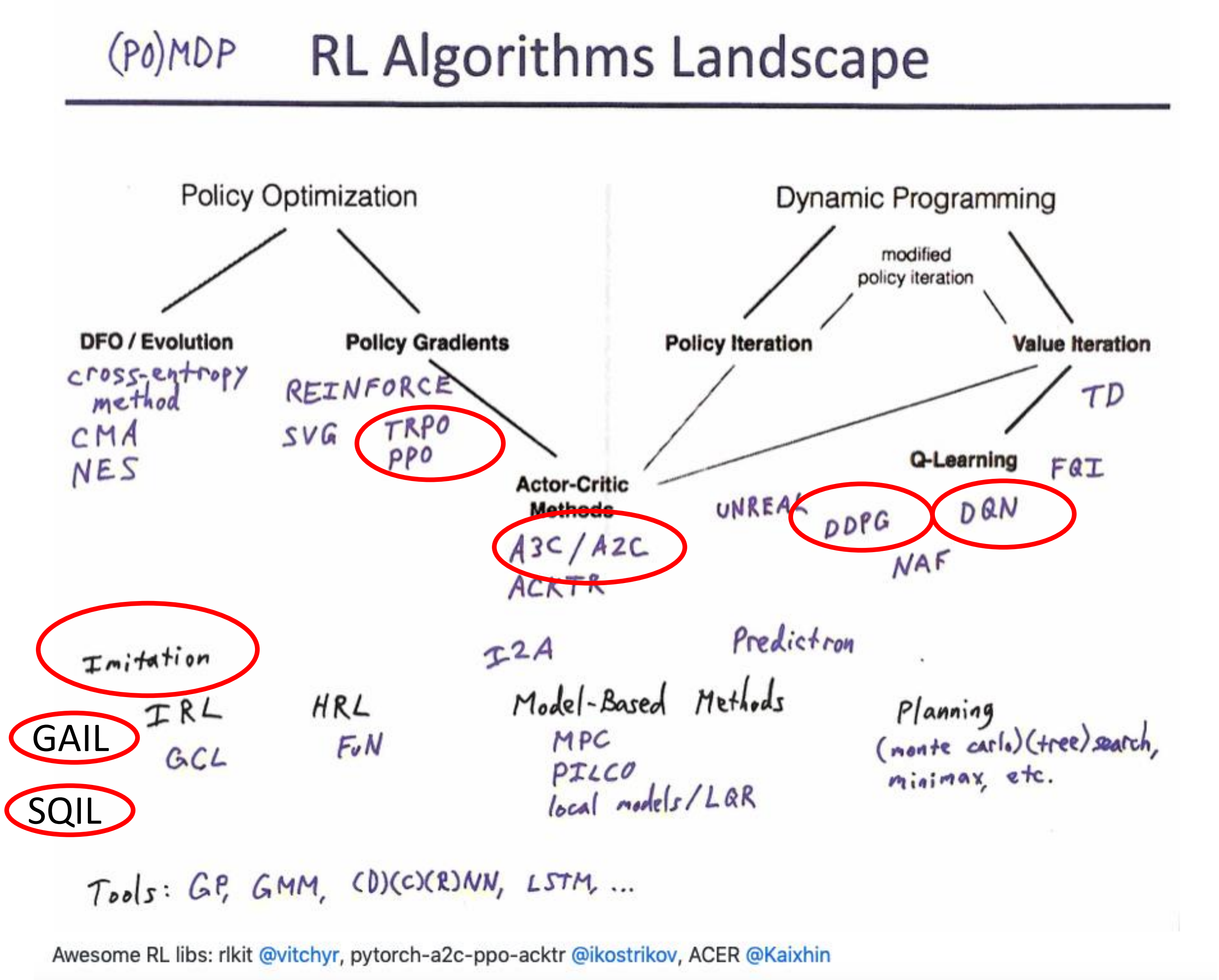
Reinforcement Learning algorithms



Reference: https://images.app.goo.gl/nPJftvi3CChYmyFDA

Reinforcement Learning with PPO

PyTorch implementation of PPO (AC) from RL-Advanture-2

https://github.com/higgsfield/RL-Adventure-2/blob/master/3.ppo.ipynb

https://github.com/TienLungSun/Reinforcement-Learning-Mobile-Robot-/tree/main/LearnPPO-AC

