Day 4. Deep Deterministic Policy Gradient

NPEX Reinforcement Learning

2020.08.31

Jaeuk Shin

CORE
Control + Optimization Research Lab

DDPG - Review

Recap : DQN aims to learn Q, and choose action greedily as follows:

$$a_t = \arg\max_a Q(s_t, a; \theta),$$

Now, instead of choosing action a_t directly by solving

$$\max_a Q(s_t, a; \theta),$$

we employ a seperate **actor network** π_ϕ and just select a_t by

$$a_t = \pi_\phi(s_t).$$

$\longrightarrow$ **Actor-Critic methods**

DDPG - Review

Updating θ?

Training $Q(s, a; \theta)$ in DDPG is simple as same as training it in DQN:

$$\textbf{DQN}: \quad y_j = r_j + \gamma \max_a Q(s'_j, a; \theta) \quad \textbf{(TD target)}$$

$$\textbf{DDPG}: \quad y_j = r_j + \gamma Q(s'_j, \pi_\phi(s_j); \theta)$$

We solve (by performing gradient descent)

$$\min_\theta \frac{1}{|B|} \sum_j |Q(s_j, \boxed{a_j}; \theta) - y_j|^2.$$

not $\pi_\phi(s_j)$!

CORE

Control + Optimization Research Lab

DDPG - Review

How to tune params ϕ of the actor π_ϕ?

Our original goal was to solve

$$\max_a Q(s_t, a; \theta),$$

so we expect π_ϕ to satisfy

$$Q(s_t, \pi_\phi(s_t); \theta) \approx \max_a Q(s_t, a; \theta)$$

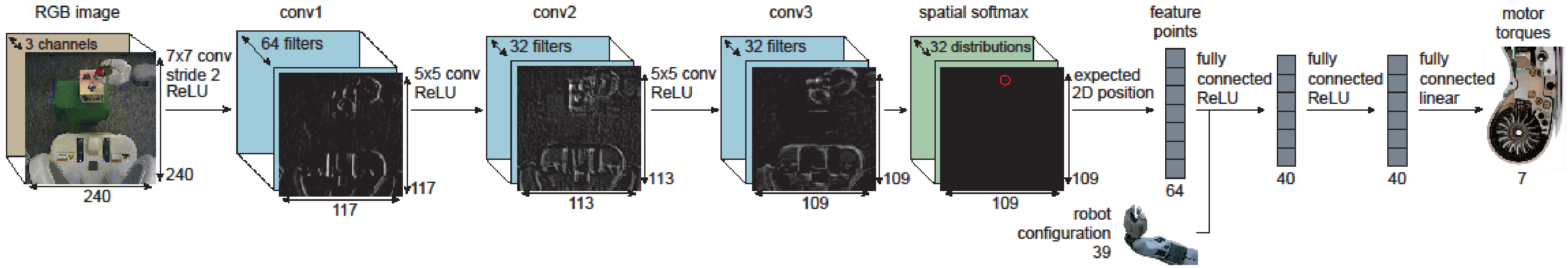
This naturally leads to the following optimization problem!

$$\max_\phi Q(s_t, \pi_\phi(s_t); \theta)$$

CORE
Control + Optimization Research Lab

DDPG - Implementation

new component : actor network

$s_t \rightarrow$  $\rightarrow a_t$

π_ϕ

This is just another **torch.nn.Module!**

DDPG - Implementation

```python
class Actor(nn.Module):
    """
    implementation of actor network mu(s)
    """

    def __init__(self, state_dim, action_dim, hidden_size1, hidden_size2):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, action_dim)

    def forward(self, state):

        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = torch.tanh(self.fc3(x))    # each entry of the action lies in (-1, 1)

        return x
```

DDPG - Implementation

```
def get_action(self, state, eval=False):

    state = torch.tensor(state, dtype=torch.float)

    with torch.no_grad():
        action = self.pi(state)
        action = action.numpy()
    if not eval:
        # for exploration, we use a behavioral policy of the form
        # \beta(s) = \pi(s) + N(0, \sigma^2)
        noise = self.sigma * np.random.randn(self.dimA)
        return action + noise
    else:
        return action
```

