Policy Gradient Methods in Deep Reinforcement Learning

Introduction and Implementation

Yinghan Sun

Control & Learning for Robotics and Autonomy (CLEAR) Lab

05/30/2024

Contact: yinghansun2@gmail.com

CLEAR
Control & Learning for Robotics and Autonomy Lab

Agenda

- ✓ **Training deep neural networks**
- ✓ **Overview of basic RL concepts**
- ✓ **Building your environments for RL training**
- ✓ **Policy gradient methods for deep RL**

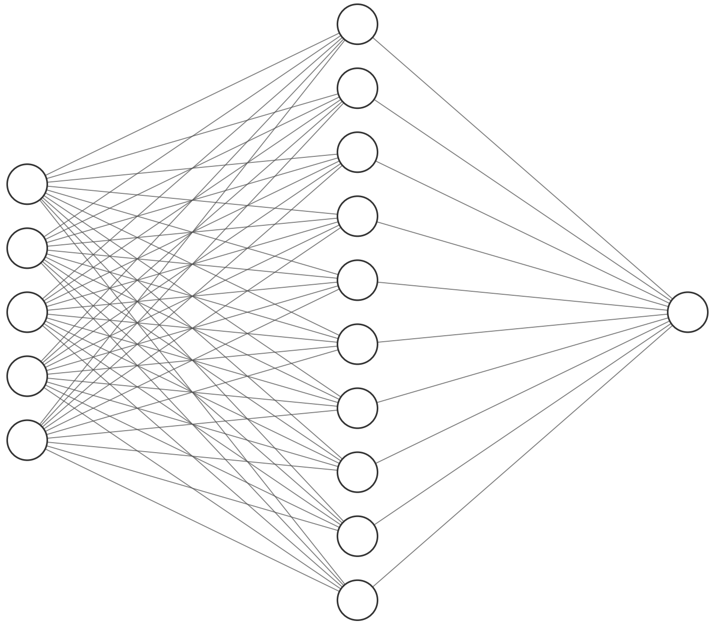
How to define a simple network?

could have multiple hidden layers

fully-connected layer

activation function

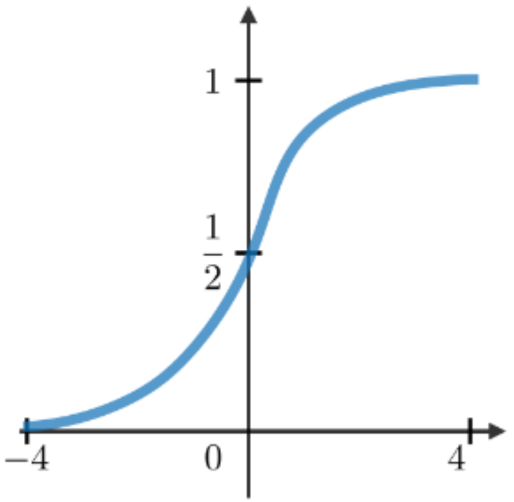
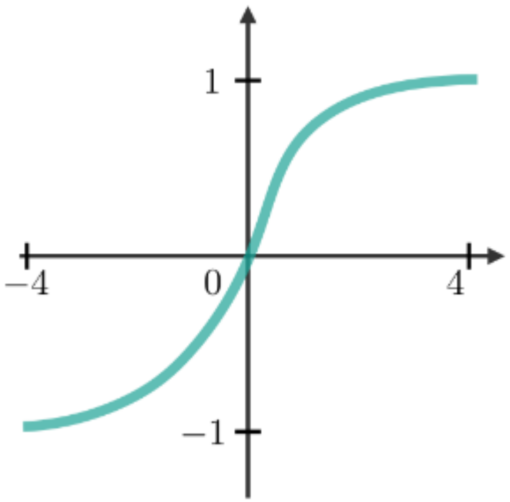
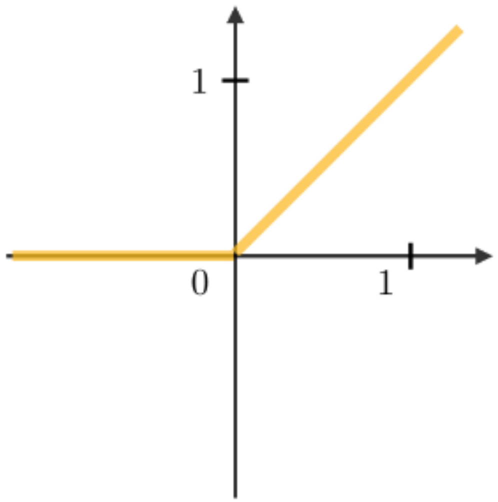
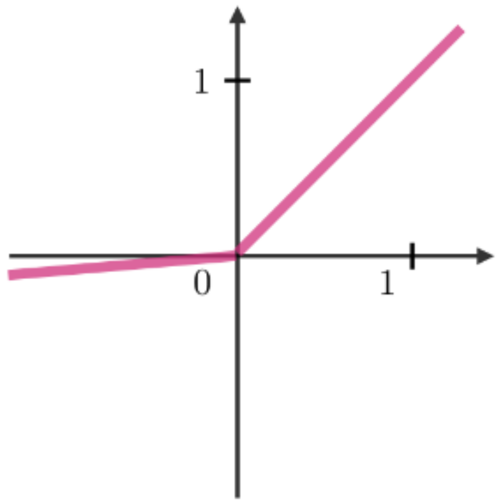
```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

if __name__ == '__main__':
    mlp_instance = MLP(5, 10, 1)
```



