

# Deep Reinforcement Learning and Function Optimization

Hanan Ather<sup>‡</sup>

Department of Mathematics and Statistics

Summer, 2023  
University of Ottawa

---

\*Statistics Canada, Modern Statistical Methods and Data Science Division

<sup>†</sup>University of Ottawa, Department of Mathematics and Statistics

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	What is Reinforcement Learning? . . . . .	4
1.2	Connections to Machine Learning . . . . .	4
1.3	History of Deep Reinforcement Learning . . . . .	5
1.4	Structure of Reinforcement Learning Algorithms . . . . .	5
1.5	Model-based and Model-free learning . . . . .	6
1.6	On-policy and Off-Policy Methods . . . . .	7
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Markov Decision Process . . . . .	7
2.2	Episodic Reinforcement Learning . . . . .	9
2.3	Policy and Return . . . . .	9
2.4	Goal of Reinforcement Learning . . . . .	10
2.5	Value and State-Value Functions . . . . .	11
2.6	The Bellman Equations . . . . .	12
2.7	Finding Optimal Policies . . . . .	13
<b>3</b>	<b>Tabular Algorithms</b>	<b>14</b>
3.1	Dynamic Programing . . . . .	14
3.2	Monte Carlo Learning . . . . .	15
3.3	Temporal difference learning . . . . .	16
3.4	Generalized Policy Iteration (GPI) . . . . .	16
3.5	Why $Q$ -function for Control? . . . . .	17
<b>4</b>	<b>Deep Reinforcement Learning</b>	<b>18</b>
4.1	Value Function Approximation . . . . .	18
4.2	State-Value Function Approximation . . . . .	18
<b>5</b>	<b>Deep Q-Networks (DQN)</b>	<b>19</b>
5.1	Why Neural Networks? . . . . .	19
5.2	Q-Learning via Function Approximation . . . . .	20
5.3	Experience Replay . . . . .	20
5.4	DQNs: Fixed Q-Targets . . . . .	21
<b>6</b>	<b>Policy Gradient Methods</b>	<b>22</b>
6.1	Policy Gradient Theorem . . . . .	23
6.2	Score Function Estimator . . . . .	23
6.3	Deriving Policy Gradient . . . . .	25
6.4	Monte Carlo Policy Gradient . . . . .	27
6.5	Baselines . . . . .	28
6.6	Generalizing Policy Gradients . . . . .	29
<b>7</b>	<b>Advantage Actor-Critic</b>	<b>29</b>
7.1	The Advantage Function . . . . .	31
7.2	Estimating Advantage . . . . .	31
7.3	Generalized Advantage Estimation . . . . .	32
<b>8</b>	<b>Advanced Policy Gradient Methods</b>	<b>33</b>
8.1	Performance Collapse and Surrogate Objective . . . . .	33
8.2	Monotonic Improvement Theory . . . . .	34
8.3	Trust Region Policy Optimization Problem . . . . .	35
8.4	Proximal Policy Optimization . . . . .	36
8.4.1	Adaptive KL Penalty Algorithm . . . . .	36
8.4.2	PPO with Clipped Surrogate Objectives . . . . .	36

<b>9 Experiments</b>	<b>37</b>
9.1 Optimization as Reinforcement Learning . . . . .	37
9.2 Optimization of single-variable continuous functions . . . . .	38
9.2.1 Direct Jump Strategy in Optimization . . . . .	40
9.3 Increasing Complexity of State Representation . . . . .	41
9.4 Reinforcement Learning for Linear Regression Optimization . . . . .	43
9.5 Generalization . . . . .	44
9.6 Multi-Task Learning for Optimizing Regression . . . . .	44
9.7 Adaptive Learning Rate Optimization in Gradient Descent . . . . .	45
9.7.1 Adaptive Learning Rate Methods . . . . .	46
9.7.2 Reinforcement Learning for Adaptive Learning Rate . . . . .	47
9.7.3 Results and Observations . . . . .	47
9.8 Future Directions . . . . .	48

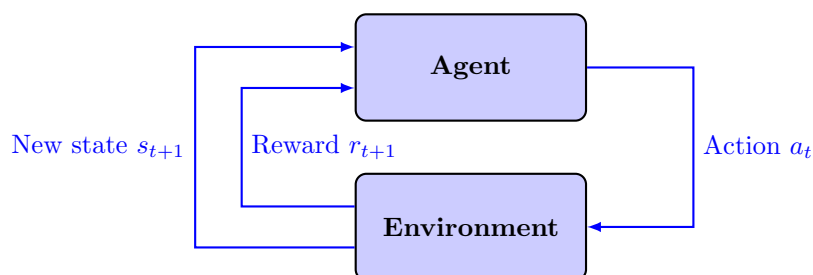


Figure 1: The MDP Framework: As illustrated in the figure, the agent, which represents an AI algorithm, and the environment, an abstract entity delivering uncertain outcomes, engage in a time-sequenced, iterative interaction. Credit: Adapted from Sutton & Barto, 2018 [1]

## §1 Introduction

### §1.1 What is Reinforcement Learning?

Reinforcement Learning (RL) is a subfield of artificial intelligence that centers around an **agent** learning to make optimal decisions by interacting with its **environment**, receiving feedback in the form of **rewards** or penalties, and adjusting its behavior accordingly to maximize cumulative rewards over time [1]. The agent is an entity that observes the environment, takes actions based on these observations, and learns from the results to improve its decisions over time. The environment is the dynamic context or ‘world’ within which the agent operates. At each time step, the agent perceives a possibly partial or noisy state of the environment, referred to as an **observation**. Based on its (possibly partial) observation of the state of the world, the agent decides an *action* to take. The environment changes when the agent acts on it. However, the environment may also evolve independently. Based on the action it took, the agent acquires a scalar *reward*, signaling the desirability of its current state. The primary objective of the agent is to maximize this cumulative reward, known as the **return**. Reinforcement learning methods equip the agent with strategies to learn behaviours that contribute towards achieving this goal.

### §1.2 Connections to Machine Learning

At a high level, machine learning is about using algorithms to learn from data, and then making predictions or decisions about new unseen data by extrapolating learned insights. Most of modern machine learning focuses on learning *functions* from data. In essence, a function takes an input (or set of inputs), performs some operations, and produces an output. In machine learning, this function is learned from the data such that it can predict the output (target variable) given a new input. Machine learning follows a straightforward methodology: select a versatile function approximator such as a deep neural network, identify a suitable loss function, and employ gradient descent for optimizing the network’s parameters. The overall learning problem in machine learning boils down to an optimization problem: find the parameters that minimize the loss function. By adjusting the parameters to achieve this, the neural network learns the function that best maps inputs to outputs given the data.

In reinforcement learning (RL), the reduction from learning problem to an optimization problem presents more complexities compared to supervised learning. A primary challenge lies in the fact that the function we aim to optimize - the agent’s expected total reward - isn’t entirely accessible for analytic operations. This is primarily due to its dependence on two unknown elements: the dynamics model, which predicts the future state of the system based on current state and actions, and the reward function, which defines the desirability of each state. These two components are usually unknown in RL, making the optimization process more challenging. Another challenge arises from the fact that the agent’s input data is significantly influenced by its own behavior. This dependency makes it difficult to design algorithms that guarantee consistent improvements, as changes in the agent’s behavior can affect the data it uses for learning. Further complicating the matter is the existence of multiple potential functions that one might aim to approximate,

which introduces an additional layer of complexity. We will delve into these various approximation targets in more detail in Section...

### §1.3 History of Deep Reinforcement Learning

Deep reinforcement learning combines the power of reinforcement learning with the versatility of neural networks for function approximation. The fusion of reinforcement learning and neural networks has a history stretching back to the early 1990s, when Tesauro's TD-Gammon [25], a backgammon playing AI equipped with a neural network value function, performed at a level comparable to top human players. Since then, neural networks have found consistent use in system identification and control.

Lin's 1993 thesis [26] furthered the exploration of this amalgamation, implementing various reinforcement learning algorithms in conjunction with neural networks in a robotics context.

Despite these promising beginnings, reinforcement learning with nonlinear function approximation remained relatively obscure for about two decades. The majority of reinforcement learning studies presented at leading machine learning conferences like NIPS and ICML were largely concentrated on theoretical results or toy problems that utilized linear or tabular function approximators.

A significant shift occurred in the early 2010s when deep learning started demonstrating extraordinary empirical success, particularly in fields like speech recognition [27] and computer vision [28]. Most conventional reinforcement learning methods, tailored to linear or tabular functions, fell short when it came to learning functions that require multi-step computation. In contrast, deep neural networks could effectively approximate these complex functions, with their performance in supervised learning scenarios providing evidence of the tractability of their optimization.

The field of deep reinforcement learning saw a resurgence of interest following the groundbreaking results by Mnih et al [29]. Their work showcased an AI learning to play a range of Atari games using screen images as inputs, and a variant of Q-learning for policy control. Their approach outperformed previous methods using evolutionary algorithms, despite employing a more challenging input representation.

Following this resurgence, numerous intriguing results emerged. Notably, Silver et al [30] demonstrated an AI system that could play the game of Go better than human experts, using a blend of supervised learning and several reinforcement learning steps for training deep neural networks, coupled with a tree search algorithm.

### §1.4 Structure of Reinforcement Learning Algorithms

Reinforcement Learning (RL) algorithms, while diverse in their approaches, share a common structural anatomy that allows for a broad high-level understanding of their operation. Essentially, these algorithms can be decomposed into three fundamental components, which are present, to varying degrees of complexity, in most, if not all, RL methodologies.

1. **Sample Generation:** This is the phase where the agent interacts with the environment by executing its current policy. The aim is to collect experience samples, consisting of states, actions, and rewards. These samples serve as the empirical basis for the agent's learning process. The way these samples are generated can depend on various strategies such as purely exploratory methods, exploitation of current knowledge, or a combination of both (exploration-exploitation tradeoff).
2. **Model Fitting/Return Estimation:** The agent leverages the collected samples to either construct a model of the environment or directly estimate the expected return (cumulative future reward) for various state-action pairs. These estimates are pivotal in guiding the agent's future actions. While model-based methods strive to understand the environment's dynamics, model-free methods circumvent this step by trying to directly learn the optimal policy or value function.
3. **Policy Improvement:** Based on the information gained from the previous steps, the agent refines its policy to better navigate the environment. The improvement could be a

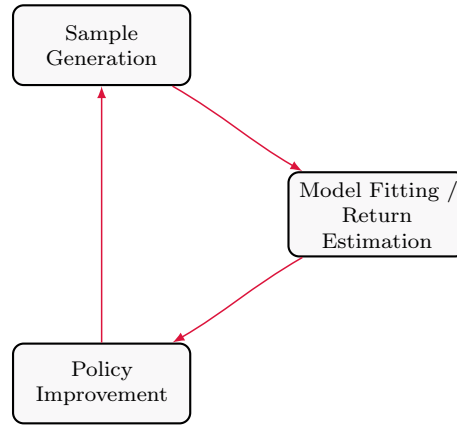


Figure 2: High-level process of RL algorithms: The algorithms start with the generation of samples through interaction with the environment (1). The samples are then used to estimate the return or fit a model (2). Based on the information gained, the policy is improved (3). This process continues in a loop until the policy converges towards an optimal policy or a predefined stopping criterion is met.

deterministic change, like in Policy Iteration, or a probabilistic one as in Policy Gradient methods. The refined policy is expected to guide the agent towards actions that maximize the cumulative reward.

These components form a loop that continues until the agent's policy converges towards an optimal policy, or a predefined stopping criterion is met. It's important to note that while these components are a simplification, the intricacies and nuances of different RL algorithms can add significant complexity to each of these steps.

## §1.5 Model-based and Model-free learning

In the literature of reinforcement learning (RL), the learning approaches can be classified primarily into two categories: **Model-based learning** and **Model-free learning** [15]. In model-based learning approaches, the agent's strategy is to learn a model of the environment before deriving a policy. The agent creates an internal representation of the environment, encapsulating its rules and dynamics. As we have seen previously, a model of an environment in RL is most commonly defined as:

$$P_{ss'}^a = \mathbb{P}(s_{t+1} = s' \mid s_t = s, a_t = a)$$

$$R_{ss'}^a = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s']$$

Where  $P_{ss'}^a$  is the transition probability from state  $s$  to  $s'$  under action  $a$ , and  $R_{ss'}^a$  is the expected immediate reward when transitioning from  $s$  to  $s'$  with action  $a$ .

Once the model is well established, the agent utilizes it as a guide to derive and refine its policy [4]. Although this approach might be more computationally demanding, it often results in a more robust and flexible policy, since the agent can exploit its comprehension of the environment to plan and adapt its actions across various situations [1].

On the other hand, model-free learning approaches involve acquiring an action policy directly, without developing a detailed understanding of the underlying dynamics of the environment. In this case, the learning agent interacts with the environment based on the rewards it receives, instead of constructing a comprehensive model of how the environment operates.

An instance of model-free methods is the Q-learning algorithm. It learns an action-value function  $Q(s, a)$ , that gives the expected utility of taking a certain action  $a$  in a certain state  $s$ , defined as [15]:

$$Q(s, a) = \mathbb{E}[r_t + \gamma Q(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a]$$

The agent uses the Q-values to make decisions, but without explicitly constructing a model

of the environment’s dynamics. This property makes model-free methods less computationally expensive compared to their model-based counterparts, albeit at the cost of having less potential to adapt to changes in the environment.

## §1.6 On-policy and Off-Policy Methods

The distinction between “on-policy” and “off-policy” methods lies in the relationship between the policy that the agent follows while exploring the environment (behaviour policy) and the policy that it learns about (target policy). The fundamental distinction lies in the relationship between the policy that the agent follows while exploring the environment (behaviour policy) and the policy it learns about (target policy).

In on-policy methods, the behaviour policy and the target policy are the same. The agent learns about and improves on the policy it uses to choose its actions. Any exploration must be part of this policy. In contrast, off-policy methods allows the agent to follow one policy (behaviour policy) while learning about a different, potentially optimal policy (target policy). The agent can explore more freely because it can take exploratory actions without directly impacting the quality of policy its learning about. “On-policy methods attempt to evaluate or improve the policy that it used to make decisions, where as off-policy methods evaluate or improve a policy different from that used to generate the data” (Sutton and Barto, 2018).

- **On-policy methods:** These are the methods where the policy that’s being improved upon is the same policy that’s used to make decisions during interaction with the environment. In other words, the agent learns about and improves the policy that it is currently following. SARSA is a common example of an on policy method. For exploration, it might use an  $\epsilon$ -greedy strategy, but the important part is that the policy it learns about ( $Q$ -values it updates) is based on the same  $\epsilon$ -greedy strategy.
- **Off-policy methods:** These are methods where the policy that’s being improved is different from the policy that’s used to make decisions during interaction with the environment. The agent follows one policy (the behaviour policy), while learning about a different one (target policy). For example, Q-learning is an off-policy method. The agent can use any exploratory strategy (like  $\epsilon$ -greedy), but the  $Q$ -values it learns are based on the assumption of a greedy policy (always choosing the action with the highest  $Q$ -value).

The benefit of off-policy methods is that they allow you to learn an optimal policy while following exploratory policy for better learning. In contrast, on-policy methods might converge faster because they are always learning about the policy they are following, but they need to balance exploration and exploitation with the policy they are learning about.

## §2 Background

### §2.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used in reinforcement learning to model decision-making in situations where an agent interacts with an environment that has probabilistic state transitions and rewards [3].

The motivation for using MDPs in reinforcement learning is to provide a structured representation of the problem, facilitating the development of algorithms that enable agents to find optimal policies for maximizing cumulative rewards over time [4].

**Definition 2.1 (Markov Decision Process)** MDP is defined by 4-tuple  $(\mathcal{S}, \mathcal{A}, \mathcal{P}_a, \mathcal{R})$ :

1. a set of states  $\mathcal{S}$ , possibly infinite
2. a set of actions  $\mathcal{A}$ , possibly infinite
3. transition probability  $\mathcal{P}_a(s', r | s, a) = \mathbb{P}[S_{t+1} = s', R_t = r | S_t = s, A_t = a]$
4. a reward  $R_{t+1} \in \mathcal{R} \subseteq \mathbb{R}$ , and corresponding probability distribution  $\mathcal{R}_a(s', s) = \mathbb{P}[r' | s, a]$  over rewards

In the realm of Reinforcement Learning, the terms **state** and **observation** bear distinct meanings. A state  $s$  is a comprehensive representation of the environment's current condition. By its very definition, it encapsulates all the requisite information about the world, leaving no detail concealed or unaccounted for. On the contrary, an **observation**  $o$  offers a more limited view. It provides a partial portrayal of the state, potentially excluding certain aspects or details. In essence, while a state offers a complete picture, an observation might only present a snippet.

When an agent possesses the capability to discern the entirety of the environment's state, we term such an environment as **fully observed**. Conversely, in situations where the agent perceives only a subset or a partial snapshot of the environment's state, we categorize the environment as **partially observed**.

★ There are instances where the symbol for states,  $s$ , is used in contexts where it would be technically more precise to use  $o$  for observations. This ambiguity particularly surfaces when discussing the agent's decision-making process. Notationally, it's often conveyed that the agent's action is dependent on the state,  $s$ . However, in a many scenarios, the agent's action depend on the observation,  $o$ , because the agent doesn't typically have complete access or knowledge of the true underlying state.

In the domain of Deep Reinforcement Learning (Deep RL), the representation of both states and observations predominantly relies on real-valued vectors or higher-order tensor. This choice of representation facilitates the handling and processing of complex data types and structures. For illustrative purposes, consider a scenario where the environment provides a visual observation; this can be aptly represented using a tensor corresponding to the RGB values of each pixel. Similarly, when we are dealing with a robotic environment, the state might encompass information about the robot's kinematics, typically encapsulated through parameters such as joint angles and velocities.

The literature offers several definitions of MDPs, particularly regarding the reward function, which is often represented as  $\mathcal{R}(s)$ ,  $\mathcal{R}(s, a)$ , or  $\mathcal{R}(s, a, s')$ . While these expressions may differ, they share equivalent expressive power. For instance, a stochastic reward function can be emulated using a deterministic reward formulation by incorporating the reward into the state description.

In addition to the state, action, and reward definitions, some problem settings also consider the **initial state distribution**, represented as  $\mu(s)$ . This distribution defines the probability that the initial state  $s_0$  is sampled from, playing a significant role in shaping the initial steps of the RL process.