PPO training using Unity, ML Agents 1.0.4, PyTorch and Tensorboard

https://youtu.be/Fz0v-aLW-6k

Train and test a mobile robot that learns to reach goal using PPO from ML Agent 10

https://youtu.be/aNPAP3v0gHc

https://youtu.be/K3mN6CDPRGc (English)

Test a mobile robot that reachs goal while avoiding obstacle using MLAgent R.10 (PPO)

https://youtu.be/mogi-8_aBuE (English)

https://youtu.be/ygmKfI5f1uM

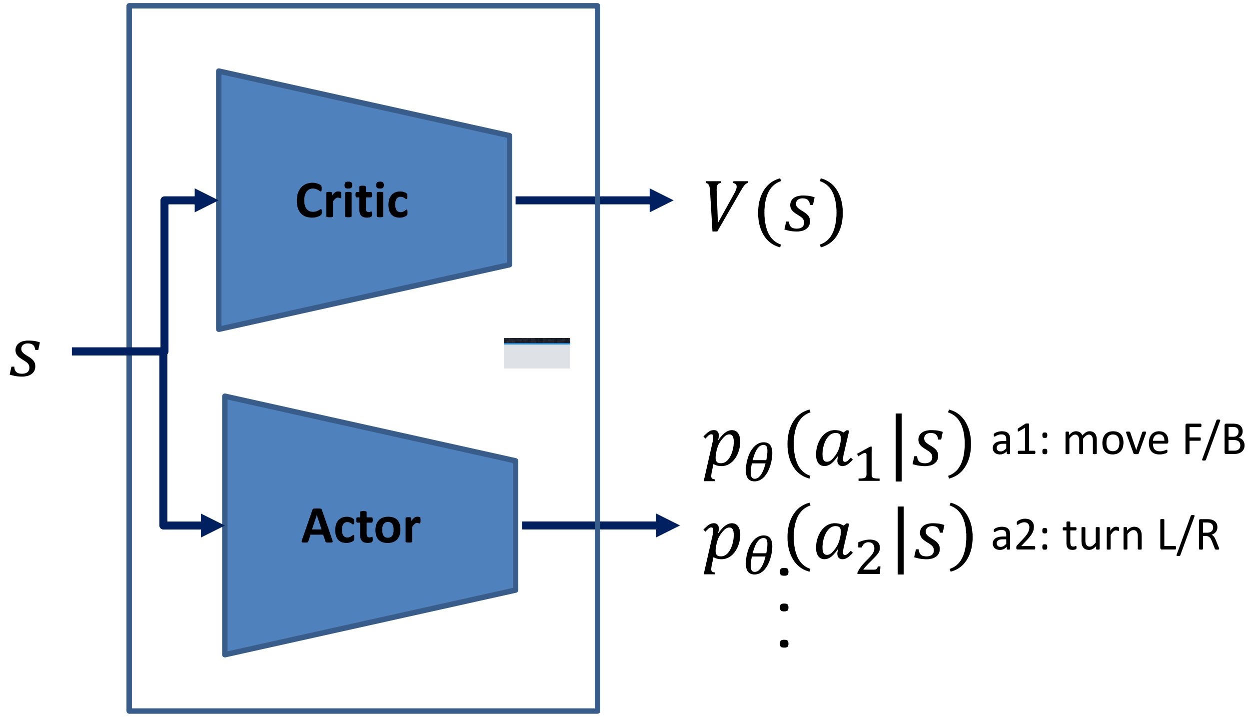
Actor critic NN

```
In [3]: from torch.distributions import Normal

        class ActorCritic(nn.Module):
            def __init__(self, num_inputs, num_outputs, hic
                super(ActorCritic, self).__init__()

                self.critic = nn.Sequential(
                    nn.Linear(num_inputs, hidden_size1),
                    nn.LayerNorm(hidden_size1),
                    nn.Tanh(),
                    nn.Linear(hidden_size1, hidden_size2),
                    nn.LayerNorm(hidden_size2),
                    nn.Tanh(),
                    nn.Linear(hidden_size2, 1),
                )

                self.actor = nn.Sequential(
                    nn.Linear(num_inputs, hidden_size1),
                    nn.LayerNorm(hidden_size1),
                    nn.Tanh(),
```



