon, action_space):
    '''epsilon-greedy based on behavior network'''
    ## TODO ##
    if np.random.random() > epsilon:
        state = torch.Tensor(state).to(self.device).view(1, -1)
        with torch.no_grad():
            self._behavior_net.eval()
            action = self._behavior_net.forward(state).argmax().item()
    else:
        action = action_space.sample()
    return action
```

Figure 6: ϵ greedy

- Behavior Network
  Update the behavior network via target network by the following formula. Use MSE as loss function.

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

Figure 7: behavior algorithm

```
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 ##
    self._behavior_net.train()
    q_value = self._behavior_net.forward(state).gather(1, action.long())

    with torch.no_grad():
        self._target_net.eval()
        q_target = reward + (1 - done) * gamma * \
            self._target_net.forward(next_state).max(dim=1)[0].view(-1, 1)

    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(), 1)
    self._optimizer.step()
```

Figure 8: Behavior Network

- Target Network
  Copy the value of behavior network to behavior network with certain frequency.

```
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())
```

Figure 9: Target Network

(b) DDPG

- Actor Network
  The network is similar to the network of DQN. The difference is

that the actor network output with hyper tangent function.

```python
class ActorNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=2, hidden_dim=(400, 300)):
        super().__init__()
        ## TODO ##
        self.tanh = nn.Tanh()
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(state_dim, hidden_dim[0])
        self.fc2 = nn.Linear(hidden_dim[0], hidden_dim[0])
        self.fc3 = nn.Linear(hidden_dim[0], hidden_dim[1])
        self.fc4 = nn.Linear(hidden_dim[1], hidden_dim[1])
        self.classify = nn.Linear(hidden_dim[1], 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.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        x = self.classify(x)
        x = self.tanh(x)
        return x
```

Figure 10: Actor Network

- Critic Network
  Output a scalar of predicted Q(s,a).

```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)
```

Figure 11: Critic Network

- Action
  Add the gaussian noise to the predicted action.

```python
def select_action(self, state, noise=True):
    '''based on the behavior (actor) network and exploration noise'''
    ## TODO ##
    state = torch.from_numpy(state).float().to(self.device)

    with torch.no_grad():
        action = self._actor_net(state).cpu().numpy()

    if noise:
        action += self._action_noise.sample()

    return action
```

Figure 12: Action

- Behavior Network of critic
  Get the target Q value with the target network of actor and critic.
  Then use MSE to calculate the loss and update the parameters.

```
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
    ## TODO ##
    q_value = self._critic_net(state,action)
    with torch.no_grad():
        action_next = target_actor_net(next_state)
        q_target = reward + (1 - done) * gamma * target_critic_net(next_state, action_next)
    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()
```

Figure 13: Critic

- Behavior Network of actor
  Measure the loss of actor with -1 * Q(s,a).

```
action = self._actor_net(state)
actor_loss = -self._critic_net(state,action).mean()
# action = ?
# actor_loss = ?
# raise NotImplementedError
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

Figure 14: Actor

- Target Network
  Use soft target update. Update the target network with $1-\tau$ of target network and τ of behavior network.

```python
def _update_target_network(target_net, net, tau):
    '''update target network by _soft_ copying from behavior network'''
    for target, behavior in zip(target_net.parameters(), net.parameters()):
        ## TODO ##
        target.data.copy_((1-tau)*target.data + tau*behavior.data)
```

Figure 15: Target network

4. Differences between your implementation and algorithms
   Basically, the implementation is similar to the algorithms. What's difference is that the gradient is clipped when updating the network parameters.

5. Implementation and the gradient of actor updating
   Because the actor hope to maximize the output value of critic, so calculating the gradient of actor is equal to minimizing the negative loss of critic.

```python
action = self._actor_net(state)
actor_loss = -self._critic_net(state,action).mean()
# action = ?
# actor_loss = ?
# raise NotImplementedError
# optimize actor
actor_net.zero_grad()
critic_net.zero_grad()
actor_loss.backward()
actor_opt.step()
```

Figure 16: Actor

6. Implementation and the gradient of critic updating
   Get the Q value with the target network of actor and critic. Then use MSE to calculate the loss and update the parameters.

$$y_i = r_i + \gamma Q^{'}(S_{t+1}, u^{'}(S_{t+1}|\theta^{u^{'}}|\theta^{Q^{'}})$$

$$L = \frac{1}{M}\sum_{i}(y_i - Q(s_i, a_i|\theta^{Q}))^2$$

```python
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
    ## TODO ##
    q_value = self._critic_net(state,action)
    with torch.no_grad():
        action_next = target_actor_net(next_state)
        q_target = reward + (1 - done) * gamma * target_critic_net(next_state, action_next)
    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()
```

Figure 17: Critic

7. Discount factor
   The discount factor is used to describe how important the state is. Usually, the factor is smaller than 1. Hence the state that is far behind the current state may influence less than the one that is near.

8. Benefits of epsilon-greedy
   It's the matter of exploration. There may be some good choices that we haven't taken under the our policy, so sometimes we need to randomly choose some action.

9. Target network
   Because the behavior network is always changing during the training process. So we fix a target network ,which not changes so often. As a consequence, the training process is much more stable.

10. Replay buffer size
    If the replay buffer is too small, the model is always training with recent steps, which may overfits the model and forget the good steps learned before. However, if the replay buffer is too large, there is less likely to sample new steps, which may slows down the training process.

11. Double-DQN
    The main difference between DQN and DDQN is the way they get the target Q value. Because DQN may encounter the problem of overestimating q-value, DDQN selects the action from the behavior network instead of target network.

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

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

Figure 18: DDQN

```
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 ##
    self._behavior_net.train()
    q_value = self._behavior_net.forward(state).gather(1, action.long())
    with torch.no_grad():
        next_action =  self._behavior_net.forward(next_state).max(dim=1)[1].view(-1,1)
        self._target_net.eval()
        q_target = reward + (1 - done) * gamma * \
                self._target_net.forward(next_state).gather(1, next_action.long())
```

Figure 19: DDQN

12. Extra hyperparameter tuning
    The population based learning method initializes many agents with random hyperparameters. Then these agent would learn from the parameters from those perform well and discard the bad ones.