Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Ângelo Gregório Lovatto

Reinforcement Learning based on Policy Gradient

São Paulo
December 2018

Reinforcement Learning based on Policy Gradient

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Leliane de Nunes Barros

São Paulo
December 2018

Abstract

Many Reinforcement Learning methods based on Policy Gradients have been developed over the last decade. Unlike other methods that search for optimal state and action values, from which an optimal policy is derived, Policy Gradient methods optimize a parameterized policy directly. This approach provides significant advantages, such as better convergence guarantees to a locally optimal policy and the ability to handle large or continuous action spaces. Combined with recent deep learning techniques, some of these methods enable complicated problems, such as navigation in robotics, to be tackled directly and with no prior knowledge. Theoretically, these algorithms make little to no assumptions about the type of policy parameterization to be used or the specifics of the tasks to be solved, but these variables can significantly influence the results. Thus, we study and empirically compare three selected methods based on Policy Gradients with respect to their performance in different environments, with varying policy architectures and combinations of hyperparameters. Environments are offered by the OpenAI Gym toolkit for reinforcement learning research and the algorithms implemented using the PyTorch framework, allowing easy computation of gradients through the backpropagation algorithm.

Keywords: Reinforcement Learning, Policy Gradients, Machine Learning.

Contents

Chapter 1

Preliminaries

Reinforcement Learning is a subfield of Machine Learning. It is the problem of learning from the interactions with the environment: at each stage an agent performs an action that can change the current state of the environment producing a reward signal and taking the environment to a next state. The reward depends on the current state and action. The agent's objective is to maximize the expected accumulated reward signal, throughout successive interactions with the environment.

Different challenges arise in this setting from those faced in classical machine learning techniques such as *supervised* or *unsupervised learning*. One such challenge is the compromise between assuring immediate rewards and choosing actions which might increase the chances of receiving greater future rewards. Moreover, the agent must prefer actions that have shown to be more effective in past interactions, but to discover such actions it has to try out actions that have not been selected before. This is called the exploration-exploitation dilemma.

Our focus is on advanced techniques for dealing with the reinforcement learning problem, called *Policy Gradient* methods. Before that, however, we briefly introduce the main concepts, definitions and models used throughout this text.

1.1 Agent-Environment Interface

Markov Decision Processes (MDPs) are stochastic processes that can model the dynamics of a system or environment. These models can be learned, if the problem in question has such an underlying structure, or constructed in order to formalize a problem and explore its properties, as is the case with the environments used for our experiments later on. This is the main model studied in the reinforcement learning literature and therefore will be considered throughout this study. The learner and decision maker is called the *agent*. The system it interacts with, comprising everything outside the agent, is called the *environment*. Both interact continuously, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent.

Formally, the agent and environment interact in a sequence of discrete timesteps. At each timestep t, the agent receives some representation $s_t \in \mathcal{S}$ of the current state of the environment and selects an action $a_t \in \mathcal{A}$, where $\mathcal{S}$ and $\mathcal{A}$ are the sets of states and actions, respectively. The agent then receives from the environment the new current state s_{t+1} and a numerical reward $r_{t+1} \in \mathbb{R}$. The sequence $s_t, a_t, r_{t+1}, s_{t+1}$ is called a transition.

It's important to note that the environment is stochastic in nature, therefore it may not return the same state and reward pair for the same current state and action in different timesteps. Therefore the agent, through its action, is only part of what determines the transition from one state to another, as well as the reward received in that transition. This

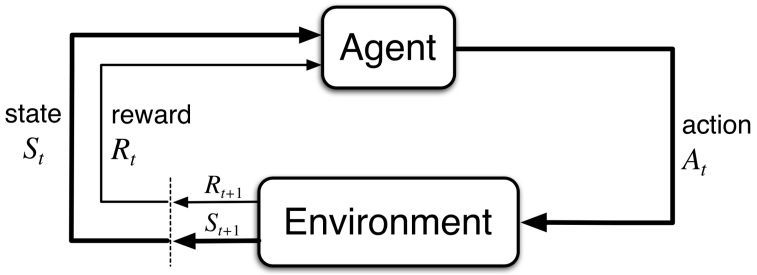
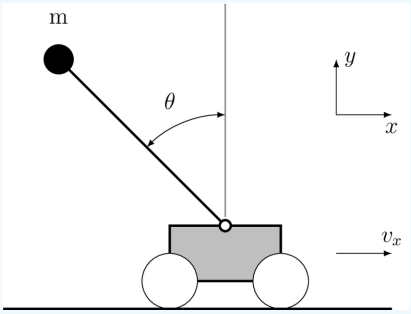


Figure 1.1: *The agent-environment interface in a Markov Decision Process. Figure source: Sutton & Barto (2018)*

makes MDPs a very general and powerful framework for modeling many problems of interest.

Example 1: CartPole

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.



1.2 Theory of MDPs

We now restrict our attention to countable MDPs and the expected discounted total reward criterion in order to formally introduce some key results from the theory of MDPs. In order to reduce clutter and avoid repetition, we provide all results from this section onward for the countable setting. However, under some technical conditions, the results also apply to continuous state-action MDPs.

A countable MDP is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}_0)$, where $\mathcal{S}$ is a countable set of states, $\mathcal{A}$ a countable set of actions and $\mathcal{P}_0$ is a transition probability kernel that assigns a probability measure over $\mathcal{S} \times \mathbb{R}$ for every $(s, a) \in \mathcal{S} \times \mathcal{A}$, which we shall denote by $\mathcal{P}_0(\cdot|s, a)$. Therefore, for every $\mathbf{K} \subset \mathcal{S} \times \mathbb{R}$, $\mathcal{P}_0(\mathbf{K}|s, a)$ gives the probability that the next state and reward belongs to $\mathbf{K}$, given that the current state and action are s and a, respectively.

The transition probability kernel gives rise to the *state transition probability kernel* $\mathcal{P}$, which gives the probability of moving from a state s to state s' by taking action a, for every $(s, a, s') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S}$:

$$\mathcal{P}(s' \mid s, a) = \mathcal{P}_0(\{s'\} \times \mathbb{R}|s, a). \tag{1.1}$$

Moreover, $\mathcal{P}_0$ also gives rise to the *immediate reward function* $\mathcal{R}$, which for every $(s, a) \in \mathcal{S} \times \mathcal{A}$ gives the expected immediate reward when action a is taken in state s: if $(s'_{(s,a)}, r_{(s,a)}) \sim \mathcal{P}_0(\cdot|s, a)$, where $\sim$ means 'sampled from', then

$$\mathcal{R}(s, a) = \mathbb{E}[r_{(s,a)}]. \tag{1.2}$$

Throughout this text, we shall assume that all rewards are bounded, i.e., there is a fixed $M \in \mathbb{R}$

such that for every $(s,a) \in \mathcal{S} \times \mathcal{A}$, $|r_{(s,a)}| \leq M$ almost surely[1]. We call an MDP finite if both $\mathcal{S}$ and $\mathcal{A}$ are finite. Given the above definitions, we may alternatively denote an MDP by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$.

An agent interacting with this MDP at timestep t receives a current state s_t and must pick an action $a_t \in \mathcal{A}$. Then, the next state and reward are sampled according to $\mathcal{P}_0$:

$$(s_{t+1}, r_{t+1}) \sim \mathcal{P}_0(\cdot|s_t, a_t). \tag{1.3}$$

A rule that defines how actions are selected is called a behaviour, and together with some initial random state s_0, defines a random state-action-reward sequence, or *trajectory*,

$$(s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \dots),$$

where (s_{t+1}, r_{t+1}) is linked to (s_t, a_t) by (1.3) and a_t is the action selected by the behaviour based on the trajectory until state s_t. The goal of the agent is to define an action selection rule that maximizes the expected discounted total reward, or *return*. The return from a timestep t is given by

$$R_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'+1}, \tag{1.4}$$

where $0 \leq \gamma \leq 1$, and the return of a trajectory is defined as R_0. Thus, if $\gamma < 1$ then rewards far into the future are worth exponentially less than immediate rewards. We call such MDPs *discounted*, and in the case that $\gamma = 1$, *undiscounted*.

In some MDPs there are what are called *terminal* states: if s is such a state, then $s_{t+k} = s$ almost surely for every $k \geq 1$ provided that $s_t = s$, which means that any action taken in s will lead to s (s is also called an absorbing state). By convention, we consider that no reward is incurred once a terminal state is reached. An *episode* is then the random time period before reaching a terminal state, assuming one is always reached. It is then common to *reset* the experiment by sampling a new initial state, usually from a fixed initial state distribution, from which a new trajectory will be generated. Such MDPs are called *episodic* and will be the main subjects of our experiments later on. In these scenarios, it is common to consider the expected total reward, i.e. when $\gamma = 1$, since the sum in equation (1.4) will be bounded, assuming there is an upper bound on the length of an episode or that the probability of longer episodes decreases at a fast enough rate.

Stationary policies represent a special class of behaviours which play a fundamental role in the theory of MDPs. A *stochastic* stationary policy (or simply stationary policy) π defines a probability distribution over the action space for every state in the state space. We use $\pi(\cdot|s)$ to denote the probability mass function over $\mathcal{A}$ for state s and thus $\pi(a|s)$ denotes the probability of action a being selected at state s. Following a stationary policy means that:

$$a_t \sim \pi(\cdot|s_t), \quad t \in \mathbb{N}. \tag{1.5}$$

A policy and an MDP together give rise to the stochastic process $((s_t, a_t, r_{t+1});\ t \geq 0)$, where $(s_{t+1}, r_{t+1}) \sim \mathcal{P}_0(\cdot|s_t, a_t)$ and $a_t \sim \pi(\cdot \,|\, s_t)$. Note that the state process $(s_t; t \geq 0)$ when following a stationary policy is a (time-homogeneous) Markov chain. For brevity, throughout this text we'll refer to stationary policies as simply policies and use Π_{stat} to denote this class of policies.

The *state value function* (or value function for short), $V^\pi : \mathcal{S} \to \mathbb{R}$, underlying a fixed policy π is defined as

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| s_0 = s\right], \quad s \in \mathcal{S}, \tag{1.6}$$

with the understanding that *(i)* $(r_{t+1}; t \geq 0)$ is the reward part of the process $((s_t, a_t, r_{t+1}); t \geq 0)$

[1] Almost surely here means with probability 1. The expression is used to indicate that the event occurs in every point in the probability space, except in a subset with probability zero.

obtained by following policy π and *(ii)* $\mathbb{P}(s_0 = s) > 0$ holds for all states $s \in \mathcal{S}$, where $\mathbb{P}$ denotes the probability of an event. The second condition is necessary in order for the conditional probability to be well-defined. Similarly, the *action-value function*, $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, underlying a policy π is defined as

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| s_0 = s, a_0 = a\right], \quad s \in \mathcal{S}, a \in \mathcal{A}, \tag{1.7}$$

subject to the same conditions and also that $\mathbb{P}(a_0 = a) > 0$ holds for all $a \in \mathcal{A}$. The following equalities follow fairly straightforward from (1.6) and (1.7):

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \,|\, s) Q^\pi(s,a), \quad s \in \mathcal{S}, \tag{1.8}$$

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s,a) V^\pi(s'), \quad (s,a) \in \mathcal{S} \times \mathcal{A}. \tag{1.9}$$

Definition 1: Bellman equations

Fix an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}_0)$, a discount factor $0 \le \gamma \le 1$ and a policy $\pi \in \Pi_{\text{stat}}$. Let r be the immediate reward function of $\mathcal{M}$. Then V^π satisfies

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s,a) V^\pi(s')\right], \quad s \in \mathcal{S}. \tag{1.10}$$

This system of equations is called the Bellman equation for V^π. Define the Bellman operator underlying π, $T^\pi : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$, by

$$(T^\pi V)(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s,a) V(s')\right], \quad s \in \mathcal{S}. \tag{1.11}$$

With the help of T^π, Equation (1.10) can be written in the compact form

$$T^\pi V^\pi = V^\pi. \tag{1.12}$$

Note that this is a linear system of equations and T^π is an affine linear operator. If $0 < \gamma < 1$ then T^π is a maximum-norm contraction and the fixed point equation $T^\pi V = V$ has a unique solution. Similarly, Q^π satisfies

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s,a) \sum_{a' \in \mathcal{A}} Q^\pi(s',a'), \quad (s,a) \in \mathcal{S} \times \mathcal{A}. \tag{1.13}$$

Redefining $T^\pi : \mathbb{R}^{\mathcal{S} \times \mathcal{A}} \to \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ as

$$(T^\pi Q)(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q(s',a'), \quad (s,a) \in \mathcal{S} \times \mathcal{A}, \tag{1.14}$$

then if $0 < \gamma < 1$, Q^π is the unique solution to $T^\pi Q = Q$.

Recall that the maximum-norm of a function $f : \mathcal{X} \to \mathbb{R}$ is defined by $\|f\|_\infty = \sup_{x \in \mathcal{X}} |f(x)|$. The above definition follows from Banach's fixed-point theorem, which also gives that, starting from any initial guess V_0, the sequence $(V_k; k \ge 0)$ generated by $V_{k+1} = T^\pi V_k$ converges to V^π at geometric rate (see appendix A of Szepesvári (2009) for a detailed explanation). It is not hard to see that the definitions for the state and action value functions given in (1.6) and (1.7) respectively satisfy the Bellman equations.

1.3 Classical Solutions and their Limitations

Control methods in reinforcement learning provide ways of finding policies that maximize the expected return. The highest achievable expected return from state $s \in \mathcal{S}$ is given by the *optimal value function* $V^* : \mathcal{S} \to \mathbb{R}$. A policy that achieves the optimal values in *all* states is called optimal. Similarly, the *optimal action-value function* $Q^* : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ gives, for every $(s, a) \in \mathcal{S} \times \mathcal{A}$, the highest achievable expected return provided that the process starts at state s and the first action chosen is a. Optimal value functions also have their corresponding Bellman equations.

Definition 2: Bellman Optimality Equations

The optimal value function satisfies the fixed-point equation

$$V^*(s) = \sup_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s, a) V^*(s') \right\}, \quad s \in \mathcal{S}.$$

Define the Bellman optimality operator, $T^* : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$, by

$$T^*V = \sup_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s, a) V(s') \right\}, \quad s \in \mathcal{S}.$$

Thus, $T^*V^* = V^*$ and if $0 < \gamma < 1$ then T^* is a maximum-norm contraction and $T^*V = V$ has a unique solution. Similarly, Q^* satisfies

$$Q^*(s) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s, a) \sup_{a' \in \mathcal{A}} Q^*(s', a'), \quad (s,a) \in \mathcal{S} \times \mathcal{A}.$$

Define the Bellman optimality operator for Q, $T^* : \mathbb{R}^{\mathcal{S} \times \mathcal{A}} \to \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$, by

$$(T^*Q)(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s, a) \sup_{a' \in \mathcal{A}} Q(s', a'), \quad (s,a) \in \mathcal{S} \times \mathcal{A}.$$

Then T^* is a maximum-norm contraction, $T^*Q^* = Q^*$ and Q^* is the unique solution to this fixed-point equation.

Value iteration algorithms generate a sequence of state value functions

$$V_{k+1} = T^*V_k \quad \text{or} \quad Q_{k+1} = T^*Q_k, \quad k \geq 0,$$

which, by Banach's fixed point theorem, converges to the optimal function at a geometric rate. These optimal functions are related by the following equations.

$$V^*(s) = \sup_{a \in \mathcal{A}} Q^*(s,a), \quad s \in \mathcal{S},$$

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \,|\, s, a) V^*(s'), \quad (s,a) \in \mathcal{S} \times \mathcal{A}.$$

Any policy $\pi \in \Pi_{\text{stat}}$ which satisfies

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

is optimal. Since the equation above implies that $\pi(\cdot|s)$ is concentrated on the actions that maximize Q^*, π is called *greedy* with respect to Q^*. Similarly, given V^*, r and $\mathcal{P}$, it is possible to compute a policy π that is greedy w.r.t. V^*, and it is not hard to see that such a policy is also optimal. Thus, value iteration gives an indirect method for finding an optimal policy.

Policy iteration methods work as follows. Fix an arbitrary initial policy π_0. At iteration $k > 0$ compute the action-value function Q^π underlying π (this is called the policy *evaluation step*). Then, define π_{k+1} as the policy which is greedy w.r.t. Q^π (this is called the policy *evaluation* step). After k iterations, policy iteration gives a policy not worse than the policy that is greedy with respect to Q_k computed by value iteration, given that both methods start from the same initial policy.

Problems begin to arise once we start considering large state or action spaces. The game of GO is a classical example where the size of the state space makes the problem intractable in practice, since value- and policy-iteration algorithms, also called dynamic programming methods, need to update the value function for all states. Several learning counterparts of these methods have been developed over the years for this reason. They can be seen as sample-based, approximate versions of the classical methods of dynamic programming, such as Q-learning (value iteration) and SARSA (policy iteration).

However, many still require the values of each state-action pair to be stored in memory, the so called *tabular* methods. As in the case of GO, this can easily get out of hand, and continuous state-action MDPs are not even a consideration. In order to handle such large state-action spaces, many methods started incorporating ideas from supervised learning, specially function approximation, by replacing the tabular value functions $Q(s, a)$ with parameterized, differentiable functions $\hat{Q}(s, a, \boldsymbol{w})$, where $\boldsymbol{w} \in \mathbb{R}^d$, $d \ll |\mathcal{S}|$. The idea is to *approximate* the real value functions by adjusting the parameter $\boldsymbol{w}$ in order to minimize some error measure between the approximated function and its real counterpart, usually measured by sampling. The intuition behind it is that $\hat{Q}$ will *generalize* to points not previously visited in the state-action space, which saves a lot in terms of computational cost. Recent advances in function approximation methods, such as neural networks, have yielded impressive results when used in conjunction with reinforcement learning methods, as is the case with Deep Q-learning (see Mnih *et al.* (2015)).

It has been shown, however, that even when using simple, linear function approximation, many algorithms are not guaranteed to converge to an optimal policy, as shown by Baird (1995) in the case of Q-learning (see section 3). The problem is connected to the fact that, by adjusting the weight vector $\boldsymbol{w}$, several action-values are affected, and this correlation may lead action-values to diverge. Moreover, all of the aforementioned methods approximate an optimal policy indirectly, i.e., by first computing value functions from which the policy is derived. This can be a problem for high-dimensional action spaces, since one needs to efficiently compute $\arg\max_{a \in \mathcal{A}} \hat{Q}(s, a)$ when computing greedy policies (or worse, one needs r and $\mathcal{P}$ when the state-value function is being approximated). *Policy gradient* methods address these difficulties by directly optimizing the policy, but also come with their own set of challenges.

1.4 Policy Gradient based Solutions

In Policy Gradients, the objective is to find a *parameterized* policy from a smoothly parameterized policy class $\Pi = \{\pi_{\boldsymbol{\theta}};\ \boldsymbol{\theta} \in \mathbb{R}^d\}$ that maximizes some performance measure $J(\boldsymbol{\theta})$. Here, $\boldsymbol{\theta} \in \mathbb{R}^d$ denotes the vector of parameters of the policy $\pi_{\boldsymbol{\theta}}$, and we assume that π is differentiable with respect to its paramater, i.e., that $\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s, a)$, $(s, a) \in \mathcal{S} \times \mathcal{A}$, exists. For simplicity, we usually write $\nabla \pi_{\boldsymbol{\theta}}(s, a)$ when it is implicit that the derivation is w.r.t. $\boldsymbol{\theta}$.

Example 2: Policy parameterizations

In general, the states in our experiments consist of vectors in $\mathbb{R}^n$ characterizing certain aspects of the state, e.g., in the CartPole problem (Example 1) the state feature vector consists of the cart's velocity, the cart's position, the pole's angle with the vertical and the pole's velocity at the tip. As for the actions, there are two situations in general: finite discrete action spaces and continuous action spaces.

For a finite discrete action space, Figure 1.2 shows how one can implement a policy parameterization using a Multilayer perceptron, where the input nodes correspond to each component

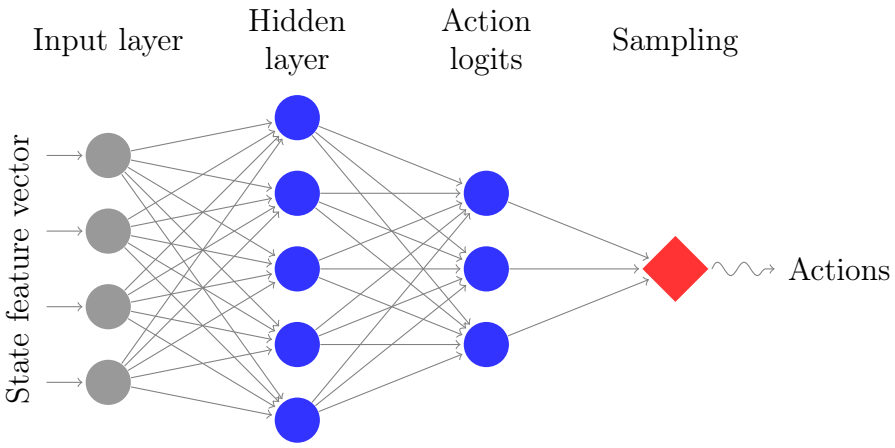


Figure 1.2: *Policy parameterization for discrete action spaces using a multilayer perceptron. The sampling node implements a softmax distribution using the action logits*

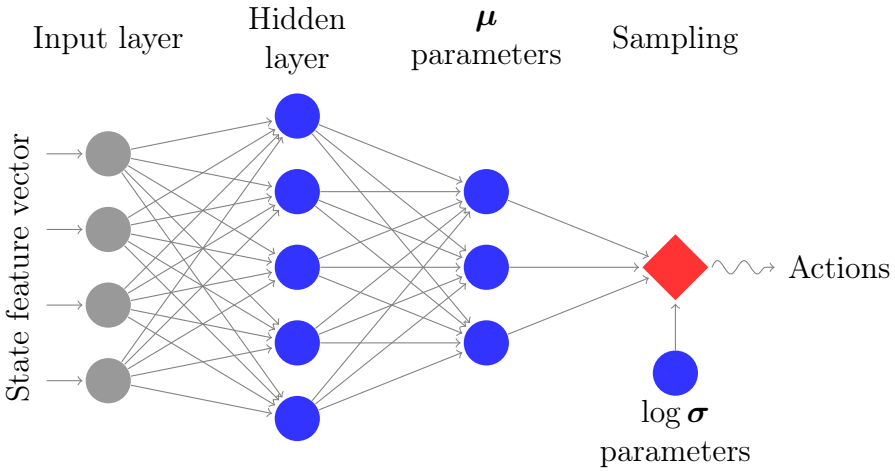


Figure 1.3: *Policy parameterization for continuous action spaces using a multilayer perceptron. The sampling node implements a multivariate normal distribution using the mean vector* $\boldsymbol{\mu}$ *and diagonal covariance matrix with entries* $\boldsymbol{\sigma}$

of the state feature vector. The network then maps these inputs to numerical values for each action, denoted by the nodes in the *logits* layer. These logits can be thought of as functions of the input vector and the weights of the connections of the network, which we denote by a single vector $\boldsymbol{\theta} \in \mathbb{R}^d$, where d is the number of weights in the network. Given a state $s \in \mathcal{S}$, let $h(s, a, \boldsymbol{\theta})$ denote the numerical value corresponding to action $a \in \mathcal{A}$. The *sampling* node combines these values to define a *softmax* distribution as follows:

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{b \in \mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}}, \quad a \in \mathcal{A}.$$

It is not hard to see that $\pi_{\boldsymbol{\theta}}(\cdot \mid s)$ is a probability mass function in the space of actions. Thus, this neural network architecture defines a policy class Π where each choice of parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ corresponds to a policy $\pi_{\boldsymbol{\theta}} \in \Pi$.

