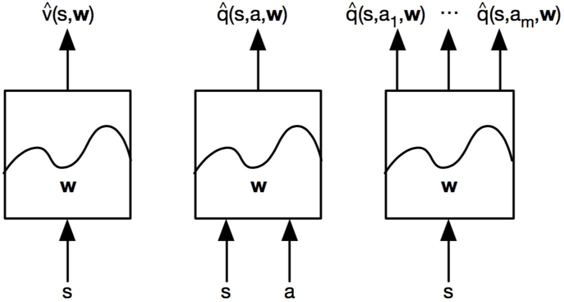
香港中文大學
The Chinese University of Hong Kong

Reinforcement Learning: function approximation

Zihao DENG
Wenqian ZHAO

Department of Computer Science and Engineering
The Chinese University of Hong Kong

Value Function Approximation



- Tablular methods: impossible to record all states for real word problems
- Function approximation: generalize from seen states to unseen states

Value Function Approximation

- Goal: find parameter vector w minimising mean-squared error between approximate value function $\hat{v}(S,w)$ and true value function $v_\pi(S)$

$$J(w) = ||v_\pi(S) - \hat{v}(S,w)||_2^2 \tag{1}$$

- Stochastic gradient descent samples the gradient

$$\Delta w = \alpha(v_\pi(s) - \hat{v}(S,w))\nabla_w \hat{v}(S,w) \tag{2}$$

- In reality we don't have the true value function $v_\pi(S)$
  - For Monte-Carlo, use discounted return G_t
  - For TD, use $R_{t+1} + \lambda \hat{v}(S_{t+1}, w)$

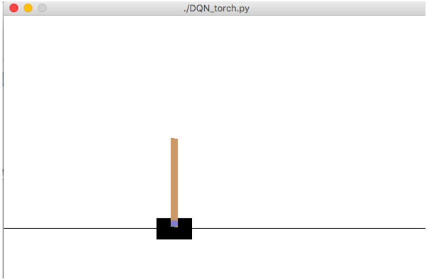
Deep Q-Networks (DQN)

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in memory D
- Sample random mini-batch of transitions (s, a, r, s') from D
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$L(w) = \mathrm{E}_{s,a,s,r' \sim D_i}[(r + \gamma \max_{a'} Q(s', a'; w^-) - Q(s, a; w))^2] \tag{3}$$

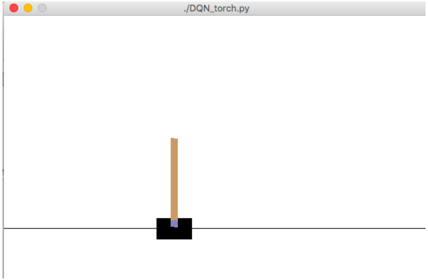
- Two important tricks in ensuring convergence: experience replay and fixed target

Deep Q-Networks(DQN): play games in OpenAI gym



- States are represented by 4-element tuples (position,cart velocity,angle,tip velocity)
- Actions can be either moving left or right
- Function approximator is a feed foward neural network
- 1 hidden layer with 10 neurons, 2 output neurons representing value estimation for two actions
- Implemented using torch and tensorflow, can stay alive for 1 minute

Deep Q-Networks(DQN): play games in OpenAI gym



- States are represented by 4-element tuples (position,cart velocity,angle,tip velocity)
- Actions can be either moving left or right
- Function approximator is a feed foward neural network
- 1 hidden layer with 10 neurons, 2 output neurons representing value estimation for two actions
- Implemented using torch and tensorflow, can stay alive for 1 minute

Markov Decision Process

- The MDP gives us a precise formulation of the environment, given a state st we select an action at and observe s_{t+1} and r_t according to the transition probabilities P:
  - S - set of possible states
  - $s_t \in S$ - state at step t
  - A - set of possible actions
  - $a_t \in A$ - selected action at step t
  - R - Reward function. The reward at step t is given by $r_{t+1} = R(s_t, a_t, s_{t+1})$
  - P - transition probabilities such that $s_{t+1}P(s|s_t, a_t)$, i.e.
  - ρ - Initial state distribution such that $s_0\rho(s)$
- Agent is defined with a policy function $\pi(a|s)$, mapping from states to actions and can be either deterministic or non-deterministic