Activation Functions

Activation functions are used at the end of a hidden unit to introduce nonlinear complexities to the model.

$\mathbb{R} \to (0, 1)$ could output a probability.

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning

How to train your network?

- Step 1: collect training data
- Step 2: import your model (network)
- Step 3: define a loss function
- Step 4: define an optimization method (optimizer)
- Step 5: start the optimization iteratively

How to train your network?

```python
import torch

X_train = torch.randn(100, 5)
y_train = torch.randn(100, 1)

model = MLP(5, 10, 1)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

for epoch in range(100):
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 10 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

learning rate

SGD: stochastic gradient decent

Review: Basic RL concepts

$S_t \in \mathcal{S}$, $a_t \in \mathcal{A}$

$P(s'|s,a) = P(S_t = s' | S_{t-1} = s, A_t = a)$

S_t: random variable; s': value

- state, action, reward, state transition …
- policy $\pi(a|s) = P(A_t = a | S_t = s)$

  a mapping from states to probabilities of selecting each possible action.
- episode, trajectory, return …

  episode: one complete play of the agent interacting with the environment.
- state-value functions, action-value functions

  traj.: $s_0, a_0, r_1, s_1, a_1, r_2, \cdots$.
- RL objective: maximize the accumulated reward

return: accumulated (discounted) reward.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

Review: Basic RL concepts

- state-value functions: estimate how good it is for the agent to be in a given state.

$$V_\pi(s) = E_\pi[G_t \mid S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\Big|\, S_t = s\right].$$

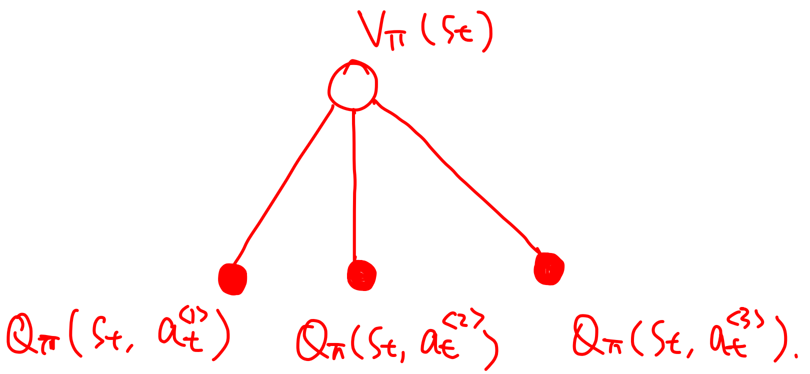
- action-value functions: estimate how good it is to perform a given action in a given state.

$$Q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\Big|\, S_t = s, A_t = a\right]$$

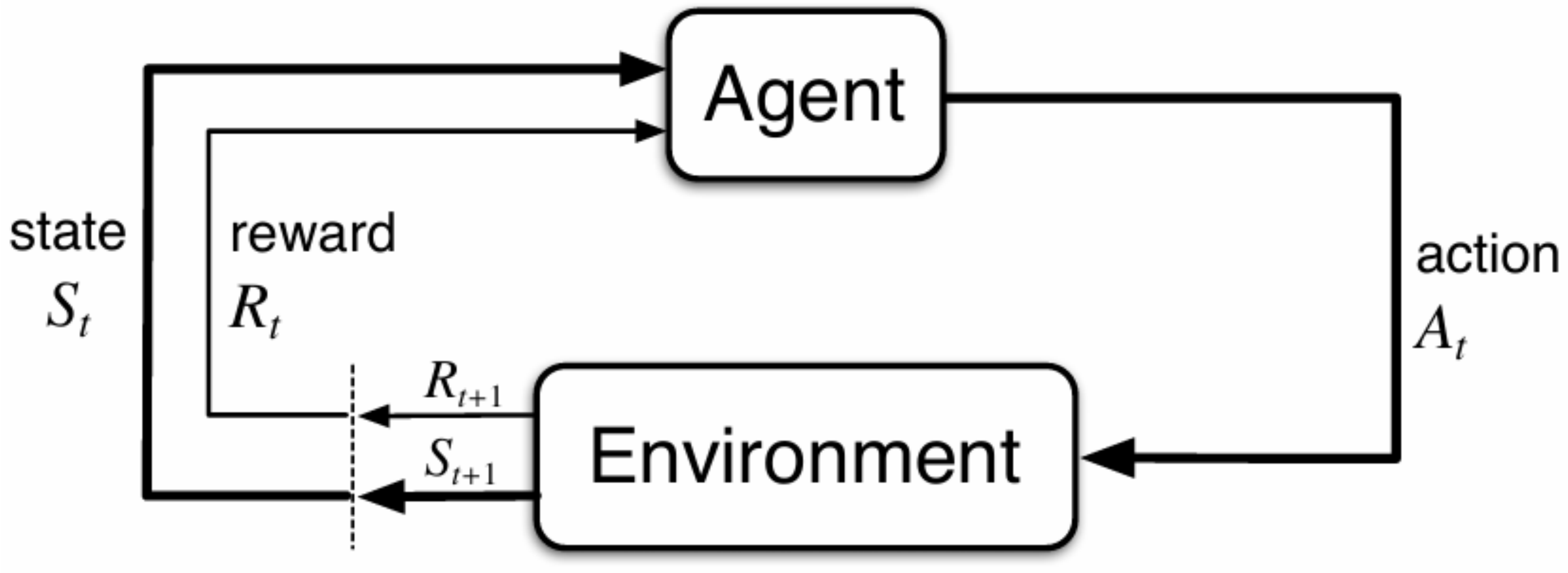
- Relationships: $\boxed{V_\pi(S_t) = \sum_{a_t} \pi(a_t \mid s_t) \cdot Q_\pi(S_t, a_t).}$

Ex. $\mathcal{A} = \{a^{(1)}, a^{(2)}, a^{(3)}\}$.

$$V_\pi(S_t) = \overset{\pi}{p}(a^{(1)} \mid s_t)\, Q_\pi(S_t, a^{(1)}) + \overset{\pi}{p}(a^{(2)} \mid S_t) \cdot Q_\pi(S_t, a^{(2)})$$
$$+ \overset{\pi}{p}(a^{(3)} \mid S_t) \cdot Q_\pi(S_t, a^{(3)}).$$



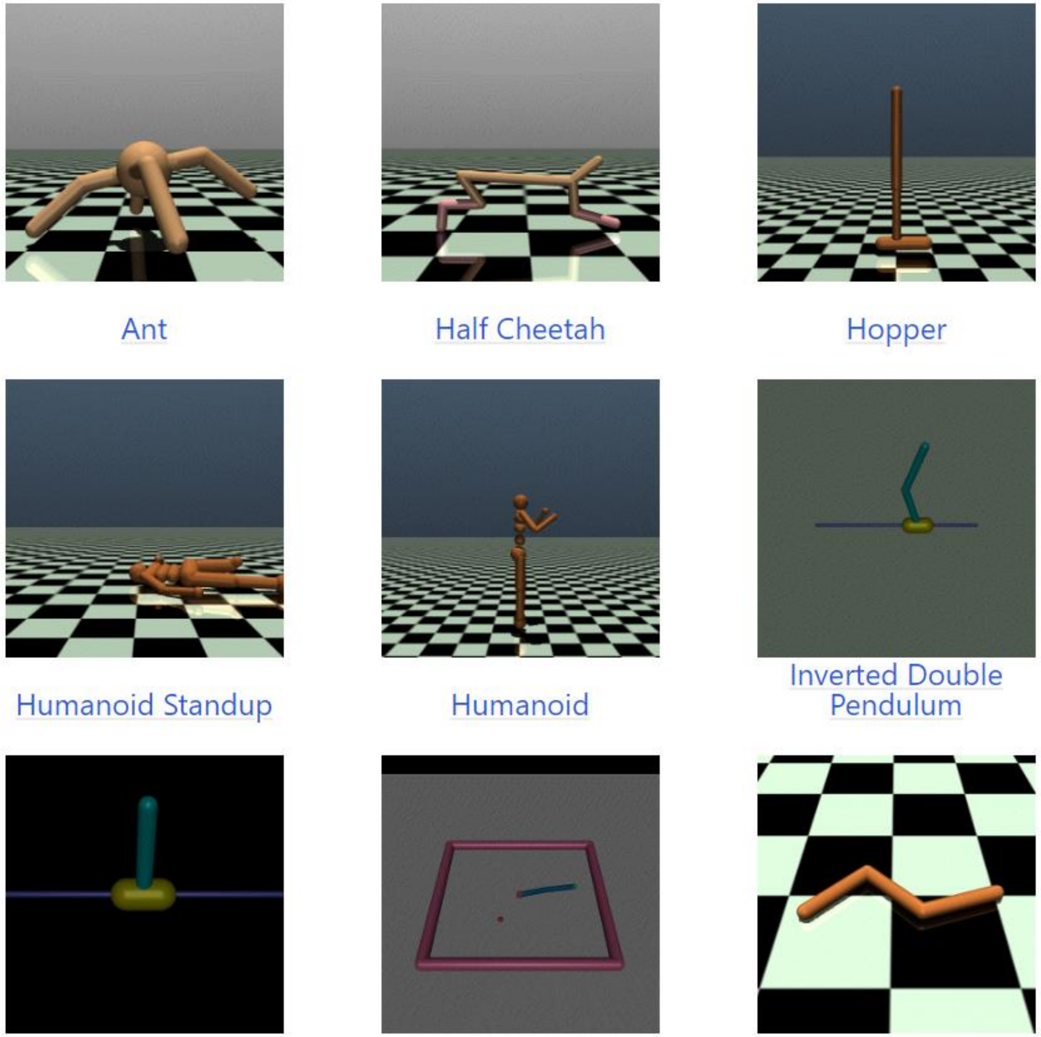
RL Training Environment



RL An Introduction, Sutton, et al.

RL Training Environment

