Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

1 Policy Gradient Algorithms

It's time to take stock of what we have learnt so far to set up context for this chapter. So far, we have covered a range of RL Control algorithms, all of which are based on Generalized Policy Iteration (GPI). All of these algorithms perform GPI by learning the Q-Value Function and improving the policy by identifying the action that fetches the best Q-Value (i.e., action value) for each state. Notice that the way we implemented this *best action identification* is by sweeping through all the actions for each state. This works well only if the set of actions for each state is reasonably small. But if the action space is large/continuous, we have to resort to some sort of optimization method to identify the best action for each state, which is potentially complicated and expensive.

In this chapter, we cover RL Control algorithms that take a vastly different approach. These Control algorithms are still based on GPI, but the Policy Improvement of their GPI is not based on consulting the Q-Value Function, as has been the case with Control algorithms we covered in the previous two chapters. Rather, the approach in the class of algorithms we cover in this chapter is to directly find the Policy that fetches the "Best Expected Returns." Specifically, the algorithms of this chapter perform a Gradient Ascent on "Expected Returns" with the gradient defined with respect to the parameters of a Policy function approximation. We shall work with a stochastic policy of the form $\pi(s, a; \boldsymbol{\theta})$, with $\boldsymbol{\theta}$ denoting the parameters of the policy function approximation π. So we are basically learning this parameterized policy that selects actions without consulting a Value Function. Note that we might still engage a Value Function approximation (call it $Q(s; a; \boldsymbol{w})$) in our algorithm, but it's role is to only help learn the policy parameters $\boldsymbol{\theta}$ and not to identify the action with the best action-value for each state. So the two function approximations $\pi(s, a; \boldsymbol{\theta})$ and $Q(s, a; \boldsymbol{w})$ collaborate to improve the policy using gradient ascent (based on gradient of "expected returns" with respect to $\boldsymbol{\theta}$). $\pi(s, a; \boldsymbol{\theta})$ is the primary worker here (known as *Actor*) and $Q(s, a; \boldsymbol{w})$ is the support worker (known as *Critic*). The Critic parameters $\boldsymbol{w}$ are optimized by minimizing a suitable loss function defined in terms of $Q(s, a; \boldsymbol{w})$ while the Actor parameters $\boldsymbol{\theta}$ are optimized by maximizing a suitable "Expected Returns" function". Note that we still haven't defined what this"Expected Returns" function is (we will do so shortly), but we already see that this idea is appealing for large/continuous action spaces where sweeping through actions is infeasible. We will soon dig into the details of this new approach to RL Control (known as *Policy Gradient*, abbreviated as PG) - for now, it's important to recognize the big picture that PG is basically GPI with Policy Improvement done as a *Policy Gradient Ascent*.

The contrast between the RL Control algorithms covered in the previous two chapters and the algorithms of this chapter actually is part of the following bigger-picture classification of learning algorithms for Control:

- Value Function-based: Here we learn the Value Function (typically with a function approximation for the Value Function) and the Policy is implicit, readily derived from the Value Function (eg: ϵ-greedy).
- Policy-based: Here we learn the Policy (with a function approximation for the Policy), and there is no need to learn a Value Function.

- Actor-Critic: Here we primarily learn the Policy (with a function approximation for the Policy, known as *Actor*), and secondarily learn the Value Function (with a function approximation for the Value Function, known as *Critic*).

PG Algorithms can be Policy-based or Actor-Critic, whereas the Control algorithms we covered in the previous two chapters are Value Function-based.

In this chapter, we start by enumerating the advantages and disadvantages of Policy Gradient Algorithms, state and prove the Policy Gradient Theorem (which provides the fundamental calculation underpinning Policy Gradient Algorithms), then go on to address how to lower the bias and variance in these algorithms, give an overview of special cases of Policy Gradient algorithms that have found success in practical applications, and finish with a description of Evolutionary Strategies that although technically not RL, resemble Policy Gradient algorithms and are quite effective in solving certain Control problems.

1.1 Advantages and Disadvantages of Policy Gradient Algorithms

Let us start by enumerating the advantages of PG algorithms. We've already said that PG algorithms are effective in large action spaces, especially high-dimensional or continuous action spaces, because in such spaces selecting an action by deriving an improved policy from an updating Q-Value function is intractable. A key advantage of PG is that it naturally *explores* because the policy function approximation is configured as a stochastic policy. Moreover, PG finds the best Stochastic Policy. This is not a factor for MDPs since we know that there exists an optimal Deterministic Policy for any MDP but we often deal with Partially-Observable MDPs (POMDPs) in the real-world, for which the set of optimal policies might all be stochastic policies. We have an advantage in the case of MDPs as well since PG algorithms naturally converge to the deterministic policy (the variance in the policy distribution will automatically converge to 0) whereas in Value Function-based algorithms, we have to reduce the ϵ of the ϵ-greedy policy by-hand and the appropriate declining trajectory of ϵ is typically hard to figure out by manual tuning. In situations where the policy function is a simpler function compared to the Value Function, we naturally benefit from pursuing Policy-based algorithms than Value Function-based algorithms. Perhaps the biggest advantage of PG algorithms is that prior knowledge of the functional form of the Optimal Policy enables us to structure the known functional form in the function approximation for the policy. Lastly, PG offers numerical benefits as small changes in $\boldsymbol{\theta}$ yield small changes in π, and consequently small changes in the distribution of occurrences of states. This results in stronger convergence guarantees for PG algorithms relative to Value Function-based algorithms.

Now let's understand the disadvantages of PG Algorithms. The main disadvantage of PG Algorithms is that because they are based on gradient ascent, they typically converge to a local optimum whereas Value Function-based algorithms converge to a global optimum. Furthermore, the Policy Evaluation of PG is typically inefficient and can have high variance. Lastly, the Policy Improvements of PG happen in small steps and so, PG algorithms are slow to converge.

1.2 Policy Gradient Theorem

In this section, we start by setting up some notation, and then state and prove the Policy Gradient Theorem (abbreviated as PGT). The PGT provides the key calculation for PG Algorithms.

1.2.1 Notation and Definitions

Denoting the discount factor as γ, we shall assume either episodic sequences with $0 \leq \gamma \leq 1$ or non-episodic (continuing) sequences with $0 \leq \gamma < 1$. We shall use our usual notation of discrete-time, countable-spaces, time-homogeneous MDPs although we can indeed extend PGT and PG Algorithms to more general settings as well. We lighten $\mathcal{P}(s, a, s')$ notation to $\mathcal{P}^a_{s,s'}$ and $\mathcal{R}(s, a)$ notation to $\mathcal{R}^a_s$ because we want to save some space in the very long equations in the derivation of PGT.

We denote the probability distribution of the starting state as $p_0 : \mathcal{N} \to [0, 1]$. The policy function approximation is denoted as $\pi(s, a; \boldsymbol{\theta}) = \mathbb{P}[A_t = a | S_t = s; \boldsymbol{\theta}]$.

The PG coverage is quite similar for non-discounted, non-episodic MDPs, by considering the average-reward objective, but we won't cover it in this book.

Now we formalize the "Expected Returns" Objective $J(\boldsymbol{\theta})$.

$$J(\boldsymbol{\theta}) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot R_{t+1}]$$

Value Function $V^\pi(s)$ and Action Value function $Q^\pi(s, a)$ are defined as:

$$V^\pi(s) = \mathbb{E}_\pi[\sum_{k=t}^{\infty} \gamma^{k-t} \cdot R_{k+1} | S_t = s] \text{ for all } t = 0, 1, 2, \ldots$$

$$Q^\pi(s, a) = \mathbb{E}_\pi[\sum_{k=t}^{\infty} \gamma^{k-t} \cdot R_{k+1} | S_t = s, A_t = a] \text{ for all } t = 0, 1, 2, \ldots$$

$J(\boldsymbol{\theta}), V^\pi, Q^\pi$ are all measures of Expected Returns, so it pays to specify exactly how they differ. $J(\boldsymbol{\theta})$ is the Expected Return when following policy π (that is parameterized by θ), *averaged over all states $s \in \mathcal{N}$ and all actions $a \in \mathcal{A}$*. The idea is to perform a gradient ascent with $J(\theta)$ as the objective function, with each step in the gradient ascent essentially pushing $\boldsymbol{\theta}$ (and hence, π) in a desirable direction, until $J(\boldsymbol{\theta})$ is maximized. $V^\pi(s)$ is the Expected Return for a specific state $s \in \mathcal{N}$ when following policy π. $Q^\pi(s, a)$ is the Expected Return for a specific state $s \in \mathcal{N}$ and specific action $a \in \mathcal{A}$ when following policy π.

We define the *Advantage Function* as:

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

The advantage function captures how much more value does a particular action provide relative to the average value across actions (for a given state). The advantage function plays an important role in reducing the variance in PG Algorithms.

Also, $p(s \to s', t, \pi)$ will be a key function for us in the PGT proof - it denotes the probability of going from state s to s' in t steps by following policy π.

We express the "Expected Returns" Objective $J(\boldsymbol{\theta})$ as follows:

$$J(\boldsymbol{\theta}) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot R_{t+1}] = \sum_{t=0}^{\infty} \gamma^t \cdot \mathbb{E}_\pi[R_{t+1}]$$

$$= \sum_{t=0}^{\infty} \gamma^t \cdot \sum_{s \in \mathcal{N}} (\sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot p(S_0 \to s, t, \pi)) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot \mathcal{R}_s^a$$

$$= \sum_{s \in \mathcal{N}} (\sum_{S_0 \in \mathcal{N}} \sum_{t=0}^{\infty} \gamma^t \cdot p_0(S_0) \cdot p(S_0 \to s, t, \pi)) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot \mathcal{R}_s^a$$

Definition 1.2.1.

$$J(\boldsymbol{\theta}) = \sum_{s \in \mathcal{N}} \rho^\pi(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot \mathcal{R}_s^a$$

where

$$\rho^\pi(s) = \sum_{S_0 \in \mathcal{N}} \sum_{t=0}^{\infty} \gamma^t \cdot p_0(S_0) \cdot p(S_0 \to s, t, \pi)$$

is the key function (for PG) that we shall refer to as *Discounted-Aggregate State-Visitation Measure*. Note that $\rho^\pi(s)$ is a measure over the set of non-terminal states, but is not a probability measure. Think of $\rho^\pi(s)$ as weights reflecting the relative likelihood of occurrence of states on a trace experience (adjusted for discounting, i.e, lesser importance to reaching a state later on a trace experience). We can still talk about the distribution of states under the measure ρ^π, but we say that this distribution is *improper* to convey the fact that $\sum_{s \in \mathcal{N}} \rho^\pi(s) \neq 1$ (i.e., the distribution is not normalized). We talk about this improper distribution of states under the measure ρ^π so we can use (as a convenience) the "expected value" notation for any random variable $f : \mathcal{N} \to \mathbb{R}$ under this improper distribution, i.e., we use the notation:

$$\mathbb{E}_{s \sim \rho^\pi}[f(s)] = \sum_{s \in \mathcal{N}} \rho^\pi(s) \cdot f(s)$$

Using this notation, we can re-write the above definition of $J(\boldsymbol{\theta})$ as:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{s \sim \rho^\pi, a \sim \pi}[\mathcal{R}_s^a]$$

1.2.2 Statement of the Policy Gradient Theorem

The Policy Gradient Theorem (PGT) provides a powerful formula for the gradient of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ so we can perform Gradient Ascent. The key challenge is that $J(\boldsymbol{\theta})$ depends not only on the selection of actions through policy π (parameterized by $\boldsymbol{\theta}$), but also on the probability distribution of occurrence of states (also affected by π, and hence by $\boldsymbol{\theta}$). With knowledge of the functional form of π on θ, it is not difficult to evaluate the dependency of actions selection on $\boldsymbol{\theta}$, but evaluating the dependency of the probability distribution of occurrence of states on $\boldsymbol{\theta}$ is difficult since the environment only provides atomic experiences at a time (and not probabilities of transitions). However, the PGT (below) comes to our rescue because the gradient of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ involves only the gradient of π with respect to $\boldsymbol{\theta}$, and not the gradient of the probability distribution of occurrence of states with respect to $\boldsymbol{\theta}$. Precisely, we have:

Theorem 1.2.1 (Policy Gradient Theorem)**.**

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in \mathcal{N}} \rho^\pi(s) \cdot \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta}) \cdot Q^\pi(s, a)$$

As mentioned above, note that $\rho^\pi(s)$ (representing the discounting-adjusted probability distribution of occurrence of states, ignoring normalizing factor turning the ρ^π measure into a probability measure) depends on $\boldsymbol{\theta}$ but there's no $\nabla_{\boldsymbol{\theta}}\rho^\pi(s)$ term in $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$.

