__/__/____

* Policy Gradient Algorithms

→ Instead of maintaining estimates of Value Functions, Search is the space of policies.

+ Simpler Description, better convergence,
+ When you have continuous actions, hard to do Q networks, Value functions are difficult to make it work so usually they are discretized. No need to do that for policy gradient methods.
+ Robust to partial observability: when you don't have access to state s but have access to f(s). Policy Gradient methods are more robust because they don't use the notion of value fundions and hence aren't tied to states.

→ Policy depends on some parameters θ. Idea is to modify policy parameters directly instead of estimating the action values.

↳ Max $J(\theta) = E[r_t] = \sum_a q_*(a)\pi_\theta(a)$

↳ $\theta = \theta + \alpha \nabla_\theta J(\theta)$ (Gradient Ascent)

For a particular policy π, $J(\theta)$ gives some kind of performance metric.

So, $J(\theta) = E[r_t] = \sum_a q_*(a)\pi_\theta(a)$ ⇒ we can think of the sum as the expected value of q_* over π_θ i.e

$J(\theta) = E[r_t]$ $\sum q_*(a)\pi(a) = E_{\pi_\theta}[q_*(a)]$

Since q_* is unknown, fix π_θ and repeatedly sample

You have $(t_1, r_{t_2}, a_{t_1}), (t_2, r_{t_3}, a_{t_2}) \ldots$ samples.

Thus $E[r_t] \approx \frac{1}{N}\sum r_i = J(\theta) = E_{\pi_\theta}[q_*(a)]$

Now, $\nabla J(\theta) = \sum q_*(a)\nabla\pi_\theta(a) = \sum q_*(a)\frac{\nabla\pi_\theta(a)}{\pi_\theta(a)}\pi_\theta = E_{\pi_\theta}\left[q_*\frac{\nabla\pi_\theta(a)}{\pi_\theta(a)}\right]$

$\approx \sum r_i \frac{\nabla\pi_\theta(a_i)}{\pi_\theta(a_i)} =$

__/__/____

* **REINFORCE**

Learning Rules should not depend on the reward

$$\Delta\theta_t = \alpha r_t \frac{\nabla \pi_\theta(a_t)}{\pi_\theta(a_t)} = \alpha r_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta}$$

(with baseline): $\Delta\theta_t = \alpha (r_t - b_t) \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta}$

↳ baseline should not depend on action

* **REINFORCE: MC POLICY GRADIENT**

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta)$$

↑ Fraction of time Agent will spend in state s while executing policy π

REINFORCE is essentially an instance of the Policy gradient theorem applied to the full MDP

$$= E_\pi \left[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right]$$

↳ mc because full return is used

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta_t)} \nabla \ln(\pi(A_t|S_t,\theta))$$

Algo:

↳ Input: Differentiable policy $\pi(a|s,\theta)$, $\alpha > 0$, $\theta \in \mathbb{R}^d$ (0 eg)

↳ For each episode, generate Trajectory,

↳ For each step of episode calculate disc Return

$$G = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

$$\theta = \theta + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\theta)$$

(no bootstrapping)

+ Unbiased estimate of Return because you're not using any q-values.

- Slow due to high Variance in estimates

↳ Related to "recurrence time" of episode

- For large state spaces, Variance Unacceptably high.

__/__/____

* REINFORCE: MC with Baseline

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \left(q_\pi(s,a) - b(s)\right) \nabla \pi(a|s,\theta)$$

↳ cannot be a function of the action because

$$\sum_a b(s) \nabla \pi(a|s,\theta) = b(s) \sum_a \nabla \pi(a|s,\theta) = b(s) \nabla \sum_a \pi(a|s,\theta) = b(s) \nabla 1 = 0$$

That means gradients are not being affected.

So $\theta_{t+1} = \theta_t + \alpha \left(G_t - b(S_t)\right) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$

+ Doesn't change expected value, but improves variance.