In a *finite* MDP, random variables  $R_t$  and  $S_t$  possess well-defined discrete probability distributions, which depend solely on the preceding state and action. For specific values  $r \in \mathcal{R}$  and  $s' \in \mathcal{S}$ , there exists a probability of these values occurring at time  $t$ , given particular values of the preceding state and action. The transition probabilities from one time step to the next can be represented as a function  $\mathcal{P} : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ , which is an ordinary deterministic function[1]:

$$\mathcal{P}(s', r | s, a) = \mathbb{P}\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\} \quad (1)$$

Markov Decision Processes (MDPs) have been used to model a wide range of decision-making problems across various domains, some real-world examples include:

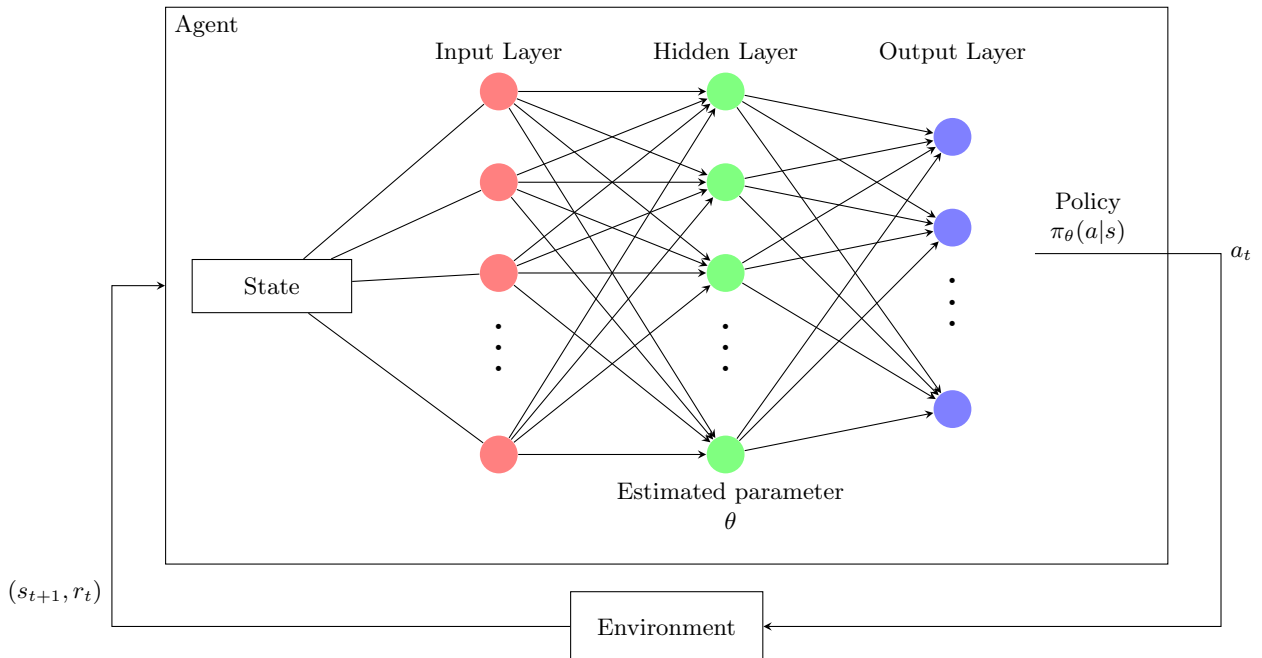
- Robotics: In robotics, MDPs are employed for navigation and path planning tasks [5]
- Finance: MDPs have been used to model and solve portfolio optimization problems [7]



- Healthcare: MDPs have been employed in healthcare for personalized treatment recommendation system [6]

## §2.2 Episodic Reinforcement Learning

In the episodic setting of reinforcement learning, the agent's experience is broken up into a series of episodes – sequences with finite number of states, actions and rewards. Episodic reinforcement learning in the fully-observed setting is defined by the following process. Each episode begins by sampling an initial state of the environment,  $s_0$ , from distribution  $\mu(s_0)$ . Each time step  $t = 0, 1, 2, \dots$ , the agent chooses an action  $a_t$ , sampled from distribution  $\pi(a_t, s_t)$ . Then the environment generates the next state and reward, according to some distribution  $\mathcal{P}(s_{t+1}, r_t | s_t, a_t)$ . The episode ends when a terminal state  $s_T$  is reached.



## §2.3 Policy and Return

In reinforcement learning algorithms, assessing the value of an agent being in a particular state is crucial. Almost all reinforcement learning algorithms involve estimates of how good it is for the agent to be in a given state [1]. However, defining “how good” a state is can be ambiguous. In RL, the value of a state is determined by anticipated future rewards, or more precisely, in terms of expected rewards.

Of course the rewards the agent can expect to receive in the future depends on the actions it will take. Accordingly, the values of states (i.e., the expected reward agent will receive from that state onwards) are defined with respect to a **policy**.

In reinforcement learning, a policy is a strategy or set of rules that guides an agent's decision-making process, determining which action to take in a given state to maximize cumulative rewards over time. Policies are just functions that take as input a state,  $S_t$ , and output a probability distribution of actions. For a given state  $s \in S$  and action  $a \in A$ , the policy is defined as [8]:

**Definition 2.2 (Policy)** A policy is a mapping  $\pi : S \rightarrow \Delta(A)$ , where  $\Delta(A)$  is the set of probability distributions over the action space  $A$ . A policy is deterministic if for any  $s \in S$ , there exists a unique  $a \in A$  such that  $\pi(a|s) = 1$ . In this case, we can identify  $\pi : S \rightarrow A$ .<sup>a</sup>

<sup>a</sup>The given definition pertains to a stationary policy, as the action distribution does not rely on time. Generally, a non-stationary policy can be characterized as a sequence of mappings  $\pi_t : S \rightarrow \Delta(A)$ , indexed by  $t$ . Notably, in scenarios with a finite horizon, employing a non-stationary policy is often essential for reward optimization

The objective in RL is to learn a policy, more precisely, the agent's objective is to find a policy that *maximizes its expected (reward) return*. The discounted return is defined as [1]:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \gamma^k \sum_{k=0}^{\infty} R_{t+k+1} \quad (2)$$

Where  $\gamma \in [0, 1]$  is the *discount-factor*<sup>1</sup>. The return is a scalar-value that essentially summarizes a (possibly) infinite sequence of immediate rewards. An important fact to note is that even though this is an infinite sum, if  $\gamma < 1$ ,  $G_t$  will be finite, as long as  $\{R_k\}$  is bounded [1]. If we set the value of  $\gamma$  to 0, the agent will only consider immediate reward, and future rewards will be of no consequence. Conversely, as  $\gamma \rightarrow 1$ , the agent values future rewards more strongly [1]. The return  $G_t$  for a given time step  $t$  can be expressed recursively as:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \quad (3)$$

By breaking down the computation of expected returns into smaller, incremental steps, the recursive property enables efficient updating of value estimates during the learning process. This is particularly useful in temporal difference learning methods, such as Q-learning and SARSA, where the value function is updated incrementally after each transition. This recursive property allows RL algorithms to leverage current estimates of future returns,  $G_{t+1}$ , to improve the current value estimate  $G_t$ . This process, known as bootstrapping, helps algorithms converge faster by taking advantage of information already available, rather than waiting to accumulate the complete return through the end of an episode. We will delve deeper into this idea in Section 3, where we explore Temporal Difference learning and Q-learning.

In deep RL, we deal with parameterized policies: policies whose outputs are computable functions that depend on a set of parameters (e.g., weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm.

## §2.4 Goal of Reinforcement Learning

In reinforcement learning, we define the agent's experience as a trajectory, which includes a sequence of states, actions, and rewards. At each time step  $t$ , the agent is in a state  $s_t$  and takes action  $a_t$  according to policy  $\pi(a_t | s_t)$ , and transition probabilities to a new state  $s_{t+1}$  according to environment's transition probabilities  $p(s_{t+1} | s_t, a_t)$ . A reward  $r_t$  is also received.

$$p_{\theta}(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^T \pi_{\theta}(a_t | s_t) p(s_{t+1} | s_t)$$

★ A **trajectory**  $\tau$ , also called an episode, is sequence of states, actions and rewards from the start to the end of an episode,  $\tau \equiv (s_1, a_1, \dots, s_T, a_T)$ . We will use the notation  $p_{\theta}(\tau)$  to denote the probability of the entire trajectory  $\tau$  under policy parameters  $\theta$ , and let  $R(\tau)$  denote the total reward of the trajectory.

<sup>1</sup>The discount factor  $\gamma$  serves to model scenarios in which a future reward is considered less desirable than an immediate reward of the same magnitude.

The distribution over trajectories under policy  $\pi_\theta$  is given by the product of an initial state distribution, the policy probabilities for the actions, and the environment's transition probabilities for the state.

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(s_t, a_t) \right]$$

The objective in reinforcement learning is to find the policy parameters  $\theta$  of a policy  $\pi_\theta(a_t | s_t)$ , which is a conditional distribution over actions  $a_t$  conditioned states  $s_t$ , with respect to the expectation of a reward function  $r(s_t, a_t)$ .

Our goal is to maximize the expected return, where the return is defined as the sum of rewards over trajectory. The expectation here is taken over the distribution of trajectories. The objective reflects the goal of finding a policy that, on average, produces trajectories with high rewards as possible, taking into account the randomness in the policy, the environment's transitions, and the initial state distribution.

## §2.5 Value and State-Value Functions

Now we can define the notion of “value of a state” formally via **value functions**. Value functions in RL are tools for estimating the long-term ‘value’ or desirability of being in a given state. They allow the agent to access how good it is to be in a certain state, based on the expected cumulative reward obtained from that point onwards. There are two types of value functions commonly used in reinforcement learning:  $V^\pi(s)$  and  $Q^\pi(s, a)$ , we define them formally below [1], [8]:

**Definition 2.3 (Value function)** The value  $V^\pi(s)$  for a fixed policy  $\pi$  at a given state  $s \in S$  represents the expected reward obtained when initiating at state  $s$  and adhering to policy  $\pi$  :

- $V^\pi(s) = \mathbb{E}_{\pi_{\theta}} [G_t | S_t = s]$  for all  $s \in S$  [1]
- finite horizon:  $V^\pi(s) = \mathbb{E}_{\pi_\theta} \left[ \sum_{t=0}^T r(s_t, a_t) | S_t = s \right]$  [8]
- infinite discounted horizon:  $V^\pi(S_t = s) = \mathbb{E}_{\pi_\theta} \left[ \sum_{t=0}^{+\infty} \gamma^t r(s_t, a_t) | S_t = s \right]$  [8]<sup>a</sup>

<sup>a</sup>Technically the notation should be:  $\mathbb{E}_{a_t \sim \pi(s_t)} \mathbb{E}_{s_t}$  since we are not just taking the expectation over the random selection of an action  $a_t$  according to the policy distribution  $\pi(a_t | s_t)$ , but we are also taking the expectation over the states  $s_t$  reached and the corresponding reward values  $r(s_t, a_t)$ . However, in RL literature/textbooks the randomization with respect to the next state and reward function doesn't seem to be explicitly mentioned (to simplify notation perhaps)

An intuitive way of understanding Value Function is that it tells us how much “accumulated future reward” we expect to obtain from a given state.

Starting from a state  $s \in S$ , to maximize reward, an agent naturally seeks a policy  $\pi$  with the largest value  $V_\pi(s)$ . A remarkable result for finite MDPs in the finite horizon case, there exists an *optimal policy* for all starting states  $s \in S$ . Formally, a policy  $\pi^*$  is optimal if for any policy  $\pi$  and any state  $s \in S$ ,  $V^{\pi^*}(s) \geq V^\pi(s)$ . We can prove that this optimal policy  $\pi^*$  is deterministic [8], and this proof can be found in Mohri et al. (2018).

**Action-Value Function ( $Q^\pi$ ):** Represents the expected cumulative reward an agent can obtain by taking an action  $a$  in a given state  $s$  and following a certain policy  $\pi$  thereafter.

**Definition 2.4 (State-action value function)** The state-action value function  $Q$  associated to a policy  $\pi$  is defined for all  $(s, a) \in S \times A$  as the expected return for taking action  $a \in A$  at state  $s \in S$  and then following policy  $\pi$ :

$$Q^\pi(s_t, a_t) = \mathbb{E}_{\pi_\theta}[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[ \sum_{k=0}^{+\infty} \gamma^k r(s_k, a_k) \mid S_t = s, A_t = a \right]$$

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi(a_t | s_t)}[Q^\pi(s_t, a_t)]$$

The motivation behind using value functions in RL is to guide the agents decision-making process. By estimating the values of states or state-action pairs, the agent can choose actions that lead to states with higher-long term rewards. We will see in next section that values serve as the basis for various RL algorithms, such as  $Q$ -learning and SARSA.

Value functions  $V^\pi$  and  $Q^\pi$  can be estimated through experience. For instance, if an agent follows policy  $\pi$  and maintains an average for each state encountered, based on the actual returns following that state, these averages will converge to  $V^\pi(s)$  for all  $s \in S$  as the number of times the state is encountered approaches infinity. By keeping separate averages for each action taken in each state, these averages will converge to the action-values  $Q^\pi(s, a)$ . Such estimation methods are referred to as **Monte Carlo methods** because they involve averaging over numerous samples of actual returns.

## §2.6 The Bellman Equations

Value functions in reinforcement learning exhibit recursive relationships, like the one established for the return in equation (3). For every policy  $\pi$  and state  $s$ , this recursive relationship exists that links the value of  $s$  to the values of its potential subsequent states [1]:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_{a \in A} \pi(a \mid s) \sum_{s'} \sum_r \mathbb{P}(s', r \mid s, a) [r + \gamma \mathbb{E}_\pi(G_{t+1} \mid S_{t+1} = s')] \\ &= \sum_{a \in A} \pi(a \mid s) \sum_{s'} \sum_r \mathbb{P}(s', r \mid s, a) [r + \gamma V^\pi(s')], \quad \forall s \in S \end{aligned}$$

The equation represents a recursive formulation of the value function, and this is known as the **Bellman Expectation Equation** [1], [4], [9]. The Bellman equation is a fundamental equation in reinforcement learning and in dynamic programming.

★ In simple terms, the Bellman Equation decomposes the value function into immediate reward from taking an action in the current state plus the discounted expected value of future states, while taking into account all future possible actions.

The Bellman equation tells us that the value of any state must be equal to the expected reward received from being in that state plus the (discounted) value of the expected next state. Another more compact way to express the Bellman equation commonly found in the literature is:

$$V^\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \cdot \sum_{s'} \mathcal{P}_{s,s'}^a [r(s, a) + \gamma V^\pi(s')] \quad (4)$$

The equation states that the value of a given state,  $V^\pi(s)$  under  $\pi$ , is equal to the immediate reward  $r(s, a)$  plus discounted value of the next state  $\gamma V^\pi(s')$ . Recall that  $V^\pi(s')$  is also an expectation, and therefore, it is considering all possible actions and transitions to the next state, weighted by their probabilities  $\mathcal{P}_{s,s'}^a = \mathbb{P}(s' \mid s, a)$ .

Bellman Equation for Action-Value Function can be expressed in the same way:

$$Q^\pi(s, a) = \mathcal{P}_{s,s'}^a[r(s, a) + \gamma \overbrace{\sum_{a'} \pi(a' | s') Q^\pi(s', a')}^{V^\pi(s)}] \quad (5)$$

$$= \mathcal{P}_{s,s'}^a[r(s, a) + \gamma V^\pi(s)] \quad (6)$$

Since,

$$V^\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a | s) Q^\pi(s, a)$$

## §2.7 Finding Optimal Policies

Value functions allow us to create a hierarchical structure among policies, where a policy  $\pi$  is deemed superior or equal to another policy  $\pi'$  if it yields an expected return that is greater than or equal to that of  $\pi'$  across all states [4]. Thus,  $\pi \geq \pi'$  if and only if the condition  $V_\pi(s) \geq V_{\pi'}(s)$  holds true for every state  $s \in S$  [8]. For finite MDPs, there is always at least one policy that is better than or equal to all other policies [1]. This is what we call the *optimal policy*. Although there might be more than one, we denote all the optimal policies by  $\pi^*$ . The optimal policies share the same state-value function and the action-value functions which we denote as  $V^*$  and  $Q^*$  [4].

$$V^*(s) = \max_{\pi} V^\pi(s), \quad \forall s \in S, a \in A$$

$$Q^*(s) = \max_{\pi} Q^\pi(s), \quad \forall s \in S, \forall a \in A$$

★ A useful fact which will be leveraging is that we can express  $Q^*$  in terms of  $V^*$ :

$$\begin{aligned} Q^*(s, a) &= \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1}] + \gamma \sum_{s' \in S} \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] V^*(S_{t+1}) \end{aligned}$$

Note that  $R_{t+1}$  is a random variable since we don't know what future state  $s'$  we will end up in, therefore, we are taking the expectation  $\mathbb{E}[R_{t+1}]$  over the dynamics of transition probabilities of the next state  $s'$  we reach.

Recall that all value functions  $v(s)$ , must satisfy the recursive/self-consistency property established in (4). This mean we can express  $V^*$  in a similar fashion. We write  $V^*$  without reference to any specific policy because it is the optimal value function. We follow a similar derivation which can be found in [1], [3], [4], [8]:

$$\begin{aligned} V^*(s) &= \max_{a \in A} Q^\pi(s), \quad \forall s \in S \\ &= \max_a \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s'} \sum_r \mathbb{P}(s', r | s, a) [r + \gamma V^*(s')], \quad \forall s \in S \end{aligned} \quad (7)$$

This is known as the **Bellman Optimality Equation**. The equation expresses that “the value of state under *optimal policy* must be equal to the expected for the best action from that state” [1], and is used to compute the optimal value function which is usually denoted as  $V^*(s)$  in

literature. The Bellman optimality equation for  $Q^*(s, a)$  is:

$$\begin{aligned} Q^*(s, a) &= \mathbb{E} \left[ R_{t+1} + \gamma \max_a Q^*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \\ &= \sum_{s', r} \mathbb{P}(s', r \mid s, a) \left[ r + \gamma \max_a Q^*(S_{t+1}, a') \right] \end{aligned}$$

### Why are $Q^*$ and $V^*$ important?

- $V^*(s)$ : When we have  $V^*$ , determining the optimal policy becomes easy. The beauty of  $V^*$  lies in its ability to transform the greedy policy into the optimal policy in terms of long-term rewards [1], as it takes into account the reward consequences of all possible future behaviors. Through  $V^*$ , the optimal expected long-term return is converted into a value that is readily and locally accessible for each state, allowing for more efficient decision-making [1].
- $Q^*(s, a)$ : With  $Q^*$  at hand, the agent can bypass one-step ahead search. For any state  $s \in \mathcal{S}$ , it can directly identify the action(s) that maximizes  $Q^*(s, a)$ . The action-value function serves as a cache for the results of one-step-ahead searches [1], offering the expected long-term return as a value that is both locally and instantly accessible for each state-action pair.

## §3 Tabular Algorithms

Thus far we have been under the assumption that we have access to a model of the Markov Decision Process (MDP) environment, which includes transition states defined by  $\mathcal{S}$  and the probabilities of the next state and reward given the current state and action<sup>2</sup>. However, in real-world scenarios, we often lack access to an MDP model. When we don't have a model, we don't have access to the environment's dynamics such as transition probabilities,  $\mathbb{P}(S_{t+1} \mid S_t, A_t)$ , or rewards functions  $r(s_{t+1}, a_t, s_t)$ . Consequently, we cannot predict the next state and the immediate reward you will receive when taking a specific action in a given state. Instead, the agent must interact with the actual MDP environment to learn about it. Interacting with the environment doesn't provide transition probabilities; it simply presents a new state and reward when an action is taken in a specific state. The environment supplies individual experiences of the next state and reward, rather than the explicit probabilities of occurrence for the next states and rewards. This raises a pertinent question: is it possible to compute the optimal value function or optimal policy without access to a model?

### §3.1 Dynamic Programming

Dynamic Programming (DP) is a mathematical approach used to solve complex problems by breaking them down into simpler subproblems. It is particularly effective when the problem has overlapping subproblems and optimal substructure. In the context of Reinforcement Learning, the two main DP algorithms are **Policy Iteration** and **Value Iteration**. These algorithms involve the evaluation and improvement of policies in order to find the optimal policy that maximizes the expected cumulative reward.

The mathematical formulation for these algorithms involves the Bellman equations. For a given policy  $\pi$ , the value function  $V^\pi$  is defined as the expected return from each state, and satisfies the following Bellman equation for policy evaluation:

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s', r} \mathbb{P}(s', r \mid s, a) [r + \gamma V^\pi(s')], \quad \forall s \in \mathcal{S} \quad (8)$$

where  $\mathbb{P}$  represents the state transition probability, and  $\gamma$  is the discount factor that determines the present value of future rewards.

