Value Iteration Convergence

Theorem. Value iteration converges. At convergence, we have found the optimal value function V* for the discounted infinite horizon problem, which satisfies the Bellman equations

$$\forall S \in S: \quad V^*(s) = \max_A \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

- Now we know how to act for infinite horizon with discounted rewards!
  - Run value iteration till convergence.
  - This produces V*, which in turn tells us how to act, namely following:

$$\pi^*(s) = \arg\max_{a \in A} \textstyle\sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

- Note: the infinite horizon optimal policy is stationary, i.e., the optimal action at a state s is the same action at all times. (Efficient to store!)

But it’s not really all over…

- What if:
  - there are lots of states?
  - we don’t know T?
  - we don’t know R?

Policy iteration

- Idea:
  - evaluate some policy
  - then make it better

Policy Evaluation

- Recall value iteration iterates:

$$V_{i+1}^*(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i^*(s')]$$

- Policy evaluation:

$$V_{i+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_i^\pi(s')]$$

  - At convergence:

$$\forall s \quad V^\pi(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^\pi(s')]$$

Exercise 2

Consider a stochastic policy $\mu(a|s)$, where $\mu(a|s)$ is the probability of taking action a when in state s. Which of the following is the correct value iteration update to perform policy evaluation for this stochastic policy?

1. $V^{\mu}_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^{\mu}_i(s'))$
2. $V^{\mu}_{i+1}(s) \leftarrow \sum_{s'} \sum_a \mu(a|s) T(s,a,s')(R(s,a,s') + \gamma V^{\mu}_i(s'))$
3. $V^{\mu}_{i+1}(s) \leftarrow \sum_a \mu(a|s) \max_{s'} T(s,a,s')(R(s,a,s') + \gamma V^{\mu}_i(s'))$

Policy Iteration

- Alternative approach:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is policy iteration
  - It's still optimal!
  - Can converge faster under some conditions

Policy Evaluation Revisited

- *Idea 1:* modify Bellman updates

$$V_0^\pi(s) = 0$$

$$V_{i+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_i^\pi(s')]$$

- Idea 2: it's just a linear system, solve with Matlab (or whatever),
  variables: $V^\pi(s)$,
  constants: T, R

$$\forall s \quad V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

Policy Iteration Guarantees

Policy Iteration iterates over:

- Policy evaluation: with fixed current policy π, find values with simplified Bellman updates:
  - Iterate until values converge

$$V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} T(s, \pi_k(s), s') \left[R(s, \pi_k(s), s') + \gamma V_i^{\pi_k}(s')\right]$$

- Policy improvement: with fixed utilities, find the best action according to one-step look-ahead

$$\pi_{k+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_k}(s')\right]$$

Theorem. Policy iteration is guaranteed to converge and at convergence, the current policy and its value function are the optimal policy and the optimal value function!

Proof sketch:

(1) Guarantee to converge: In every step the policy improves. This means that a given policy can be encountered at most once. This means that after we have iterated as many times as there are different policies, i.e., (number actions)$^{\text{(number states)}}$, we must be done and hence have converged.

(2) Optimal at convergence: by definition of convergence, at convergence $\pi_{k+1}(s) = \pi_k(s)$ for all states s. This means $\forall s \;\; V^{\pi_k}(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_i^{\pi_k}(s')\right]$
Hence V^{π_k} satisfies the Bellman equation, which means V^{π_k} is equal to the optimal value function V*.

Points

- Value iteration won’t work if we don’t know the prob of new state from action
- also policy iteration
- So state-value

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$

How good is a state?
The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

How good is a state-action pair?
The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman equation

The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following **Bellman equation**:

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

Intuition: if the optimal state-action values for the next time-step Q*(s',a') are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s',a')$

The optimal policy **π*** corresponds to taking the best action in any state as specified by Q*

Solving for the optimal policy

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

Q_i will converge to Q* as i -> infinity

What's the problem with this?
Not scalable. Must compute Q(s,a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

Solution: use a function approximator to estimate Q(s,a). E.g. a neural network!