DDPG - Implementation

```
target = rewards + self.gamma * mask * self.targ_Q(next_observations, self.targ_pi(next_observations))

out = self.Q(observations, actions)
loss_ftn = MSELoss()
loss = loss_ftn(out, target)
self.Q_optimizer.zero_grad()
loss.backward()
self.Q_optimizer.step()

pi_loss = - torch.mean(self.Q(observations, self.pi(observations)))
self.pi_optimizer.zero_grad()
pi_loss.backward()
self.pi_optimizer.step()
```

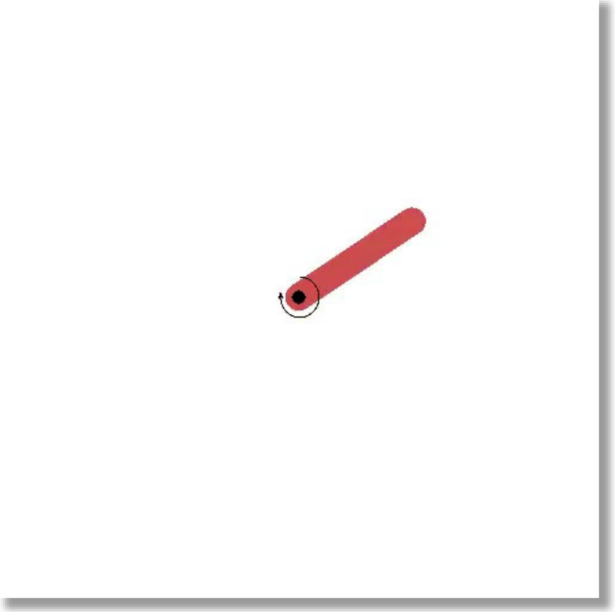
This is how we train DDPG!

Tip. freeze networks properly

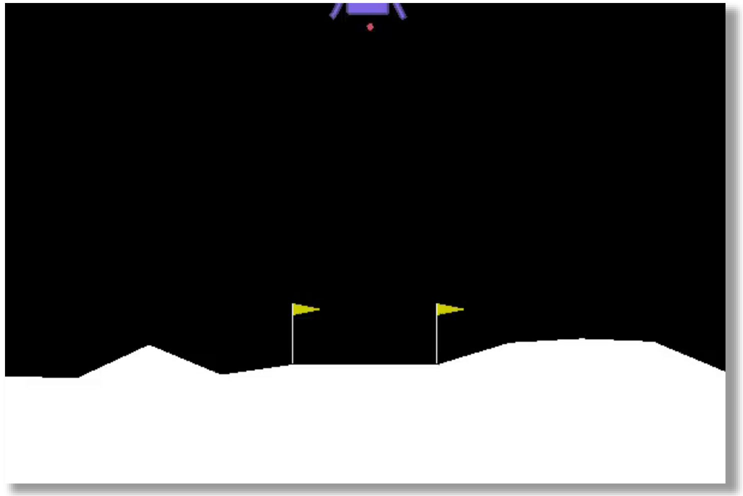
DDPG - Experiments

Task 1. Pendulum-v0 (see plot.py & test.py)

toy problem $\longrightarrow$



Task 2. LunarLanderContinuous-v2



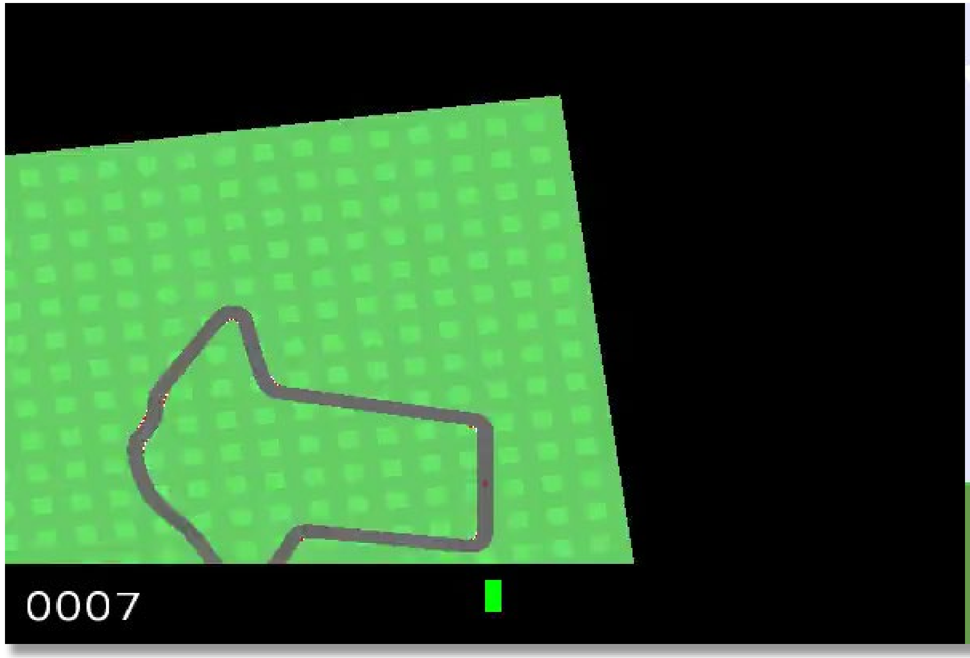
$\longleftarrow$ bit challenging!