```python
class TemplateEnv:

    def __init__(self):
        # define obs space, action space ...

    def reset(self):
        ...
        obs = self._get_observation()
        return obs

    def step(self, action):
        ...
        rewards = self.__compute_reward()
        dones = self.__get_done_info()
        obs = self.__get_observation(action)
        infos = {}
        return obs, rewards, dones, infos

    def _get_observation(self, action):
        ...
        return obs

    def _compute_rewards(self):
        ...
        return rewards

    def _get_done_infos(self):
        return dones
```



Ant Half Cheetah Hopper

Humanoid Standup Humanoid Inverted Double Pendulum

Inverted Pendulum Reacher Swimmer

https://www.gymlibrary.dev/environments/mujoco/

Exploration and Exploitation

- **Exploitation:** make the best decision given current information.

$$a_t = \arg\max_a Q(s, a)$$

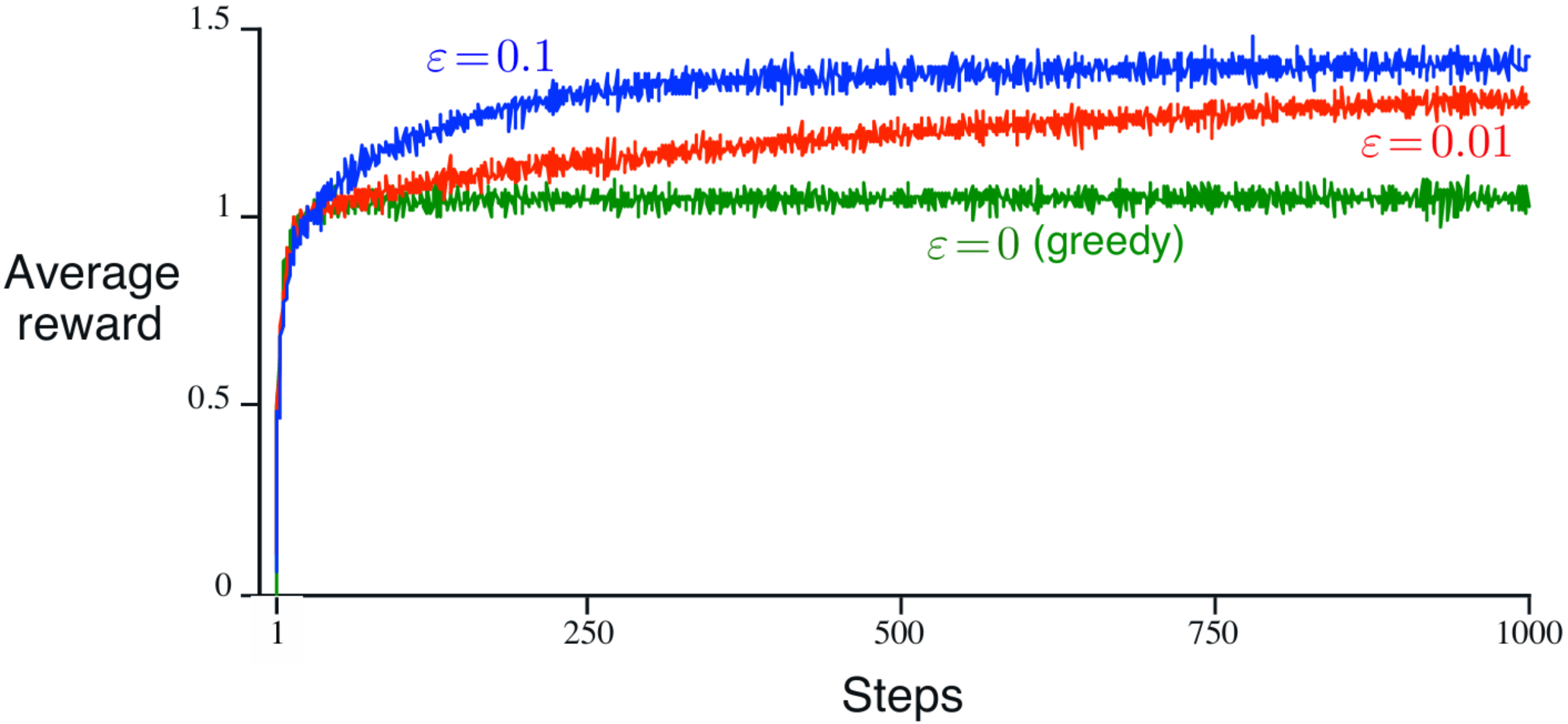
- **Exploration:** to gather more information.

$$a_t = \begin{cases} \arg\max_a Q(s, a), & \text{with probability } 1 - \epsilon \\ \text{a random action}, & \text{with probability } \epsilon \end{cases}$$

- The best long-term strategy may involve short-term sacrifices. So we need to gather enough information to make the best overall decisions.

Exploration and Exploitation

The greedy method performed significantly worse in the long run because it often got stuck performing suboptimal actions.



RL An Introduction, Sutton, et al.

Representation of a Policy

- **Numeric expression**

- **Parametric expression (with parameters determined / undetermined)**

- **Neural networks**

Define the Policy Network (Actor)