One can deal with continuous action spaces in a similar way, as shown by Figure 1.3. Here, we assume that actions consist of vectors in $\mathbb{R}^m$, e.g., the joint torques in a robot. The parameters in this network consist of the connection weights, $\boldsymbol{\theta}' \in \mathbb{R}^{d'}$, and a vector $\log \boldsymbol{\sigma} \in \mathbb{R}^m$, denoted by the node below the sampling node, and thus $\boldsymbol{\theta} = (\boldsymbol{\theta}', \log \boldsymbol{\sigma})^\intercal \in \mathbb{R}^d$. This network maps the state vector to a vector $\boldsymbol{\mu} = \boldsymbol{\mu}(s, a, \boldsymbol{\theta}') \in \mathbb{R}^m$ (the $\boldsymbol{\mu}$ parameters layer) and the sampling node combines it with $\log \boldsymbol{\sigma}$ to define a *multivariate normal distribution* as follows:

$$\pi_{\boldsymbol{\theta}}(\cdot \mid s) = \mathcal{N}(\boldsymbol{\mu}, \operatorname{diag}(e^{2 \log \boldsymbol{\sigma}})),$$

where $\operatorname{diag}(e^{2 \log \boldsymbol{\sigma}}) = \operatorname{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_m^2)$. This is equivalent to a collection of m normal distributions with mean μ_i and variance σ_i^2, respectively. Note that the variance does not depend on the state input, which allows for the policy to control its exploratory behaviour independently.

Policy gradient methods perform stochastic gradient ascent on the surface of J induced by Π in order to find an approximately optimal policy $\pi_{\boldsymbol{\theta}}$, thus applying the following update rule to the policy parameter,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k \widehat{\nabla} J(\boldsymbol{\theta}_k), \quad k \geq 0, \tag{1.15}$$

where α_k is called the step size or learning rate. The hat symbol means that we're using an approximation of the gradient. If $\widehat{\nabla} J(\boldsymbol{\theta}_k)$ is an unbiased estimate of the gradient, i.e., $\mathbb{E}[\widehat{\nabla} J(\boldsymbol{\theta}_k)] = \nabla J(\boldsymbol{\theta}_k)$ for every k, then the sequence $(\boldsymbol{\theta}_k)$ will converge to a stationary point of J almost surely, provided that the sequence (α_k) satisfies the *Robbins-Monro (RM) conditions*,

$$\sum_{k=0}^{\infty} \alpha_k = \infty, \qquad \sum_{k=0}^{\infty} \alpha_k^2 < +\infty,$$

and assuming ∇J is Lipschitz continuous and the error in the approximation of $\nabla J(\boldsymbol{\theta}_k)$ satisfies standard conditions in stochastic gradient methods, as in Bertsekas & Tsitsiklis (2000). We assume these conditions are satisfied.

Definition 3: On-policy distribution of states

Fix an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho_0)$ and a policy π, where $\rho_0 : \mathcal{S} \to [0, 1]$ denotes the initial state probability distribution. Then, the probability of visiting any state $s \in \mathcal{S}$ at time t is given by

1. $\mathbb{P}(s_0 = s) = \rho_0(s)$,
2. $\mathbb{P}(s_t = s) = \sum_{s' \in \mathcal{S}} \mathbb{P}(s_{n-1} = s') \sum_{a \in \mathcal{A}} \pi(a|s') \mathcal{P}(s' \mid s, a), \quad t > 0.$

Let ρ_π denote the discounted state visitation frequencies:

$$\rho_\pi(s) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s), \quad s \in \mathcal{S}. \tag{1.16}$$

Then, normalizing the frequencies to sum up to one, the on-policy distribution of states is given by

$$\begin{aligned} d^\pi(s) &= \left(\sum_{s' \in \mathcal{S}} \rho_\pi(s') \right)^{-1} \rho_\pi(s) \\ &= \left(\sum_{s' \in \mathcal{S}} \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s) \right)^{-1} \rho_\pi(s) \\ &= \left(\sum_{t=0}^{\infty} \sum_{s' \in \mathcal{S}} \gamma^t \mathbb{P}(s_t = s) \right)^{-1} \rho_\pi(s) \\ &= \left(\sum_{t=0}^{\infty} \gamma^t \right)^{-1} \rho_\pi(s) \\ &= (1-\gamma) \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s), \qquad s \in \mathcal{S}. \end{aligned}$$

Note that this requires that $\gamma \in (0, 1)$. When dealing with episodic MDPs in which, for all policies, every episode reaches a terminal state within some finite time T, it is useful to consider Equation 1.16 with the sum modified to sum to T rather than ∞. Then, using $\gamma = 1$ yields

$$d^\pi(s) = \frac{\rho_\pi(s)}{T}, \qquad s \in \mathcal{S},$$

which is the average number of times s is visited in an episode.

The performance measure $J(\boldsymbol{\theta})$ to be considered here is the expected return by following policy $\pi_{\boldsymbol{\theta}}$ in an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho_0)$:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \right],$$

where the subscript of $\mathbb{E}$ enumerates the variables being integrated over and $s_0 \sim \rho_0$, $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot \,|\, s_t)$ and $s_{t+1} \sim \mathcal{P}(\cdot \,|\, s_t, a_t)$ for $t \geq 0$. The problem is that this function (and most other performance measures of interest) depends on the distribution of states induced by $\pi_{\boldsymbol{\theta}}$ and the MDP. Since reinforcement learning methods avoid using any knowledge of the dynamics of the MDP, this poses a challenge at first. Fortunately, the *Policy Gradient Theorem* provides an expression for the gradient $\nabla J(\boldsymbol{\theta})$ with respect to the policy parameters without the derivative of the state distribution. The following theorem and proof are adapted from Sutton *et al.* (2000).

Theorem 1: Policy Gradient

Given an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho_0)$ and a smoothly parameterized policy class $\Pi = \{\pi_{\boldsymbol{\theta}};\ \boldsymbol{\theta} \in \mathbb{R}^d\}$, the gradient of the expected return $J(\boldsymbol{\theta}) = \mathbb{E}_{s_{0:\infty}, a_{0:\infty}} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \right]$ following $\pi_{\boldsymbol{\theta}}$ for any $\boldsymbol{\theta} \in \mathbb{R}^d$ is given by

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^{\pi_{\boldsymbol{\theta}}}(s, a) \nabla \pi_{\boldsymbol{\theta}}(a|s). \tag{1.17}$$

The proof is given in appendix A.1. Theorem 1 provides the basis for policy gradient algorithms

by providing an expression for the gradient of the performance measure that doesn't depend on the state distribution and can be sampled from experience:

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \sum_{s\in\mathcal{S}} d^{\pi_{\boldsymbol{\theta}}}(s) \sum_{a\in\mathcal{A}} Q^{\pi_{\boldsymbol{\theta}}}(s,a)\nabla\pi_{\boldsymbol{\theta}}(a|s) \\ &= \mathbb{E}_{s_{0:\infty}}\left[\sum_{t=0}^{\infty}\gamma^t \sum_{a\in\mathcal{A}} Q^{\pi_{\boldsymbol{\theta}}}(s_t,a)\nabla\pi_{\boldsymbol{\theta}}(a|s_t)\right].\end{aligned} \tag{1.18}$$

Although the given quantity is only proportional to the gradient, it is so by a constant, which can be absorbed into the learning rate schedule (α_k) in (2.1). Different algorithms may have variations with respect to the parameter update rule or of how Q^π is estimated. Nevertheless, the underlying principles are the same: we define a performance measure, choose a differentiable policy parameterization and estimate the gradient of performance through experience.

Chapter 2

Vanilla Policy Gradient

The first policy gradient algorithm we'll be looking at is commonly known as "vanilla" policy gradient. The term "vanilla" comes from the fact that the algorithm makes little to no changes to the original gradient yielded by the policy gradient theorem. By keeping the gradient estimate unbiased, all the theoretical convergence guarantees are satisfied, although the variance of such estimate may cause the algorithm to converge very slowly or achieve an unsatisfactory low local optimum of $J(\boldsymbol{\theta})$.

Such issues can be addressed by using reinforcement baselines to discriminate better-than-average actions from others, while still keeping the gradient estimate unbiased, as proposed by Williams (1992). Other approaches deliberately introduce some bias into the gradient estimate in order to reduce the variance of the estimates in high-dimensional problems such as robot control. In what follows, we'll be considering episodic MDPs in which every episode terminates in a finite amount of time.

2.1 REINFORCE update

The most straightforward way to estimate (1.18) is to use the sampled action a_t and the return from timestep t (log denotes the natural logarithm):

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}_{s_{0:\infty}}\left[\sum_{t=0}^{\infty}\gamma^t\sum_{a\in\mathcal{A}}Q^{\pi_{\boldsymbol{\theta}}}(s_t,a)\nabla\pi_{\boldsymbol{\theta}}(a\,|\,s_t)\right]\\
&= \mathbb{E}_{s_{0:\infty}}\left[\sum_{t=0}^{\infty}\gamma^t\sum_{a\in\mathcal{A}}Q^{\pi_{\boldsymbol{\theta}}}(s_t,a)\pi_{\boldsymbol{\theta}}(a\,|\,s_t)\frac{\nabla\pi_{\boldsymbol{\theta}}(a\,|\,s_t)}{\pi_{\boldsymbol{\theta}}(a\,|\,s_t)}\right]\\
&= \mathbb{E}_{\substack{s_{0:\infty}\\a_{0:\infty}}}\left[\sum_{t=0}^{\infty}\gamma^t Q^{\pi_{\boldsymbol{\theta}}}(s_t,a_t)\nabla\log\pi_{\boldsymbol{\theta}}(a_t\,|\,s_t)\right] && (\nabla\log x=\tfrac{\nabla x}{x})\\
&= \mathbb{E}_{\substack{s_{0:\infty}\\a_{0:\infty}}}\left[\sum_{t=0}^{\infty}\gamma^t R_t\nabla\log\pi_{\boldsymbol{\theta}}(a_t\,|\,s_t)\right] && (Q^{\pi}(s,a)=\mathbb{E}[R_t\,|\,s_t=s,a_t=a]).
\end{aligned}$$

Thus, we have a quantity that can be sampled after each episode, or *roll-out*, yielding the update

$$\boldsymbol{\theta}_k=\boldsymbol{\theta}_{k+1}+\alpha_k\left(\sum_{t=0}^{\infty}\gamma^t R_t\nabla\log\pi_{\boldsymbol{\theta}}(a_t\,|\,s_t)\right). \tag{2.1}$$

This equation has an intuitive interpretation: if the return obtained after taking action a_t in state s_t is positive, $R_t\nabla\log\pi_{\boldsymbol{\theta}}(a_t\,|\,s_t)$ points in the direction of increasing the log of the probability, also called *likelihood*, of that action given the state input s_t. If the return from the roll-out is positive, the probability of that trajectory is increased.

Theorem 1 can also be extended to include a *baseline*:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} d^{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in \mathcal{A}} \left(Q^{\pi_{\boldsymbol{\theta}}}(s,a) - b(s)\right) \nabla \pi_{\boldsymbol{\theta}}(a \,|\, s)$$

since

$$\sum_{a \in \mathcal{A}} \nabla \pi_{\boldsymbol{\theta}}(a \,|\, s) b = b \sum_{a \in \mathcal{A}} \nabla \pi_{\boldsymbol{\theta}}(a \,|\, s) = b \nabla \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a \,|\, s) = b \nabla \mathbf{1} = 0,$$

as long as b doesn't depend on the action. The resulting parameter update rule is

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k+1} + \alpha_k \left(\sum_{t=0}^{\infty} \gamma^t \left(R_t - b(s_t)\right) \nabla \log \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t) \right). \tag{2.2}$$

This rule was originally proposed by Williams (1992) and used by the author to define *episodic* REINFORCE algorithms. The baseline b, originally called a *reinforcement baseline*, does not introduce bias into the estimator, but it may play an important role in minimizing the variance of the gradient estimate.

An interesting choice of baseline is the state-value function V^{π}, since substituting it for b in the above expression for the policy gradient makes the term weighting $\nabla \pi(a \,|\, s)$ equivalent to the *advantage function* underlying π:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s), \quad (s,a) \in \mathcal{S} \times \mathcal{A}. \tag{2.3}$$

Intuitively, A^{π} tells by how much taking action a in state s differs from the default behavior of the policy (randomly sampling an action), and it's easy to see that its expected value is zero in all states (where the expectation is w.r.t. the probability distribution of actions defined by the policy in each state). This is specially interesting in problems where the rewards are all either negative or positive, as in Example 1, allowing better-than-average actions to be discriminated from worse-than-average ones. Of course, V^{π} is not known and must be estimated in practice.

2.2 Practical algorithm

Although one can implement (2.2) with just a single roll-out, it is better to average the policy gradient estimate across several roll-outs in order to obtain a more robust estimate:

$$\nabla J(\boldsymbol{\theta}) \approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{\infty} \gamma^t \left(R_t^{(i)} - b(s_t^{(i)}) \right) \nabla \log \pi_{\boldsymbol{\theta}}(a_t^{(i)} \,|\, s_t^{(i)}),$$

where the superscript indicates which roll-out the variables are calculated from. As alluded to in the previous section, the baseline considered here will be an approximation to V^{π}. One simple way to do so when implementing $b_{\boldsymbol{\phi}}$ using a non-linear function approximator with parameter vector $\boldsymbol{\phi}$, as explained by Schulman *et al.* (2015b), is to solve a nonlinear regression problem:

$$\min_{\boldsymbol{\phi}} \sum_{n=1}^{N} \left\| b(s_n) - R_n \right\|^2, \tag{2.4}$$

where n indexes over all sampled trajectories and timesteps. This approach is also used for value function estimation in the classical methods mentioned in Section 1.3. Algorithm 1 implements this approach and is usually referred to as the "vanilla" policy gradient algorithm, for the reasons enunciated at the beginning of this chapter.

Algorithm 1 "Vanilla" policy gradient

Input: Parameterized policy $\pi_{\boldsymbol{\theta}}$, learning rate schedule (α_k), baseline b, discount parameter γ

Initialize policy parameter vector $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, baseline b
for k=1, 2, 3, ... **do**
  Generate a set of episodes $s_0, a_0, r_1, s_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$, following $\pi_{\boldsymbol{\theta}_k}$
  for each episode i and timestep t **do**
    Compute the return $R_t^{(i)} = \sum_{k=t}^{T-1} \gamma^{k-t} r_{t+1}$
    Compute the advantage estimate $A_t^{(i)} = R_t^{(i)} - b(s_t^{(i)})$
  end for
  Fit baseline to the empirical returns by minimizing $\left\| b(s_t^{(i)}) - R_t^{(i)} \right\|^2$ summed over all episodes and timesteps
  $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \alpha_k \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \gamma^t A_t^{(i)} \nabla \log \pi_{\boldsymbol{\theta}}(a_t^{(i)} \mid s_t^{(i)})$
end for

2.3 Simple Experiments

Throughout this monograph, we'll consider control problems available in the OpenAI Gym (GYM) toolkit by Brockman *et al.* (2016), which offers a variety of environments implementing the agent-environment interface of Figure 1.1. For this section, we picked two environments, described in detail in the following subsection, from the `classic_control` module from GYM and applied Algorithm 1 in order to approximately solve them. The experiments were designed to explore the following questions:

- How does the learning rate schedule influence the learning processes? Is it better to have faster or slower decaying step sizes?
- How do different policy architectures influence the final performance? Do bigger networks benefit from better expressivity or does the higher variance from having more parameters negatively impact performance?

2.3.1 Environments Details

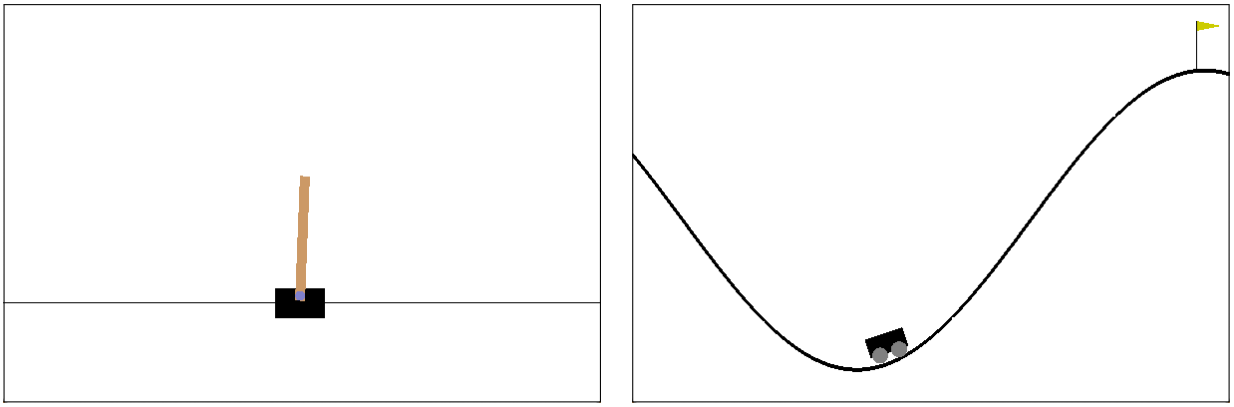


Figure 2.1: *Renderings of the* `CartPole-v0` *(left) and* `MountainCar-v0` *(right) environments.*

Two environments were selected for the experiments: `CartPole-v0` (example 1) and `MountainCarContinuous-v0`. In the former, the agent controls a cart on a horizontal line with a pole attached to it by an un-actuated joint by choosing between the actions *move left* and *move right*

at every timestep. The state feature vector consists of: the cart's velocity, the cart's position, the pole's angle with the vertical and the pole's velocity at the tip. A reward of +1 is incurred at every timestep, and the episode ends if the pole is more than 12^o from the vertical, the cart's position is more than ± 2.4 (with the origin being the center of the display) or the episode length is more than 200 timesteps.

While the goal in the first is about maintaining the pole balanced for as long as possible and positive rewards are plentiful and available from the beginning, the second poses a harder challenge. The agent controls a car that starts at a valley between two hills by regulating the car's engine strength, choosing an action $a \in [-1, 1]$ at each timestep, where negative values push the car to the left and positives, to the right. The state feature vector consists of the car's position and speed. The agent receives a penalty of $-0.1a^2$ at every timestep and a reward of $+100$ if it reached the top of the right hill. A crucial factor in this problem is that the car's engine doesn't have enough force to push it directly to the top of the right hill, therefore the agent must learn to swing the car back and forth between the hills in order to gain enough momentum to make it over the right hill. The penalty in the amount of force applied to the engine makes it so that an optimal policy must not only make it to the top but do it so using the least amount of force. However, if the agent never stumbles upon a sequence of actions (sampled by the stochastic policy) that take the car to the top, it will very likely instead learn to decrease the force applied by engine, settling for a local optimum where the car stays at the valley receiving 0 reward.

Figure 2.1 shows renderings of the two environments. Environments are considered solved when the average return over a certain number of consecutive episodes is greater than a given threshold. Each GYM environment has its own `EnvSpec` instance which specifies the requirements for solving it. In `CartPole-v0`, the policy must achieve a return > 195, which corresponds to balancing the pole for at least 195 timesteps over 100 consecutive episodes; while in `MountainCarContinuous-v0` a return of > 90 must be achieved over 100 consecutive episodes. The `EnvSpec` also includes the timestep limit for the episodes, which can be seen as the maximum time period before reaching a terminal state. Since all environments have this limit, we'll consider the undiscounted expected return criterion.

2.3.2 Experimental Setup

Implementation details

All the algorithms in this monograph were implemented in the *Python* programming language using the *PyTorch* framework by Paszke *et al.* (2017), allowing easy computation of gradients through the *backpropagation* algorithm, the backbone of many deep learning applications to this day. One of the advantages of using automatic differentiation libraries is that it allows one to estimate the policy gradient using the following formula:

$$\begin{aligned}\mathbb{E}_{\substack{s_{0:\infty}\\a_{0:\infty}}}\left[\sum_{t=0}^{T}\gamma^t A_t \nabla \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right] &= \nabla \mathbb{E}_{\substack{s_{0:\infty}\\a_{0:\infty}}}\left[\sum_{t=0}^{T}\gamma^t A_t \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right] \\ &\approx \nabla \frac{1}{N}\sum_{i=1}^{m}\sum_{t=0}^{T}\gamma^t A_t^{(i)} \log \pi_{\boldsymbol{\theta}}(a_t^{(i)} \mid s_t^{(i)}),\end{aligned}$$

where A_t refers to the estimate of the advantage function $A^{\pi}(s_t, a_t)$, calculated in step 6 of the algorithm, treated as a constant. The first line is valid assuming the order of differentiation and integration (with respect to the probabilities of states and actions) can be exchanged. Sufficient conditions for this are given by the Leibniz integral rule, which we assume are satisfied.[1].

In the second line, instead of dividing the sum of all likelihood-advantage pairs by the number

[1]It is not unreasonable to assume so, since most function approximation methods used for the policy, such as neural networks, are continuously differentiable.

of episodes sampled, we take the mean over all the pairs, therefore substituting $\frac{1}{N}$ for $\frac{1}{m}$, were N is the total number of transitions. This doesn't change the direction of the gradient and helps keep its magnitude controlled even when episodes differ in length.

As a final note, most stochastic optimization procedures consider minimization of objective functions, therefore we take the negative of the gradient estimate and consider stochastic gradient descent instead.