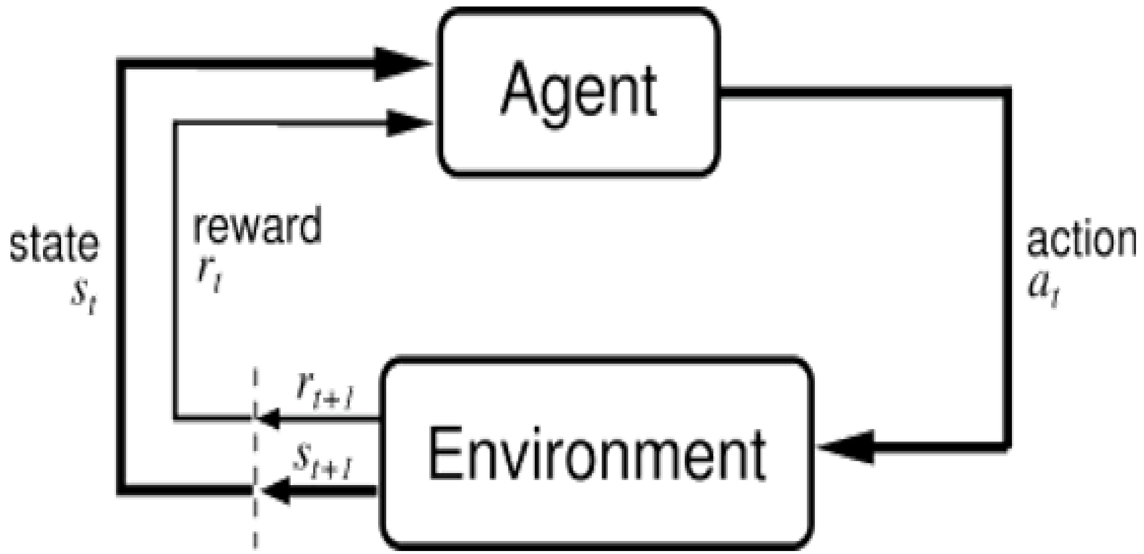
李宏毅 DRL Lectures
https://www.youtube.com/watch?v=z95ZYgPgXOY&list=PLJV_el3uVTsODxQFgzMzPLa16h6B8kWM_

Train the NN

```
In [18]: frame_idx  = 0
         max_frames = 5000   #15000
         env.reset()
         early_stop = False
         __printDetails = False

         while frame_idx < max_frames and not early_stop: #
             if(__printDetails):
                 print("Frame = ", frame_idx, end=", ")
             log_probs = []
             values    = []
             states    = []
             actions   = []
             rewards   = []
             masks     = []
             entropy = 0

             step_result = env.get_steps(behaviorName)
             DecisionSteps = step_result[0]
             state = DecisionSteps.obs[0]
             if(__printDetails):
                 print("step", end = ":")
             for step in range(num_steps):
                 if(__printDetails and (step+1) % 5==0):
                     print(step+1, end = ", ")
                 state = torch.FloatTensor(state).to(device
                 dist, value = model(state)
```



(Sutton and Barto, 1998)