```python
import torch
import torch.nn as nn
import torch.distributions import Normal

from mlp import MLP

class Actor(nn.Module):
    def __init__(self, obs_dim, hidden_dim, action_dim, init_noise_std=1.0):
        super(Actor, self).__init__()
        self.actor = MLP(obs_dim, hidden_dim, action_dim)
        self.std = nn.Parameter(init_noise_std * torch.ones(action_dim))
        self.distribution = None

    def update_distribution(self, obs):
        mean = self.actor(obs)
        self.distribution = Normal(mean, mean * 0. + self.std)

    def act(self, obs):
        self.update_distribution(obs)
        return self.distribution.sample()

    def act_inference(self, obs):
        action_mean = self.actor(obs)
        return action_mean
```

exploration

exploitation

REINFORCE: Vanilla Policy Gradient

Recall: RL goal $\quad \max \underbrace{\mathbb{E}_{s\sim\rho_{\pi_\theta}(s),\, a\sim\pi_\theta(a|s)}[G]}_{J(\theta)}$

$V(s)$.
$Q(s,a)$
$A(s,a)$
$\ldots$

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{s\sim\rho_{\pi_\theta}(s),\, a\sim\pi_\theta(a|s)}[G].$$

$$= \mathbb{E}_{s\sim\rho_{\pi_\theta}(s)}\left[\int_{a\in A} \nabla_\theta \pi_\theta(a|s)\cdot G \, da\right]$$

$$= \mathbb{E}_{s\sim\rho_{\pi_\theta}(s)}\left[\int_{a\in A} \frac{\pi_\theta(a|s)}{\pi_\theta(a|s)} \nabla_\theta \pi_\theta(a|s)\cdot G \, da\right]$$

$$= \mathbb{E}_{s\sim\rho_{\pi_\theta}(s)}\left[\int_{a\in A} \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}\cdot G \, da\right]$$

$$= \mathbb{E}_{s\sim\rho_{\pi_\theta}(s),\, a\sim\pi_\theta(a|s)}\left[\nabla_\theta \log \pi_\theta(a|s)\cdot G\right]$$

REINFORCE: Vanilla Policy Gradient

1. Initialize the policy parameter θ at random.
2. Generate one trajectory on policy π_θ: $S_1, A_1, R_2, S_2, A_2, \dots, S_T$.
3. For t=1, 2, ... , T:
    1. Estimate the the return G_t;
    2. Update policy parameters: $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)$

REINFORCE: Vanilla Policy Gradient