Markov Decision Process

- Given an MDP and a policy, an episode can be produced by repeating of:
  - $a_t \pi(a|s_t)$
  - $s_{t+1} P(s|s_t, a_t)$
  - $r_{t+1} = r(s_t, a_t, s_{t+1})$
- which produce:

$$\text{episode} := s_0, a_0, r_1, s_1, a_1, r_1, \ldots, s_{\tau 1}, a_{\tau 1}, r_{\tau 1}, s_\tau$$

- Optimal Solution gives:

$$\max_\pi E[\sum_{t=1}^{\tau} r_t] \tag{4}$$

Markov Decision Process

- Value function defined as :

$$V^{\pi}(s) = E_{\pi}[\sum_{t=1}^{\tau} r_t | s_0 = s] \tag{5}$$

$$V^{*}(s) = \max_{\pi} E_{\pi}[\sum_{t=1}^{\tau} r_t | s_0 = s] \tag{6}$$

- Bellman equation: A recursive relation for value function:

$$V^{*}(s) = \max_{a \in A} E[r_{t+1} + V^{*}_{s_{t+1}} | s_t = s, a_t = t] \tag{7}$$

- $(TV)(s)$ is the Bellman operator and we can recursively calculate it and update $V(s)$ (value iteration) to reach optimal (Monotonicity and Contraction mapping)

$$(TV)(s) = \max_{a \in A} E[r_{t+1} + V^{*}_{s_{t+1}} | s_t = s, a_t = t] \tag{8}$$

$$V_{k+1} = TV_k \tag{9}$$

- policy iteration use the same idea, but instead of updating value function, it update Policy π

Markov Decision Process

- Another approach: State-Value Function: define a quantity $Q : S \times A \to \mathbb{R}$:

$$Q^{\pi}(s,a) = \bar{R}(s,a) + \sum_{s' \in S}^{T} P_{s,a}(s') V^{\pi}(s') \tag{10}$$

- Recursively Calculate optimal by using Bellman Operator:
  - $FQ(s,a) = \bar{R}(s,a) + \sum_{s' \in S}^{T} P_{s,a}(s') \max_{a' \in A} Q(s',a')$
  - $Q(s,a) = FQ(s,a)$

- Greedy action selection is simple:

$$\pi(s) = \arg\max_{a' \in A} Q(s_{t+1}, a') \tag{11}$$

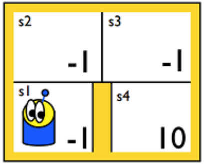
MDP vs RL

- Difference between Markov Decision Process and Reinforcement learning:
  - MDP: the transition matrix $P(s'|s,a)$ is known $\rightarrow$ used to find the optimal agent
  - RL: $P(s'|s,a)$ unknown and need to be learned :
    i. Iteracting with environment
    ii. Requiring explicit knowledge of P
- But the same idea of state-action value function can be used for Reinforcement learning

Q-learning and SARSA

- Q-learning:

  i. Build a Q-table which stores $Q(s, a)$ for each s and a (randomly initialized). i.e.



$\alpha = .7$

	↑	↓	←	→
S_1	0	0	0	0
S_2	0	0	0	0
S_3	0	0	0	0
S_4	0	0	0	0

Q-Table

Q-learning and SARSA

- Q-learning:

  ii. update $Q(s,a)$ with:

$$Q_{k+1} := (1-\gamma_k)Q_k + \gamma_k(r + \max_{a' \in A} Q_k(s', a')) \tag{12}$$

where γ_k is the learning rate, with $\sum_{k=1}^{\infty} \gamma_k = \infty$ and $\sum_{k=1}^{\infty} \gamma_k^2 < \infty$:

$$Q_{k+1} = Q_k + \gamma_k(r + \max_{a' \in A} Q_k(s', a') - Q_k(s,a)) \tag{13}$$

- Q-learning Demo: google

Q-learning and SARSA

- Q-learning result:

