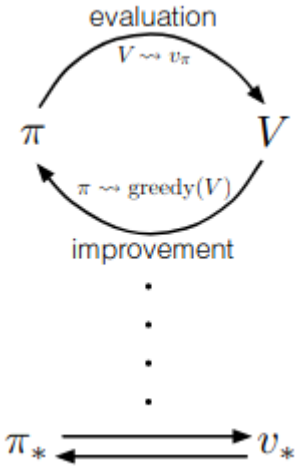
Expected return G_t is defined as some specific function of the reward sequence, in the simplest case the return is the sum of the rewards.

If natural notion of final time step → subsequences = episodes. Episodes all end in the same terminal state. (S&B: 54)

It has an upper and lower bound if the rewards are bounded:

$$G_{max/min} = r_{max/min} \frac{1}{1-\gamma}$$

Generalized Policy Iteration methods (dynamic programming, builds on policy improvement theorem)



Value Iteration:

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

Bellman Equation (used in Q-Learning)

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

Definitions of q and v

$$q_\pi(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_\pi \sum_{i=1} \gamma^i r_{t+i}$$

$$v(s_t) = \mathbb{E}_\pi \sum_{i=0} \gamma^i r_{t+i}$$

Recursive Definitions

$$q(s,a) = r(s,a) + \gamma(\sum_{a'} \pi(a'|s')q(s',a'))$$

$$v(s) = \sum_a \pi(a|s)(r(s,a) + \gamma v(s'))$$

Estimators:

$$Q(s,a) = r(s,a) + \frac{1}{k}\sum_k [\gamma G_\pi^k(s')]$$

$$V(s) = \frac{1}{k}\sum_k G_\pi^k(s)$$

Backwards Calculation of Return G

$$G_0 = r_0 + \gamma r_1 + \gamma^2 r_2 = r_0 + \gamma[r_1 + \gamma r_2] = r_0 + \gamma G_1$$

Monte Carlo Methods

Difference to Generalized Policy Iteration methods (dynamic programming):

- We don't know the state transition probabilities and the reward function of the environment.
- We only loop over the sampled episode once instead of doing a complete loop for every step.
- In Monte Carlo methods, estimates for each state are independent; in DP, the estimate for one state builds upon the estimates of other states → makes Monte Carlo methods computationally attractive
- Advantages of Monte Carlo methods: ability to learn from actual experience, from simulated experience, and ability to generate many sample episodes starting from the state of interest

Alternative for the Evaluation Step

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$(Returns(S_t))$

Importance Sampling

If the episode that we use to do the evaluation comes from a different policy than the one we are optimizing, we need to do importance sampling.

IS is a general method from statistics to approximate expected values of one distribution by sampling from another, different distribution.

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}\,|\,S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}\,|\,S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}.$$

To calculate the value estimate, we weigh the return G by the importance sampling ratio.

The ratio (rho) is the probability of following the episode's trajectory under the target policy and the probability of following the trajectory under the behavior policy that it was sampled from.

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}.$$

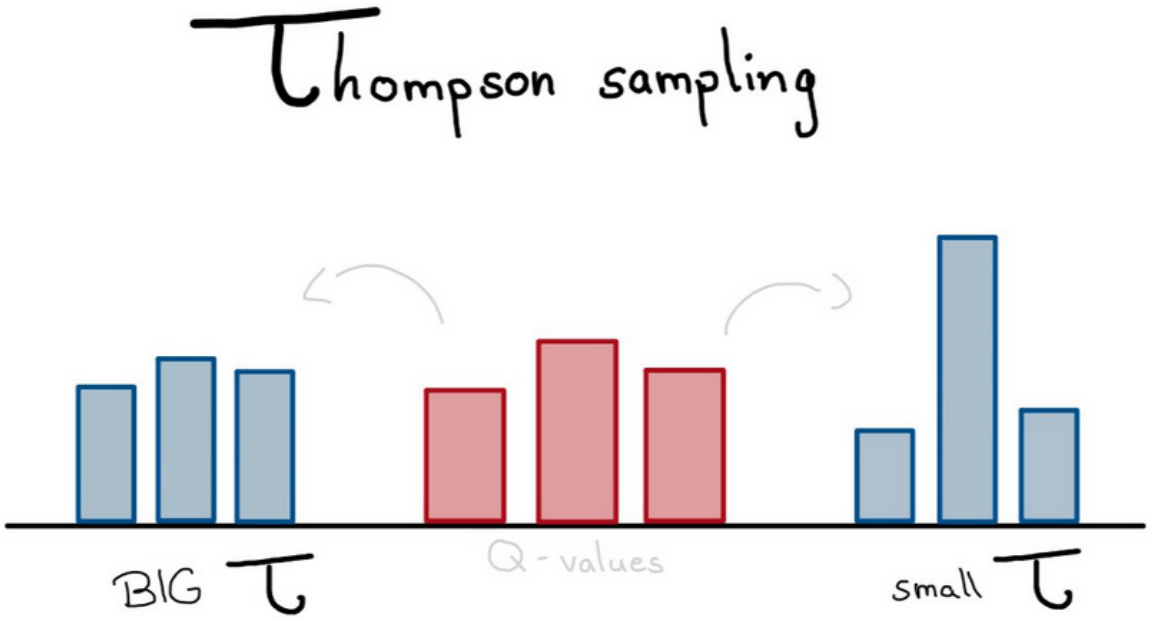
Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of π
    Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