```python
import torch

from actor import Actor

class REINFORCE:

    def __init__(self, state_dim, hidden_dim, action_dim, learning_rate, gamma, device):
        self.actor = Actor(state_dim, hidden_dim, action_dim).to(device)
        self.optimizer = torch.optim.Adam(self.actor.parameters(), lr=learning_rate)
        self.gamma = gamma
        self.device = device

    def update(self, rollout_buffer):
        G = 0
        self.optimizer.zero_grad()
        for i in reversed(range(len(rollout_buffer.reward_list))):
            reward = rollout_buffer.reward_list[i]
            obs = rollout_buffer.obs_list[i, :]

            action_log_prob = torch.log(self.actor(obs))
            G = self.gamma * G + reward
            loss = -action_log_prob * G
            loss.backward()
        self.optimizer.step()
```

Training an Agent

```python
import gym
import torch

from rollout_buffer import RolloutBuffer

# hyper-params: lr, num_episodes, hidden_dim, gamma, device, ...
env = gym.make('CartPole-v0') # make environment
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
rollout_buffer = RolloutBuffer()
rl_alg = REINFORCE(state_dim, hidden_dim, action_dim, learning_rate, gamma, device)

return_list = [] # for recording
for episode_idx in range(10):
    episode_return = 0
    obs = env.reset()
    done = False
    while not done:
        action = rl_alg.actor.act(obs)
        next_obs, reward, done, _ = env.step(action)
        rollout_buffer.add(obs, action, next_obs, reward, done)
        obs = next_obs
        episode_return += reward
    return_list.append(episode_return)
    rl_alg.update(rollout_buffer)
    print('episode = {}, return = {}'.format(episode_idx, return_list[-1]))
```

Variance Reduction: Baselines

- **Approach:** subtract a baseline value from the return.

$$g = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right],$$

where Ψ_t may be one of the following:

1. $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory.
2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t.
3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of previous formula.
4. $Q^\pi(s_t, a_t)$: state-action value function.
5. $\boxed{A^\pi(s_t, a_t)\text{: advantage function.}}$
6. $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$: TD residual.

Increasing the likelihood of actions that do *better* than the average return at each state, and **decreasing** … *worse* …

- **A common approach:** subtract the state-value from action-value.

  How much better than average a given action is compared to the average return at a particular state.

- It can be shown that the baseline does not change the policy gradient.

- How to estimate the advantage? Generalized Advantage Estimation (GAE).

GAE, Schulman, et al. ICLR 2016.

Value Estimation: Monte-Carlo

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow$ average$(Returns(S_t))$