Also note that:

$$\nabla_{\boldsymbol{\theta}}\pi(s,a;\boldsymbol{\theta}) = \pi(s,a;\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} \log \pi(s,a;\boldsymbol{\theta})$$

$\nabla_{\boldsymbol{\theta}} \log \pi(s,a;\boldsymbol{\theta})$ is the Score function (Gradient of log-likelihood) that is commonly used in Statistics.

Since ρ^π is the *Discounted-Aggregate State-Visitation Measure*, we can sample-estimate $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ by calculating $\gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta})) \cdot Q^\pi(S_t, A_t)$ at each time step in each trace experience (noting that the state occurrence probabilities and action occurrence probabilities are implicit in the trace experiences, and ignoring the probability measure-normalizing factor), and update the parameters $\boldsymbol{\theta}$ (according to Stochastic Gradient Ascent) using each atomic experience's $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ estimate.

We typically calculate the Score $\nabla_{\boldsymbol{\theta}} \log \pi(s,a;\boldsymbol{\theta})$ using an analytically-convenient functional form for the conditional probability distribution $a|s$ (in terms of $\boldsymbol{\theta}$) so that the derivative of the logarithm of this functional form is analytically tractable (this will be clear in the next section when we consider a couple of examples of canonical functional forms for $a|s$). In many PG Algorithms, we estimate $Q^\pi(s,a)$ with a function approximation $Q(s,a;\boldsymbol{w})$. We will later show how to avoid the estimate bias of $Q(s,a;\boldsymbol{w})$.

Thus, the PGT enables a numerical estimate of $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ which in turn enables *Policy Gradient Ascent*.

1.2.3 Proof of the Policy Gradient Theorem

We begin the proof by noting that:

$$J(\boldsymbol{\theta}) = \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot V^\pi(S_0) = \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^\pi(S_0, A_0)$$

Calculate $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ by it's product parts $\pi(S_0, A_0; \boldsymbol{\theta})$ and $Q^\pi(S_0, A_0)$.

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) = &\sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}}\pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^\pi(S_0, A_0) \\ &+ \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} Q^\pi(S_0, A_0)\end{aligned}$$

Now expand $Q^\pi(S_0, A_0)$ as:

$$\mathcal{R}_{S_0}^{A_0} + \sum_{S_1 \in \mathcal{N}} \gamma \cdot \mathcal{P}_{S_0,S_1}^{A_0} \cdot V^\pi(S_1) \text{ (Bellman Policy Equation)}$$

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) = &\sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}}\pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^\pi(S_0, A_0) \\ &+ \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}}(\mathcal{R}_{S_0}^{A_0} + \sum_{S_1 \in \mathcal{N}} \gamma \cdot \mathcal{P}_{S_0,S_1}^{A_0} \cdot V^\pi(S_1))\end{aligned}$$

Note: $\nabla_\theta \mathcal{R}_{S_0}^{A_0} = 0$, so remove that term.

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) = &\sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}}\pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^\pi(S_0, A_0) \\ &+ \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}}(\sum_{S_1 \in \mathcal{N}} \gamma \cdot \mathcal{P}_{S_0,S_1}^{A_0} \cdot V^\pi(S_1))\end{aligned}$$

Now bring the $\nabla_{\boldsymbol{\theta}}$ inside the $\sum_{S_1 \in \mathcal{N}}$ to apply only on $V^{\pi}(S_1)$.

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^{\pi}(S_0, A_0) + \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot \sum_{S_1 \in \mathcal{N}} \gamma \cdot \mathcal{P}^{A_0}_{S_0, S_1} \cdot \nabla_{\boldsymbol{\theta}} V^{\pi}(S_1)$$

Now bring $\sum_{S_0 \in \mathcal{N}}$ and $\sum_{A_0 \in \mathcal{A}}$ inside the $\sum_{S_1 \in \mathcal{N}}$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^{\pi}(S_0, A_0) + \sum_{S_1 \in \mathcal{N}} \sum_{S_0 \in \mathcal{N}} \gamma \cdot p_0(S_0) \cdot (\sum_{A_0 \in \mathcal{A}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot \mathcal{P}^{A_0}_{S_0, S_1}) \cdot \nabla_{\boldsymbol{\theta}} V^{\pi}(S_1)$$

Note that $\sum_{A_0 \in \mathcal{A}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot \mathcal{P}^{A_0}_{S_0, S_1} = p(S_0 \to S_1, 1, \pi)$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^{\pi}(S_0, A_0) + \sum_{S_1 \in \mathcal{N}} \sum_{S_0 \in \mathcal{N}} \gamma \cdot p_0(S_0) \cdot p(S_0 \to S_1, 1, \pi) \cdot \nabla_{\boldsymbol{\theta}} V^{\pi}(S_1)$$

Now expand $V^{\pi}(S_1)$ to $\sum_{A_1 \in \mathcal{A}} \pi(S_1, A_1; \boldsymbol{\theta}) \cdot Q^{\pi}(S_1, A_1)$

.

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^{\pi}(S_0, A_0) + \sum_{S_1 \in \mathcal{N}} \sum_{S_0 \in \mathcal{N}} \gamma \cdot p_0(S_0) \cdot p(S_0 \to S_1, 1, \pi) \cdot \nabla_{\boldsymbol{\theta}} (\sum_{A_1 \in \mathcal{A}} \pi(S_1, A_1; \boldsymbol{\theta}) \cdot Q^{\pi}(S_1, A_1))$$

We are now back to when we started calculating gradient of $\sum_a \pi \cdot Q^{\pi}$. Follow the same process of calculating the gradient of $\pi \cdot Q^{\pi}$ by parts, then Bellman-expanding Q^{π} (to calculate its gradient), and iterate.

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{S_0 \in \mathcal{N}} p_0(S_0) \cdot \sum_{A_0 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_0, A_0; \boldsymbol{\theta}) \cdot Q^{\pi}(S_0, A_0) +$$

$$\sum_{S_1 \in \mathcal{N}} \sum_{S_0 \in \mathcal{N}} \gamma \cdot p_0(S_0) \cdot p(S_0 \to S_1, 1, \pi) \cdot (\sum_{A_1 \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_1, A_1; \boldsymbol{\theta}) \cdot Q^{\pi}(S_1, A_1) + \ldots)$$

This iterative process leads us to:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{t=0}^{\infty} \sum_{S_t \in \mathcal{N}} \sum_{S_0 \in \mathcal{N}} \gamma^t \cdot p_0(S_0) \cdot p(S_0 \to S_t, t, \pi) \cdot \sum_{A_t \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_t, A_t; \boldsymbol{\theta}) \cdot Q^{\pi}(S_t, A_t)$$

Bring $\sum_{t=0}^{\infty}$ inside $\sum_{S_t \in \mathcal{N}} \sum_{S_0 \in \mathcal{N}}$ and note that

$\sum_{A_t \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(S_t, A_t; \boldsymbol{\theta}) \cdot Q^{\pi}(S_t, A_t)$ is independent of t

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in \mathcal{N}} \sum_{S_0 \in \mathcal{N}} \sum_{t=0}^{\infty} \gamma^t \cdot p_0(S_0) \cdot p(S_0 \to s, t, \pi) \cdot \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta}) \cdot Q^{\pi}(s, a)$$

Remember that $\sum_{S_0 \in \mathcal{N}} \sum_{t=0}^{\infty} \gamma^t \cdot p_0(S_0) \cdot p(S_0 \to s, t, \pi) \stackrel{\text{def}}{=} \rho^{\pi}(s)$. So,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta}) \cdot Q^{\pi}(s, a)$$

$$\mathbb{Q}.\mathbb{E}.\mathbb{D}.$$

This proof is borrowed from the Appendix of the famous paper by Sutton, McAllester, Singh, Mansour on Policy Gradient Methods for Reinforcement Learning with Function Approximation (Sutton et al. 2001).

Note that using the "Expected Value" notation under the improper distribution implied by the Discounted-Aggregate State-Visitation Measure ρ^{π}, we can write the statement of PGT as:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta})) \cdot Q^{\pi}(s, a) \\ &= \mathbb{E}_{s \sim \rho^{\pi}, a \sim \pi}[(\nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta})) \cdot Q^{\pi}(s, a)] \end{aligned}$$

As explained earlier, since the state occurrence probabilities and action occurrence probabilities are implicit in the trace experiences, we can sample-estimate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ by calculating $\gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta})) \cdot Q^{\pi}(S_t, A_t)$ at each time step in each trace experience, and update the parameters $\boldsymbol{\theta}$ (according to Stochastic Gradient Ascent) with this calculation.

1.3 Score function for Canonical Policy Functions

Now we illustrate how the Score function $\nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta})$ is calculated using an analytically-convenient functional form for the conditional probability distribution $a|s$ (in terms of $\boldsymbol{\theta}$) so that the derivative of the logarithm of this functional form is analytically tractable. We do this for a couple of canonical functional forms for $a|s$, one for finite action spaces and one for single-dimensional continuous action spaces.

1.3.1 Canonical $\pi(s, a; \boldsymbol{\theta})$ for Finite Action Spaces

For finite action spaces, we often use the Softmax Policy. Assume $\boldsymbol{\theta}$ is an m-vector $(\theta_1, \ldots, \theta_m)$ and assume feature vector $\boldsymbol{\phi}(s, a)$ is given by: $(\phi_1(s, a), \ldots, \phi_m(s, a))$ for all $s \in \mathcal{N}, a \in \mathcal{A}$.

We weight actions using linear combinations of features, i.e., $\boldsymbol{\phi}(s, a)^T \cdot \boldsymbol{\theta}$, and we set the action probabilities to be proportional to exponentiated weights, as follows:

$$\pi(s, a; \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\phi}(s,a)^T \cdot \boldsymbol{\theta}}}{\sum_{b \in \mathcal{A}} e^{\boldsymbol{\phi}(s,b)^T \cdot \boldsymbol{\theta}}} \text{ for all } s \in \mathcal{N}, a \in \mathcal{A}$$

Then the score function is given by:

$$\nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) = \boldsymbol{\phi}(s, a) - \sum_{b \in \mathcal{A}} \pi(s, b; \boldsymbol{\theta}) \cdot \boldsymbol{\phi}(s, b) = \boldsymbol{\phi}(s, a) - \mathbb{E}_{\pi}[\boldsymbol{\phi}(s, \cdot)]$$

The intuitive interpretation is that the score function for an action a represents the "advantage" of the feature vector for action a over the mean feature vector (across all actions), for a given state s.

1.3.2 Canonical $\pi(s, a; \boldsymbol{\theta})$ for Single-Dimensional Continuous Action Spaces

For single-dimensional continuous action spaces (i.e., $\mathcal{A} = \mathbb{R}$), we often use a Gaussian distribution for the Policy. Assume $\boldsymbol{\theta}$ is an m-vector $(\theta_1, \ldots, \theta_m)$ and assume the state features vector $\boldsymbol{\phi}(s)$ is given by $(\phi_1(s), \ldots, \phi_m(s))$ for all $s \in \mathcal{N}$.

We set the mean of the gaussian distribution for the Policy as a linear combination of state features, i.e., $\boldsymbol{\phi}(s)^T \cdot \boldsymbol{\theta}$, and we set the variance to be a fixed value, say σ^2. We could make the variance parameterized as well, but let's work with fixed variance to keep things simple.

The Gaussian policy selects an action a as follows:

$$a \sim \mathcal{N}(\boldsymbol{\phi}(s)^T \cdot \boldsymbol{\theta}, \sigma^2) \text{ for a given } s \in \mathcal{N}$$

Then the score function is given by:

$$\nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) = \frac{(a - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{\theta}) \cdot \boldsymbol{\phi}(s)}{\sigma^2}$$

This is easily extensible to multi-dimensional continuous action spaces by considering a multi-dimensional gaussian distribution for the Policy.

The intuitive interpretation is that the score function for an action a is proportional to the feature vector for given state s scaled by the "advantage" of the action a over the mean action (note: each $a \in \mathbb{R}$).