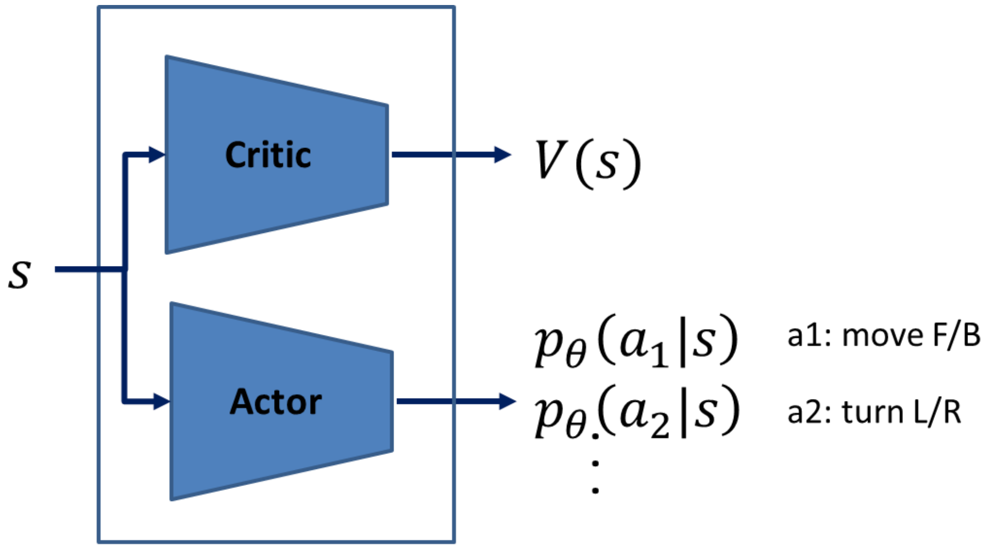
Sampling data

$$(\vec{s}_1, \vec{a}_1, v_1, r_1, log\vec{p}_1, \vec{s}_2)$$

$$(\vec{s}_2, \vec{a}_2, v_2, r_2, log\vec{p}_2, \vec{s}_3)$$

$$\vdots$$

$$(\vec{s}_{20}, \vec{a}_{20}, v_{20}, r_{20}, log\vec{p}_{20}, \vec{s}_{21}) \ v_{21}$$



Compute GAE

In [4]:
```
def compute_gae(next_value, rewards, masks, values, gamma=0.99, tau=0.95):
    values = values + [next_value]
    gae = 0
    returns = []
    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
        gae = delta + gamma * tau * masks[step] * gae
        returns.insert(0, gae + values[step])
    return returns
```

$(\vec{s}_1, \vec{a}_1, v_1, r_1, log\vec{p}_1, \vec{s}_2)$

$(\vec{s}_2, \vec{a}_2, v_2, r_2, log\vec{p}_2, \vec{s}_3)$

$\vdots$

$(\vec{s}_{20}, \vec{a}_{20}, v_{20}, r_{20}, log\vec{p}_{20}, \vec{s}_{21})\ v_{21}$

$\Delta_{19} = r_{19} + \gamma * v_{20} * mask_{19} - v_{19}$

$gae_{19\sim20} = \Delta_{19} + \gamma * \tau * mask_{19} * gae_{20}$

$return_{19} = gae_{19\sim20} + v_{19}$

…

$\Delta_{20} = r_{20} + \gamma * v_{21} * mask_{20} - v_{20}$

$gae_{20} = \Delta_{20} + \gamma * \tau * mask_{20} * gae_{initial}$

$return_{20} = gae_{20} + v_{20}$

$\Delta_1 = r_1 + \gamma * v_2 * mask_1 - v_1$

$gae_{1\sim20} = \Delta_1 + \gamma * \tau * mask_1 * gae_{2\sim20}$

$return_1 = gae_{1\sim20} + v_1$

Data collected from all agents

```
next_state = torch.FloatTensor(next_state).to(device)
_, next_value = model(next_state)
returns = compute_gae(next_value, rewards, masks, values)

returns   = torch.cat(returns).detach()
log_probs = torch.cat(log_probs).detach()
values    = torch.cat(values).detach()
states    = torch.cat(states)
actions   = torch.cat(actions)
advantage = returns - values
```

$$N * 20 \quad \begin{bmatrix} \vec{s}_{1,step1} \\ \vdots \\ \vec{s}_{N,step1} \\ \vdots \\ \vec{s}_{1,step20} \\ \vdots \\ \vec{s}_{N,step1} \end{bmatrix} \begin{bmatrix} \vec{a}_{1,step1} \\ \vdots \\ \vec{a}_{N,step1} \\ \vdots \\ \vec{a}_{1,step20} \\ \vdots \\ \vec{a}_{N,step1} \end{bmatrix} \begin{bmatrix} \overrightarrow{logp}_{1,step1} \\ \vdots \\ \overrightarrow{logp}_{N,step1} \\ \vdots \\ \overrightarrow{logp}_{1,step20} \\ \vdots \\ \overrightarrow{logp}_{N,step1} \end{bmatrix} \begin{bmatrix} v_{1,step1} \\ \vdots \\ v_{N,step1} \\ \vdots \\ v_{1,step20} \\ \vdots \\ v_{N,step20} \end{bmatrix} \begin{bmatrix} return_{1,step1} \\ \vdots \\ return_{N,step1} \\ \vdots \\ return_{1,step20} \\ \vdots \\ return_{N,step20} \end{bmatrix} \begin{bmatrix} gae_{1,step1} \\ \vdots \\ gae_{N,step1} \\ \vdots \\ gae_{1,step20} \\ \vdots \\ gae_{N,step20} \end{bmatrix}$$