RL An Introduction, Sutton, et al.

Value Estimation: Temporal Difference

- **Incremental implementation for updating the value estimation**

- **General equation for value estimation update**

- **Temporal difference methods**

- **TD target, TD error …**

Value Estimation: Temporal Difference

- traj : $s_0, a_0, r_1, s_1, a_1, r_2, \cdots$ $\leftarrow$
- estimation of value functions : (via Monte-Carlo)

average of all measurements : $V_\pi^{[m]}(s_t) = g_t^{[1]} + \cdots + g_t^{[m]}$

m-th estimation of $V_\pi(s_t)$.

m-th measurement of $V_\pi(s_t)$.

current measurement.

$g_t \quad V(s_t) = r_{t+1} \qquad g_t$

$g_{t-1} \quad V(s_{t-1}) = r_t + \gamma r_{t+1} \qquad r_t + \gamma g_t$

$g_{t-2} \quad V(s_{t-2}) = r_{t-1} + \gamma r_t + \gamma^2 r_{t+1} \qquad r_{t-1} + \gamma g_{t-1}$

need to store every estimations !

- Incremental implementation :

$$V_\pi^{[m+1]}(s_t) = \frac{1}{m+1}\sum_{i=1}^{m+1} g_t^{[i]}$$

$$= \frac{1}{m+1}\left(g_t^{[m+1]} + \sum_{i=1}^{m} g_t^{[i]}\right)$$

$$= \frac{1}{m+1} g_t^{[m+1]} + \frac{1}{m+1}\cdot m \cdot \frac{1}{m}\cdot\sum_{i=1}^{m} g_t^{[i]}$$

$$= \frac{1}{m+1} g_t^{[m+1]} + \frac{m}{m+1}\cdot V_\pi^{[m]}(s_t)$$

$$= \frac{1}{m+1}\left(g_t^{[m+1]} + \underbrace{m}_{m+1-1}\cdot V_\pi^{[m]}(s_t)\right)$$

$$= \frac{1}{m+1}\left((m+1)V_\pi^{[m]}(s_t) + g_t^{[m+1]} - V_\pi^{[m]}(s_t)\right)$$

$$= V_\pi^{[m]}(s_t) + \frac{1}{m+1}\left(g_t^{[m+1]} - V_\pi^{[m]}(s_t)\right).$$

just need to store the last estimation.

Value Estimation: Temporal Difference

$$V_\pi^{[m+1]}(s_t) = V_\pi^{[m]}(s_t) + \frac{1}{m+1}\left(g_t^{[m+1]} - V_\pi^{[m]}(s_t)\right)$$

MC- estimation

New Estimation = Old Estimation + α (New Measurement − Old Estimation)

Now focusing on the new measurement:

Monte-Carlo: $g_t = r_{t+1} + \underbrace{\gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots}_{\gamma g_{t+1}}$ must wait until the end of the episode

Temporal-Difference: $\approx r_{t+1} + \gamma \tilde{V}_\pi(s_{t+1})$ need to wait only until the next time step.

using last estimation to replace. could be $V_\pi^{[m]}(s_{t+1})$.

TD target

$$V_\pi^{[m+1]}(s_t) = V_\pi^{[m]}(s_t) + \alpha \Big(\underbrace{r_{t+1}^{[m+1]} + \gamma \tilde{V}_\pi(s_{t+1}) - V_\pi^{[m]}(s_t)}_{\text{TD- error}} \Big)$$

TD- estimation

Value Estimation: Temporal Difference

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\textit{terminal}) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

RL An Introduction, Sutton, et al.

Define the Value Network (Critic)

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Critic(nn.Module):

    def __init__(self, obs_dim, hidden_dim):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(obs_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, 1)

    def forward(self, obs):
        x = F.relu(self.fc1(obs))
        return self.fc2(x)
```