<sup>2</sup>By model we mean the transition states define by  $\mathcal{P}$ , probabilities of next state and reward, given current state and reward

In Policy Iteration, we alternately evaluate the current policy using the Bellman equation for policy evaluation, and improve the policy by making it greedy with respect to the current value function:

$$\pi'(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V^\pi(s')], \quad \forall s \in \mathcal{S} \quad (9)$$

Value Iteration combines the policy evaluation and policy improvement steps into a single update:

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V_k(s')], \quad \forall s \in \mathcal{S} \quad (10)$$

The iterations continue until the value function (or the policy in policy iteration) converges, and the final value function represents the maximum expected return from each state, under the optimal policy.

In practical scenarios, Dynamic Programming (DP) is not commonly used to solve Reinforcement Learning problems. This is primarily due to the following reasons:

- It's challenging to extend DP to handle continuous actions and states. DP methods typically require discretized states and actions, which can be impractical or infeasible for many real-world problems.
- DP methods necessitate access to the environment's model, specifically, the transition probabilities  $P_{ss'}^a$ . In reality, such detailed information about the environment is rarely available. Instead, we can only obtain samples from the environment by interacting with it and collecting experiences. These experiences can then be used to approximate the expectations via sampling-based methods, such as Temporal Difference (TD) learning methods.

However, there are advantages to using DP, including: DP methods provide mathematically precise solutions that can be formally expressed and analyzed. For smaller-scale problems, with a few thousand states or few tens to hundreds of actions, DP could be the most suitable method. It offers stability, simplicity, and straightforward convergence guarantees. While DP isn't always practical for larger or continuous problems, the concepts it introduces are foundational to understanding more advanced reinforcement learning techniques we will be looking in the next section(s).

### §3.2 Monte Carlo Learning

Monte Carlo (MC) methods in reinforcement learning involve learning from complete sample returns. For a given policy,  $\pi$ , MC methods estimate the value-function  $v_\pi(s)$  or  $q_\pi(s,a)$  as the average of the returns observed in a number of episodes.

The return  $G_t$  at a time-step  $t$  in an episode is defined as the cumulative discounted reward from that time-step onwards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

where  $R_{t+i}$  is the reward after  $i$  steps, and  $\gamma$  is the discount factor.

The estimate  $V(s)$  for the value of a state  $s$  is obtained by averaging the returns following all visits to  $s$ . We maintain a sum  $S(s)$  and a count  $N(s)$  for each state  $s$ :

$$S(s) = \sum_{t \in \text{visits}(s)} G_t, \quad N(s) = \text{number of visits to } s$$

Finally, the value of state  $s$  is estimated as:

$$v_\pi(s) \approx V(s) = \frac{S(s)}{N(s)}$$

One major advantage of Monte Carlo methods is that they do not require a model of the environment; they learn directly from raw episodes of experience. These methods are both conceptually simple and easy to implement.

**Algorithm 1** Monte Carlo ES (Exploring Starts) for estimating  $\pi \approx \pi^*$ 


---

```

1: procedure MONTECARLOES( $\mathcal{S}, \mathcal{A}, \gamma$ )
2:   Initialize  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$ 
3:   Initialize  $Q(s, a)$  arbitrarily for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ 
4:   Initialize Returns( $s, a$ ) as an empty list, for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ 
5:   while True do ▷ Loop forever (for each episode)
6:     Choose  $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$  randomly such that all pairs have probability  $> 0$ 
7:     Generate an episode from  $S_0, A_0$ , following  $\pi : S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ 
8:      $G \leftarrow 0$ 
9:     for each step of episode,  $t = T - 1, T - 2, \dots, 0$  do
10:       $G \leftarrow \gamma G + R_{t+1}$ 
11:      if the pair  $S_t, A_t$  is unique in  $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$  then
12:        Append  $G$  to Returns( $S_t, A_t$ )
13:         $Q(S_t, A_t) \leftarrow \text{average}(\text{Returns}(S_t, A_t))$ 
14:         $\pi(S_t) \leftarrow \max_a Q(S_t, a)$ 
15:      end if
16:    end for
17:  end while
18: end procedure

```

---

### §3.3 Temporal difference learning

The Temporal-Difference (TD) learning method is prevalently employed in reinforcement learning applications [15]. TD learning is a method for estimating the expected cumulative future reward, or “value,” of each state in an environment. This value is updated at each time increment based on the difference between the predicted value of a state and the observed value after taking an action and transitioning to a new state [1], [4]. TD learning leverages a hybrid approach that combines Monte Carlo’s model-free experience-based value estimation with the benefits of dynamic programming’s capability to calculate values utilizing present estimates alone [10].

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)) \quad (11)$$

- We refer to  $R_{t+1} + \gamma \cdot V(S_{t+1})$  as the **TD target**
- We refer to  $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)$  as **TD error**

The TD target is a (biased) estimate of  $G_t$  [1]. The TD control algorithm can update the Q-value function after each time step.

### §3.4 Generalized Policy Iteration (GPI)

Virtually every reinforcement learning (RL) control algorithm has its roots in the fundamental principle of Generalized Policy Iteration (GPI). However, the exact implementation of GPI varies and differs from one algorithm to another. Therefore, prior to delving into the specifics of RL Control algorithms, its crucial to thoroughly understand the overarching concept of GPI.

The term Generalized Policy Iteration (GPI) refers to the approach of interweaving Policy-Evaluation and Policy-Improvement procedures. GPI assess the Value function for a given policy,  $\pi$ , using *any* Policy Evaluation method, and subsequently improves the policy using *any* Policy Improvement method.

That is, the policy consistently improves itself in relation to the Value Function, while the Value Function moves towards the Value Function corresponding to the policy. If both the evaluation process and improvement process reach an equilibrium (i.e., no further changes transpire), then the resulting policy and Value Function must be optimal.

The value function reaches stability once it aligns with the current policy, i.e., when the policy evaluation step is complete. The policy, on the other hand, stabilizes when it becomes greedy with respect to the current value function, that is, during the policy improvement step. Both processes reach an equilibrium when a policy that is greedy with respect to its own evaluation



function is found. This is the optimal policy, where any further changes neither improve the value function nor the policy itself.

Within the Generalized Policy Iteration framework, both an *approximate policy* and *approximate value function* are maintained. The Value Function is continually modified to more closely approximate the true Value Function of the current policy, while the policy is perpetually improved in relation to the existing Value Function. While these two modifications are somewhat counteractive – each sets a dynamic target for the other – their combined effect drives both the policy and Value Function towards optimally. Policy evaluation is performed through experience from many episodes, with the approximate value function converging asymptotically to the true function.

One of the most important features of the GPI framework is its ability to accommodate 'partial' policy improvement. The term 'partial' here encompasses a broad range of computations, any of which contribute either to a full policy evaluation or towards policy improvement. Consequently, GPI offers flexibility to transition between policy evaluation and policy improvement without the need for completely exhaustive policy evaluation or complete policy improvement. For instance, we do not have to wait for policy evaluation computations to fully converge before proceeding with policy improvement.

The policy improvement theorem provides a guarantee that each successive policy  $\pi_{k+1}$  is uniformly superior to, or at least as effective as, its predecessor  $\pi_k$ . In situations where  $\pi_{k+1}$  is equivalent to  $\pi_k$ , they are both classified as optimal policies. This implies that the overall process will ultimately converge to an optimal policy and corresponding optimal value function.

In the context of Monte Carlo policy iteration, it is intuitively appealing to alternate between evaluation and improvement on a per-episode basis. Following each episode, the observed returns serve as inputs for policy evaluation, after which the policy undergoes refinement at all the states encountered during the episode.

### §3.5 Why $Q$ -function for Control?

The  $Q$ -function serves as the an evaluative metric for state-action pairs  $(s, a)$ , under a specific policy  $\pi$ . Recall that the value of  $(s, a)$  is the expected cumulative discounted reward obtained from taking action  $a$  in state  $s$ , followed by adherence to policy  $\pi$  in the subsequent course of actions. In situations where an environmental model is not available, alongside determining the value of various states, an agent needs to also consider alternative moves and the expected consequences of making those moves.

★ The  $Q^\pi(s, a)$  function assigns a numerical value to each potential action, this value can be used to determine of the most optimal move to execute in a given state.

One of the key advantages of  $Q^\pi(s, a)$  is its provision of a direct strategy for agent action. Agents can compute  $Q^\pi(s, a)$  for each feasible action  $a \in A(s)$  within a given state  $s$ , and select the action with the highest value. When  $Q^\pi(s, a)$  is optimal, it gives the maximum expected value from taking action  $a$  in state  $s$ , representing the best potential performance if the agent were to operate optimally in all future states. Hence, a knowing  $Q(s, a)$  is automatically yields an optimal policy.

On the other hand, in order to utilize  $V^\pi(s)$  for action selection, the agent needs to experiment with each of the available actions  $a \in A(s)$  in a state  $s$ , observe the consequent state transition to  $s'$ , and keep track the resulting reward. Only then can an agent can act optimally by choosing the action leading to the highest expected cumulative discounted reward,  $\mathbb{E}[r + V^\pi(s')]$ . However, in situations where state transitions are stochastic, that is, where taking action  $a$  in state  $s$  can result in varying next states  $s'$ , the agent might have to replicate this process repeatedly to acquire a reliable estimate of the expected value from a specific action. This one-step look-ahead requirement for  $V^\pi$  often poses a challenge in RL Control problems.  $Q^\pi(s, a)$  circumvents this issue by directly learning the value of  $(s, a)$ , thereby storing the one-step look-ahead for each action  $a$  in every state  $s$ . Therefore, for Control algorithms, which select actions based on a learned value function generally approximate the approximate  $Q^\pi(s, a)$ .

## §4 Deep Reinforcement Learning

### §4.1 Value Function Approximation

Function approximation in reinforcement learning is a method used to estimate state-value functions and the policy functions. The value function is not longer represented as a table but instead as a parameterized functional form with a weight vector  $\mathbf{w}$  residing in  $\mathbb{R}^d$ .

Commonly, we denote  $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$  to signify the approximate value of state  $s$  given the weight vector  $\mathbf{w}$ . For instance,  $\hat{v}$  could be a function computed by a neural network, with  $\mathbf{w}$  serving as the weight vector for all connections. By fine-tuning these weights, the neural network progressively refines its approximation, moving ever closer to accurately representing the true value function. This continual fine-tuning allows the network to adapt and improve its predictions over time, enhancing its understanding of the environment and its dynamics. Often, the number of weights is fewer than the number of states. As a consequence, when a single state undergoes an update, the modification proliferates from that state, influencing the values of other states.

Similar to supervised learning, value function approximation just boils down to finding the specific parameter vector, denoted as  $\mathbf{w}$ , which minimizes the loss between the real value function,  $V_\pi(s)$ , and its approximation, denoted as  $\hat{V}(s; \mathbf{w})$ . Generally use mean squared error and define the loss as

$$J(\mathbf{w}) = \mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s; \mathbf{w}))^2]$$

By minimizing this loss, we improve the accuracy of the approximated value function, bringing it closer to the true value function and thereby improving the performance of our model. We can use gradient descent to find a local minimum:

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where,

$$\begin{aligned} \Delta_{\mathbf{w}} J(\mathbf{w}) &= \Delta_{\mathbf{w}} \mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s; \mathbf{w}))^2] \\ &= \mathbb{E}_\pi[2(V^\pi(s) - \hat{V}(s; \mathbf{w})) \Delta_{\mathbf{w}} \hat{V}(s; \mathbf{w})] \end{aligned}$$

The majority of prediction methods can be viewed as updates to an estimated value function that nudges its value at a specific state toward a ‘backed-up value’ or an update target. An individual update can be represented as  $S_t \mapsto U_t$ , where  $S_t$  represents the state and  $U_t$  denotes the update target towards which the estimated value of  $S_t$  is shifted. For example, the Monte Carlo Update for value prediction, characterized as  $S_t \rightarrow G_t$ , and the Temporal Difference (TD(0)) update, depicted as  $S_t \rightarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t)$ .

Each update in reinforcement learning can be viewed as a training instance analogous to supervised learning. Function approximation methods in reinforcement learning receive examples of the input-output behavior of the function they aim to approximate. This allows us to use supervised learning techniques for value function approximation by treating each  $s \rightarrow u$  update as a training instance. The resulting approximate function is then interpreted as an estimated value function.

### §4.2 State-Value Function Approximation

Analogous to value function approximation, we also use function approximation to estimate the state-value functions:  $\hat{Q}_\pi(S_t, A_t)$ . In control problems the parametric approximation of the action-value function is represented as  $\hat{Q}(s, a, \mathbf{w}) \approx q_*(s, a)$ , where  $\mathbf{w} \in \mathbb{R}^d$  is a finite dimensional weight vector. Instead of  $S_t \mapsto U_t$ , we now consider examples of the form  $(S_t, A_t) \mapsto U_t$ .

- $S_t$  represents the state at time  $t$
- $A_t$  is the action taken at time  $t$ .

Here, we’re using  $(S_t, A_t)$  as an example to predict  $U_t$ , which is our update target. The update target  $U_t$  can be any approximation of the action-value function such as the Monte Carlo return,  $n$ -step SARSA target, or Q-learning target. The key idea here is to use experiences to incrementally

improve the approximation of the state-value, with the aim of improving the policy based on that approximation. Like in supervised learning the most convenient way to accomplish this is by minimizing the mean-squared error between the true <sup>3</sup> action-value function  $Q^\pi(s, a)$  and the approximate action-value function:

$$J(w) = \mathbb{E}_\pi[(Q^\pi(s, a) - \hat{Q}^\pi(s, a; w))^2]$$

Use stochastic gradient descent to find a local minimum

$$\begin{aligned}\Delta(w) &= \alpha \nabla_w J(w) \\ &= \alpha \mathbb{E}[Q^\pi(s, a) - \hat{Q}^\pi(s, a; w)] \nabla_w \hat{Q}^\pi(s, a; w)\end{aligned}$$

Just as in policy evaluation, the true state-action value function for a state is not known, and as such, we utilize a **target value** as a substitute.

- **Monte Carlo Target:**

$$\Delta w = \alpha \left( G_t + \gamma \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right) \nabla_w \hat{Q}(s, a; w)$$

- **SARSA Target:**

$$\Delta w = \alpha \left( r + \gamma \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right) \nabla_w \hat{Q}(s, a; w)$$

- **Q-learning:**

$$\Delta w = \alpha \left( r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right) \nabla_w \hat{Q}(s, a; w)$$

Theoretically, any supervised learning approach could be employed to estimate value functions. However, not all function approximation techniques are equally appropriate or efficacious when applied to reinforcement learning. Many statistical machine learning algorithms rely on a static training set derived from a stationary distribution. In contrast, reinforcement learning generates data sequentially as the agent interacts with its environment or a model of the environment. It is important that learning can occur in an online environment, and to facilitate this, supervised learning techniques that can approximate non-stationary target functions are required. For instance, in control methods based on Generalized Policy Iteration, our objective is to learn  $Q_\pi$  while  $\pi$  undergoes changes. Even if the policy remains constant, the target values of training examples become non-stationary if they are generated by bootstrapping methods like Dynamic Programming or Temporal Difference learning. Thus, it is crucial to recognize that methods incapable of handling such non-stationary data are less suitable for reinforcement learning.

## §5 Deep Q-Networks (DQN)

### §5.1 Why Neural Networks?

Linear value function approximators assume that the value function is essentially a weighted combination of a set of features, each of which is a function of the state. This approach to value function approximation can be effective given an appropriate set of features. However, it often necessitates the careful manual design of this feature set, which can be a complex and time-consuming process.

An alternative is to leverage a more sophisticated class of function approximators that can process states directly without the need for an explicit feature specification. Local representations, such as those based on kernel approaches, have certain appealing properties. These include the ability to converge under certain conditions. However, these methods often struggle to scale effectively when dealing with large spaces and data sets.

<sup>3</sup>In practice we won't have access to the true action-value function

Deep Neural Network (DNN) approximators offer significant benefits in this context. First and foremost, DNNs are universal function approximators, capable of representing a wide variety of complex functions. Compared to shallow networks, they can potentially represent the same functions with exponentially fewer nodes or parameters, which is a significant advantage when dealing with high-dimensional data. Furthermore, the parameters of a DNN can be learned using stochastic gradient descent, an efficient and well-established optimization technique. This makes DNNs a powerful tool for function approximation in reinforcement learning, particularly when dealing with large and complex state spaces.

## §5.2 Q-Learning via Function Approximation

In Deep Reinforcement Learning we can use deep neural networks to represent the Value function, Policy, or Model. For now, we focus on how we can approximate the value function, in subsequent sections we will see how deep neural networks can also be employed directly for policy approximation. As in the case of policy evaluation, the true state-action value function is unknown. Therefore, we substitute it with a surrogate, or an approximation, which we refer to as the **target value**, denoted as  $Q(s, a)$ . This target value function serves as our best estimate for the true state-action value function. This is similar to the gradient calculation in linear value function approximation, but here the gradient is computed with respect to a more complex function, a deep neural network, rather than a simple linear function:

$$\Delta w = \alpha \left( r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right) \nabla_w \hat{Q}(s, a; w)$$

Taking a step back, it's important to recall that in tabular Q-learning each state-action pair is assigned a  $Q$ -value, and these values are updated iteratively until the values converge to obtain the optimal  $Q$ -function,  $Q^*(s, a)$ . However, now we are adopting a different approach, instead of maintaining a table of  $Q$ -values, we will use a function approximator (like a neural network) parameterized by weights, to represent the  $Q$ -function. The objective in this case is to adjust the weights of the function approximator to minimize the mean squared error (MSE) between the predicted  $Q$ -values and the target  $Q$ -values via stochastic gradient descent. The target  $Q$ -values are based on the Bellman equation, using the current estimates of the  $Q$ -values, rather than the true  $Q$ -values (which are unknown). This allows us to derive a gradient for updating the weights of the function approximator.

However, this strategy can lead to divergence in Q-learning with Value Function approximation (VFA), posing a significant challenge to the learning process. Two primary issues contribute to this problem: the correlations between samples, and the non-stationarity of the targets. These factors can disrupt the learning dynamics and lead to instability. Deep Q-Learning (DQN) was developed to address these issues. The technique introduces two key modifications to the traditional Q-learning procedure:

1. **Experience Replay:** DQN stores the agent's experiences at each time-step in a data set known as a replay buffer. During the learning updates, DQN randomly samples mini-batches from the replay buffer. This strategy breaks the correlation between consecutive samples, making the data more independent identically distributed (i.i.d.) data, which is more amenable to learning.
2. **Fixed Q-targets:** In DQN, the network weights used to compute the target  $Q$ -values are held fixed for a number of updates, and only then updated to the current weights of the  $Q$ -network. This strategy reduces the non-stationarity of the targets, leading to more stable learning.

## §5.3 Experience Replay

Algorithms that operate off-policy, like DQN, have the advantage of re-using experiences, rather than discarding after one update. Off-policy methods, such as Deep Q-Networks (DQN), learn from the experiences of a different policy (the behavior policy), while improving another (the target policy). The advantage here is that experiences (state, action, reward, next state tuples)

can be stored in a replay buffer and can be sampled multiple times. Since each experience can be reused multiple times, this makes the learning process much more data efficient. This notion was first described in 1992 by Long-Ji Lin's paper, where he introduced the term "experience replay". Lin's critical insight was that Temporal Difference (TD) learning, could be inherently slow due to its dependence on a trial and error approach for data acquisition. Deep Q-Networks (DQNs) introduced by Mnih et al., 2013, have successfully employed Experience Replay to enhance the sample efficiency and stability of learning in Reinforcement Learning environments. The Experience Replay mechanism forms the foundation for facilitating temporally uncorrelated learning processes by maintaining a replay buffer, also called the experience pool, which stores the agent's interactions with the environment.