For each of the above two examples (finite action spaces and continuous action spaces), think of the "features advantage" of an action as the compass for the Gradient Ascent. The gradient estimate for an encountered action is proportional to the action's "features advantage" scaled by the action's Value Function. The intuition is that the Gradient Ascent encourages picking actions that are yielding more favorable outcomes (*Policy Improvement*) so as to ultimately get to a point where the optimal action is selected for each state.

1.4 REINFORCE Algorithm (Monte-Carlo Policy Gradient)

Now we are ready to write our first Policy Gradient algorithm. As ever, the simplest algorithm is a Monte-Carlo algorithm. In the case of Policy Gradient, a simple Monte-Carlo calculation provides us with an algorithm known as REINFORCE, due to R.J.Williams (Williams 1992), which we cover in this section.

We've already explained that we can calculate the Score function using an analytical derivative of a specified functional form for $\pi(S_t, A_t; \boldsymbol{\theta})$ for each atomic experience (S_t, A_t, R_t, S_{t+1}). What remains is to obtain an estimate of $Q^\pi(S_t, A_t)$ for each atomic experience (S_t, A_t, R_t, S_{t+1}). REINFORCE uses the trace experience return G_t for (S_t, A_t), while following policy π, as an unbiased sample of $Q^\pi(S_t, A_t)$. Thus, at every time step (i.e., at every atomic experience) in each episode, we estimate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ by calculating $\gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta})) \cdot G_t$ (noting that the state occurrence probabilities and action occurrence probabilities are implicit in the trace experiences), and update the parameters $\boldsymbol{\theta}$ at the end of each episode (using each atomic experience's $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ estimate) according to Stochastic Gradient Ascent as follows:

$$\Delta \boldsymbol{\theta} = \alpha \cdot \gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta})) \cdot G_t$$

where α is the learning rate.

This Policy Gradient algorithm is Monte-Carlo because it is not bootstrapped (complete returns are used as an unbiased sample of Q^π, rather than a bootstrapped estimate). In terms of our previously-described classification of RL algorithms as Value Function-based or Policy-based or Actor-Critic, REINFORCE is a Policy-based algorithm since REINFORCE does not involve learning a Value Function.

Now let's write some code to implement the REINFORCE algorithm. In this chapter, we will focus our Python code implementation of Policy Gradient algorithms to continuous action spaces, although it should be clear based on the discussion so far that the Policy Gradient approach applies to arbitrary action spaces (we've already seen an example of the policy function parameterization for discrete action spaces). To keep things simple, the function `reinforce_gaussian` below implements REINFORCE for the simple case of single-dimensional continuous action space (i.e. $\mathcal{A} = \mathbb{R}$), although this can be easily extended to multi-dimensional continuous action spaces. So in the code below, we work with a generic state space given by `TypeVar('S')` and the action space is specialized to `float` (representing $\mathbb{R}$).

As seen earlier in the canonical example for single-dimensional continuous action space, we assume a Gaussian distribution for the policy. Specifically, the policy is represented by an arbitrary parameterized function approximation using the class `FunctionApprox`. As a reminder, an instance of `FunctionApprox` represents a probability distribution function f of the conditional random variable variable $y|x$ where x belongs to an arbitrary domain $\mathcal{X}$ and $y \in \mathbb{R}$ (probability of y conditional on x denoted as $f(x; \boldsymbol{\theta})(y)$ where $\boldsymbol{\theta}$ denotes the parameters of the `FunctionApprox`). Note that the `evaluate` method of `FunctionApprox` takes as input an `Iterable` of x values and calculates $g(x; \boldsymbol{\theta}) = \mathbb{E}_{f(x;\boldsymbol{\theta})}[y]$ for each of the x values. In our case here, x represents non-terminal states in $\mathcal{N}$ and y represents actions in $\mathbb{R}$, so $f(s; \boldsymbol{\theta})$ denotes the probability distribution of actions, conditional on state $s \in \mathcal{N}$, and $g(s; \boldsymbol{\theta})$ represents the *Expected Value* of (real-numbered) actions, conditional on state $s \in \mathcal{N}$. Since we have assumed the policy to be Gaussian,

$$\pi(s, a; \boldsymbol{\theta}) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(a-g(s;\boldsymbol{\theta}))^2}{2\sigma^2}}$$

To be clear, our code below works with the `@abstractclass FunctionApprox` (meaning it is an arbitrary parameterized function approximation) with the assumption that the probability distribution of actions given a state is Gaussian whose variance σ^2 is assumed to be a constant. Assume we have m features for our function approximation, denoted as $\boldsymbol{\phi}(s) = (\phi_1(s), \ldots, \phi_m(s))$ for all $s \in \mathcal{N}$.

σ is specified in the code below as `policy_stdev`. The input `policy_mean_approx0: FunctionApprox[NonTermina` specifies the function approximation we initialize the algorithm with (it is up to the user of `reinforce_gaussian` to configure `policy_mean_approx0` with the appropriate functional form for the function approximation, the hyper-parameter values, and the initial values of the parameters $\boldsymbol{\theta}$ that we want to solve for).

The Gaussian policy (of the type `GaussianPolicyFromApprox`) selects an action a (given state s) by sampling from the gaussian distribution defined by mean $g(s; \boldsymbol{\theta})$ and variance σ^2.

The score function is given by:

$$\nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) = \frac{(a - g(s; \boldsymbol{\theta})) \cdot \nabla_{\boldsymbol{\theta}} g(s; \boldsymbol{\theta})}{\sigma^2}$$

The outer loop of `while True:` loops over trace experiences produced by the method `simulate_actions` of the input `mdp` for a given input `start_states_distribution` (specify-

ing the initial states distribution $p_0 : \mathcal{N} \to [0, 1]$), and the current policy π (that is parameterized by $\boldsymbol{\theta}$, which updates after each trace experience). The inner loop loops over an `Iterator` of `step: ReturnStep[S, float]` objects produced by the `returns` method for each trace experience.

The variable `grad` is assigned the value of the negative score for an encountered (S_t, A_t) in a trace experience, i.e., it is assigned the value:

$$-\nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta}) = \frac{(g(S_t; \boldsymbol{\theta}) - A_t) \cdot \nabla_{\boldsymbol{\theta}} g(S_t; \boldsymbol{\theta})}{\sigma^2}$$

We negate the sign of the score because we are performing Gradient Ascent rather than Gradient Descent (the `FunctionApprox` class has been written for Gradient Descent). The variable `scaled_grad` multiplies the negative of score (`grad`) with γ^t (`gamma_prod`) and return G_t (`step.return_`). The rest of the code should be self-explanatory.

`reinforce_gaussian` returns an `Iterable` of `FunctionApprox` representing the stream of updated policies $\pi(s, \cdot; \boldsymbol{\theta})$, with each of these `FunctionApprox` being generated (using `yield`) at the end of each trace experience.

```
import numpy as np
from rl.distribution import Distribution, Gaussian
from rl.policy import Policy
from rl.markov_process import NonTerminal
from rl.markov_decision_process import MarkovDecisionProcess, TransitionStep
from rl.function_approx import FunctionApprox, Gradient

S = TypeVar('S')

@dataclass(frozen=True)
class GaussianPolicyFromApprox(Policy[S, float]):
    function_approx: FunctionApprox[NonTerminal[S]]
    stdev: float

    def act(self, state: NonTerminal[S]) -> Gaussian:
        return Gaussian(
            mu=self.function_approx(state),
            sigma=self.stdev
        )

def reinforce_gaussian(
    mdp: MarkovDecisionProcess[S, float],
    policy_mean_approx0: FunctionApprox[NonTerminal[S]],
    start_states_distribution: Distribution[NonTerminal[S]],
    policy_stdev: float,
    gamma: float,
    episode_length_tolerance: float
) -> Iterator[FunctionApprox[NonTerminal[S]]]:
    policy_mean_approx: FunctionApprox[NonTerminal[S]] = policy_mean_approx0
    yield policy_mean_approx
    while True:
        policy: Policy[S, float] = GaussianPolicyFromApprox(
            function_approx=policy_mean_approx,
            stdev=policy_stdev
        )
        trace: Iterable[TransitionStep[S, float]] = mdp.simulate_actions(
            start_states=start_states_distribution,
            policy=policy
        )
        gamma_prod: float = 1.0
        for step in returns(trace, gamma, episode_length_tolerance):
            def obj_deriv_out(
                states: Sequence[NonTerminal[S]],
                actions: Sequence[float]
            ) -> np.ndarray:
```

```
                return (policy_mean_approx.evaluate(states) -
                        np.array(actions)) / (policy_stdev * policy_stdev)
            grad: Gradient[FunctionApprox[NonTerminal[S]]] = \
                policy_mean_approx.objective_gradient(
                    xy_vals_seq=[(step.state, step.action)],
                    obj_deriv_out_fun=obj_deriv_out
            )
            scaled_grad: Gradient[FunctionApprox[NonTerminal[S]]] = \
                grad * gamma_prod * step.return_
            policy_mean_approx = \
                policy_mean_approx.update_with_gradient(scaled_grad)
            gamma_prod *= gamma
        yield policy_mean_approx
```

The above code is in the file rl/policy_gradient.py.

1.5 Optimal Asset Allocation (Revisited)

In this chapter, we will test the PG algorithms we implement on the Optimal Asset Allocation problem of Chapter **??**, specifically the setting of the class `AssetAllocDiscrete` covered in Section **??**. As a reminder, in this setting, we have a single risky asset and at each of a fixed finite number of time steps, one has to make a choice of the quantity of current wealth to invest in the risky asset (remainder in the riskless asset) with the goal of maximizing the expected utility of wealth at the end of the finite horizon. Thus, this finite-horizon MDP's state at any time t is the pair (t, W_t) where $W_t \in \mathbb{R}$ denotes the wealth at time t, and the action at time t is the investment $x_t \in \mathbb{R}$ in the risky asset.

We provided an ADP backward-induction solution to this problem in Chapter **??**, implemented with `AssetAllocDiscrete` (code in rl/chapter7/asset_alloc_discrete.py). Now we want to solve it with PG algorithms, starting with REINFORCE. So we require a new interface and hence, we implement a new class `AssetAllocPG` with appropriate tweaks to `AssetAllocDiscrete`. The key change in the interface is that we have inputs `policy_feature_funcs`, `policy_mean_dnn_spec` and `policy_stdev` (see code below). `policy_feature_funcs` represents the sequence of feature functions for the `FunctionApprox` representing the mean action for a given state (i.e., $g(s; \boldsymbol{\theta}) = \mathbb{E}_{f(s;\boldsymbol{\theta})}[a]$ where f represents the policy probability distribution of actions for a given state). `policy_mean_dnn_spec` specifies the architecture of a deep neural network a user would like to use for the `FunctionApprox`. `policy_stdev` represents the fixed standard deviation σ of the policy probability distribution of actions for any state. Unlike the backward-induction solution of `AssetAllocDiscrete` where we had to model a separate MDP for each time step in the finite horizon (where the state for each time step's MDP is the wealth), here we model a single MDP across all time steps with the state as the pair of time step index t and the wealth W_t. `AssetAllocState = Tuple[int, float]` is the data type for the state (t, W_t).

```
from rl.function_approx import DNNSpec

AssetAllocState = Tuple[int, float]

@dataclass(frozen=True)
class AssetAllocPG:
    risky_return_distributions: Sequence[Distribution[float]]
    riskless_returns: Sequence[float]
    utility_func: Callable[[float], float]
    policy_feature_funcs: Sequence[Callable[[AssetAllocState], float]]
    policy_mean_dnn_spec: DNNSpec
    policy_stdev: float
    policy_mean
    initial_wealth_distribution: Distribution[float]
```

The method `get_mdp` below sets up this MDP (should be self-explanatory as the construction is very similar to the construction of the single-step MDPs in `AssetAllocDiscrete`).

```
from rl.distribution import SampledDistribution
    def time_steps(self) -> int:
        return len(self.risky_return_distributions)

    def get_mdp(self) -> MarkovDecisionProcess[AssetAllocState, float]:
        steps: int = self.time_steps()
        distrs: Sequence[Distribution[float]] = self.risky_return_distributions
        rates: Sequence[float] = self.riskless_returns
        utility_f: Callable[[float], float] = self.utility_func

        class AssetAllocMDP(MarkovDecisionProcess[AssetAllocState, float]):

            def step(
                self,
                state: NonTerminal[AssetAllocState],
                action: float
            ) -> SampledDistribution[Tuple[State[AssetAllocState], float]]:

                def sr_sampler_func(
                    state=state,
                    action=action
                ) -> Tuple[State[AssetAllocState], float]:
                    time, wealth = state.state
                    next_wealth: float = action * (1 + distrs[time].sample()) \
                        + (wealth - action) * (1 + rates[time])
                    reward: float = utility_f(next_wealth) \
                        if time == steps - 1 else 0.
                    next_pair: AssetAllocState = (time + 1, next_wealth)
                    next_state: State[AssetAllocState] = \
                        Terminal(next_pair) if time == steps - 1 \
                        else NonTerminal(next_pair)
                    return (next_state, reward)

                return SampledDistribution(sampler=sr_sampler_func)

            def actions(self, state: NonTerminal[AssetAllocState]) \
                    -> Sequence[float]:
                return []

        return AssetAllocMDP()
```