Q-learning and SARSA

- Exploration: $\epsilon - greedy$

$$a_t = \begin{cases} \arg\max_{a \in A} Q_{s_t,a}, & \text{w.p. } 1-\epsilon. \\ unif(A), & \text{w.p. } \epsilon. \end{cases} \tag{14}$$

- SARSA:
  - update based on the current play (s, a, r, s', a')

$$Q_{k+1} = Q_k + \gamma_k(r + Q_k(s', a') - Q_k(s, a)) \tag{15}$$

  - Similar to Q-learning but is On-policy

Deep Q-Networks

- Drawback of Q-learning and SARSA:

  • Q-table can be too big if environment is complicate i.e. $1^6 \times 1^3$ maze

- Alternative Algorithm: DQN:

  • Use a function approximator to estimate action-value function with Q-Network

- Steps:

  i store transition$(s_t, s_t, r_{t+1}, s_{t+1})$ in memory
  ii sample mini-batch of transitions, optimise MSE between Q-network and Q-learning targets:

$$\text{minimize } L_w = E_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s', a'; w^-) - Q(s, a; w))^2] \tag{16}$$

  iii

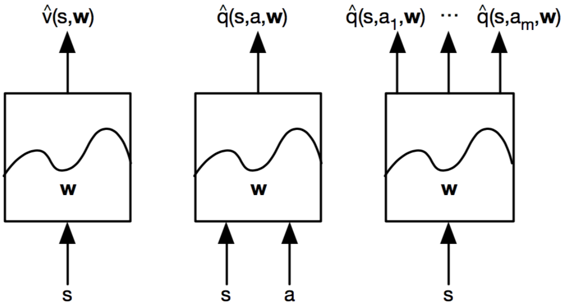
Value Iteration: Problem

- Recall: greedy action selection

$$\pi(s) = \arg\max_{a' \in A} Q(s_{t+1}, a') \tag{17}$$

- Problem: deterministic, strategy fixed, not practical in Partially-Observed environment

Value Function Approximation



- Tablular methods: impossible to record all states for real word problems
- Function approximation: generalize from seen states to unseen states

Value Function Approximation

- Goal: find parameter vector w minimising mean-squared error between approximate value function $\hat{v}(S, w)$ and true value function $v_\pi(S)$

$$J(w) = ||v_\pi(S) - \hat{v}(S, w)||_2^2 \tag{18}$$

- Stochastic gradient descent

$$\Delta w = \alpha(v_\pi(s) - \hat{v}(S, w))\nabla_w \hat{v}(S, w) \tag{19}$$

- In reality we don't have the true value function $v_\pi(S)$
  - For Monte-Carlo, use discounted return G_t
  - For TD, use $R_{t+1} + \lambda \hat{v}(S_{t+1}, w)$

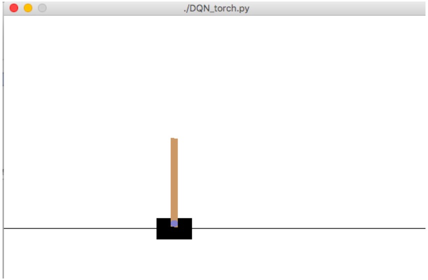
Deep Q-Networks (DQN)

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in memory D
- Sample random mini-batch of transitions (s, a, r, s') from D
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$L(w) = \mathrm{E}_{s,a,s,r' \sim D_i}[(r + \gamma \max_{a'} Q(s', a'; w^-) - Q(s, a; w))^2] \tag{20}$$

- Two important tricks in ensuring convergence: experience replay and fixed target

Deep Q-Networks(DQN): play games in OpenAI gym



- States are represented by 4-element tuples (position,cart velocity,angle,tip velocity)
- Actions can be either moving left or right
- Function approximator is a feed foward neural network
- 1 hidden layer with 10 neurons, 2 output neurons representing value estimation for two actions
- Implemented using torch and tensorflow, can stay alive for 1 minute

Policy based method

Value based method (previous sildes):

