Greedy Actor-Critic: A New Conditional Cross-Entropy Method for Policy Improvement

by

Samuel Neumann

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Actor-Critics are a popular class of algorithms for control. Their ability to learn complex behaviours in continuous-action environments make them directly applicable to many real-world scenarios. These algorithms are composed of two parts – a critic and an actor. The critic learns to critique actions taken by the actor. The actor, a policy, uses this criticism to learn to attain higher rewards. Generally, this policy is improved by matching it to the Boltzmann distribution over action values. In this thesis, we introduce an alternative policy update, based on the cross-entropy method (CEM). This Conditional CEM (CCEM) applies the CEM to maximize an action-value critic conditioned on state. The algorithm works by initializing the actor policy as a wide distribution and iteratively concentrating on highly valued actions by using a maximum likelihood update toward the top percentile of an empirical action distribution. This empirical action distribution is generated by an additional, entropy regularized proposal policy that also concentrates on maximally valued actions, although more slowly. Under ideal conditions, the CCEM guarantees policy improvement and tracks the expected solution of the CEM across states. Finally, we introduce a new actor-critic algorithm called Greedy Actor-Critic that uses the CCEM for policy improvement. We empirically show that Greedy Actor-Critic can perform better than Soft Actor-Critic on a suite of classic control environments and is somewhat less sensitive to hyperparameters than SAC is on this environmental suite.

Preface

This thesis is an extension of a submission to the International Conference on Machine Learning in 2022. This submission was in collaboration with Sungsu Lim, Ajin Joseph, Yangchen Pan, Adam White, and Martha White. Lim, Joseph, and Pan performed preliminary work upon which this thesis is based and introduced the original implementation of the Conditional Cross-Entropy Optimization Algorithm, which is adapted and extended in this thesis. Adam White, Martha White, and Andrew Patterson helped with the analysis of Figures 3.1, 3.2, and 3.3. Both Adam and Martha White helped with the analysis of Figures 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 7.1, and 7.2. Martha White developed the policy improvement proof for the CCEM, outlined in Section 4.3. All other contributions are my own.

To my wife Abigail and my daughter Bryn, who have each been a strong support for me throughout my academic adventures.

Acknowledgements

This thesis could not have been completed without the academic guidance of others throughout my entire life. My entire academic career, from my very first day of school until today, has been influenced by so many individuals, space fails me to list each one here. Nevertheless, I am extremely grateful to all these individuals who have shaped my mind.

Over the past two years, I have been immersed in reinforcement learning research, and I cannot believe how much I have grown throughout this journey. I would like to thank my advisor Adam White for his continued support and guidance. I could not have made it to this place today without him. I am honoured to have worked with him and learned from him throughout the past few years. I would also like to thank Martha White, who first agreed to work with me on this project and who introduced me to Adam. She provided me with the opportunities I needed to begin working on this project, and she has been a support for me throughout my graduate journey.

I would like to acknowledge support from NSERC through the Canada Graduate Scholarship, Alberta Innovates through the Graduate Student Scholarship, and Alberta Student Aid through the Alberta Graduate Excellence Scholarship. These organizations have provided major funding during the last few years and have supported the creation of this thesis. The support of these organizations has greatly influenced both my work at the University of Alberta and my thought process as a graduate student. Their support has opened up countless academic opportunities for me.

I would like to thank the examining committee: Adam White, Martha White, and Martin Müeller for taking the time to read through my thesis, provide me with feedback, and conduct the oral examination.

Finally, I would like to acknowledge and thank my parents, who homeschooled me for the majority of grade school. Being homeschooled completely shaped my worldview. My parents knew where I struggled and where I excelled academically. They knew where to put my focus in school, and they shaped my future completely. My parents were the best teachers anyone could ask for. Homeschooling forced me to learn how to teach myself, and it taught me what hard work looks like. These qualities have tremendously helped me excel academically through my graduate studies. I would not be where I am today without my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reinforcement learning (RL) is the study of sequential decision making. The canonical description is that of some intelligent agent interacting with its environment. The interaction is not without purpose. Rather, the agent receives rewards for selecting actions, and it is this cumulative reward signal that the agent seeks to maximize over time.

The method by which the reward signal is maximized is through the agent's policy, which is a function mapping environmental states to probability distributions over actions. By learning an appropriate policy, the agent can learn to take actions which maximize cumulative future rewards. Two classes of algorithms exist for learning this policy. Value-based methods learn this policy indirectly. Policy-search methods actively search the space of policies to find one which maximizes cumulative future rewards.

Benefits exist for both approaches. Arguably, value-based methods are faster to train in discrete-action environments, where the number of available actions is finite. Value-based methods are also better understood than policy-search methods. Compared to value-based methods, policy-search methods are often harder to work with due to the sensitivity of their performance (Henderson et al., 2018; Islam et al., 2017; Neumann et al., 2022; Pourchot et al., 2019). Even so, policy-search methods can find better stochastic policies than value-based methods (Sutton et al., 2018). Furthermore, the application of value-based methods to continuous-action environments is not straightforward, and we must turn to policy-search methods in these cases. Because our world is inherently continuous, policy-search methods are an important tool for real-world learning and are the subject of this thesis.

The study of policy-search methods has produced many successful algorithms over the years. Many of the earliest reinforcement learning algorithms were policy-search algorithms which could solve a variety of now considered simple problems (Barto et al., 1983; Sutton, 1984; Williams, 1992; Williams et al., 1991; Witten, 1977). Recently, there has been a surge of algorithmic development in this area. Many algorithms are able to effectively solve complex simulations (Abdolmaleki et al., 2018; Ciosek et al., 2018, 2020; Degris et al., 2012b; Fujimoto et al., 2018; Haarnoja et al., 2018, 2019; Lillicrap et al., 2016; Mnih et al., 2016; Schulman et al., 2015, 2017; Silver et al., 2014). Others have been used effectively in robotic control (Haarnoja et al., 2018, 2019; Khan et al., 2020).

The class of actor-critic algorithms has been especially popular recently. This class of algorithms learns an actor, which interacts with the environment, and a critic, which critiques the actor. The actor learns which actions lead to maximum reward through the constructive criticism of the critic. These algorithms have become popular because they are sample efficient and can learn in real-time, in contrast to other forms of policy-search which can only learn once the environmental interaction is over. Actor-Critic algorithms are especially well-suited for control problems where actions are naturally continuous. Since our world is inherently continuous, actor-critic algorithms are useful for solving real-world control problems.

Despite their popularity, actor-critic algorithms possess a few limitations. The policy that is learned is generally restricted to a class of distributions, and an optimal policy may not exist in this class of distributions. In addition, the critic must be sufficiently accurate in order to improve the actor adequately. Perhaps the most pressing issue is that these algorithms are fragile and hard to use. Actor-Critic algorithms generally exhibit sensitivity to hyperparameters (Duan et al., 2016; Haarnoja et al., 2018, 2019; Henderson et al., 2018; Islam et al., 2017; Neumann et al., 2022; Pourchot et al., 2019). Because of this, hyperparameter selection for these algorithms is generally both computationally expensive and time consuming. Further exacerbating this issue, a given hyperparameter setting for an algorithm may work well on one environment but not another. Even with a well-tuned hyperparameter setting, these algorithms can exhibit high variability in performance (Clary et al., 2018; Henderson et al., 2018; Islam et al., 2017). Finally, small, code-level optimizations have been shown to drastically affect the performance of these algorithms; performance improvements of some algorithms may simply be due to implementation and not novel algorithmic techniques(Engstrom et al., 2020; Tomar et al., 2020). Better understanding actor-critic algorithms is essential for addressing these limitations and progressing in algorithmic development.

To learn a policy, many policy-search algorithms utilize action-value functions. An action value function measures the value of an action in a given state, defined to be the expected sum of future rewards received after taking that action in a given state. We typically learn an approximation to each action's true value using TD learning (Sutton, 1988). If the agent can estimate action values, then it can learn to take highly valued actions more often.

The operator that changes the agent's policy to take highly valued actions more often is called the *greedification* or *policy improvement* operator. Many policy-search and actor-critic algorithms use a greedification operator that performs distribution matching, whereby the learned policy is gradually matched to some target distribution that is known a priori to be performant (Chan et al., 2021). Given an appropriate selection for a target policy, the policy improvement operator can guarantee policy improvement: the updated policy will achieve at least the same amount of reward over time as that achieved by the previous policy, if not more. One popular choice of target policy is the Boltzmann policy, which selects an action proportional to the exponential of that action's value. An improvement operator based on the Boltzmann policy can guarantee policy improvement (Chan et al., 2021; Haarnoja et al., 2018, 2019).

This Boltzmann policy does have several limitations. The policy improvement guarantee is a theoretical guarantee that requires certain assumptions to be satisfied – assumptions that are almost never satisfied in practice. Even if all assumptions are satisfied, this guarantee exists for a slightly modified problem rather than for the problem which we actually care about in practice. Although this modified problem can be more easily solved due to increased exploration (Mei et al., 2019; Ziebart et al., 2008) and smoother optimization landscapes (Ahmed et al., 2019; Shani et al., 2020), it introduces a trade off between finding the optimal policy on the original problem and an improved learning process.

This thesis introduces a new greedification operator for policy-search based on the Cross-Entropy Method (Rubinstein, 1997, 1999, 2001) (CEM). The idea is intuitive and simple: sample many actions and take the most valuable of these actions more often. We call this greedification operator the Conditional CEM (CCEM), since it is somewhat equivalent to the CEM applied on a state-by-state basis. The CCEM slowly concentrates on actions that result in the highest cumulative future reward. The CCEM is agnostic to the policy parameterization or form of the action-value function; it is modular and can be inserted to learn a policy in any existing actor-critic algorithm. In this thesis, we introduce a specific algorithm that uses the CCEM to learn its policy, called Greedy Actor-Critic.

The CCEM has several advantages which we will explore throughout the remainder of this thesis, and which we summarize here. First, the CCEM is a 0-order, global optimization strategy; the algorithm explicitly searches for the global maximum of a non-concave action-value function and can be used to optimize non-differentiable action-value functions, as in the case of discrete-action environments. Similar to the Boltzmann policy, the CCEM provides guaranteed policy improvement. In contrast to the Boltzmann policy, this guarantee is for both the original problem and the entropy regularized problem. Finally, to prevent policy collapse, CCEM incorporates entropy regularization through a proposal policy used only within the CCEM update. The proposal policy selects which actions the algorithm should reason about and update with; the entropy regularization ensures the proposal policy is wide, providing a broad range of actions to reason about while still concentrating on the actions of maximal value over time. Furthermore, as we show empirically, the CCEM can reduce the sensitivity of actor-critic algorithms to certain hyperparameters such as the entropy regularization scale.

Contributions.

- We develop a novel policy improvement operator, the Conditional Cross-Entropy Method (CCEM), for actor-critic algorithms which is designed to reduce the hyperparameter sensitivity of these algorithms, particularly the entropy scale. The CCEM guarantees policy improvement when the true action-value function is known. The CCEM tracks the expected CEM optimizer across states.

- We develop a novel actor-critic algorithm, Greedy Actor-Critic, which uses the CCEM for policy improvement and performs well empirically on a small classic control suite. Furthermore, Greedy Actor-Critic exhibits lower hyperparameter sensitivity than two baseline algorithms, Soft Actor-Critic and Vanilla Actor-Critic, on this suite of environments.

Chapter 2

Background

In this section, we provide a summary of fundamental concepts and notations used throughout this thesis. In the first section, we describe the RL problem formulation, objectives, and value functions. After that, we discuss policy optimization, one of the major forms of model-free reinforcement learning for control. The next section discusses the cross-entropy optimization algorithm, which is the foundation upon which our novel policy greedification operator is built upon. Readers familiar with reinforcement learning can skip to Section 2.4. Readers familiar with the cross-entropy optimization method can skip Section 2.4.

2.1 The Reinforcement Learning Framework

In the reinforcement learning framework, an intelligent agent interacts with an environment by taking actions which alter the environmental state. Upon taking an action in some state, the agent receives a scalar reward, and the environment transitions to a new state. Informally, the agent's goal is to maximize the sum of rewards it receives over time. In this section, we formalize this goal as well as solution methods for reaching this goal.

2.1.1 Markov Decision Processes

In a formal mathematical sense, the interaction between the agent and environment is formalized as a Markov Decision Process (MDP). MDPs can be described as a tuple $(\mathscr{S}, d_0, \mathscr{A}, \mathscr{R}, \mathscr{P}, \gamma)$. $\mathscr{S}$ denotes the set of all environmental states. The state which the MDP starts in is drawn from a starting state distribution d_0. $\mathscr{A}$ denotes the set of all, possibly multi-dimensional, actions. $\mathscr{R} : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$ is the reward function, where $\mathscr{R}(s, a, s')$ denotes the reward for taking action a in state s and transitioning to state s'[1]. We assume that the rewards are bounded $r_{min} \leq \mathscr{R}(s, a, s') \leq r_{max}$ for some $r_{min}, r_{max} \in \mathbb{R}$. $\mathscr{P} : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to [0, \infty)$ is the one-step state transition dynamics. Finally, $\gamma \in [0, 1]$ is the discount factor which determines the importance of delayed, future rewards.

The agent-environment interaction is as follows. At each discrete timestep t, the agent finds itself in some environmental state $S_t \in \mathscr{S}$. Based on this state, the agent selects an action $A_t \in \mathscr{A}$, which is drawn from its policy $\pi(\cdot \mid S_t)$, a function which maps states to distributions over actions. Upon taking action A_t, the environment transitions the agent to a new state S_{t+1} determined by the transition dynamics $\mathscr{P}$ and provides the agent with a reward $R_{t+1} = \mathscr{R}(S_t, A_t, S_{t+1})$.

2.1.2 The Optimization Problem

In an MDP, rewards are given based on the state the agent is in and the action it takes. The *return* is defined as the infinite sum of discounted[2] future rewards after timestep t:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

where γ is a discount factor.

We consider two different problem formulations. Episodic MDPs are those which naturally break the agent-environment interaction into disjoint episodes, whereas in continuing MDPs, the agent-environment interaction goes on indefinitely with no end. In order for Equation 2.1 to be well-defined in episodic

[1]We assume rewards are deterministic. In general, rewards may be stochastic, and the reward for taking action a in state s and transitioning to state s' may be drawn from the distribution $\mathscr{R}(s, a, s')$.

[2]We ignore the average reward formulation of RL in this thesis.

environments, we consider episode termination to be the entering of an absorbing state which transitions only to itself and returns only a reward of 0. In episodic MDPs, $\gamma \in [0, 1]$. In continuing MDPs, we must restrict γ to be in $[0, 1)$ in order for Equation 2.1 to be well-defined.

Informally, the goal of the agent is to find some policy that attains maximum expected return. The reward hypothesis (Sutton et al., 2018) puts this more formally:

> *That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

In a more mathematical sense, we denote π^* as a non-unique policy that attains maximum expected return, and we refer to it as an *optimal policy*. Mathematically, the agent's goal is to solve the following optimization problem:

$$J(\pi) \doteq \mathbb{E}_\pi[G_t] \tag{2.2}$$

$$\pi^* = \arg\max_\pi J(\pi) \tag{2.3}$$

where the expectation is with respect to the agent's policy π and implicitly with respect to the start state distribution d_0 and the transition dynamics $\mathscr{P}$.

2.1.3 Value Functions

To solve the maximization problem in Equation 2.3, many RL algorithms use value functions. State value functions measure the value of a state $s \in \mathcal{S}$, which is the expected return after arriving in s and following the agent's policy π thereafter:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \end{aligned} \tag{2.4}$$

How the agent arrives in this state in left unaccounted for by the state-value function.

An action-value function measures the expected return after taking action a in state s and then following the agent's policy π thereafter:

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \end{aligned} \tag{2.5}$$

How the agent arrives in this state or select this action is left unaccounted for by the action-value function. State- and action-value functions have the relation that the value of a state s is the expected action value in that state:

$$v_\pi(s) = \mathbb{E}_\pi[q_\pi(s, A)] \tag{2.6}$$

There is exactly one state-value function v_* such that $v_*(s) \geq v_\pi(s)\ \forall s \in \mathcal{S},\ \forall \pi$. This is known as the *optimal value function.* Any optimal policy necessarily induces this value function such that $v_{\pi^*}(s) = v_*(s)\ \forall s \in \mathcal{S}$. Solving the optimization problem in Equation 2.3 reduces to finding some policy that induces v_*.

Soft Value Functions

Soft value functions are value functions which take the stochasticity of the learned policy into account. Soft state-value functions are defined as the expected return after arriving in a state $s \in \mathcal{S}$, augmented by the entropy of the policy:

$$v_\pi^\tau(s) \doteq \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k [R_{t+k+1} + \tau \mathscr{H}_{t+k}] \mid S_t = s\right] \tag{2.7}$$

where $\mathscr{H}_t = \mathscr{H}(\pi(\cdot \mid S_t))$ is the Shannon entropy of the policy in state S_t defined as $\mathscr{H}(\pi(\cdot \mid S_t)) = -\mathbb{E}_\pi[\ln \pi(\cdot \mid S_t)]$. $\tau \in \mathbb{R}^+$ determines the relative importance of rewards and entropy and is referred to as the *entropy scale*. Similarly, soft action-value functions are defined as the expected return after taking an action $a \in \mathcal{A}$ in state s, augmented by the entropy of the policy:

$$\begin{aligned} q_\pi^\tau(s,a) &\doteq \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi^\tau(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \sum_{k=1}^{\infty} \gamma^k (R_{t+k+1} + \tau \mathscr{H}_{t+k}) \mid S_t = s, A_t = a\right] \end{aligned} \tag{2.8}$$

Soft value functions are generalizations of value functions, which can be recovered when $\tau = 0$.