The methods `start_states_distribution` and `policy_mean_approx` below create the `SampledDistribution` of start states and the `DNNApprox` representing the mean action for a given state respectively. Finally, the `reinforce` method below simply collects all the ingredients and passes along to `reinforce_gaussian` to solve this asset allocation problem.

```
from rl.function_approx import AdamGradient, DNNApprox
    def start_states_distribution(self) -> \
            SampledDistribution[NonTerminal[AssetAllocState]]:

        def start_states_distribution_func() -> NonTerminal[AssetAllocState]:
            wealth: float = self.initial_wealth_distribution.sample()
            return NonTerminal((0, wealth))

        return SampledDistribution(sampler=start_states_distribution_func)

    def policy_mean_approx(self) -> \
            FunctionApprox[NonTerminal[AssetAllocState]]:
        adam_gradient: AdamGradient = AdamGradient(
            learning_rate=0.003,
            decay1=0.9,
            decay2=0.999
        )
        ffs: List[Callable[[NonTerminal[AssetAllocState]], float]] = []
```

```
        for f in self.policy_feature_funcs:
            def this_f(st: NonTerminal[AssetAllocState], f=f) -> float:
                return f(st.state)
            ffs.append(this_f)
        return DNNApprox.create(
            feature_functions=ffs,
            dnn_spec=self.policy_mean_dnn_spec,
            adam_gradient=adam_gradient
        )

    def reinforce(self) -> \
            Iterator[FunctionApprox[NonTerminal[AssetAllocState]]]:
        return reinforce_gaussian(
            mdp=self.get_mdp(),
            policy_mean_approx0=self.policy_mean_approx(),
            start_states_distribution=self.start_states_distribution(),
            policy_stdev=self.policy_stdev,
            gamma=1.0,
            episode_length_tolerance=1e-5
        )
```

The above code is in the file rl/chapter13/asset_alloc_pg.py.

Let's now test this out on an instance of the problem for which we have a closed-form solution (so we can verify the REINFORCE solution against the closed-form solution). The special instance is the setting covered in Section **??** of Chapter **??**. From Equation (**??**), we know that the optimal action in state (t, W_t) is linear in a single feature defined as $(1 + r)^t$ where r is the constant riskless rate across time steps. So we need to set up the function approximation `policy_mean_dnn_spec: DNNSpec` as linear in this single feature (no hidden layers and identity function as the output layer activation function), and check if the optimized weight (coefficient of this single feature) matches up with the closed-form solution of Equation (**??**).

Let us use similar settings that we had used in Chapter **??** to test `AssetAllocDiscrete`. In the code below, we create an instance of `AssetAllocPG` with time steps $T = 5$, $\mu = 13\%, \sigma = 20\%, r = 7\%$, coefficient of CARA $a = 1.0$. We set up `risky_return_distributions` as a sequence of identical `Gaussian` distributions, `riskless_returns` as a sequence of identical riskless rate of returns, and `utility_func` as a `lambda` parameterized by the coefficient of CARA a. We set the probability distribution of wealth at time $t = 0$ (start of each trace experience) as $\mathcal{N}(1.0, 0.1)$, and we set the constant standard deviation σ of the policy probability distribution of actions for a given state as 0.5.

```
steps: int = 5
mu: float = 0.13
sigma: float = 0.2
r: float = 0.07
a: float = 1.0
init_wealth: float = 1.0
init_wealth_stdev: float = 0.1
policy_stdev: float = 0.5
```

Next, we print the closed-form solution of the optimal action for states at each time step (note: the closed-form solution for optimal action is independent of wealth W_t, and is only dependent on t).

```
base_alloc: float = (mu - r) / (a * sigma * sigma)
for t in range(steps):
    alloc: float = base_alloc / (1 + r) ** (steps - t - 1)
    print(f"Time {t:d}: Optimal Risky Allocation = {alloc:.3f}")
```

This prints:

```
Time 0: Optimal Risky Allocation = 1.144
Time 1: Optimal Risky Allocation = 1.224
Time 2: Optimal Risky Allocation = 1.310
Time 3: Optimal Risky Allocation = 1.402
Time 4: Optimal Risky Allocation = 1.500
```

Next we set up an instance of `AssetAllocPG` with the above parameters. Note that the `policy_mean_dnn_spec` argument to the constructor of `AssetAllocPG` is set up as a trivial neural network with no hidden layers and the identity function as the output layer activation function. Note also that the `policy_feature_funcs` argument to the constructor is set up with the single feature function $(1 + r)^t$.

```
from rl.distribution import Gaussian
from rl.function_approx import
risky_ret: Sequence[Gaussian] = [Gaussian(mu=mu, sigma=sigma)
                                 for _ in range(steps)]
riskless_ret: Sequence[float] = [r for _ in range(steps)]
utility_function: Callable[[float], float] = lambda x: - np.exp(-a * x) / a
policy_feature_funcs: Sequence[Callable[[AssetAllocState], float]] = \
    [
        lambda w_t: (1 + r) ** w_t[1]
    ]
init_wealth_distr: Gaussian = Gaussian(mu=init_wealth, sigma=init_wealth_stdev)
policy_mean_dnn_spec: DNNSpec = DNNSpec(
    neurons=[],
    bias=False,
    hidden_activation=lambda x: x,
    hidden_activation_deriv=lambda y: np.ones_like(y),
    output_activation=lambda x: x,
    output_activation_deriv=lambda y: np.ones_like(y)
)
aad: AssetAllocPG = AssetAllocPG(
    risky_return_distributions=risky_ret,
    riskless_returns=riskless_ret,
    utility_func=utility_function,
    policy_feature_funcs=policy_feature_funcs,
    policy_mean_dnn_spec=policy_mean_dnn_spec,
    policy_stdev=policy_stdev,
    initial_wealth_distribution=init_wealth_distr
)
```

Next, we invoke the method `reinforce` of this `AssetAllocPG` instance. In practice, we'd have parameterized the standard deviation of the policy probability distribution just like we parameterized the mean of the policy probability distribution, and we'd have updated those parameters in a similar manner (the standard deviation would converge to 0, i.e., the policy would converge to the optimal deterministic policy given by the closed-form solution). As an exercise, extend the function `reinforce_gaussian` to include a second `FunctionApprox` for the standard deviation of the policy probability distribution and update this `FunctionApprox` along with the updates to the mean `FunctionApprox`. However, since we set the standard deviation of the policy probability distribution to be a constant σ and since we use a Monte-Carlo method, the variance of the mean estimate of the policy probability distribution is significantly high. So we take the average of the mean estimate over several iterations (below we average the estimate from iteration 10000 to iteration 20000).

```
reinforce_policies: Iterator[FunctionApprox[
    NonTerminal[AssetAllocState]]] = aad.reinforce()
```

```
num_episodes: int = 10000
averaging_episodes: int = 10000
policies: Sequence[FunctionApprox[NonTerminal[AssetAllocState]]] = \
    list(itertools.islice(
        reinforce_policies,
        num_episodes,
        num_episodes + averaging_episodes
    ))
for t in range(steps):
    opt_alloc: float = np.mean([p(NonTerminal((init_wealth, t)))
                                for p in policies])
    print(f"Time {t:d}: Optimal Risky Allocation = {opt_alloc:.3f}")
```

This prints:

```
Time 0: Optimal Risky Allocation = 1.215
Time 1: Optimal Risky Allocation = 1.300
Time 2: Optimal Risky Allocation = 1.392
Time 3: Optimal Risky Allocation = 1.489
Time 4: Optimal Risky Allocation = 1.593
```

So we see that the estimate of the mean action for the 5 time steps from our implementation of the REINFORCE method gets fairly close to the closed-form solution.

The above code is in the file rl/chapter13/asset_alloc_reinforce.py. As ever, we encourage you to tweak the parameters and explore how the results vary.

As an exercise, we encourage you to implement an extension of this problem. Along with the risky asset allocation choice as the action at each time step, also include a consumption quantity (wealth to be extracted at each time step, along the lines of Merton's Dynamic Portfolio Allocation and Consumption problem) as part of the action at each time step. So the action at each time step would be a pair (c, a) where c is the quantity to consume and a is the quantity to allocate to the risky asset. Note that the consumption is constrained to be non-negative and at most the amount of wealth at any time step (a is unconstrained). The reward at each time step is the Utility of Consumption.

1.6 Actor-Critic and Variance Reduction

As we've mentioned in the previous section, REINFORCE has high variance since it's a Monte-Carlo method. So it can take quite long for REINFORCE to converge. A simple way to reduce the variance is to use a function approximation for the Q-Value Function instead of using the trace experience return as an unbiased sample of the Q-Value Function. Variance reduction happens from the simple fact that a function approximation of the Q-Value Function updates gradually (using gradient descent) and so, does not vary enormously like the trace experience returns would. Let us denote the function approximation of the Q-Value Function as $Q(s, a; \boldsymbol{w})$ where $\boldsymbol{w}$ denotes the parameters of the function approximation. We refer to $Q(s, a; \boldsymbol{w})$ as the *Critic* and we refer to the $\pi(s, a; \boldsymbol{\theta})$ function approximation as the *Actor*. The two function approximations $\pi(s, a; \boldsymbol{\theta})$ and $Q(s, a; \boldsymbol{w})$ collaborate to improve the policy using gradient ascent (guided by the PGT, using $Q(s, a; \boldsymbol{w})$ in place of the true Q-Value Function $Q^\pi(s, a)$). $\pi(s, a; \boldsymbol{\theta})$ is called *Actor* because it is the primary worker and $Q(s, a; \boldsymbol{w})$ is called *Critic* because it is the support worker. The intuitive way to think about this is that the Actor updates policy parameters in a direction that is suggested by the Critic.

Bear in mind though that the efficient way to use the Critic is in the spirit of GPI, i.e., we don't take $Q(s, a; \boldsymbol{w})$ for the current policy (current $\boldsymbol{\theta}$) all the way to convergence (thinking about updates of $\boldsymbol{w}$ for a given Policy as Policy Evaluation phase of GPI). Instead, we switch between Policy Evaluation (updates of $\boldsymbol{w}$) and Policy Improvement (updates of $\boldsymbol{\theta}$) quite frequently. In fact, with a bootstrapped (TD) approach, we would update both $\boldsymbol{w}$ and $\boldsymbol{\theta}$ after each atomic experience. $\boldsymbol{w}$ is updated such that a suitable loss function is minimized. This can be done using any of the usual Value Function approximation methods we have covered previously, including:

- Monte-Carlo, i.e., $\boldsymbol{w}$ updated using trace experience returns G_t.
- Temporal-Difference, i.e., $\boldsymbol{w}$ updated using TD Targets.
- TD(λ), i.e., $\boldsymbol{w}$ updated using targets based on eligibility traces.
- It could even be LSTD if we assume a linear function approximation for the critic $Q(s, a; \boldsymbol{w})$.