REINFORCE with Baseline – Actor Critic

Algorithm 1 Vanilla Policy Gradient Algorithm

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.
8: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
9: **end for**

https://spinningup.openai.com/en/latest/algorithms/vpg.html

REINFORCE with Baseline – Actor Critic

```
import torch
import torch.nn.functional as F

from actor import Actor
from critic import Critic
from gae import GAE

class ActorCritic:

    def __init__(self, obs_dim, hidden_dim, action_dim, actor_lr, critic_lr, gamma, lmbda, device):
        self.actor = Actor(obs_dim, hidden_dim, action_dim).to(device)
        self.critic = Critic(obs_dim, hidden_dim).to(device)
        # initialize optimizers, parameters …

    def update(self, obs, rewards, next_obs, dones):
        td_target = rewards + self.gamma * self.critic(next_obs) * (1 - dones)
        td_error = td_target - self.critic(obs)
        advantage = GAE(self.gamma, self.lmbda, td_error)

        log_probs = torch.log(self.actor(obs))
        actor_loss = torch.mean(-log_probs * advantage.detach())
        critic_loss = torch.mean(F.mse_loss(self.critic(obs), td_target.detach()))

        self.actor_optimizer.zero_grad(); self.critic_optimizer.zero_grad()
        actor_loss.backward(); critic_loss.backward()
        self.actor_optimizer.step(); self.critic_optimizer.step()
```

More Discussions

If time permits:

- Importance Sampling
- Trust Region Policy Optimization (TRPO)
- Proximal Policy Optimization (PPO)