- **Experience Collection:** As the agent operates within the environment, each interaction generates a tuple of state, action, reward, and next state - denoted as  $(s, a, r, s')$ . This tuple, also known as an experience or transition, is stored in a data structure called the replay buffer or replay memory denoted as  $\mathcal{D}$ . This buffer has a fixed size, so older experiences are discarded when the buffer is full.
- **Random Sampling:** Instead of only learning from the most recent experience as in traditional Q-learning, DQNs also learn from a random sample of previous experiences drawn from the replay buffer. This is known as experience replay.

$$(s, a, r, s') \sim \mathcal{D}$$

- **Computing Q-value Targets and Loss:** The agent learns by computing a loss between the Q-value predictions from the DQN and the target Q-values computed from sampled experiences. The target Q-value for an action is the reward for taking that action plus the maximum Q-value for the next state, discounted by a factor gamma ( $\gamma$ ):

$$r + \gamma \max_{a'} Q(s', a'; \mathbf{w})$$

- **Network Update:** The parameters of the DQN are then updated using a form of gradient descent to update the network weights.

$$\Delta w = \alpha \left( r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}) - \hat{Q}(s, a; \mathbf{w}) \right) \nabla_w \hat{Q}(s, a; \mathbf{w})$$

## §5.4 DQNs: Fixed Q-Targets

The second key improvements introduced by Mnih et. al. in Human-level Control through Deep Reinforcement Learning to help to stabilize training was the use of **Target Networks**. It was motivated by the fact that in original DQN algorithm,  $Q(s, a)$  is constantly changing because it depends on  $\hat{Q}(s, a; \mathbf{w})$ . During training the Q-network parameters  $\mathbf{w}$  are adjusted to minimize the difference between  $Q(s, a)$  and  $\hat{Q}(s, a; \mathbf{w})$ , but this is difficult when  $Q(s, a)$  changes at each training step.

To help improve stability, we use a second neural network, with different weights,  $\mathbf{w}^-$  called the target network.

This is simply a lagged copy of the Q-network  $\hat{Q}(s, a; \mathbf{w})$ <sup>4</sup>. This network is a copy of the original network, which gets updated less frequently (i.e., the weights are "frozen" for a number of steps before being updated to match the current weights of the original network). This use of a target network helps to stabilize the training process by making the targets more consistent across updates. As the name suggests, the target network  $\hat{Q}(s, a; \mathbf{w}^-)$  is used to compute the

<sup>4</sup>The original DQN update does not include the use of a target network. The use of a target network is an extension to the original DQN, which was introduced to further stabilize the training process.

estimated  $Q(s, a)$  from the Bellman Equation:

$$\begin{aligned} Q(s, a) &= r + \max_{a'} \hat{Q}(s, a; \mathbf{w}) && \text{Original DQN update} \\ Q(s, a) &= r + \max_{a'} \hat{Q}(s, a; \mathbf{w}^-) && \text{Modified Bellman update} \end{aligned}$$

Periodically  $\mathbf{w}^-$  is updated to the current values for  $\mathbf{w}$ . This is known as a **replacement update**. The update frequency for  $\mathbf{w}^-$  is problem dependent. For example, in the Atari games it is common to update  $\mathbf{w}^-$  every 1,000–10,000 environment steps. For simpler problems it is not necessary to wait as long. Updating  $\mathbf{w}^-$  every 100–1,000 time steps will be sufficient.

## §6 Policy Gradient Methods

Thus far, all of the methods we looked at have been centered on value functions. These methods relied on learning the values associated with different actions, by estimating the  $Q^\pi(s, a)$  or  $V^\pi(s)$  functions, followed by the selection of actions based on their estimated values. The policies function were implicitly/indirectly derived from the estimates of state-action value function. **Policy Gradient Methods** are a class of reinforcement learning algorithms that learn a parameterized policy that can select actions without a value function.

We denote the policy's parameter vector as  $\theta$ , which is an element of the  $d$ -dimensional real number space  $\mathbb{R}^d$ . In this notation,  $\pi_\theta(a | s)$  represents the probability that the policy parameterized by  $\theta$  will select action  $a$  when the environment is in state  $s$  at time  $t$ . More formally,  $\pi_\theta(a | s) = \mathbb{P}_\theta\{A_t = a | S_t = s\}$ , where  $S_t$  and  $A_t$  represent the state of the environment and the action taken by the agent at time  $t$ , respectively. This function  $\pi_\theta(a | s)$  forms the basis of our policy and dictates how the agent should act in different states for optimal learning and performance.

As we saw in chapter 1, 12, objective function of RL is:

$$\theta^* = \arg \max_{\theta} \underbrace{\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]}_{J(\theta)}$$

Before delving into the optimization of the Reinforcement Learning (RL) objective, let's first understand how we can evaluate it. Given a policy  $\pi_\theta$ , we aim to estimate the value of the RL objective, denoted as  $J(\theta)$ . This objective is the expected cumulative reward for trajectories  $\tau$  sampled according to the policy  $\pi_\theta$ , formally expressed as  $\mathbb{E}_{\tau \sim p_\theta(\tau)} [\sum_t r(\mathbf{s}_t, \mathbf{a}_t)]$ , where  $r(\mathbf{s}_t, \mathbf{a}_t)$  represents the reward at time  $t$  for state-action pair  $(\mathbf{s}_t, \mathbf{a}_t)$ .

As we can run our policy, which essentially means sampling from the initial state distribution and the transition probabilities, we can approximate  $J(\theta)$  by generating "rollouts" or trajectories from our policy. Thus, the RL objective can be empirically approximated as:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[ \sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=0}^{+\infty} r(s_{i,t}, a_{i,t}) \quad (12)$$

Here, we run our policy in the "real-world"  $N$  times to generate  $N$  sampled trajectories. The notation  $s_{i,t}$  and  $a_{i,t}$  represents the state and action respectively at time step  $t$  in the  $i^{th}$  sampled trajectory. Having generated these samples using  $\pi_\theta(a | s)$ , we can obtain an unbiased estimate for the expected total reward by summing the rewards along each sampled trajectory and subsequently averaging these sums over all samples. The accuracy of our approximation improves as  $N$  increases, implying that the more samples we generate, the closer our estimate is to the actual expected value.

In practical applications of Reinforcement Learning (RL), our main goal isn't merely to estimate the objective  $J(\theta)$ , but rather to improve it. To this end, we need an efficient approach for



estimating the gradient of the objective, enabling us to enhance the policy  $\pi_\theta$ .

Policy gradient methods offer a viable strategy for this task. These methods work by learning the policy parameter  $\theta$  through the gradient of a scalar performance measure  $J(\theta)$ , akin to the loss function in conventional machine learning. The overarching aim of policy gradient methods is to maximize performance, achieved by making updates that emulate gradient ascent in the performance measure  $J$ . This iterative process can be succinctly represented as:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \quad (13)$$

Here,  $\theta_{t+1}$  refers to the updated policy parameter,  $\theta_t$  represents the current policy parameter, and  $\alpha$  is the learning rate, determining the magnitude of updates in each step. Importantly,  $\widehat{\nabla J(\theta_t)}$  symbolizes a stochastic estimate of the gradient of the performance measure  $J$ , with respect to  $\theta_t$  at time  $t$ . The "hat" signifies that it is an estimated, rather than the exact, gradient.

The distinctive feature of this estimate is that it can be calculated through sampling, without requiring knowledge of the initial state probabilities or transition probabilities. The expectation of this estimate approximates the true gradient of the performance measure  $J$ . This mechanism effectively allows us to adjust  $\theta$  in the direction that is expected to yield the maximum improvement in  $J$  on average.

★ Policy gradient methods just boil down to learning the policy parameters based on the gradient of some scalar performance measure  $J(\theta)$  with respect to the policy parameters. Policy gradient methods seek to maximize performance (i.e. maximize cumulative reward) so the updates approximate gradient *ascent* in  $J$ :

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$$

We will see that all policy gradient methods essentially follow this recipe.

## §6.1 Policy Gradient Theorem

**Theorem 6.1 (Policy Gradient Theorem)** — For any differentiable policy  $\pi_\theta(a|s)$ , the gradient of the expected return  $J(\theta)$  with respect to the policy parameters  $\theta$  is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot Q^{\pi_\theta}(s_t, a_t)]$$

where  $Q^{\pi_\theta}$  is the action-value function induced by following policy  $\pi_\theta$ . This expression states that the direction of the greatest increase of the expected return is given by the expected value of the product of two terms: the gradient of the log-probability of the action taken at time step  $t$  and corresponding action-value.

<sup>a</sup>.

<sup>a</sup>All policy gradient methods are based on this theorem, and their differences lie in how they estimate the action-value function  $Q_\pi(s, a)$ , and how they deal with the high variance of the policy gradient estimator.

*Proof.* Chapter 13.2 in Sutton & Barto has a nice derivation of the policy gradient theorem for episodic tasks and discrete states. □

This is a foundational theorem for policy gradient methods as it allows us to estimate the gradient of the performance measure with respect to the policy parameters by sampling trajectories under the policy. The estimate is then used to update the policy parameters using gradient ascent, with the aim of improving the policy performance.

## §6.2 Score Function Estimator

In many machine learning problems, particularly in reinforcement learning, we often need to optimize the expectation of a function  $f(x)$  under a probability distribution  $p(x; \theta)$ , parameterized by  $\theta$ . The function  $f(x)$  could, for example, represent the reward or cost associated with the action or state  $x$ . To optimize this expectation, we often leverage the properties of the **score function**. In statistics, the score function represents the gradient of the log-likelihood function

concerning the parameter vector. When evaluated at a specific point on this vector, the score function reveals the slope of the log-likelihood function, highlighting its sensitivity to minute changes in parameter values.

$$\text{score} = \nabla_{\theta} \log p_{\theta}(x)$$

To compute the gradient of this expectation, we can't directly swap the gradient and the expectation (or integral or sum), because the expectation is over  $x$  and the gradient is w.r.t  $\theta$ , and  $x$  and  $\theta$  are not independent (because  $x$  is drawn from a distribution parameterized by  $\theta$ ). Here, a technique known as the *log-derivative trick* or *score function trick* comes to our rescue. This trick allows us to rewrite the gradient of the expectation in a way that enables us to estimate it by sampling from the distribution  $p(x; \theta)$ . This trick relies on the following identity: The log-derivative trick just consists of applying the chain rule to a composite function in which the outer most term is a logarithm. In our case the score function happens to be in such a form, with log being the external term and  $p_{\theta}(x)$  the internal one.

★ It follows that we can apply the log-derivative trick straight away to get:

$$\nabla_{\theta} \log p_{\theta}(x) = \frac{1}{p_{\theta}(x)} \cdot \nabla_{\theta} p_{\theta}(x) = \frac{\nabla_{\theta} p_{\theta}(x)}{p_{\theta}(x)}$$

The last term is called the *score ratio*. The log-derivative trick can be quite helpful. For instance, exploiting this trick we can notice an interesting property of the score function: its expected value is equal to zero:

$$\mathbb{E}_{p_{\theta}}[\nabla_{\theta} \log p_{\theta}(x)] = \mathbb{E}_{p_{\theta}} \left[ \frac{\nabla_{\theta} p_{\theta}(x)}{p_{\theta}(x)} \right] = \int p_{\theta}(x) \frac{\nabla_{\theta} p_{\theta}(x)}{p_{\theta}(x)} dx = \nabla_{\theta} \int p_{\theta}(x) dx = \nabla_{\theta} \cdot 1 = 0$$

The property is fundamental in the context of control variates, which is a variance reduction technique used in Monte Carlo Methods. It exploits information about the errors in estimates of known quantities to reduce error of an estimate of an unknown quantity.

The expectation of this score function under the distribution  $p(x; \theta)$  is denoted as:

$$\mathbb{E}_{x \sim p(x|\theta)}[f(x)] = \int_x p(x; \theta) f(x) dx \quad (14)$$

for discrete variables, or

$$\mathbb{E}_{x \sim p(x|\theta)}[f(x)] = \sum_x p(x; \theta) f(x) \quad (15)$$

for discrete variables.

When our goal is to maximize this expectation with respect to the parameters  $\theta$ , we need to compute the gradient of this expectation. The Policy Gradient methods perform exactly this operation.

$$\nabla_{\theta} \mathbb{E}_{x \sim p(x|\theta)}[f(x)] = \mathbb{E}_{x \sim p(x|\theta)}[f(x) \nabla_{\theta} \log p(x; \theta)] \quad (16)$$

*Proof.*

$$\begin{aligned} \nabla_{\theta} \mathbb{E}_x[f(x)] &= \nabla_{\theta} \sum_x p(x) f(x) && \text{(definition of expectation)} \\ &= \sum_x \nabla_{\theta} p(x) f(x) && \text{(swap sum and gradient)} \\ &= \sum_x p(x) \frac{\nabla_{\theta} p(x)}{p(x)} f(x) && \text{(both multiply and divide by } p(x)) \\ &= \sum_x p(x) \nabla_{\theta} \log p(x) f(x) && \text{(use the fact that } \nabla_{\theta} \log(z) = \frac{1}{z} \nabla_{\theta} z) \\ &= \mathbb{E}_x[f(x) \nabla_{\theta} \log p(x; \theta)] && \text{(definition of expectation)} \end{aligned}$$



$$\begin{aligned}
\nabla_{\theta} \mathbb{E}_{p_{\theta}}[f(x)] &= \nabla_{\theta} \int p_{\theta}(x) f(x) dx \\
&= \int \nabla_{\theta} p_{\theta}(x) f(x) dx \\
&= \int p_{\theta}(x) \frac{\nabla_{\theta} p_{\theta}(x)}{p_{\theta}(x)} f(x) dx \\
&= \int p_{\theta}(x) \nabla_{\theta} \log p_{\theta}(x) f(x) dx \\
&= \mathbb{E}_{p_{\theta}} \left[ \underbrace{\nabla_{\theta} \log p_{\theta}(x)}_{\text{score}} \underbrace{f(x)}_{\text{cost}} \right]
\end{aligned}$$

□

In the context of policy gradients,  $p(x; \theta)$  is the policy  $\pi(a|s; \theta)$ ,  $f(x)$  is the return (or some estimate of it) following action  $x$  (often denoted as  $a$  for action in policy gradient methods), and we seek to maximize the expected return by adjusting the policy parameters  $\theta$ . The resulting policy gradient estimate is:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) Q^{\pi_{\theta}}(s_t, a_t) \right] \quad (17)$$

where  $J(\theta)$  denotes the expected reward under policy  $\pi$  parameterized by  $\theta$ ,  $\tau$  is a trajectory generated by the policy  $\pi_{\theta}$ ,  $T$  is the trajectory's length,  $\log \pi_{\theta}(a_t | s_t)$  represents the log-probability of taking action  $a_t$  under state  $s_t$ , and  $Q^{\pi_{\theta}}(s_t, a_t)$  is the action-value function of action  $a_t$  at state  $s_t$  under policy  $\pi_{\theta}$ .

### §6.3 Deriving Policy Gradient

Here, we consider the case of a stochastic, parameterized policy  $\pi_{\theta}$ . We aim to maximize the expected return  $J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} R(\tau)$ . For the purposes of this derivation, we'll take  $R(\tau)$  to give the finite-horizon undiscounted return, but the derivation for infinite-horizon discounted return setting is almost identical.

We would like to optimize the policy by gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_{\theta} J(\theta) \big|_{\theta_k}$$

We now consider a policy parameterized by  $\theta$ , represented as  $\pi_{\theta}$ . This policy is inherently stochastic. Our main objective is to maximize the expected return, which can be formally expressed as:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [R(\tau)]$$

Here,  $\tau$  is indicative of a trajectory, and  $R(\tau)$  computes the return corresponding to this trajectory. For this derivation,  $R(\tau)$  represents the finite-horizon undiscounted return. However, it's worth noting that the derivation would remain largely analogous if we were considering the infinite-horizon discounted return. To optimize the policy, gradient ascent is employed, updating the policy parameters in the direction of the gradient to achieve higher returns:

$$\theta_{k+1} = \theta_k + \alpha \nabla_{\theta} J(\theta) \big|_{\theta_k}$$

Where  $\alpha$  is the learning rate that determines the step size in the direction of the gradient.

The gradient of the policy performance,  $\nabla_{\theta} J(\theta)$ , is called the **policy gradient**. Algorithms crafted around this optimization technique bear the name *policy gradient algorithms*. In the following discourse, we embark on deriving the elementary form of this expression. Subsequent sections will delve deeper, enhancing this rudimentary form to culminate in the sophisticated versions predominantly employed in standard policy gradient algorithm implementations.

We'll begin by laying out a few facts which are useful for deriving the analytical gradient.

1. **Trajectory Distribution.** The probability of a trajectory  $\tau = (s_1, \mathbf{a}_1, \dots, s_T, \mathbf{a}_T)$  given that actions come from  $\pi_\theta$  is

$$p_\theta(s_1, \mathbf{a}_1, \dots, s_T, \mathbf{a}_T) = p(s_1) \prod_{t=1}^T \pi_\theta(\mathbf{a}_t | s_t) p(s_{t+1} | s_t)$$

2. **Log-Derivative Trick** The log-derivative trick is based on a simple rule from calculus: the derivative of  $\log x$  with respect to  $x$  is  $\frac{1}{x}$ . When rearranged and combined with chain rule, we get:

$$\nabla_\theta P_\theta(\tau) = P_\theta(\tau) \nabla_\theta \log P_\theta(\tau)$$

3. **Log-Probability of a Trajectory** The log-prob of a trajectory is

$$\log [p_\theta(s_1, \mathbf{a}_1, \dots, s_T, \mathbf{a}_T)] = \log p(s_1) + \sum_{t=1}^T \log \pi_\theta(\mathbf{a}_t | s_t) + \sum_{t=1}^T \log p(s_{t+1} | s_t)$$

4. **Grad-Log-Probability of a Trajectory.**

$$\nabla_\theta \left[ \log p(s_1) + \sum_{t=1}^T \log \pi_\theta(\mathbf{a}_t | s_t) + \log p(s_{t+1} | s_t) \right] = \nabla_\theta \sum_{t=1}^T \log \pi_\theta(\mathbf{a}_t | s_t)$$

The derivation of the score function gradient estimator tells us that

$$\nabla_\theta \mathbb{E}_{\tau \sim p_\theta} [R(\tau)] = \mathbb{E}_{\tau \sim p_\theta} [\nabla \log p_\theta(\tau) R(\tau)]$$

**Proposition 6.2** (Derivation of Policy Gradient Estimator).

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) R(\tau) \right] \quad (18)$$

*Proof.*

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)] \\ &= \nabla_\theta \int_{\tau} P_\theta(\tau) R(\tau) \quad \text{definition of expectation} \\ &= \int_{\tau} \nabla_\theta P_\theta(\tau) R(\tau) \quad \text{bring gradient under integral} \\ &= \int_{\tau} \nabla_\theta P_\theta(\tau) \nabla_\theta \log P_\theta(\tau) R(\tau) \quad \text{Log-derivative trick} \\ &= \mathbb{E}_{\tau \sim \pi_\theta} [\nabla_\theta \log P_\theta(\tau) R(\tau)] \\ &= \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) R(\tau) \right] \end{aligned}$$

□

It is somewhat remarkable that we are able to compute the policy gradient without knowing anything about the system dynamics, which are encoded in the transition probabilities. The intuitive interpretation is that we collect a trajectory, and *increase its log probability based on how “good” the trajectory was*. That is, if the reward  $R(\tau)$  is very high, we ought to move in the direction in parameter space that increases  $\log p_\theta(\tau)$ .