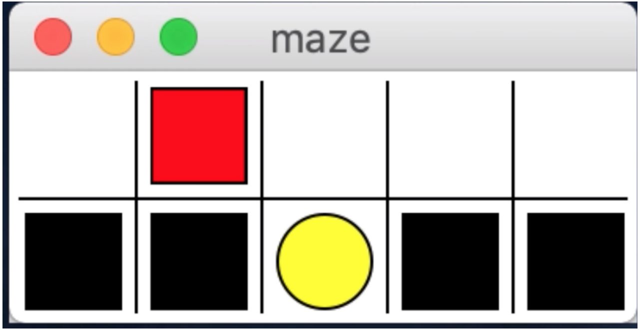
- Main focus is on state-action value evaluation
- Policy improvement is based on greedy or ϵ-greedy strategy w.r.t state-action values
- Return deterministic policy

Policy based method (slides after this page):

- Policy is a function of observations
- Policy improvement is based on gradient w.r.t some objective function
- State-action value not neccessary for policy updates
- Return stochastic policy

Policy based method

What's wrong with value based methods?



- Main problem: deterministic policy
- No good for partially observable environment
- Should I go left or right?

Policy Gradient: problem formulation

- Policy is a function of observation: $\pi_\theta(\cdot)$
- Trajectory τ: $\{s_0, a_0, r_0, s_1, a_1, r_1, ...\}$ is treated as random variable
- Distribution of τ is determined by policy π_θ
- For each trajectory, total reward is defined as $R(\tau)$
- Ultimate goal: optimize expectation $E_{\pi_\theta}[R(\tau)]$ w.r.t θ

Policy Gradient: approximate the gradient

- What does the gradient look like?

$$
\begin{aligned}
\nabla_\theta E_{\pi_\theta}[R(\tau)] &= \nabla_\theta \sum_\tau P_\theta(\tau) R(\tau) \\
&= \sum_\tau \nabla_\theta P_\theta(\tau) R(\tau) \\
&= \sum_\tau P_\theta(\tau) \frac{\nabla_\theta P_\theta(\tau)}{P_\theta(\tau)} R(\tau) = E_{\pi_\theta}[\nabla_\theta \ln P_\theta(\tau) R(\tau)]
\end{aligned}
\tag{21}
$$

- Equation 21 tell us: gradient can be represented as an expectation
- Why it is important: expectation can be approximated by sampling

Policy Gradient: approximate the gradient

- Why the gradient even exists?

$$P(\tau) = P(s_0)\prod_{i=0}^{\infty}\pi_\theta(a_i, s_i)P(s_{i+1}|s_i, a_i) \tag{22}$$

- Assumption: there is an underlying MDP specifying $P(s_{i+1}|s_i, a_i)$ and $P(s_0)$

$$\begin{aligned}
\nabla_\theta \ln P(\tau) &= \nabla_\theta \ln[P(s_0)\prod_{i=0}^{\infty}\pi_\theta(a_i, s_i)P(s_{i+1}|s_i, a_i)] \\
&= \nabla_\theta \ln P(s_0) + \nabla_\theta \sum_{i=0}^{\infty}[\ln \pi_\theta(a_i, s_i) + \ln P(s_{i+1}|s_i, a_i)] \\
&= \nabla_\theta \sum_{i=0}^{\infty} \ln \pi_\theta(a_i, s_i)
\end{aligned} \tag{23}$$

Policy Gradient: understanding the formula

Combine all equation in previous slides, one important formula:

$$\nabla_\theta E_{\pi_\theta}[R(\tau)] = E_{\pi_\theta}[R(\tau)\nabla_\theta \sum_{s_i,a_i\in\tau} \ln \pi_\theta(a_i, s_i)] \tag{24}$$

Intuition from equation 24, adjustment magnitude of policy on $\pi_\theta(a, s)$:

- In proportion to the total reward gained from trajectories containing (a, s)
  - Rationale: good actions lead to good trajectories, while bad actions lead to bad ones
  - What about good actions in trajectories with bad overall performance? work on it later
- In inverse proportion to the probability of performing action a on state s
  - consider actions sampled frequently but with small positive effect
  - mitigate case where 'not-so-good' actions are rewarded frequently

Vanilla Policy Gradient: **REINFORCE**

So far, we obtain the first policy gradient algotithm called **REINFORCE** [Williams, R. J.]