* Actor-Critic Methods:

→ Learn both a policy & state-value function

Actor — Param: θ ; Critic (evaluates state-action) — Param: ω

$$b(s) = E_{\pi_\theta}[G_t | S_t = s] = V_{\pi_\theta}(s)$$

$$\text{So, } \theta_{t+1} = \theta_t + \alpha \left(\underbrace{G_{t:t+1}}_{1\text{-step}} - \hat{V}(S_t,\omega)\right) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

$$= \theta_t + \alpha \underbrace{\left(R_{t+1} + \gamma \hat{V}(S_{t+1},\omega) - \hat{V}(S_t,\omega)\right)}_{\delta_t} \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

$$= \theta_t + \alpha \delta_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

Input: differentiable $\hat{V}(s,\omega)$, $\pi(a|s,\theta)$, $\alpha^\theta, \alpha^\omega > 0$ (e.g. $\alpha^\omega = 0.1\alpha^\theta$)

For each episode, initialize S, $I = 1$

For each step: $A \sim \pi(\cdot|S,\theta)$. Take A, observe S', R

$\delta = R + \gamma \hat{V}(S',\omega) - \hat{V}(S,\omega)$

$\omega = \omega + \alpha^\omega \delta \nabla \hat{V}(S,\omega)$, $\theta = \theta + \alpha^\theta I \delta \nabla \ln \pi(A|S,\theta)$

$I = I\gamma$, $S = S'$

__/__/____

→ G_t Although unbiased causes high variance. Now by using One-Step bootstrapped estimate, ↑ bias ↓ variance, accelerate learning.

→ Use $\hat{V}$ as baseline & estimate for q_π → Stabilizes learning
→ $\alpha^\theta < \alpha^w$ so critic can converge to actor. Otherwise target keeps moving

* Basic Actor Critic Algorithm

1- Take $a \sim \pi_\theta(a|s)$. Receive S, A, S', R

Critic
2- Update w using $(S, R + \gamma \hat{V}_w(S'))$ (Batch mode, online, Replay Buffer)
3- $\hat{\delta}(s,a) = R + \gamma \hat{V}_w(S') - \hat{V}_w(S)$

Actor
4- $\theta \leftarrow \theta + \alpha \hat{\delta}(s,a) \cdot \nabla_\theta \ln \pi_\theta(a|s)$

→ Step 2 & 4 are not always distinct. A shared network could update both.

Advantage Actor Critic.
Note that the Advantage Function $A_\pi(s,a) = q_\pi(s,a) - V_\pi(s) = \delta_\pi(s,a)$ for that part of TD-Error. Not expectation over all actions.

* A3C - Async Advantage Actor Critic

→ In order to learn π_θ, we want generate a lot of samples. Instead of doing it in sequence we can generate a lot of samples in parallel.

→ Varying episode lengths can cause the parameter updates to be inconsistent. Still works if the change is small enough.

→ Each agent is independent and hence sequence-actions are independent and thus the correlated sequence problem does not exist. as each agent will be using different sets of experiences.

→ A3C does not need to freeze targets either as it does not bootstrap. Uses n-step Return until the end. If episodes are nonterminal then you are bootstrapping much later.

__/__/____

* A2C - Synchronous Advantage Actor Critic

→ Remove Async updates. Train agents independently. Update only after all threads finish.

→ True Gradients are always computed & applied.

* Compatible Parametrization

→ Substituting $\hat{q}(s,a,\omega)$ for $q_\pi(s,a)$ may introduce bias. It can be proved the approximator has no bias if the following two conditions are satisfied.

$\hat{q}$ has to be linear in characteristic eligibility.

1. $\hat{q}(s,a,\omega) = \nabla_\theta \ln \pi_\theta(a|s)^T \omega$
   ↳ ω s.t. when we project characteristic elig along ω, we get same along dir of $\hat{q}$
2. ω minimizes MSE

$$\omega = \operatorname{argmin} E_{s\sim\rho^{\pi_\theta}, a\sim\pi_\theta}\left[\left(\hat{q}(s,a,\omega) - q_{\pi_\theta}(s,a)\right)^2\right]$$

* (Deep) Deterministic Policy Gradient

→ DPGs is typically used with continuous actions. If policy is deterministic then gradients become 0 & cannot really use policy gradient.