One of the salient features of the policy gradient method is its ability to optimize the policy without explicit knowledge of the system dynamics. These dynamics are typically captured in the form of transition probabilities. The underlying intuition for this is anchored in the principle of trajectory optimization. Essentially, given a trajectory, the aim is to *enhance its log probability*

*contingent upon the trajectory's quality.* In more concrete terms, if a trajectory garners a high reward, denoted by  $R(\tau)$ , the logical step is to adjust the policy parameters in a manner that augments the log probability  $\log p_\theta(\tau)$ . This adjustment indicates a positive correlation between the quality of a trajectory and its likelihood under the policy.

★ **Note: trajectory length and time-dependence.** Here, we are considering trajectories with length  $T$ , whereas the definition of MDPs and POMDPs above assumed variable or infinite length, and stationary (time-independent) dynamics. The derivations in policy gradient methods are much easier to analyze with fixed length trajectories – otherwise we end up with infinite sums. The fixed-length case can be made to mostly subsume the variable-length case, by making  $T$  very large, and instead of trajectories ending, the system goes into a sink state with zero reward.

In the above derivation, we have expressed the policy gradient as an expectation. By the properties of expectations, we can approximate this expression using the sample mean. Suppose we collect a collection of trajectories, denoted by  $\mathcal{D} = \{\tau_i\}_{i=1}^N$ . Each trajectory,  $\tau_i$ , is a result of the agent interacting with its environment according to the policy  $\pi_\theta$ . Given this, the policy gradient can be approximated as:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) R(\tau),$$

where  $|\mathcal{D}|$  represents the total number of trajectories in the collection.

To effectively estimate the policy gradient using this approach, it is crucial that our policy representation facilitates the calculation of  $\nabla_\theta \log \pi_\theta(a | s)$ . Furthermore, our ability to execute the policy in a given environment and accumulate a relevant trajectory dataset is equally vital.

Now lets come back to our expression for policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) R(\tau) \right]$$

Taking a step with this gradient pushes up the log-probabilities of each action in proportional to  $R(\tau)$ , the sum of all rewards ever obtained. But this doesn't make much sense. Rewards obtained before taking an actions should have no bearing on how good that action was: only rewards that come after.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[ \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) \sum_{t'=t}^T r_{t'} \right]$$

In this form, actions are only reinforced based on rewards obtained after they are taken <sup>5</sup>.

We'll call this form the *reward-to-go policy gradient*, because the sum of rewards after a point in a trajectory,

$$\hat{R}_t = \sum_{t'=t}^T r_{t'}$$

is called **reward-to-go** from that point, and this policy gradient expresion depends on the reward-to-go from state-action pairs.

## §6.4 Monte Carlo Policy Gradient

The policy gradient theorem can be used to derive a basic policy-gradient algorithm: the REINFORCE algorithm (Williams, 1992) [21]. The REINFORCE algorithm, also known as the Monte Carlo Policy Gradient method, is a simple method for learning policies directly from the policy gradient. The algorithm starts by initialize policy parameters  $\theta$  arbitrarily. For each episode:

- Generate an episode by following policy  $\pi_\theta$ :  $S_0, A_0, R_1, \dots, S_T$ , where  $T$  is the final time step.

<sup>5</sup>The formula we started with included terms fro reinforcing actions proportional to past rewards, all of which

- For each time step of the episode  $t = 0, 1, \dots, T - 1$ :
  - Compute the return  $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T$ .
  - Update policy parameters using the policy gradient:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t | S_t).$$

This algorithm makes use of the policy gradient theorem as it uses full-episode rollouts to compute the returns  $G_t$  and then it updates the policy parameters in the direction that maximizes the expected return in the long run.

The policy is updated in the direction of more reward by increasing the log-probability of the taken action proportional to the received return  $G_t$ . If the return is positive, the log-probability of the taken action is increased, which in turn increases the probability of this action being selected in the same state in the future. On the other hand, if the return is negative, the update decreases the log-probability of the taken action, thus discouraging the selection of this action in the same state.

While simple and effective, the REINFORCE algorithm suffers from high variance in the gradient estimates, which can make the learning unstable. Various techniques, such as using a baseline or using more advanced policy gradient methods, can be employed to reduce the variance and improve the performance.

---

**Algorithm 2** REINFORCE algorithm
 

---

**Require:** Initialize weights  $\boldsymbol{\theta}$  of a policy network  $\pi_{\boldsymbol{\theta}}$

- 1: **for** each episode **do**
  - 2:   Sample a trajectory following  $\pi(\boldsymbol{\theta})$ :  $\tau = (s_0, a_0, r_0, \dots, s_T, a_T, r_T)$
  - 3:   Set  $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = 0$
  - 4:   **for** each step of the episode  $t = 0, 1, \dots, T$  **do**
  - 5:      $R_t(\tau) \leftarrow \sum_{t'=t}^T \gamma^{t'-t} r_{t'}$
  - 6:      $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) + R_t(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t | s_t)$
  - 7:   **end for**
  - 8:    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ <sup>6</sup>
- 

We are interested in finding how we could shift the distribution through its parameter  $\boldsymbol{\theta}$  to increase the scores of its samples, as judged by  $f$ . In the case if policy gradients the  $f(\mathbf{x})$  is our reward function or **advantage function**, and  $p(\mathbf{x}; \boldsymbol{\theta})$  is the policy distribution or policy network.

## §6.5 Baselines

Next we will discuss how we can modify policy gradient calculation to reduce its variance, and in this way obtain a version of the policy gradient that can be used as a practical reinforcement learning algorithm. It turns out that we can show that subtracting a constant  $b$  from our rewards in policy gradient will not actually change the gradient in expectation, although it will change its variance. Meaning that doing this trick will keep our gradient estimator unbiased. So we are going to use the same convenient identity from before:

$$\mathbb{E}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) b] = \int p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) b d\tau = \int \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau) \log p_{\boldsymbol{\theta}}(\tau) b d\tau = b \nabla_{\boldsymbol{\theta}} \int p_{\boldsymbol{\theta}}(\tau) d\tau = b \nabla_{\boldsymbol{\theta}} \cdot 1 = 0$$

This means that subtracting  $b$  will keep our policy gradient unbiased but it will actually alter its variance.

The policy gradient theorem can be generalized to include a comparison of the action value to an arbitrary *baseline*  $b(s)$  without changing it in expectation:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[ \sum_{t=0}^T \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t | s_t) \left( \sum_{t'=t}^T R(s_{t'}, a_{t'}, s_{s'+1}) - b(t) \right) \right]$$

The baseline can be any function, even a random variable, as long as it does not vary with  $a$ ; the equation remains valid because the subtracted quantity is zero.

The most common choice for baseline is the on-policy value function  $V^\pi(s_t)$ . Recall that this is the average return an agent gets if it starts in state  $s_t$  and then acts according to policy  $\pi$  for the rest of its action. Empirically, the choice of  $b(s_t) = V^\pi(s_t)$  has the desirable effect of reducing variance in the sample estimate for the policy gradient. This results in faster and more stable policy learning. It is also appealing from a conceptual angle: it encodes the intuition that an agent gets what it expects, it should “feel” neutral about it.

In practice  $V^\pi(s_t)$  cannot be computed exactly, so it has to be approximated. This is usually done with a neural network  $V_\phi(s_t)$ , which is updated concurrently with the policy (so that the value network always approximates the value function of the most recent policy).

The simplest method for learning  $V_\phi$  used in most implementations of policy optimization algorithms is to minimize the mean-square-error objective:

$$\phi_k = \arg \min_{\phi} \mathbb{E}_{s_t, \hat{R}_t \sim \pi_k} \left[ \left( V_\phi(s_t) - \hat{R}_t \right)^2 \right]$$

where  $\pi_k$  is the policy at epoch  $k$ .

## §6.6 Generalizing Policy Gradients

What we have seen so far is that the policy gradient has the general form

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \Phi_t \right],$$

where  $\Phi_t$  could be

$$\Phi_t = R(\tau),$$

or

$$\Phi_t = \sum_{t'=t}^T R(s_{t'}, a_{t'}, s_{t'+1}),$$

or

$$\Phi_t = \sum_{t'=t}^T R(s_{t'}, a_{t'}, s_{t'+1}) - b(t)$$

All of these choices lead to the same expected value of the policy gradient, despite having different variances. There are two more valid choices of weights  $\Phi_t$  which are important.

**On-Policy Action Value Function.** The choice

$$\Phi_t = Q^{\pi_{\theta}}(s_t, a_t)$$

is also valid.

**Advantage Function.** Recall that the advantage of an action is defined by  $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ , which describes how much better or worse an action is than other actions on average (relative to current policy). The formulation of policy gradients with advantage functions is extremely common, and there are many different ways of estimating advantage function used in different algorithms.

## §7 Advantage Actor-Critic

In the domain of Reinforcement Learning (RL), one-step return in Temporal Difference (TD) learning has demonstrated a favorable trade-off. Although it introduces a bias, its variance is often significantly reduced compared to the actual return. The **Actor-Critic** paradigm combines the two concepts in RL: policy gradients and value functions. This methodology is characterized by its dual components that are trained in tandem:

**Algorithm 3** Vanilla Policy Gradient (VPG)**Require:** Initial policy parameter  $\theta_0$ , initial value function parameters  $\phi_0$ 

- 1: **for**  $k = 1, 2, \dots$  **do**
- 2:   Collect a set of trajectories  $\mathcal{D}_k = \{\tau_i\}$  by executing policy  $\pi_k = \pi_{\theta_k}$  in the environment.
- 3:   Compute the rewards-to-go  $\hat{R}_t$ .
- 4:   Compute advantage estimates,  $\hat{A}_t$  based on current value function  $V_{\phi_k}$ .
- 5:   Estimate the policy gradient:

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \hat{A}_t$$

- 6:   Update the policy parameters using the policy gradient:

$$\theta_{k+1} \leftarrow \theta_k + \alpha_k \hat{g}_k$$

- 7:   Fit value function by regression:

$$\phi_{k+1} =$$

- 8: **end for**

- The *actor*: Responsible for learning a parameterized policy that dictates the agent's behavior.
- The *critic*: Tasked with estimating the value of state-action pairs, offering a gauge for the decisions made by the actor.

One of the driving forces behind the actor-critic design is the insight that a well-trained critic can provide a more informative feedback signal to the actor than the raw, sometimes sparse, rewards that can be obtained directly from the environment<sup>7</sup>.

The policy gradient theorem tells us that we can improve this policy by ascending the gradient of the expected cumulative reward. In REINFORCE, this gradient is estimated using Monte Carlo samples of the return, and the policy update rule is as follows:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a | s) R(\tau)$$

In Advantage Actor-Critic methods, we instead use the advantage function  $A(s, a)$  as a reinforcing signal. The advantage function measures how much better an action  $a$  is compared to the average action in state  $s$ . Therefore, the update rule in Advantage Actor-Critic methods is:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a | s) A(s, a)$$

The Advantage function  $A(s, a)$  is defined as the difference between the Q-value of taking action  $a$  in state  $s$  and the V-value of state  $s$ :

$$A(s, a) = Q(s, a) - V(s)$$

★ REINFORCE solely relies on the total reward from entire trajectories, whereas Advantage Actor-Critic refines this approach by using the advantage function to inform its updates

$$\text{REINFORCE: } \theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a | s) R(\tau)$$

$$\text{Advantage Actor-Critic: } \theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(a | s) A(s, a)$$

The central idea behind actor-critic methods is that these two components, the actor and the critic, work together to find the optimal policy. The critic informs the actor how to update its policy, by evaluating the chosen actions, and the actor uses this feedback to make better-informed actions in the future.

<sup>7</sup>This suggests that the internal evaluation of decisions, via the critic, might expedite and refine the learning process of the agent's behavior, compared to merely relying on external feedback.

## §7.1 The Advantage Function

It's worth understanding that the advantage function has a number of useful properties that are quintessential for grasping its significance in reinforcement learning.

1. **Expectation of Zero:** Consider the mathematical relation:

$$\mathbb{E}_{a \in A}[A^\pi(s_t, a)] = 0.$$

This conveys that when all possible actions possess equivalent reward, the value of  $A^\pi$  will be zero across all actions. In the process of training policies using  $A^\pi$ , this ensures that the likelihood of choosing any action remains invariant. Compare this to reward signals rooted in absolute state or state-action values in equivalent scenarios might have a constant uniform value, it would actively encourage (if positive) or discourage (if negative) the action taken.

2. **Intuitive Feedback:** Envision a circumstance where the selected action is inferior compared to the average, yet still anticipates a positive return; that is,

$$Q^\pi(s_t, a_t) > 0 \text{ but } A^\pi(s_t, a_t) < 0.$$

The logical inference here is that the likelihood of such an action should decrease due to the presence of superior alternatives. In this context, feedback from  $A^\pi(s_t, a_t)$  closely aligns with our intuition, decreasing the probability of action. However, utilizing  $V^\pi$  or  $Q^\pi$  in conjunction with a baseline might paradoxically amplify its likelihood due to the positive return.

3. **Comparative Evaluation:** The advantage function is a relative measure. For a specific state  $s$  and its corresponding action  $a$ , it juxtaposes the value of the state-action duo,  $Q^\pi(s, a)$ , against the value of the state,  $V^\pi(s)$ . This paradigm is instrumental in ensuring actions are neither unduly penalized for the policy's current unfavorable state nor excessively rewarded for an advantageous one. This perspective recognizes that an action predominantly sways future trajectories, without influencing the preceding sequence of events that resulted the policy into its current state.

## §7.2 Estimating Advantage

To compute the advantage function  $A^\pi$ , estimates for  $Q^\pi$  and  $V^\pi$  are essential. One approach could be to separately train different neural networks to learn  $Q^\pi$  and  $V^\pi$ . However, this approach presents two main drawbacks: (1) the estimates might not align well with each other, and (2) the learning process becomes less efficient. Therefore, it is more common to learn  $V^\pi$  and then use it in conjunction with rewards from a trajectory to approximate  $Q^\pi$ .

There are two primary reasons why learning  $V^\pi$  is generally more advantageous than learning  $Q^\pi$ . First,  $Q^\pi$  is a more complex function, which requires more samples for an accurate estimation. This can become an issue, especially in scenarios where both the actor and the critic are being trained simultaneously. Second, computing  $V^\pi$  from  $Q^\pi$  can be computationally demanding. To obtain  $V^\pi$  from  $Q^\pi$ , one must calculate the values for all possible actions in a given state  $s$ , and then compute an action-probability weighted average.

Let's examine how to approximate  $Q^\pi$  using  $V^\pi$ . The  $Q$  function, representing the expected return of executing action  $a$  in state  $s$  and subsequently following policy  $\pi$ , can be formulated as a combination of the immediate reward and the expected future rewards, as given by the value function. Specifically, the  $Q$  function can be decomposed into the immediate reward  $r(s, a)$  accrued by taking action  $a$  in state  $s$ , in addition to the discounted sum of expected future rewards under policy  $\pi$ .

$$Q^\pi(s_t, a_t) = r(s, a) + \gamma \mathbb{E}_{\tau \sim \pi}[V^\pi(s)]$$

If we assume for a moment that we have a perfect estimate of the  $V^\pi(s)$ , then the  $Q$ -function

can be rewritten as a mix of the expected rewards for  $n$  time steps, followed by  $V^\pi(s_{n+1})$ :

$$\begin{aligned} Q^\pi(s_t, a_t) &= \mathbb{E}_{\tau \sim \pi}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_{t+n}] \gamma^{n+1} [V^\pi(s_{t+n+1})] \\ &\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_{t+n} + \hat{V}^\pi(s_{t+n+1}) \end{aligned}$$

We employ a single trajectory of rewards  $(r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_{t+n})$  as a substitute for the expectation, and insert  $\hat{V}^\pi$ , which is learned by the critic. This equation balances bias against variance. The  $n$ -step actual rewards are unbiased but exhibit high variance since they are derived from a single trajectory. In contrast,  $\hat{V}^\pi$  has lower variance as it averages over all trajectories observed thus far, but it is biased due to its reliance on a function approximator.

The rationale for combining these two kinds of estimates lies in the typical behavior of actual rewards' variance as the time steps away from  $t$  increase. Near the time  $t$ , the benefits of an unbiased estimate may outweigh the variance introduced. As  $n$  increases, the variance is likely to become increasingly problematic, making it more effective to switch to an estimate with lower variance but some bias. The number of steps of actual rewards, denoted by  $n$ , controls this trade-off between bias and variance.

By combining the  $n$ -step estimate for  $Q^\pi$  with  $\hat{V}^\pi$ , we arrive at a formula for approximating the advantage function:

$$\begin{aligned} A_{\text{Nstep}}^\pi &= Q^\pi(s_t, a_t) - V^\pi(s_t) \\ &\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n r_{t+n} + \hat{V}^\pi(s_{t+n+1}) - \hat{V}^\pi(s_t) \end{aligned}$$

The number of steps of actual rewards, denoted by  $n$ , modulates the variance in the advantage estimator and serves as a hyperparameter that requires tuning. A smaller  $n$  yields an estimator with lower variance but higher bias, while a larger  $n$  produces an estimator with increased variance but reduced bias.

### §7.3 Generalized Advantage Estimation

Reinforcement learning algorithms, while powerful, are often plagued with the challenge of variance in their estimates. In particular, the advantage function, which measures the relative quality of an action taken in a state compared to the average action, is susceptible to high variance, especially when based on the  $n$ -step return estimate.

Generalized Advantage Estimation (GAE), proposed by Schulman et al., [?] addresses the dilemma of choosing a fixed number of steps,  $n$ , for the return estimate. Instead of constraining ourselves to a singular choice of  $n$ , GAE ingeniously takes a weighted average of advantages calculated over different values of  $n$  ranging from 1 to  $k$ . This approach aims to achieve a balance: reduce the variance substantially while keeping the introduced bias minimal.

At a high level, one can intuitively think of GAE as an estimator that leverages information from multiple temporal horizons. By not committing to a fixed horizon, it captures a broader perspective of the environment dynamics and policy performance.

**TD Error:** Temporal Difference (TD) error is a measure of the mismatch between the estimated value of a state and the observed return. It's given by:

$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \quad (19)$$

where  $r_t$  is the reward at time  $t$  and  $\gamma$  is the discount factor.

**Generalized Advantage Estimation:** The main idea behind GAE is to compute a weighted average of the  $n$ -step truncated advantage estimators. This can help in reducing the variance and stabilizing the training. The generalized advantage estimator is given by:

$$\begin{aligned} A_t^{GAE(\gamma, \lambda)} &= (1 - \lambda)(A_t^1 + \lambda A_t^2 + \lambda^2 A_t^3 + \dots) \\ &= \delta_t + (\gamma \lambda) \delta_{t+1} + (\gamma \lambda)^2 \delta_{t+2} + \dots \end{aligned} \quad (20)$$

where  $\lambda$  is a hyperparameter that determines the trade-off between bias and variance of the



advantage estimates. Using the TD errors, this estimator can be computed in a recursive manner:

$$A_t^{GAE(\gamma, \lambda)} = \delta_t + \gamma \lambda A_{t+1}^{GAE(\gamma, \lambda)} \quad (21)$$

With GAE, one can smoothly interpolate between the one-step TD error (high variance, low bias) and the full Monte Carlo return (low variance, high bias) by adjusting the value of  $\lambda$ .