2.1.4 Generalized Policy Iteration

Many algorithms in RL extensively utilize value functions. These value functions satisfy recursive relationships, known as Bellman equations, which allow them to be effectively learned. The recursive relationships relate the (soft) value of a state or state-action pair to the (soft) value of a successive state or state-action pair respectively. The Bellman equations for each type of value function are:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
v_\pi^\tau(s) &= \mathbb{E}_\pi[R_{t+1} + \tau \mathscr{H}_t + \gamma v_\pi^\tau(S_{t+1}) \mid S_t = s] \\
q_\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma \mathbb{E}_\pi[q_\pi(S_{t+1}, A_{t+1}) \mid S_{t+1}] \mid S_t = s, A_t = a] \\
q_\pi^\tau(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi^\tau(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(\tau \mathscr{H}_{t+1} + \mathbb{E}_\pi[q_\pi^\tau(S_{t+1}, A_{t+1}) \mid S_{t+1}]) \mid S_t = s, A_t = a]
\end{aligned}$$

A process known as Generalized Policy Iteration (GPI) uses the Bellman equations to find an optimal policy. GPI refers to the general idea of interleaving two processes, policy evaluation and policy improvement. Policy evaluation refers to learning the value function of some policy, typically through an approximate dynamic programming algorithm utilizing the Bellman equations, such as TD learning or Sarsa (Rummery et al., 1994; Sutton, 1988; Sutton et al., 2018). Policy improvement refers to improving a policy using its learned value function estimate, typically by increasing the likelihood with which highly-valued actions are selected. We refer to the operator which improves the policy as the *policy improvement* or *greedification* operator. Given some policy π, the greedification operator produces another policy π' such that:

$$v_\pi(s) \le v_{\pi'}(s) \quad \forall s \in \mathcal{S} \tag{2.9}$$

We say that π' is at least as good as π since it will achieve at least as much return from each state as π, if not more. If the equality is strict, then we say that π' is better than π. The classical policy improvement theorem states that the interleaving of these two processes, policy evaluation and policy improvement, will eventually recover the optimal policy and optimal value function in a finite number of iterations (Sutton et al., 2018).

Whereas GPI refers to interleaving policy evaluation and policy improvement, policy iteration (PI) refers to the extreme case of GPI. In PI, each stage is carried out to completion. Policy evaluation exactly computes the value function of the policy. Policy improvement exactly improves the policy.

Sarsa

Many different GPI algorithms exist, but perhaps the most popular are temporal-difference (TD) methods (Sutton, 1988). TD methods are attractive because they allow learning in real-time from raw trial-and-error interactions with the environment, without any model of the environment dynamics. Sarsa is such a TD algorithm (Rummery et al., 1994; Sutton et al., 2018). Sarsa works by initializing an action value function estimate $Q(s, a) \approx q_\pi(s, a)$. At each step of the agent-environment interaction, the agent finds itself in state S_t. An action A_t is selected from the agent's policy, usually an ε-greedy policy where the probability of sampling each action is:

$$\pi(a \mid S_t) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q(S_t, a') \\ \frac{\varepsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{2.10}$$

The action A_t is sent to the environment, transitioning to a new state S_{t+1} and providing the agent with a reward R_{t+1}. The Sarsa algorithm then updates the action-value estimate as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{2.11}$$

where $\alpha \in \mathbb{R}^+$ is called the step-size and $A_{t+1} \sim \pi(\cdot \mid S_{t+1})$. The ε-greedy policy is then improved implicitly due to the updated action value function estimate.

2.1.5 Function Approximation and Policy Iteration

Until now, we have focused on algorithms in a tabular setting, in which the policy and value function could be stored in a table. This is possible when the state and action spaces are small and discrete. In many interesting problems though, the state space $\mathcal{S}$ is continuous; in such cases we can no longer store our estimates in a table and must turn to function approximation, the process of approximating an unknown function using a parameterized function. When using function approximation in the context of GPI, we refer to GPI as *Approximate Policy Iteration* (API), since a function approximator approximates the true underlying function of interest. In contrast to PI, API only approximates each step of GPI.

We can extend the Sarsa algorithm above to work in the function approximation setting. Consider the approximate action-value function $q_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$. Semi-gradient Sarsa (Sutton et al., 2018) is an extension of Sarsa to work in the function approximation setting. This algorithm utilizes the following update:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(R_{t+1} + \gamma q_{\boldsymbol{\theta}}(S_{t+1}, A_{t+1}) - q_{\boldsymbol{\theta}}(S_t, A_t))\nabla_{\boldsymbol{\theta}} q_{\boldsymbol{\theta}}(S_t, A_t) \tag{2.12}$$

where $\alpha \in \mathbb{R}^+$ is the step-size. We refer to this algorithm as a semi-gradient algorithm because the update above uses only a part of the gradient of the true objective function of interest (Sutton et al., 2018).

One of the simplest ways to parameterize a value function is with linear function approximation. In this case, our value function approximation is $q_{\boldsymbol{\theta}}(s, a) = \boldsymbol{\theta}^\top \mathcal{F}$, where $\mathcal{F} \in \mathbb{R}^n$ are some features and $\boldsymbol{\theta} \in \mathbb{R}^{n \times p}$. In the discrete action setting we typically use state features $\mathcal{F} = \mathcal{F}(s)$ with $p = |\mathcal{A}|$. In the continuous action setting, we typically use state and action features $\mathcal{F} = \mathcal{F}(s, a)$ with p being the action dimensionality $\mathcal{A} \subseteq \mathbb{R}^p$. Linear function approximation is simple to use and computationally efficient. Linear function approximators in RL are also well-understood, and certain convergence guarantees exist(Sutton et al., 2008, 2009, 2016). Unfortunately, the quality of approximation may depend on the state features. In contrast, nonlinear function approximators such as neural networks are able to learn complicated, nonlinear features and can provide better approximations than linear function approximators in some cases. Nevertheless, the nonlinear setting is not as well understood as the linear case.

The previous examples considered discrete action spaces, but action spaces can be continuous as well. In such a case, we must turn to function approximation to approximate both value function and policy. The value function can be approximated as previously described. To be able to efficiently select actions in environments with continuous action spaces, we use a parameterized policy $\pi_{\boldsymbol{\phi}}$ with parameters $\boldsymbol{\phi}$. For example, we can approximate a Gaussian policy $\pi_{\boldsymbol{\phi}}(\cdot \mid s) = \mathcal{N}(\mu, \sigma)$ using linear function approximation:

$$\begin{bmatrix} \mu \\ \ln \sigma \end{bmatrix} = \boldsymbol{\phi}^\top \mathscr{F}(s)$$

with the state features $\mathscr{F}(s) \in \mathbb{R}^m$ and $\boldsymbol{\phi} \in \mathbb{R}^{m \times 2}$. Similar to the case of value-functions, we can use linear or non-linear function approximators to learn a policy.

2.2 A Gentle Introduction to Policy Optimization

In the previous section, we discussed parameterized policies, but we did not describe how these policies are learned. In this section, we discuss policy optimization, the process of searching the space of policies, or a subset thereof, to find one which maximizes Equation 2.3. We first discuss one of the seminal results of policy optimization. Next, we discuss difficulties in learning the parameters of a parameterized policy and how to circumvent some of these issues. Next, we discuss actor-critic algorithms, and theory tying many policy optimization algorithms together. Finally, we discuss a number of specific actor-critic algorithms considered in this thesis.

2.2.1 The Policy Gradient Theorem

One of the seminal results of policy optimization is the policy gradient theorem (Sutton et al., 1999), which provides the gradient of Equation 2.3:

$$\begin{aligned}\nabla_\phi J(\pi_\phi) &= \nabla \mathbb{E}_{\pi_\phi}[G_t] \\ &= \mathbb{E}_{S_t \sim d_{\pi_\phi}, A_t \sim \pi_\phi}\left[q_{\pi_\phi}(S_t, A_t) \nabla \ln \pi_\phi(A_t \mid S_t)\right] \\ &= \mathbb{E}_{S_t \sim d_{\pi_\phi}, A_t \sim \pi_\phi}\left[G_t \nabla \ln \pi_\phi(A_t \mid S_t)\right]\end{aligned} \tag{2.13}$$

where d_{π_ϕ} is the normalized discounted state visitation distribution[3] defined as:

$$d_{\pi_\phi}(s) \doteq (1-\gamma)\sum_{k=0}^{\infty} \gamma^k \mathbb{P}(S_k = s) \tag{2.14}$$

and $\mathbb{P}$ denotes the probability of some event. Policy gradient methods comprise a class of algorithms that utilize Equation 2.13 in the policy improvement step of approximate policy iteration. These algorithms generally estimate the return G_t using Monte Carlo rollouts or a value function and greedify the policy by taking an ascent step in the direction of the policy gradient $\nabla_\phi J(\pi_\phi)$. Many on-policy methods (Schulman et al., 2015, 2017; Williams, 1992) attempt to obtain unbiased samples of the policy gradient and update the policy parameters in this unbiased direction. In general though, an unbiased estimate of the policy gradient is difficult to obtain.

2.2.2 Difficulties with Unbiased Gradient Estimates

The difficulty in obtaining unbiased estimates of the policy gradient stems from a sampling issue: full return trajectories from states weighted under d_{π_ϕ} must be sampled. Many episodes of agent-environment interaction are required to obtain a sufficiently accurate, unbiased estimate of the return in the policy gradient, Equation 2.13, making this sampling procedure inefficient. Furthermore, in practice we cannot sample from d_{π_ϕ} but rather sample from the so-called *on-policy distribution*, the proportion of time spent in each state. In this case, the policy gradient is weighted by γ^t, decreasing its magnitude

[3]In many works, d_{π_ϕ} is not a distribution because it does not integrate to 1. Here, we have normalized d_{π_ϕ} so that Equation 2.13 is well-defined.

as time goes on and increasing the number samples required to estimate the gradient.

2.2.3 Increasing Sample Efficiency

Because obtaining an unbiased policy gradient estimate is inefficient, many algorithms utilize tricks to increase sample efficiency while also increasing bias in the policy gradient estimate. Typical algorithms ignore the γ^t term when sampling from the on-policy distribution, introducing bias while increasing sample efficiency (Nota et al., 2020). To further increase sample efficiency, off-policy algorithms do not even attempt to sample states according to the discounted state visitation distribution d_{π_ϕ} (Degris et al., 2012b; Fujimoto et al., 2018; Haarnoja et al., 2018, 2019; Lillicrap et al., 2016; Silver et al., 2014; Wang et al., 2017). Instead, these algorithms sample states from other distributions, typically induced by an experience replay buffer (Lin, 1992), which is a method of storing transitions of states, actions, and rewards in a buffer and later using these transitions to update the policy parameters. This practice increases sample efficiency by allowing the algorithm to make updates at arbitrary states arbitrarily often, but incurs the risk of increasing the bias of the gradient estimate. The bias of the policy gradient has been analyzed for the on-policy (Nota et al., 2020; Thomas, 2014) and off-policy (Graves et al., 2021; Imani et al., 2018) settings.

2.2.4 Actor-Critic Algorithms

Many algorithms further increase sample efficiency – at the cost of increased bias to the gradient estimate – by using biased estimates of the return G_t in the form of a value function approximation. These algorithms are collectively known as actor-critic algorithms and are the focus of this thesis. Actor-Critic algorithms are comprised of two parts. The actor is a learned parametric policy. The critic is a value function, typically a parametric value-function approximation learned using some temporal difference algorithm. Actor-Critic algorithms use the critic as a biased estimate of the return in Equation 2.13

(Degris et al., 2012a,b; Mnih et al., 2016). An estimate of the policy gradient can be constructed with a state-value critic, an action-value critic, or both:

$$\begin{aligned}
\nabla_{\boldsymbol{\phi}} J(\pi_{\boldsymbol{\phi}}) &\approx \mathbb{E}_{S_t \sim \mathscr{D}, A_t \sim \pi_{\boldsymbol{\phi}}} \left[(R_{t+1} + \gamma v_{\boldsymbol{w}}(S_{t+1}) - b(S_t)) \nabla \ln \pi_{\boldsymbol{\phi}}(A_t \mid S_t) \right] \\
\nabla_{\boldsymbol{\phi}} J(\pi_{\boldsymbol{\phi}}) &\approx \mathbb{E}_{S_t \sim \mathscr{D}, A_t \sim \pi_{\boldsymbol{\phi}}} \left[(q_{\boldsymbol{\theta}}(S_t, A_t) - b(S_t)) \nabla \ln \pi_{\boldsymbol{\phi}}(A_t \mid S_t) \right]
\end{aligned} \tag{2.15}$$

where $v_{\boldsymbol{w}}$ is a state-value approximation, $q_{\boldsymbol{\theta}}$ is an action-value approximation, and $\mathscr{D}$ denotes some distribution which induces an importance over states; for example $\mathscr{D}$ could be the discounted state visitation distribution or a distribution induced by a replay buffer. Typically, a baseline b that depends only on state is also incorporated. A common choice for a baseline is the state-value, approximated using either a parameterized state-value function approximation or using a sampled expectation as in Equation 2.6. The baseline serves to decrease the variance in the gradient estimate without introducing any additional bias. Action-value baselines can also be used (Liu et al., 2018; Wu et al., 2018) but may not have any benefit over state-value baselines (Tucker et al., 2018).

The use of Equation 2.15 is not a strict requirement for actor-critic algorithms, and many actor-critic algorithms utilize different methods to learn a policy (Fujimoto et al., 2018; Haarnoja et al., 2018, 2019; Lillicrap et al., 2016; Silver et al., 2014). Although this class of algorithms may seem somewhat incohesive, recent research has shown that a large number of these algorithms can actually be seen as instances of API, where the policy improvement operator for each algorithm is based on the same distribution – the Boltzmann distribution.

2.2.5 A Common Framework for Policy Optimization

In this section, we discuss theory which ties together many policy gradient and actor-critic algorithms into a common framework. We begin by discussing the Boltzmann distribution and its use in the policy improvement step of many policy gradient and actor-critic algorithms. Next, we discuss the entropy-regularized RL objective which is induced by the Boltzmann distribution. Finally, we discuss theoretical guarantees of using the Boltzmann distribution in the policy improvement step.

Recent work has unified many policy gradient and actor-critic algorithms into a common framework as approximate policy iteration (Chan et al., 2021; Ghosh et al., 2020; Lazić et al., 2021; Tomar et al., 2020; Vieillard et al., 2020). In particular, many of these algorithms implement the policy improvement step of API by matching the learned policy to the Boltzmann policy, defined as:

$$\mathscr{B}_\tau q^\tau_\pi(s,a) = \frac{\exp(q^\tau_\pi(s,a)\tau^{-1})}{\int_{\mathscr{A}} \exp(q^\tau_\pi(s,b)\tau^{-1})db} \tag{2.16}$$

with entropy scale parameter $\tau \in \mathbb{R}^+$. When τ is non-zero, the Boltzmann distribution induces a new, entropy-regularized RL objective:

$$J_\tau(\pi) \doteq \mathbb{E}_\pi \left[G_t + \tau \sum_{k=0}^{\infty} \mathscr{H}(\pi(\cdot \mid S_{t+k})) \right] \tag{2.17}$$

$$\pi^*_\tau = \arg\max_\pi J_\tau(\pi) \tag{2.18}$$

This objective is referred to as the maximum-entropy or entropy-regularized objective. Algorithms that optimize this objective for non-zero τ are sometimes referred to as maximum-entropy algorithms. As $\tau \to 0$, the original objective (Equation 2.3) is recovered.

The Boltzmann policy provides guaranteed policy improvement, which is likely a major reasons for its widespread adoption in actor-critic algorithms. The Boltzmann policy is guaranteed to be at least as good as the current policy π under the soft value functions: the soft value of any state under $\mathscr{B}_\tau q_\pi$ is at least as high as the soft value of any state under π, if not higher:

$$v^\tau_{\mathscr{B}_\tau q^\tau_\pi}(s) \geq v^\tau_\pi(s) \quad \forall s \in \mathcal{S}$$

By completely matching π to the Boltzmann policy, the updated policy π' will be guaranteed to be at least as good as the previous policy. In theory, we could eventually recover an optimal policy due to the policy improvement theorem.

Generally, recovering an optimal policy is not possible in practice. Since tractable policies are preferred, parameterized policies π_ϕ belonging to a specific distributional family are most often used. Hence, the learned policy π_ϕ

may belong to a different class of distributions than $\mathscr{B}_\tau q^\tau_{\pi_\phi}$, meaning that these distributions cannot be matched exactly. Even if a general policy were used, local optimization algorithms would typically be unable to completely match this policy to the Boltzmann policy.

One may wonder why not use the Boltzmann directly instead of a parameterized policy π_ϕ. Although this is theoretically possible, it is not practically feasible. Sampling from the Boltzmann policy is prohibitively expensive. Instead, many policy gradient and actor-critic algorithms use a parameterized policy, π_ϕ, and match this policy to the Boltzmann policy by reducing a KL divergence. Sampling from the parameterized policy is simple and mimics sampling from the Boltzmann policy.

2.3 Algorithms Considered in this Thesis

In this thesis, we focus on four different actor-critic algorithms. The first of these algorithms is a novel algorithm which we introduce in Chapter 4. In Chapter 3 we consider a case study on two actor-critic algorithms: the original variant of Soft Actor-Critic (Haarnoja et al., 2018) and Deep Deterministic Policy Gradient (Lillicrap et al., 2016). In later chapters, we compare our novel actor-critic algorithm to two baseline algorithms, the modern version of Soft Actor-Critic (Haarnoja et al., 2019) and Vanilla Actor-Critic.

2.3.1 Soft Actor-Critic

Soft Actor-Critic (Haarnoja et al., 2018, 2019) is an off-policy actor-critic algorithm which maximizes the entropy regularized objective in Equation 2.17. This algorithm has two different widely used variants, the original variant (SAC) was proposed by Haarnoja et al. (2018) and a modern variant (SAC-M) was proposed by Haarnoja et al. (2019). We compare our novel actor-critic algorithm to a SAC-M baseline (Haarnoja et al., 2019) for a number of reasons. First, SAC-M is widely considered to outperform its predecessor SAC. Second, using Soft Actor-Critic as a baseline allows us to compare the

Boltzmann target policy over soft action values to the target policy used by our novel algorithm. Third, SAC-M optimizes a different objective from our novel algorithm and allows us to make comparisons between these objective functions. Finally, SAC-M possesses many tricks to improve performance and has widely reported success, making it a strong baseline algorithm on many challenging benchmark environments[4].