Function approximation how?

Solving for the optimal policy: Q-learning

Remember: want to find a Q-function that satisfies the Bellman Equation:

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

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$

Iteratively try to make the Q-value close to the target value (y_i) it should have, if Q-function corresponds to optimal Q* (and optimal policy **π***)

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute to multiple weight updates => greater data efficiency

Two cases

- We know all probabilities, rewards
  - we’ve dealt with this; change value iteration equations as required
  - not that exciting cause it doesn’t happen very often
    - if we do know all this stuff, the set of states and actions is small
    - so we don’t really need a network model of Q
- We *don’t* know all probabilities, or rewards
  - this means we have to estimate them
    - likely as a result of acting
    - quite possibly in simulation
  - and we have to be very careful about errors in estimation

Estimating rewards

- The future looks like:

$$\begin{array}{lr} s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots & p \\ s_0, a_0, r_0, s'_1, a'_1, r'_1, s'_2, a'_2, r'_2, \ldots & p' \\ s_0, a_0, r_0, s^*_1, a^*_1, r^*_1, s^*_2, a^*_2, r^*_2, \ldots & p^* \\ \ldots & \ldots \end{array}$$

  - and there are lots of them - we can’t see every trajectory

Sampling and the WLLN - I

- Generally, we can estimate expectations (WLLN)

$$x_i \sim p(x)$$ (Recall ~ means IID samples)

Then

$$\frac{1}{N}\sum_i f(x_i) \to \mathbb{E}_{p(x)}[f(x)] = \int f(x)p(x)dx$$

$$\frac{1}{N}\sum_i f(x_i) = \mathbb{E}_{p(x)}[f(x)] + \xi$$

Sampling and WLLN - II

- But this means we could estimate the E
  - draw samples of trajectories
    - same as run simulator with policy pi some number of times
    - average rewards over simulations
  - Issues:
    - you need a simulator (or patience)
    - there could be serious errors in the estimate

$$\begin{array}{lr}
s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots & p \\
s_0, a_0, r_0, s'_1, a'_1, r'_1, s'_2, a'_2, r'_2, \ldots & p' \\
s_0, a_0, r_0, s^*_1, a^*_1, r^*_1, s^*_2, a^*_2, r^*_2, \ldots & p^* \\
\ldots & \ldots
\end{array}$$

Variance in estimates - WLLN, II

$$x_i \sim p(x)$$

$$\frac{1}{N}\sum_i f(x_i) \to \mathbb{E}_{p(x)}[f(x)] = \int f(x)p(x)dx$$

$$\frac{1}{N}\sum_i f(x_i) = \mathbb{E}_{p(x)}[f(x)] + \xi$$

$\mathbb{E}[\xi] = 0$ $\mathbb{E}[\xi^2]$

Importance sampling and WLLN - III

$$x_i \sim q(x)$$

$$\frac{1}{N}\sum_i \frac{f(x_i)p(x_i)}{q(x_i)} \to \int \frac{f(x)p(x)}{q(x)}q(x)dx = \int f(x)p(x)dx$$

Importance sampling and WLLN - IV

$$x_i \sim q(x)$$

$$\frac{1}{N}\sum_i \frac{f(x_i)p(x_i)}{q(x_i)} = \mathbb{E}_{p(x)}\left[f(x)\right] + \zeta$$