Sampling a batch of data to train NN

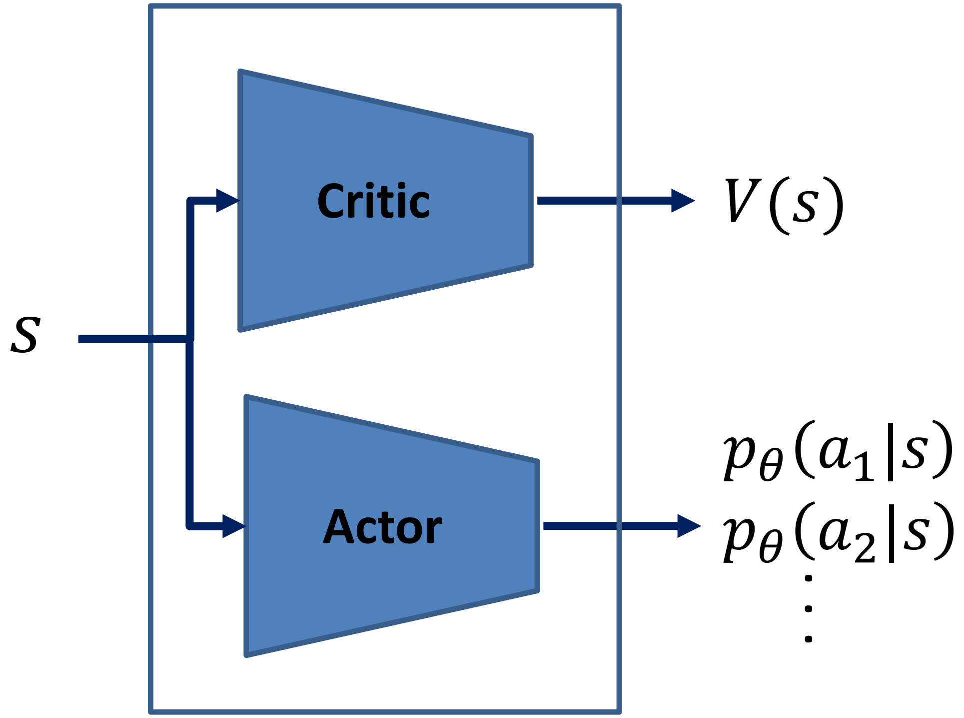
```
In [5]: import numpy as np

        def ppo_iter(mini_batch_size, states, actions, log_probs, returns, advantage):
            batch_size = states.size(0)
            for _ in range(batch_size // mini_batch_size):
                rand_ids = np.random.randint(0, batch_size, mini_batch_size)
                yield states[rand_ids, :], actions[rand_ids, :], log_probs[rand_ids, :]
```

$$
\begin{bmatrix} \vec{s}_{1,step1} \\ \vdots \\ \vec{s}_{N,step1} \\ \vdots \\ \vec{s}_{1,step20} \\ \vdots \\ \vec{s}_{N,step1} \end{bmatrix}
\begin{bmatrix} \vec{a}_{1,step1} \\ \vdots \\ \vec{a}_{N,step1} \\ \vdots \\ \vec{a}_{1,step20} \\ \vdots \\ \vec{a}_{N,step1} \end{bmatrix}
\begin{bmatrix} \overrightarrow{logp}_{1,step1} \\ \vdots \\ \overrightarrow{logp}_{N,step1} \\ \vdots \\ \overrightarrow{logp}_{1,step20} \\ \vdots \\ \overrightarrow{logp}_{N,step1} \end{bmatrix}
\begin{bmatrix} return_{1,step1} \\ \vdots \\ return_{N,step1} \\ \vdots \\ return_{1,step20} \\ \vdots \\ return_{N,step20} \end{bmatrix}
\begin{bmatrix} gae_{1,step1} \\ \vdots \\ gae_{N,step1} \\ \vdots \\ gae_{1,step20} \\ \vdots \\ gae_{N,step20} \end{bmatrix}
\quad N * 20
$$