Thompson Sampling
(softmax over Q values with temperature)

$$\pi(a|s) = \frac{e^{\tau^{-1} Q(s,a)}}{\sum_{a' \in A} e^{\tau^{-1} Q(s,a')}}$$



Epsilon Greedy

To get our policy we take the argmax action with probability (1-ε) and all other actions with equal probability (epsilon / number of actions)

Variance and bias

Monte Carlo sampling has large variance and using bootstrapping (using an estimate of the state Value to estimate the state value) adds a bias.

Temporal Difference Error

TD error — the difference between the estimated value of S_t and the better estimate R_{t+1} + V (S_{t+1}). Because the TD error depends on the next state and next reward, it is not available until one time step later, i.e. δ_t is the error in V(S_t), available at time t + 1. (p. 121)

$$\delta_{TD(1)} = r(s,a) + \gamma V(s') - V(s)$$

r(s,a) + gamma * (V(s')) is the new estimate for V(s). We could also use a Monte Carlo Estimate, or a n-step SARSA estimate or a TD(lambda) estimate.

SARSA:

- on-policy because we take the Q value for the actually performed action.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Expected SARSA:

- on-policy because we take the expected Q value over actions (whereas we only took one action in the state)
- if pi is a greedy policy while the behavior is exploratory, Expected SARSA = Q-Learning

n-step SARSA/ TD(n):

- combination of monte carlo value estimation and bootstrapping (using a previous tabular or deep-Q value estimate)

n-step return:

$$\left(\sum_{t}^{n-1} \gamma^t r_t\right) + \gamma^n Q(s_n, a_n)$$

TD(lambda):

- **Continuous version of TD(n), where lambda determines where we are on the continuum from MC to one-step TD methods**

Exponentially weighted average over all different n-step returns.

$$G^{\lambda}_{t:t+k} = \hat{v}(S_t,\mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t}\delta'_i,$$

where

$$\delta'_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_{t-1}).$$

Truncated lambda-return:

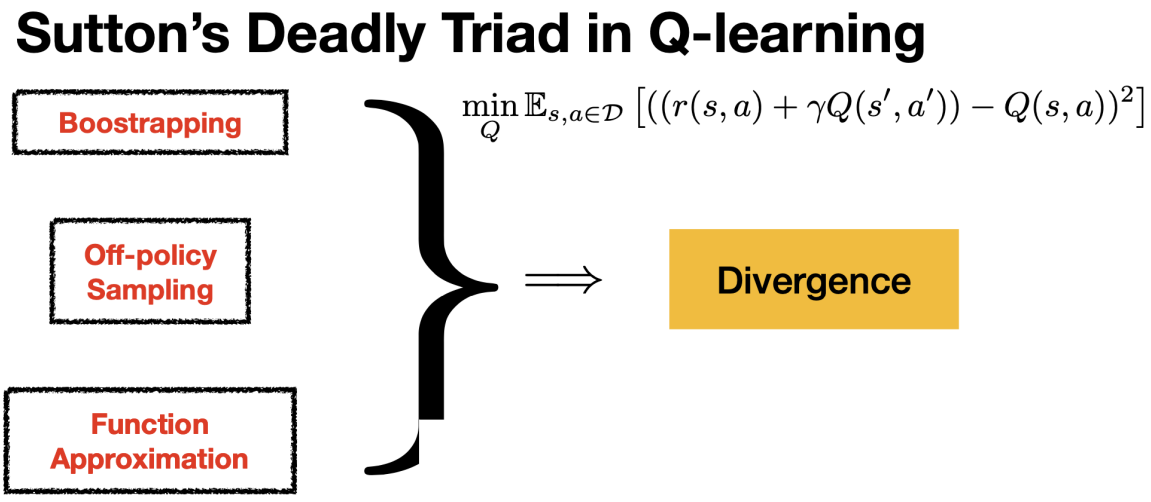
$$G^{\lambda}_{t:h} \doteq (1-\lambda)\sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}, \qquad 0 \le t < h \le T.$$

$$G^{h}_{TD(\lambda)} = [(1-\lambda)\textstyle\sum_{n}^{h-1} G_{TD(n)}\lambda^{n-1}] + [\lambda^{h} G_{MC}]$$

SARSA(lambda)-return for action-state values

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^{n}\hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \qquad t+n < T,$$

$$\delta_t \doteq R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t),$$



Off-policy training: training on a distribution of transitions other than that produced by the target policy (S&B: 264)

Why deep learning for RL problems?