## §8 Advanced Policy Gradient Methods

Policy gradient methods, while powerful, often face the risk of performance collapse. This issue is particularly prevalent in on-policy algorithms, making them sample-inefficient due to their inability to reuse data. To counteract these challenges, Proximal Policy Optimization (PPO) was introduced by Schulman et al. [20].

PPO aims to ensure stable and consistent policy improvement through a *surrogate objective function*. This function not only prevents abrupt performance degradation but also facilitates the use of off-policy data. By modifying the original objective  $J(\theta)$  with the PPO objective, methods like REINFORCE or Actor-Critic can achieve more stable and sample-efficient training.

### §8.1 Performance Collapse and Surrogate Objective

Given a policy  $\pi_\theta$ , its optimization involves updating the policy parameters  $\theta$  based on the policy gradient  $\nabla_\theta J(\theta)$ . This method is considered indirect since it seeks the optimal policy in the policy space which we do not have direct control over. The underlying reason for this indirectness can be better understood by distinguishing between policy and parameter spaces.

During the optimization process, we traverse a sequence of policies  $\pi_1, \pi_2, \pi_3, \dots, \pi_n$  within the set of all possible policies, known as the policy space  $\Pi$ . It's worth noting that  $\Pi$  can contain an uncountably infinite number of policies<sup>8</sup>:

$$\Pi = \{\pi_i : i \in \mathbb{R}\}$$

When the policy is parameterized as  $\pi_\theta$ , we can define the parameter space for  $\theta$ . Every distinct  $\theta$  parameterizes a unique instance of the policy. The parameter space is given by:

$$\Theta = \{\theta : \theta \in \mathbb{R}^m\}$$

While the objective  $J(\pi_\theta)$  is determined using trajectories produced by a policy in the policy space,  $\pi_\theta \in \Pi$ , the search for the optimal policy primarily takes place in the parameter space by identifying the right parameters,  $\theta \in \Theta$ . Essentially, our control is in  $\Theta$ , but the outcome is in  $\Pi$ . In practice, the magnitude of parameter updates is controlled using a learning rate  $\alpha$ :

$$\Delta\theta = \alpha \nabla_\theta J(\pi_\theta)$$

However, a key challenge arises as the mappings between the policy space and the parameter space might not always align congruently. Even if two pairs of parameters, such as  $\theta_1, \theta_2$  and  $\theta_2, \theta_3$ , have equivalent distances in the parameter space, their corresponding policies might not have the same distances in the policy space. More formally, given the equality  $d(\theta_1, \theta_2) = d(\theta_2, \theta_3)$ , it doesn't necessarily imply that:

$$d_\theta(\pi_{\theta_1}, \pi_{\theta_2}) = d_\theta(\pi_{\theta_2}, \pi_{\theta_3}) \Leftrightarrow d_\pi(\pi_{\theta_1}, \pi_{\theta_2}) = d_\pi(\pi_{\theta_2}, \pi_{\theta_3})$$

The non-congruent mapping between the policy space  $\Pi$  and the parameter space  $\Theta$  presents a potential problem. Specifically, the optimal learning rate  $\alpha$  can fluctuate based on the location of the current policy  $\pi_\theta$  within  $\Pi$  and how the current parameter  $\theta$  relates to the vicinity of  $\pi_\theta$ . In an ideal scenario, an algorithm would adaptively adjust the step size in the parameter space contingent on these dynamics. To derive an adaptive step size based on how a particular

<sup>8</sup>This implies that policies can be indexed by real numbers, i.e.,  $i \in \mathbb{R}$

update in parameter space will affect policy space, we first need a way to measure the difference in performance between two policies.

## §8.2 Monotonic Improvement Theory

In reinforcement learning optimization, a crucial goal is to ensure non-decreasing policy performance. To achieve this, we define the measure of performance differences between two policies, thereby enhancing the policy gradient objective for monotonic improvement.

Addressing step size is pivotal. By constraining the step size, we mitigate risks of performance degradation, leading to the **Monotonic Improvement Theory**.

Given an existing policy  $\pi$  and its subsequent iteration  $\pi'$ , the *relative policy performance identity* quantifies the difference in their objectives.

**Theorem 8.1** (Relative Policy Performance Identity) — For any policies  $\pi$  and  $\pi'$ :

$$J(\pi') - J(\pi) = \sum_{t=0}^{\infty} \mathbb{E}_{\tau \sim \pi'} [\gamma^t A_{\pi}(s_t, a_t)]$$

*Proof.*

$$\begin{aligned} J(\pi') - J(\pi) &= \mathbb{E}_{\tau \sim \pi'} \left[ \sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right] \\ &= \mathbb{E}_{\tau \sim \pi', s_{t+1}, r_t \sim p(\cdot | s_t, a_t), a_t \sim \pi'} \left[ \sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right] \\ &= \mathbb{E}_{\tau \sim \pi'} \left[ \sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \gamma V_{\pi}(s_{t+1}) - V_{\pi}(s_t)) \right] \\ &= \mathbb{E}_{\tau \sim \pi'} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \sum_{t=0}^{\infty} \gamma + t + 1 V_{\pi}(s_{t+1}) - \sum_{t=0}^{\infty} V_{\pi}(s_t) \right] \\ &= \mathbb{E}_{\tau \sim \pi'} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \right] + \mathbb{E}_{\tau \sim \pi'} \left[ \sum_{t=0}^{\infty} \gamma + t + 1 V_{\pi}(s_{t+1}) - \sum_{t=0}^{\infty} V_{\pi}(s_t) \right] \\ &= J(\pi') + \mathbb{E}_{\tau \sim \pi'} \left[ \sum_{t=0}^{\infty} \gamma + t + 1 V_{\pi}(s_{t+1}) - \sum_{t=0}^{\infty} V_{\pi}(s_t) \right] \\ &= J(\pi') - V_{\pi}(s_0) \\ &= J(\pi') - J(\pi) \end{aligned}$$

□

The relative policy performance identity,  $J(\pi') - J(\pi)$ , quantifies policy improvement. A positive difference indicates the improvement of the new policy  $\pi'$  over  $\pi$ . In policy iteration, the goal is to select a  $\pi'$  that maximizes this difference. Hence, maximizing  $J(\pi'_\theta)$  is equivalent to maximizing this identity:

$$\max_{\pi'} J(\pi') \Leftrightarrow \max_{\pi'} (J(\pi') - J(\pi))$$

This perspective underscores the importance of ensuring non-negative (monotonic) improvements in each policy iteration, i.e.,  $J(\pi') - J(\pi) \geq 0$ . In the worst-case scenario, choosing  $\pi' = \pi$  yields no improvement. This formulation prevents performance collapses during training, which achieves the desired robustness in policy optimization.

Given the expression  $\mathbb{E}_{\tau \sim \pi'} [\sum_t^{\infty} A(s_t, a_t)]$ , the expectation requires sampling trajectories from the new policy  $\pi'$  for an update. However,  $\pi'$  is only available after the update. We need an approach that leverages the available old policy,  $\pi$ .

By assuming that successive policies,  $\pi$  and  $\pi'$ , are close (in terms of KL divergence), we infer that their state distributions are analogous. Thus, the relative policy performance identity in 8.1

can be approximated using trajectories from the old policy,  $\tau \sim \pi$ , and adjusted by importance sampling weights<sup>9</sup> given by  $\frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)}$ . This effectively adjusts the returns generated under  $\pi$  by the ratio of action probabilities between  $\pi$  and  $\pi'$ . The resulting approximation is termed the **surrogate objective**:

$$\begin{aligned} J(\pi') - J(\pi) &= \mathbb{E}_{\tau \sim \pi} \left[ \sum_t A^\pi(s_t, a_t) \right] \\ &\approx \mathbb{E}_{\tau \sim \pi} \left[ \sum_t A^\pi(s_t, a_t) \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)} \right] \\ &= J_\pi^{\text{CPI}}(\pi') \end{aligned}$$

We call  $J_\pi^{\text{CPI}}(\pi')$  the surrogate objective because it contains a ratio of new and old policies,  $\pi'$  and  $\pi$ , and the superscript CPI stand for “conservative policy iteration”.

Before using  $J_\pi^{\text{CPI}}(\pi')$  as the objective for the policy gradient algorithm, there is one final requirement to satisfy. Specifically,  $J_\pi^{\text{CPI}}(\pi')$  serves as an estimate for  $J(\pi') - J(\pi)$ , hence it inherently contains some error. For the robustness of our approach, it’s essential to ensure  $J(\pi') - J(\pi) \geq 0$  when utilizing the approximation  $J_\pi^{\text{CPI}}(\pi') \approx J(\pi') - J(\pi)$ . This necessitates a deeper understanding of the approximation error.

When successive policies  $\pi$  and  $\pi'$  are proximate, as measured by their KL divergence, one can write a relative policy performance bound. The absolute error, defined as the difference between the actual objective  $J(\pi')$  and the anticipated improvement  $J_\pi^{\text{CPI}}(\pi')$ , can be constrained with respect to the KL divergence of  $\pi$  and  $\pi'$ . This is introduced in the paper “Constrained Policy Optimization” by Achiam et al. (2017) [22]:

$$|J(\pi') - J(\pi) - J_\pi^{\text{CPI}}(\pi')| \leq C \sqrt{E_t[KL(\pi'(a_t|s_t)||\pi(a_t|s_t))]} \quad (22)$$

Here,  $C$  is a constant and  $KL$  denotes the Kullback-Leibler divergence. The above inequality emphasizes that when  $\pi$  and  $\pi'$  are close in their distributions, KL divergence on the right hand side is low, and the approximation error of  $J_\pi^{\text{CPI}}(\pi')$  for  $J(\pi') - J(\pi)$  is minimal.

### §8.3 Trust Region Policy Optimization Problem

For optimization, one approach is to impose a constraint on the expectation of the KL divergence:

$$E_t[KL(\pi'(a_t|s_t)||\pi(a_t|s_t))] \leq \delta$$

Here,  $\delta$  bounds the KL divergence, constraining how much the new policy  $\pi'$  can deviate from the old policy  $\pi$ . This ensures the selection of candidate policies close to  $\pi$  in the policy space, defining a *trust region*. The constraint  $E_t[KL(\pi'(a_t|s_t)||\pi(a_t|s_t))]$  is termed the trust region constraint. Notably, this constraint operates on an expectation over a singular time step  $t$ .

Expressing the objective  $J_\pi^{\text{CPI}}(\pi')$  with this perspective:

$$J_\pi^{\text{CPI}}(\pi') = \mathbb{E}_t \left[ \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)} A^\pi(s_t, a_t) \right]$$

And, given that the objective maximization is with respect to  $\theta$ , the policies are represented in terms of  $\theta$ :

$$J^{\text{CPI}}(\pi') = \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A^{\pi_{\theta_{old}}}(s_t, a_t) \right]$$

Combining the surrogate objective with the trust region constraint, the optimization problem can be written as:

<sup>9</sup>Importance sampling estimates a desired distribution using samples from a different, known distribution.

$$\max_{\theta} \mathbb{E}_t \left[ \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A^{\pi_{\theta_{old}}}(s_t, a_t) \right] \quad \text{subject to } \mathbb{E}_t[\text{KL}(\pi_{\theta}(a_t|s_t) || \pi_{\theta_{old}}(a_t|s_t))] \leq \delta \quad (23)$$

It's worth noting that  $J_{\pi}^{\text{CPI}}(\pi')$  is a linear approximation to  $J(\pi') - J(\pi)$  given that its gradient matches the policy gradient. It also guarantees monotonic improvement to within an error bound. To account for potential errors and to ensure continued improvement, the trust region constraint is imposed. By ensuring policy changes remain within this trust region, we can overcome significant performance degradation [22].

Several algorithms aim to address this trust region optimization challenge, notable among them are Natural Policy Gradient (NPG) [23], Trust Region Policy Optimization (TRPO) [24], and Constrained Policy Optimization (CPO) [22].

The theoretical foundations behind these algorithms are intricate, compounded by the challenge that these algorithms are hard to implement. Furthermore, computing their gradients can be computationally taxing. Additionally, selecting an optimal value for  $\delta$  proves to be non-trivial. These algorithms are beyond the scope of this project. However, their inherent challenges underscore the motivation for our next algorithm.

## §8.4 Proximal Policy Optimization

The Proximal Policy Optimization Algorithms (PPO) was introduced by Schulman et al. in 2017 [20]. With its ease of implementation, computational efficiency, and the elimination of the need to select  $\delta$ , PPO has become one of the most popular policy gradient algorithms.

PPO is a family of algorithms designed to address the trust-region constrained policy optimization problem through efficient heuristics. Two primary variants exist: one based on adaptive KL penalties and the other on a clipped objective. To delve deeper, let's simplify the surrogate objective  $J_{\pi}^{\text{CPI}}(\pi)$ . Define

$$r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

and denote  $A^{\pi_{\theta_{old}}}$  as  $A_t$  for conciseness, since we know advantages are always calculated using the older policy. This allows us to express the objective as:

$$J^{\text{CPI}}(\pi) = \mathbb{E}_t \left[ \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A^{\pi_{\theta_{old}}}(s_t, a_t) \right] = \mathbb{E}_t[r_t(\theta)A_t]$$

### 8.4.1 Adaptive KL Penalty Algorithm

The first PPO variant employs an adaptive KL penalty, described by the objective:

$$\max_{\theta} \mathbb{E}_t[r_t(\theta)A_t] - \beta \text{KL}(\pi_{\theta}(a_t|s_t) || \pi_{\theta_{old}}(a_t|s_t)) \quad (24)$$

Here,  $\beta$  is an adaptive coefficient dictating the magnitude of the KL penalty. A higher  $\beta$  value results in a more pronounced difference between  $\pi_{\theta}$  and  $\pi_{\theta_{old}}$ . A primary challenge lies in the selection of a constant coefficient, as different problems exhibit distinct characteristics, finding a universal  $\beta$  is challenging. Moreover, within a single problem, the loss landscape undergoes changes with policy iterations. Thus, a  $\beta$  value that was efficient initially might prove sub-optimal later on. To address this, the PPO algorithm introduces a heuristic-based rule for dynamically updating  $\beta$ . After each policy update,  $\beta$  is recalibrated, ensuring its applicability for the subsequent iteration.

### 8.4.2 PPO with Clipped Surrogate Objectives

The *PPO-Clip* variant of PPO modifies the surrogate objective without relying on the KL constraint. The objective is defined as:

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

Where:

- $\hat{\mathbb{E}}_t$  is the empirical expectation over timesteps.
- $r_t(\theta)$  represents the probability ratio of new to old policies.
- $\hat{A}_t$  is the advantage estimate at time  $t$ .
- $\epsilon$  is a small hyperparameter, typically set to 0.1 or 0.2.

The clip operation ensures the ratio  $r_t(\theta)$  is within the interval  $[1 - \epsilon, 1 + \epsilon]$ . This objective controls policy updates to ensure stability. The objective function includes two terms, and optimization attempts to maximize the minimum of these two terms. The first,  $r_t(\theta)\hat{A}_t$ , is the conventional policy gradient objective. The second,  $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t$ , is a clipped variant, constraining policy deviations if the ratio  $r_t(\theta)$  exits  $[1 - \epsilon, 1 + \epsilon]$ .

## §9 Experiments

### §9.1 Optimization as Reinforcement Learning

Optimization is a critical area of study in mathematics with applications in many scientific and industrial domains. Traditional optimization techniques have played instrumental roles in diverse arenas. For instance, they've streamlined supply chain operations, enhanced financial portfolio returns, optimized energy production, and significantly improved machine learning models. These successes underscore the ubiquitous importance and effectiveness of optimization methodologies across various fields.

Inspired by seminal works such as Andrychowicz et al.'s [19] and Li and Malik's [18], this research explores the use of RL in function optimization, with the convex functions serving as a test cases. We explore automating the design of unconstrained optimization algorithms, which are some of the most powerful and ubiquitous tools in all areas of science and engineering. We view the the algorithmic design problem from the perspective of reinforcement learning, and we formulate the algorithmic design problem as a reinforcement learning problem. *Under this framework, any particular optimization algorithm just corresponds to a policy.* We reward optimization algorithms that converge quickly and penalize those that do not. Learning an optimization policy is then reduces to finding an optimal policy, which can be solved using any reinforcement learning method.

To initiate the optimization,  $x^{(0)}$  is chosen from the function's domain. At every successive iteration, a step vector,  $\Delta x$ , is derived based on a specific formula which serves as the means to adjust the current iterate. Mathematically, the step vector is often expressed as a function of past and current gradients of the objective function. In gradient descent, this vector is a scaled version of the negative gradient, whereas, in momentum-based approaches, it resembles a scaled exponential moving average of the gradients. A concise representation of this iterative procedure can be found in the following algorithm [18]:

---

#### Algorithm 4 Continuous Optimization Framework

---

```

1: Input: Objective function  $f$ 
2:  $x^{(0)} \leftarrow$  random point in the domain of  $f$ 
3: for  $i = 1, 2, \dots$  do
4:    $\Delta x \leftarrow \pi(\{x^{(j)}, f(x^{(j)}), \nabla f(x^{(j)})\}_{j=0}^{i-1})$ 
5:   if Stopping condition then return  $x^{(i-1)}$ 
6:   end if
7:    $x^{(i)} \leftarrow x^{(i-1)} + \Delta x$ 
8: end for
```

---

The presented framework encapsulates a broad spectrum of existing optimization algorithms. The uniqueness of each optimization algorithm is attributed to the distinct choice of the function  $\pi$ . First-order optimization methods use a  $\pi$  that's solely dependent on the gradients of the objective function. Second-order techniques integrate both the gradient and the Hessian of the

objective function to determine  $\pi$ . For instance, gradient descent can be exemplified by the subsequent selection of  $\pi$ :

$$\pi(\{x^{(j)}, f(x^{(j)}), \nabla f(x^{(j)})\}_{j=0}^{i-1}) = \gamma \nabla f(x^{(i-1)})$$

Here,  $\gamma$  represents the step size or, equivalently, the learning rate. Furthermore, extending this understanding to the gradient descent method with momentum, we deduce:

$$\pi(\{x^{(j)}, f(x^{(j)}), \nabla f(x^{(j)})\}_{j=0}^{i-1}) = \gamma \left( \sum_{j=0}^{i-1} \alpha^{i-1-j} \nabla f(x^{(j)}) \right)$$

In this context,  $\gamma$  retains its meaning as the step size, while  $\alpha$  is the momentum decay factor.

Therefore, learning the functional  $\pi$  effectively translates to learning the underlying optimization algorithm. However, since modeling general functionals is challenging, we restrict the dependence of  $\pi$  on the objective function  $f$  to the objective values and the gradients evaluated at the current and past locations. Hence,  $\pi$  can be modeled as a function from the objective values and gradients taken along the trajectory so far,  $\{x^{(j)}, f(x^{(j)}), \nabla f(x^{(j)})\}_{j=0}^{i-1}$ , to the step vector.

We observe that the execution of an optimization algorithm execution parallels the execution of a fixed policy within a Markov Decision Process (MDP). The state amalgamates the current location, objective values, and gradients spanning current to past locations. Meanwhile, the action corresponds to the step vector, and transition probabilities are partly defined by the location update equation,  $x^{(i)} \leftarrow x^{(i-1)} + \Delta x$ . Consequently, the policy  $\pi$  is synonymous with the MDP's fixed policy. For this reason, we will use  $\pi$  to denote the policy at hand. Under this formulation, searching over all policies corresponding to searching over all possible first-order optimization algorithms.