Algorithm 1 Generic Policy Gradient

```
Initialize policy parameters θ, learning rate α
for each iteration do
    Collect trajectories {τ_1, τ_2, τ_3, ...τ_k} using policy π_θ
    Estimate gradient grad^ = (1/k) Σ_{i=1}^{k} [R(τ_i) ∇_θ Σ ln π_θ(a, s)]
    Update policy θ ← θ + α · grad^
end for
Return policy π_θ
```

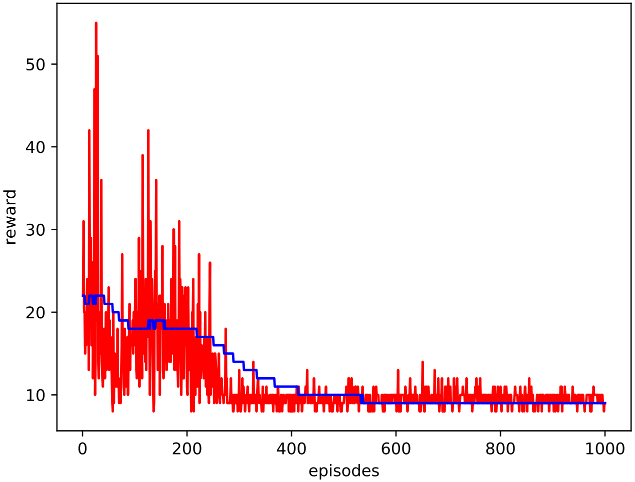
$$\widehat{grad} = \frac{1}{k}\sum_{i=1}^{k}\left[R(\tau_i)\nabla_\theta \sum \ln \pi_\theta(a,s)\right]$$

$$\theta \leftarrow \theta + \alpha \cdot \widehat{grad}$$

Vanilla Policy Gradient: **REINFORCE**

Experiment on CartPole game:

Bad performance, even not converge after 1000 episodes of training



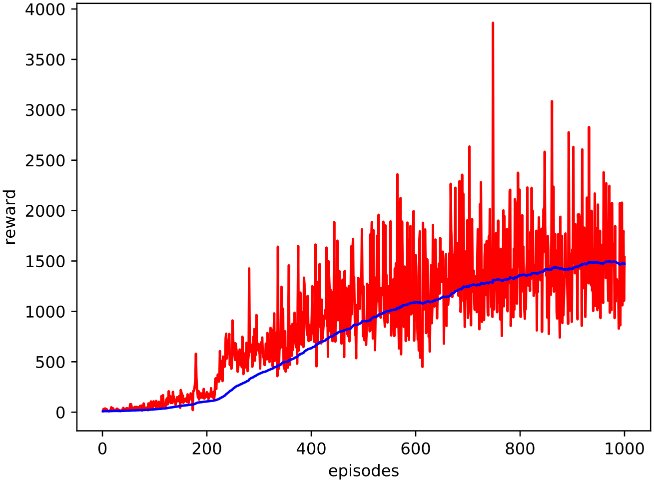
Improvement for **REINFORCE**: baseline

- One natural quetion: what if reward is always positive?
- Do we have to always increase $\pi_\theta(a, s)$ because $R(\tau)$ is positive (as in formula 24)?
- Actually, we only cares about the relative performance of trajectories
- Observation from formula 25: we can remove any constant term A from the expectation with out introducing bias.
- A can be the average performance for all trajectories, it is referenced as a baseline

$$
\begin{aligned}
E_{\pi_\theta}[\sum_a A \cdot \nabla \ln \pi(a, S)] &= \sum_a \pi_\theta(a, S) A \frac{\nabla_\theta \pi_\theta(a, S)}{\pi_\theta(a, S)} \\
&= A \sum_a \nabla_\theta \pi(a, S) \\
&= A \cdot \nabla_\theta \sum_a \pi_\theta(a, S) = A \nabla_\theta(1) = 0
\end{aligned} \tag{25}
$$