This method of calculating the gradient of $J(\boldsymbol{\theta})$ can be thought of as *Approximate Policy Gradient* due to the bias of the Critic $Q(s, a; \boldsymbol{w})$ (serving as an approximation of $Q^\pi(s, a)$), i.e.,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{s \in \mathcal{N}} \rho^\pi(s) \cdot \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta}) \cdot Q(s, a; \boldsymbol{w})$$

Now let's implement some code to perform Policy Gradient with the Critic updated using Temporal-Difference (again, for the simple case of single-dimensional continuous action space). In the function `actor_critic_gaussian` below, the key changes (from the code in `reinforce_gaussian`) are:

- The Q-Value function approximation parameters $\boldsymbol{w}$ are updated after each atomic experience as:

$$\Delta \boldsymbol{w} = \beta \cdot (R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

  where β is the learning rate for the Q-Value function approximation.
- The policy mean parameters $\boldsymbol{\theta}$ are updated after each atomic experience as:

$$\Delta \boldsymbol{\theta} = \alpha \cdot \gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t; A_t; \boldsymbol{\theta})) \cdot Q(S_t, A_t; \boldsymbol{w})$$

  (instead of $\alpha \cdot \gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta})) \cdot G_t$).

```
from rl.approximate_dynamic_programming import QValueFunctionApprox
from rl.approximate_dynamic_programming import NTStateDistribution

def actor_critic_gaussian(
    mdp: MarkovDecisionProcess[S, float],
    policy_mean_approx0: FunctionApprox[NonTerminal[S]],
    q_value_func_approx0: QValueFunctionApprox[S, float],
    start_states_distribution: NTStateDistribution[S],
    policy_stdev: float,
    gamma: float,
    max_episode_length: float
) -> Iterator[FunctionApprox[NonTerminal[S]]]:
    policy_mean_approx: FunctionApprox[NonTerminal[S]] = policy_mean_approx0
    yield policy_mean_approx
    q: QValueFunctionApprox[S, float] = q_value_func_approx0
    while True:
        steps: int = 0
```

```
gamma_prod: float = 1.0
state: NonTerminal[S] = start_states_distribution.sample()
action: float = Gaussian(
    mu=policy_mean_approx(state),
    sigma=policy_stdev
).sample()
while isinstance(state, NonTerminal) and steps < max_episode_length:
    next_state, reward = mdp.step(state, action).sample()
    if isinstance(next_state, NonTerminal):
        next_action: float = Gaussian(
            mu=policy_mean_approx(next_state),
            sigma=policy_stdev
        ).sample()
        q = q.update([(
            (state, action),
            reward + gamma * q((next_state, next_action))
        )])
        action = next_action
    else:
        q = q.update([((state, action), reward)])

    def obj_deriv_out(
        states: Sequence[NonTerminal[S]],
        actions: Sequence[float]
    ) -> np.ndarray:
        return (policy_mean_approx.evaluate(states) -
                np.array(actions)) / (policy_stdev * policy_stdev)

    grad: Gradient[FunctionApprox[NonTerminal[S]]] = \
        policy_mean_approx.objective_gradient(
            xy_vals_seq=[(state, action)],
            obj_deriv_out_fun=obj_deriv_out
    )
    scaled_grad: Gradient[FunctionApprox[NonTerminal[S]]] = \
        grad * gamma_prod * q((state, action))
    policy_mean_approx = \
        policy_mean_approx.update_with_gradient(scaled_grad)
    yield policy_mean_approx
    gamma_prod *= gamma
    steps += 1
    state = next_state
```

The above code is in the file rl/policy_gradient.py. We leave it to you as an exercise to implement the update of $Q(s, a; \boldsymbol{w})$ with TD(λ), i.e., with eligibility traces.

We can reduce the variance of this Actor-Critic method by subtracting a Baseline Function $B(s)$ from $Q(s, a; \boldsymbol{w})$ in the Policy Gradient estimate. This means we update the parameters $\boldsymbol{\theta}$ as:

$$\Delta\boldsymbol{\theta} = \alpha \cdot \gamma^t \cdot \nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta}) \cdot (Q(S_t, A_t; \boldsymbol{w}) - B(S_t))$$

Note that the Baseline Function $B(s)$ is only a function of state s (and not of action a). This ensures that subtracting the Baseline Function $B(s)$ does not add bias. This is because:

$$\begin{aligned}
&\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta}) \cdot B(s) \\
=&\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot B(s) \cdot \nabla_{\boldsymbol{\theta}} (\sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta})) \\
=&\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot B(s) \cdot \nabla_{\boldsymbol{\theta}} 1 \\
=&0
\end{aligned}$$

A good Baseline Function $B(s)$ is a function approximation $V(s;\boldsymbol{v})$ of the State-Value Function $V^{\pi}(s)$. So then we can rewrite the Actor-Critic Policy Gradient algorithm using an estimate of the Advantage Function, as follows:

$$A(s,a;\boldsymbol{w},\boldsymbol{v}) = Q(s,a;\boldsymbol{w}) - V(s;\boldsymbol{v})$$

With this, the approximation for $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is given by:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{s\in\mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a\in\mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s,a;\boldsymbol{\theta}) \cdot A(s,a;\boldsymbol{w},\boldsymbol{v})$$

The function `actor_critic_advantage_gaussian` in the file rl/policy_gradient.py implements this algorithm, i.e., Policy Gradient with two Critics $Q(s,a;\boldsymbol{w})$ and $V(s;\boldsymbol{v})$, each updated using Temporal-Difference (again, for the simple case of single-dimensional continuous action space). Specifically, in the code of `actor_critic_advantage_gaussian`:

- The Q-Value function approximation parameters $\boldsymbol{w}$ are updated after each atomic experience as:

$$\Delta\boldsymbol{w} = \beta_{\boldsymbol{w}} \cdot (R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

  where $\beta_{\boldsymbol{w}}$ is the learning rate for the Q-Value function approximation.
- The State-Value function approximation parameters $\boldsymbol{v}$ are updated after each atomic experience as:

$$\Delta\boldsymbol{v} = \beta_{\boldsymbol{v}} \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{v}) - V(S_t; \boldsymbol{v})) \cdot \nabla_{\boldsymbol{v}} V(S_t; \boldsymbol{v})$$

  where $\beta_{\boldsymbol{v}}$ is the learning rate for the State-Value function approximation.
- The policy mean parameters $\boldsymbol{\theta}$ are updated after each atomic experience as:

$$\Delta\boldsymbol{\theta} = \alpha \cdot \gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t; A_t; \boldsymbol{\theta})) \cdot (Q(S_t, A_t; \boldsymbol{w}) - V(S_t; \boldsymbol{v}))$$

A simpler way is to use the TD Error of the State-Value Function as an estimate of the Advantage Function. To understand this idea, let δ^{π} denote the TD Error for the *true* State-Value Function $V^{\pi}(s)$. Then,

$$\delta^{\pi} = r + \gamma \cdot V^{\pi}(s') - V^{\pi}(s)$$

Note that δ^{π} is an unbiased estimate of the Advantage function $A^{\pi}(s,a)$. This is because

$$\mathbb{E}_{\pi}[\delta^{\pi}|s,a] = \mathbb{E}_{\pi}[r + \gamma \cdot V^{\pi}(s')|s,a] - V^{\pi}(s) = Q^{\pi}(s,a) - V^{\pi}(s) = A^{\pi}(s,a)$$

So we can write Policy Gradient in terms of $\mathbb{E}_{\pi}[\delta^{\pi}|s,a]$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s\in\mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a\in\mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s,a;\boldsymbol{\theta}) \cdot \mathbb{E}_{\pi}[\delta^{\pi}|s,a]$$

In practice, we use a function approximation for the TD error, and sample:

$$\delta(s,r,s';\boldsymbol{v}) = r + \gamma \cdot V(s';\boldsymbol{v}) - V(s;\boldsymbol{v})$$

This approach requires only one set of critic parameters $\boldsymbol{v}$, and we don't have to worry about the Action-Value Function Q.

Now let's implement some code for this TD Error-based PG Algorithm (again, for the simple case of single-dimensional continuous action space). In the function `actor_critic_td_error_gaussian` below:

- The State-Value function approximation parameters $\boldsymbol{v}$ are updated after each atomic experience as:

$$\Delta \boldsymbol{v} = \alpha_{\boldsymbol{v}} \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{v}) - V(S_t; \boldsymbol{v})) \cdot \nabla_{\boldsymbol{v}} V(S_t; \boldsymbol{v})$$

  where $\alpha_{\boldsymbol{v}}$ is the learning rate for the State-Value function approximation.
- The policy mean parameters $\boldsymbol{\theta}$ are updated after each atomic experience as:

$$\Delta \boldsymbol{\theta} = \alpha_{\boldsymbol{\theta}} \cdot \gamma^t \cdot (\nabla_{\boldsymbol{\theta}} \log \pi(S_t; A_t; \boldsymbol{\theta})) \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{v}) - V(S_t; \boldsymbol{v}))$$

  where $\alpha_{\boldsymbol{\theta}}$ is the learning rate for the Policy Mean function approximation.

```
from rl.approximate_dynamic_programming import ValueFunctionApprox

def actor_critic_td_error_gaussian(
    mdp: MarkovDecisionProcess[S, float],
    policy_mean_approx0: FunctionApprox[NonTerminal[S]],
    value_func_approx0: ValueFunctionApprox[S],
    start_states_distribution: NTStateDistribution[S],
    policy_stdev: float,
    gamma: float,
    max_episode_length: float
) -> Iterator[FunctionApprox[NonTerminal[S]]]:
    policy_mean_approx: FunctionApprox[NonTerminal[S]] = policy_mean_approx0
    yield policy_mean_approx
    vf: ValueFunctionApprox[S] = value_func_approx0
    while True:
        steps: int = 0
        gamma_prod: float = 1.0
        state: NonTerminal[S] = start_states_distribution.sample()
        while isinstance(state, NonTerminal) and steps < max_episode_length:
            action: float = Gaussian(
                mu=policy_mean_approx(state),
                sigma=policy_stdev
            ).sample()
            next_state, reward = mdp.step(state, action).sample()
            if isinstance(next_state, NonTerminal):
                td_target: float = reward + gamma * vf(next_state)
            else:
                td_target = reward
            td_error: float = td_target - vf(state)
            vf = vf.update([(state, td_target)])

            def obj_deriv_out(
                states: Sequence[NonTerminal[S]],
                actions: Sequence[float]
            ) -> np.ndarray:
                return (policy_mean_approx.evaluate(states) -
                        np.array(actions)) / (policy_stdev * policy_stdev)

            grad: Gradient[FunctionApprox[NonTerminal[S]]] = \
                policy_mean_approx.objective_gradient(
                    xy_vals_seq=[(state, action)],
                    obj_deriv_out_fun=obj_deriv_out
            )
            scaled_grad: Gradient[FunctionApprox[NonTerminal[S]]] = \
                grad * gamma_prod * td_error
            policy_mean_approx = \
                policy_mean_approx.update_with_gradient(scaled_grad)
            yield policy_mean_approx
            gamma_prod *= gamma
            steps += 1
            state = next_state
```

The above code is in the file rl/policy_gradient.py.

Likewise, we can implement an Actor-Critic algorithm using Eligibility Traces (i.e., TD(λ)) for the State-Value Function Approximation and also for the Policy Mean Function Approximation. The updates after each atomic experience to parameters $\boldsymbol{v}$ of the State-Value function approximation and parameters $\boldsymbol{\theta}$ of the policy mean function approximation are given by:

$$\Delta \boldsymbol{v} = \alpha_{\boldsymbol{v}} \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{v}) - V(S_t; \boldsymbol{v})) \cdot \boldsymbol{E}_{\boldsymbol{v}}$$
$$\Delta \boldsymbol{\theta} = \alpha_{\boldsymbol{\theta}} \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{v}) - V(S_t; \boldsymbol{v})) \cdot \boldsymbol{E}_{\boldsymbol{\theta}}$$

where the Eligibility Traces $\boldsymbol{E}_{\boldsymbol{v}}$ and $\boldsymbol{E}_{\boldsymbol{\theta}}$ are updated after each atomic experience as follows:

$$\boldsymbol{E}_{\boldsymbol{v}} \leftarrow \gamma \cdot \lambda_{\boldsymbol{v}} \cdot \boldsymbol{E}_{\boldsymbol{v}} + \nabla_{\boldsymbol{v}} V(S_t; \boldsymbol{v})$$

$$\boldsymbol{E}_{\boldsymbol{\theta}} \leftarrow \gamma \cdot \lambda_{\boldsymbol{\theta}} \cdot \boldsymbol{E}_{\boldsymbol{\theta}} + \gamma^t \cdot \nabla_{\boldsymbol{\theta}} \log \pi(S_t, A_t; \boldsymbol{\theta})$$

where $\lambda_{\boldsymbol{v}}$ and $\lambda_{\boldsymbol{\theta}}$ are the TD(λ) parameters respectively for the State-Value Function Approximation and the Policy Mean Function Approximation.

We encourage you to implement in code this Actor-Critic algorithm using Eligibility Traces.