Algorithm settings

Keeping in line with the theory, the Harmonic series, $\alpha_k = \frac{1}{k}$, was chosen as the general learning rate schedule, which is well-known to satisfy the RM conditions enunciated in Section 1.4. In order to control the number of steps between decreases in learning rate, a δ variable was added as follows:

Algorithm 1 hyperparameters	
Iterations	200
Batch size	20 episodes
γ	1

$$\alpha_k = \frac{1}{\lfloor k/\delta \rfloor + 1}, \quad \delta \in \mathbb{N}.$$

The choice of δ is indicated in Figure 2.2 using the convention 'SGDδ', where SGD stands for stochastic gradient descent[2].

For step 8 of Algorithm 1, we used the L-BFGS algorithm in order to approximately solve the non-linear regression problem enunciated in (2.4). Although this is a somewhat arbitrary choice, it is implemented in PyTorch as an `optimizer` and gives good results in practice. A maximum number of 10 iterations was used to fit the baseline to the empirical returns. This approach for updating the baseline will be kept throughout this monograph.

Architectures

All policies throughout this monograph were implemented using *multilayer perceptron* architectures, illustrated in figures 1.2 and 1.3 for discrete and continuous action spaces respectively. Thus, the main variations between different policy parameterizations are the number of hidden layers, their respective sizes (number of units) and the type of non-linear activation for each perceptron. The approach adopted here was to use the same non-linear activation for all units in the network. This allows us to denote different architectures using the convention 'Mlp:*size-size-...:activation*', where the *sizes* are numbers indicating the size of each hidden layer, in order from input to output. This makes clear the correspondence between the results and the architectures used to achieve them in the plots that follow. The baselines use the same architecture of the policy, with the difference that the output layer is composed of a single unit with linear activation, as their purpose is to output a numerical value.

The one exception is a linear policy parameterization where each action logit or mean is given by $\boldsymbol{s}^\intercal \boldsymbol{\theta}_a$, $a \in \mathcal{A}$, where $\boldsymbol{s}$ denotes the state feature vector and the parameter vectors $\boldsymbol{\theta}_a$ are distinct. This type of policy is denoted 'Linear' in the figures that follow. The baseline used in this case is a simple multilayer perceptron with one hidden layer of 20 units with hyperbolic tangent activations, or *tanh*.

2.3.3 Experimental Results

Since the requirements for solving the environments require that the average return exceed a certain threshold over 100 consecutive episodes, we plot the 'AverageReturn' per iteration in figures 2.2 and 2.3, a running average of the returns obtained over the last 100 episodes. Each line indicates the average result over multiple trials and the area of the same color, with lower opacity, surrounding

[2]In retrospect, this was an error, as the learning rates should be considerably smaller, specially for larger networks

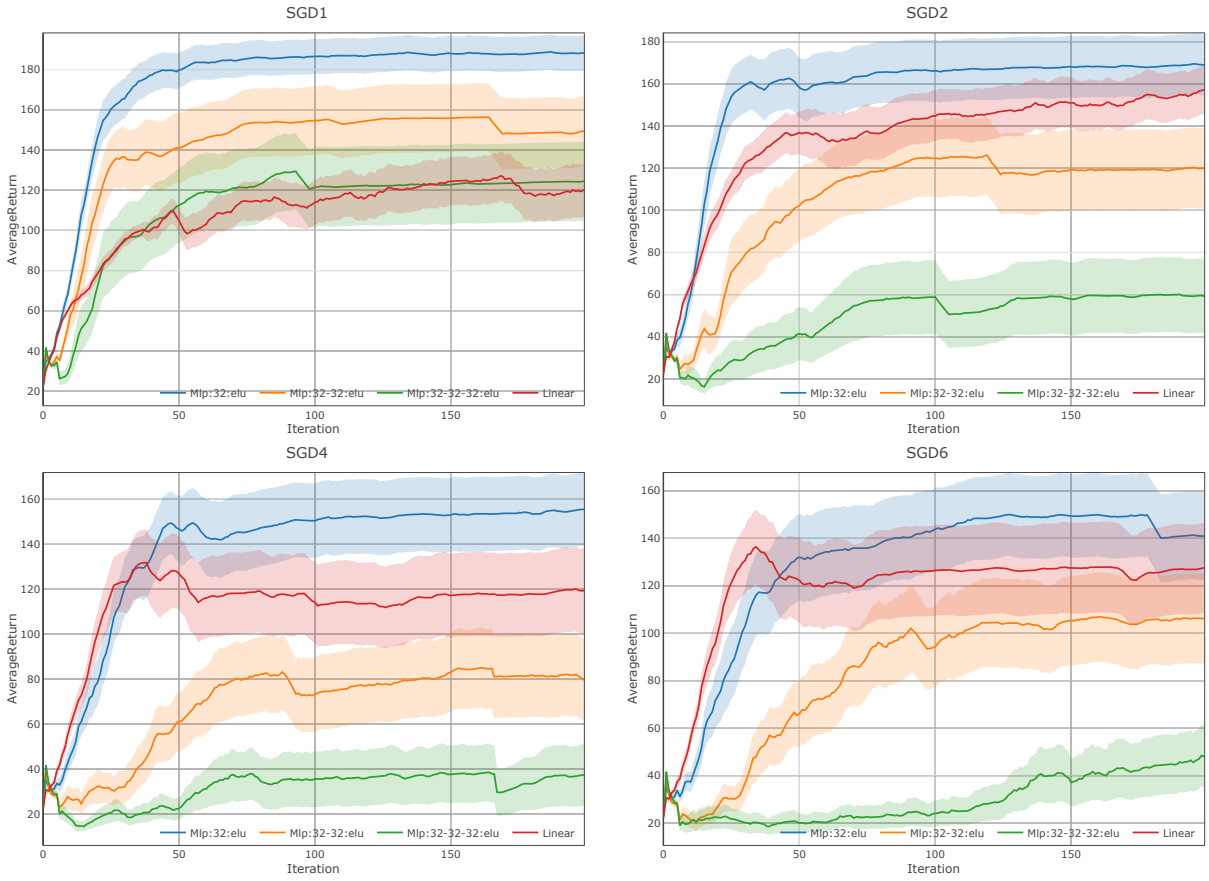


Figure 2.2: *Means and standard errors using Algorithm 1 in the* `CartPole-v0` *environment. Each line corresponds to a policy architecture, where 'Mlp' denotes Multilayer perceptron (see Figure 1.2), 'elu' denotes Exponential Linear Unit (*ELU*) and 'Linear' denotes a simple linear policy. Each plot corresponds to a step size schedule, where* SGD *stands for stochastic gradient descent and the postfix number indicates how slowly the learning rates decay.*

it denotes the sample standard error, the empirical estimate of the sampling distribution's standard deviation:

$$\text{sample standard error} = \frac{\text{sample standard deviation}}{\sqrt{\text{number of samples}}}. \tag{2.5}$$

Cartpole balancing task

For this environment, 21 trials with different initial random seeds were collected for each architecture and optimizer. Results are shown in figure 2.2. On average, the two simpler architectures outperformed the rest across all step size schedules, with the exception of the first. However, note that the results have high variance. This illustrates one of the problems of policy gradient methods: although asymptotic convergence to a local minimum is guaranteed if the gradient estimate is unbiased, its variance may occasionally cause the algorithm to compute bad steps. A bad step in parameter space, i.e., one that results in a policy with worse performance, is specially harmful in reinforcement learning applications, compared to supervised learning, since it also affects the state visitation frequencies. Thus, the new policy may receive positive rewards less often, which in turn produces even worse gradient estimates.

Mountain car task

In this task, 7 trials for each of three architectures were collected with different initial random seeds. In all of our experiments, no combination of learning rate schedule and architecture was

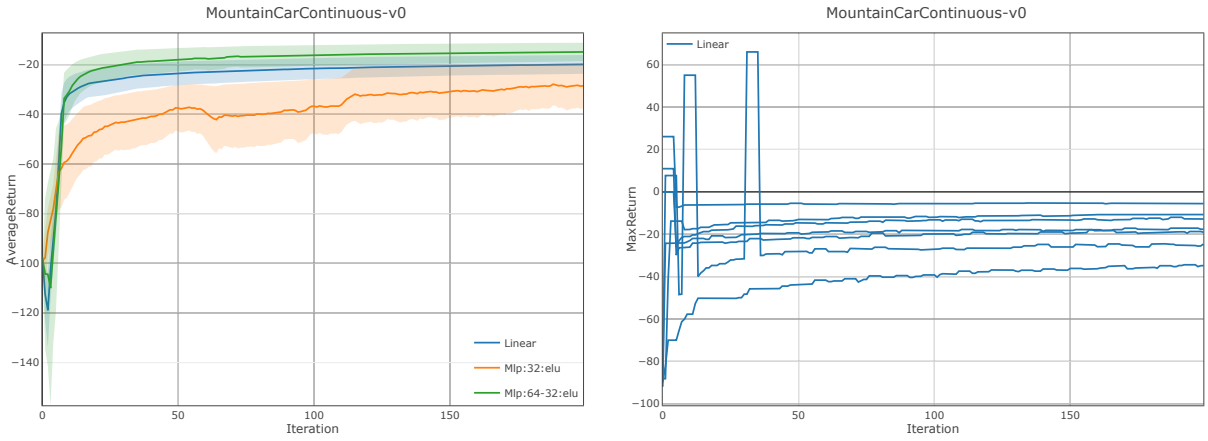


Figure 2.3: *Algorithm 1 results in the* `MountainCarContinuous-v0` *environment. Left: means and standard errors over of the average returns for different policy architectures. Right: maximum return over the 100 preceding episodes for each iteration, where each line corresponds to a different trial, all using a linear policy.*

able to obtain a positive average return. Figure 2.3 (left) illustrates this, where the learning rate schedule $\alpha_k = \frac{1}{k}$ was used. Although this might suggest that none of the agents managed to reach the top of the right mountain at some point, Figure 2.3 (right) shows that, at least in a few of the trials, this isn't true. One possible explanation is that since successful episodes are very few and far between, the contribution of data from these episodes to the gradient calculated in step 9 is negligible when combined with all the other data from unsuccessful ones. However, as we'll see later, different algorithms make it possible to obtain better results using the same policy, initial seeds and batch size.

2.4 Actor-critic Methods and Generalized Advantage Estimation

In the previous Section we saw how the noise in the policy gradient estimate can affect the performance of the policy and the speed of improvement. Actor-critic methods seek to mitigate this problem by using the learned value function $\hat{V}$ not only as a baseline, but also in the estimate of the expected return R_t, also called the *monte-carlo* estimate of $Q^\pi(s_t, a_t)$ in classic reinforcement learning literature. Specifically, they combine the observed rewards following s_t, a_t with the estimate of the value of a later state, $\hat{V}(s_{t+n})$, yielding the n-step return:

$$\hat{Q}_t^{(n)} \coloneqq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \hat{V}(s_{t+n}),$$

which is a biased estimate of $Q^\pi(s_t, a_t)$ and introduces dependence on the quality of the state-value estimator. This is a technique known as *bootstrapping* and is directly analogous to *temporal difference* methods in classic reinforcement learning literature (Sutton & Barto (2018)). In practice, such estimates, although biased, can greatly increase the speed of learning because of their lower variance, which decreases with the degree of bootstrapping: the fraction of the estimate approximated using $\hat{V}$. Thus, the later we truncate the sequence of returns, the closer the estimate is to the empirical return R_t and the higher the variance. Another benefit of using these estimates is that one doesn't need to wait for an episode to end in order to calculate the approximation of $Q^\pi(s_t, a_t)$, making it possible to collect a batch of N transitions instead of m episodes, which have variable lengths, and thus the running time of the algorithm doesn't depend on how well or poorly the agent performs in each iteration of the algorithm.

A recent method incorporating these ideas is *Generalized Advantage Estimation*, Schulman *et al.*

(2015b). Consider the setting where the objective is to maximize the *undiscounted* expected return in an episodic MDP. The policy gradient can be written as

$$\boldsymbol{g} \coloneqq \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}} \left[\sum_{t=0}^{\infty} A^{\pi}(s_t, a_t) \nabla \log \pi_{\boldsymbol{\theta}}(s_t, a_t) \right]. \tag{2.6}$$

This is just a simpler form of the extended version of Theorem 1 using $V^{\pi}(s)$ as the baseline. This form, as discussed earlier, can be estimated after collecting a batch of roll-outs to produce an unbiased estimate of the gradient, although one with high variance. In generalized advantage estimation instead, the gradient being estimated is a discounted approximation to the policy gradient, defined as follows:

$$\boldsymbol{g}^{\boldsymbol{\gamma}} \coloneqq \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}} \left[\sum_{t=0}^{\infty} A^{\pi,\gamma}(s_t, a_t) \nabla \log \pi_{\boldsymbol{\theta}}(s_t, a_t) \right], \tag{2.7}$$

where $\gamma \in [0, 1]$, $A^{\pi,\gamma}(s, a) \coloneqq Q^{\pi,\gamma}(s, a) - V^{\pi,\gamma}(s)$ and the discounted value functions $Q^{\pi,\gamma}$ and $V^{\pi,\gamma}$ are given by (1.7) and (1.6) respectively. Here, the superscript γ indicates that the discounting is a hyperparameter of the approximation, not of the problem formulation. Furthermore, note that $\boldsymbol{g}^{\boldsymbol{\gamma}}$ is not equivalent to the policy gradient for the discounted problem formulation, since it's missing a γ^t term weighting the advantage function. It follows that the variance $\boldsymbol{g}^{\boldsymbol{\gamma}}$ is lower than that of g because of the downweighting of delayed rewards.

In order to produce accurate estimates of the discounted approximation to the gradient, one needs to estimate the discounted advantage function $A^{\pi,\gamma}(s_t, a_t)$ for every timestep of the episode. Let V be an approximation of the discounted value function and define $\delta_t^V = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$. It follows that, if $V = V^{\pi,\gamma}$, then δ_t^V is an unbiased estimate of the discounted advantage function, since

$$\begin{aligned}
\mathbb{E}\left[\delta_t^V\right] &= \mathbb{E}\left[r_{t+1} + \gamma V^{\pi,\gamma}(s_{t+1}) - V^{\pi,\gamma}(s_t) \,|\, s_t, a_t\right] \\
&= \mathbb{E}\left[r_{t+1} + \gamma V^{\pi,\gamma}(s_{t+1}) \,|\, s_t, a_t\right] - V^{\pi,\gamma}(s_t) \\
&= Q^{\pi,\gamma}(a_t, s_t) - V^{\pi,\gamma}(s_t) = A^{\pi,\gamma}(s_t, a_t), \qquad \text{(eq. 1.9)}
\end{aligned}$$

where $(s_{t+1}, r_{t+1}) \sim \mathcal{P}_0(\cdot \,|\, s_t, a_t)$. However, this is only true if we have the true value function $V^{\pi,\gamma}$. Since δ_t^V is too reliant on the quality of the approximation V, because of its high degree of bootstrapping, it is worth considering the more general k-step estimates of the advantage, denoted by $\hat{A}_t^{(k)}$ and defined as follows:

$$\hat{A}_t^{(1)} \coloneqq r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \qquad = \delta_t^V \tag{2.8}$$

$$\hat{A}_t^{(2)} \coloneqq r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) - V(s_t) \qquad = \delta_t^V + \gamma \delta_{t+1}^V \tag{2.9}$$

$$\hat{A}_t^{(3)} \coloneqq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V(s_{t+3}) - V(s_t) \qquad = \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V \tag{2.10}$$

$$\vdots$$

$$\hat{A}_t^{(k)} \coloneqq r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+k} + \gamma^k V(s_{t+k}) - V(s_t) = \sum_{l=0}^{k} \gamma^l \delta_{t+l}^V. \tag{2.11}$$

Note that this is analogous to the n-step return definition. Moreover, as $k \to \infty$, we obtain

$$\hat{A}_t^{(\infty)} = R_t^{\gamma} - V(s_t) = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V,$$

where R_t^{γ} is the return as defined in (1.4). It follows that $\hat{A}_t^{(\infty)}$ can be used in place of the true discounted advantage function in timestep t without introducing bias into the discounted gradient estimate, by the following proposition (where the arguments $s_{i:j}$, $a_{i:j}$ enumerate which trajectory

variables the functions may depend on).

Proposition 1

Suppose that $\hat{A}_t$ can be written in the form $\hat{A}_t(s_{0:\infty}, a_{0:\infty}) = Q_t(s_{0:\infty}, a_{0:\infty}) - b(s_{0:s_t}, a_{0:t-1})$, such that for all (s_t, a_t), $\mathbb{E}_{s_{t+1:\infty}, a_{t+1:\infty}}\left[Q_t(s_{0:\infty}, a_{0:\infty}) \mid s_t, a_t\right] = Q^{\pi,\gamma}(s_t, a_t)$. Then

$$\mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}}\left[\hat{A}_t(s_{0:\infty}, a_{0:\infty}) \nabla \log \pi_{\boldsymbol{\theta}}(s_t, a_t)\right] = \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}}\left[A^{\pi,\gamma}(s_t, a_t) \nabla \log \pi_{\boldsymbol{\theta}}(s_t, a_t)\right]$$

Proof. See Schulman *et al.* (2015b). □

The generalized advantage estimator is an exponentially weighted average of all the k-step estimates, parameterized by $\gamma, \lambda \in [0, 1]$, defined as follows:

$$\begin{aligned}
\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} &:= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \cdots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda\left(\delta_t^V + \gamma \delta_{t+1}^V\right) + \lambda^2\left(\delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V\right) + \cdots\right) \\
&= (1-\lambda)\big(\delta_t^V\left(1 + \lambda + \lambda^2 + \cdots\right) + \gamma \delta_{t+1}^V\left(\lambda + \lambda^2 + \lambda^3 + \cdots\right) + \\
&\qquad \gamma^2 \delta_{t+2}^V\left(\lambda^2 + \lambda^3 + \lambda^4 + \cdots\right) + \cdots\big) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma \delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2 \delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \cdots\right) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V. \qquad (2.12)
\end{aligned}$$

This is closely analogous to the λ-return defined in the TD(λ) algorithm (Sutton & Barto (2018)), although there the objective is to obtain an estimate of the state-value function whereas here the advantage function is of concern. This remarkably simple expression for the estimator has two notable cases, corresponding to the values of $\lambda = 0$ and $\lambda = 1$:

$$\begin{aligned}
\hat{A}_t^{\mathrm{GAE}(\gamma,0)} &:= \delta_t^V &&= r_{t+1} + \gamma V(s_{t+1}) - V(s_t), \\
\hat{A}_t^{\mathrm{GAE}(\gamma,1)} &:= \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V &&= R_t^\gamma - V(s_t);
\end{aligned}$$

The first is the low-variance, high-bias estimate of the advantage which is highly dependent on the quality of V as an estimator of $V^{\pi,\gamma}$. The second is an unbiased, high-variance estimate which satisfies Proposition 1 regardless of V. A value of $0 < \lambda < 1$ is a compromise between the two, therefore λ allows for flexible adjustment of the degree of bootstrapping. It's important to note that this is different from the role of γ, which is to control the magnitude of the estimate by downweighting delayed rewards.

Algorithm 2 implements the vanilla policy gradient together with generalized advantage estimation, therefore it is a procedure for optimizing a policy with respect to the expected total reward. As alluded to earlier, it collects N transitions instead of m episodes. This means that some trajectories might not be completed, i.e., the collecting of transitions might stop before a terminal state is reached. Since generalized advantage estimation already relies on the quality of the value function estimator V, the approach adopted here is to use bootstrapping to calculate R_t^γ for the unfinished trajectories:

$$R_t^\gamma \approx \sum_{l=t}^{T-1} \gamma^{l-t} r_{l+1} + \gamma^{T-t} V(s_T),$$

with the understanding that T is the length of the particular trajectory and equality holds when s_T is terminal and $V(s_T) = 0$. The benefit of this approach is that each iteration of the outer loop

has fixed time period, assuming each transition has a fixed computational cost. Moreover, note that for each trajectory the residuals δ_t^V can be precomputed in parallel, which allows the estimators $\hat{A}_t^{\text{GAE}(\gamma,\lambda)}$ to be computed by a single backwards pass through the trajectory timesteps.

Algorithm 2 Vanilla + GAE

Input: Parameterized policy $\pi_{\boldsymbol{\theta}}$, learning rate schedule (α_k), baseline V, GAE parameters γ, λ

Initialize policy parameter vector $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, baseline V
for k=1, 2, 3, ... **do**
  Collect N transitions (s_n, a_n, r_{n+1}) following $\pi_{\boldsymbol{\theta}_k}$
  for each trajectory $(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T)$ **do**
    if final state s_T is terminal **then**
      $V(s_T) \leftarrow 0$
    end if
    Compute $R_t^\gamma = \sum_{l=t}^{T-1} \gamma^{l-t} r_{l+1} + \gamma^{T-t} V(s_T)$ for every timestep
    Compute $\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{T-1} (\gamma\lambda)^l \delta_{t+l}^V$ for every timestep
  end for
  Fit baseline by minimizing $\|V(s_n) - R_n^\gamma\|^2$ summed over all timesteps
  $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \alpha_k \frac{1}{N} \sum_{n=1}^{N} \hat{A}_n^{\text{GAE}(\gamma,\lambda)} \nabla \log \pi_{\boldsymbol{\theta}}(a_n \mid s_n)$
end for

2.5 Experiments with GAE

The goal in the next few experiments is to explore the following questions:

- What's the influence of γ and λ in the learning process?
- Is it possible to obtain near-optimal policies with generalized advantage estimation and larger networks?

2.5.1 Experiment Settings

To compare the influence of the GAE parameters, Algorithm 2 was implemented in the cartpole balancing task using a multilayer perceptron policy with a single hidden layer of 32 units with *elu* activations, given that this architecture was the best performing, on average, in the previous experiments. The learning rate schedule (α_k) used for this task was $\alpha_k = \frac{1}{k}$. The number of environments given in the table to the right is relevant when doing sample batches, since more environments with the same number of samples may lead to more trajectories left unfinished.

Algorithm 2 hyperparameters	
Iterations	100
Batch size	2000 timesteps
Envs. in parallel	8

The other experiments used a bigger network with 2 hidden layers of 64 *elu* units each, which correspond to a total of 4,096 parameters just between the two hidden layers. In order to make the most out of this architecture and obtain better results, the learning rates were handled using the Adam optimizer by Kingma & Ba (2014). Adam is a stochastic optimization method requiring only gradient estimates which implements an adaptive learning rate schedule and, as claimed by the authors, is well-suited for problems using a large number of parameters. The only hyperparameter required is the initial learning rate which was set to 0.01, denoted 1e-2 in the plots.