Algorithm 1: Modern Soft Actor-Critic

```
1  Input: $b \in \mathbb{N}$; $\alpha_{critic}$, $\alpha_{actor} \in \mathbb{R}^+$; $\beta \in (0, 1]$; $\tau \in \mathbb{R}$
2  Initialize: parameters $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_1^{targ}, \boldsymbol{\theta}_2^{targ}, \boldsymbol{\phi}$ and replay buffer $\mathscr{B}$;
3  Define: $q^\tau_{\boldsymbol{\theta}_{min}}(S, A) = \min_{i \in \{1,2\}} q^\tau_{\boldsymbol{\theta}_i}(S, A)$
4  Obtain: initial state $S_{current}$
5  while $S_{current}$ not terminal do
6      Take action $A_{current} \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S_{current})$ and observe $R_{next}$, $S_{next}$
7      Add $(S_{current}, A_{current}, R_{next}, S_{next})$ to replay buffer $\mathscr{B}$
8      Sample a random batch of transitions $B = (S, A, R, S')_{i=1}^b \sim \mathscr{B}$
9      Update parameters $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$ with $A' \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S')$:
10         $g_i \leftarrow \nabla_{\boldsymbol{\theta}_i} \sum_{(S,A,R,S') \in B} (R + \gamma q^\tau_{\boldsymbol{\theta}^{targ}_{min}}(S', A') - \tau \ln(\pi_{\boldsymbol{\phi}}(A' \mid S')) - q^\tau_{\boldsymbol{\theta}_i}(S, A))^2$
11         $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i + g_i \alpha_{critic}$
12     Update target parameters $\boldsymbol{\theta}_1^{targ}, \boldsymbol{\theta}_2^{targ}$:
13         $\boldsymbol{\theta}_i^{targ} \leftarrow (1 - \beta)\boldsymbol{\theta}_i^{targ} + \beta \boldsymbol{\theta}_i$
14     Update parameters $\boldsymbol{\phi}$:
15         $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha_{actor} \nabla \sum_{S \in B} \mathscr{KL}(\pi_{\boldsymbol{\phi}}(\cdot \mid S) \mid\mid \mathscr{B}_\tau q^\tau_{\boldsymbol{\theta}_{min}}(\cdot \mid S))$
16     $S_{current} \leftarrow S_{next}$
   end
```

Modern Soft Actor-Critic, shown in Algorithm 1, utilizes experience replay to learn both actor and critic. The critic consists of two soft action-value functions, $q^\tau_{\boldsymbol{\theta}_1}$ and $q^\tau_{\boldsymbol{\theta}_2}$, which can alleviate positive bias known to degrade performance of some algorithms (Fujimoto et al., 2018; Hasselt, 2010). The effective action-value predicted by the critic is the minimum action-value predicted by $q^\tau_{\boldsymbol{\theta}_1}$ and $q^\tau_{\boldsymbol{\theta}_2}$, shown in line 3 of Algorithm 1. The critic is learned using the Sarsa algorithm (Rummery et al., 1994; Sutton et al., 2018) (Algorithm 1, lines 9 to 11). States, actions, rewards, and next states are sampled from an experience replay buffer (Algorithm 1, line 8), and the next action is sampled on-policy

[4]See https://spinningup.openai.com/en/latest/spinningup/bench.html

(Algorithm 1, line 9). The next-action value for the Sarsa update is predicted as the minimum action-value of two target networks (Algorithm 1, line 10). The two target networks are learned by slowly updating the weights of each target network toward the weights of the respective action-value function of the critic (Algorithm 1, line 13).

```
Algorithm 2: Soft Actor-Critic
1  Input: $b \in \mathbb{N}$; $\alpha_{action}, \alpha_{state}, \alpha_{actor} \in \mathbb{R}^+$; $\beta \in (0,1]$; $\tau \in \mathbb{R}$
2  Initialize: parameters $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{w}, \boldsymbol{w}^{targ}, \boldsymbol{\phi}$, and replay buffer $\mathscr{B}$;
3  Define: $q^{\tau}_{\boldsymbol{\theta}_{min}}(S, A) = \min_{i \in \{1,2\}} q^{\tau}_{\boldsymbol{\theta}_i}(S, A)$
4  Obtain: initial state $S_{current}$
5  while $S_{current}$ not terminal do
6      Take action $A_{current} \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S_{current})$ and observe $R_{next}, S_{next}$
7      Add $(S_{current}, A_{current}, R_{next}, S_{next})$ to replay buffer $\mathscr{B}$
8      Sample a random batch of transitions $B = (S, A, R, S')_{i=1}^{b} \sim \mathscr{B}$
9      Update parameters $\boldsymbol{w}$ with $A' \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S)$:
10         $g_{\boldsymbol{w}} \leftarrow \nabla_{\boldsymbol{w}} \sum_{S \in B} \left[ v^{\tau}_{\boldsymbol{w}}(S) - q^{\tau}_{\boldsymbol{\theta}_{min}}(S, A') + \tau \ln \pi_{\boldsymbol{\phi}}(A' \mid S) \right]^2$
11         $\boldsymbol{w} \leftarrow \boldsymbol{w} + g_{\boldsymbol{w}} \alpha_{state}$
12     Update parameters $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$:
13         $g_{\boldsymbol{\theta}_i} \leftarrow \nabla_{\boldsymbol{\theta}_i} \sum_{(S,A,R,S') \in B} (R + \gamma v^{\tau}_{\boldsymbol{w}^{targ}}(S') - q^{\tau}_{\boldsymbol{\theta}}(S, A))^2$
14         $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i + g_{\boldsymbol{\theta}_i} \alpha_{action}$
15     Update target parameters $\boldsymbol{w}^{targ}$:
16         $\boldsymbol{w}^{targ} \leftarrow (1 - \beta) \boldsymbol{w}^{targ} + \beta \boldsymbol{w}$
17     Update parameters $\boldsymbol{\phi}$:
18         $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha_{actor} \nabla \sum_{S \in B} \mathscr{KL}(\pi_{\boldsymbol{\phi}}(\cdot \mid S) \mid\mid \mathscr{B}_{\tau} q^{\tau}_{\boldsymbol{\theta}_{min}}(\cdot \mid S))$
19     $S_{current} \leftarrow S_{next}$
   end
```

To learn a policy $\pi_{\boldsymbol{\phi}}$, SAC-M minimizes the KL divergence (denoted as $\mathscr{KL}$) between its learned parametric policy and the Boltzmann distribution over soft action-values, where the Boltzmann distribution is approximated using the double soft action-value critic, rather than the true soft action-value function (Algorithm 1, line 15). Soft Actor-Critic was originally developed for the continuous action setting (Haarnoja et al., 2018, 2019), but can be extended to the discrete action setting. In the continuous action setting, SAC-M typically utilizes a squashed Gaussian policy which ensures actions are sampled in the legal range for the environment. Furthermore Gaussian policies allow for the use of the reparameterization trick to estimate the gradient in line 15 of Algorithm 1, which can result in a lower variance estimate (Haarnoja et al., 2018, 2019). In the case of discrete actions, we can compute, rather than estimate, this gradient exactly (assuming the true action-values are known) using the likelihood trick (Chan et al., 2021).

Modern Soft Actor-Critic also has the ability to automatically tune the entropy scale hyperparameter τ during learning. We do not use automatic entropy tuning in our experiments in future chapters since we are interested in characterizing the sensitivity of SAC-M to the entropy scale. Because of this, we do not include any further discussion on automatic entropy tuning, neither is it included in Algorithm 1.

We now briefly list the major differences between SAC and SAC-M. The overall SAC algorithm is shown in Algorithm 2:

1. SAC learns a state-value function $v^{\tau}_{\boldsymbol{w}}$ (Algorithm 2, lines 9 to 11) while SAC-M does not.

2. SAC uses a target state-value function. SAC-M uses target action-value functions.

3. The target for the action-value update in SAC is computed with a state-value target network (Algorithm 2, lines 12 to 14). SAC-M uses an action-value target network.

4. The target for the action-value update in SAC utilizes a state-value approximation (Algorithm 2, lines 12 to 14). The target for the action-value update in SAC-M uses on-policy actions and an action-value approximation (Algorithm 1, lines 9 to 11).

2.3.2 Vanilla Actor-Critic

The next baseline we consider is an off-policy actor-critic algorithm called Vanilla Actor-Critic (VanillaAC), shown in Algorithm 3. VanillaAC maximizes

Algorithm 3: Vanilla Actor-Critic

```
1  Input: $b, K \in \mathbb{N}$; $\alpha_{critic}, \alpha_{actor} \in \mathbb{R}^+$; $\beta \in (0, 1]$; $\tau \in \mathbb{R}$
2  Initialize: parameters $\boldsymbol{\theta}, \boldsymbol{\theta}^{targ}, \boldsymbol{\phi}$, and replay buffer $\mathscr{B}$
3  Define: $\hat{v}_{\boldsymbol{\theta}}(s) = \frac{1}{K}\sum_{i=1}^{K} q_{\boldsymbol{\theta}}(s, A)$ with $A \sim \pi(\cdot \mid s)$
4  Obtain: initial state $S_{current}$
5  while $S_{current}$ not terminal do
6      Take action $A_{current} \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S_{current})$ and observe $R_{next}$, $S_{next}$
7      Add $(S_{current}, A_{current}, R_{next}, S_{next})$ to replay buffer $\mathscr{B}$
8      Sample a random batch of transitions $B = (S, A, R, S')_{i=1}^{b} \sim \mathscr{B}$
9      Update parameters $\boldsymbol{\theta}$ with $A' \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S')$:
10         $g \leftarrow \nabla_{\boldsymbol{\theta}} \sum_{(S,A,R,S') \in B} (R + \gamma q_{\boldsymbol{\theta}^{targ}}(S', A') - q_{\boldsymbol{\theta}}(S, A))^2$
11         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + g\alpha_{critic}$
12     Update target parameters $\boldsymbol{\theta}^{targ}$:
13         $\boldsymbol{\theta}^{targ} \leftarrow (1 - \beta)\boldsymbol{\theta}^{targ} + \beta\boldsymbol{\theta}$
14     Update parameters $\boldsymbol{\phi}$:
15         $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha_{actor} \nabla \sum_{S \in B} \mathscr{KL}[\pi_{\boldsymbol{\phi}}(\cdot \mid S) \mid\mid \mathscr{B}_{\tau}(q_{\boldsymbol{\theta}}(\cdot \mid S) - \hat{v}_{\boldsymbol{\theta}}(S))]$
16     $S_{current} \leftarrow S_{next}$
   end
```

the standard RL objective (Equation 2.3). We compare our novel algorithm to a VanillaAC baseline for a number of reasons. First, VanillaAC uses the Boltzmann target policy over action values, a different target policy from the one used by SAC-M. Using VanillaAC as a baseline allows us to compare this version of the Boltzmann target policy to the target policy used by our novel algorithm. Second, both VanillaAC and our novel algorithm optimize the same objective, allowing us to compare the quality of each optimization process. Third, VanillaAC still utilizes entropy regularization to improve the learning process, as does our novel algorithm. Finally, VanillaAC possesses fewer tricks to improve performance than SAC-M. VanillaAC therefore provides a baseline of minimum performance to exceed, since it is a basic algorithm.

Vanilla Actor-Critic utilizes experience replay to learn both actor and critic. The critic itself consists of a single action-value function, in contrast to the critic of SAC-M which uses two soft action-value functions. This action-value function is learned using a Sarsa update (Algorithm 3, line 10) with states, actions, rewards, and next states sampled from an experience replay buffer

(Algorithm 3, line 8) and the next action sampled on-policy (Algorithm 3, line 9). Vanilla Actor-Critic utilizes a critic target network to predict the next action-value in the Sarsa update (Algorithm 3, line 10). The target network is updated through a polyak average of target network and critic weights (Algorithm 3, line 13).

Vanilla Actor-Critic learns a parameterized policy by minimizing a KL divergence between the learned policy and the Boltzmann distribution over action-values (Algorithm 3, line 15). Equivalently, VanillaAC utilizes the gradient in Equation 2.15, but with an added entropy regularization term controlled by τ. An approximate state-value baseline is incorporated into the gradient by sampling a number of actions and computing the average action-value of the sample (Algorithm 3, lines 3 and 15). This provides a lower variance gradient estimate without introducing any additional bias.

2.3.3 Deep Deterministic Policy Gradient.

Deep Deterministic Policy Gradient (DDPG) is an off-policy actor-critic algorithm, outlined in Algorithm 4. DDPG maximizes the standard RL objective in Equation 2.3 and learns a deterministic policy. We do not use DDPG as a baseline against which we compare our novel actor-critic algorithm. Instead, we include DDPG in a case study in Chapter 3 where we consider the current state of deep actor-critic algorithms.

DDPG uses experience replay to learn both actor and critic. Unlike SAC, SAC-M, and VanillaAC which use a Sarsa algorithm to learn a critic, DDPG uses an approximate Q-learning update rule to learn its critic, which consists of a single action-value function (Algorithm 4, lines 9 to 11). DDPG uses a target critic to provide the next action-value in the target of the critic update. DDPG also keeps a deterministic target policy which provides the approximate action of maximal value for the Q-learning update (Algorithm 4, line 10). Both the target critic and target policy are learned through a polyak average of network weights (Algorithm 4, lines 14 to 16).

DDPG learns a parameterized, deterministic policy and can be seen as

Algorithm 4: Deep Deterministic Policy Gradient

```
1  Input: $b \in \mathbb{N}$; $\alpha_{critic}$, $\alpha_{actor} \in \mathbb{R}^+$; $\beta \in (0, 1]$, $\sigma \in \mathbb{R}^+$
2  Initialize: parameters $\boldsymbol{\theta}, \boldsymbol{\theta}^{targ}, \boldsymbol{\phi}, \boldsymbol{\phi}^{targ}$; replay buffer $\mathscr{B}$;
3  Obtain: initial state $S_{current}$
4  while $S_{current}$ not terminal do
5      Take action $A_{crurent} = \pi_{\boldsymbol{\phi}}(S_{current}) + \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, \sigma)$
6      Observe $R_{next}$, $S_{next}$
7      Add $(S_{current}, A_{current}, R_{next}, S_{next})$ to replay buffer $\mathscr{B}$
8      Sample a random batch of size $B = (S, A, R, S')_{i=1}^{b} \sim \mathscr{B}$
9      Update parameters $\boldsymbol{\theta}$:
10         $g \leftarrow \nabla_{\boldsymbol{\theta}} \sum_{(S,A,R,S') \in B} \left[R + \gamma q_{\boldsymbol{\theta}^{targ}}(S', \pi_{\boldsymbol{\phi}^{targ}}(S')) - q_{\boldsymbol{\theta}}(S, A)\right]^2$
11         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + g\alpha_{critic}$
12     Update parameters $\boldsymbol{\phi}$:
13         $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha_{actor} \nabla_{\boldsymbol{\phi}} \sum_{S \in B} q_{\boldsymbol{\theta}}(S, \pi_{\boldsymbol{\phi}}(S))$
14     Update target parameters $\boldsymbol{\theta}^{targ}, \boldsymbol{\phi}^{targ}$:
15         $\boldsymbol{\theta}^{targ} \leftarrow (1 - \beta)\boldsymbol{\theta}^{targ} + \beta\boldsymbol{\theta}$
16         $\boldsymbol{\phi}^{targ} \leftarrow (1 - \beta)\boldsymbol{\phi}^{targ} + \beta\boldsymbol{\phi}$
17     $S_{current} \leftarrow S_{next}$
   end
```

reducing a reverse KL divergence to the Boltzmann distribution in a somewhat degenerate form (Chan et al., 2021). Because DDPG tries to mimic Q-learning in a continuous action setting, its policy attempts to track the maximally-valued action under its critic $q_{\boldsymbol{\theta}}$. Since the critic is differentiable with respect to its action argument and the policy is deterministic, DDPG improves its policy by taking an ascent step in the direction of the gradient of the critic with respect to the policy parameters (Algorithm 4, lines 12 to 13).

Since DDPG learns a deterministic policy, it may not explore sufficiently well. Because of this, random noise is added to the action (Algorithm 4, line 5). Any kind of random noise can be added to the action, but generally noise drawn from a Gaussian distribution (as outlined in Algorithm 4) or an Ornstein-Uhlenbeck process (Uhlenbeck et al., 1930) is used.

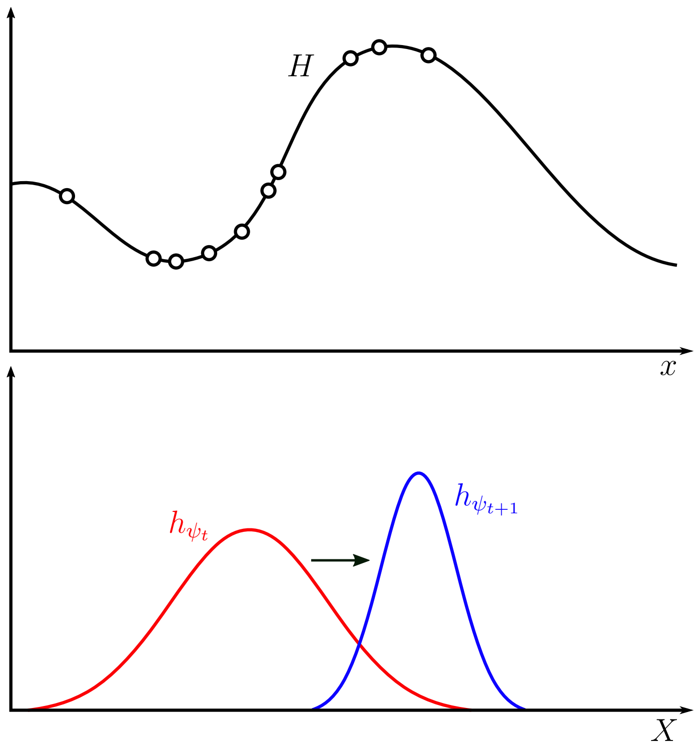

Figure 2.1: Visual depiction of a single CEM optimization update. At the beginning of the update, $\{X_i\}_{i=1}^N$ are sampled from the density $h_{\boldsymbol{\psi}_t}$ (red). The images of these samples under H (open circles on H) are then ordered. The parameters are updated to produce $h_{\boldsymbol{\psi}_{t+1}}$ (blue), which places higher density than $h_{\boldsymbol{\psi}_t}$ on the top percentile of $\{X\}_{i=1}^N$ under H. These steps are repeated until convergence to a maximum of H.

2.4 The Cross-Entropy Method for Optimization

In this section, we provide a brief discussion of the Cross-Entropy Optimization Method (CEM), the optimization algorithm upon which our novel policy greedification algorithm is based. The concern of this section is to understand the CEM algorithm at an intuitive level: the algorithm is left in a somewhat black-box form. For a more formal, mathematical derivation of the CEM, see Appendix B.

The CEM algorithm is a 0-order, global optimization algorithm (Rubinstein, 1999, 2001) originally developed for rare-event probability estimation (Rubinstein, 1997). The algorithm is not new to RL and has been used in the control setting before (Boer et al., 2005; Mannor et al., 2003; Pourchot et al., 2019; Szita et al., 2006). The CEM makes intuitive sense, even without a complete understanding of the underlying mathematics, and is comprised of two phases which are iteratively repeated until the maximum of a function H is discovered. At a high-level, these phases are (1) randomly sample a number of points and (2) increase the probability of sampling points near those which resulted in the highest images under H.