PPO update

In [6]:
```
def ppo_update(ppo_epochs, mini_batch_size, sta
    for _ in range(ppo_epochs):
        for state, action, old_log_probs, retur
vantages):
            dist, value = model(state)
            entropy = dist.entropy().mean()
            new_log_probs = dist.log_prob(actio
```



$$L = c_v L_v + L_\pi + c_{reg} L_{reg}$$

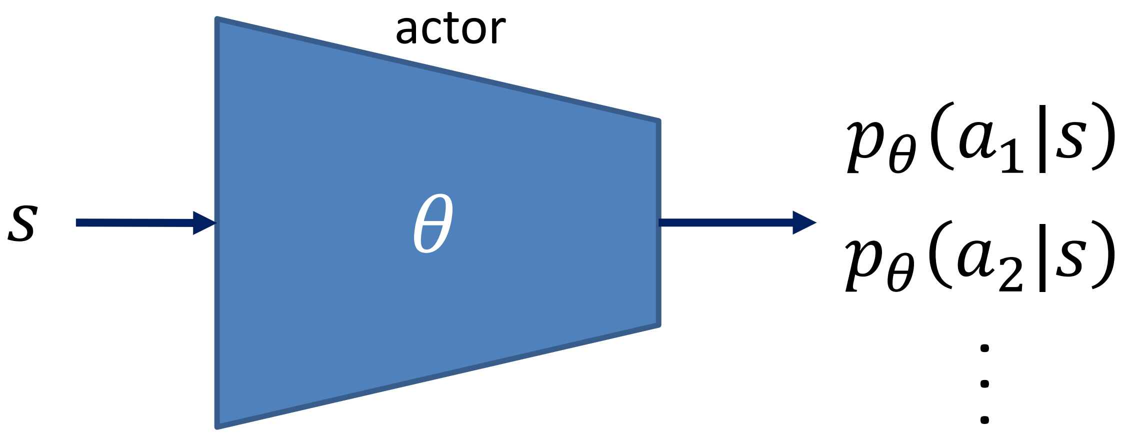
$$L_v = MSE(return - v)$$

$$return_i = gae_{i\sim 20} + v_i$$

$$gae_{i\sim 20} = \Delta_i + \gamma * \tau * mask_i * gae_{i+1\sim 20}$$

$$\Delta_i = r_i + \gamma * v_{i+1} * mask_i - v_i$$

Use $\nabla\bar{R}_\theta$ to update policy network



$$\theta^{\pi'} \leftarrow \theta^{\pi} + \eta\nabla\bar{R}_\theta$$

Policy gradient $\nabla\bar{R}_\theta$

$$\tau = \left(s_1, a_1, r_1, s_2, a_2, r_{2,} \cdots s_{\mathrm{T}}, a_{\mathrm{T}}\right)$$

$$p_\theta(\tau) = p(s_1)p_\theta(a_1|s_1)p(s_2|s_1, a_1)p_\theta(a_2|s_2)p(s_3|s_2, a_2)\cdots$$

$$\mathrm{R}(\tau) = \sum_{\mathrm{t}=1}^{\mathrm{T}} \mathrm{r_t}$$

$$\bar{R}_\theta = \sum R(\tau)\, p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)]$$

Max $E[\bar{R}_\theta]$

$$\max_\theta E[\bar{R}_\theta]$$

Gradient of the expected value

$$\nabla\bar{R}_\theta = \sum R(\tau)\,\nabla p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla \log p_\theta(\tau)] \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla \log p_\theta(\tau^n)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log p_\theta(a_t^n|s_t^n)$$

Tips to calculate $\nabla \bar{R}_\theta$

$$\nabla \bar{R}_\theta = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log p_\theta(a_t^n|s_t^n)$$

Add a baseline to calculate the reward

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} (R(\tau^n) - b)\nabla \log p_\theta(a_t^n|s_t^n), \qquad b \approx E[R(\tau)]$$