Improvement for **REINFORCE**: baseline

Experiment on CartPole game:
Better than before: an upgoing trend of rewards



Improvement for **REINFORCE**: advantage function

- Why we award/punish an action (s,a) based on the entire trajectory reward?
- Markov property: the action a_t only affects rewards after time t.

$$E_{\pi_\theta}[R_{0:i-1}(\tau)\nabla_\theta \ln \pi_\theta(a_i, s_i)] = 0 \tag{26}$$

- Actually, we can exploit the markov property to refine formula 24

$$\begin{aligned} \nabla_\theta E_{\pi_\theta}[R(\tau)] &= E_{\pi_\theta}[\sum(\nabla_\theta \ln \pi_\theta(a_i, s_i)(R_{0:i-1}(\tau) + R_{i:\infty}(\tau)))] \\ &= E_{\pi_\theta}[\sum(\nabla_\theta \ln \pi_\theta(a_i, s_i)R_{i:\infty}(\tau))] \end{aligned} \tag{27}$$

- Recall that subtraction of baseline doesn't change the expectation

$$\nabla_\theta E_{\pi_\theta}[R(\tau)] = E_{\pi_\theta}[\sum \nabla_\theta \ln \pi_\theta(a_i, s_i)(R_{i:\infty}(\tau) - V_{\pi_\theta}(s_i))] \tag{28}$$

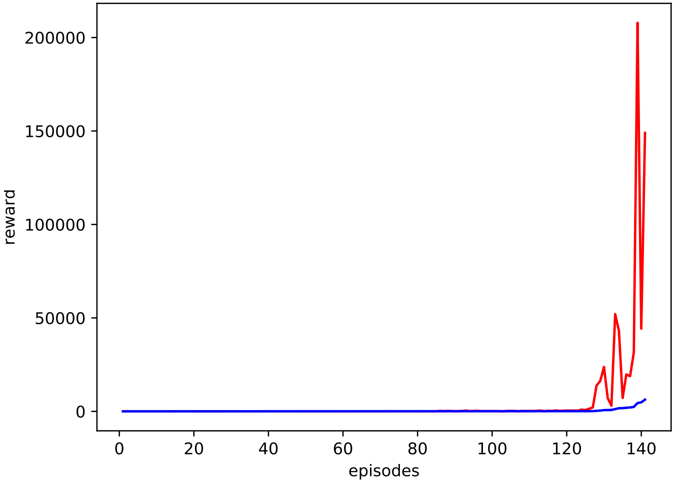
Improvement for **REINFORCE**: advantage function

- As shown in equation 28, $R_{i:\infty}(\tau) - V_{\pi_\theta}(s_i)$ is the actual term that determines the magnitude we adjust probability $\pi_\theta(a_i, s_i)$
- $R_{i:\infty}(\tau) - V_{\pi_\theta}(s_i)$ is also known as the advantage function
- Rationale: extra reward gained when performing certain action a_i on state s_i compared to average reward from that state under policy π_θ

Improvement for **REINFORCE**: advantage function

Experiment on CartPole game:

Only after 141 episodes of training, surviving time boosted to 20k!



Actor-Critic: a combination

So far, we mainly focused on pure value-based and pure policy-based methods ...

- Value-based: problem of deterministic policy in partially observed environments
- Policy-based: credit assignment problem (delay between action and reward)
- Why not combine them?
- Still use parameterized function for policy
- But also add an estimator for state values to approximate advantage function
- Stochastic policy agent without problem of update delay!

Actor-Critic: algorithm

Now there are two function to learn: policy function π_θ is known as actor and value estimator $\hat{V}_\varphi$ is known as critic, hence the model named **Actor-Critic**.

