Lab8

鄭余玄

1 Introduction

In this lab, we are going to perform experiments on CartPole-v0 and Pendulum-v0 using DQN and DDPG implemented in the *DeepRL* framework. The framework and algorithms are introduced in Section 2 and the setup of those two games are described in Section 3 and 4. Finally, the episodic reward during training and testing is shown in Section 5.

2 DeepRL framework

DeepRL framework provides several implementation of deep reinforcement learning algorithms. In this lab, we are going to use the `DQNAgent` and `DDPGAgent`, described in Subsection 2.1 and 2.2 respectively. The neural network wrapper is described in Section 2.3.

The training starts from `run_step` defined in `deep_rl.misc`. Then, the agent iterates over training and testing by invoking `agent.step()` until the maximum timestep is reached.

2.1 DQNAgent

When `step` method is invoked in `DQNAgent`, the behaviour network in `DQNActor` first steps. The action selection of `DQNActor` is using ε-greedy:

```
if np.random.rand() < config.random_action_prob():
  # epsilon
  action = np.random.randint(0, len(q_values))
else:
  # greedy
  action = np.argmax(q_values)
```

The transition is stored and a random sampled minibatch of transitions are returned in `DQNAgent.step`. Then, the behavior network is updated in `DQNAgent`, where the loss is

$$L_Q(s,a;\theta) = (y_t^Q - Q(s,a;\theta))^2$$

, and its gradient is

$$\nabla_\theta L_Q(s,a;\theta) = 2(y_t^Q - Q(s,a;\theta))\nabla_\theta Q(s,a;\theta)$$

.

```
# target Q: y_t^Q = r + γ max_a Q̂(s', a; θ^-)
q_next = self.target_network(next_states).detach()
q_next = rewards + self.config.discount * q_next.max()
# loss = (y_t^Q - Q(s, a; θ))^2
loss = (q_next - q).pow(2).mean()
```

For every update frequency (`self.config.target_network_update_freq`), the parameters of the target network is updated by the parameters of the behaviour network:

```
self.target_network.load_state_dict(self.network.state_dict())
```

2.2 DDPGAgent

When `step` method is invoked in `DDPGAgent`, the action selects according to the behaviour network:

```
# a_t = μ(s_t, θ^μ) + N_t where N_t is sampled from some random process
# (we use OrnsteinUhlenbeckProcess here)
action = self.network(self.state) + self.random_process.sample()
```

The transition is stored and a minibatch of transitions are sampled. Then, the critic updates the behavior network, where the loss is

$$L_Q(s,a;\theta^Q) = (y_t^Q - Q(s,a;\theta^Q))^2$$

, and its gradient is

$$\nabla_{\theta^Q} L_Q(s,a;\theta^Q) = 2(y_t^Q - Q(s,a;\theta^Q))\nabla_{\theta^Q} Q(s,a;\theta^Q)$$

.

```python
# $a_{t+1} = \mu'(s_{t+1}|\theta^{\mu'})$
a_next = self.target_network.actor(state_next)
# target Q: $y_t^Q = r + \gamma Q'(s_{t+1}, a_{t+1}|\theta^{Q'})$
q_next = self.target_network.critic(state_next, a_next).detach()
q_next = reward + config.discount * q_next
# loss = $(y_t^Q - Q(s_t, a_t|\theta^Q))^2$
critic_loss = (q_next - q).pow(2).mean()
```

On the otherhand, the loss of actor policy is

$$J(\theta^\mu) \approx \mathbb{E}_{(s_t,a_t)\sim\mu}[Q(s_t, a_t; \theta^Q)]$$

, and its gradient is:

$$\boldsymbol{\nabla}_{\theta^\mu} J(\theta^\mu) \approx \mathbb{E}_{(s_t,a_t)\sim\mu}\left[\boldsymbol{\nabla}_a Q(s_t, a; \theta^Q)\Big|_{a=\mu(s_t;\theta^\mu)} \right] \boldsymbol{\nabla}_{\theta^\mu} \mu(s_t; \theta^\mu)$$

.

```python
policy_loss = -self.network.critic(state.detach(), action).mean()
```

Notice that the `state` is detached but the `action` is not, which enables the gradient to flow through the actor policy $\boldsymbol{\nabla}_{\theta^\mu} \mu(s_t; \theta^\mu)$.