$$\mathbb{E}\left[\zeta\right] = 0 \qquad \mathbb{E}\left[\zeta^2\right]$$

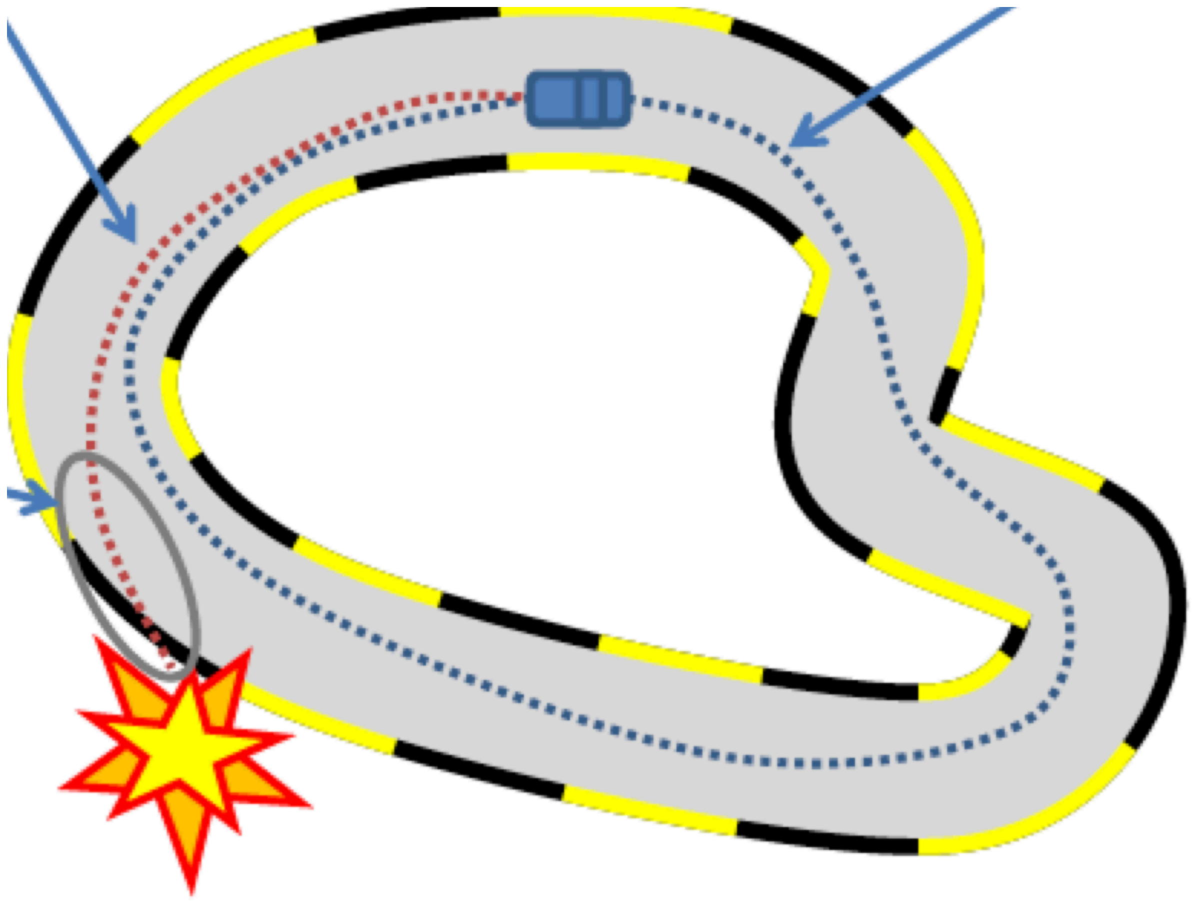
Errors in estimating rewards

- The future looks like:

$$\begin{array}{lr} s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots & p \\ s_0, a_0, r_0, s'_1, a'_1, r'_1, s'_2, a'_2, r'_2, \ldots & p' \\ s_0, a_0, r_0, s^*_1, a^*_1, r^*_1, s^*_2, a^*_2, r^*_2, \ldots & p^* \\ \ldots & \ldots \end{array}$$

  - and there are lots of them - we can't see every trajectory
- Notice that r_k could depend very strongly on a_1 (say)
- This creates a problem
  - samples far in the future depend strongly on early choices
  - results in variance in the sampled estimates

Errors in estimating rewards are important



Training the Q-network: Experience Replay

Learning from batches of consecutive samples is problematic:

- Samples are correlated => inefficient learning
- Current Q-network parameters determines next training samples (e.g. if maximizing action is to move left, training samples will be dominated by samples from left-hand size) => can lead to bad feedback loops

Address these problems using **experience replay**

- Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
- Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples

Each transition can also contribute to multiple weight updates => greater data efficiency

Learning a policy

Policy Gradients

What is a problem with Q-learning?
The Q-function can be very complicated!