The overall algorithm can be like this (there are many variants, this is what we use in our experiment)

Algorithm 2 Actor-Critic

Initialize actor parameters θ, critic parameters φ, learning rate α
for each episode **do**
    Interact with environment for some time, get trajectory $\tau : \{s_0, a_0, r_0, s_1, a_0, r_0, ..., s_k\}$
    If s_k is Terminal, $R_{k:\infty} \leftarrow 0$ **else** $R_{k:\infty} \leftarrow \hat{V_\varphi}(s_k)$
    for each step i in $k-1:0$ **do**
        Estimate return $R_{i:\infty} \leftarrow r_i + R_{i+1:\infty}$
        Estimate advantage $A(a_i, s_i) \leftarrow R_{i:\infty} - \hat{V_\varphi}(s_i)$
    end for
    Estimate policy gradient $\hat{grad} = \frac{1}{k-1}\sum_{i=1}^{k-1}\sum[A(a_i, s_i)\nabla_\theta \ln \pi_\theta(a_i, s_i)]$
    Update actor $\theta \leftarrow \theta + \alpha \cdot \hat{grad}$
    Calculate loss for critic: $l(\varphi) \leftarrow \frac{1}{k-1}\sum_{i=0}^{k-1} A^2(a_i, s_i)$
    Update critic $\varphi \leftarrow \varphi + \alpha \cdot \nabla_\varphi l(\varphi)$
end for
Return policy π_θ

Further improvement: clipped objective function

- In previous section, policy gradient method works by computing gradient estimator in form

$$\hat{g} = \hat{E}_t[\nabla_\theta \ln \pi_\theta(a_t, s_t)\hat{A}_t] \tag{29}$$

- The estimator $\hat{g}$ can be obtained by differentiating the objective

$$L^{PG}(\theta) = \hat{E}_t[\ln \pi_\theta(a_t, s_t)\hat{A}_t] \tag{30}$$

- Multiple steps to optimize L^{PG} on same trajectory: destructively large policy updates [Schulman, John et al.].
- To improve sample efficiency, they adopt strategy of clipping the surrogate objective function in form $L^{CPI}(\theta)$:

Further improvement: clipped objective function

$$L^{CPI}(\theta) = \hat{E}_t[\frac{\pi_\theta(a_t, s_t)}{\pi_{\theta_{old}}(a_t, s_t)}\hat{A}_t] \tag{31}$$

- $\pi_{\theta_{old}}$: fixed term generated by old policy
- π_θ: current policy being optimized.
- The ratio $\frac{\pi_\theta(a_t,s_t)}{\pi_{\theta_{old}}(a_t,s_t)}$ is denoted as $r_t(\theta)$
- $r_t(\theta)$ measures the difference between current policy and old policy
- we don't want too big a update step, hence some constraint based on $r_t(\theta)$
- In practise we use the gradient of following objective function

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{32}$$

Further improvement: clipped objective function

The algorithm is known as Proximal Policy Optimization [Schulman, John et al.]

Only with minor modification on previous Actor-Critic algorithm:

Algorithm 3 Proximal Policy Optimization

Initialize actor parameters θ, critic parameters φ, learning rate α, clip coefficient ϵ

Initialize old policy $\pi_{\theta_{old}} \leftarrow \pi_\theta$

for each episode **do**

    for each time period in episode **do**

        denote current state as s_0, continue interacting with environment for some time

        get trajectory $\tau : \{s_0, a_0, r_0, s_1, a_0, r_0, ..., s_k\}$

        If s_k is Terminal, $R_{k:\infty} \leftarrow 0$ **else** $R_{k:\infty} \leftarrow \hat{V_\varphi(s_k)}$

        for each step i in $k-1:0$ **do**

            Estimate return $R_{i:\infty} \leftarrow r_i + R_{i+1:\infty}$

            Estimate advantage $A(a_i, s_i) \leftarrow R_{i:\infty} - \hat{V_\varphi(s_i)}$

        end for

        calculate loss for actor $l(\theta) \leftarrow \frac{1}{k}\sum_{t=0}^{k-1} \min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$

        apply multiple steps gradient ascent on $l(\theta)$ to update θ

        calculate loss for critic: $l(\varphi) \leftarrow \frac{1}{k-1}\sum_{i=0}^{k-1} A^2(a_i, s_i)$

        apply multiple steps gradient descent on $l(\varphi)$ to update φ

        renew old policy $\pi_{\theta_{old}} \leftarrow \pi_\theta$

    end for

end for

Return policy π_θ

Further improvement: high score buffer replay