Finally, performs a soft update toward the target network:

```python
# soft update
tau = self.config.target_network_mix
# $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$, $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
for target_param, param in zip(self.target_network.parameters(),
                                self.network.parameters()):
  target_param.copy_(tau * param + (1. - tau) * target_param)
```

2.3 Network

The submodule `deep_rl.network` provides several useful predefined network architectures. A network is decoupled into the head part and the body part. The head part network serves as the entire neural network, which forwards through the body part and finally the *real* head part. The body part consists of basic building blocks of the neural network, such as fully-connected layers and convolutional layers.

For DQN, we use `VanillaNet` for the head part, and the `FCBody` for the body part; for DDPG, we use `DeterministicActorCriticNet` for the head part, and the `FCBody` for the body part

3 CartPole-v0

The network input is a 4-dimension (`config.state_dim`) vector (Cart Position, Cart Velocity, Pole Angle, Pole Velocity at Tip) rather than an image. The first layer whose input dimension is 4 and output dimension is 32 is a fully-connected layer with the ReLU activation function, which is implemented by `FCBody` and plays the role `body` in `VanillaNet`. The last layer whose input dimension 32 and output dimension is 2 (`config.action_dim`) is also a fully-connected layer, which is implemented in the `fc_head` of `VanillaNet`.

```
config.network_fn = lambda: VanillaNet(
  config.action_dim, FCBody(config.state_dim, hidden_units=(32, )))
```

The hyperparameters are defined as the following:

```
# optimizer RMSprop with lr 0.0005
config.optimizer_fn = lambda params: torch.optim.RMSprop(params, 0.0005)
# experience buffer size: 5000 (with batch size 128)
config.replay_fn = lambda: Replay(memory_size=int(5e4), batch_size=128)
# epsilon decays from 1 with rate 0.995
config.random_action_prob = LinearSchedule(1.0, 0.0066, 1e4)
# discount factor (gamma) = 0.95
config.discount = 0.95
# target network update frequency: every 50 steps
config.target_network_update_freq = 50
```

4 Pendulum-v0

The actor network input is a 3-dimension (`config.state_dim`) vector. The first two layers are fully-connected layers with the ReLU activation function, which is implemented by `FCBody`. The last layer whose output dimension is 1 (`config.action_dim`)) is also a fully-connected layer but with `tanh` activation function, which is implemented in `fc_action` of `DeterministicActorCriticNet`.

The critic network input is also the state vector of dimension `config.state_dim`. The first two layers are fully-connected layers with the ReLU activation function; however, the input of the second layer is not only the output of the first layer but also with the concatenation with an action vector, which is implemented in `TwoLayerFCBodyWithAction`). The last layer whose output dimension is 1 (`config.action_dim`) is a fully-connected layer, which is implemented in `fc_critic` of `DeterministicActorCriticNet`.

Both actor and critic network utilizes the Adam optimizer. The learning rate for the actor network is 0.0001, while the critic network is 0.001.

```
config.network_fn = lambda: DeterministicActorCriticNet(
  config.state_dim,
  config.action_dim,
  actor_body=FCBody(config.state_dim, (400, 300), gate=F.relu),
  critic_body=TwoLayerFCBodyWithAction(
    config.state_dim, config.action_dim, (400, 300), gate=F.relu),
  actor_opt_fn=lambda params: torch.optim.Adam(params, lr=1e-4),
  critic_opt_fn=lambda params: torch.optim.Adam(params, lr=1e-3))
```

The hyperparameters are defined as the following:

```
# experience buffer size: 10,000 (with batch size 64)
config.replay_fn = lambda: Replay(memory_size=int(1e4), batch_size=64)
# discount factor (gamma) = 0.95
config.discount = 0.99
# tau = 0.001
config.target_network_mix = 1e-3
```