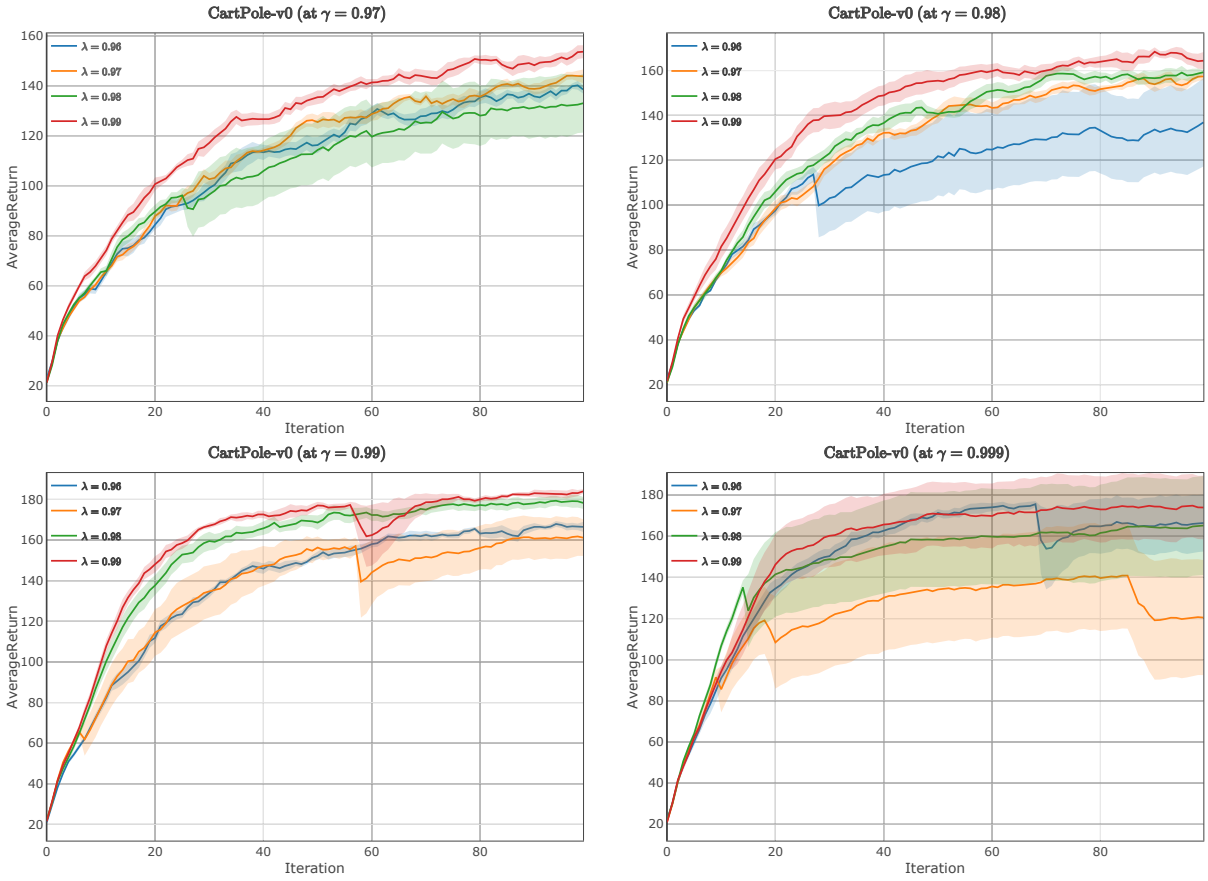


Figure 2.4: *Means and standard errors of learning curves in the cartpole task using generalized advantage estimation with varying values of γ and λ.*

2.5.2 Experiment Results

GAE parameters comparison

Figure 2.4 shows the results of a total of 16 different combinations of the GAE parameters γ and λ averaged across 7 different trials for each. In general, as γ increases, the average return increases as well. This is understandable for this particular environment because the returns are directly related to the length of the episode, therefore the downweighting of delayed rewards introduced by gamma decreases the value of later states. The higher standard error shown when using $\gamma = 0.999$ illustrates its role in reducing the variance of the estimator. As noted by Schulman *et al.* (2015b), the best results use intermediate values of γ and λ. The higher standard error at $\gamma = 0.98$ and $\lambda = 0.97$ is due to one of the trials having a premature drop in the average policy *entropy* (see Figure 2.5), defined as

$$H(\pi) = \mathbb{E}_{s\sim d^{\pi}}\left[-\sum_{a\in\mathcal{A}} \pi(a \mid s) \log \pi(a \mid s)\right], \tag{2.13}$$

which is a way of measuring how action probabilities are spread. Lower values indicate that the average probability is concentrated in a few of the actions (in this case, one action). Thus, the policy keeps choosing the same action regardless of the average return it incurs. It's unclear what causes this behaviour, which is observed in a few of our experiments.

Since actor-critic methods are more reliant on the quality of the value function estimator, the *explained variance* of the baseline was used to judge its effectiveness, defined as

$$1 - \frac{\mathrm{Var}\left[R_t - b(s_t)\right]}{\mathrm{Var}\left[R_t\right]}, \tag{2.14}$$

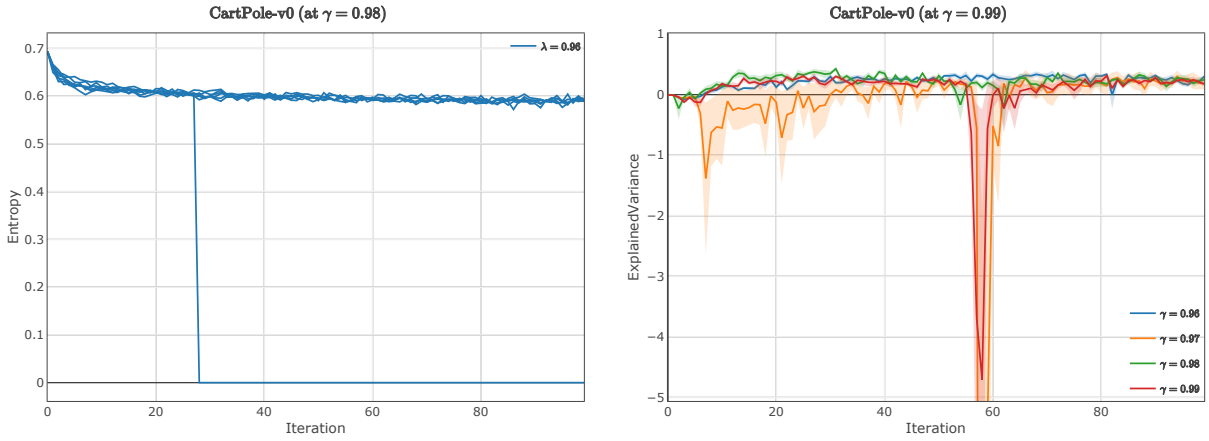


Figure 2.5: *Left: average entropy curves for individual trials in the cartpole task using generalized advantage estimation with* $\gamma = 0.98$ *and* $\lambda = 0.97$*. Right: average explained variance of the baselines for* $\gamma = 0.99$ *and different values of* λ*.*

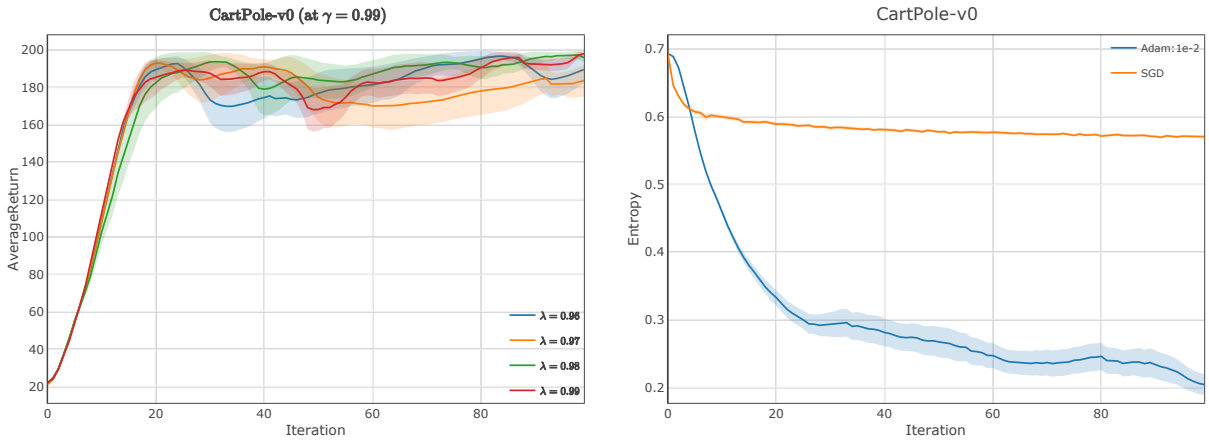


Figure 2.6: *Left: Means and standard errors for the cartpole balancing task using the Adam optimizer and generalized advantage estimation with different values of* λ*. Right: average entropy comparison between experiments using the Adam optimizer and stochastic gradient descent with learning rate schedule* $\alpha_k = \frac{1}{k}$*.*

where the random variables from which the variances are calculated are the ones sampled in a single iteration of Algorithm 2. This gives a quantity at every iteration which indicates how close the values predicted by the baseline are to the empirical returns collected in that iteration, normalized by the observed variance of the returns. Generally, a positive explained variance indicates that the baseline is a good fit to the data, whereas negative values are observed otherwise. Figure 2.5 (right) shows the explained variance per iteration for the trials using $\gamma = 0.99$. Note that the drop in explained variance for $\lambda = 0.97$ and $\lambda = 0.99$ coincides with the drop in performance shown in the corresponding plot in Figure 2.4.

Adam with large network

The average returns obtained using the larger network with the Adam method are shown in Figure 2.6 (right). The results were averaged over 7 different trials for each combination of $\gamma = 0.99$ and λ with values $0.96, 0.97, 0.98, 0.99$. With the exception of $\lambda = 0.97$, for all the other values of λ at least 4 trials achieved the maximum return of 200. Figure 2.6 (right) shows the difference in the average policy entropy between the aforementioned experiments and the ones using the simple stochastic gradient descent method of Figure 2.4, with the same values of γ and λ. We see that the final policies obtained using the Adam optimizer with the bigger network are more deterministic

on average, which is in line with the theory since optimal policies are greedy with respect to the optimal action values Q^*.

Chapter 3

Natural Policy Gradient

Although policy gradients have strong convergence properties and allow for flexibility in their implementation, they tend to converge very slowly and sometimes reach suboptimal plateaus of performance. Moreover, the vanilla gradient is *non-covariant*, i.e., a simple reparameterization of the policy may lead to a different gradient direction.

Natural gradients are a covariant gradient ascent rule that define the ascent direction with respect to the distance on the manifold induced by the parameters, rather than the parameters themselves. This is a very well studied problem in pattern recognition and statistical inference. Kakade (2002) introduced this to reinforcement learning and defined the natural policy gradient. This direction has interesting theoretical properties and its use in large scale problems has shown strong empirical performance.

3.1 Trajectory Distribution Manifold

Recall that our performance measure, $J(\boldsymbol{\theta}) = \mathbb{E}_{s_{0:\infty}, a_{0:\infty}}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t)\right]$, is defined as a function of the policy parameters. However, it is also possible to think of the return as a function on the set of probability distributions of trajectories, with the expected return defined as an expectation over the space of trajectories.

Definition 4: Trajectory space and probability

In a countable MDP, a trajectory τ of length n is defined as a sequence of states and actions of length n starting at an initial state-action pair s_0, a_0 and ending at a state s_n:

$$\tau = (s_0, a_0, s_1, \ldots, s_{n-1}, a_{n-1}, s_n), \quad |\tau| = n. \tag{3.1}$$

The space of all such trajectories of length n is defined by $\mathcal{T}^n = (\mathcal{S}, \mathcal{A})^n \times \mathcal{S}$. Given a policy π and an initial state distribution ρ_0, the probability of a trajectory τ of any length is simply the probability of sampling each state and action sequentially following π:

$$Pr(\tau \,|\, \pi) = \rho_0(s_0) \prod_{t=0}^{|\tau|-1} \mathcal{P}(s_{t+1} \,|\, s_t, a_t)\pi(a_t \,|\, s_t), \quad \tau \in \mathcal{T}^n. \tag{3.2}$$

The discounted trajectory distribution is defined over the set of all trajectories of length $1, \ldots, n$, denoted $\mathcal{T}_+^n = \bigcup_{t=1}^n \mathcal{T}^n$, each with probability

$$Pr^\gamma(\tau \,|\, \pi) = \frac{1-\gamma}{1-\gamma^n} \gamma^{|\tau|-1} Pr(\tau \,|\, \pi), \quad \tau \in \mathcal{T}_+^n. \tag{3.3}$$

It is easy to see for an episodic MDP with episodes of length up to n that definition 4 allows us

to rewrite the expected return as

$$\mathbb{E}_{\substack{s_{0:n-1}\\a_{0:n-1}}}\left[\sum_{t=0}^{n-1}\gamma^t\mathcal{R}(s_t,a_t)\right]=\mathbb{E}_{\tau\in\mathcal{T}^n}\left[\sum_{t=0}^{n-1}\gamma^t\mathcal{R}(s_t,a_t)\right], \tag{3.4}$$

where $\tau\sim Pr(\cdot|\pi)$, which makes explicit that the expected return can be seen as a function on the set of probability distributions over trajectories, induced by π and the MDP. For the infinite-horizon, discounted case, the following proposition makes the same relation explicit by using a different trajectory distribution.

Proposition 2: Expected return over trajectories

Fix an MDP $(\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R},\rho_0)$ and a policy π. Then if $\gamma\in(0,1)$ the following holds:

$$J(\pi)=\mathbb{E}_{\substack{s_{0:\infty}\\a_{0:\infty}}}\left[\sum_{t=0}^{\infty}\gamma^t\mathcal{R}(s_t,a_t)\right]=\frac{1}{1-\gamma}\lim_{n\to\infty}\mathbb{E}_{\tau\in\mathcal{T}_+^n}\left[\mathcal{R}(s_{|\tau|-1},a_{|\tau|-1})\right],$$

where $\tau\sim Pr^{\gamma}(\cdot\,|\,\pi)$.

The proof is given in appendix A.2. Thus, for each choice of parameter vector $\boldsymbol{\theta}$ corresponds a probability distribution $Pr(\cdot\,|\,)$ over the space of trajectories, and the set of all such distributions defines a *manifold* $S=\{Pr(\cdot\,|\,\pi_{\boldsymbol{\theta}});\ \boldsymbol{\theta}\in\mathbb{R}^d\}$ (a statistical manifold in this case) with coordinate system $\varphi:\mathbb{R}^d\to S$. A manifold is called *differentiable* if the mapping from the coordinate space to the set (in this case, $Pr(\cdot\,|\,\pi_{\boldsymbol{\theta}})$) is differentiable. The study of parameterized manifolds in general is the domain of differential geometry, and *information geometry* uses its tools on problems from a variety of fields, e.g., statistics, information theory and control theory, in order to visualize them geometrically and from that develop new tools and insights. Of special importance to reinforcement learning and machine learning in general is the derivation of the true steepest ascent direction when the parameter space has such an underlying structure, called the *natural gradient*.

3.2 Natural Gradient

Given a function $L(\boldsymbol{\xi}):\Xi\to\mathbb{R}$, $\Xi\subset\mathbb{R}^d$, the steepest ascent direction with respect to the current parameters is defined as

$$\max_{\Delta\boldsymbol{\xi}}L(\boldsymbol{\xi}+\Delta\boldsymbol{\xi})$$

subject to

$$\|\Delta\boldsymbol{\xi}\|^2<\epsilon,$$

where ϵ is an infinitesimal. When the parameter space on which the function L is defined is a Euclidean space with an orthonormal coordinate system, the squared norm above is the Euclidean norm, $\sum_{i=0}^{d}\Delta\xi_i^2$. In that case, the steepest ascent direction is the gradient $\nabla L(\boldsymbol{\xi})$.

However, the parameter space may not be the canonical Euclidean space and L may be defined on a differentiable manifold S with coordinate system $\varphi:\Xi\to S$. If, additionally, an inner product $\langle\ ,\ \rangle_G$ is defined on the *tangent space* of S, the local linear approximation to the manifold at any point $\varphi(\boldsymbol{\xi})$[1], then the squared length is given by

$$\|\Delta\boldsymbol{\xi}\|^2=\sum_{i,j}g_{ij}(\boldsymbol{\xi})\Delta\xi_i\Delta\xi_j.$$

[1]This may be getting into too much detail, as there's a whole chapter in the book by Amari & Nagaoka (2007) introducing these ideas. The key point to take away here is that an inner product defined on this space gives a local approximation between two points in the manifold, given their coordinates.

Such a manifold is called a *Riemannian manifold* (or space) and the matrix $G = (g_{ij})$ is called the Riemannian metric tensor, which depends on $\boldsymbol{\xi}$ in general and must be symmetric positive definite. Amari (1998) argued that most supervised learning problems have this structure and named the steepest ascent direction in such a space the natural gradient, giving the following theorem and proof, adapted from the original paper.

Theorem 2: Natural gradient

In a Riemannian space, the steepest ascent direction of $L(\boldsymbol{\xi})$ is given by

$$\widetilde{\nabla} L(\boldsymbol{\xi}) = G^{-1} \nabla L(\boldsymbol{\xi}), \tag{3.5}$$

where G^{-1} is the inverse of the Riemannian metric tensor and ∇J is the conventional gradient.

Proof. We put $\Delta\boldsymbol{\xi} = \epsilon \boldsymbol{d}$ and search for the direction $\boldsymbol{d}$ that maximizes

$$\lim_{\epsilon \to 0} \frac{L(\boldsymbol{\xi} + \epsilon \boldsymbol{d}) - L(\boldsymbol{\xi})}{\epsilon} = \nabla L(\boldsymbol{\xi})^\intercal \boldsymbol{d},$$

subject to

$$\|\boldsymbol{d}\|_G^2 = 1,$$

where $\|\boldsymbol{d}\|_G^2 = \boldsymbol{d}^\intercal G \boldsymbol{d}$. Any solution to the constrained optimization problem above must satisfy the KKT conditions[2]: for some $\lambda \in \mathbb{R}$,

$$\begin{aligned} \nabla_{\boldsymbol{d}} \left(\nabla L(\boldsymbol{\xi})^\intercal \boldsymbol{d} - \lambda \boldsymbol{d}^\intercal G \boldsymbol{d} \right) &= 0, \\ \boldsymbol{d}^\intercal G \boldsymbol{d} &= 1, \\ \lambda(\boldsymbol{d}^\intercal G \boldsymbol{d} - 1) &= 0; \end{aligned}$$

which gives

$$\begin{aligned} &\nabla L(\boldsymbol{\xi}) = \lambda 2 G \boldsymbol{d} \\ &\implies \boldsymbol{d} \propto G^{-1} \nabla L(\boldsymbol{\xi}). \end{aligned}$$

□

Following the work of Amari regarding natural gradients in supervised learning applications, Kakade (2002) proposed the *natural policy gradient*, which already showed promising signs of improvement by achieving strong empirical results in the complicated MDP of Tetris, a challenge at the time. However, Kakade's work lacked a proper differentiable manifold and distance metric, since it was originally considered that the policy mapped the parameters to a collection of statistical manifolds, the sets of action probability distributions for each state of the MDP. Later, Bagnell & Schneider (2003) defined a proper manifold, the set of distributions over trajectories induced by the policy and the MDP. This is made explicit by definition 4 and allows for the derivation of an important metric, which we briefly introduce in the next section.

3.3 Fisher Information Metric

Consider the general case: a statistical model over a set $\mathcal{X}$, that is, a parameterized probability distribution function $p_{\boldsymbol{\xi}}$, $\boldsymbol{\xi} \in \Xi \subset \mathbb{R}^n$, over $\mathcal{X}$. In this setting, the *Fisher information matrix* is defined as

$$G(\boldsymbol{\xi}) = \mathbb{E}_{p_{\boldsymbol{\xi}}} \left[\nabla \log p(x; \boldsymbol{\xi}) \nabla \log p(x; \boldsymbol{\xi})^\intercal \right], \tag{3.6}$$

where the gradient is implicitly with respect to $\boldsymbol{\xi}$ and $\mathbb{E}_{p_{\boldsymbol{\xi}}}$ denotes expectation w.r.t. to the distribution p with coordinates $\boldsymbol{\xi}$ (in the language of manifolds), $\mathbb{E}_{p_{\boldsymbol{\xi}}}[f(x)] \coloneqq \int f(x) p(x; \boldsymbol{\xi}) dx$. We abuse

[2] see the book by Wright & Nocedal (1999) for a detailed view of the KKT optimality conditions.

notation here by using the integral expression for both countable and continuous sets, and $p_{\boldsymbol{\xi}}$ for both probability mass and density functions. An alternative form for this matrix can be derived by noting that

$$\int_{\mathcal{X}} p(x;\boldsymbol{\xi})\nabla \log p(x;\boldsymbol{\xi})dx = \int_{\mathcal{X}} \nabla p(x;\boldsymbol{\xi})dx = \nabla \int_{\mathcal{X}} p(x;\boldsymbol{\xi})dx = \nabla \mathbf{1} = 0,$$

therefore

$$\begin{aligned} 0 &= \nabla \int_{\mathcal{X}} p(x;\boldsymbol{\xi})\nabla \log p(x;\boldsymbol{\xi})dx \\ &= \int_{\mathcal{X}} p(x;\boldsymbol{\xi})\nabla \log p(x;\boldsymbol{\xi})\nabla \log p(x;\boldsymbol{\xi})^{\intercal} + \int_{\mathcal{X}} p(x;\boldsymbol{\xi})\nabla^2 \log p(x;\boldsymbol{\xi})dx \\ &\iff G(\boldsymbol{\xi}) = -\mathbb{E}_{p_{\boldsymbol{\xi}}}\left[\nabla^2 \log p(x;\boldsymbol{\xi})\right], \end{aligned} \tag{3.7}$$

assuming the order of integration and differentiation can be freely rearranged. This matrix is symmetric and also positive semidefinite: for any vector $\boldsymbol{d} \in \mathbb{R}^n$,

$$\boldsymbol{d}^{\intercal} G(\boldsymbol{\xi})\boldsymbol{d} = \int_{\mathcal{X}} p(x;\boldsymbol{\xi}) \left(\boldsymbol{d}^{\intercal}\nabla \log p(x;\boldsymbol{\xi})\right)^2 dx \geq 0.$$

We assume further that it is positive definite, which is equivalent to saying that the vectors $\nabla \log p(x;\boldsymbol{\xi})$, when seen as functions on $\mathcal{X}$, are linearly independent. Under these assumptions, G is a Riemannian metric on $S = \{p_{\boldsymbol{\xi}} \mid \boldsymbol{\xi} \in \Xi\}$, also called the Fisher information metric, and (S, G) defines a Riemannian manifold.