At a mathematical level, consider an objective function $H : \text{dom}(H) \to \mathbb{R}$ where $\text{dom}(H)$ is the domain of H. Consider a probability distribution with density $h_{\boldsymbol{\psi}}$ parameterized by $\boldsymbol{\psi} \in \mathbb{R}^{|\boldsymbol{\psi}|}$ and support over $\text{dom}(H)$. The idea of the CEM is to iteratively update $\boldsymbol{\psi}$ such that $h_{\boldsymbol{\psi}}$ slowly concentrates support on $\arg\max_{\boldsymbol{x} \in \text{dom}(H)} H$. In the discussion below, we consider $\text{dom}(H) = \mathbb{R}$, but the algorithm can be trivially extended to higher dimensions and discrete functions.

Algorithm 5 outlines the intuitive algorithm and Figure 2.1 graphically depicts a single CEM optimization update. We start the update at arbitrary time $t \in \mathbb{N}$ with density $h_{\boldsymbol{\psi}_t}$. In the first phase of the CEM update, we sample $N \in \mathbb{N}$ actions from $h_{\boldsymbol{\psi}_t}$ (Algorithm 5, line 3). Denote the set of sampled values as $I = \{x_1, x_2, \ldots, x_N\}$. In the second phase of the CEM update, we then order these actions based on their images under H : $H_1 = x_{i_1} \leq H_2 = x_{i_2} \leq \ldots \leq H_N = x_{i_N}$ and construct an empirical percentile distribution that contains the $\lfloor \rho N \rfloor$ actions of largest magnitude under H for some $\rho \in (0, 1)$: $I^* = \left\{ x_{i_{\lceil (1-\rho)N \rceil}}, x_{i_{\lceil (1-\rho)N \rceil + 1}}, \ldots, x_N \right\}$ (Algorithm 5, line 4). We complete the second phase by updating $\boldsymbol{\psi}_t$ to increase the density of the actions in I^* under h, which can be done by increasing the log-likelihood of these actions[5] (Algorithm 5, line 5):

$$\boldsymbol{\psi}_{t+1} \leftarrow \boldsymbol{\psi}_t + \alpha \nabla_{\boldsymbol{\psi}} \sum_{x \in I^*} \ln h_{\boldsymbol{\psi}}(x) \tag{2.19}$$

By iteratively repeating this process over and over, we will eventually recover an optimum of H which can be estimated by:

$$\max_{x \in \mathbb{R}} H(x) \approx \frac{1}{K} \sum_{i=1}^{K} H(X_i) \qquad \text{where } X_i \sim h_{\boldsymbol{\psi}_T} \tag{2.20}$$

for some $K \in \mathbb{N}$ and where T is the final iteration of the algorithm (Algorithm 5, line 6). Generally, the parameters $\boldsymbol{\psi}_0$ are initialized such that $h_{\boldsymbol{\psi}_0}$ is

[5] In reality, $\boldsymbol{\psi}_{t+1}$ should be computed as $\arg\max_{\boldsymbol{\psi}} \frac{1}{N} \sum_{x \in I^*} \ln h_{\boldsymbol{\psi}}(x)$, but that is not important for this discussion based on intuition.

wide, which increases the probability of converging to the global optimum of H.

Algorithm 5: Cross-Entropy Method for Optimization

```
1 Input: objective function $H$; initial density $h$; $\rho \in (0,1)$; $N, K \in \mathbb{N}$
2 while not converged do
3     $I \leftarrow N$ random samples from $h$;
4     $I^* \leftarrow \lfloor \rho N \rfloor$ samples in $I$ with highest magnitude under $H$;
5     Increase the probability of sampling the values in $I^*$ under $h$;
  end
6 return $\max_{x \in \mathbb{R}} H \approx \frac{1}{K} \sum_{i=1}^{K} H(X_i)$        where $X_i \sim h$
```

The CEM has a number of benefits over gradient-based optimization algorithms. First, the CEM is a global optimization algorithm. Although this does not guarantee that a global optimum will be found, global optimization algorithms generally find global optima somewhat more frequently than local optimization algorithms do. Second, the CEM can be used to optimize a discontinuous function, whereas gradient based optimization algorithms cannot be used at function discontinuities. Finally, the CEM can be used to optimize both continuous and discrete functions whereas gradient-based optimization algorithms are limited to optimizing continuous functions.

2.5 Summary

In this chapter, we discussed background topics needed for future chapters. We began our discussion by formalizing the reinforcement learning problem. In particular, we discussed MDPs, value functions, optimal value functions, policies, and optimal policies. We also discussed the generalized policy iteration framework whereby iteratively learning the value function of a policy, then adjusting the policy based on its value function, will result in the eventual discovery of an optimal value function and optimal policy.

Next, we briefly discussed policy optimization. We talked about the policy gradient theorem and issues with attaining an unbiased sample of the gradient of the reinforcement learning objective in Equation 2.3. We discussed a few commonly used techniques to increase sample efficiency while increasing bias when estimating this gradient. Finally, we discussed actor-critic algorithms.

In the following section, we tied many policy gradient and actor-critic methods together into a common framework. In particular, we discussed how these algorithms can be seen as instances of approximate policy iteration where the policy greedification step moves the learned policy toward the Boltzmann policy over actions values. We discussed how the Boltzmann policy induces a new reinforcement learning objective, Equation 2.17.

Next, we briefly outlined a number of actor-critic algorithms, Soft Actor-Critic (Haarnoja et al., 2018, 2019) (SAC and SAC-M), Vanilla Actor-Critic (VanillaAC), and Deep Deterministic Policy Gradient (Lillicrap et al., 2016) (DDPG). Soft Actor-Critic is a supposed state-of-the-art algorithm which works in the maximum entropy framework and has many tricks to improve performance such as a double action-value critic and reparameterized action sampling. VanillaAC does not work in the maximum entropy framework and does not have so many tricks to improve performance but still utilizes entropy regularization. DDPG is an actor-critic algorithm that learns a deterministic policy and utilizes an approximation to Q-learning to learn a critic.

Finally, we discussed the Cross-Entropy Optimization Method (CEM) at an intuitive level, without heavily discussing its mathematical derivation. This algorithm is a 0-order, global optimization algorithm that works by repeatedly randomly sampling values and optimizing over only these sampled values. By iteratively repeating this random sampling procedure, the algorithm slowly concentrates on the maximum of a function. The algorithm can also be used to optimize discontinuous and discrete functions where gradient-based optimization algorithms are not available.

Chapter 3

The Fragility of State-of-the-Art Actor-Critic Methods: A Case Study

One of first jobs of an algorithm designer is to understand existing algorithms. Without understanding existing algorithms, the designer cannot know how to improve these algorithms nor address their limitations. In this section, we dive into a case study attempting to better understand a supposed state-of-the-art algorithm, Soft Actor-Critic (Haarnoja et al., 2018). In particular, we attempt to reproduce the experiments of Haarnoja et al. (2018) comparing SAC to DDPG on the HalfCheetah environment. During this process, we come to different conclusions from those of Haarnoja et al. (2018).

We begin with a few important notes. The purpose of this chapter is neither to degrade a specific paper or algorithm nor to imply that such a paper was purposefully designed to mislead. Instead, the purpose of this section is to try to understand a supposed state-of-the-art algorithm at a deeper level. We study SAC because it is a popular, highly-cited algorithm with widely-reported success[1]. Furthermore, the algorithm is widely regarded as state-of-the-art (Ciosek et al., 2019; Haarnoja et al., 2018, 2019; Yu et al., 2021). Finally, this section is not meant to be a comprehensive list of all issues addressed by our novel actor-critic algorithm which is introduced in Chapter 4.

During our attempt to reproduce the comparison of SAC and DDPG on

[1]See `https://spinningup.openai.com/en/latest/spinningup/bench.html`

HalfCheetah as implemented in OpenAI Gym (Brockman et al., 2016), we tried to be as fair as possible to the original work of Haarnoja et al. (2018). We used the original SAC codebase[2] (SAC-CB) with the tuned hyperparameters reported by Haarnoja et al. (2018). We used a DDPG baseline implemented in the RLLab codebase[3] (RLLab-CB) with the hyperparameters set to those used by default in the codebase (Duan et al., 2016), except that the replay buffer capacity, batch size, and network architectures were adjusted to match those used by SAC. Although Haarnoja et al. (2018) did not mention the hyperparameters used for DDPG, their GitHub repository left a number of hints on the configuration of the DDPG baseline as well as the implementation used[4]. In this chapter, we evaluate algorithms in a deterministic fashion by removing any exploration noise for DDPG and selecting only the modal action for SAC as was also done by Haarnoja et al. (2018).

3.1 Reproducing Published Results

In this section, we discuss our attempt to recreate the results of Haarnoja et al. (2018). We encountered a number of difficulties during this process. First, the default implementation of a number of policies in SAC-CB utilize some form of regularization in addition to entropy regularization. Since the paper does not mention regularization, we did not include it in our experiments. Second, many code examples in SAC-CB use action normalization. Since Haarnoja et al. (2018) do not mention action normalization, we did not include it in our experiments either. Finally, two versions of environment wrappers exist in these codebases to alter the interface of OpenAI Gym environments (Brockman et al., 2016) to be compatible with the algorithms from these codebases. One wrapper is implemented in SAC-CB, another in RLLab-CB. We assumed that SAC used the environment wrapper from its own codebase, SAC-CB. Since the

[2]The original SAC codebase can be found at `github.com/haarnoja/sac`

[3]The RLLab codebase can be found at `github.com/rll/rllab`

[4]See `https://github.com/rail-berkeley/softlearning/issues/27`. It is mentioned that the original paper used the implementation of DDPG from RLLab-CB with hyperparameters set similarly to those used by SAC where applicable.

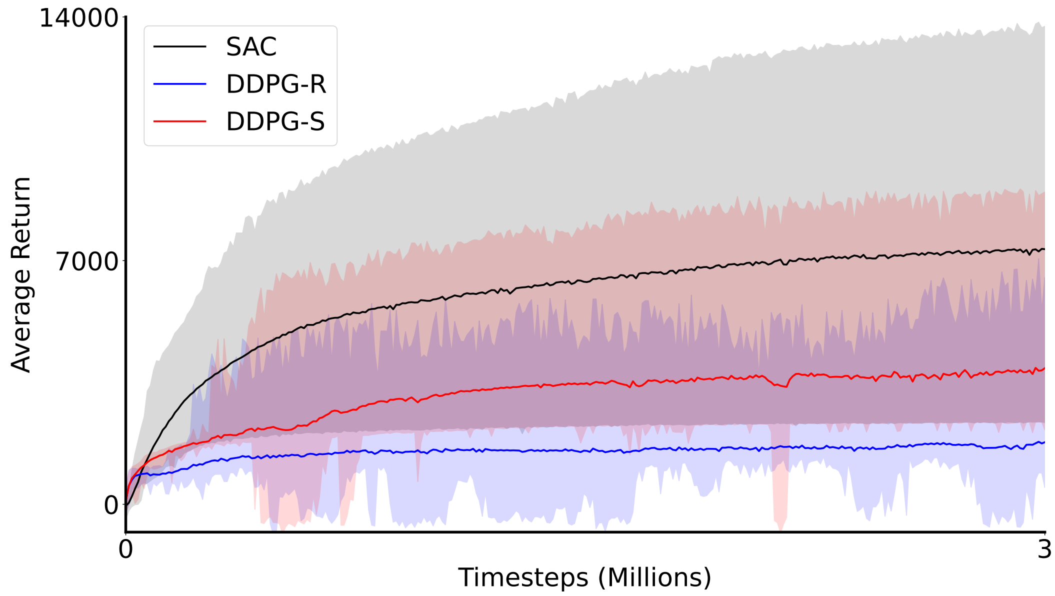

Figure 3.1: Our attempt to recreate the learning curves of Haarnoja et al. (2018) on HalfCheetah. Solid lines denote mean performance over 30 runs with shaded regions denoting minimum and maximum performance. The average performance of SAC is lower than reported by Haarnoja et al. (2018).

DDPG baselines is compatible with both implementations, we included two versions of DDPG, one run on the RLLab-CB environment wrapper (DDPG-R) and another run on the SAC-CB environment wrapper (DDPG-S).

We ran SAC and DDPG on the OpenAI Gym (Brockman et al., 2016) implementation of HalfCheetah for 3 million timesteps over 30 runs. For each of the 30 runs, the episodic return was recorded and averaged over runs to produce the learning curves in Figure 3.1, with shaded regions denoting minimum and maximum performance. Compared to the learning curves reported by Haarnoja et al. (2018), SAC exhibits lower average performance and higher variation in performance.

Upon studying RLLab-CB more closely, we identified a potential bug in the environmental wrapper which causes episode cutoffs to be handled incorrectly. Some algorithms in RLLab-CB have the option to bootstrap on episode cutoffs, considering treating the cutoff state as terminal – using a next-state value of 0 in the TD-error. In this case, the environment is no longer Markovian. Even if this option is explicitly turned off, some algorithms will still incorrectly

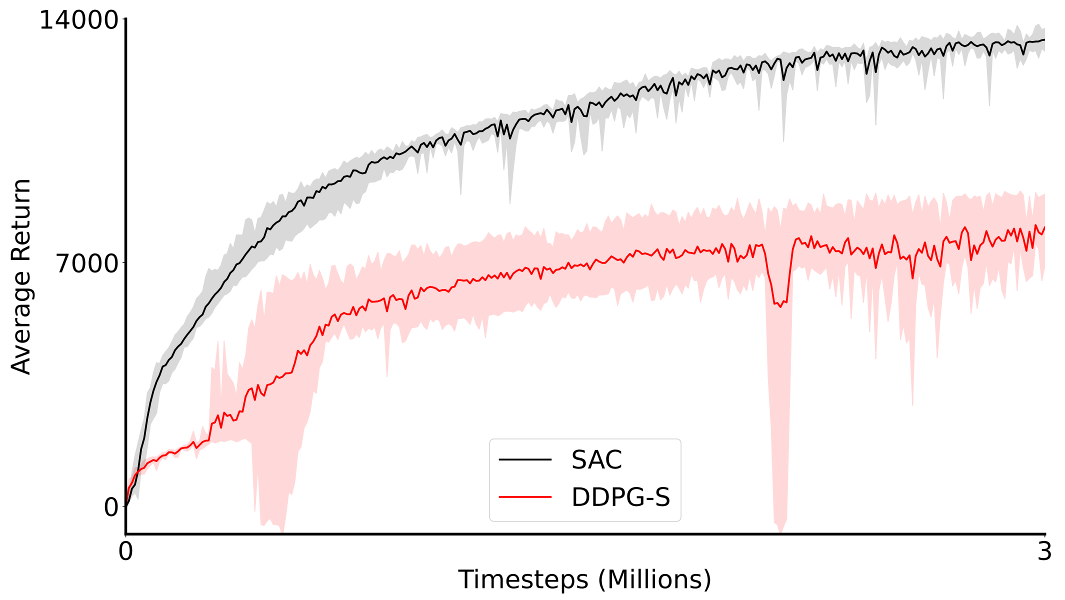

Figure 3.2: Our best attempt to recreate the results of Haarnoja et al. (2018) on HalfCheetah. Solid lines represent mean performance over 5 runs with shaded regions denoting minimum and maximum performance. We tuned random seeds by selecting the 5 runs which induced the highest performance for each algorithm.

bootstrap on episode cutoffs[5]. Given the default settings in RLLab-CB, this implementation flaw will occur. Although we cannot know if this bug affected the results of Haarnoja et al. (2018), the DDPG baseline may have been given a significant disadvantage.

A significant amount of guess work was required to reproduce the results reported by Haarnoja et al. (2018) to the best of our abilities, shown in Figure 3.2. Perhaps Haarnoja et al. (2018) used too few seeds, resulting in optimistic estimates of expected performance and variance in performance. A number of lucky seeds could have been inadvertently used for which SAC performs well. We emulate this by tuning seeds in Figure 3.2, where we report performance over the 5 runs for which each algorithm attained highest performance out of all 30 runs. These results closely match the original results reported by Haarnoja et al. (2018).

[5]This occurs when the wrapped environment has an episode cutoff less than or equal to the cutoff set in the wrapper, which happens by default in the codebase.

3.2 Tuning the Baseline

During our study, we noticed that the performance of DDPG on HalfCheetah was under-reported by Haarnoja et al. (2018)[6]. To better gauge the performance of SAC, we ran similar experiments to those previously described except with a DDPG baseline tuned for HalfCheetah. We used the tuned hyperpa-

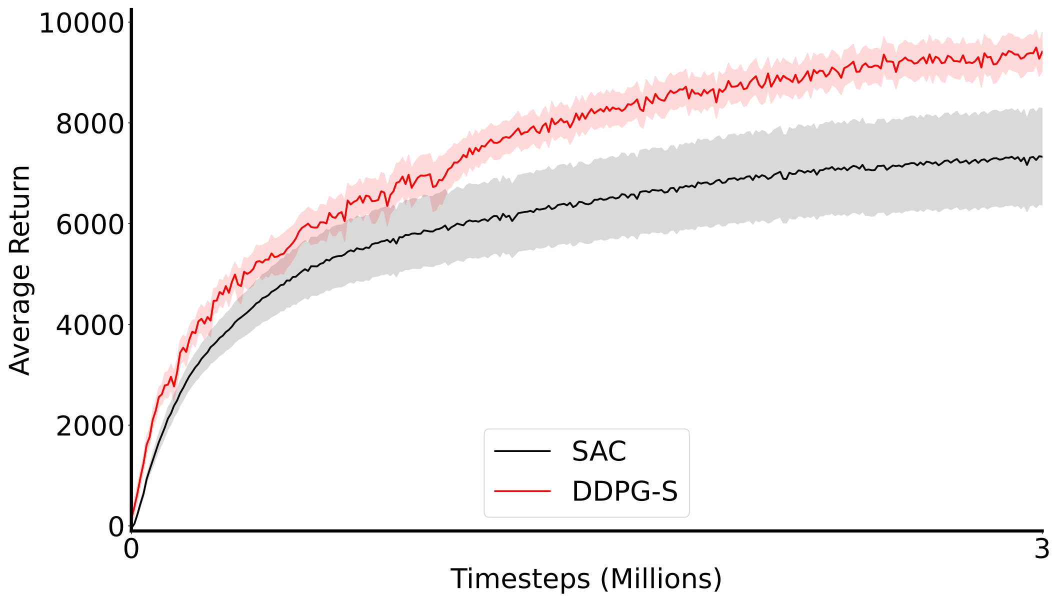

Figure 3.3: Performance of tuned SAC and DDPG on HalfCheetah. Solid lines denote mean performance over 30 runs with shaded regions denote standard error from the mean. The DDPG baseline now performs competitively with SAC.

rameters for DDPG as reported by SpinningUp[7]. We also used uncorrelated, unbounded Gaussian noise for action exploration in DDPG as we found it to outperform OU noise (Uhlenbeck et al., 1930).