Now let's compare these methods on the `AssetAllocPG` instance we had created earlier to test REINFORCE, i.e., for time steps $T = 5$, $\mu = 13\%, \sigma = 20\%, r = 7\%$, coefficient of CARA $a = 1.0$, probability distribution of wealth at the start of each trace experience as $\mathcal{N}(1.0, 0.1)$, and constant standard deviation σ of the policy probability distribution of actions for a given state as 0.5. The `__main__` code in rl/chapter13/asset_alloc_pg.py evaluates the mean action for the start state of $(t = 0, W_0 = 1.0)$ after each episode (over 50,000 episodes) for each of the above-implemented PG algorithms' function approximation for the policy mean. It then plots the progress of the evaluated mean action for the start state over the 50,000 episodes (each point plotted as an average over a batch of 200 episodes), along with the benchmark of the optimal action for the start state from the known closed-form solution. Figure 1.1 shows the graph, validating the points we have made above on bias and variance of these algorithms.

Actor-Critic methods were developed in the late 1970s and 1980s, but not paid attention to in the 1990s. In the past two decades, there has been a revival of Actor-Critic methods. For a more detailed coverage of Actor-Critic methods, see the paper by Degris, White, Sutton (Degris, White, and Sutton 2012).

1.7 Overcoming Bias with Compatible Function Approximation

We've talked a lot about reducing variance for faster convergence of PG Algorithms. Specifically, we've talked about the following proxies for $Q^\pi(s, a)$ in the form of Actor-Critic algorithms in order to reduce variance.

- $Q(s, a; \boldsymbol{w})$
- $A(s, a; \boldsymbol{w}, \boldsymbol{v}) = Q(s, a; \boldsymbol{w}) - V(s; \boldsymbol{v})$
- $\delta(s, s', r; \boldsymbol{v}) = r + \gamma \cdot V(s'; \boldsymbol{v}) - V(s; \boldsymbol{v})$

However, each of the above proxies for $Q^\pi(s, a)$ in PG algorithms have a bias. In this section, we talk about how to overcome bias. The basis for overcoming bias is an important Theorem known as the *Compatible Function Approximation Theorem*. We state and prove this theorem, and then explain how we could use it in a PG algorithm.

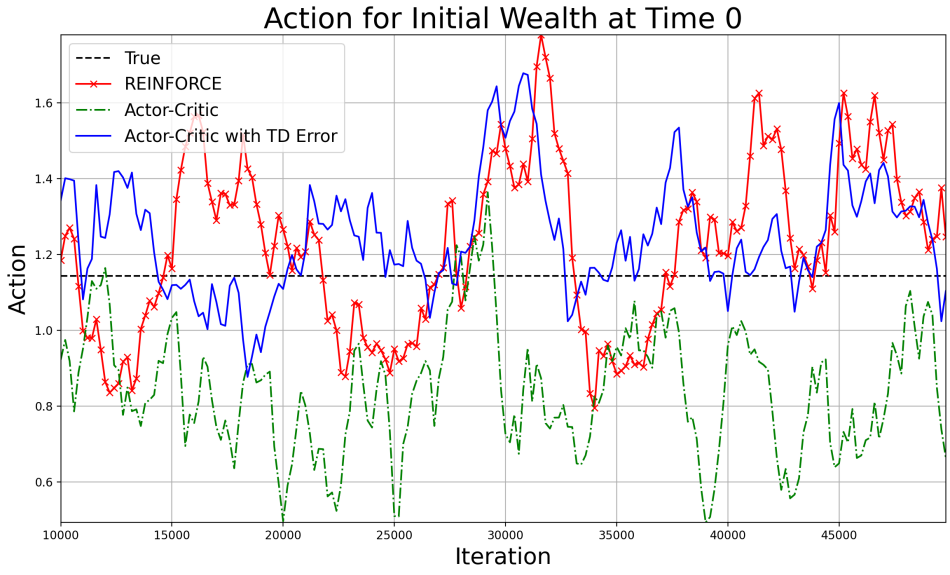

Figure 1.1: Progress of PG Algorithms

Theorem 1.7.1 (Compatible Function Approximation Theorem)**.** *Let $\boldsymbol{w}_{\boldsymbol{\theta}}^*$ denote the Critic parameters $\boldsymbol{w}$ that minimize the following mean-squared-error for given policy parameters $\boldsymbol{\theta}$:*

$$\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot (Q^{\pi}(s, a) - Q(s, a; \boldsymbol{w}))^2$$

Assume that the data type of $\boldsymbol{\theta}$ is the same as the data type of $\boldsymbol{w}$ and furthermore, assume that for any policy parameters $\boldsymbol{\theta}$, the Critic gradient at $\boldsymbol{w}_{\boldsymbol{\theta}}^*$ is compatible *with the Actor score function, i.e.,*

$$\nabla_{\boldsymbol{w}} Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*) = \nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) \text{ for all } s \in \mathcal{N}, \text{ for all } a \in \mathcal{A}$$

Then the Policy Gradient using critic $Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*)$ is exact:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta}) \cdot Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*)$$

Proof. For a given $\boldsymbol{\theta}$, since $\boldsymbol{w}_{\boldsymbol{\theta}}^*$ minimizes the mean-squared-error as defined above, we have:

$$\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot (Q^{\pi}(s, a) - Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*)) \cdot \nabla_{\boldsymbol{w}} Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*) = 0$$

But since $\nabla_{\boldsymbol{w}} Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*) = \nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta})$, we have:

$$\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot (Q^{\pi}(s, a) - Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*)) \cdot \nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) = 0$$

Therefore,

$$\begin{aligned} &\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot Q^{\pi}(s, a) \cdot \nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) \\ = &\sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^*) \cdot \nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) \end{aligned}$$

$$\text{But } \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot Q^{\pi}(s, a) \cdot \nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta})$$

$$\begin{aligned} \text{So, } \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s, a; \boldsymbol{\theta}) \cdot Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^{*}) \cdot \nabla_{\boldsymbol{\theta}} \log \pi(s, a; \boldsymbol{\theta}) \\ &= \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(s, a; \boldsymbol{\theta}) \cdot Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^{*}) \end{aligned}$$

$$\mathbb{Q}.\mathbb{E}.\mathbb{D}.$$

□

This proof originally appeared in the famous paper by Sutton, McAllester, Singh, Mansour on Policy Gradient Methods for Reinforcement Learning with Function Approximation (Sutton et al. 2001).

This means if the compatibility assumption of the Theorem is satisfied, we can use the critic function approximation $Q(s, a; \boldsymbol{w}_{\boldsymbol{\theta}}^{*})$ and still have exact Policy Gradient (i.e., no bias due to using a function approximation for the Q-Value Function). However, note that in practice, we invoke the spirit of GPI and don't take $Q(s, a; \boldsymbol{w})$ to convergence for the current $\boldsymbol{\theta}$. Rather, we update both $\boldsymbol{w}$ and $\boldsymbol{\theta}$ frequently, and this turns out to be good enough in terms of lowering the bias.

A simple way to enable Compatible Function Approximation is to make $Q(s, a; \boldsymbol{w})$ a linear function approximation, with the features of the linear function approximation equal to the Score of the policy function approximation, as follows:

$$Q(s, a; \boldsymbol{w}) = \sum_{i=1}^{m} \frac{\partial \log \pi(s, a; \boldsymbol{\theta})}{\partial \theta_i} \cdot w_i \text{ for all } s \in \mathcal{N}, \text{ for all } a \in \mathcal{A}$$

which means the feature functions $\boldsymbol{\eta}(s, a) = (\eta_1(s, a), \eta_2(s, a), \ldots, \eta_m(s, a))$ of the linear function approximation are given by:

$$\eta_i(s, a) = \frac{\partial \log \pi(s, a; \boldsymbol{\theta})}{\partial \theta_i} \text{ for all } s \in \mathcal{N}, \text{ for all } a \in \mathcal{A}, \text{ for all } i = 1, \ldots, m$$

This means the feature functions $\boldsymbol{\eta}$ is identically equal to the *Score*. Note that although here we assume $Q(s, a; \boldsymbol{w})$ to be a linear function approximation, the policy function approximation $\pi(s, a; \boldsymbol{\theta})$ can be more flexible. All that is required is that $\boldsymbol{\theta}$ consists of exactly m parameters (matching the number of number of parameters m of $\boldsymbol{w}$) and that each of the partial derivatives $\frac{\partial \log \pi(s,a;\boldsymbol{\theta})}{\partial \theta_i}$ lines up with a corresponding feature $\eta_i(s, a)$ of the linear function approximation $Q(s, a; \boldsymbol{w})$. This means that as $\boldsymbol{\theta}$ updates (as a consequence of Stochastic Gradient Ascent), $\pi(s, a; \boldsymbol{\theta})$ updates, and consequently the feature functions $\boldsymbol{\eta}(s, a) = \nabla_{\theta} \log \pi(s, a; \boldsymbol{\theta})$ update. This means the feature vector $\boldsymbol{\eta}(s, a)$ is not constant for a given (s, a) pair. Rather, the feature vector $\boldsymbol{\eta}(s, a)$ for a given (s, a) pair varies in accordance with $\boldsymbol{\theta}$ varying.

If we assume the canonical function approximation for $\pi(s, a; \boldsymbol{\theta})$ for finite action spaces that we had described in Section 1.3, then:

$$\boldsymbol{\eta}(s, a) = \boldsymbol{\phi}(s, a) - \sum_{b \in \mathcal{A}} \pi(s, b; \boldsymbol{\theta}) \cdot \boldsymbol{\phi}(s, b) \text{ for all } s \in \mathcal{N} \text{ for all } a \in \mathcal{A}$$

Note the dependency of feature vector $\boldsymbol{\eta(s,a)}$ on $\boldsymbol{\theta}$.

If we assume the canonical function approximation for $\pi(s,a;\boldsymbol{\theta})$ for single-dimensional continuous action spaces that we had described in Section 1.3, then:

$$\boldsymbol{\eta}(s,a) = \frac{(a - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{\theta}) \cdot \boldsymbol{\phi}(s)}{\sigma^2} \text{ for all } s \in \mathcal{N} \text{ for all } a \in \mathcal{A}$$

Note the dependency of feature vector $\boldsymbol{\eta}(s,a)$ on $\boldsymbol{\theta}$.

We note that any compatible linear function approximation $Q(s,a;\boldsymbol{w})$ serves as an approximation of the advantage function because:

$$\sum_{a \in \mathcal{A}} \pi(s,a;\boldsymbol{\theta}) \cdot Q(s,a;\boldsymbol{w}) = \sum_{a \in \mathcal{A}} \pi(s,a;\boldsymbol{\theta}) \cdot (\sum_{i=1}^{m} \frac{\partial \log \pi(s,a;\boldsymbol{\theta})}{\partial \theta_i} \cdot w_i)$$

$$= \sum_{a \in \mathcal{A}} (\sum_{i=1}^{m} \frac{\partial \pi(s,a;\boldsymbol{\theta})}{\partial \theta_i} \cdot w_i) = \sum_{i=1}^{m} (\sum_{a \in \mathcal{A}} \frac{\partial \pi(s,a;\boldsymbol{\theta})}{\partial \theta_i}) \cdot w_i$$

$$= \sum_{i=1}^{m} \frac{\partial}{\partial \theta_i} (\sum_{a \in \mathcal{A}} \pi(s,a;\boldsymbol{\theta})) \cdot w_i = \sum_{i=1}^{m} \frac{\partial 1}{\partial \theta_i} \cdot w_i = 0$$

Denoting $\nabla_{\boldsymbol{\theta}} \log \pi(s,a;\boldsymbol{\theta})$ as the score column vector $\boldsymbol{SC}(s,a;\boldsymbol{\theta})$ and assuming compatible linear-approximation critic:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s,a;\boldsymbol{\theta}) \cdot \boldsymbol{SC}(s,a;\boldsymbol{\theta}) \cdot (\boldsymbol{SC}(s,a;\boldsymbol{\theta})^T \cdot \boldsymbol{w_{\theta}^*}) \\
&= \sum_{s \in \mathcal{N}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(s,a;\boldsymbol{\theta}) \cdot (\boldsymbol{SC}(s,a;\boldsymbol{\theta}) \cdot \boldsymbol{SC}(s,a;\boldsymbol{\theta})^T) \cdot \boldsymbol{w_{\theta}^*} \\
&= \mathbb{E}_{s \sim \rho^{\pi}, a \sim \pi}[\boldsymbol{SC}(s,a;\boldsymbol{\theta}) \cdot \boldsymbol{SC}(s,a;\boldsymbol{\theta})^T] \cdot \boldsymbol{w_{\theta}^*}
\end{aligned}$$