We model the update formula as a neural network. Therefore, by learning the weights of the neural network, we can learn an optimization algorithm. Parameterize the update formula as a neural network has two appealing properties:

- Universality in functional approximation due to the inherent expressiveness of neural networks.
- Efficiency in search operations due to the amenability of neural networks to backpropagation.

Our approach uses on reinforcement learning to determine the policy  $\pi$ . For optimization algorithms, the key performance metric is convergence speed. To penalize policies that converge slowly or show undesirable behaviors, the reward for a given state is set as the negative of the objective value at the current location, thus promoting faster attainment of the function's minimum.

To do so, we need to define a reward function, which should penalize policies that exhibit undesirable behaviour during their executing. For optimization algorithms, the key performance metric is convergence speed. To penalize policies that converge slowly or show undesirable behaviors, the reward for a given state is set as the negative of the objective value at the current location. This encourages the policy to reach the minimum of the objective function as quickly as possible.

## §9.2 Optimization of single-variable continuous functions

We begin by investigating the an RL approach for optimization of single-variable continuous functions. We employ the Policy Gradient method, REINFORCE, to learn an optimal policy function for minimizing. We tested the model on a simple quadratic function  $f(x) = x^2$ .

The **state** consists of:

- Current point  $x$  in the domain
- A history of previous values
- Gradients at the previous points

**Actions and Rewards:** An action is a proposed step in the domain of the function. After each action, the reward is given as the negative value of the function at the new point, prompting the agent to find the minimum.

**Policy Model:** The agent’s policy is represented using a neural network, `PolciyNetwork`, which takes the environment state as input and outputs the mean of a normal distribution. The input layer size corresponds to the dimensions of the state (3 in this case). The network has a hidden layer of 128 neurons with ReLU activation, and the output layer consists of 1 neuron with a tanh activation function, representing the mean of the proposed normal distribution for action sampling. The actual action is sampled from this distribution. The agent was trained over 1500 episodes on the function  $f(x) = x^2$ , initialized at a random point between  $-1$  and  $1$ . The learning rate was set to  $1 \times 10^{-3}$  and the Adam optimizer was used for policy updates. The cumulative reward for episode was recorded to monitor the agents performance.

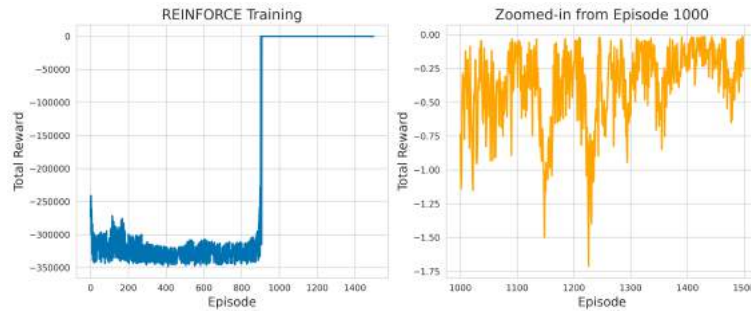


Figure 3: Optimization of a single-variable continuous function. This diagram displays the same reward function at two different scales. In the first diagram, it appears that the agent achieves the optimal reward instantly. However, when we adjust the scale of the y-axis in the second diagram, we observe variability in the agent’s reward.

Visual inspection of the plotted results in Figure 3 indicates that over time the agent is learning to move in the direction the minimum point of the function. One interesting observation is that initially the agent achieves a very poor total reward (i.e., the agent takes steps that move away from the minimum, since by definition the reward is negative value of the function), and after some number of episodes, the begins to perform much “better”, relatively speaking. We noticed in our training was always this relative spike in the performance of the agent, however, the exact number of episodes it took for the agent to learn a policy that could achieve this performance spike was very sparse. Our hypothesis for this behaviour is, the agent’s trajectories are significantly influenced by its past trajectories. And our policy (neural network) updates its parameters based on those trajectories. We hypothesize that, in the ‘early’ phase of training, which in the above diagram would be the first 800 episodes, the agent might be stuck in a negative feedback loop where it keeps generating “bad” trajectories and uses these “bad trajectories” to update its parameters in a non-optimal way.

We compared our learned policy to classic gradient descent. We can see in Figure 4 that RL agent achieves function values comparable to those of gradient descent. While gradient descent steadily approaches and stabilizes at the function’s minimum, the RL agent exhibits oscillatory behavior, never truly converging. There are many reasons why the agent exhibits the behaviour, most likely explanation is that we are using Gaussian policy parameterized by neural network with a constant variance term. There remains some irreducible variance due to the design choices we made for the policy. This can be easily remedied applying by various techniques such as decaying the step size, decaying the variance of the gaussian policy based on the number of iterations or even treating the variance,  $\sigma^2$ , as a learnable parameter which our policy outputs. Our experiment highlights both the strengths and challenges of using Reinforcement Learning for optimization tasks. It’s evident that while RL could potentially serve as a tool for optimization, it may not always be the most practical approach. Traditional optimization techniques like gradient descent are much simpler to use, as they require no training. However, it is possible that in complex landscapes where traditional methods underperform, our RL methodology for learning an optimization algorithm could be applied and over perform relative to traditional optimization



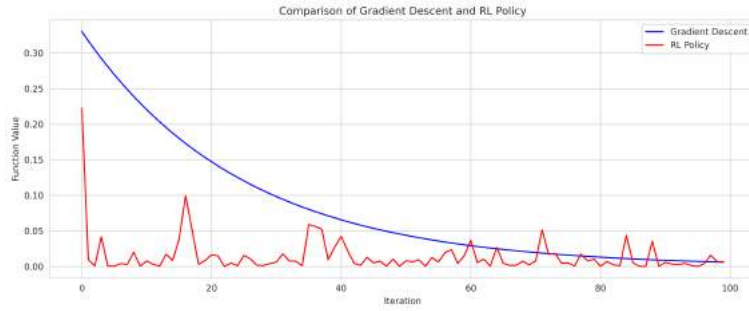


Figure 4: Performance comparison between Gradient Descent and the RL-based optimization policy over 100 iterations.

algorithms.

### 9.2.1 Direct Jump Strategy in Optimization

In the context of Reinforcement Learning, the agent’s strategy of directly jumping towards the optimal point is of particular interest. The “direct jump” strategy can be succinctly described by the following action:

$$a = x^* - x_i$$

where:

- $x^*$  represents the point where the function  $f(x)$  achieves its minimum.
- $x_i$  is the current point in the optimization process.
- $a$  is the action or step taken towards the optimal point.

For convex functions, which possess a single global minimum, the direct jump strategy is highly effective. An illustrative example is the function:

$$f(x) = x^2$$

Here,  $x^* = 0$  is the global minimum. For any starting point  $x_i$ , moving in the direction given by  $x^* - x_i$  will lead towards this minimum. A prerequisite for this strategy’s effectiveness is the knowledge of  $x^*$ . If  $x^*$  is unknown, which is often the case in many real-world optimization problems, the strategy cannot be applied. Consider the function:

$$f(x) = x^2$$

The global minimum is at  $x^* = 0$ . For a starting point  $x_i = 3$ , the strategy suggests a step of  $x^* - x_i = -3$ , indicating movement to the left.

The aim of this analysis is to evaluate whether the trained policy in the reinforcement learning optimization environment has learned a strategy that closely resembles the direct jump strategy of taking steps according to the formula  $a = x^* - x_i$ . We select a range of starting points  $x_i$  and for each  $x_i$ , evaluate the predicted action from the trained policy. The predicted action is the mean of a normal distribution. The expected action for any  $x_i$  is calculated as  $a = x^* - x_i$ . We plot the expected action against the predicted mean action to visually inspect the closeness. If the mean predicted actions closely follow the line of expected actions, we can infer that the policy has learned a strategy resembling the direct jump.

To evaluate how well the policy has learned the direct jump strategy, we can use: Relative Distance and Root Mean Square Error (RMSE).

$$\text{Relative Distance} = \frac{1}{n} \sum_{i=1}^n \frac{|a_{\text{predicted},i} - a_{\text{expected},i}|}{|a_{\text{expected},i}| + \epsilon}$$



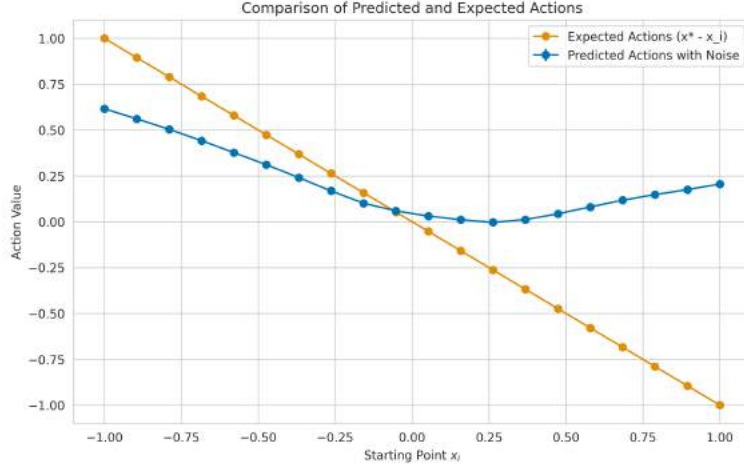


Figure 5: Performance comparison between Gradient Descent and the RL-based optimization policy over 100 iterations.

where  $n$  is the number of test points,  $a_{\text{predicted},i}$  and  $a_{\text{expected},i}$  are the predicted and expected actions for the  $i$ -th test point respectively, and  $\epsilon$  is a small constant to avoid division by zero.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (a_{\text{predicted},i} - a_{\text{expected},i})^2}$$

Upon testing the policy, the following metric values were obtained:

- **RMSE Value:** 0.5808
- **Relative Distance Value:** 0.8972

These values offer two distinct measures of how closely the policy approximates the direct jump strategy across the test points. However, it is difficult to make any conclusive statements about whether our agent has *actually* learned this strategy. It could be an interesting direction to explore what type of policy our agent actually learned. Further work could focus on methods for interpreting the learned policy.

### §9.3 Increasing Complexity of State Representation

In many RL scenarios, the current state of the environment encapsulates all the necessary information for the agent to make a decision. However, in certain situations, an agent's decision might benefit from considering historical data, i.e., states, actions, or rewards from previous time steps. This implies that this historical perspective can provide the agent with context, especially in environments where the current state might not capture all relevant dynamics.

Our experiment seeks to investigate this idea by examining the impact of varying lengths of historical data on the learning efficiency and effectiveness of a neural network-based policy.

We use a simple optimization environment where the agent's objective is to minimize a quadratic function. This environment, though elementary, serves as a controlled setup to understand the nuances of our investigation.

For our policy, we utilize a neural network that takes in the current state of the environment (the current point on the function) and outputs an action (a step direction and size to move along the function). The complexity arises when we augment the input to this network with historical data. Specifically, we experiment with different lengths of history, denoted by  $h$ , where  $h$  is the number of previous states (points and their respective function values) the network considers.

We train separate policies for each  $h$  value. Each policy is trained from scratch, ensuring no knowledge carryover between experiments. The training process involves the agent interacting with the environment, receiving rewards, and updating the policy to improve future decisions.

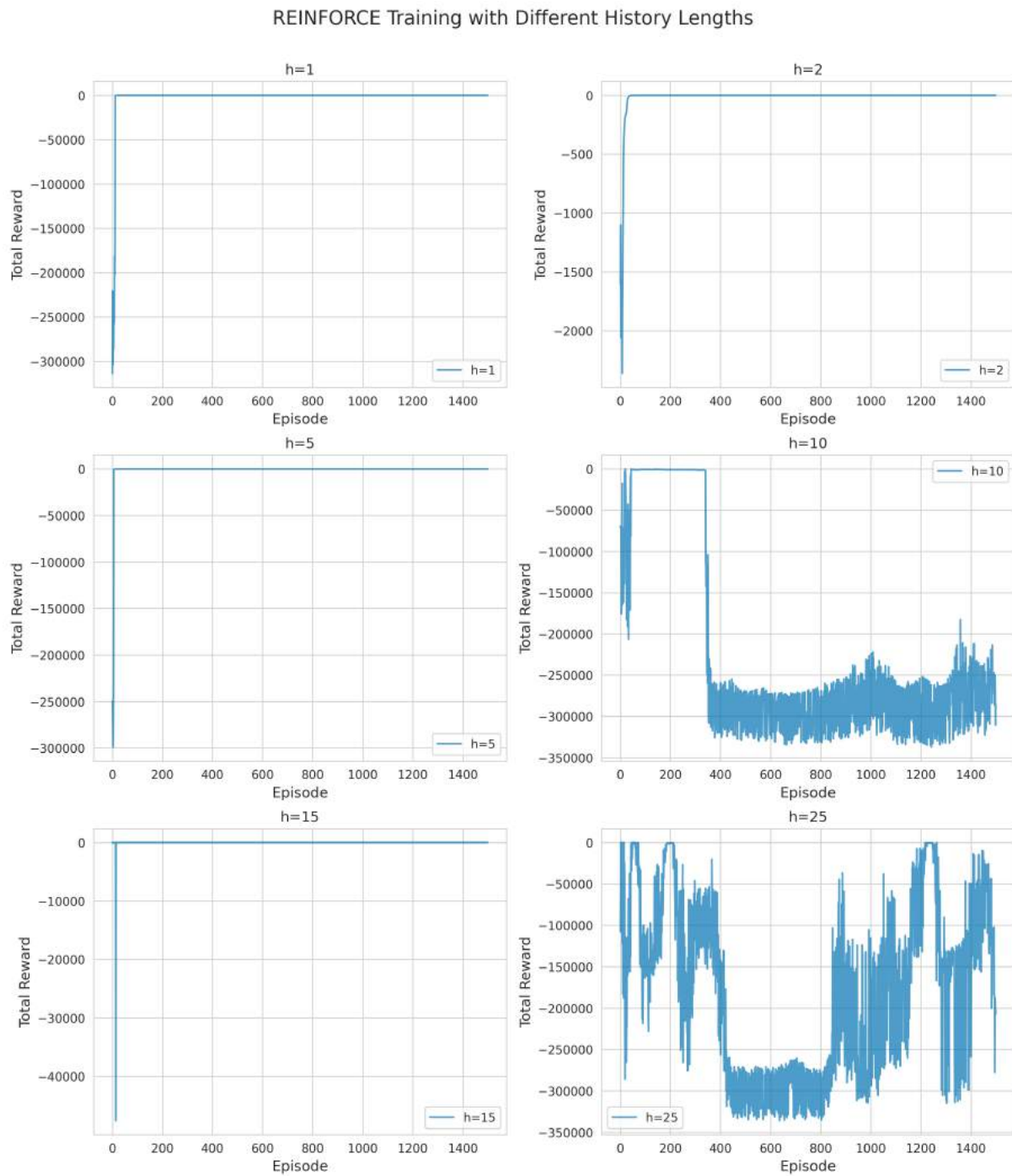


Figure 6: Optimization of single-variable continuous function

## §9.4 Reinforcement Learning for Linear Regression Optimization

Next we experiment employing reinforcement learning (RL) to optimize the parameters of a linear regression model. The goal is to leverage RL to find the best values for the coefficients that minimize the mean squared error (MSE) on synthetic data. Consider a simple linear regression model defined by:

$$y = wx + b$$

where  $y$  is the dependent variable,  $x$  is the independent variable, and  $w$  and  $b$  are the weight and bias respectively, which need optimization. Synthetic data was generated using the equation:

$$y = 2.0x - 3.0 + \text{noise}$$

where the noise follows a standard normal distribution. A total of 100 samples were generated. The environment, `LinearRegressionEnv`, is responsible for:

- Evaluating the current model using MSE.
- Updating the model parameters  $w$  and  $b$  based on the agent's action.
- Providing a reward, which is the negative of the MSE, as we aim to minimize it.

The agent's policy, `PolicyNetworkLinear`, is a neural network that takes the current state (values of  $w$  and  $b$ ) and outputs the changes  $\Delta w$  and  $\Delta b$ . The actions are sampled from a normal distribution centered around these outputs. The agent interacts with the environment for 500 episodes, where each episode consists of 100 steps. At each step, the agent:

- Observes the current state.
- Chooses an action based on its policy.
- Receives a reward from the environment.
- Adjusts its policy based on the reward using the REINFORCE algorithm.

The total reward for each episode, which is the cumulative negative MSE, is recorded. Upon training completion, the cumulative reward per episode was plotted. An upward trend in this graph indicates that the agent is gradually improving its policy to reduce the MSE over time.

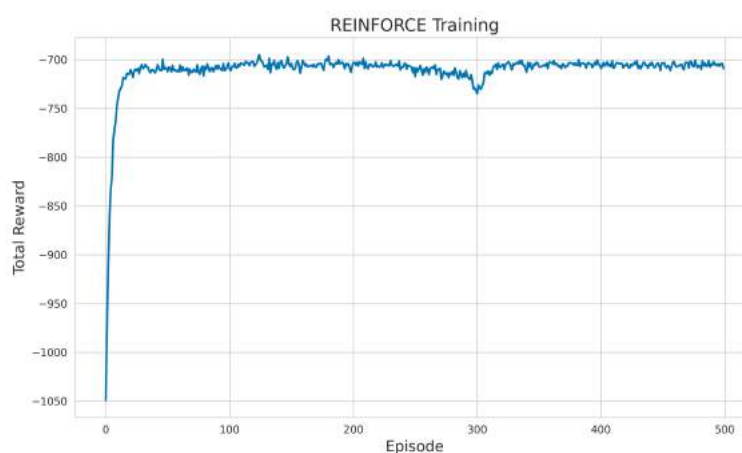


Figure 7: After looping through the environments we compute the average reward across all environments for that episode

One important limitation with this step is our agent has only been trained on one dataset. There is no guarantee that the agent's policy will *generalize* to a new linear regression problem. In the next section we discuss the idea of generalization more in depth.

## §9.5 Generalization

One of the main ideas of machine learning, regardless of domain, is to train on a finite set of examples and then generalize to a broader class from which these examples were drawn. In our specific context of developing optimization algorithms, it's valuable to outline what these 'examples' and 'class' signify.

**Contextualizing Objective Functions:** Imagine a scenario where generalization wasn't our priority. Here, we'd evaluate an optimizer on the identical objective functions that were used during its training. If we limited our training to a single objective function, an optimal optimizer would merely memorize the solution. Such an optimizer would always converge to the pre-memorized optimum in a single step, regardless of its starting point.

Expanding this viewpoint, when these objective functions act as loss functions for training other models, it leads into meta-learning. This paradigm of developing optimization algorithms, designed to improve the learning of base models, has been explored in seminal works like (Li & Malik, 2016) [18] and (Andrychowicz et al., 2016) [19]. A related exploration is found in (Bengio et al., 1991), which delved into learning Hebbian synaptic rules. Notably, while this rule took into account certain dimensions of the current iteration, it remained independent of the objective function, limiting its ability to generalize across varied objective functions.

**Motivating Multiple Environments:** Returning to our foundational concept, learning requires training on numerous instances to generalize across a wider class. In our pursuit of learning an optimizer for training base-models, this principle translates to the need for multiple objective functions (or environments) during training. This will force the optimizer to generalize across many optimization tasks.