We ran SAC and DDPG on HalfCheetah for 3 million timesteps over 30 runs on the environmental wrapper in SAC-CB. The episodic return was recorded and averaged over all 30 runs to generate the learning curves in Figure 3.3, with shaded regions denoting standard error from the mean perfor-

[6]Fujimoto et al. (2018) report higher performance for a modified version of DDPG on HalfCheetah. SpinningUp also reports significantly higher performance for DDPG on HalfCheetah: https://spinningup.openai.com/en/latest/spinningup/bench.html

[7]See https://spinningup.openai.com/en/latest/spinningup/bench.html

mance. A tuned version of DDPG seems competitive with SAC on HalfCheetah, in contradiction to the performance of DDPG reported by Haarnoja et al. (2018).

3.3 Conclusion

Our journey to better understand SAC has led us to different conclusions from those drawn by Haarnoja et al. (2018). In particular, the performance of SAC as reported by Haarnoja et al. (2018) seems optimistic on HalfCheetah. SAC may not be as performant as originally reported, and perhaps better algorithms can be constructed by focusing on solving the original MDP rather than an entropy regularized one. This is the subject of the next chapter.

Chapter 4

The Conditional Cross-Entropy Method and Greedy Actor-Critic

Although the Boltzmann policy is a popular choice for policy greedification, it has limitations. The primary of which is that algorithms based on the Boltzmann are sensitive to the choice of entropy (Chan et al., 2021; Haarnoja et al., 2018, 2019; Neumann et al., 2022; Pourchot et al., 2019). Given that actor-critic algorithms can be viewed as utilizing API, a natural question is which greedification operators are sensible for use within actor-critic algorithms. In this section, we propose and motivate a new greedification operator based on the CEM algorithm. The *Conditional CEM* (CCEM) algorithm greedifies a policy using a CEM update to find maximally valued actions per-state given an action-value function.

4.1 The Conditional Cross-Entropy Optimization Algorithm

We are interested in utilizing the CEM optimization algorithm to optimize an action-value critic. Unfortunately, this process is not straightforward. The CEM algorithm finds a single best set of parameters for a single optimization problem. For example, the CEM can be used to optimize the parameters of a neural network or linear function approximator, which has been studied in RL before (Boer et al., 2005; Mannor et al., 2003; Szita et al., 2006). In contrast, using the CEM to optimize an action-value function has not been thoroughly studied, perhaps since performing this optimization on an action-value critic can be seen as a sequence of non-stationary optimization problems, and the CEM was not designed for this. In this section, we study how the CEM can be adapted to optimize an action-value critic.

Our goal is to adapt the CEM to find maximally valued actions in each state for a parameterized action-value critic $q_{\boldsymbol{\theta}} \approx q_{\pi_{\boldsymbol{\phi}}}$:

$$a^* \doteq \underset{a \in \mathcal{A}}{\arg\max}\, q_{\boldsymbol{\theta}}(s, \cdot) \tag{4.1}$$

where the action-value critic can be learned through any TD method such as Sarsa(λ). We could simply use the CEM to find these highly valued actions without learning the weights for any parametric policy, and simply take one of these highly valued actions in the environment (Kalashnikov et al., 2018); such a procedure throws away prior information obtained from the optimization process and can be prohibitively expensive with high-dimensional action spaces. A better approach is to implicitly learn the parameters of a policy which places high density on these highly valued actions. Such a process retains valuable information obtained during the optimization procedure, allows for better generalization between states, and results in a policy that is simple and efficient to sample from. Furthermore, this policy can be seen as caching an approximate CEM solution at each state. Once we know the approximate solution at each state, we can select actions by sampling rather than completely re-applying the CEM procedure in each state.

The CCEM extends the CEM to (1) be conditioned on state and (2) be learned iteratively over time[1]. Unlike the CEM which uses a single distribution over which to optimize a single and fixed function, the CCEM utilizes a parameterized policy $\pi_{\boldsymbol{\phi}}(\cdot \mid s)$ to optimize a parameterized action-value critic $q_{\boldsymbol{\theta}}$. The parametric critic can be seen as a set of objective functions to optimize, one objective per state. The parametric policy can be seen as a set

[1]The CEM does optimize a function iteratively, but the CCEM differs by learning the expected CEM solution to a sequence of optimization problems – one for each state – which are changing over time due to a changing action value critic. Therefore, we say that the CCEM extends the CEM to be learned iteratively over time.

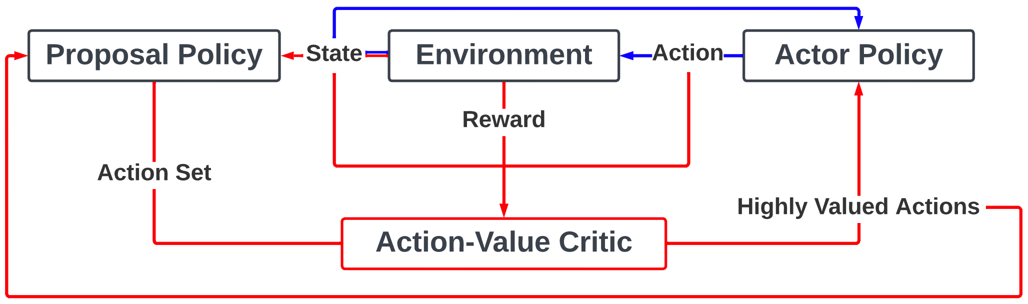

Figure 4.1: The CCEM works by keeping a separate proposal policy which samples an action set to reason about. A set of actions is sampled from the proposal policy and the density of the $\lfloor \rho N \rfloor$ actions of highest value is then increased in both the proposal policy and the actor policy. The proposal policy uses entropy regularization while the actor policy does not. The blue arrows denote agent-environment interactions, while the red arrows denote processes occurring during learning.

of distributions, conditioned on environmental state, used in the CEM optimization process. The CCEM adapts the parametric policy to concentrate on maximally valued actions under the action-value critic on a state-by-state basis. The CCEM is not simply the CEM applied on a set of objective functions. Since the critic is learned over time, the set of objective functions to optimize is non-stationary. Since the policy and critic are both parameterized, state aliasing and generalization between action distributions will occur. These two key facts are accounted for in the CCEM but not in the original formulation of the CEM.

Another key difference between the CEM for optimization and the CCEM is that the CCEM uses a separate parameterized *proposal policy* $\tilde{\pi}_{\phi^{prop}}$ which samples actions to reason about while a parameterized *actor policy* π_ϕ selects actions to take in the environment. Both the proposal and actor policies track the actions of maximal value per-state over time. The crucial detail is that the proposal policy concentrate more slowly than the actor policy to ensure a broad range of actions are considered at each iteration of the optimization process. This is both theoretically and practically important because the action-value critic, the function over which we optimize, is non-stationary. The algorithm

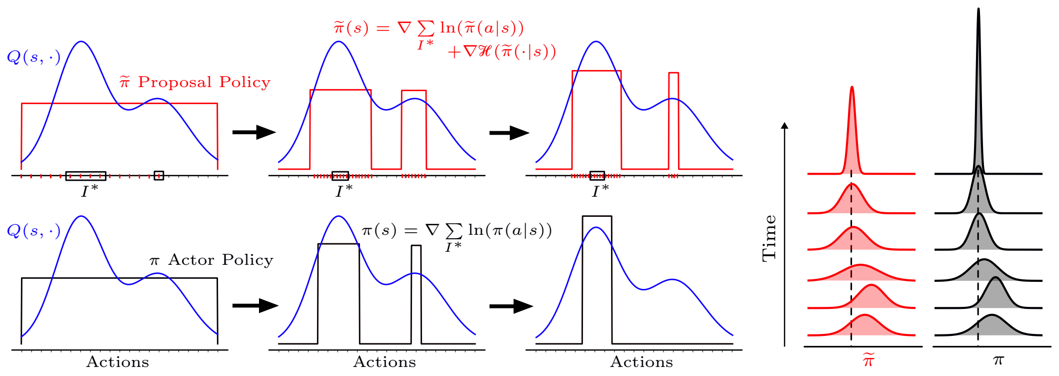

Figure 4.2: **Left** An example progression of the CCEM for multiple updates in a single state. In general, the CCEM is run once per state, with a new state sampled after each update. We use a uniform actor (black) and proposal (red) policy in the figure. From left to right, we see in the first sub-plot a number of actions being sampled (shown as red ticks on the x-axis), with the highest valued actions denoted in black boxes as I^*. In the next sub-plot, the actor and proposal policies have been updated, placing more density on I^*. A new set of actions is then sampled. In the rightmost sub-plot, we see that the actor and proposal policies have been once again updated, but that the actor policy more quickly concentrates on highly valued actions than does the proposal policy. **Right** An actual progression of the CCEM using Gaussian policies on the action values shown in the leftmost figure.

must actively track the optimum of a changing function. We do not want to settle on the optimum of a single instance of the action-value critic in time. To ensure this proposal policy concentrates more slowly than the actor policy, we utilize entropy regularization in the proposal policy[2]. In the actor policy, we do not use entropy regularization, which allows it to concentrate more quickly on greedy actions. This usage of entropy regularization mitigates policy collapse. Figure 4.1 graphically shows the relationship between the environment, actor policy, proposal policy, and critic during the CCEM update. Figure 4.2 demonstrates how the actor and proposal policies change during the CCEM update.

The full CCEM algorithm is presented in Algorithm 6. Similarly to the CEM optimization algorithm, the CCEM uses a two-phase, multi-level ap-

[2]The theory requires a number of conditions to be satisfied for the proposal policy to ensure that it changes slowly enough. These conditions are not always satisfied in practice.

proach for optimization. We first choose a threshold value ζ_t and then update the parameters of the actor and proposal policies to increase the likelihood of actions which are valued higher than this threshold. To choose ζ_t, we use the same repeated random sampling procedure as the CEM with hyperparameter $\rho_t \in (0, 1)$. In general ρ_t may depend on time, but we will use a constant $\rho_t = \rho \in (0, 1)$ for simplicity. Furthermore, the general case of the CCEM can utilize ρ_{actor} for the actor policy and ρ_{prop} for the proposal policy, but we will use $\rho = \rho_{actor} = \rho_{prop}$ for simplicity[3].

The CCEM algorithm proceeds in the following manner. The proposal policy $\widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(\cdot \mid s)$ is sampled to provide a set of $N \in \mathbb{N}$ actions to reason about $I(s) = \{a_1, a_2, \ldots, a_N\}$ in state $s \in \mathcal{S}$ (Algorithm 6, line 6). Similarly to the CEM, we then order each action based on its value (Algorithm 6, line 7) such that:

$$q_{\boldsymbol{\theta}}^{(1)} = q_{\boldsymbol{\theta}}(s, a_{i_1}) \leq q_{\boldsymbol{\theta}}^{(2)} = q_{\boldsymbol{\theta}}(s, a_{i_2}) \leq \ldots \leq q_{\boldsymbol{\theta}}^{(N)} = q_{\boldsymbol{\theta}}(s, a_{i_N}) \tag{4.2}$$

where the superscript denotes an ordering of the action-values $q_{\boldsymbol{\theta}}^{(i)}$. We then set ζ_t as the empirical $(1-\rho)$-percentile of action-values (Algorithm 6, line 9):

$$\zeta_t \doteq q_{\boldsymbol{\theta}}^{(\lceil (1-\rho)N \rceil)} = q_{\boldsymbol{\theta}}(s, a_{i_{\lceil (1-\rho)N \rceil}}) \tag{4.3}$$

We then construct a set $I^*(s) = \{a_i \mid q_{\boldsymbol{\theta}}(s, a_i) \geq \zeta_t,\ a_i \in I(s)\}$ of highly valued actions (Algorithm 6, line 10) and update the parameters of the actor policy using a gradient ascent step on the log-likelihood of actions $I^*(s)$ (Algorithm 6, line 14). We perform a similar update for the proposal policy, except that we utilize entropy regularization (Algorithm 6, lines 11 to 13). Upon transitioning to the next state, the entire algorithm is run again.

One caveat to the CCEM exists that warrants discussion. In later sections, we discuss theoretical results which require that the actor and proposal policies come from the same distributional family. Empirically, we noticed that the algorithm performed satisfactorily when using proposal and actor policies

[3]This perhaps suggests another method to keep the proposal policy highly stochastic, other than entropy regularization

Algorithm 6: Conditional Cross-Entropy Method

```
1  Input: $N \in \mathbb{N}$, $\rho \in (0,1)$, $q_{\boldsymbol{\theta}}$, $s \in \mathcal{S}$, $\alpha, \tau \in \mathbb{R}^+$
2  if $\mathcal{A}$ is discrete and finite then
3      $I^*(s) \leftarrow \arg\max_{a \in \mathcal{A}} \; q_{\boldsymbol{\theta}}(s, a)$
4  else
5      Get a sorted list of action samples:
6          $I(s) \leftarrow \{a_1, a_2, \ldots, a_N\}$ for $a_i \sim \widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(\cdot \mid s)$
7          $q_{\boldsymbol{\theta}}^{(1)} = q_{\boldsymbol{\theta}}(s, a_{i_1}) \leq \ldots \leq q_{\boldsymbol{\theta}}^{(N)} = q_{\boldsymbol{\theta}}(s, a_{i_N})$
8      Construct the empirical percentile distribution:
9          $\zeta_t \leftarrow q_{\boldsymbol{\theta}}^{(\lceil (1-\rho) N \rceil)}$
10         $I^*(s) \leftarrow \{a_i \mid q_{\boldsymbol{\theta}}(s, a_i) \geq \zeta_t, \; a_i \in I(s)\}$
11     Update the proposal policy
12         $g \leftarrow \sum_{a \in I^*(s)} \nabla_{\boldsymbol{\phi}^{prop}} \ln \widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(a \mid s) + \tau \nabla_{\boldsymbol{\phi}^{prop}} \mathcal{H}(\widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(\cdot \mid s))$
13         $\boldsymbol{\phi}^{prop} \leftarrow \boldsymbol{\phi}^{prop} + \alpha g$
14 $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha \sum_{a \in I^*(s)} \nabla_{\boldsymbol{\phi}} \ln \pi_{\boldsymbol{\phi}}(a \mid s)$
```

from different distributional families. Future research should more carefully characterize the theoretical and empirical implications of selecting actor and proposal policies from differing distributional families. For the purposes of this thesis, we will take each distribution to be in the same family in order to match the theory presented in later sections.

The CCEM can be seen in the following, perhaps overly simplistic, manner. Upon transitioning to state $s \in \mathcal{S}$, we perform a CEM optimization update on the action-value critic, using samples drawn from the agent's proposal policy and using entropy regularization where applicable. If we ignored this CEM update at all other states and assume the action-value critic is not changing, then the next time we transition to state s, we will perform this CEM update again. As time goes on, this CEM update is performed over and over again, and the CEM optimization algorithm is in effect recovered in this state under two different policies. One policy, the proposal policy, is kept wider than the other, the actor policy. The caveat is that the CEM updates at states other than s are not actually ignored and the action-value critic is learned concurrently with the policy. This intertwining of CEM updates at each state while updating the action-value critic causes the CCEM algorithm to track the expected CEM optimizer across states. By *track*, we mean that the CCEM slowly solves a non-stationary problem. By *expected*, we mean that this solution is the average solution across states. In the next section, we more carefully characterize this phrase.

CCEM for Discrete Actions. The discussion above has solely considered the continuous action setting. The CCEM attempts to find an action of maximal value in each state. In the continuous action setting, this is accomplished through randomly sampling actions. In the discrete action setting though, this optimization procedure is greatly simplified. We instead perform a maximum likelihood update on all maximally valued actions in the set $I^*(s) = \{a \mid q_{\boldsymbol{\theta}}(s,a) = \max_{b \in \mathscr{A}} q_{\boldsymbol{\theta}}(s,b)\}$ (Algorithm 6, line 3). We no longer have the proposal policy $\widetilde{\pi}_{\boldsymbol{\phi}^{prop}}$ as it is no longer necessary. Furthermore, entropy regularization is no longer used.

4.2 The CCEM Tracks the Expected CEM Optimizer

In this section, we discuss the asymptotic solution of the CCEM. First proven by Neumann et al. (2022), the CCEM tracks the expected solution found by the CEM across states. We first informally reiterate this result and outline the necessary assumptions thereof. We then discuss the significance of this theoretical result and comment on its limitations.

4.2.1 Informal Result

We would like to understand the properties of the stochastic CCEM algorithm. At an intuitive level, we would expect the CCEM to behave similarly to the CEM on a state-by-state basis as outlined at the close of the previous section. There are two caveats though. The first is that the CCEM uses a parameterized policy conditioned on state, representing possibly infinitely many different probability distributions, where we have one distribution per state. Aliasing between action distributions will therefore occur. In contrast, the CEM considers only a single distribution for optimization purposes and does not take into account any such aliasing. Second, the CCEM optimizes an action-value critic, which is a function changing with time and is an inaccurate approximation to the true function we would like to optimize, the true action-value function.

To address the first difficulty, Neumann et al. (2022) identify a number of conditions on the policy parameterization to ensure well-behaved CEM updates and use an ODE that takes expectations over states to analyze the algorithm. To address the second difficulty, Neumann et al. (2022) use a two-timescale stochastic approximation approach where the action-value critic $q_{\boldsymbol{\theta}}$ changes more slowly than the policy $\pi_{\boldsymbol{\phi}}$ which allows the policy to track the maximal action. To account for its own parameters changing, the policy itself has two timescales. Actions for the maximum likelihood step are sampled according to a distribution parameterized by older (slower-moving) parameters. This ensures that the maximum-likelihood update to the primary (faster-moving) parameters uses samples from what look like a fixed distribution. These two distributions correspond to the proposal (slow) and actor (fast) policies in the CCEM algorithm.