Although this choice is not unique, the Fisher information metric has extensive theory behind it and a collection of interesting properties, some of which that are unique to it, that make it appealing to use. Among them is the fact that the metric is invariant under coordinate transformation, i.e., if the difference between probability distributions p and p' is measured by the inner product $\langle\ ,\ \rangle_G$ in two different parameterizations of p, e.g. $\varphi : S \to \Xi$ and $\psi : S \to \Psi$, $\Xi, \Psi \subset \mathbb{R}^d$, the difference will have the same size in both. Furthermore, the metric can be derived by considering "distances" on probability distribution spaces. The relative entropy, or Kullback-Leiber divergence, between two distributions q and p is defined as

$$D_{\mathrm{KL}}\left(q \parallel p\right) = \int_{\mathcal{X}} q(x) \log \frac{q(x)}{p(x)} dx. \tag{3.8}$$

This is a natural divergence on changes in a distribution and is notably invariant to parameterization. Although it is clearly not symmetric in general, its second order Taylor expansion agrees with the Fisher information (up to scale): fixing a point $\boldsymbol{\xi}'$,

$$\begin{aligned} &D_{\mathrm{KL}}\left(\boldsymbol{\xi}' \parallel \boldsymbol{\xi}\right) \approx D_{\mathrm{KL}}\left(\boldsymbol{\xi}' \parallel \boldsymbol{\xi}'\right) + \left(\nabla_{\boldsymbol{\xi}} D_{\mathrm{KL}}\left(\boldsymbol{\xi}' \parallel \boldsymbol{\xi}\right)|_{\boldsymbol{\xi}=\boldsymbol{\xi}'}\right)^{\intercal}(\boldsymbol{\xi} - \boldsymbol{\xi}') + \tfrac{1}{2}(\boldsymbol{\xi} - \boldsymbol{\xi}')^{\intercal} H (\boldsymbol{\xi} - \boldsymbol{\xi}'), \\ &\quad \text{where } \nabla_{\boldsymbol{\xi}} D_{\mathrm{KL}}\left(\boldsymbol{\xi}' \parallel \boldsymbol{\xi}\right)|_{\boldsymbol{\xi}=\boldsymbol{\xi}'} = -\int_{\mathcal{X}} p_{\boldsymbol{\xi}'}(x)\nabla_{\boldsymbol{\xi}'} \log p_{\boldsymbol{\xi}'}(x)dx = -\int_{\mathcal{X}} \nabla_{\boldsymbol{\xi}'} p_{\boldsymbol{\xi}'}(x)dx = 0 \\ &\quad \text{and } H = \nabla^2_{\boldsymbol{\xi}} D_{\mathrm{KL}}\left(\boldsymbol{\xi}' \parallel \boldsymbol{\xi}\right)|_{\boldsymbol{\xi}=\boldsymbol{\xi}'} = -\int_{\mathcal{X}} p_{\boldsymbol{\xi}'}(x)\nabla^2_{\boldsymbol{\xi}'} \log p_{\boldsymbol{\xi}'}(x)dx = -\mathbb{E}_{p_{\boldsymbol{\xi}'}}[\nabla^2_{\boldsymbol{\xi}} \log p(x;\boldsymbol{\xi})], \end{aligned} \tag{3.9}$$

where we overload notation by letting $D_{\mathrm{KL}}\left(\boldsymbol{\xi}' \parallel \boldsymbol{\xi}\right) \equiv D_{\mathrm{KL}}\left(p_{\boldsymbol{\xi}'} \parallel p_{\boldsymbol{\xi}}\right)$. Thus, the Fisher information matrix $G(\boldsymbol{\xi})$ gives a second order approximation of the KL divergence between $p_{\boldsymbol{\xi}}$ and a distribution $p_{\boldsymbol{\xi}'}$, $\boldsymbol{\xi}' \in \Xi$: $(\boldsymbol{\xi}' - \boldsymbol{\xi})^{\intercal} G(\boldsymbol{\xi})(\boldsymbol{\xi}' - \boldsymbol{\xi})$. For further information on this metric, also known as the Fisher-Rao metric in statistical inference, and on information geometry in general, see Amari & Nagaoka (2007).

3.4 Natural Policy Gradient

In order to define the *natural policy gradient*, we must first establish the Fisher information metric in the space of probability distributions over trajectories induced by the policy and the MDP, as given by definition 4. This turns out to have a convenient expression that can be sampled from experiences, as shown by the following theorem.

Theorem 3: Fisher information metric

Fix an MDP, a differentiable policy $\pi_{\boldsymbol{\theta}}$ and let $d^{\pi_{\boldsymbol{\theta}}}$ be as in definition 3. Then, the Fisher information metric on the statistical manifold of paths is given by

$$F(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} d^{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a \,|\, s) \left(-\nabla^2 \log \pi_{\boldsymbol{\theta}}(a \,|\, s)\right). \tag{3.10}$$

The proof is given in appendix A.3. Theorem 3 has an interesting interpretation: the Fisher information matrix on the trajectory distribution manifold becomes the average of the Fisher information matrices of the policy for each state of the MDP, weighted by their respective visitation frequencies.

Thus, given the Fisher information matrix $F(\boldsymbol{\theta})$, the natural policy gradient is defined following Theorem 2:

$$\tilde{\boldsymbol{g}} = F(\boldsymbol{\theta})^{-1} \boldsymbol{g}, \tag{3.11}$$

where $\boldsymbol{g}$ denotes the policy gradient.

3.5 Practical Considerations

Consider the episodic setting. Since the Fisher information matrix given by Theorem 3 is defined as an expectation with respect to the on-policy distribution of states and actions, a straightforward way of estimating $F(\boldsymbol{\theta})$ would be calculate the matrix $-\nabla^2 \log \pi_{\boldsymbol{\theta}}(a_n \,|\, s_n)$ for every timestep across several episodes and take the average of the results:

$$F(\boldsymbol{\theta}) \approx -\frac{1}{N} \sum_{n=0}^{N-1} \nabla^2 \log \pi_{\boldsymbol{\theta}}(a_n \,|\, s_n),$$

where n indexes over all timesteps in a batch of trajectories. However, this turns out to be impractical, since forming and storing this matrix in memory can be too costly, let alone inverting it afterwards. As an example, consider a Multilayer perceptron with two hidden layers of 64 units each: the number of connection weights between these two layers alone is 4,096. This means a policy using this architecture would have a parameter vector $\boldsymbol{\theta}$ of at least 4,096 elements, therefore the hessian would be a matrix with at least 16,777,216 entries.

To circumvent this issue, we estimate the natural policy gradient directly by using the *conjugate gradient* (CG) algorithm by Wright & Nocedal (1999). CG is an iterative procedure that approximates the solution $\tilde{\boldsymbol{x}}$ to a linear system $A\boldsymbol{x} = \boldsymbol{b}$, where $A \in \mathbb{R}^{n \times n}$ is non-singular and $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$, by repeatedly evaluating $A\boldsymbol{x}$, generating a sequence $(\boldsymbol{x}_k; k \geq 0)$ that converges to the solution $\tilde{\boldsymbol{x}}$. Thus, one only needs to provide a function $f(\boldsymbol{x}) = A\boldsymbol{x}$. We call the matrix-vector product in our setting the *Fisher vector product*, $F(\boldsymbol{\theta})\boldsymbol{v}$, since we're trying to solve the system $F(\boldsymbol{\theta})\boldsymbol{v} = \boldsymbol{g}$.

In order to approximate the Fisher vector product without forming the full matrix, note that the Fisher information matrix can be expressed as:

$$\begin{aligned} F(\boldsymbol{\theta}) &\propto \sum_{s \in \mathcal{S}} d^{\pi_{\boldsymbol{\theta}}}(s) \nabla^2_{\boldsymbol{\theta}} D_{\mathrm{KL}} \left(\pi_{\boldsymbol{\theta}'}(\cdot \,|\, s) \parallel \pi_{\boldsymbol{\theta}}(\cdot \,|\, s)\right) |_{\boldsymbol{\theta}'=\boldsymbol{\theta}} \\ &= \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}} \left[\nabla^2_{\boldsymbol{\theta}} D_{\mathrm{KL}} \left(\pi_{\boldsymbol{\theta}'}(\cdot \,|\, s) \parallel \pi_{\boldsymbol{\theta}}(\cdot \,|\, s)\right) |_{\boldsymbol{\theta}'=\boldsymbol{\theta}}\right], \end{aligned} \tag{3.12}$$

following Equation (3.9) and Theorem 3. Thus, given a batch of episodes, let $\boldsymbol{g}_{\mathrm{KL}}$ denote the gradient of the average KL divergence over the sampled states:

$$\begin{aligned}\boldsymbol{g}_{\mathrm{KL}} &= \frac{1}{N}\sum_{n=0}^{N-1}\nabla_{\boldsymbol{\theta}} D_{\mathrm{KL}}\left(\pi_{\boldsymbol{\theta}'}(\cdot\,|\,s)\parallel\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)\right)|_{\boldsymbol{\theta}'=\boldsymbol{\theta}} \\ &= \nabla_{\boldsymbol{\theta}}\frac{1}{N}\sum_{n=0}^{N-1} D_{\mathrm{KL}}\left(\pi_{\boldsymbol{\theta}'}(\cdot\,|\,s)\parallel\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)\right)|_{\boldsymbol{\theta}'=\boldsymbol{\theta}},\end{aligned}$$

where n indexes over all timesteps in the batch of episodes. Then, the Fisher vector product is approximately the gradient of the gradient vector product, since

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\left((\boldsymbol{g}_{\mathrm{KL}})^{\intercal}\boldsymbol{v}\right) &= \left(\frac{1}{N}\sum_{n=0}^{N-1}\nabla^2_{\boldsymbol{\theta}} D_{\mathrm{KL}}\left(\pi_{\boldsymbol{\theta}'}(\cdot\,|\,s)\parallel\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)\right)|_{\boldsymbol{\theta}'=\boldsymbol{\theta}}\right)\boldsymbol{v} \\ &\approx F(\boldsymbol{\theta})\boldsymbol{v}.\end{aligned}$$

The formula above is convenient since it doesn't involve forming any matrices and the gradients can be calculated by the backpropagation algorithm. A general policy optimization procedure incorporating this method is given by Algorithm 3. Notice that the algorithm leaves room for using either the unbiased gradient, with or without a baseline, or the biased gradient estimate resulting from actor-critic methods. These have no influence in the definition of the Fisher information matrix, therefore the natural policy gradient may be calculated with respect to either of the gradient estimates.

Algorithm 3 Natural Policy Gradient

Input: Parameterized policy $\pi_{\boldsymbol{\theta}}$, learning rate schedule (α_k)

1: Initialize policy parameter vector $\boldsymbol{\theta}_0 \in \mathbb{R}^n$
2: **for** k=1, 2, 3, ... **do**
3: Collect a set of trajectories following $\pi_{\boldsymbol{\theta}_k}$
4: Compute $\hat{A}_n$ for all timesteps using any advantage estimation algorithm
5: Compute the policy gradient $\boldsymbol{g}$
6: Use CG and approximate Fisher vector products to solve $F(\boldsymbol{\theta})\tilde{\boldsymbol{g}} = \boldsymbol{g}$
7: Update the policy parameters

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \alpha_k\tilde{\boldsymbol{g}}$$

8: **end for**

3.6 Experiments

We designed a set of experiments to explore the properties of the natural policy gradient and compare the performance of the policies obtained here with the ones obtained by the vanilla policy gradient. The main focus was to test if simply substituting the original gradient in some of the experiments of chapter 2 with the natural would yield substantial improvements.

Mountain car with natural gradients

We kept the same settings used to obtain the results in Figure 2.3, but substituted the vanilla gradient estimate with the approximate result of solving for $F(\boldsymbol{\theta})\tilde{\boldsymbol{g}} = \boldsymbol{g}$ using the conjugate gradient method. Figure 3.1 shows the results for the individual trials using the linear policy architecture.

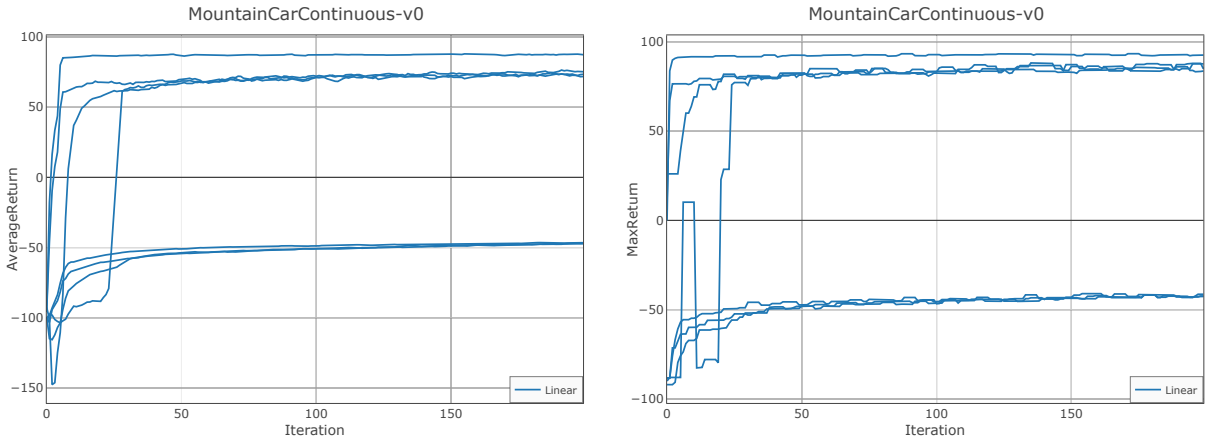


Figure 3.1: *Results from substituting the natural policy gradient for the normal one in the experiments of Figure 2.3 using the linear policy architecture. Left: average returns for individual trials. Right: maximum returns for individual trials, where each point indicates the maximum total reward obtained in the last 100 episodes.*

Unlike the vanilla policy gradient results, in all the trials which the agent managed to reach the top of the right mountain at some point, the policy achieved a positive average reward. This means that although successful episodes are rare, their contribution is not lost among the other data collected in their respective iterations. Bagnell & Schneider (2003) showed for a simple 2-state MDP that the natural policy gradient is similar to the original except that it removes the weighting by the stationary distribution of states:

$$\sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} A^{\pi_{\boldsymbol{\theta}}}(s,a)\nabla\pi_{\boldsymbol{\theta}}(a\,|\,s).$$

Although it is difficult to prove whether or not this generalizes to larger, continuous MDPs such as this environment, it would explain how the algorithm can take advantage of the limited data available from rarely visited states.

β	MeanKL
0.1	0.006
0.2	0.028
0.3	0.070
0.4	0.135
0.5	0.229
0.6	0.358

Table 3.1: *Average kl divergence between policies after one update in the natural policy gradient direction, using the same base learning rate scaled by β.*

Although these results are very impressive, problems began to arise when applying the same procedure for the two other architectures of the experiment. It turns out that this procedure is very sensitive to the learning rates α_k. In order to show this, a scaling parameter β was added to the learning rate schedule as follows:

$$\alpha_k = \beta\frac{1}{k}, \quad k \in \mathbb{N}.$$

By varying the value of β, it was observed that the average KL divergence between the initial policy and the one computed after one update in the natural policy gradient direction varied greatly, as shown in Table 3.1. As a point of reference, an average KL divergence of 0.01 between successive policies is considered acceptable and is used by Schulman *et al.* (2015a) in their experiments with

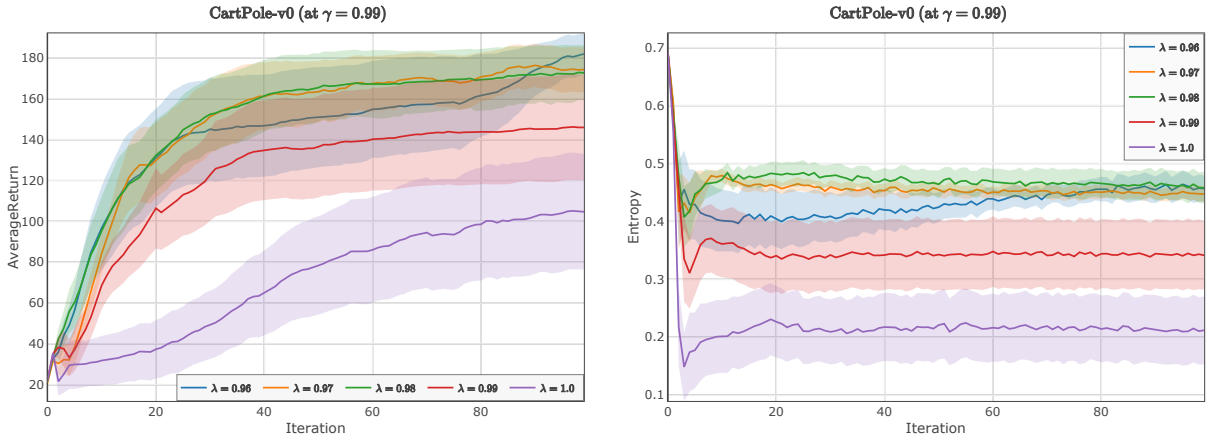


Figure 3.2: *Generalized advantage estimation with natural gradients. Comparison of average results over 7 trials for each value of λ.*

the more advanced algorithm of Trust Region Policy Optimization. This made it difficult to find an appropriate step size schedule that satisfied the RM conditions for stochastic optimization, since as the learning rate has to decrease over time, the average KL divergence between successive policies after just a few iterations becomes too little to produce any significant change in performance.

Actor-critic with natural gradients

Nevertheless, we repeated the experiments of Figure 2.4, replacing the biased gradient $\boldsymbol{g}^{\gamma}$ obtained by generalized advantage estimation by the natural gradient in the same fashion of the `MountainCarContinuous-v0` experiments. Also, trials with $\lambda = 1$ were added in order to consider estimates with no bootstrapping. Figure 3.2 shows the average results by fixing $\gamma = 0.99$. The average returns obtained with the natural gradient were, on average, worse than the ones in the original experiments. This is most likely due to the observed premature drop in entropy in the first few iterations, caused by the larger step sizes, after which it remains relatively stable. Thus, if the algorithm does not perform the right updates in the first few iterations, there is little improvement to be expected in the subsequent ones. Figure 3.2 also shows evidence of this, noticing how the worst performing configurations have, in general, the larger drops in average policy entropy in the first 5 or so iterations.

Chapter 4

Trust Region Policy Optimization

While the natural policy gradient can provide a substantial improvement over the vanilla one, it still relies on an ad-hoc choice of step-size given by stochastic gradient ascent methods, which provide no information on how to exploit the structure of the reinforcement learning problem. Trust Region Policy Optimization, introduced by Schulman *et al.* (2015a), offers a solution by considering a constrained optimization problem where the objective is to obtain the best policy with respect to the expected return, with a penalty on large deviations from the current policy. The resulting algorithm has a strong theoretical foundation and, given a large enough batch size, outperforms the other algorithms seen so far on a variety of problems.

4.1 Policy Improvement Bounds

The approach to optimizing a policy considered so far can be seen as an instance of *generalized policy iteration*, where every iteration of the algorithm consists of two phases. The first is evaluating the policy by collecting data obtained from following it, from which quantities such as the advantage for each state-action pair observed will be estimated. The second is taking the policy improvement step by shifting the parameters in the direction that is believed to produce the best improvement to the policy, with respect to the expected return criterion. In general, it is desirable to take only a small step in the direction of improvement, since both the vanilla and natural gradients are only local approximations to the objective function. However, this doesn't take into account all the information about the learning problem, and as it turns out, there's more insight into how much of an improvement one can expect depending on the size of the change in the policy.

As usual, we consider an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \rho_0)$. For now, consider the general problem of trying to find the best policy in general, not restricted to a parameterized policy class $\Pi = \{\pi_{\boldsymbol{\theta}};\ \boldsymbol{\theta} \in \mathbb{R}^d\}$, where $J(\pi) = \mathbb{E}_{s_{0:\infty}, a_{0:\infty}}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t)\right]$ is the expected return following policy π. In the context of generalized policy iteration, one seeks to find the best policy $\tilde{\pi}$ given information about the current policy π. The following identity is due to Kakade & Langford (2002) and provides useful insight into the expected return of policy $\tilde{\pi}$ using the advantage function of π:

$$
\begin{aligned}
J(\tilde{\pi}) - J(\pi) &= J(\tilde{\pi}) - \mathbb{E}_{s_0}\left[V^{\pi}(s_0)\right] \\
&= J(\tilde{\pi}) + \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}}\left[\sum_{i=1}^{\infty} \gamma^i V^{\pi}(s_i) - \sum_{j=0}^{\infty} \gamma^j V^{\pi}(s_j)\right] \\
&= J(\tilde{\pi}) + \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}}\left[\sum_{t=0}^{\infty} \gamma^{t+1} V^{\pi}(s_{t+1}) - \gamma^t V^{\pi}(s_t)\right] \\
&= \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) + \gamma^{t+1} V^{\pi}(s_{t+1}) - \gamma^t V^{\pi}(s_t)\right]
\end{aligned}
$$

$$
\begin{aligned}
&= \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi}(s_t, a_t) \right] \\
&= \sum_{s \in \mathcal{S}} \rho_{\tilde{\pi}}(s) \sum_{a \in \mathcal{A}} \tilde{\pi}(a \mid s) A^{\pi}(s, a);
\end{aligned} \tag{4.1}
$$

where the last line follows from the previous since the expectation is with respect to $s_0 \sim \rho_0$, $a_t \sim \tilde{\pi}(\cdot \mid s_t)$ and $s_{t+1} \sim \mathcal{P}(\cdot \mid s_t, a_t)$ for $t \geq 0$, taking $\rho_{\tilde{\pi}}$ as in definition 3. While this identity would allow one to evaluate the expected return of $\tilde{\pi}$ given A^{π} and $\rho_{\tilde{\pi}}$, the dependency on the latter makes this difficult to optimize in practice, since it would require trajectories sampled by following $\tilde{\pi}$.