## §9.6 Multi-Task Learning for Optimizing Regression

In multi-task learning (MTL), a single model (or policy, in the reinforcement learning context) is trained on multiple tasks simultaneously. The aim is to improve generalization by leveraging the commonalities and differences across tasks. The primary objective in MTL is to achieve good performance on all tasks using a shared representation. Contrastingly, meta-learning has a slightly different goal. The model is trained over a variety of tasks with the aim of quickly adapting to new, unseen tasks with minimal training. The main objective in meta-learning is adaptation to new tasks.

In our setup, a single policy is trained across different synthetic datasets (or tasks) to optimize the linear regression objective. The goal is to have one policy that excels across all these tasks and adapts to new tasks. In mathematical terms, the objective function for MTL might be expressed as:

$$\min_{\pi} \sum_{i=1}^N L_{\tau_i}(\pi)$$

Where:

- $\pi$  represents the policy under training.
- $L_{\tau_i}$  denotes the loss on task  $\tau_i$ .
- $N$  signifies the total number of tasks.

The shared policy  $\pi$  is trained with the intent of minimizing the combined loss across all tasks. By training the policy on multiple regression tasks concurrently, the goal is to learn a more generalized understanding of optimization tasks, thereby making the policy robust across a broad spectrum of regression problems. In our experiment, the concept of multi-task learning is inherent in the following ways:

1. **Multiple Environments with Unique Characteristics:** Each environment represents a unique linear regression task, defined by its own set of parameters (weights and biases). By training the policy across multiple such environments, we essentially train it to solve multiple tasks.

2. **Shared Policy Across Tasks:** The core idea in MTL is that tasks share some underlying structure or patterns. In our case, the policy network is shared across all linear regression tasks. It tries to learn a generalized strategy that can perform well on a variety of linear regression tasks with different parameters.
3. **Training Procedure:** In each episode during training, a random environment (task) is chosen from the set of training environments. The policy interacts with this environment for the episode's duration. Over multiple episodes, the policy gets exposed to various tasks, allowing it to learn from the diversity of challenges.
4. **Benefit of MTL in Our Experiment:** The goal is for the policy to find a general strategy to minimize the loss function of linear regression problems, irrespective of the specific parameters of the problem. Training on a single environment might result in overfitting to that specific task. However, by training across multiple tasks, we aim to make the policy more robust and generalized.
5. **Implicit Knowledge Transfer:** By experiencing different tasks, the policy implicitly transfers knowledge among tasks. For instance, the strategy it learns from one task might help it better understand or solve another related task.

During the testing phase, we evaluate the policy on environments it hasn't seen before. This tests the policy's ability to generalize its learning from the training tasks to new, unseen tasks. It's analogous to evaluating how well the knowledge from trained tasks is transferred to new tasks in MTL. The hope is that the diversity of tasks will lead to a more robust and general policy that can effectively tackle a wide range of linear regression problems. In our exploration, we aimed to investigate the feasibility of using reinforcement learning (RL) to optimize linear regression functions. Traditional optimization methods, such as gradient descent, have been well-understood and widely adopted for linear regression problems. However, we sought to discover if an RL-based approach could generalize across various datasets, potentially providing a more adaptable solution.

Our methodology involved the creation of synthetic datasets with known parameters. We then trained an RL policy on a subset of these datasets, with the goal of minimizing the mean squared error (MSE) of a linear regression model. The trained policy was then compared against classic gradient descent on the same datasets in terms of convergence speed and final MSE.

Table 1: Comparison of Gradient Descent and RL Policy on datasets

Dataset #	Gradient Descent MSE (100 iterations)	RL Policy MSE
1	0.1625	0.3345
2	0.1312	0.4164
3	0.0501	0.1333
4	0.0276	0.4367

From our experiments, it became evident that while the RL policy demonstrated the ability to optimize across various datasets, its performance in terms of MSE was not as competitive as classic gradient descent, especially when both were constrained to a similar number of iterations. However, the RL policy's ability to acutally generalize across datasets offers an interesting avenue for further exploration.

In future work, it might be beneficial to explore more sophisticated RL algorithms, train on a wider variety of datasets, or consider more complex regression models to better understand the potential and limitations of RL in optimization tasks.

## §9.7 Adaptive Learning Rate Optimization in Gradient Descent

At its core, Gradient Descent (GD) iteratively adjusts model parameters in order to minimize a given cost or loss function. Mathematically, the update rule for GD is expressed as:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

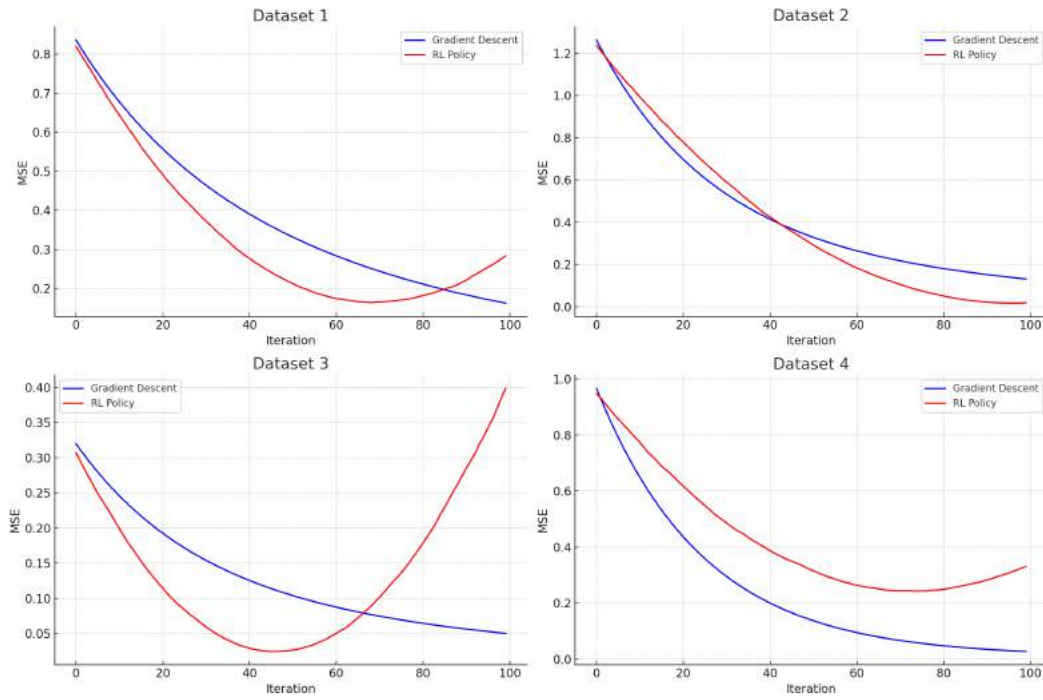


Figure 8: Optimization of single-variable continuous function

Where:

- $\theta$  represents the model parameters.
- $\alpha$  is the learning rate determining the step size.
- $\nabla J(\theta_t)$  is the gradient of the cost function  $J$  with respect to the parameters  $\theta$  at iteration  $t$ .

The learning rate  $\alpha$  plays a pivotal role in the convergence of GD. If  $\alpha$  is set too high, the algorithm might overshoot the optimal point and diverge. Conversely, if it's too low, GD may either converge exceedingly slowly or become trapped in local minima.

### 9.7.1 Adaptive Learning Rate Methods

Over the years, several algorithms have been proposed to adjust the learning rate adaptively, negating the need for manual tuning. Notable among these are:

- **Adagrad:** Scales the learning rate inversely proportional to the square root of the sum of historical squared gradients. The update rule is:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla J(\theta_t)$$

Where  $G_t$  is a diagonal matrix where each diagonal element  $i, i$  is the sum of the squares of the gradients w.r.t.  $\theta_i$  up to time step  $t$ , and  $\epsilon$  is a smoothing term to prevent division by zero.

- **RMSprop:** Modifies Adagrad to use a moving average of squared gradients. This addresses Adagrad's aggressive, monotonically decreasing learning rate.
- **Adam:** Combines ideas from both RMSprop and momentum optimization to compute adaptive learning rates for each parameter.

These methods, while effective, rely on predefined mathematical formulations to adjust the learning rate. They don't "learn" the best adjustments based on the data, which leads us to our novel approach.

### 9.7.2 Reinforcement Learning for Adaptive Learning Rate

Our method harnesses the power of Reinforcement Learning (RL) to dynamically adjust the learning rate in GD. In this framework:

- **State** ( $s$ ): Represents the current state of the optimization process. Formally:

$$s = (\text{current gradient}, \text{current loss})$$

- **Action** ( $a$ ): Dictates the adjustment to the learning rate. It's the output of our policy  $\pi$  conditioned on the current state:

$$a = \pi(s)$$

With  $a$  constrained between  $[-1, 1]$  after processing through a tanh activation.

- **Reward** ( $r$ ): Quantifies how good our action was. It's defined as the negative difference between the initial and final loss after taking action  $a$ :

$$r = \text{Initial Loss} - \text{Final Loss}$$

The goal of the RL agent is to maximize the expected cumulative reward, which in this context translates to minimizing the loss function efficiently.

We use the REINFORCE algorithm, a Monte Carlo Policy Gradient method, to train our policy. Given a trajectory  $\tau$  of states, actions, and rewards, the objective is to maximize the expected cumulative reward. The gradient of the expected reward under this policy is given by:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} \left[ \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R_t \right]$$

Where  $R_t$  is the cumulative reward from time  $t$ . Our policy network, given a state, outputs the adjustment factor for the learning rate. The network architecture is structured as:

- **Input Layer:** 2-dimensional vector (current gradient, current loss)
- **Hidden Layer:** 128 neurons with ReLU activation
- **Output Layer:** Single neuron with tanh activation, yielding a value between  $[-1, 1]$

The choice of the tanh activation function ensures that the action (adjustment factor) remains bounded. This is crucial to prevent extreme learning rate adjustments that could destabilize the optimization process.

We employed the REINFORCE algorithm, a form of policy gradient method, to train our policy network. Given a trajectory  $\tau$  of states, actions, and rewards, the objective is to maximize the expected cumulative reward. The update rule for the policy's parameters  $\theta$  is expressed as: We constructed synthetic linear regression datasets to simulate an environment for testing the learning rate optimization. Each dataset is generated with random weights and biases, with added noise. The environment's task is to determine the optimal weights and biases using gradient descent, by leveraging the policy network for learning rate adjustments.

Our primary evaluation metric was the Mean Squared Error (MSE) trajectory over a set number of iterations. This metric gives a clear indication of the optimization's efficiency. By comparing the MSE trajectory of our RL-guided gradient descent to traditional methods, we can assess our approach's effectiveness.

### 9.7.3 Results and Observations

Upon evaluating our trained policy across various synthetic linear regression datasets, we made several observations: We found that the RL-guided gradient descent converged faster than gradient descent with fixed learning rate. Given a loss function  $L(\mathbf{w})$  that the model aims to minimize, the learning rate  $\alpha$  plays a crucial role in the update equation:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla L(\mathbf{w}_t).$$



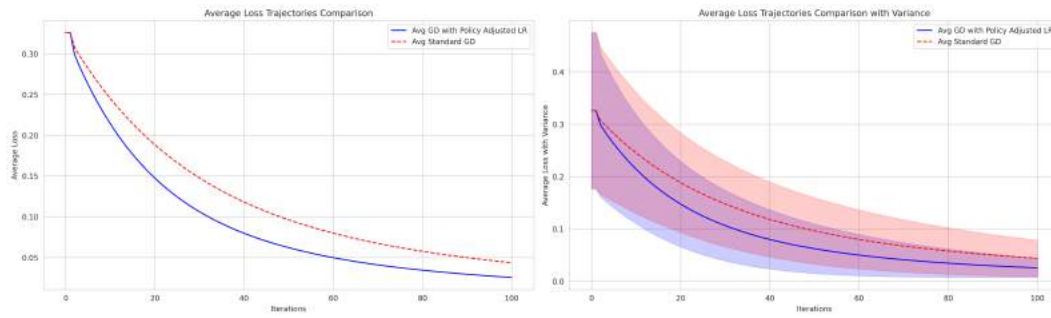


Figure 9: This figure juxtaposes the convergence behavior of traditional gradient descent (Standard GD) with our reinforcement learning guided gradient descent (GD with Policy Adjusted LR) over multiple synthetic linear regression environments. Left plot illustrates the average loss trajectories for both methods across iterations. Right plot depicts the same average loss trajectories but supplements them with shaded regions representing the variance in loss values across different test environments.

- If  $\alpha$  is too high, the updates may overshoot the global or local minima, leading to divergence or oscillation.
- If  $\alpha$  is too low, the algorithm may converge too slowly or become stuck in suboptimal local minima.

These observations suggest that our RL approach to adaptive learning rate optimization, especially for tasks with simple convex loss landscapes, is promising.

## §9.8 Future Directions

While our study primarily focused on linear regression tasks, there are vast potential applications for this approach. Future work could:

- Extending this reinforcement learning-based optimization to more complex classes of functions such as non-convex loss functions
- Integrate other reinforcement learning algorithms or architectures, such as actor-critic methods or deep Q-learning, to verify if they offer added improvements.
- Experiment with imitation learning to learn an optimal policy by leveraging data from optimization algorithms such as RMSpop or Adam to “teach” the agent how to optimize functions.

In conclusion, we investigated techniques for learning optimization algorithms. We framed this as a reinforcement learning challenge, where any optimization algorithm can be modeled as a policy. The task of learning an optimization algorithm thus simplifies to identifying the optimal policy. We used policy gradient methods to train our agent (optimizer) for simple classes of convex objective functions. We found that the performance of these learned optimizers was poor relative to even basic gradient descent. Additionally, we utilized reinforcement learning to develop a policy for adaptively adjusting the learning rate in gradient descent. Upon comparing gradient descent with an adaptive learning rate against gradient descent with a fixed rate, we found that the adaptive method converges more rapidly.

## References

- [1] Sutton, Richard S and Barto, Andrew G *Reinforcement learning: An introduction*. MIT press, 2018.
- [2] Jordan, M.I. and Mitchell, T.M. *Machine learning: Trends, perspectives, and prospects*. Science, 349(6245):255–260, 2015.
- [3] Puterman, Martin L *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.
- [4] Russell, Stuart J and Norvig, Peter *Artificial intelligence: a modern approach*. MIT press, 2016.
- [5] Thrun, Sebastian and Burgard, Wolfram and Fox, Dieter *Probabilistic Robotics*. MIT Press, 2005. <https://docs.ufpr.br/~danielsantos/ProbabilisticRobotics.pdf>
- [6] Komorowski, Matthieu and Celi, Leo Anthony and Badawi, Omar and Gordon, Anthony C. and Faisal, A. Aldo *The Artificial Intelligence Clinician learns optimal treatment strategies for sepsis in intensive care*. Nature Medicine, 24:1716–1720, 2018. <https://finale.seas.harvard.edu/publications/guidelines-reinforcement-learning-healthcare>
- [7] Buehler, Stefan and Ranganathan, Arun and Gupta, Rahul *Reinforcement Learning for Finance*. Stanford University, 2019. <https://stanford.edu/~ashlearn/RLForFinanceBook/book.pdf>
- [8] Mohri, Mehryar and Rostamizadeh, Afshin and Talwalkar, Ameet *Foundations of Machine Learning*. The MIT Press, 2018. <https://cs.nyu.edu/~mohri/mlbook/>
- [9] Bellman, Richard *Dynamic Programming and Stochastic Control Processes*. Information and Control, 1:228–239, 1958.
- [10] Sutton, Richard S. *Learning to Predict by the Methods of Temporal Difference*. Machine Learning, 3:9–44, 1988.
- [11] Watkins, Christopher JCH. *Learning from Delayed Rewards*. PhD Thesis, University of Cambridge, England, 1989.
- [12] Watkins, Christopher JCH and Dayan, Peter *Q-learning*. Machine Learning, 8(3):279–292, 1992.
- [13] Marc G. Bellemare and Will Dabney and Mark Rowland *Distributional Reinforcement Learning*. MIT Press, 2023.
- [14] Ian Goodfellow and Yoshua Bengio and Aaron Courville *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>
- [15] Dimitri Bertsekas *Reinforcement Learning and Optimal Control*. MIT Press, 2019.
- [16] Kober, Jens and Bagnell, J. Andrew and Peters, Jan *Reinforcement Learning in Robotics: A Survey*. International Journal of Robotics Research, 2013. [https://www.ri.cmu.edu/pub\\_files/2013/7/Kober\\_IJRR\\_2013.pdf](https://www.ri.cmu.edu/pub_files/2013/7/Kober_IJRR_2013.pdf)
- [17] Van Hasselt, Hado and Guez, Arthur and Silver, David *Deep Reinforcement Learning with Double Q-learning*. Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016. <https://www.davidsilver.uk/wp-content/uploads/2020/03/doubledqn-1.pdf>
- [18] Li, Ke and Malik, Jitendra. *Learning to Optimize*. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720, United States, [Online]. Available: <https://arxiv.org/pdf/1606.01885.pdf>

- [19] Andrychowicz, Marcin; Denil, Misha; Colmenarejo, Sergio Gómez; Hoffman, Matthew W.; Pfau, David; Schaul, Tom; Shillingford, Brendan; de Freitas, Nando. *Learning to learn by gradient descent by gradient descent*. Google DeepMind, University of Oxford, Canadian Institute for Advanced Research, [Online]. Available: <https://arxiv.org/pdf/1606.01885.pdf>
- [20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov *Proximal Policy Optimization Algorithms*. arXiv preprint arXiv:1707.06347, 2017. [Online]. Available: <https://arxiv.org/pdf/1707.06347.pdf> bibitemGAE2015 John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. *HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION*. Department of Electrical Engineering and Computer Science, University of California, Berkeley. arXiv preprint arXiv:1506.02438, 2015. [Online]. Available: <https://arxiv.org/pdf/1506.02438.pdf>
- [21] Ronald J. Williams. *Simple statistical gradient-following algorithms for connectionist reinforcement learning*. Machine Learning, volume 8, pages 229–256, 1992. [Online]. Available: <https://people.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf>
- [22] Joshua Achiam, David Held, Aviv Tamar, Pieter Abbeel. *Constrained Policy Optimization*. Proceedings of the International Conference on Machine Learning (ICML), 2017. [Online]. Available: <https://doi.org/10.48550/arXiv.1705.10528>
- [23] Sham Kakade. *A Natural Policy Gradient*. Gatsby Computational Neuroscience Unit, London, UK. [Online]. Available: <http://www.gatsby.ucl.ac.uk>
- [24] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. *Trust Region Policy Optimization*. ICML 2015. arXiv preprint arXiv:1502.05477, 2015. [Online]. Available: <https://arxiv.org/pdf/1502.05477.pdf>
- [25] G. Tesauro. *Temporal difference learning and TD-Gammon*. In: Communications of the ACM, 38(3):58–68, 1995.
- [26] L.-J. Lin. *Reinforcement learning for robots using neural networks*. Tech. rep., DTIC Document, 1993.
- [27] G. E. Dahl, D. Yu, L. Deng, and A. Acero. *Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition*. In: IEEE Transactions on Audio, Speech, and Language Processing, 20(1):30–42, 2012.
- [28] A. Krizhevsky, I. Sutskever, and G. E. Hinton. *Imagenet classification with deep convolutional neural networks*. In: Advances in Neural Information Processing Systems, 2012, pp. 1097–1105.
- [29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. *Playing Atari with Deep Reinforcement Learning*. arXiv preprint arXiv:1312.5602, 2013. [Online]. Available: <https://arxiv.org/abs/1312.5602>.
- [30] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. *Mastering the game of Go with deep neural networks and tree search*. Nature, 529(7587):484–489, 2016.