Informal Result: Let $\boldsymbol{\theta}_t$ be the parameters of the action-value critic with stepsize $\alpha_{\boldsymbol{\theta}}$, $\boldsymbol{\phi}_t$ the policy parameters with stepsize $\alpha_{\boldsymbol{\phi}}$, and $\boldsymbol{\phi}_t^{prop}$ a more slowly changing set of policy parameters set to $\boldsymbol{\phi}_t^{prop} = (1 - \alpha_{\boldsymbol{\phi}^{prop}})\boldsymbol{\phi}_{t-1}^{prop} + \alpha_{\boldsymbol{\phi}^{prop}}\boldsymbol{\phi}_t$ for stepsize $\alpha_{\boldsymbol{\phi}^{prop}} \in (0, 1]$. Assume:

1. States $S_t \in \mathcal{S}$ are sampled from a fixed marginal distribution
2. $\nabla \ln \pi_{\boldsymbol{\phi}}(\cdot \mid s)$ is locally Lipschitz w.r.t. $\boldsymbol{\phi}$ $\forall s \in \mathcal{S}$
3. Parameters $\boldsymbol{\phi}_t$ and $\boldsymbol{\theta}_t$ remain bounded almost surely
4. Stepsizes are chosen for three different timescales: $\boldsymbol{\phi}_t$ evolves faster than $\boldsymbol{\phi}_t^{prop}$, and $\boldsymbol{\phi}_t^{prop}$ evolves faster than $\boldsymbol{\theta}_t$

Then the CCEM actor tracks the expected CEM optimizer across states [4].

Why is this result useful? The primary concern is that the CCEM is not a gradient-descent approach and so requires a different analysis to ensure that the stochastic noise in the update remains bounded and is asymptotically negligible. In particular, the underlying ODE for the CCEM update needs to be characterized. Furthermore, any classical results of the CEM do not immediately apply to the CCEM because such results assume distribution parameters can be computed directly (for example, the mean and variance of a Gaussian) and are not the outputs of parameterized functions such as neural networks. The conditions and result stated above ensure that the CCEM updates are well-behaved and that the stochastic noise is asymptotically negligible.

4.2.2 Limitations

As is often the case, the theory does not perfectly characterize the CCEM algorithm we use in practice. The theory requires a number of assumptions to hold, but in practice these assumptions are generally relaxed. The above theoretical result, although useful, has several limitations.

First, in practice we do not use the update $\boldsymbol{\phi}_t^{prop} = (1 - \alpha_{\boldsymbol{\phi}^{prop}})\boldsymbol{\phi}_{t-1}^{prop} + \alpha_{\boldsymbol{\phi}^{prop}}\boldsymbol{\phi}_t$ to learn the weights of the proposal policy. We instead use a maximum likelihood update on the top percentile of actions with entropy regularization. The entropy regularization causes the proposal policy to concentrate more slowly than the actor policy. Hence, the principle of a proposal policy which learns slower than the actor policy is kept, but does not perfectly match the theory. Future research should more carefully study the theoretical and empirical implications of using an entropy regularized proposal policy in place of this parameter update.

Second, as previously mentioned, we do not generally in practice require that the proposal policy and actor policy be from the same family of dis-

[4]We say the CCEM actor *tracks* the expected CEM optimizer because $\boldsymbol{\theta}$ and $\boldsymbol{\phi}^{prop}$ are changing with time.

tributions even though the theory assumes so. Future research should more carefully characterize the theoretical implications of this choice.

Finally, the theory assumes that the state distribution is fixed. Such an assumption is necessarily unfulfilled. The actor policy π_ϕ affects which states are encountered. Nevertheless, this assumption is a standard first-step assumption when analyzing off-policy algorithms (Jaakkola et al., 1994) and allows us to simply ask if the algorithm will eventually recover greedy actions across states. Future research should study the theoretical implications of having a changing state distribution when using the CCEM algorithm.

4.3 The CCEM Guarantees Policy Improvement in an Idealized Setting

In this section, we prove that the CCEM provides guaranteed policy improvement in an idealized setting, when the algorithm has access to the true action-value function and performs a complete optimization on each step. Analyzing the properties of policy improvement operators in an idealized setting is common (Chan et al., 2021; Ghosh et al., 2020; Haarnoja et al., 2018, 2019) and was even used in the original policy improvement theorem (Sutton et al., 2018). In fact, theoretical policy improvement is a desirable characteristic; it shows that in the best case scenario our algorithm will converge to an optimal policy. Without such theoretical guarantees, the utility of a policy improvement operator may be in question. The above discussion of the CCEM assumed an action-value critic is learned. Here, we assume that the true action-value function q_π is known.

As discussed in Chapter 2, many actor-critic algorithms reduce the distance between the learned policy and the Boltzmann policy. A reasonable question is whether the CCEM also uses distribution matching, and if so, which target policy the CCEM matches the learned policy to. The CCEM does perform distribution matching, as we now discuss.

Let $\mathscr{T}_\rho(s)\ \forall s \in \mathcal{S}$ be defined such that the following holds:

$$\int\limits_{\{a\in\mathscr{A} \mid q_\pi(s,a)\geq \mathscr{T}_\rho(s)\}} \pi(a \mid s)\ da = \rho \tag{4.4}$$

that is, $\mathscr{T}_\rho(s)$ is the $(1-\rho)$-percentile of action-values. Then, the *percentile-greedy policy* of π is defined as:

$$\varrho_\rho^\pi(a \mid s) = \begin{cases} \frac{\pi(a|s)}{\rho} & \text{if } q_\pi(s,a) \geq \mathscr{T}_\rho(s) \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

This percentile-greedy policy represents the target of our CCEM update, which matches the actor policy to $\varrho_\rho^{\pi_\phi}$ by minimizing $\mathbb{KL}\left(\varrho_\rho^{\pi_\phi} \mid\mid \pi_\phi\right)$ The percentile-greedy target policy is approximated using an empirical percentile distribution.

Intuitively, the percentile-greedy policy should be an improvement over the learned actor policy π_ϕ since it redistributes probability density from low-valued actions to high-valued actions proportional to the density of high-valued actions under π_ϕ. We now formally show that the percentile-greedy policy is indeed an improvement over the actor policy.

Theorem 4.3.1 *For a given policy π, action-value function q_π, and $\rho \in (0,1)$, the percentile-greedy policy ϱ_ρ^π of π is guaranteed to be at least as good as π in all states:*

$$v_{\varrho_\rho^\pi}(s) = \int\limits_{\mathscr{A}} \varrho_\rho^\pi(a \mid s) q_{\varrho_\rho^\pi}(s,a)da \geq \int\limits_{\mathscr{A}} \pi(a \mid s) q_\pi(s,a)da = v_\pi(s)$$

Proof. The proof follows a simple modification of the standard policy improvement theorem. Notice that by the definition of percentiles, for any state $s \in \mathcal{S}$ we have:

$$\begin{aligned} \int\limits_{\mathscr{A}} \varrho_\rho^\pi(a \mid s) q_\pi(s,a)da &= \int\limits_{\{a\in\mathscr{A} \mid q_\pi(s,a)\geq \mathscr{T}_\rho\}} \frac{\pi(a \mid s)}{\rho} q_\pi(s,a)da \\ &= \mathbb{E}_{\varrho_\rho^\pi}[q_\pi(s,A)] \\ &\geq \int\limits_{\mathscr{A}} \pi(a \mid s) q_\pi(s,a)da \\ &= \mathbb{E}_\pi[q_\pi(s,A)] \end{aligned}$$

Using these properties, we can derive a policy improvement guarantee. For any state $s \in \mathcal{S}$:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[q_\pi(s, A)] \leq \mathbb{E}_{\varrho_\rho^\pi}[q_\pi(s, A)] \\
&= \mathbb{E}_{\varrho_\rho^\pi}[R_{t+1} + \gamma \mathbb{E}_\pi[q_\pi(S_{t+1}, A_{t+1})] \mid S_t = s] \\
&\leq \mathbb{E}_{\varrho_\rho^\pi}[R_{t+1} + \gamma \mathbb{E}_{\varrho_\rho^\pi}[q_\pi(S_{t+1}, A_{t+1})] \mid S_t = s] \\
&\leq \mathbb{E}_{\varrho_\rho^\pi}[R_{t+1} + \gamma R_{t+2} + \gamma^2 \mathbb{E}_\pi[q_\pi(S_{t+2}, A_{t+2})] \mid S_t = s] \\
&\leq \dots \\
&\leq \mathbb{E}_{\varrho_\rho^\pi}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-1} R_T \mid S_t = s] \\
&= \mathbb{E}_{\varrho_\rho^\pi}[q_{\varrho_\rho^\pi}(s, A)] = v_{\varrho_\rho^\pi}(s)
\end{aligned}$$

Q.E.D.

Therefore, if we perform a full optimization to reduce $\mathbb{KL}(\varrho_\rho^{\pi_\phi} \mid\mid \pi_\phi) = 0$ and if we have access to the true action-value function q_{π_ϕ}, then the newly learned policy is guaranteed to be at least as good as the previous policy:

$$v_{\pi_{\phi_{t-1}}}(s) \leq v_{\pi_{\phi_t}}(s) \quad \forall s \in \mathcal{S}$$

and an iterative application of the CCEM will eventually recover an optimal policy in a finite number of steps. Of course in practice, we do not have access to the true action-value function, nor can we completely minimize the distance between $\varrho_\rho^{\pi_\phi}$ and π_ϕ.

This result is a sanity check to ensure that our algorithm utilizes a sensible target policy. Although this policy improvement requirement seems simple, it is often not satisfied in practice since the Boltzmann policy does not induce policy improvement with respect to the state and action values. Instead, the Boltzmann policy provides guaranteed policy improvement in terms of the soft state and action values, meaning that methods based on the Boltzmann distribution may uncover an optimal policy for the entropy-regularized version of the original MDP. Theorem 4.3.1 holds equally both for the state and action values as well as for the soft state and soft action values, meaning that the percentile-greedy policy can provide guaranteed policy improvement for an MDP or its entropy-regularized counterpart.

If this percentile-greedy policy is so great, why do we use a parameterized policy instead of directly using the percentile-greedy policy? Similarly to the case of the Boltzmann policy, computing ϱ_{ρ}^{π} would be onerous in the continuous action setting and so we cannot simply use it. Instead, we use a parameterized policy to mimic the percentile-greedy policy. In the discrete-action setting, the percentile-greedy policy is the greedy policy with respect to the true action values q_{π}. Indeed, it would be desirable to utilize such a policy, but unfortunately in practice q_{π} is unknown and we use an action-value critic, an inaccurate approximation to q_{π}. Using a greedy policy with respect to the critic can degrade performance by reducing exploration. Slowly matching a parameterized policy to the greedy policy with respect to the critic can ensure enough stochasticity is kept in the policy for adequate exploration.

4.4 A Full Reinforcement Learning Algorithm: Greedy Actor-Critic

The CCEM itself is not a full reinforcement learning algorithm but rather an algorithm for policy improvement. In this section, we discuss a new actor-critic algorithm which utilizes the CCEM for policy improvement and approximately satisfies the theoretical conditions required by the CCEM. We call this algorithm Greedy Actor-Critic (GreedyAC). Although this algorithm is agnostic to the specific function approximation scheme used and whether on-policy or off-policy data is used, we specifically discuss the case of using neural network function approximation and off-policy data generated from an experience replay buffer. The full GreedyAC algorithm is given in Algorithm 7.

Greedy Actor-Critic learns a single action-value critic $q_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta} \in \mathbb{R}^{|\boldsymbol{\theta}|}$ using a semi-gradient Sarsa update; a target critic network $q_{\boldsymbol{\theta}^{targ}}$ with parameters $\boldsymbol{\theta}^{targ} \in \mathbb{R}^{|\boldsymbol{\theta}|}$ provides the next action-value for the Sarsa update as is common practice with deep reinforcement learning algorithms (Algorithm 7, line 11). At each step, a batch B of states, actions, rewards, and next-states is sampled from an experience replay buffer $\mathscr{B}$ (Algorithm 7, line 9). For each tuple in the batch, denoted as (S, A, R, S'), an on-policy action, A', is sampled and the critic parameters are updated by minimizing the mean squared Bellman residual:

$$\frac{1}{2|B|} \sum_{(S,A,R,S') \in B} \left(R + \gamma q_{\boldsymbol{\theta}^{targ}}(S', A') - q_{\boldsymbol{\theta}}(S, A)\right)^2$$

resulting in the gradient:

$$g_{critic} \doteq -\frac{1}{|B|} \sum_{(S,A,R,S') \in B} (R + \gamma q_{\boldsymbol{\theta}^{targ}}(S', A') - q_{\boldsymbol{\theta}}(S, A)) \nabla_{\boldsymbol{\theta}} q_{\boldsymbol{\theta}}(S, A)$$

where $A' \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S')$ in both of the preceding equations. The parameters of the critic are updated using a single gradient descent step, and the parameters of the target network are updated using a polyak average between target network and critic weights (Algorithm 7, lines 10 to 13):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{critic} g_{critic} \tag{4.6}$$

$$\boldsymbol{\theta}^{targ} \leftarrow (1 - \beta)\boldsymbol{\theta}^{targ} + \beta\boldsymbol{\theta} \tag{4.7}$$

where $\alpha_{critic} \in \mathbb{R}$ is the step-size for learning the critic weights and $\beta \in (0, 1)$ controls the amount by which the target network parameters are updated toward the critic parameters.

To learn the actor policy $\pi_{\boldsymbol{\phi}}$ and proposal policy $\widetilde{\pi}_{\boldsymbol{\phi}^{prop}}$, Greedy Actor-Critic utilizes the CCEM algorithm. This involves (1) obtaining the sets $I^*(S)$ for each state S in the batch B, (2) estimating the entropy of the proposal policy, and (3) updating the actor and proposal policies with the respective gradients. To start, $N \in \mathbb{N}$ actions are sampled from the proposal policy $\widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(\cdot \mid S)$ for each state $S \in B$. Denote each of these sets $I(S)$ for each $S \in B$. To obtain the sets $I^*(S)$, the action-values of each action in $I(S)$ are calculated using $q_{\boldsymbol{\theta}}$. $I^*(S)$ is then comprised of the $\lfloor \rho N \rfloor$ actions of highest value. To estimate the entropy of the proposal policy in each state S, we uniformly randomly select an action $A_S^{\mathscr{H}}$ from $I(S)$, where the superscript $\mathscr{H}$ simply denotes that this action is used for entropy regularization, and the subscript S indexes state.

```
Algorithm 7: Greedy Actor-Critic
1  Input: $b, N \in \mathbb{N}$; $\alpha_{critic}$, $\alpha_{actor}$, $\alpha_{prop} \in \mathbb{R}^+$; $\beta \in (0,1]$; $\rho \in (0,1)$
2  if $\mathscr{A}$ is discrete and finite then
3      Initialize: parameters $\boldsymbol{\theta}, \boldsymbol{\theta}^{targ}, \boldsymbol{\phi}, \boldsymbol{\phi}^{prop}$, and replay buffer $\mathscr{B}$
4  else
5      Initialize: parameters $\boldsymbol{\theta}, \boldsymbol{\theta}^{targ}, \boldsymbol{\phi}$, and replay buffer $\mathscr{B}$
   end
6  Obtain: initial state $S_{current}$
   while $S_{current}$ not terminal do
7      Take action $A_{current} \sim \pi_{\boldsymbol{\phi}}(\cdot \mid S_{current})$ and observe $R_{next}$, $S_{next}$
8      Add $(S_{current}, A_{current}, R_{next}, S_{next})$ to replay buffer $\mathscr{B}$
9      Sample a random batch $B = (S, A, R, S')_{i=1}^{b} \sim \mathscr{B}$
10     Update parameters $\boldsymbol{\theta}$ using Sarsa with batch $B$:
11         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{critic} g_{critic}$
12     Update target parameters $\boldsymbol{\theta}^{targ}$:
13         $\boldsymbol{\theta}^{targ} \leftarrow (1-\beta)\boldsymbol{\theta}^{targ} + \beta\boldsymbol{\theta}$
14     Update parameters $\boldsymbol{\phi}$ using Algorithm 6 and $S \in B$
15     if $\mathscr{A}$ is continuous or countably infinite then
16         Update parameters $\boldsymbol{\phi}^{prop}$ using Algorithm 6 and $S \in B$
17     $S_{current} \leftarrow S_{next}$
   end
```

We then estimate the gradient of the entropy as[5]:

$$\nabla \mathscr{H}(\widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(\cdot \mid S)) \approx -\ln \widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(A_S^{\mathscr{H}} \mid S)\ \nabla_{\boldsymbol{\phi}^{prop}} \ln \widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(A_S^{\mathscr{H}} \mid S).$$

The following gradients are then calculated:

$$g_{actor} \doteq \frac{1}{|B|} \sum_{S \in B} \sum_{a \in I^*(S)} \nabla_{\boldsymbol{\phi}} \ln \pi_{\boldsymbol{\phi}}(a \mid S)$$

$$g_{prop} \doteq \frac{1}{|B|} \sum_{S \in B} \left(\sum_{a \in I^*(S)} \nabla_{\boldsymbol{\phi}^{prop}} \ln \widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(a \mid S) + \tau \nabla_{\boldsymbol{\phi}^{prop}} \mathscr{H}(\widetilde{\pi}_{\boldsymbol{\phi}^{prop}}(\cdot \mid S)) \right)$$

where $\tau \in \mathbb{R}$ controls the degree of entropy regularization to the proposal policy. The parameters of the actor and proposal policy are then updated by following a single gradient ascent step, resulting in a maximum-likelihood

[5]Alternatively, the analytic form of the gradient of the entropy of a distribution can be used when possible.

update on the actions in $I^*(S)$ for each $S \in B$ (Algorithm 7, lines 14 to 16):

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha_{actor} g_{actor} \tag{4.8}$$

$$\boldsymbol{\phi}^{prop} \leftarrow \boldsymbol{\phi}^{prop} + \alpha_{prop} g_{prop} \tag{4.9}$$

where $\alpha_{actor} \in \mathbb{R}$ and $\alpha_{prop} \in \mathbb{R}$ are the actor policy and proposal policy step-sizes. Although the general case of Greedy Actor-Critic allows for separate step-sizes for the actor and proposal policies, we use a single step-size for both, $\alpha_{actor} = \alpha_{prop}$, which reduces the number of hyperparameters we sweep in our experiments.

The above discussion focuses on the continuous-action version of GreedyAC. In the case of discrete-action environments, GreedyAC is simplified. Although we still use Algorithm 6 for the policy improvement step, GreedyAC no longer uses a proposal policy. Instead of repeatedly randomly sampling from a proposal policy to generate the empirical percentile distribution $I^*(S)$, we instead set $I^*(S) = \arg\max_{a \in \mathcal{A}} q_{\boldsymbol{\theta}}(S, a)$ in each state $S \in \mathcal{S}$.