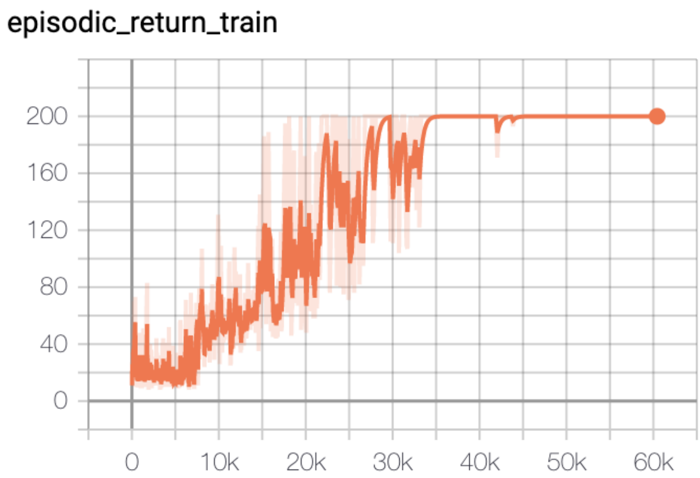
5 Result

Figure 1 shows the episodic return of both training and testing over 60K steps. The average rewards is 200 during 100 testing episodes.

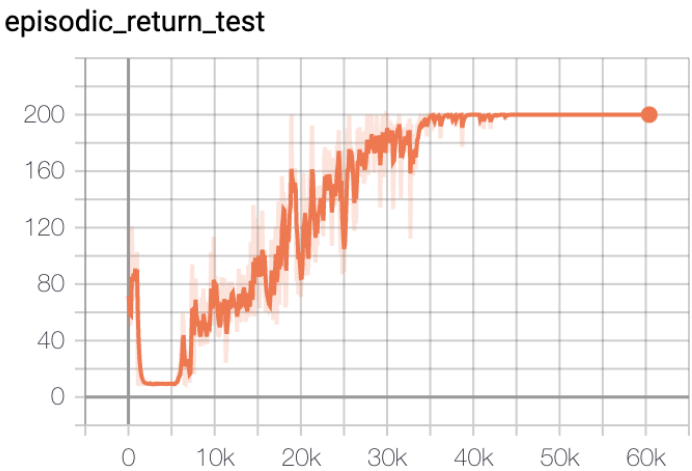
Figure 2 shows the episodic return of training over 1M steps. The highest episodic reward is -0.23 during training.

6 Discussion

CartPole-v0 plays very well after 35K steps, while Pendulum-v0 does not improve much even after 100K steps. Also, at the very beginning of the Pendulum-v0, the score could accidentally reach very high score (about -2).



(a) train



(b) test

Figure 1: CartPole-v0 Episodic Return

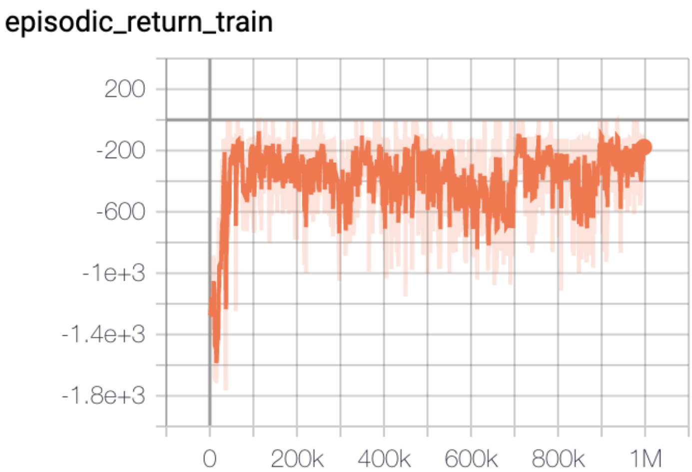


Figure 2: Pendulum-v0 Episodic Return during training