Trust Region Policy Optimization (TRPO) considers the following approximation to the expression of $J(\tilde{\pi})$ given above:

$$
L_{\pi}(\tilde{\pi}) = J(\pi) + \sum_{s \in \mathcal{S}} \rho_{\pi}(s) \sum_{a \in \mathcal{A}} \tilde{\pi}(a \mid s) A^{\pi}(s, a). \tag{4.2}
$$

This formula is easier to optimize in practice, since it replaces the dependency on the state visitation frequencies of $\tilde{\pi}$ with that of π, requiring only trajectories sampled from π. Moreover, if the policies considered belong to a smoothly parameterized policy class $\Pi = \{\pi_{\boldsymbol{\theta}};\ \boldsymbol{\theta} \in \mathbb{R}^d\}$, then the approximation above matches (4.1) to first order, i.e., for any $\boldsymbol{\theta}' \in \mathbb{R}^d$,

$$
\begin{aligned}
L_{\pi_{\boldsymbol{\theta}'}}(\pi_{\boldsymbol{\theta}'}) &= J(\pi_{\boldsymbol{\theta}'}) \\
\nabla_{\boldsymbol{\theta}} L_{\pi_{\boldsymbol{\theta}'}}(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}=\boldsymbol{\theta}'} &= \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}=\boldsymbol{\theta}'},
\end{aligned} \tag{4.3}
$$

where the first line follows by noting that the expected advantage for each state is 0, leaving only $J(\pi_{\boldsymbol{\theta}'})$, and the second follows immediately by noting that $\sum_{s \in \mathcal{S}} \rho_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in \mathcal{A}} \nabla \pi_{\boldsymbol{\theta}}(a \mid s) A^{\pi_{\boldsymbol{\theta}}}(s, a)$ is the policy gradient using $V^{\pi}(s)$ as the baseline. This indicates that a sufficiently small change in the parameters that increases $L_{\pi_{\boldsymbol{\theta}}}$ will also increase J. However, it is not clear by how much one can change the current policy and still guarantee an improvement in J.

Schulman *et al.* (2015a) address this issue by providing an explicit lower bound on the performance of policy $\tilde{\pi}$ based on the value of L_{π} and a distance measure between $\tilde{\pi}$ and π. Specifically, the authors consider the maximum KL divergence between the policies, defined as

$$
D_{\mathrm{KL}}^{\max}(\pi \parallel \tilde{\pi}) = \max_{s \in \mathcal{S}} D_{\mathrm{KL}}\left(\pi(\cdot \mid s) \parallel \tilde{\pi}(\cdot \mid s)\right), \tag{4.4}
$$

and the maximum advantage of $\tilde{\pi}$ relative to π:

$$
\epsilon = \max_{s \in \mathcal{S}} \left| \mathbb{E}_{a \sim \tilde{\pi}(\cdot \mid s)} \left[A^{\pi}(s, a) \right] \right|. \tag{4.5}
$$

The following result gives a lower bound on the performance of $\tilde{\pi}$ based on the surrogate objective $L_{\pi}(\tilde{\pi})$ and the two previous definitions. We defer the proof to the original paper since it is very involved and out of the scope of this text[1].

Theorem 4: Policy improvement bounds

Let $D_{\mathrm{KL}}^{\max}(\pi \parallel \tilde{\pi})$ denote the maximum KL divergence and ϵ denote the maximum advantage between the policies $\tilde{\pi}$ and π. Then the following equation holds:

$$
J(\tilde{\pi}) \geq L_{\pi}(\tilde{\pi}) - \frac{2\epsilon\gamma}{(1-\gamma)^2} D_{\mathrm{KL}}^{\max}(\pi \parallel \tilde{\pi}). \tag{4.6}
$$

Theoretically, a simple policy optimization procedure based on Theorem 4 would work as follows:

1. pick an initial policy π_0;

[1] The results in the original paper are with respect to the expect discounted cost objective, which is equivalent to minimizing the expected negative return

2. for iteration $k = 1, 2, \ldots$ do

   (a) compute all the advantages $A^{\pi_k}(s, a)$;

   (b) solve the penalized optimization problem:

$$\pi_{k+1} = \arg\max_{\pi} L_{\pi_k}(\pi) - \frac{2\epsilon\gamma}{(1-\gamma)^2} D_{\mathrm{KL}}^{\max}(\pi_k \parallel \pi);$$

which is guaranteed to generate a sequence of monotonically improving policies $J(\pi_0) \leq J(\pi_1) \leq J(\pi_2) \leq \ldots$. This can be shown by letting $M_{\pi_k}(\pi) = L_{\pi_k}(\pi) - \frac{2\epsilon\gamma}{(1-\gamma)^2} D_{\mathrm{KL}}^{\max}(\pi_k \parallel \pi)$ and noting that

$$\begin{aligned} J(\pi) &\geq M_{\pi_k}(\pi) && \text{by Theorem 4,} \\ J(\pi_k) &= M_{\pi_k}(\pi_k), && \text{therefore} \\ J(\pi_{k+1}) - J(\pi_k) &\geq M_{\pi_k}(\pi_{k+1}) - M_{\pi_k}(\pi_k). \end{aligned}$$

This means that by maximizing $M_{\pi_k}(\pi)$ at every iteration, the expected return of consecutive policies in non-decreasing. The next Section discusses TRPO, an algorithm that approximates this procedure, and how the ideas discussed so far can be implemented in practice.

4.2 TRPO update

We now return to the problem of finding approximately optimal policies within a smoothly parameterized policy class $\Pi = \{\pi_{\boldsymbol{\theta}};\ \boldsymbol{\theta} \in \mathbb{R}^d\}$. Since a policy $\pi_{\boldsymbol{\theta}} \in \Pi$ can be identified by its parameter vector $\boldsymbol{\theta}$, we overload the notation of the previous Section by substituting $\boldsymbol{\theta}$ for $\pi_{\boldsymbol{\theta}}$, therefore the optimization problem at every iteration becomes

$$\boldsymbol{\theta}_{k+1} = \arg\max_{\boldsymbol{\theta}} L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) - \frac{2\epsilon}{(1-\gamma)^2} D_{\mathrm{KL}}^{\max}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta}). \tag{4.7}$$

However, the expression above poses two problems: (a) as $\gamma \to 1$, the penalty becomes large, therefore the maximum KL divergence between the policies must be too small in order to guarantee improvement; (b) the maximum KL constraint is difficult to estimate in practice, since it involves a maximization over the entire state space.

TRPO addresses these concerns by substituting the penalty in 4.7 by a constraint on the average KL divergence between the policies, resulting in the following approximation to the optimization problem:

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \arg\max_{\boldsymbol{\theta}} L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) \\ \text{subject to } & \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta}) \leq \delta, \end{aligned} \tag{4.8}$$

where

$$\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta}) = \mathbb{E}_s\left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot \mid s) \parallel \pi_{\boldsymbol{\theta}}(\cdot \mid s))\right], \quad s \sim d^{\pi_{\boldsymbol{\theta}_k}},$$

denotes the average KL divergence between the policies. In the context of non-linear optimization, the constraint in 4.8 defines a *trust region* in which the surrogate objective $L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta})$ can be used as an approximation to $J(\boldsymbol{\theta})$. This formula is more convenient since the constraint is defined as an expectation over states sampled by following $\pi_{\boldsymbol{\theta}_k}$, which is easier to estimate in practice.

In order to define a solution to problem 4.8, TRPO uses a linear approximation to the objective and a quadratic approximation to the constraint, following the *Taylor* expansions:

$$L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) \approx L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}_k) + (\nabla_{\boldsymbol{\theta}} L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_k})^{\intercal}(\boldsymbol{\theta} - \boldsymbol{\theta}_k), \tag{4.9}$$

$$\begin{aligned} \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta}) \approx \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta}_k) + \left(\nabla_{\boldsymbol{\theta}} \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta})\,|_{\boldsymbol{\theta}=\boldsymbol{\theta}_k}\right)^{\intercal}(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \\ \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^{\intercal} \nabla_{\boldsymbol{\theta}}^2 \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta})\,|_{\boldsymbol{\theta}=\boldsymbol{\theta}_k}(\boldsymbol{\theta} - \boldsymbol{\theta}_k). \end{aligned} \tag{4.10}$$

Using the approximations above, the problem becomes finding $\boldsymbol{\theta}_{k+1}$ such that

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \arg\max_{\boldsymbol{\theta}} \boldsymbol{g}^\intercal(\boldsymbol{\theta} - \boldsymbol{\theta}_k) \\ \text{subject to } & \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^\intercal F(\boldsymbol{\theta}_k)(\boldsymbol{\theta} - \boldsymbol{\theta}_k) \leq \delta, \end{aligned} \tag{4.11}$$

where $\boldsymbol{g}$ is the policy gradient of $\pi_{\boldsymbol{\theta}_k}$, which follows from equation (4.3), and $F(\boldsymbol{\theta}_k)$ is the Fisher information matrix, which follows from Equation (3.9). The problem above is equivalent to searching for the vector $\boldsymbol{d\theta} = \boldsymbol{\theta} - \boldsymbol{\theta}_k$ which maximizes the inner product with the policy gradient of $\pi_{\boldsymbol{\theta}_k}$, constrained by the value of $\boldsymbol{d\theta}^\intercal F(\boldsymbol{\theta}_k)\boldsymbol{d\theta}$. Then, the *Lagrangian function* of this problem is defined as

$$\mathcal{L}(\boldsymbol{d\theta}, \lambda) = -\boldsymbol{g}^\intercal \boldsymbol{d\theta} - \lambda\left(\delta - \tfrac{1}{2}\boldsymbol{d\theta}^\intercal F(\boldsymbol{\theta}_k)\boldsymbol{d\theta}\right), \quad \lambda \in \mathbb{R}.$$

Thus, any solution $\boldsymbol{d\theta}$ must satisfy the KKT conditions: for some $\lambda \in \mathbb{R}$,

$$\begin{aligned} \nabla_{\boldsymbol{d\theta}}\mathcal{L}(\boldsymbol{d\theta}, \lambda) &= 0, \\ \left(\delta - \tfrac{1}{2}\boldsymbol{d\theta}^\intercal F(\boldsymbol{\theta}_k)\boldsymbol{d\theta}\right) &\geq 0, \\ \lambda\left(\delta - \tfrac{1}{2}\boldsymbol{d\theta}^\intercal F(\boldsymbol{\theta}_k)\boldsymbol{d\theta}\right) &= 0, \\ \lambda &\geq 0. \end{aligned} \tag{4.12}$$

Assuming $\boldsymbol{\theta}_k$ is not a stationary point of J and $F(\boldsymbol{\theta}_k)$ is positive definite, then $\boldsymbol{d\theta}$ must be of the form

$$-\lambda \boldsymbol{d\theta} = F(\boldsymbol{\theta}_k)^{-1}\boldsymbol{g} \neq 0 \Longrightarrow \delta = \tfrac{1}{2}\boldsymbol{d\theta}^\intercal F(\boldsymbol{\theta}_k)\boldsymbol{d\theta}.$$

Therefore the unique solution satisfying these constraints is given by $\boldsymbol{d\theta} = \beta \boldsymbol{s}$, where $\boldsymbol{s} = F(\boldsymbol{\theta}_k)^{-1}\boldsymbol{g}$ and

$$\begin{aligned} \delta = \tfrac{1}{2}\beta \boldsymbol{s}^\intercal F(\boldsymbol{\theta}_k)\beta \boldsymbol{s} = \beta^2 \tfrac{1}{2}\boldsymbol{s}^\intercal F(\boldsymbol{\theta}_k)\boldsymbol{s} \\ \Longleftrightarrow \beta = \sqrt{\frac{2\delta}{\boldsymbol{s}^\intercal F(\boldsymbol{\theta}_k)\boldsymbol{s}}}. \end{aligned}$$

Thus, the update in policy parameters at iteration k is given by

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \left(\sqrt{\frac{2\delta}{\tilde{\boldsymbol{g}}^\intercal F(\boldsymbol{\theta}_k)\tilde{\boldsymbol{g}}}}\right)\tilde{\boldsymbol{g}}, \tag{4.13}$$

where $\tilde{\boldsymbol{g}}$ is the natural gradient of $J(\boldsymbol{\theta}_k)$ as given by Equation (3.11). Thus, as alluded to in the beginning of this chapter, TRPO provides a principled way of choosing the step size along the natural policy gradient direction. This also means that it benefits from all the advantages of using this search direction, discussed in the previous chapter. It is important to note, however, that the policy optimization method of TRPO is not an instance of stochastic optimization method, but an approximation to the non-linear optimization problem discussed in Section 4.1.

4.3 Practical Algorithm

In real applications, one does not have access to the advantage values $A^{\boldsymbol{\theta}_k}(s, a)$ for each state and action or the objective function J, therefore these quantities must be estimated (again, we overload notation by replacing $\pi_{\boldsymbol{\theta}}$ with $\boldsymbol{\theta}$ to reduce clutter). Since the term $J(\boldsymbol{\theta}_k)$ in the definition

of $L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta})$ is constant, one only needs to estimate the second term:

$$\begin{aligned}\sum_{s\in\mathcal{S}} \rho_{\boldsymbol{\theta}_k}(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a \mid s) A^{\boldsymbol{\theta}_k}(s,a) &= \mathbb{E}_{s_{0:\infty}} \left[\sum_{t=0}^{\infty} \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a \mid s_t) A^{\boldsymbol{\theta}_k}(s_t,a) \right] \\ &= \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}} \left[\sum_{t=0}^{\infty} \frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_k}(a_t \mid s_t)} A^{\boldsymbol{\theta}_k}(s_t,a_t) \right] \\ &\approx \frac{1}{N} \sum_{n=1}^{N} \frac{\pi_{\boldsymbol{\theta}}(a_n \mid s_n)}{\pi_{\boldsymbol{\theta}_k}(a_n \mid s_n)} A^{\boldsymbol{\theta}_k}(s_n,a_n),\end{aligned}$$

where the expectations are with respect to trajectories sampled by following $\pi_{\boldsymbol{\theta}_k}$ and n indexes over all timesteps in a batch of episodes collected by following $\pi_{\boldsymbol{\theta}_k}$. Section 3.5 already discusses a method of estimating the Fisher information matrix, which is adopted here as well. Finally, the advantages can be estimated by any advantage estimation algorithm, e.g., Generalized Advantage Estimation.

TRPO also uses a line search in order to compute the final step in the parameters. This is done for two reasons: (a) because of estimation errors, the step computed by (4.13) might violate the KL divergence constraint; (b) The actual improvement in the surrogate objective $L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta})$ might be too small, either because the policy is near optimal or due to estimation errors in the search direction. Algorithm 4 implements a backtracking line search in order to address theses concerns. Note that the algorithm computes exponentially decaying step sizes along the direction computed by TRPO, with an upper bound U on the number of iterations. In line 3, M is a sufficiently large constant so that whenever the constraint is violated the negative term dominates the rest.

Algorithm 5 outlines the Trust Region Policy Optimization approach discussed so far, which may be combined with Generalized Advantage Estimation .

Algorithm 4 Line Search for TRPO

Input: Initial step $\boldsymbol{\Delta\theta}$, backtrack ratio α, accept ratio ϵ, max. backtracks U

1: **for** $i = 0, 1, 2, \ldots, U$ **do**
2: $\quad\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_k + \alpha^i \boldsymbol{\Delta\theta}$
3: $\quad$Compute $f(\boldsymbol{\theta}) = L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) - M \max\{\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k \parallel \boldsymbol{\theta}) - \delta, 0\}$ using sample estimates
4: $\quad$**if** $f(\boldsymbol{\theta}) - L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}_k) \geq \epsilon \boldsymbol{g}^{\intercal}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)$ **then**
5: $\quad\quad$**return** $\alpha^i \boldsymbol{\Delta\theta}$
6: $\quad$**end if**
7: **end for**
8: **return 0**

4.4 Experiments I

The experiments in this Section had in mind the following objectives:

- Briefly compare the performance of policies obtained with TRPO with those obtained by the previous methods in the classical control environments used so far.
- Test this policy optimization approach on larger problems and analyze its effectiveness.

Algorithm 5 TRPO

Input: Parameterized policy $\pi_{\boldsymbol{\theta}}$, trust region constraint δ

1: Initialize policy parameter vector $\boldsymbol{\theta}_0 \in \mathbb{R}^d$
2: **for** $k = 1, 2, 3, \ldots$ **do**
3: Collect a set of trajectories following $\pi_{\boldsymbol{\theta}_k}$
4: Compute $\hat{A}_n$ for all timesteps using any advantage estimation algorithm
5: Estimate the natural policy gradient $\tilde{\boldsymbol{g}}$ using CG algorithm
6: Estimate the initial step $\boldsymbol{\Delta\theta} = \left(\sqrt{\frac{2\delta}{\tilde{\boldsymbol{g}}^\intercal F(\boldsymbol{\theta}_k)\tilde{\boldsymbol{g}}}}\right)\tilde{\boldsymbol{g}}$
7: Use Algorithm 4 to compute the final step $\tilde{\boldsymbol{\Delta}}\boldsymbol{\theta}$ and update

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \tilde{\boldsymbol{\Delta}}\boldsymbol{\theta}$$

8: **end for**

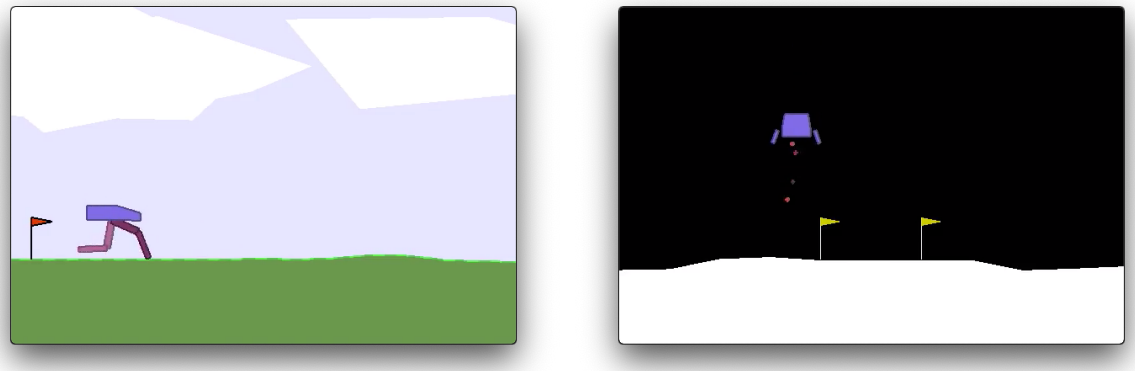


Figure 4.1: *Renderings of the* `BipedalWalker-v2` *and* `LunarLanderContinuous-v2` *environments respectively from left to right*

4.4.1 Task Details

Continuous Control in Box2D

The `Box2D` module from GYM offers continuous control problems using the physics simulator of the same name[2], an engine designed for 2-dimensional games. These are larger problems than the ones from the `classical control` module, specifically in the number of dimensions of state and action representations. The `BipedalWalker-v2` and `LunarLanderContinuous-v2` environment were selected among these in order to analyze the performance of the learning process using TRPO with generalized advantage estimation.

In `BipedalWalker-v2`, the agent controls a simple 2-legged robot by adjusting the motor torque in each of the 4 joints. Reward is given by moving forward, reaching a total +300 up to the far end, and -100 reward is given if the robot falls to the ground. A small penalty is incurred depending on the amount of torque used. The state feature vector consists of the robot's hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 Lidar rangefinder measurements, important for the `BipedalWalkerHardcore-v2` environment, a version with added obstacles in the robot's path. There are no coordinates in the state vector, totaling 24 dimensions. The documentation claims the Lidar observations are less useful in this simpler version. This could be a problem because of the added state dimensionality, which incurs a heavier computational cost and possibly more variance. We couldn't, however, run

[2] http://box2d.org

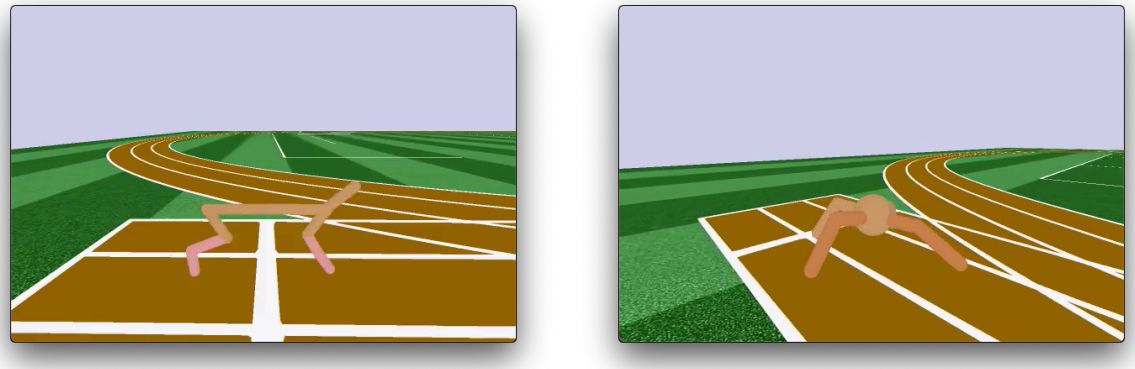


Figure 4.2: *Renderings of the* `RoboschoolHalfCheetah-v1` *and* `RoboschoolAnt-v1` *environments respectively from left to right*

trials without these features, seeing that each trial took about 45 minutes, making it difficult to test every configuration twice for this purpose.

In `LunarLanderContinuous-v2`, the agent controls a lander (spaceship) with 2 landing pads by controlling the thrust of the main and lateral engines. A landing pad is always present in coordinates $(0, 0)$ (center of the display), while the terrain surrounding it may vary between episodes. The lander starts at the top center of the display. A penalty is incurred by moving away from the landing pad, landing without crashing gives 100 reward, crashing gives -100 and each leg contact with the ground gives 10. The action is a 2-dimensional real vector with values in the range $[-1, 1]$ (higher values are clipped). The first coordinate controls the main engine, where negative values result in no throttle and values in $[0, 1]$ throttle from 50% to 100% of the engine's power. The second coordinate controls the lateral engines, with the only options being *on* and *off*, where values in $[-1, -0.5]$ fire the left engine, $[0.5, 1]$ fire right engine and $[-0.5, 0.5]$ turn both off. There is no fuel limit. The state feature vector consists of the lander's position, velocity, angle, angular velocity and two binary values indicating whether each landing pad is in contact with the ground or not, totaling 8 state features.

Figure 4.1 shows renderings of the bipedal walker and lunar lander tasks. The timestep limits for the episodes of each task are 1600 for the former and 1000 for the latter. The tasks are considered solved when a reward of $+300$ and $+200$, respectively, is achieved.

Simulated Robot Control

We also considered simulated robotics environments implemented in the open-source *Roboschool*[3] module from GYM. The `RoboschoolHalfCheetah-v1` and `RoboschoolAnt-v1` environments were selected to see if it's possible to obtain good results using TRPO generalized advantage estimation.

In both environments the objective is to make the robot run as fast as possible without falling. `RoboschoolHalfCheetah-v1` is considered the easier of the two because the agent controls a cheetah-like robot restricted to 2 dimensions, hence the name *halfcheetah*, and has real-valued state and action vectors with 25 and 5 dimensions respectively. The timestep limit for each episode is 1000 and the task is considered solved when a reward of 3000 is achieved over 100 consecutive episodes. On the other hand, `RoboschoolAnt-v1` has a fully 3-dimensional robot with four legs and real-valued state and action vectors with 27 and 7 dimensions respectively. The timestep limit for each episode is 1000 and the task is considered solved once a reward of 2500 is achieved in 100 consecutive episodes.