Example: a robot grasping an object has a very high-dimensional state => hard to learn exact value of every (state, action) pair

But the policy can be much simpler: just close your hand
Can we learn a policy directly, e.g. finding the best policy from a collection of policies?

Policy Gradients

Formally, let's define a class of parametrized policies: $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

We want to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$

How can we do this?

Gradient ascent on policy parameters!

Policy gradients - core idea

$$\left.\begin{cases} s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \dots & p \\ s_0, a_0, r_0, s'_1, a'_1, r'_1, s'_2, a'_2, r'_2, \dots & p' \\ s_0, a_0, r_0, s^*_1, a^*_1, r^*_1, s^*_2, a^*_2, r^*_2, \dots & p^* \\ \dots & \dots \end{cases}\right\} \text{ under } \pi(\theta)$$

$$\text{Compute } J(\theta) = \mathbb{E}_{p(\tau)}\left[\sum_t \gamma^t r_t\right] \approx \frac{1}{N}\sum_\tau \left[\sum_t \gamma^t r_t\right]$$

$$\left.\begin{cases} s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \dots & p \\ s_0, a_0, r_0, s'_1, a'_1, r'_1, s'_2, a'_2, r'_2, \dots & p' \\ s_0, a_0, r_0, s^*_1, a^*_1, r^*_1, s^*_2, a^*_2, r^*_2, \dots & p^* \\ \dots & \dots \end{cases}\right\} \text{ under } \pi(\theta')$$

$$\text{Compute } J(\theta') = \mathbb{E}_{p(\tau)}\left[\sum_t \gamma^t r_t\right] \approx \frac{1}{N}\sum_\tau \left[\sum_t \gamma^t r_t\right]$$

REINFORCE algorithm

Mathematically, we can write:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau \sim p(\tau;\theta)} \left[r(\tau) \right] \\ &= \int_{\tau} r(\tau) p(\tau;\theta) \mathrm{d}\tau \end{aligned}$$

Where r(τ) is the reward of a trajectory $\tau = (s_0, a_0, r_0, s_1, \ldots)$

REINFORCE algorithm

Expected reward: $J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)} [r(\tau)]$

$$= \int_\tau r(\tau) p(\tau;\theta) \mathrm{d}\tau$$

Now let's differentiate this: $\nabla_\theta J(\theta) = \int_\tau r(\tau) \nabla_\theta p(\tau;\theta) \mathrm{d}\tau$

Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick: $\nabla_\theta p(\tau;\theta) = p(\tau;\theta) \frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta) \nabla_\theta \log p(\tau;\theta)$

If we inject this back:

$$\nabla_\theta J(\theta) = \int_\tau \left(r(\tau) \nabla_\theta \log p(\tau;\theta) \right) p(\tau;\theta) \mathrm{d}\tau$$

$$= \mathbb{E}_{\tau \sim p(\tau;\theta)} [r(\tau) \nabla_\theta \log p(\tau;\theta)]$$

Can estimate with Monte Carlo sampling

REINFORCE algorithm

$$\nabla_\theta J(\theta) = \int_\tau \left(r(\tau)\nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta)\mathrm{d}\tau$$
$$= \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]$$

Can we compute those quantities without knowing the transition probabilities?

We have: $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$

Thus: $\log p(\tau;\theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$

And when differentiating: $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

Doesn't depend on transition probabilities!