Note that $\mathbb{E}_{s \sim \rho^{\pi}, a \sim \pi}[\boldsymbol{SC}(s,a;\boldsymbol{\theta}) \cdot \boldsymbol{SC}(s,a;\boldsymbol{\theta})^T]$ is the Fisher Information Matrix $\boldsymbol{FIM}_{\rho^{\pi},\pi}(\boldsymbol{\theta})$ with respect to $s \sim \rho^{\pi}, a \sim \pi$. Therefore, we can write $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ more succinctly as:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{FIM}_{\rho^{\pi},\pi}(\boldsymbol{\theta}) \cdot \boldsymbol{w_{\theta}^*} \tag{1.1}$$

Thus, we can update $\boldsymbol{\theta}$ after each atomic experience at time step t by calculating the gradient of $J(\boldsymbol{\theta})$ for the atomic experience as the outer product of $\boldsymbol{SC}(S_t, A_t; \boldsymbol{\theta})$ with itself (which gives a $m \times m$ matrix), then multiply this matrix with the vector $\boldsymbol{w}$, and then scale by γ^t, i.e.

$$\Delta \boldsymbol{\theta} = \alpha_{\boldsymbol{\theta}} \cdot \gamma^t \cdot \boldsymbol{SC}(S_t, A_t; \boldsymbol{w}) \cdot \boldsymbol{SC}(S_t, A_t; \boldsymbol{w})^T \cdot \boldsymbol{w}$$

The update for $\boldsymbol{w}$ after each atomic experience is the usual Q-Value Function Approximation update with Q-Value loss function gradient for the atomic experience calculated as:

$$\Delta \boldsymbol{w} = \alpha_{\boldsymbol{w}} \cdot (R_{t+1} + \gamma \cdot \boldsymbol{SC}(S_{t+1}, A_{t+1}; \boldsymbol{\theta})^T \cdot \boldsymbol{w} - \boldsymbol{SC}(S_t, A_t; \boldsymbol{\theta})^T \cdot \boldsymbol{w}) \cdot \boldsymbol{SC}(S_t, A_t; \boldsymbol{\theta})$$

This completes our coverage of the basic Policy Gradient Methods. Next, we cover a couple of special Policy Gradient Methods that have worked well in practice - Natural Policy Gradient and Deterministic Policy Gradient.

1.8 Policy Gradient Methods in Practice

1.8.1 Natural Policy Gradient

Natural Policy Gradient (abbreviated NPG) is due to a paper by Kakade (Kakade 2001) that utilizes the idea of Natural Gradient first introduced by Amari (Amari 1998). We won't cover the theory of Natural Gradient in detail here, and refer you to the above two papers instead. Here we give a high-level overview of the concepts, and describe the algorithm.

The core motivation for Natural Gradient is that when the parameters space has a certain underlying structure (as is the case with the parameters space of $\boldsymbol{\theta}$ in the context of maximizing $J(\boldsymbol{\theta})$), the usual gradient does not represent it's steepest descent direction, but the Natural Gradient does. The steepest descent direction of an arbitrary function $f(\boldsymbol{\theta})$ to be minimized is defined as the vector $\Delta\boldsymbol{\theta}$ that minimizes $f(\boldsymbol{\theta} + \Delta\boldsymbol{\theta})$ under the constraint that the length $|\Delta\boldsymbol{\theta}|$ is a constant. In general, the length $|\Delta\boldsymbol{\theta}|$ is defined with respect to some positive-definite matrix $\boldsymbol{G}(\boldsymbol{\theta})$ governed by the underlying structure of the $\boldsymbol{\theta}$ parameters space, i.e.,

$$|\Delta\boldsymbol{\theta}|^2 = (\Delta\boldsymbol{\theta})^T \cdot \boldsymbol{G}(\boldsymbol{\theta}) \cdot \Delta\boldsymbol{\theta}$$

We can show that under the length metric defined by the matrix $\boldsymbol{G}$, the steepest descent direction is:

$$\nabla^{nat}_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \boldsymbol{G}^{-1}(\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

We refer to this steepest descent direction $\nabla^{nat}_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$ as the Natural Gradient. We can update the parameters $\boldsymbol{\theta}$ in this Natural Gradient direction in order to achieve steepest descent (according to the matrix $\boldsymbol{G}$), as follows:

$$\Delta\boldsymbol{\theta} = -\alpha \cdot \nabla^{nat}_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

Amari showed that for a supervised learning problem of estimating the conditional probability distribution of $y|x$ with a function approximation (i.e., where the loss function is defined as the KL divergence between the data distribution and the model distribution), the matrix $\boldsymbol{G}$ is the Fisher Information Matrix for $y|x$.

Kakade specialized this idea of Natural Gradient to the case of Policy Gradient (naming it Natural Policy Gradient) with the objective function $f(\boldsymbol{\theta})$ equal to the negative of the Expected Returns $J(\boldsymbol{\theta})$. This gives the Natural Policy Gradient $\nabla^{nat}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ defined as:

$$\nabla^{nat}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{FIM}^{-1}_{\rho^{\pi},\pi}(\boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

where $\boldsymbol{FIM}_{\rho_{\pi},\pi}$ denotes the Fisher Information Matrix with respect to $s \sim \rho^{\pi}, a \sim \pi$.

We've noted in the previous section that if we enable Compatible Function Approximation with a linear function approximation for $Q^{\pi}(s,a)$, then we have Equation (1.1), i.e.,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{FIM}_{\rho^{\pi},\pi}(\boldsymbol{\theta}) \cdot \boldsymbol{w}^*_{\boldsymbol{\theta}}$$

This means:

$$\nabla^{nat}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{w}^*_{\boldsymbol{\theta}}$$

This compact result enables a simple algorithm for Natural Policy Gradient (NPG):

- After each atomic experience, update Critic parameters $\boldsymbol{w}$ with the critic loss gradient as:

$$\Delta \boldsymbol{w} = \alpha_{\boldsymbol{w}} \cdot (R_{t+1} + \gamma \cdot \boldsymbol{SC}(S_{t+1}, A_{t+1}; \boldsymbol{\theta})^T \cdot \boldsymbol{w} - \boldsymbol{SC}(S_t, A_t; \boldsymbol{\theta})^T \cdot \boldsymbol{w}) \cdot \boldsymbol{SC}(S_t, A_t; \boldsymbol{\theta})$$

- After each atomic experience, update Actor parameters $\boldsymbol{\theta}$ in the direction of $\boldsymbol{w}$:

$$\Delta \boldsymbol{\theta} = \alpha_{\boldsymbol{\theta}} \cdot \boldsymbol{w}$$

1.8.2 Deterministic Policy Gradient

Deterministic Policy Gradient (abbreviated DPG) is a creative adaptation of Policy Gradient wherein instead of a parameterized function approximation for a stochastic policy, we have a parameterized function approximation for a deterministic policy for the case of continuous action spaces. DPG is due to a paper by Silver, Lever, Heess, Degris, Wiestra, Riedmiller (Silver et al. 2014). DPG is expressed in terms of the Expected Gradient of the Q-Value Function and can be estimated much more efficiently than the usual (stochastic) PG. (Stochastic) PG integrates over both the state and action spaces, whereas DPG integrates over only the state space. As a result, computing (stochastic) PG would require more samples if the action space has many dimensions.

In Actor-Critic DPG, the Actor is the function approximation for the deterministic policy and the Critic is the function approximation for the Q-Value Function. The paper by Silver et al. provides a Compatible Function Approximation Theorem for DPG to overcome Critic approximation bias. The paper also shows that DPG is the limiting case of (Stochastic) PG, as policy variance tends to 0. This means the usual machinery of PG (such as Actor-Critic, Compatible Function Approximation, Natural Policy Gradient etc.) is also applicable to DPG.

We use the notation $a = \pi_D(s; \boldsymbol{\theta})$ to represent (a potentially multi-dimensional) continuous-valued action a equal to the value of a deterministic policy function approximation π_D (parameterized by $\boldsymbol{\theta}$), when evaluated for a state s.

The core idea of DPG can be understood intuitively by orienting on the basics of GPI and specifically, on Policy Improvement in GPI. For continuous action spaces, greedy policy improvement (with an argmax over actions, for each state) is problematic. So a simple and attractive alternative is to move the policy in the direction of the gradient of the Q-Value Function (rather than globally maximizing the Q-Value Function, at each step). Specifically, for each state s that is encountered, the policy approximation parameters $\boldsymbol{\theta}$ are updated in proportion to $\nabla_{\boldsymbol{\theta}} Q(s, \pi_D(s; \boldsymbol{\theta}))$. Note that the direction of policy improvement is different for each state, and so the average direction of policy improvements is given by:

$$\mathbb{E}_{s \sim \rho^{\mu}}[\nabla_{\boldsymbol{\theta}} Q(s, \pi_D(s; \boldsymbol{\theta}))]$$

where ρ^{μ} is the same Discounted-Aggregate State-Visitation Measure we had defined for PG (now for exploratory behavior policy μ).

Using chain-rule, the above expression can be written as:

$$\mathbb{E}_{s \sim \rho^{\mu}}[\nabla_{\boldsymbol{\theta}} \pi_D(s; \boldsymbol{\theta}) \cdot \nabla_a Q^{\pi_D}(s, a)\Big|_{a=\pi_D(s;\boldsymbol{\theta})}]$$

Note that $\nabla_{\boldsymbol{\theta}} \pi_D(s; \boldsymbol{\theta})$ is a Jacobian matrix as it takes the partial derivatives of a potentially multi-dimensional action $a = \pi_D(s; \boldsymbol{\theta})$ with respect to each parameter in $\boldsymbol{\theta}$. As we've pointed out during the coverage of (stochastic) PG, when $\boldsymbol{\theta}$ changes, policy π_D changes,

which changes the state distribution ρ^{π_D}. So it's not clear that this calculation indeed guarantees improvement - it doesn't take into account the effect of changing $\boldsymbol{\theta}$ on ρ^{π_D}. However, as was the case in PGT, Deterministic Policy Gradient Theorem (abbreviated DPGT) ensures that there is no need to compute the gradient of ρ^{π_D} with respect to $\boldsymbol{\theta}$, and that the update described above indeed follows the gradient of the Expected Return objective function. We formalize this now by stating the DPGT.

Analogous to the Expected Returns Objective defined for (stochastic) PG, we define the Expected Returns Objective $J(\boldsymbol{\theta})$ for DPG as:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi_D}[\sum_{t=0}^{\infty} \gamma^t \cdot R_{t+1}] \\ &= \sum_{s \in \mathcal{N}} \rho^{\pi_D}(s) \cdot \mathcal{R}_s^{\pi_D(s;\boldsymbol{\theta})} \\ &= \mathbb{E}_{s \sim \rho^{\pi_D}}[\mathcal{R}_s^{\pi_D(s;\boldsymbol{\theta})}] \end{aligned}$$

where

$$\rho^{\pi_D}(s) = \sum_{S_0 \in \mathcal{N}} \sum_{t=0}^{\infty} \gamma^t \cdot p_0(S_0) \cdot p(S_0 \to s, t, \pi_D)$$

is the Discounted-Aggregate State-Visitation Measure when following deterministic policy $\pi_D(s; \boldsymbol{\theta})$.