4.5 Conclusion

In this section, we discussed the Conditional Cross-Entropy Optimization Method, a new policy improvement operator for actor-critic algorithms. We discussed the differences between the continuous action and discrete action versions of the CCEM as well as a number of theoretical results. Particularly, we discussed how the CCEM tracks the expected CEM optimizer. We also discussed how, given idealized conditions, the CCEM provides guaranteed policy improvement, unlike policy improvement operators based on the Boltzmann policy. Finally, we discussed Greedy Actor-Critic, a full actor-critic algorithm that utilizes the CCEM for policy improvement.

Chapter 5

Experiment Setup

This chapter describes the experimental setup used to empirically study the CCEM and GreedyAC. We first describe the environments that we run experiments on. Next, we discuss the hyperparameters swept, neural network architectures, policy parameterizations, and other relevant experimental details. We compare GreedyAC in a number of settings to two baseline algorithms: SAC-M and VanillaAC, which are described in detail in Chapter 2. For the remainder of this thesis, we will no longer work with the variant of SAC introduced by Haarnoja et al. (2018). Instead, we only work with the modern version, SAC-M, introduced by Haarnoja et al. (2019). For simplicity, we will refer to SAC-M simply as *SAC* for the rest of this thesis.

5.1 Environments

The empirical study is run on small, yet challenging environments which allow for extensive experimental repetition leading to both statistical significance of results and a careful exploration of hyperparameters and sensitivity. We study the performance and hyperparameter sensitivity of the aforementioned algorithms on three classic control environments: Mountain Car (Sutton et al., 2018), Pendulum (Degris et al., 2012a), and Acrobot (Sutton et al., 2018). We use both continuous and discrete action versions of all environments. The Mountain Car and Acrobot environments are episodic. The agent-environment interaction naturally breaks into separate episodes. The Pendulum environment is continuing; the agent-environment interaction goes on forever and does not naturally end. These classic control environments are a challenge for deep RL algorithms (Ghiassian et al., 2020), and performance differences on simple environments such as these have been shown to extend to more complex environments (Obando-Ceron et al., 2021).

In the Mountain Car environment, the agent controls a car at the bottom of a valley with a hill on each side. The goal is to drive the car up the hill on the right, but the car is underpowered and cannot directly drive up the hill. The agent must learn to rock the car back and forth, from one hill to the next, until it reaches to goal state at the top of the right-hand hill. State observations consist of the car's horizontal position and velocity in $[-1.2, 0.6] \times [-0.07, 0.07]$. The agent starts with a velocity of 0 in a position drawn randomly from $[-0.6, -0.4]$, which is near the bottom of the valley. Actions consist of the force to apply to the car and are bounded in $[-1.0, 1.0]$ for continuous actions. For discrete actions, the force is chosen from the set $\{-1, 0, 1\}$. The reward is -1 per step.

In the Pendulum environment, the agent controls a pendulum attached at a fixed base. The goal is to swing the pendulum and hold it in an upright position. Similarly to Mountain Car, the force applied to the pendulum is underpowered, and the pendulum cannot be directly swung into the upright position. State observations consist of the angle of the pendulum, normalized to be in $[-\pi, \pi)$, and the angular velocity of the pendulum in $[-1, 1]$. The environment starts with the pendulum facing straight downwards, an angle of $-\pi$, with angular velocity of 0. Actions consist of the torque applied to the pendulum at its fixed base and are bounded in $[-2.0, 2.0]$ for continuous actions. For discrete actions, the torque is chosen from the set $\{-2, 0, 2\}$. The reward is the cosine of the angle from the positive y-axis, where an angle of 0 indicates the pendulum coincides with the positive y-axis (i.e. the pendulum faces upwards).

In the Acrobot environment, the agent controls a double hinged pendulum attached at a fixed base. The goal is to swing the end of the second link above the fixed base by one link's length. State observations consist of the angle of each link, normalized in $[-\pi, \pi)$, and the angular velocity of each link, in $[-4\pi, 4\pi]$ for the first link and in $[-9\pi, 9\pi]$ for the second link. An angle of 0 indicates the respective link is facing straight downwards. The agent starts with random angles and angular velocities drawn uniformly randomly from the interval $[-0.1, 0.1]$. Actions consist of the torque to apply to the pendulum at its fixed base and are bounded in $[-1.0, 1.0]$ for continuous actions. For discrete actions, the torque is chosen from the set $\{-1, 0, 1\}$. The reward is -1 per step.

We used the following environment-specific settings in our experiments. All environments use a discount factor of $\gamma = 0.99$. Episodes are cut off at 1,000 timesteps. At episode cutoffs, the agent is transitioned to a random state in the domain of starting states. Episode cutoffs do not induce a termination to the agent-environment interaction. Each experiment is run for 100,000 timesteps, and the agent is updated at each step.

5.2 Experiment Details

For each algorithm, we swept and tuned hyperparameters. To begin, we swept all hyperparameter combinations over 10 initial runs with different random seeds for each run. Hyperparameters were tuned by selecting the hyperparameter setting which resulted in the agent achieving the highest score, for example the highest average episodic return over runs. Each experiment used a different definition of score depending on the goal of the experiment, and the exact definition of score is mentioned in the experimental descriptions in each subsequent chapter. After a hyperparameter setting was chosen, an additional 30 experiments were run with these chosen hyperparameters for different random seeds from the initial 10 runs.

We now describe the hyperparameters swept for all algorithms. Critic step-sizes were swept in the set $\alpha_{critic} \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. Actor step-sizes were set to be $\alpha_{actor} = \kappa \times \alpha_{critic}$, and κ was swept in the set

$\{10^{-3}, 10^{-2}, 10^{-1}, 1.0, 2.0, 10.0\}$. For experience replay, the buffer capacity was set at 100,000 samples so that all environmental transitions were stored in the buffer. A constant batch size of 32 was used for all algorithms. Target networks were updated at each timestep using a polyak average with fixed $\beta = 0.01$. Entropy scales were swept $\tau \in \{10^{-3}, 10^{-2}, 10^{-1}, 1.0, 10.0\}$. For Greedy Actor-Critic, we fixed $\rho = 0.1$ and sampled $N = 30$ actions per state. For Vanilla Actor-Critic, we also sampled a total of 30 actions where 1 was used to predict the action-value in the gradient, 1 was used to estimate the entropy, and 28 were used to estimate a state-value baseline using a stochastic sample of Equation 2.6.

The continuous and discrete action versions of environments require different policy parameterizations. For continuous action environments, we parameterized the learned policies as Gaussian distributions for GreedyAC and VanillaAC. For SAC, we used a squashed Gaussian policy following the original work (Haarnoja et al., 2018, 2019) since we found it performed better with a squashed Gaussian than with a Gaussian policy. For discrete actions, we parameterized the learned policies as Softmax distributions for all algorithms. All algorithm evaluation was performed during training and includes the stochastic noise in the learned policies. That is, actions are sampled from the learned policies, rather than selecting the modal actions.

For all algorithms, the actor and critic were parameterized using neural network function approximators. For the critic, we used three hidden layers of 64 hidden units each with ReLU activations. For the parameterized policies, we also used three hidden layers of 64 hidden units with ReLU activations. In the case of Gaussian or squashed Gaussian policies, the networks predicted the mean and log standard deviation of the policy. For Gaussian policies, the mean was computed by passing the output of the last hidden layer through a hyperbolic tangent nonlinearity and then rescaling to ensure the mean was within the action bounds of the environment. This was not required for the squashed Gaussian policy used by SAC, since this policy bounds actions to be within the action range of the environment. In the case of Softmax policies, the networks predicted one logit for each action.

Chapter 6

Experiments with Hyperparameter Tuning

In this chapter, we discuss our first experiments. In these experiments, we run GreedyAC, VanillaAC, and SAC on the three classic on control environments of Mountain Car, Pendulum, and Acrobot with both discrete and continuous actions. In this chapter, we refer to environment $\mathscr{E}$ using continuous actions as $\mathscr{E}$-CA and using discrete actions as $\mathscr{E}$-DA.

These experiments consider the performance of all three algorithms under two different hyperparameter tuning regimes. The first of which is the per-environment hyperparameter tuning regime. This regime reflects the best-case performance of an algorithm, which is when the algorithm can be intricately tuned to a specific problem environment. Typically, this regime is available only in simulation, but is a common method to analyze actor-critic algorithms. The second regime we consider is one in which hyperparameters are tuned across environments. This regime reflects how sensitive an algorithm is to its hyperparameters and whether the algorithm is able to find a single hyperparameter setting that works well in an assortment of problems. An algorithm that is insensitive to hyperparameters should see little degradation in performance when moving from the per-environment regime to the across-environment regime. Ideally, we would like our algorithms to have equal performance in both regimes, indicating that the algorithm performs well and is insensitive to hyperparameters across different environments.

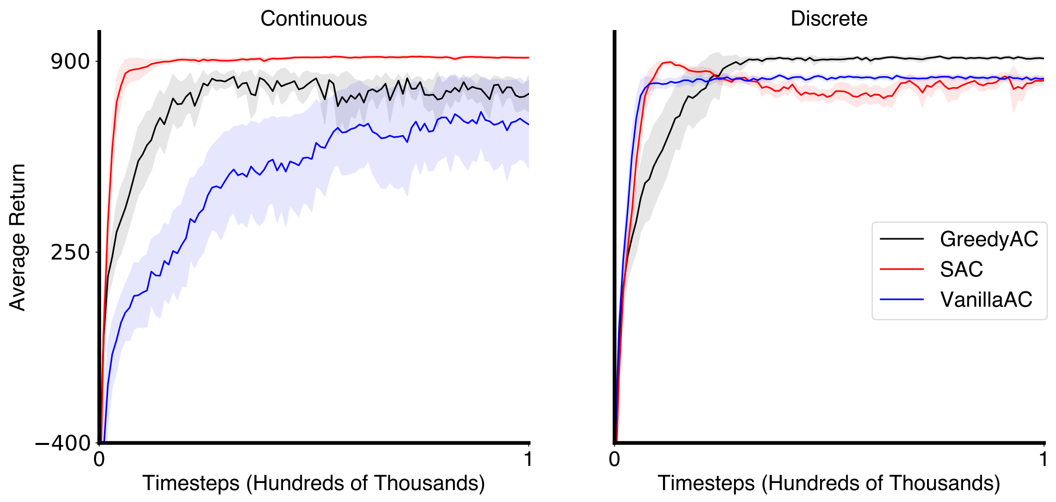

Figure 6.1: Learning curves on **Pendulum** when tuning **per-environment**. Learning curves are averaged over 30 runs with shaded regions denoting standard error from the mean performance. On Pendulum-CA, SAC learned faster than both GreedyAC and VanillaAC. On Pendulum-DA, GreedyAC learned slowest, but exhibited highest final performance. Hyperparameters were tuned over an initial 10 runs with different random seeds from the results shown here.

6.1 Per-Environment Hyperparameter Tuning

Our first experiment examines how well each algorithm can perform if tuned to a single problem – on a per-environment basis. Each algorithm's hyperparameters were tuned on each of the six environments by selecting the hyperparameter setting inducing the highest performance, calculated as the average episodic return per run over 10 initial runs. We tuned for discrete and continuous action environments separately. The chosen hyperparameter settings are listed in Appendix A. After the hyperparameters were tuned, an additional 30 runs were conducted on each environment.

For each of 30 runs with tuned hyperparameters, the episodic return was recorded and averaged over runs to produce the mean learning curves with standard error in Figures 6.1, 6.2, and 6.3 for Pendulum, Mountain Car, and Acrobot respectively. For the continuing environment Pendulum, learning curves were constructed by averaging each episode's performance over runs since each run had exactly 100 episodes completed. For the episodic environ-

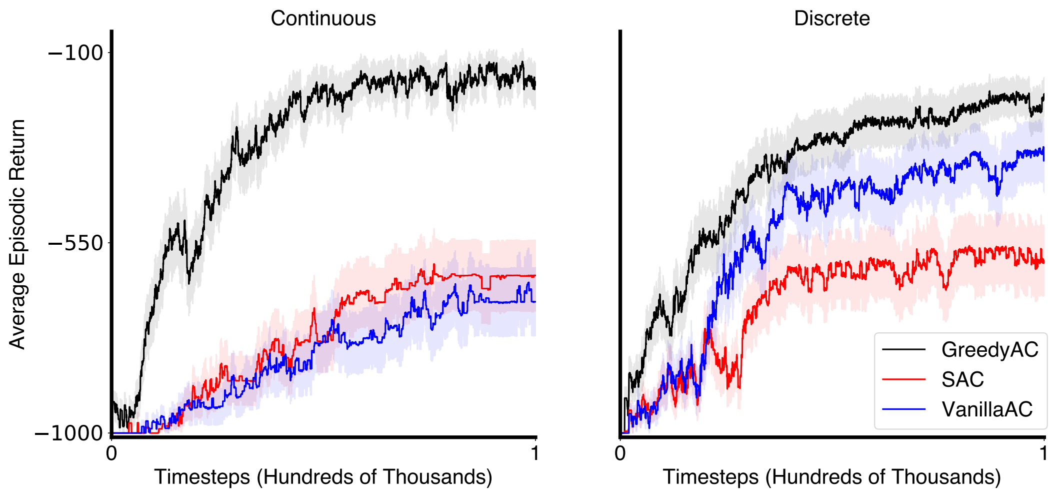

Figure 6.2: Learning curves on **Mountain Car** when tuning **per-environment**. Learning curves are averaged over 30 runs with shaded regions denoting standard error from the mean performance. GreedyAC was able to learn quickly on Mountain Car. GreedyAC learned faster than both baselines on Mountain Car-CA. On Mountain Car-DA, GreedyAC matched the performance of VanillaAC in terms of learning speed and final episodic return. Compared to GreedyAC and VanillaAC, SAC learned slowest and attained lowest final performance on Mountain Car-DA. Hyperparameters were tuned over an initial 10 runs with different random seeds from the results shown here.

ments, each run had a different number of episodes completed in the budget 100,000 timesteps. To average over runs, we repeated each episode's return n times, where n is the number of completed timesteps in each episode, resulting in 100,000 readings of performance for each run. These readings were then averaged over runs to produce the learning curves.

On Pendulum, all algorithms performed well, as depicted in Figure 6.1, indicating that this environment is perhaps easy for these algorithms to solve. On Pendulum-DA, VanillaAC and SAC learned faster than GreedyAC, yet exhibited lower final performance than that of GreedyAC. Upon convergence, all algorithms exhibited stable performance. On Pendulum-CA, SAC attained the highest episodic return over time and exhibited nearly zero variance both across runs and across consecutive episodes. SAC was also able to attain high average episodic return faster than GreedyAC and VanillaAC. GreedyAC learned slower than SAC on Pendulum-CA and exhibited lower final perfor-

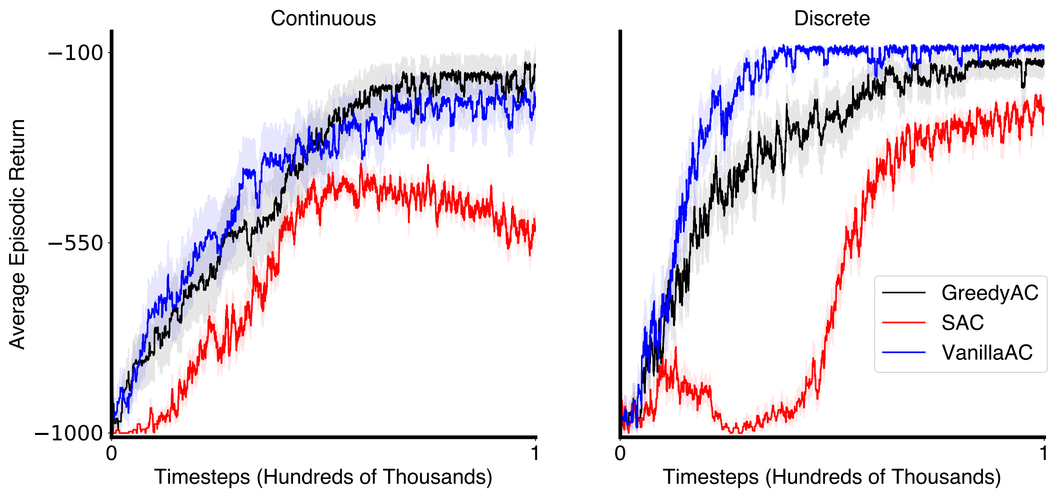

Figure 6.3: Learning curves on **Acrobot** when tuning **per-environment**. Learning curves are averaged over 30 runs with shaded regions denoting standard error from the mean performance. On Acrobot-CA, all algorithms begin learning at the same speed. VanillaAC and GreedyAC converge to a similar final performance, higher than the final performance of SAC. On Acrobot-DA, GreedyAC nearly matches the performance of VanillaAC. Hyperparameters were tuned over an initial 10 runs with different random seeds from the results shown here.

mance. VanillaAC learned the slowest and attained the lowest average episodic return over time.

Mountain Car proved to be somewhat more difficult for SAC, as shown in Figure 6.2. Overall, GreedyAC performed well on Mountain Car. Under the discrete-action setting, GreedyAC approximately matched the performance of the VanillaAC baseline, and both algorithms performed satisfactorily. SAC performed worse out of all three algorithms on Mountain Car-DA. On Mountain Car-CA, GreedyAC exhibited the highest final performance of all three algorithms and also learned the fastest. VanillaAC and SAC performed similarly on Mountain Car-CA, exhibiting poor performance overall.

Figure 6.3 outlines the performance of each algorithm on Acrobot. On Acrobot-DA, both VanillaAC and GreedyAC learned rapidly, yet VanillaAC converged to its final performance faster than GreedyAC did. SAC began the experiment learning at approximately the same speed as VanillaAC and GreedyAC. After a short learning interval, the performance of SAC degraded.

After this degradation, SAC began learning again and converged to a final performance similar to that exhibited by GreedyAC. On Acrobot-CA, all algorithms learned at approximately the same speed until about halfway through the experiment. At this point, GreedyAC and VanillaAC continued to learn, converging to a similar final performance. On the other hand, the performance of SAC began to degrade at this point. The final performance of SAC was lower than that of GreedyAC and VanillaAC. Although common knowledge is that actor-critic algorithms struggle to learn on the Acrobot environment, both VanillaAC and GreedyAC performed satisfactorily.