Figure 4.2 shows renderings of the two *roboschool* environments. Not much is known in terms of what each state and action feature mean in each task, other than joint torques, due to a lack

[3]A brief description of *roboschool* can be found here: https://blog.openai.com/roboschool/

Line search configurations	
accept ratio	0.1
backtrack ratio	0.8
max. backtracks	15

Table 4.1: *Configurations for Algorithm 4*

	MountainCar	CartPole	Bipedal	LunarLander	Cheetah	Ant
Iterations	100	100	1000	500	5000	5000
Batch size	20	2000	10000	10000	10000	10000
Num. Envs.	10	8	16	16	16	16
δ constraint	0.001	0.001	0.001	0.001	0.001	0.001

Table 4.2: *Configurations for each environment. A more conservative* KL *constraint was chosen than the one used in the original* TRPO *paper.*

of documentation. Although the ant robot is similar to the 3D quadruped used by Schulman *et al.* (2015b), the *roboschool* version used here is a re-tuned, heavier variant that encourages the robot to keep 2 or more legs on the ground, therefore it is unclear whether or not the reward functions described in the paper apply here. Nevertheless, these environments provide a good testing ground to show what's possible to do using Trust Region Policy Optimization.

4.4.2 Experimental Setup

The mountain car task used the same configurations as the experiments in sections 2.3 and 4.4, with linear policy architectures. Moreover, the policies used the same initial seeds for each trial, therefore all methods started with equal policies in corresponding trials. Similarly, the same configurations for the experiments with the Adam algorithm in Section 2.5 were used for the cartpole balancing task, with the exception that the policy architecture used with TRPO had one hidden layer of 32 units with ELU activations. This was done in order to test whether the more advanced policy optimization procedure could do better than the Adam algorithm with a simpler network.

The same policy architecture was used for both Box2D tasks, consisting of three hidden layers with 100, 50 and 25 units respectively and ELU activations. Also for both tasks, 7 trials were collected for each combination of generalized advantage estimation parameters.

For the simulated robotics tasks, several policy architectures were considered, as these choices produced substancially different results. These tasks took considerably more time to run, therefore only a few trials were possible.

Specific hyperparameters for each task are given in Table 4.2. The configurations used for the line search part of TRPO are given in Table 4.1 and were used across all tasks.

4.4.3 Experiment Results

Classical control tasks

Trust Region Policy Optimization proved to be an outstanding method for the simple, classical control problems considered in the last two chapters. Figure 4.3 (left) shows how this method manages to beat the vanilla policy gradient method with the Adam optimizer across several combinations of generalized advantage estimation parameters. Moreover, the difference in the learning curves for TRPO was so little that the standard error is almost imperceptible in the plot. This can be a bit misleading, as the standard error divides the sample standard deviation by the square root of the sample size, therefore the difference between the results of different trials is a bit larger than

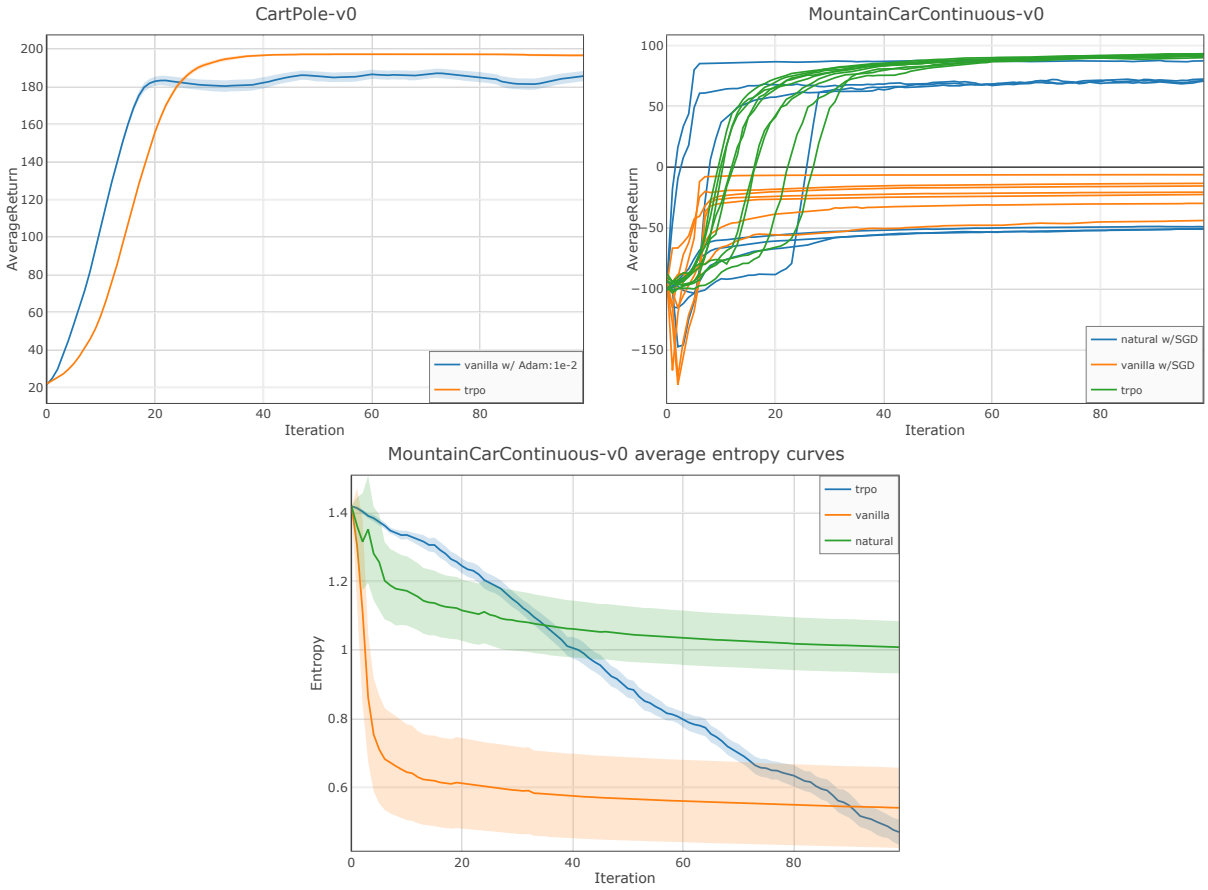


Figure 4.3: *Top left: comparison of algorithms 2 and 5 in the cartpole task over multiple trials, 7 for each combination of γ and λ at the values $0.97, 0.98, 0.99, 1$. Results show the mean and standard error over all combinations for each algorithm. Top right: average returns for individual trials of the three algorithms in the mountain car task, all using the same policy architecture, initial seeds and number of episodes per iteration.* GAE *was not used in this task. Bottom: means and standard errors of the average entropy per iteration for different algorithm, taken from the same experiment on the top right plot.*

the image suggests. Nevertheless, this showcases how robust the algorithm is to different hyperparameters.

In the `MountainCar` problem, the difference in relation to the previous methods is even more pronounced, as shown in Figure 4.3 (right). Surprisingly, even though all methods started with the same policies for each trial, TRPO managed to achieve positive average return across all trials, even those with initial policies that didn't reach the flag in the first batch of episodes. By analyzing the average entropy over time obtained with the different algorithms, it is evident that TRPO was more conservative in its updates, therefore the policies remained exploratory for a greater amount of time. This is probably what allowed the car to occasionally reach the flag on the hill and therefore replicate the behaviour. It is also worth noting that all but one of the trials managed to solve the problem, obtaining an average return of +90.

Box2D tasks

These tasks took considerably more time to run, with each trial of the bipedal walker task taking about 45 minutes on a dual core laptop and each of the lunar lander ones taking about 50 minutes on an eight-core machine, albeit with a lower clock-speed (precise clock speeds were not available). We searched for the best combinations of GAE parameters among the values of $\gamma = 0.98, 0.99$ and $\lambda = 0.97, 0.98, 0.99$. Figure 4.4 shows the average results for each of the tasks and parameter

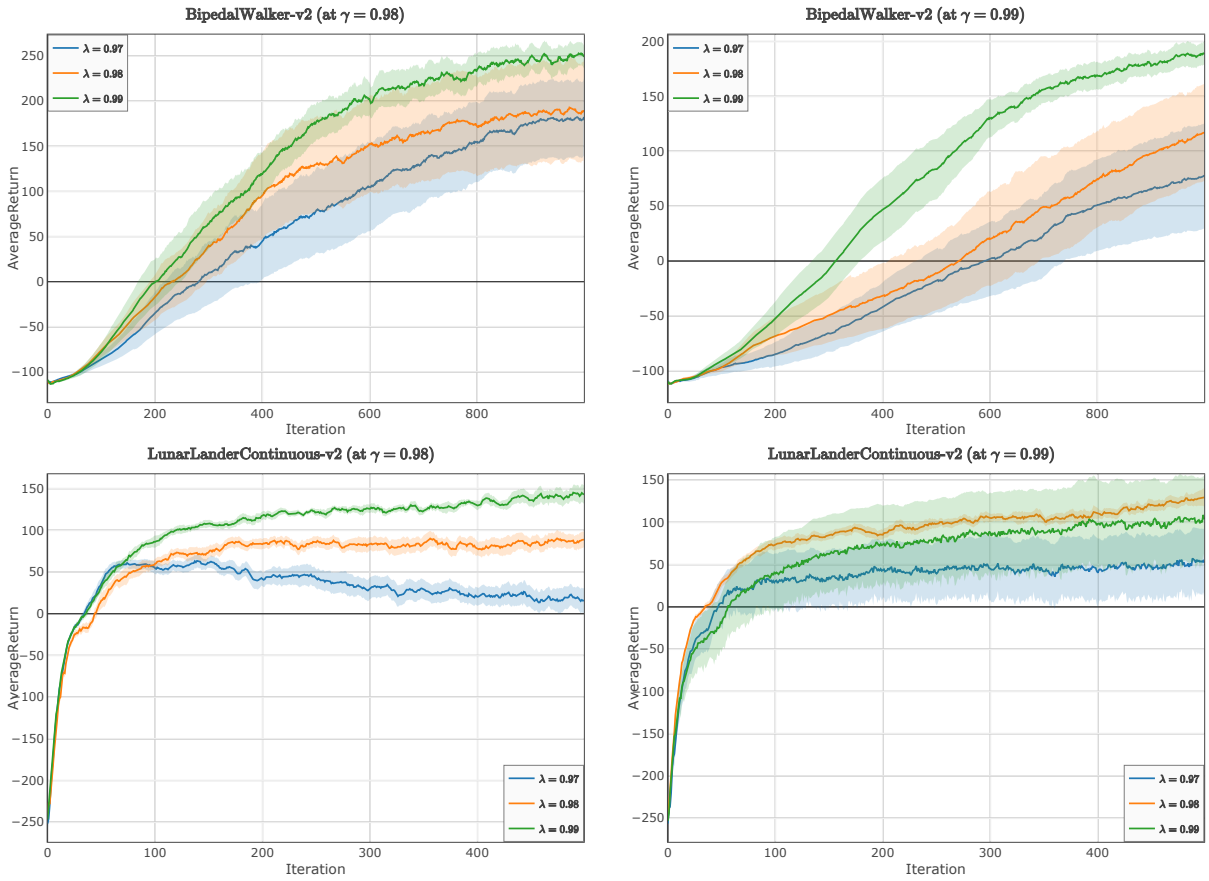


Figure 4.4: *Learning curves using* TRPO *and* GAE *in the bipedal walker and lunar lander tasks.*

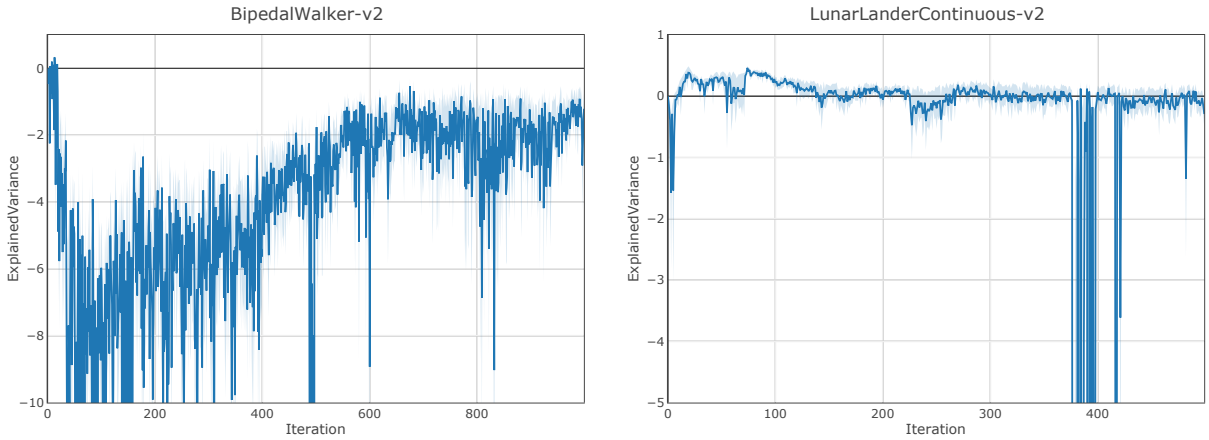


Figure 4.5: *Average explained variances of the baselines in the bipedal walker (left) and lunar lander tasks (right), taken from the same experiments of Figure 4.4*

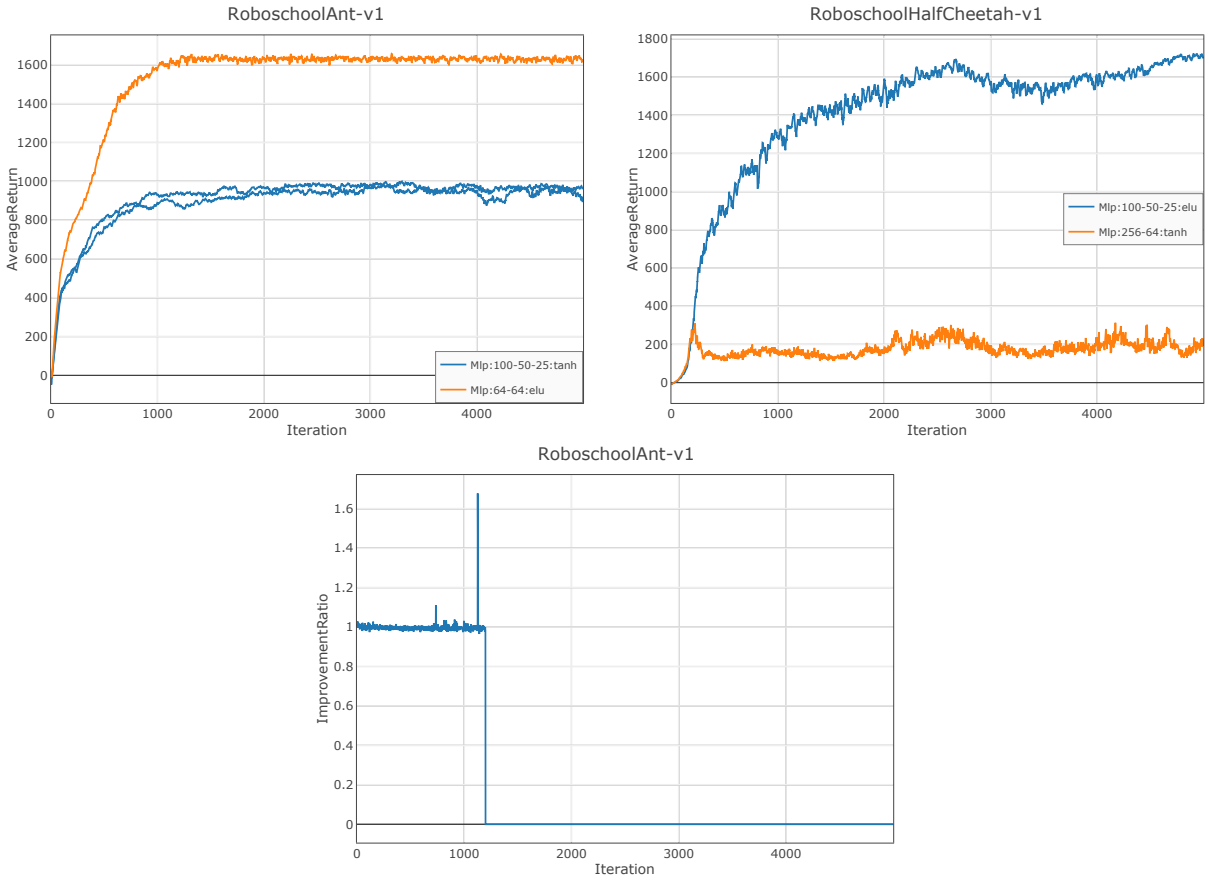


Figure 4.6: *Top left and right: average returns over time for the half cheetah and ant robots, from left to right, respectively. Bottom: ratio between the actual improvement in the surrogate objective and the one predicted by the linear approximation (line 4 of Algorithm 4)*

combinations. It is interesting to see that the best results were achieved, in general, with lower values of γ and higher ones of λ, contrary to what was observed by Schulman *et al.* (2015b) in their paper introducing generalized advantage estimation. Figure 4.5 shows evidence of what might have caused this issue.

Analyzing the average explained variance of the baselines for the bipedal walker experiments, it was found that the ones used were not a good fit to the data, either due to their architecture or to the optimization method used to fit the returns. This could explain why higher values of λ achieved better results, since the degree of bootstrapping is lower and thus the influence of the value function approximator on the results. The lower values of γ may have helped compensating this issue by reducing the variance of the discounted advantage estimates. Unfortunately, only one trial managed to solve the environment, with hyperparameters $\gamma = 0.98$ and $\lambda = 0.97$.

The baselines in the lunar lander task showed the same issue, albeit less severe than in the bipedal walker task, as far as the explained variance can tell. In this setting, the contrast in the standard error of the results between the experiments with different values of γ is evidence of its role in reducing the variance of the gradient estimates. Unfortunately, none of the trials achieved an average return enough to solve the environment. The best performing policy used $\gamma = 0.98$ and $\lambda = 0.99$ and achieved an average return of 184. Many of the best performing policies managed to land the spaceship safely, although at times not on the landing pad.

Simulated robotics tasks

These experiments were the most time consuming ones by far, with each trial of the ant robot taking about 5 hours and the half cheetah one taking 4.5 and 3 hours for the two trials. The

difference in time of the half cheetah trials is due to the robot falling to the ground a lot, which in turn causes the episode to reset. It turns out that the more resets occur during the sampling part of the algorithm, the longer it takes to run. This is due to resets being more costly, as the environment has to reset and store information about the episode such as its length and total reward. This is not as pronounced in the previous tasks as those are computationally cheaper.

Figure 4.6 (top) shows the average returns over time obtained in both environments. Both took two attempts to achieve reasonable results, albeit far from the requirements for solving the environments. Looking at the average return curves for the half cheetah tasks, it's unclear whether or not running the experiment with the best result for additional iterations would have yielded better results. It is also mysterious why the change in policy architecture yielded a substantially better result. This highlights some of the strengths and weaknesses of deep learning in general: impressive results can be achieved in practice, but the computational cost and amount of hyperparameters to adjust is large and difficult to manage. Nevertheless, the best performing policy actually learned a running animation stable enough that the robot would not fall.

For the ant-like robot, the first attempts at solving the environment used the same architecture proposed by Schulman *et al.* (2015b) in their experiments with the 3D-quadruped. However, here we opted for a smaller batch size and larger number of iterations (see Table 4.2). Again, a change in policy architecture yielded substantially better results. Even with different initial policies, the two trials with the aforementioned policy architecture yielded strikingly similar results. Switching to a shallower architecture (smaller number of hidden layers) produced better results, quickly converging to a plateau from which no noticeable improvement in the average return was obtained. Analyzing why this might have happened, it was found that the line search returned a step size of zero after iteration ~ 1200. Figure 4.6 (bottom) shows the improvement ratio between the surrogate objective and the linear approximation of the improvement:

$$\frac{L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) - L_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}_k)}{\boldsymbol{g}^\intercal(\boldsymbol{\theta} - \boldsymbol{\theta}_k)},$$

where 0 is recorded whenever all the iterations of the line search fail, i.e., the improvement in the surrogate objective isn't large enough in relation to the one predicted by the linear approximation. Therefore, the algorithm achieved what is probably a local maximum, where no direction provided substantial improvement. These are all approximations and speculations, of course, as the functions and variables computed by the algorithm are sample approximation to the true ones. Nevertheless, the best policy made the robot walk steadily, alternating between pulling itself with one and two legs.

4.5 Experiments II

4.5.1 Motivation and Settings

	LunarLander	Ant
Iterations	500	1000
Batch size	10000	5000/10000
Num. Envs.	16	16
δ constraint	0.01	0.01

Table 4.3: *Configurations for each environment. A more liberal* KL *constraint was chosen than the one used in Table 4.2.*

A few potential problems were noticed after the experiments of the previous section: (a) the KL constraint used might have been too small to allow progress in a reasonable number of iterations; (b) the architectures used might have been too large for the problems considered.

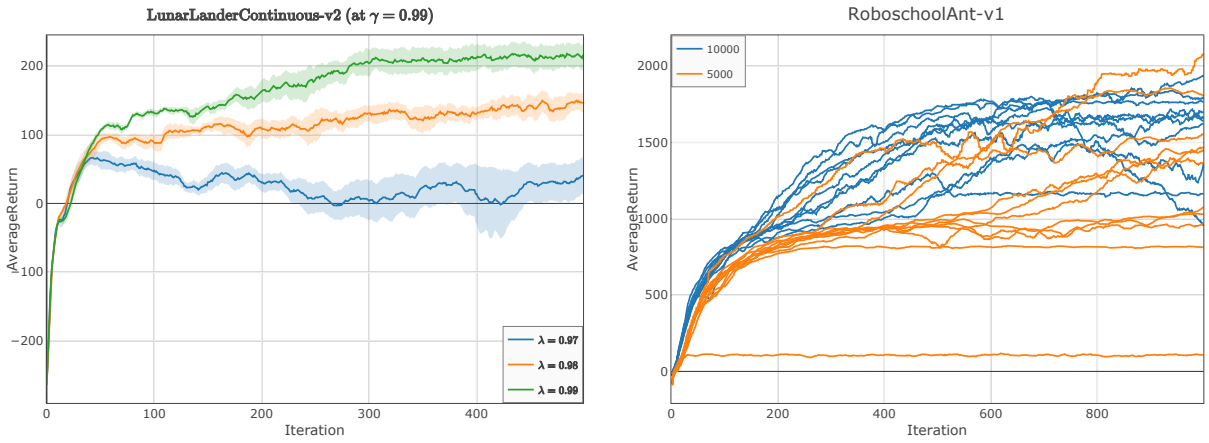


Figure 4.7: *Left: means and standard errors over 7 trials of the average returns obtained in the lunar lander task. Right: average returns for individual trials of the ant robot task, labeled by the batch size used. All trials used* GAE *values of* $\gamma = 0.99$ *and* $\lambda = 0.98$ *and the same initial seed.*

Thus, we ran the additional experiments for the lunar lander and ant robot tasks using the hyperparameters from Table 4.3. The architectures used were one with a single hidden layer of 32 units and one with two hidden layers of 32 units each for the lunar lander and ant robot tasks respectively. Both configurations used ELU activations.