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} \left(\sum_{t'}^{T_n} r_{t'}^n - b\right)\nabla \log p_\theta(a_t^n|s_t^n)$$

Assign suitable time delayed credit

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} \left(\sum_{t'}^{T_n} \gamma^{t'-t} r_{t'}^n - b\right)\nabla \log p_\theta(a_t^n|s_t^n), \gamma < 1$$

$$A^\theta(s_t, a_t) = \left(\sum_{t'}^{T_n} \gamma^{t'-t} r_{t'}^n - b\right)$$

Off-policy $\nabla\bar{R}_\theta$

On-policy

$$\nabla\bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A^\theta(s_t,a_t)\nabla\log p_\theta(a_t^n|s_t^n), \gamma < 1 \qquad A^\theta(s_t,a_t) = \left(\sum\nolimits_{t'}^{T_n}\gamma^{t'-t}r_{t'}^n - b\right)$$

Importance sampling

$$E_{x\sim\mathrm{p}}[f(x)] = E_{x\sim\mathrm{q}}\left[f(x)\frac{p(x)}{q(x)}\right]$$

$$Var_{x\sim\mathrm{q}}\left[f(x)\frac{p(x)}{q(x)}\right] = E_{x\sim\mathrm{q}}\left[\left(f(x)\frac{p(x)}{q(x)}\right)^2\right] - \left(E_{x\sim\mathrm{q}}\left[f(x)\frac{p(x)}{q(x)}\right]\right)^2$$

$$= E_{x\sim\mathrm{p}}\left[f(x)^2\frac{p(x)}{q(x)}\right] - \left(E_{x\sim\mathrm{p}}[f(x)]\right)^2$$

Off-policy

$$\nabla\bar{R}_\theta = E_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\nabla\log p_\theta(a_t^n|s_t^n)\right]$$

Loss function of $\nabla\bar{R}_\theta$

Off-policy

$$\nabla\bar{R}_\theta = E_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\nabla\log p_\theta(a_t^n|s_t^n)\right]$$

Sampling efficiency

Loss function

$$J^{\theta'}(\theta) = E_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\right]$$

Proximal policy optimization (PPO)

$$J_{PPO}^{\theta'}(\theta) = J^{\theta'}(\theta) - \beta KL(\theta,\theta')$$

$$J_{PPO2}^{\theta'}(\theta) = \sum_{(s_t,a_t)} min\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t), clip\left(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right)A^{\theta'}(s_t,a_t)\right)$$

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Actor-critic strategy to calculate $\nabla \bar{R}_\theta$

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}\left(\sum_{t'}^{T_n}\gamma^{t'-t} r_{t'}^n - b\right)\nabla \log p_\theta(a_t^n|s_t^n)$$

$$G_t^n = \sum_{t'}^{T_n}\gamma^{t'-t} r_{t'}^n$$

unstable when sampling amount is not large enough

$$\boxed{V^{\pi_\theta}(s_t^n)}$$ Expected value of b

Use expected value to reduce sampling variance

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}\left(\boxed{\sum_{t'}^{T_n}\gamma^{t'-t} r_{t'}^n} - \boxed{b}\right)\nabla \log p_\theta(a_t^n|s_t^n)$$

$$\boxed{E[G_t^n] = Q^{\pi_\theta}(s_t^n, a_t^n)}$$ Expected value of G_t^n

Use one neural network that estimates V

$$Q^{\pi_\theta}(s_t^n, a_t^n) = \mathrm{E}[r_t^n + V^{\pi_\theta}(s_{t+1}^n)] = r_t^n + V^{\pi_\theta}(s_{t+1}^n)$$

$$Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n) = r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)$$

$$\boxed{A^\theta(s_t, a_t) = \left(r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)\right)}$$

Temporal difference to calculate V

$$A^{\theta}(s_t, a_t) = \left(r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)\right)$$

Monte-Carlo approach $V^{\pi_\theta}(s_a) \leftrightarrow G_a$ Until the end of the episode, the cumulated reward is G_a

Temporal-difference approach

$$V^{\pi_\theta}(s_t) + r_t = V^{\pi_\theta}(s_{t+1})$$

$$V^{\pi_\theta}(s_t) - V^{\pi_\theta}(s_{t+1}) \leftrightarrow r_t$$