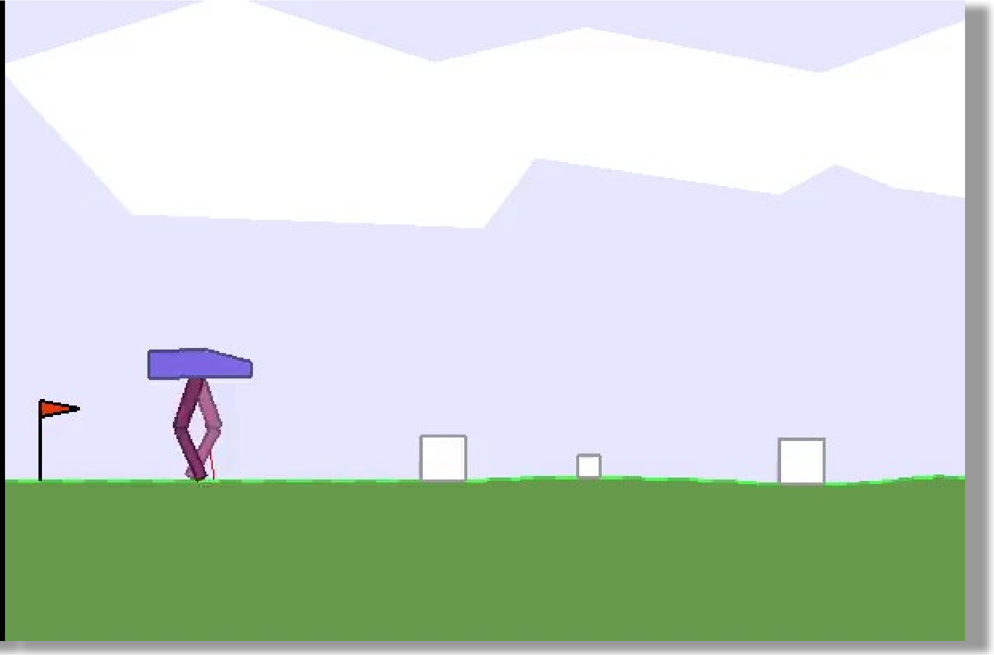
DDPG - Experiments

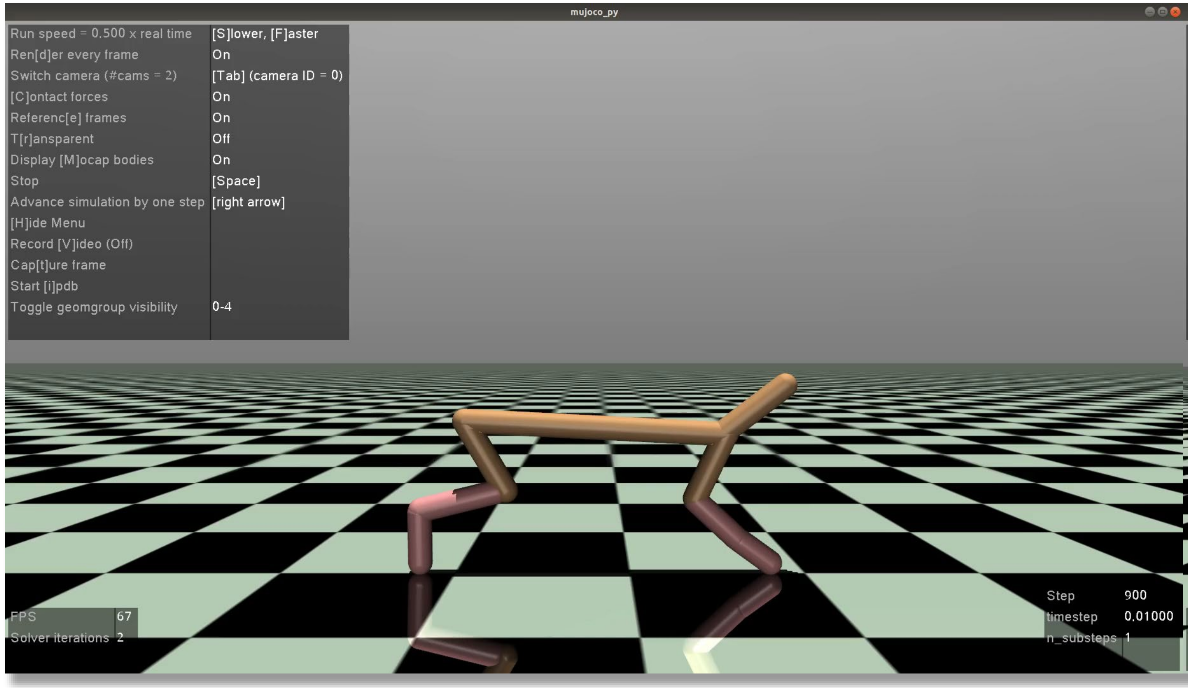
Install **Box2D** environments(including LunarLander) by

pip install Box2D