Therefore when sampling a trajectory τ, we can estimate J(θ) with

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Sampled estimates

Get samples of trajectories (simulator)

$$\left\{ \begin{array}{lr} s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots & p \\ s_0, a_0, r_0, s_1', a_1', r_1', s_2', a_2', r_2', \ldots & p' \\ s_0, a_0, r_0, s_1^*, a_1^*, r_1^*, s_2^*, a_2^*, r_2^*, \ldots & p^* \\ \ldots & \ldots \end{array} \right\} \text{ under } \pi(\theta)$$

Gradient is

$$\frac{1}{N} \sum_{\tau} \left[\sum_{t} r_t \nabla_\theta \log \pi_\theta(a_t | s_t) \right]$$

Intuition

Gradient estimator: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$

Interpretation:

- If r(τ) is high, push up the probabilities of the actions seen
- If r(τ) is low, push down the probabilities of the actions seen

Might seem simplistic to say that if a trajectory is good then all its actions were good. But in expectation, it averages out!

However, this also suffers from high variance because credit assignment is really hard. Can we help the estimator?

Variance reduction

Gradient estimator: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$

First idea: Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Second idea: Use discount factor γ to ignore delayed effects

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Variance reduction: Baseline

Problem: The raw value of a trajectory isn't necessarily meaningful. For example, if rewards are all positive, you keep pushing up probabilities of actions.

What is important then? Whether a reward is better or worse than what you expect to get

Idea: Introduce a baseline function dependent on the state.
Concretely, estimator is now:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

How to choose the baseline?

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

Variance reduction techniques seen so far are typically used in “Vanilla REINFORCE”

How to choose the baseline?

A better baseline: Want to push up the probability of an action from a state, if this action was better than the **expected value of what we should get from that state**.

Q: What does this remind you of?

A: Q-function and value function!

Intuitively, we are happy with an action a_t in a state s_t if $Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$ is large. On the contrary, we are unhappy with an action if it's small.

Using this, we get the estimator: $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$

Actor-Critic Algorithm

Problem: we don’t know Q and V. Can we learn them?

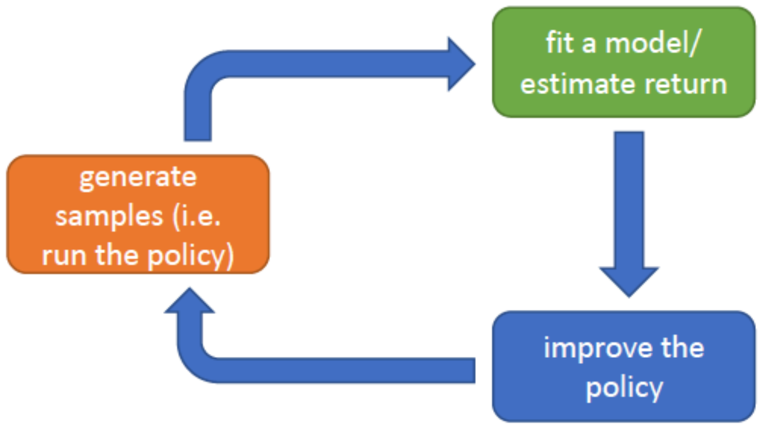
Yes, using Q-learning! We can combine Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic** (the Q-function).

- The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust
- Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
- Can also incorporate Q-learning tricks e.g. experience replay
- **Remark:** we can define by the **advantage function** how much an action was better than expected

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Why so many RL algorithms?

- Different tradeoffs
  - Sample efficiency
  - Stability & ease of use
- Different assumptions
  - Stochastic or deterministic?
  - Continuous or discrete?
  - Episodic or infinite horizon?
- Different things are easy or hard in different settings
  - Easier to represent the policy?
  - Easier to represent the model?


fit a model/ estimate return
generate samples (i.e. run the policy)
improve the policy

Levine, ND

Blog post entitled: “Why deep reinforcement learning doesn’t work”

https://www.alexirpan.com/2018/02/14/rl-hard.html