→ Expectation over state & actions complicates gradient estimation.

For deterministic policies, the performance objective is

$$J(\theta) = \int_S \mu(s)\, q_{\pi_\theta}(s, \pi_\theta(s))\, ds =$$

→ In a DPG, $\nabla_\theta J(\theta) = \int \mu^\pi(s) \nabla_a q_\pi(s,a) \cdot \nabla_\theta \pi_\theta(s) \Big|_{a=\pi_\theta(s)} ds$

→ we also avoid taking a max action by moving in the direction of q_π. Value of the new policy action is guaranteed to be higher than previous action.

3 → 1 → 4 → 2

__/__/____

→ For a wide class of problems DPG is a limiting case of the SPG,
UPDATE EQ:

$$\delta_t = r_t + \gamma \hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w)$$
$$w = w + \alpha_w \delta_t \nabla_w \hat{q}(s_t, a_t, w)$$
$$\theta = \theta + \alpha_\theta \nabla_a \hat{q}(s_t, a_t, w) \nabla_\theta \pi_\theta(s_t) \big|_{a=\pi_\theta(s_t)}$$

→ To ensure exploration,
(i) environment should be stochastic (or)
ii) use off-policy actor-critic where behaviour policy differs from estimation policy

__/__/____

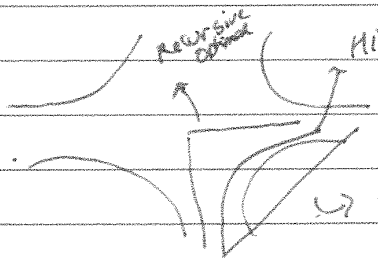
* Hierarchichal RL

Instead of thinking about actions repeating over as over in the State Space, think of Subproblems to solve.

+ Temporal Abstraction
+ Ability transfer Subproblems to other bigger problems.
+ More Powerful / Meaningful State Abstraction
  ↳ Reduced statespace, Greater usability

* Types of Optimality

→ Flat optimality < Hierarchical optimality < Recursive Optimality
(Lowest cost)



* Semi-MDP

→ Previously $S_t \to A_t - R_{t+1} - S_{t+1} \ldots$
For Semi MDP's,

$$S_t - A_t \cdots R_{t+T} - S_{t+T} - A_{t+T} \cdots S_{t+T+T'}$$

→ Actions take different amount of time to complete. So optimality depends not only on where the action takes but also on How long it takes.

$$\langle S, A, P, R \rangle$$

$$P(s', \tau \mid s, a), \quad E[r \mid s, a, s', \tau]$$

__/__/____

* SMDP Q-Learning

$$Q(s,a) = Q(s,a) + \alpha\left[r_{t+\tau} + \gamma^{\tau} \max_{a'} Q(s',a') - Q(s,a)\right]$$

* Options $O = \langle I_o, \pi_o, \beta_o \rangle$

- Macro Actions
- Encapsulates Policy

I_o : Initiation set

π_o : Policy (Between start & end state Policy)

$\beta_o : S \to [0,1]$

→ Markov options : π_o depends on the current state

→ Semi-Markov Options : π_o depends on history since option started

MDP + OPTIONS = SMDP

* Intra Option Q-Learning

π_o :

s_1	s_2	s_n
a_1	a_2	a_n
r_1	r_2	r_n

$$Q(s_1,a_1) = Q(s_1,a_1) + \alpha\left[r_1 + \gamma \max_{a} Q(s_2,a) - Q(s_1,a_1)\right]$$

→ Cannot use SARSA here because never had the choice to choose action in s_2, Following π_o. Hence you have to be off-policy and use Q-Learning.