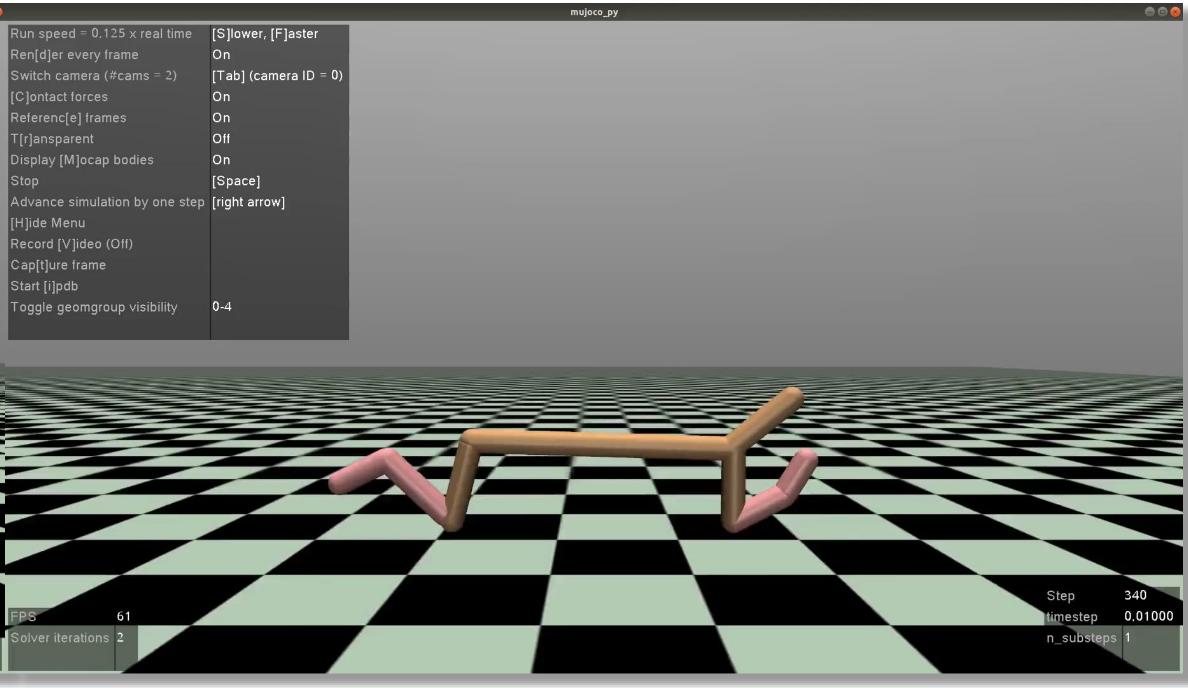
includes some challenging problems(try these with DDPG!)







untrained



trained

CORE
Control + Optimization Research Lab

Thank you!