With a derivation similar to the proof of the PGT, we have the DPGT, as follows:

Theorem 1.8.1 (Deterministic Policy Gradient Theorem)**.** *Given an MDP with action space $\mathbb{R}^k$, with appropriate gradient existence conditions,*

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{s \in \mathcal{N}} \rho^{\pi_D}(s) \cdot \nabla_{\boldsymbol{\theta}} \pi_D(s; \boldsymbol{\theta}) \cdot \nabla_a Q^{\pi_D}(s, a)\Big|_{a=\pi_D(s;\boldsymbol{\theta})} \\ &= \mathbb{E}_{s \sim \rho^{\pi_D}}[\nabla_{\boldsymbol{\theta}} \pi_D(s; \boldsymbol{\theta}) \cdot \nabla_a Q^{\pi_D}(s, a)\Big|_{a=\pi_D(s;\boldsymbol{\theta})}] \end{aligned}$$

In practice, we use an Actor-Critic algorithm with a function approximation $Q(s, a; \boldsymbol{w})$ for the Q-Value Function as the Critic. Since the policy approximated is Deterministic, we need to address the issue of exploration - this is typically done with Off-Policy Control wherein we employ an exploratory (stochastic) behavior policy, while the policy being approximated (and learnt with DPG) is the target (deterministic) policy. The Expected Return Objective is a bit different in the case of Off-Policy - it is the Expected Q-Value for the target policy under state-occurrence probabilities while following the behavior policy, and the Off-Policy Deterministic Policy Gradient is an approximate (not exact) formula. We avoid importance sampling in the Actor because DPG doesn't involve an integral over actions, and we avoid importance sampling in the Critic by employing Q-Learning. As a result, for Off-Policy Actor-Critic DPG, we update the Critic parameters $\boldsymbol{w}$ and the Actor parameters $\boldsymbol{\theta}$ after each atomic experience in a trace experience generated by the behavior policy.

$$\Delta \boldsymbol{w} = \alpha_{\boldsymbol{w}} \cdot (R_{t+1} + \gamma \cdot Q(S_{t+1}, \pi_D(S_{t+1}; \boldsymbol{\theta}); \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

$$\Delta\boldsymbol{\theta} = \alpha_{\boldsymbol{\theta}} \cdot \nabla_{\boldsymbol{\theta}} \pi_D(S_t; \boldsymbol{\theta}) \cdot \nabla_a Q(S_t, a; \boldsymbol{w})\Big|_{a=\pi_D(S_t;\boldsymbol{\theta})}$$

Critic Bias can be resolved with a Compatible Function Approximation Theorem for DPG (see Silver et al. paper for details). Instabilities caused by Bootstrapped Off-Policy Learning with Function Approximation can be resolved with Gradient Temporal Difference (GTD).

1.9 Evolutionary Strategies

We conclude this chapter with a section on Evolutionary Strategies - a class of algorithms to solve MDP Control problems. We want to highlight right upfront that Evolutionary Strategies are technically not RL algorithms (for reasons we shall illuminate once we explain the technique of Evolutionary Strategies). However, Evolutionary Strategies can sometimes be quite effective in solving MDP Control problems and so, we give them appropriate coverage as part of a wide range of approaches to solve MDP Control. We cover them in this chapter because of their superficial resemblance to Policy Gradient Algorithms (again, they are not RL algorithms and hence, not Policy Gradient algorithms).

Evolutionary Strategies (abbreviated as ES) actually refers to a technique/approach that is best understood as a type of Black-Box Optimization. It was popularized in the 1970s as *Heuristic Search Methods*. It is loosely inspired by natural evolution of living beings. We focus on a subclass of ES known as Natural Evolutionary Strategies (abbreviated as NES).

The original setting for this approach was quite generic and not at all specific to solving MDPs. Let us understand this generic setting first. Given an objective function $F(\boldsymbol{\psi})$, where $\boldsymbol{\psi}$ refers to parameters, we consider a probability distribution $p_{\boldsymbol{\theta}}(\boldsymbol{\psi})$ over $\boldsymbol{\psi}$, where $\boldsymbol{\theta}$ refers to the parameters of the probability distribution. The goal in this generic setting is to maximize the average objective $\mathbb{E}_{\boldsymbol{\psi}\sim p_{\boldsymbol{\theta}}}[F(\boldsymbol{\psi})]$.

We search for optimal θ with stochastic gradient ascent as follows:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}(\mathbb{E}_{\boldsymbol{\psi}\sim p_{\boldsymbol{\theta}}}[F(\boldsymbol{\psi})]) &= \nabla_{\boldsymbol{\theta}}(\int_{\boldsymbol{\psi}} p_{\boldsymbol{\theta}}(\boldsymbol{\psi}) \cdot F(\boldsymbol{\psi}) \cdot d\boldsymbol{\psi}) \\
&= \int_{\boldsymbol{\psi}} \nabla_{\boldsymbol{\theta}}(p_{\boldsymbol{\theta}}(\boldsymbol{\psi})) \cdot F(\boldsymbol{\psi}) \cdot d\boldsymbol{\psi} \\
&= \int_{\boldsymbol{\psi}} p_{\boldsymbol{\theta}}(\boldsymbol{\psi}) \cdot \nabla_{\boldsymbol{\theta}}(\log p_{\boldsymbol{\theta}}(\boldsymbol{\psi})) \cdot F(\boldsymbol{\psi}) \cdot d\boldsymbol{\psi} \\
&= \mathbb{E}_{\boldsymbol{\psi}\sim p_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}}(\log p_{\boldsymbol{\theta}}(\boldsymbol{\psi})) \cdot F(\boldsymbol{\psi})] \qquad (1.2)
\end{aligned}$$

Now let's see how NES can be applied to solving MDP Control. We set $F(\cdot)$ to be the (stochastic) *Return* of an MDP. $\boldsymbol{\psi}$ corresponds to the parameters of a deterministic policy $\pi_{\boldsymbol{\psi}} : \mathcal{N} \to \mathcal{A}$. $\boldsymbol{\psi} \in \mathbb{R}^m$ is drawn from an isotropic m-variate Gaussian distribution, i.e., Gaussian with mean vector $\boldsymbol{\theta} \in \mathbb{R}^m$ and fixed diagonal covariance matrix $\sigma^2 \boldsymbol{I_m}$ where $\sigma \in \mathbb{R}$ is kept fixed and $\boldsymbol{I_m}$ is the $m \times m$ identity matrix. The average objective (*Expected Return*) can then be written as:

$$\mathbb{E}_{\boldsymbol{\psi}\sim p_{\boldsymbol{\theta}}}[F(\boldsymbol{\psi})] = \mathbb{E}_{\boldsymbol{\epsilon}\sim\mathcal{N}(0,\boldsymbol{I_m})}[F(\boldsymbol{\theta} + \sigma \cdot \boldsymbol{\epsilon})]$$

where $\boldsymbol{\epsilon} \in \mathbb{R}^m$ is the standard normal random variable generating $\boldsymbol{\psi}$. Hence, from Equation (1.2), the gradient ($\nabla_{\boldsymbol{\theta}}$) of *Expected Return* can be written as:

$$\mathbb{E}_{\boldsymbol{\psi}\sim p_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}}(\log p_{\boldsymbol{\theta}}(\boldsymbol{\psi})) \cdot F(\boldsymbol{\psi})]$$

$$= \mathbb{E}_{\boldsymbol{\psi} \sim \mathcal{N}(\boldsymbol{\theta}, \sigma^2 \boldsymbol{I_m})}[\nabla_{\boldsymbol{\theta}}(\frac{-(\boldsymbol{\psi} - \boldsymbol{\theta})^T \cdot (\boldsymbol{\psi} - \boldsymbol{\theta})}{2\sigma^2}) \cdot F(\boldsymbol{\psi})]$$

$$= \frac{1}{\sigma} \cdot \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I_m})}[\boldsymbol{\epsilon} \cdot F(\boldsymbol{\theta} + \sigma \cdot \boldsymbol{\epsilon})]$$

Now we come up with a sampling-based algorithm to solve the MDP. The above formula helps estimate the gradient of *Expected Return* by sampling several $\boldsymbol{\epsilon}$ (each $\boldsymbol{\epsilon}$ represents a *Policy* $\pi_{\boldsymbol{\theta}+\sigma\cdot\boldsymbol{\epsilon}}$), and averaging $\boldsymbol{\epsilon} \cdot F(\boldsymbol{\theta} + \sigma \cdot \boldsymbol{\epsilon})$ across a large set (n) of $\boldsymbol{\epsilon}$ samples.

Note that evaluating $F(\boldsymbol{\theta} + \sigma \cdot \boldsymbol{\epsilon})$ involves playing an episode for a given sampled $\boldsymbol{\epsilon}$, and obtaining that episode's *Return* $F(\boldsymbol{\theta} + \sigma \cdot \boldsymbol{\epsilon})$. Hence, we have n values of $\boldsymbol{\epsilon}$, n *Policies* $\pi_{\boldsymbol{\theta}+\sigma\cdot\boldsymbol{\epsilon}}$, and n *Returns* $F(\boldsymbol{\theta} + \sigma \cdot \boldsymbol{\epsilon})$.

Given the gradient estimate, we update $\boldsymbol{\theta}$ in this gradient direction, which in turn leads to new samples of $\boldsymbol{\epsilon}$ (new set of *Policies* $\pi_{\boldsymbol{\theta}+\sigma\cdot\boldsymbol{\epsilon}}$), and the process repeats until $\mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I_m})}[F(\boldsymbol{\theta} + \sigma \cdot \boldsymbol{\epsilon})]$ is maximized.

The key inputs to the algorithm are:

- Learning rate (SGD Step Size) α
- Standard Deviation σ
- Initial value of parameter vector $\boldsymbol{\theta_0}$

With these inputs, for each iteration $t = 0, 1, 2, \ldots$, the algorithm performs the following steps:

- Sample $\boldsymbol{\epsilon_1}, \boldsymbol{\epsilon_2}, \ldots \boldsymbol{\epsilon_n} \sim \mathcal{N}(0, \boldsymbol{I_m})$.
- Compute Returns $F_i \leftarrow F(\boldsymbol{\theta_t} + \sigma \cdot \boldsymbol{\epsilon_i})$ for $i = 1, 2, \ldots, n$.
- $\boldsymbol{\theta_{t+1}} \leftarrow \boldsymbol{\theta_t} + \frac{\alpha}{n\sigma} \sum_{i=1}^{n} \boldsymbol{\epsilon_i} \cdot F_i$

On the surface, this NES algorithm looks like PG because it's not Value Function-based (it's Policy-based, like PG). Also, similar to PG, it uses a gradient to move the policy towards optimality. But, ES does not interact with the environment (like PG/RL does). ES operates at a high-level, ignoring the (state, action, reward) interplay. Specifically, it does not aim to assign credit to actions in specific states. Hence, ES doesn't have the core essence of RL: *Estimating the Q-Value Function for a Policy and using it to Improve the Policy*. Therefore, we don't classify ES as Reinforcement Learning. Rather, we consider ES to be an alternative approach to RL Algorithms.

What is the effectiveness of ES compared to RL? The traditional view has been that ES won't work on high-dimensional problems. Specifically, ES has been shown to be data-inefficient relative to RL. This is because ES resembles simple hill-climbing based only on finite differences along a few random directions at each step. However, ES is very simple to implement (no Value Function approximation or back-propagation needed), and is highly parallelizable. ES has the benefits of being indifferent to distribution of rewards and to action frequency, and is tolerant of long horizons. A paper from OpenAI Research (Salimans et al. 2017) shows techniques to make NES more robust and more data-efficient, and they demonstrate that NES has more exploratory behavior than advanced PG algorithms.

1.10 Key Takeaways from this Chapter

- Policy Gradient Algorithms are based on GPI with Policy Improvement as a Stochastic Gradient Ascent for "Expected Returns" Objective $J(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ are the parameters of the function approximation for the Policy.

- Policy Gradient Theorem gives us a simple formula for $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ in terms of the score of the policy function approximation (i.e., gradient of the log of the policy with respect to the policy parameters $\boldsymbol{\theta}$).
- We can reduce variance in PG algorithms by using a critic and by using an estimate of the advantage function for the Q-Value Function.
- Compatible Function Approximation Theorem enables us to overcome bias in PG Algorithms.
- Natural Policy Gradient and Deterministic Policy Gradient are specialized PG algorithms that have worked well in practice.
- Evolutionary Strategies are technically not RL, but they resemble PG Algorithms and can sometimes be quite effective in solving MDP Control problems.

Bibliography

Amari, S. 1998. "Natural Gradient Works Efficiently in Learning." *Neural Computation* 10 (2): 251–76.

Degris, Thomas, Martha White, and Richard S. Sutton. 2012. "Off-Policy Actor-Critic." *CoRR* abs/1205.4839. http://arxiv.org/abs/1205.4839.

Kakade, Sham M. 2001. "A Natural Policy Gradient." In *NIPS*, edited by Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani, 1531–38. MIT Press. http://dblp.uni-trier.de/db/conf/nips/nips2001.html#Kakade01.

Salimans, Tim, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning." *arXiv Preprint arXiv:1703.03864*.

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. 2014. "Deterministic Policy Gradient Algorithms." In *ICML*, 32:387–95. JMLR Workshop and Conference Proceedings. JMLR.org. http://dblp.uni-trier.de/db/conf/icml/icml2014.html#SilverLHDWR14.

Sutton, R., D. Mcallester, S. Singh, and Y. Mansour. 2001. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." MIT Press.

Williams, R. J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8: 229–56.