Too many states in natural environments for Q-learning to be plausible (since Q-learning needs to track a value for each single state separately, similar states are treated as distinct states)

→ deep learning model that can approximate *Q(s,a)*

DQN PSEUDOCODE

initialize: $Q_{net}(s, a)$ and an empty experience replay buffer ERP of maximum size N
for each episode:

s = initial state
for each timestep t:
sample action a_t
obtain reward $r_t$and next state s_{t+1}
store $\{s_t, a_t, r_t, s_{t+1}\}$ in ERP
for m random samples $\{s_t, a_t, r_t, s_{t+1}\} \in ERP$ (t is here different from the t above!)
compute $q_{target} = y_t = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$
train $Q_{net}(s, a)$ via backpropagation minimizing $(y_t - Q_{net}(s_t, a_t))^2$
$s_t = s_{t+1}$

Delayed DQN

Not use the updated Q value for creating new targets until after a delay

Other DQN improvements:

- (clipped) Double Q-Learning
- Prioritized Replay
- Dueling Networks (learn Advantage and value and reconstruct Q values)
- Multi-step Learning (like SARSA(n) but as Q(n))
- Distributional RL
- Noisy Nets (introduce noise to the activations or weights of the network)

Policy Gradient

PG-Theorem: To get the policy gradients we don't need the transition probabilities.

REINFORCE

1. samples trajectories by running current policy
2. compute estimate of policy gradient (scaled by rewards)
3. apply scaled policy gradients to policy via gradient ascent

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Natural policy gradient (something addressed by TRPO):

- PG can be difficult to use because different parameters affect the policy distribution to very different degrees. Natural policy gradients are constrained which helps in the optimization

DDPG (Deep Deterministic Policy Gradient)

- DDPG is like Q-learning but with a policy network to estimate the max step for the Q-value.
- DDPG uses Delayed Q-Learning updates via Polyak averaging (slowly tracking the learned networks instead of a hard delay).
- It computes targets with the Bellman equation just like Q-learning, to update the Q-function.
- To update the policy pi (here it is called mu), we take the gradient ascent in the direction that maximizes the Q value.

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

TD3 Twin Delayed DDPG

Like DDPG but with three improvements:

- *Clipped Double-Q Learning*. TD3 learns *two* Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.
- *"Delayed" Policy Updates*. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.
- *Target Policy Smoothing*. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Algorithm 1 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if j mod `policy_delay` $= 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

Advantage Actor Critic

What is an issue with policy gradient methods? High variance due to the monte-carlo sampling with a finite number of samples.

To reduce the variance, we replace the monte carlo estimate from REINFORCE by the Q-Value estimate. In addition to that, subtracting a baseline of this Q-Value then leads to lower overall values and thus also to lower variance. We can subtract a constant and the policy gradient theorem still holds without introducing any additional bias error. *"The expected value of the gradient for the policy does not change even with a state dependent baseline"*

Generalized Advantage Estimation

Like in TD(lambda), GAE computes all advantage estimates for each n, as TD Lambda does with TD(n). It then takes the exponentially weighted average of these.

$$GAE(\gamma, \lambda) = \sum_{l=0}^{\infty} [(\gamma\lambda)^l \delta_{TD(1)_l}]$$

$$\hat{A}^{\pi}_{\text{GAE}}(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'} \qquad \delta_{t'} = r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) + \gamma \hat{V}^{\pi}_{\phi}(\mathbf{s}_{t'+1}) - \hat{V}^{\pi}_{\phi}(\mathbf{s}_{t'})$$

(Source: slides from Lecture 5 CS 294-112, http://rail.eecs.berkeley.edu/deeprlcourse/)

Q-Function (state-action-values)

Policy Pi (gives max Q actions)

Value Function (baseline)

Alternative: instead of Q and V, we only have A

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

TRPO

- policy that was used to sample the data and the current policy that we're optimizing need to be close/similar.
- To ensure this, we use a KL measure to constrain the optimization

$$\nabla v_\pi(s_0) \propto \sum_{s \in S} [\mu(s) \sum_a [\pi(a|s) q(s,a) \frac{\nabla \pi(a|s)}{\pi(a|s)}]]$$

mu(s) would need to change given that we have changed our policy and thus the policy gradient theorem would not apply anymore if we do not make sure that the new policy is in a "trust region" close to the old one.

PPO

- simplification of TRPO
- penalty term version:
  - instead of having hard constraint like in TRPO, we penalize for KL Divergence between old and new policy
  - need to choose hyperparameter Beta (scales KL penalty)

- clipped version (preferred):
  - no KL Divergence and no constraint at all

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \quad \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right)$$

If the advantage is positive:

If the advantage is negative:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s,a)) \right),$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

SAC

- In SAC, we represent the log std devs as outputs from the neural network, meaning that they depend on state in a complex way.

- Use tanh “squashing” to bound the continuous action space.

- clipped double-Q
- both Q-functions are learned with MSBE minimization
- polyak averaging the Q-network parameters over the course of training
- regularizes for entropy (entropy should be high)
- trains a stochastic policy (unlike TD3 and DDPG)
- could handle discrete action space with small changes