4.5.2 Experiment Results

As it turned out, using a more liberal KL constraint in the lunar lander task further separated the best trials from the worst ones. Figure 4.7 (left) illustrates this, when contrasted with the corresponding plot in Figure 4.4. This was to be expected, since a larger trust region implies more reliance on the quality of the approximations made by Algorithm 5, therefore bad steps have a larger impact. However, it's important to note that the experiments in this Section used a simpler network, which might have played a role in the differences between experiments. Most notably, the larger step sizes allowed the best trials to quickly achieve an average reward of $+200$, successfully solving the environment.

Using a simpler policy architecture and larger trust region constraint also produced better results in the ant-like walker environment. However, an important issue was noted in these experiments: running the same configuration multiple times produced different results. This includes setting the seeds for the random number generators used by the policy and environment. Figure 4.7 (right) shows the results for several runs of the same configurations, which only differ by batch size used per iteration. This might be an issue with the Roboschool environments, since all the other experiments were repeatable under the same configurations.

Nevertheless, a lot of the results obtained using these configurations achieved average returns far above the observed in the previous Section under a smaller number of iterations. Interestingly, the only trial which managed to achieve an average return of $+2000$ used a smaller batch size of 5000 samples. This is probably an outlier, as the trials with smaller batch sizes produced worse policies in general. Nevertheless, the best policy produced a steady gait in which the robot used all four legs to move itself.

Chapter 5

Conclusions

Policy gradient methods offer solutions for reward-related learning problems which enable integration of reinforcement learning theory with deep learning methods and models. Thus, these can be applied to more realistic tasks when compared with other reinforcement learning solutions. In this work, we studied and analyzed three policy gradient methods:

- Vanilla Policy Gradient, which allows one to reduce the reinforcement learning problem to that of stochastic optimization by providing an expression for the gradient of the performance function that can be sampled from experiences;
- Natural Policy Gradient, which explores the underlying structure of the parameter space, defining a new gradient direction that takes into account divergences between trajectory distributions induced by policies and is invariant to reparameterization;
- Trust Region Policy Optimization (TRPO), which takes a different approach from the preceding ones by making approximations to a monotonically improving policy optimization procedure, improving upon the properties of the previous methods.

Although the first two were introduced in the late 90's and have fallen out of favor in the modern deep learning era; studying these methods and the underlying theory behind them provides the fundamentals of the policy gradient approach. Moreover, these ideas are still present in modern techniques such as TRPO, ACKTR and A2C. The third algorithm, TRPO, serves as an entry point to the more advanced methods that followed it in recent years, bridging the gap between them and the fundamental theory behind policy gradient methods.

The Vanilla and Natural Policy Gradient methods were not suitable to the more complicated tasks included in the `Box2D` and `Roboschool` modules from OpenAI Gym, i.e., there was no stable improvement in the average return per iteration obtained using these methods. Therefore the comparison between the three methods was constrained to the simple environments `CartPole-v0` and `MountainCarContinuous-v0`. The Vanilla method outperformed the Natural in the cartpole task, achieving higher average returns over multiple trials using the same number of iterations, batch size, and Generalized Advantage Estimation parameters. This difference can be explained by the sensitivity of the Natural method to the learning rates used to update the policy parameters, i.e., small differences in learning rate may cause large divergences between successive policies. Since learning rates have to change over the course of stochastic gradient ascent algorithms, the Natural method is too unstable to be straightforwardly applied. On the other hand, this method outperformed the Vanilla in the mountain car task, obtaining positive average returns while the latter was never able to do the same. This task showed that the Natural method can find good policies even if rewards and the states in which they occur are rare in the beginning when the policies are more exploratory. Trust Region Policy Optimization outperformed both previous methods in both tasks while also showing itself to be more robust to different hyperparameter choices, generally yielding steady improvement in average return per iteration. Moreover, it was easier to achieve the average return requirements for solving each task using this algorithm.

TRPO was also effective in more complicated tasks where state and action spaces have a larger number of dimensions than in the simpler classical control environments mentioned above. In the tasks provided by the `Box2D` (`LunarLanderContinuous-v2` and `BipedalWalker-v2`) and `Roboschool` (`RoboschoolHalfCheetah-v1` and `RoboschoolAnt-v1`) GYM modules, stable policy improvement (w.r.t. average returns) was observed for many hyperparameter combinations. Its trust region approach helps to prevent large drops in performance, observed occasionally in previous methods, since changes in the trajectory distribution are kept to a reasonable neighborhood. However, approximately maximizing the average return is a laborious task, as there are no clear signs that indicate whether or not the maximum return has been achieved for a particular policy architecture and hyperparameter combination. Our experiments showed that, in general, complicated architectures were not necessary for solving the `LunarLanderContinuous-v2` and `RoboschoolAnt-v1` environments. Unfortunately, we couldn't test this hypothesis in the other tasks as these experiments take hours to run each.

Other considerations involve the cost of implementing and running these algorithms in practice. The Vanilla method is the cheapest to implement and run in practice, meaning it requires the least amount of code and computation (only one gradient evaluation by backpropagation) per batch of data collected. Both the Natural method and TRPO require approximating the average Hessian of KL divergence of the policy; a process that requires multiple gradient evaluations for a single policy update. Compared to the Natural method, TRPO only adds the cost of computing the stepsize via a line search. Given the instability of the former and the impressive results of the latter, TRPO asserts itself as the superior method. One might consider using the Vanilla method over it in smaller problems given its simplicity and lower computational costs. However, TRPO is substantially more efficient in high-dimensional problems.

Appendix A

Proofs

A.1 Proof of Theorem 1

To keep the notation simple, we leave it implicit that π is a function of $\boldsymbol{\theta}$, and all gradients are implicitly with respect to $\boldsymbol{\theta}$. First, note that the derivative of the state value function is given by

$$\begin{aligned}
\nabla V^\pi(s) &= \nabla \sum_{a\in\mathcal{A}} \pi(a|s) Q^\pi(s,a) && \text{(eq. 1.8)} \\
&= \sum_{a\in\mathcal{A}} \left[Q^\pi(s,a) \nabla\pi(a|s) + \pi(a|s) \nabla Q^\pi(s,a) \right] \\
&= \sum_{a\in\mathcal{A}} \left[Q^\pi(s,a) \nabla\pi(a|s) + \pi(a|s) \nabla \left(\mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}(s' \mid s,a) V^\pi(s') \right) \right] && \text{(eq. 1.9)} \\
&= \sum_{a\in\mathcal{A}} \left[Q^\pi(s,a) \nabla\pi(a|s) + \gamma\pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}(s' \mid s,a) \nabla V^\pi(s') \right]. && (\star)
\end{aligned}$$

Thus, the gradient of $J(\boldsymbol{\theta})$ can be written as

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla \mathbb{E}_{\substack{s_{0:\infty} \\ a_{0:\infty}}} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \right] \\
&= \nabla \sum_{s\in\mathcal{S}} \rho_0(s) \mathbb{E}_{\substack{s_{1:\infty} \\ a_{0:\infty}}} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \middle| s_0 = s \right] \\
&= \sum_{s\in\mathcal{S}} \rho_0(s) \nabla V^\pi(s) && \text{(eq. 1.6)} \\
&= \sum_{s\in\mathcal{S}} \rho_0(s) \sum_{a\in\mathcal{A}} \left[Q^\pi(s,a) \nabla\pi(a|s) + \gamma\pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}(s' \mid s,a) \nabla V^\pi(s') \right] \\
&\;\;\vdots \\
&= \sum_{s\in\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s) \sum_{a\in\mathcal{A}} Q^\pi(s,a) \nabla\pi(a|s)
\end{aligned}$$

after repeated unrolling of ($\star$). Using definition 3, we obtain

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \sum_{s\in\mathcal{S}} \rho_\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a) \nabla\pi(a|s) \\
&= \sum_{s'\in\mathcal{S}} \rho_\pi(s') \sum_{s\in\mathcal{S}} \frac{\rho_\pi(s)}{\sum_{s'\in\mathcal{S}} \rho_\pi(s')} \sum_{a\in\mathcal{A}} Q^\pi(s,a) \nabla\pi(a|s)
\end{aligned}$$

$$= \frac{1}{1-\gamma} \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s,a) \nabla \pi(a|s),$$

for $\gamma \in (0,1)$ and

$$\nabla J(\boldsymbol{\theta}) = T \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s,a) \nabla \pi(a|s),$$

for the episodic undiscounted case, since returns after reaching a terminal state are zero.

A.2 Proof of Proposition 2

Note that for any function $f(s_t, a_t)$ where $0 \leq t < n$:

$$\begin{aligned}
\sum_{\tau \in \mathcal{T}^n} Pr(\tau \,|\, \pi) f(s_t, a_t) &= \sum_{s_0 \in \mathcal{S}} \sum_{a_0 \in \mathcal{A}} \sum_{s_1 \in \mathcal{S}} \sum_{a_1 \in \mathcal{A}} \cdots \sum_{a_{n-1} \in \mathcal{A}} \sum_{s_n \in \mathcal{S}} \rho_0(s_0) \prod_{i=0}^{n-1} \mathcal{P}(s_{i+1} \,|\, s_i, a_i) \pi(a_i \,|\, s_i) f(s_t, a_t) \\
&= \sum_{s_0 \in \mathcal{S}} \cdots \sum_{a_t \in \mathcal{A}} \sum_{s_{t+1} \in \mathcal{S}} \rho_0(s_0) \prod_{i=0}^{t} \mathcal{P}(s_{i+1} \,|\, s_i, a_i) \pi(a_i \,|\, s_i) f(s_t, a_t) \\
&= \sum_{\tau \in \mathcal{T}^{t+1}} Pr(\tau \,|\, \pi) f(s_t, a_t),
\end{aligned}$$

since $f(s_t, a_t)$ doesn't depend on later states and actions and, given s_{t+1},

$$\sum_{a_{t+1} \in \mathcal{A}} \cdots \sum_{s_n \in \mathcal{S}} \prod_{j=t+1}^{n-1} \mathcal{P}(s_{j+1} \,|\, s_j, a_j) \pi(a_j \,|\, s_j) = 1.$$

Expanding upon the expected return definition, note that we can add a final state to the expectation without changing its value, therefore:

$$\begin{aligned}
\mathbb{E}_{\substack{s_{0:n} \\ a_{0:n-1}}} \left[\sum_{t=0}^{n-1} \gamma^t \mathcal{R}(s_t, a_t) \right] &= \sum_{s_0 \in \mathcal{S}} \sum_{a_0 \in \mathcal{A}} \sum_{s_1 \in \mathcal{S}} \sum_{a_1 \in \mathcal{A}} \cdots \sum_{a_{n-1} \in \mathcal{A}} \sum_{s_n \in \mathcal{S}} \rho_0(s_0) \prod_{i=0}^{n-1} \mathcal{P}(s_{i+1} \,|\, s_i, a_i) \pi(a_i \,|\, s_i) \cdots \\
&\qquad (\mathcal{R}(s_0, a_0) + \gamma \mathcal{R}(s_1, a_1) + \cdots + \gamma^n \mathcal{R}(s_{n-1}, a_{n-1})) \\
&= \sum_{t=0}^{n-1} \sum_{\tau \in \mathcal{T}^{t+1}} Pr(\tau \,|\, \pi) \gamma^t \mathcal{R}(s_t, a_t) \\
&= \frac{1-\gamma^n}{1-\gamma} \sum_{\tau \in \mathcal{T}_+^n} Pr^{\gamma}(\tau \,|\, \pi) \mathcal{R}(s_{|\tau|-1}, a_{|\tau|-1}),
\end{aligned}$$

and thus

$$\lim_{n \to \infty} \mathbb{E}_{\substack{s_{0:n} \\ a_{0:n-1}}} \left[\sum_{t=0}^{n-1} \gamma^t \mathcal{R}(s_t, a_t) \right] = \frac{1}{1-\gamma} \mathbb{E}_{\tau \in \mathcal{T}_+^n} \left[\mathcal{R}(s_{|\tau|-1}, a_{|\tau|-1}) \right].$$

A.3 Proof of Theorem 3

For simplicity, we leave it implicit that π is a function of $\boldsymbol{\theta}$ and that all derivations are with respect to $\boldsymbol{\theta}$. The fisher information metric on the manifold of probability distributions over trajectories of length n is defined as

$$F^n(\boldsymbol{\theta}) = \mathbb{E}_{\tau \in \mathcal{T}^n} \left[\nabla \log Pr(\tau_{0:n} \,|\, \pi) \nabla \log Pr(\tau \,|\, \pi)^{\intercal} \right]$$

$$
\begin{aligned}
&= -\mathbb{E}_{\tau\in\mathcal{T}^n}\left[\nabla^2 \log Pr(\tau\,|\,\pi)\right] \\
&= -\sum_{\tau\in\mathcal{T}^n} Pr(\tau\,|\,\pi)\nabla\left(\nabla \log Pr(\tau\,|\,\pi)\right)^\intercal \\
&= -\sum_{\tau\in\mathcal{T}^n} Pr(\tau\,|\,\pi)\nabla\left(\sum_{t=0}^{n-1}\nabla \log \pi(a_t\,|\,s_t)\right)^\intercal \\
&= -\sum_{\tau\in\mathcal{T}^n} Pr(\tau\,|\,\pi)\sum_{t=0}^{n-1}\nabla^2 \log \pi(a_t\,|\,s_t).
\end{aligned}
$$

Where the second line follows from the first by equation 3.7 and the fourth follows by substituting definition 4 for $\Pr(\tau\,|\,\pi)$ and noting that $\mathcal{P}(\cdot\,|\,s_t, a_t)$ does not depend on $\boldsymbol{\theta}$. We can then rewrite the sum over all trajectories probabilities as

$$
\underbrace{\sum_{x_n\in\mathcal{S}}\sum_{a_{n-1}\in\mathcal{A}}\cdots\underbrace{\sum_{x_1\in\mathcal{S}}\underbrace{\sum_{a_0\in\mathcal{A}}\sum_{x_0\in\mathcal{S}}\underbrace{\rho_0(x_0)}_{\mathbb{P}(s_0=x_0)}\,\pi(a_0\,|\,x_0)\mathcal{P}(x_1\,|\,x_0,a_0)}_{\mathbb{P}(s_1=x_1)}\cdots\pi(a_{n-1}\,|\,x_{n-1})\mathcal{P}(x_n\,|\,x_{n-1},a_{n-1})}_{\mathbb{P}(s_n=x_n)}}_{},
$$

which, substituted for $\sum_{\tau\in\mathcal{T}^n}\Pr(\tau\,|\,\pi)$ above gives

$$
\begin{aligned}
F^n(\boldsymbol{\theta}) &= \sum_{t=0}^{n-1}\sum_{s\in\mathcal{S}}\mathbb{P}(s_t=s)\sum_{a\in\mathcal{A}}\pi(a\,|\,s)\sum_{s'\in\mathcal{S}}\mathbb{P}(s_n=s'\,|\,s_t=s,a_t=a)\left(-\nabla^2\log\pi(a\,|\,s)\right) \\
&= \sum_{t=0}^{n-1}\sum_{s\in\mathcal{S}}\mathbb{P}(s_t=s)\sum_{a\in\mathcal{A}}\pi(a\,|\,s)\left(-\nabla^2\log\pi(a\,|\,s)\right) \\
&= n\sum_{s\in\mathcal{S}}d^\pi(s)\sum_{a\in\mathcal{A}}\pi(a\,|\,s)\left(-\nabla^2\log\pi(a\,|\,s)\right).
\end{aligned}
$$

If all episodes terminate in n timesteps or less, this gives an appropriate metric on the tangent space. For the discounted infinite horizon formalism, consider the formulation of the expected return with respect to the discounted trajectory distribution given in proposition 2. We define the fisher information metric with respect to this distribution and take the limit as trajectory lengths tend to infinity:

$$
\begin{aligned}
F^{n,\gamma}(\boldsymbol{\theta}) &= \mathbb{E}_{\tau\in\mathcal{T}_+^n}\left[-\nabla^2\log Pr^\gamma(\tau\,|\,\pi)\right] \\
&= \frac{1-\gamma}{1-\gamma^n}\sum_{\tau\in\mathcal{T}_+^n}Pr^\gamma(\tau\,|\,\pi)\sum_{t=0}^{|\tau|-1}\left(\nabla^2\log\pi(a_t\,|\,s_t)\right) \\
&= \frac{1-\gamma}{1-\gamma^n}\sum_{k=1}^{n}\sum_{\tau\in\mathcal{T}^k}\gamma^{k-1}Pr(\tau\,|\,\pi)\sum_{t=0}^{k-1}\left(\nabla^2\log\pi(a_t\,|\,s_t)\right) \\
&= \frac{1-\gamma}{1-\gamma^n}\sum_{k=1}^{n}\gamma^{k-1}F^n(\boldsymbol{\theta}) \\
&= \frac{1-\gamma}{1-\gamma^n}\sum_{k=1}^{n}\gamma^{k-1}\sum_{t=0}^{k-1}\sum_{s\in\mathcal{S}}\mathbb{P}(s_t=s)\sum_{a\in\mathcal{A}}\pi(a\,|\,s)\left(-\nabla^2\log\pi(a\,|\,s)\right) \\
&= \sum_{s\in\mathcal{S}}\underbrace{\frac{1-\gamma}{1-\gamma^n}\sum_{k=1}^{n}\gamma^{k-1}\sum_{t=0}^{k-1}\mathbb{P}(s_t=s)}_{\star}\sum_{a\in\mathcal{A}}\pi(a\,|\,s)\left(-\nabla^2\log\pi(a\,|\,s)\right).
\end{aligned}
$$

As $n \to \infty$, the term $\star$ becomes

$$
\begin{aligned}
&(1-\gamma)\left(\mathbb{P}(s_0=s)+\gamma\left(\mathbb{P}(s_0=s)+\mathbb{P}(s_1=s)\right)+\gamma^2(\mathbb{P}(s_0=s)+\mathbb{P}(s_1=s)+\mathbb{P}(s_2=s))+\cdots\right)\\
&\quad=(1-\gamma)\Big(\mathbb{P}(s_0=s)(1+\gamma+\gamma^2+\cdots)+\mathbb{P}(s_1=s)(\gamma+\gamma^2+\gamma^3+\cdots)+\\
&\qquad\qquad\mathbb{P}(s_2=s)(\gamma^2+\gamma^3+\gamma^4+\cdots)+\cdots\Big)\\
&\quad=(1-\gamma)\left(\mathbb{P}(s_0=s)(\tfrac{1}{1-\gamma})+\mathbb{P}(s_1=s)(\tfrac{\gamma}{1-\gamma})+\mathbb{P}(s_2=s)(\tfrac{\gamma^2}{1-\gamma})+\cdots\right)\\
&\quad=\sum_{t=0}^{\infty}\gamma^t\mathbb{P}(s_t=s),
\end{aligned}
$$

and thus

$$
\begin{aligned}
\lim_{n\to\infty}F^{n,\gamma}(\boldsymbol{\theta})&=\sum_{s\in\mathcal{S}}\sum_{t=0}^{\infty}\gamma^t\mathbb{P}(s_t=s)\sum_{a\in\mathcal{A}}\pi(a\,|\,s)\left(-\nabla^2\log\pi(a\,|\,s)\right)\\
&=\frac{1}{1-\gamma}\sum_{s\in\mathcal{S}}d^{\pi}(s)\sum_{a\in\mathcal{A}}\pi(a\,|\,s)\left(-\nabla^2\log\pi(a\,|\,s)\right).
\end{aligned}
$$

Bibliography

Amari(1998) Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276. Cited in page 27

Amari & Nagaoka(2007) Shun-ichi Amari & Hiroshi Nagaoka. *Methods of information geometry*, volume 191. American Mathematical Soc. Cited in page 26, 28

Bagnell & Schneider(2003) J Andrew Bagnell & Jeff Schneider. Covariant policy search. In *IJCAI*, volume 18, pages 1019–1024. Citeseer. Cited in page 27, 31

Baird(1995) Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pages 30–37. Elsevier. Cited in page 6

Bertsekas & Tsitsiklis(2000) Dimitri P Bertsekas & John N Tsitsiklis. Gradient convergence in gradient methods with errors. *SIAM Journal on Optimization*, 10(3):627–642. Cited in page 8

Brockman *et al.***(2016)** Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang & Wojciech Zaremba. Openai gym, 2016. Cited in page 13

Kakade & Langford(2002) Sham Kakade & John Langford. Approximately optimal approximate reinforcement learning. In *ICML*, volume 2, pages 267–274. Cited in page 33

Kakade(2002) Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538. Cited in page 25, 27

Kingma & Ba(2014) Diederik P Kingma & Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. Cited in page 20

Mnih *et al.***(2015)** Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski *et al.* Human-level control through deep reinforcement learning. *Nature*, 518(7540):529. Cited in page 6

Paszke *et al.***(2017)** Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga & Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*. Cited in page 14

Schulman *et al.***(2015a)** John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan & Philipp Moritz. Trust region policy optimization. In *ICML*. Cited in page 31, 33, 34

Schulman *et al.***(2015b)** John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan & Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*. Cited in page 12, 17, 19, 21, 40, 43, 44

Sutton & Barto(2018) Richard S Sutton & Andrew G Barto. *Reinforcement learning: An introduction*. MIT press. Cited in page 2, 17, 19

Sutton *et al.***(2000)** Richard S Sutton, David A McAllester, Satinder P Singh & Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063. Cited in page 9

Szepesvári(2009) Csaba Szepesvári. Reinforcement learning algorithms for mdps. *Dept. of Computing Science Report TR09-13, University of Alberta, Ca.* Cited in page 4

Williams(1992) Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256. Cited in page 11, 12

Wright & Nocedal(1999) Stephen Wright & Jorge Nocedal. Numerical optimization. *Springer Science*, 35(67-68):7. Cited in page 27, 29