GreedyAC exhibits satisfactory performance on this classic control suite of environments. The algorithm generally exhibits higher performance than SAC, which is widely regarded as state-of-the-art. Furthermore, GreedyAC performs similarly to the VanillaAC baseline in many cases. We note that in the discrete action setting, GreedyAC does not utilize entropy regularization. This can be seen as an advantage or disadvantage for GreedyAC. On the one hand, this may make the algorithm easier to tune than the baseline algorithms. On the other hand, the baseline algorithms were given 5 times more hyperparameter settings to tune over than GreedyAC was given, perhaps giving the baselines an advantage. Overall on this small classic control suite, GreedyAC can outperform both baseline algorithms, Whether this is a general characteristic of these algorithms under the per-environment tuning regime is unclear and is a topic for future research.

6.2 Across-Environment Hyperparameter Tuning

In this section, we discuss our second experiment. This goal of this experiment is to see whether or not Greedy Actor-Critic can find a single hyperparameter setting that works well across all six environments in the classic control suite. Actor-Critic algorithms are known to be sensitive to hyperparameters (Chan et al., 2021; Degris et al., 2012a; Haarnoja et al., 2018; Pourchot et al., 2019).

Such sensitivity demonstrates the difficulty of applying these algorithms to new problems and is relevant for both pure and applied settings. Often, finding a hyperparameters setting that works well for a single problem is computationally expensive. Such a hyperparameter setting is often appropriate only for the original problem for which it was found. If this hyperparameter setting were applied on a new problem, the algorithm would most likely perform poorly.

One may wonder why hyperparameter sensitivity is problematic at all when parallel computing exists that allows extensive hyperparameter sweeps in a short amount of time using simulators. The difficulty is that although this extensive experimentation is possible when using simulators, it is not generally possible in the real world, limiting the applicability of actor-critic algorithms to industry. This is what we focus on in this section: can Greedy Actor-Critic find a single hyperparameter setting which works well across the environments in the classic control suite?

6.2.1 Hyperparameter Selection

In order to tune an algorithm across all six environments, we must take the performance on each environment into account. Because each environment has a different reward scheme from the others, we used a normalization approach to ensure the performance on each environment contributed approximately equally to the hyperparameter selection process.

For each environment, we normalized performance by first finding the highest episodic return achieved over all algorithms, for all hyperparameter settings, and over all of the 10 initial runs. Denote this value as $G^*(\mathscr{E})$ for environment $\mathscr{E}$. We used $G^*(\mathscr{E})$ as an approximation to the highest attainable return on environment $\mathscr{E}$. Table 6.1 lists these approximately-optimal returns for each environment for both continuous and discrete action settings.

After the approximately optimal return was found, we normalized performance as:

$$N(p(\mathscr{E})) \doteq 1 - \frac{G^*(\mathscr{E}) - p(\mathscr{E})}{|G^*(\mathscr{E})|}$$

where $p(\mathscr{E})$ is the performance on environment $\mathscr{E}$ to normalize. In this ex-

Environment	Continuous	Discrete
Acrobot	-56	-56
Mountain Car	-65	-83
Pendulum	930	932

Table 6.1: Approximately optimal returns on each environment for continuous and discrete action settings. The optimal return is the highest return possible on each environment and is approximated by the highest return achieved over all algorithms, hyperparameter settings, and runs.

periment, the performance $p(\mathscr{E})$ was the average episodic return. Each hyperparameter setting was then assigned a score equal to the average normalized performance over all environments, episodes, and runs. The hyperparameter setting which achieved the highest score was then selected for each algorithm. These settings are listed in Appendix A. After the hyperparameters were tuned, an additional 30 runs were conducted for each algorithm using different random seeds from the initial 10 runs.

6.2.2 Results and Discussion

Considering how the performance should change in comparison to the previous section is useful. Because each algorithm is forced to select a single hyperparameter setting across all environments, we would expect to see some degradation in performance when compared to the results of the preceding section which focused on finding hyperparameter settings on an per-environment basis. Algorithms that are sensitive to their hyperparameter settings should experience a larger degradation in performance compared to those which are more robust to changes in their hyperparameter settings. Figures 6.4, 6.5, and 6.6 show the learning curves for each algorithm on Pendulum, Mountain Car, and Acrobot respectively. These figures were constructed in the same manner discussed in Section 6.1.

Figure 6.4 shows the performance of each algorithm on Pendulum. On Pendulum-CA, all algorithms performed poorly. GreedyAC and VanillaAC each learned faster than SAC did. On the other hand, GreedyAC achieved the highest episodic return over time on Pendulum-DA and, upon convergence,

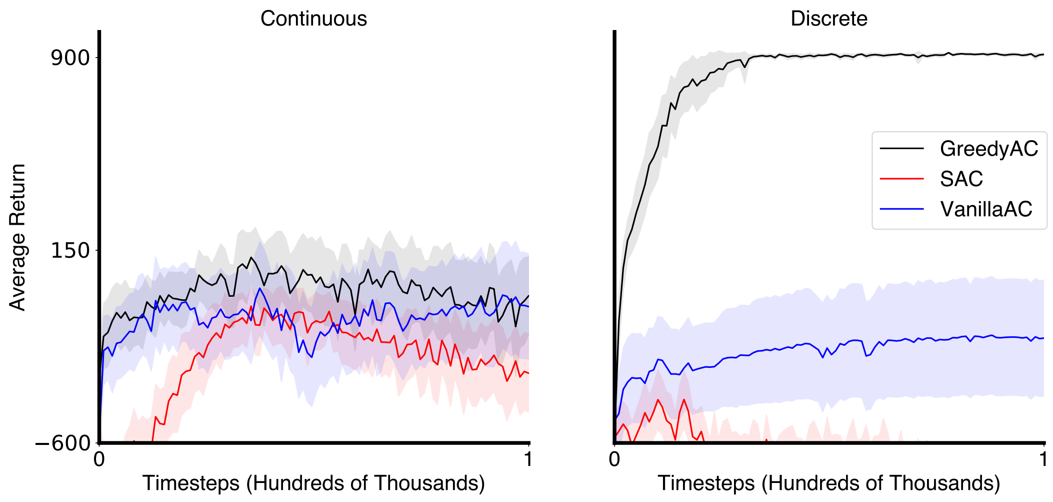

Figure 6.4: Learning curves on **Pendulum** when tuning **across-environments**. On Pendulum-CA, all algorithms exhibited low performance, with SAC learning slowest. On Pendulum-DA, GreedyAC learned fastest and exhibited highest final performance. Upon convergence, GreedyAC also exhibited low variation in performance across runs and on consecutive episodes. Hyperparameters were tuned over an initial 10 runs with different random seeds from the results shown here.

evinced stable performance both across runs and on consecutive episodes. On the other hand, SAC performed worse on Pendulum-DA, than on Pendulum-CA. VanillaAC achieved similar episodic return and stability in performance on both the discrete and continuous action versions of Pendulum.

On both continuous and discrete action versions of Mountain Car, SAC achieved the lowest performance, shown in Figure 6.5. For nearly all episodes, SAC achieved the lowest average episodic return possible for both the continuous and discrete action versions of Mountain Car. VanillaAC exhibited higher performance on Mountain Car-CA than it did on Mountain Car-DA. GreedyAC matched the performance of VanillaAC on Mountain Car-CA and outperformed both baseline algorithms on Mountain Car-DA.

The learning curves on Acrobot are shown in Figure 6.6. On Acrobot-DA, GreedyAC and VanillaAC began learning at the same speed until approximately halfway through the experiment. After this point, the learning speed of VanillaAC declined. SAC learned slowest but matched the final performance

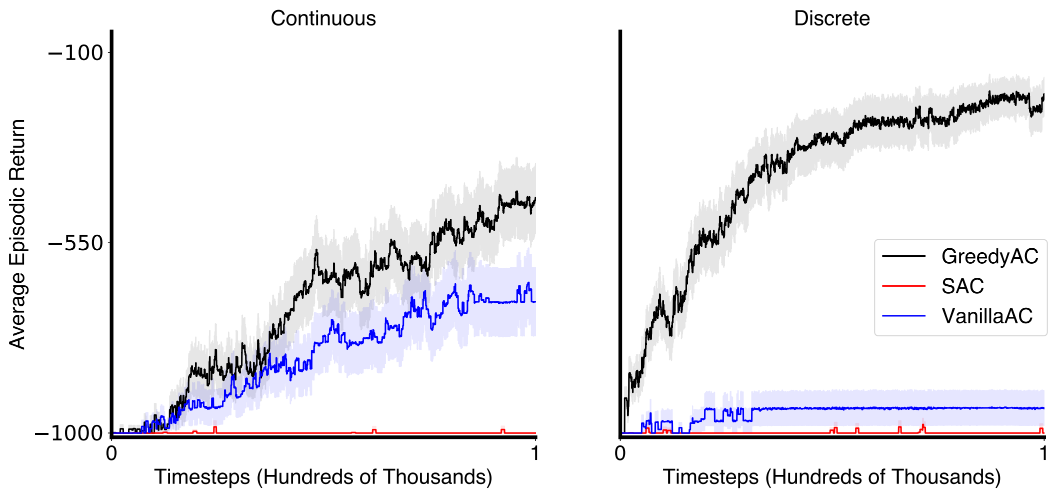

Figure 6.5: Learning curves on **Mountain Car** when tuning **across-environment**. Learning curves are averaged over 30 runs with shaded regions denoting standard error from the mean performance. GreedyAC learned faster on Mountain Car-DA than it did on Mountain Car-CA. SAC failed to learn on both environments, and VanillaAC did not learn on Mountain Car-DA. Hyperparameters were tuned over an initial 10 runs with different random seeds from the results shown here.

of GreedyAC. On Acrobot-CA, all algorithms learned at a similar pace until approximately halfway through learning, at which point the performance of SAC began to degrade. GreedyAC matched the final performance of VanillaAC on Acrobot-CA.

On this small classic control suite, GreedyAC seems less sensitive to its hyperparameters than either SAC or VanillaAC. Although the algorithm sometimes learns slower in this regime than under per-environment tuning, it performs satisfactorily. In the across-environment tuning regime here, VanillaAC and SAC each perform well in several of the environments, but are not able to match the performance of GreedyAC; GreedyAC performs at least as well as the baseline algorithms, and is able to outperform the baseline algorithms in a number of cases. Whether or not GreedyAC can find a single hyperparameter setting that works well across a general set of environments is unclear. Furthermore, whether GreedyAC can achieve higher performance than the baseline algorithms when tuning across a general set of environments is also unclear.

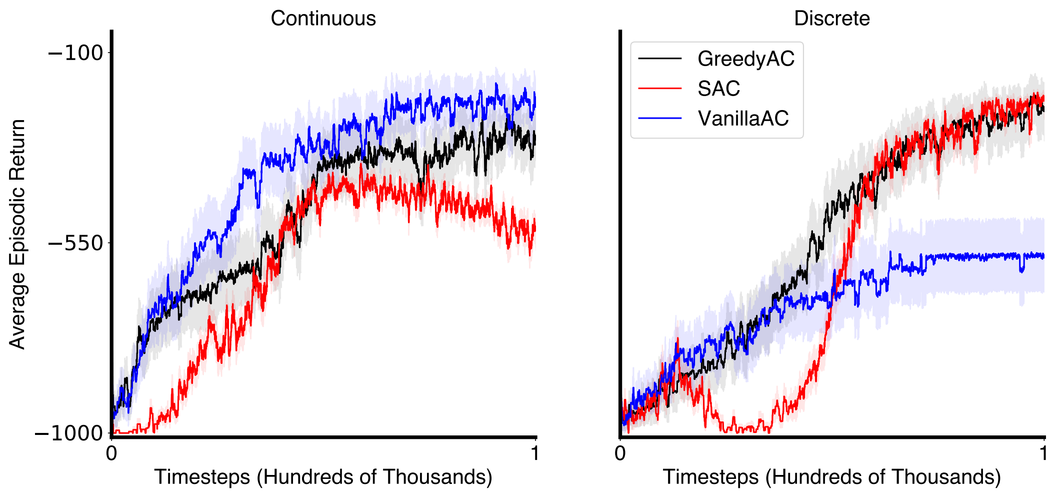

Figure 6.6: Learning curves on **Acrobot** when tuning **across-environment**. Learning curves are averaged over 30 runs with shaded regions denoting standard error from the mean performance. On Acrobot-CA, all algorithms began the experiment learning at approximately the same speed, and GreedyAC matched the final performance of VanillaAC and outperformed SAC. On Acrobot-DA, GreedyAC matched the final performance of SAC. Hyperparameters were tuned over an initial 10 runs with different random seeds from the results shown here.

6.3 Conclusion

In this chapter, we discussed the performance of GreedyAC, VanillaAC, and SAC under two different hyperparameter tuning regimes. The per-environment regime is when hyperparameters are tuned to a specific environment. This regime demonstrates the approximate best-case performance of an algorithm. The across-environment regime is when hyperparameters are tuned across a set of environments. This regime demonstrates the sensitivity of an algorithm to its hyperparameters across environments. If the algorithm is robust to hyperparameters, then a single hyperparameter setting should exist for which the performance of the algorithm is high on each environment. An ideal case would be when the performance of the algorithm in this regime coincides with its performance in the per-environment regime, indicating little sensitivity to hyperparameters.

We saw that under per-environment tuning, both GreedyAC and VanillaAC performed well on all environments. In comparison, SAC performed best on Pendulum. When switching to the across-environment regime, we saw that GreedyAC was somewhat more robust to its hyperparameters than both baseline algorithms on this small classic control suite. Whether this is a general characteristic of the algorithm is unclear.

Chapter 7

Hyperparameter Sensitivity Analysis

The previous chapter provided an analysis of GreedyAC in terms of overall performance for a fixed hyperparameter setting, tuned on a per- or across-environment basis. The across-environment tuning experiments provided a small glimpse into the sensitivity of GreedyAC to its hyperparameters. A complete sensitivity analysis of the algorithm to specific hyperparameters was lacking in the previous chapter, as we only studied hyperparameters as a complete set.

In this chapter, we first provide a systematic analysis of the sensitivity of GreedyAC to a single hyperparameter that many actor-critic algorithms are sensitive to, the entropy scale (Chan et al., 2021; Haarnoja et al., 2018). Then, we discuss the sensitivity of GreedyAC to the number of action samples used in the CCEM update. Because GreedyAC does not use entropy regularization nor action sampling in the discrete action setting, this chapter is restricted to continuous action environments alone. Because of this, whenever we refer to an environment in this chapter, we drop the *-CA* suffix and the environment should be taken as the continuous action version.

7.1 Entropy Scale Sensitivity

In this section, we discuss the sensitivity of GreedyAC to the entropy scale hyperparameter and compare to the sensitivity of SAC. Since entropy regularization affects the proposal policy of GreedyAC, rather than its actor policy, we would expect GreedyAC to be somewhat less sensitive to the entropy scale hyperparameter than SAC is. For GreedyAC, the entropy scale affects the actions we reason about for the policy update, but only indirectly affects the actions taken in the environment.

Figure 7.1 depicts the sensitivity regions of GreedyAC and SAC to the entropy scale hyperparameter. The plot is generated by sweeping over all hyperparameters listed in Chapter 5 over 40 runs in the following manner. We first fixed the entropy scale to one of the values swept over, which is listed in Chapter 5. Then, we fixed the critic step-size to one of values swept over, also listed in Chapter 5. We then tuned over all other hyperparameters to find the setting for the fixed entropy scale and critic step-size that resulted in the highest average episodic return over all runs. This value was then plotted on the y-axis for the fixed entropy scale and critic step-size. Once this had been done for each entropy scale and critic step-size, we were left with a set of lines. Each line denoted the sensitivity of the algorithm to the critic step-size at a set entropy scale. The region between all these lines was then filled in, with the critic step-size being plotted on a logarithmic scale.

The width of the resulting shape outlines the variability in performance across all entropy scales for a given critic step-size. If the plot is narrow, this indicates low sensitivity to the entropy scale hyperparameter and that the step-size, rather than the entropy scale, was the major factor in performance variability. On the other hand, a wide shape indicates high sensitivity to the entropy scale. If the plot has a sharp U-shape, then this indicates sensitivity to the critic step-size. If the plot is nearly horizontal, this indicates low sensitivity to the critic step-size. Narrow bands near the top of the plot indicate desirable performance – achieving high episodic return with low sensitivity to the entropy scale. Undesirable performance is characterized by wide bands, especially near the bottom of the y-axis. Optimal performance would be characterized by a single horizontal line at the top of each plot, indicating no sensitivity to either hyperparameter and optimal episodic returns achieved.

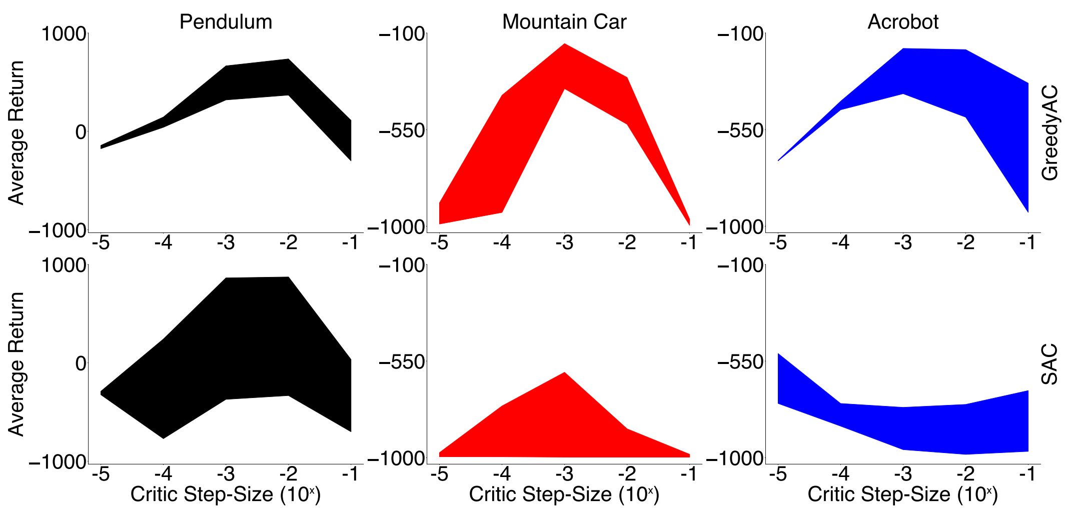

Figure 7.1: A **sensitivity region** plot for the entropy scale hyperparameter for Greedy Actor-Critic (top) and Soft Actor-Critic (bottom) in the continuous action setting over 40 runs. At a given critic step-size, the width of the shape indicates the range of performance across all entropy scales. Desirable performance is characterized by a narrow band near the top of the y-axis. GreedyAC generally exhibits less sensitivity to the entropy scale than SAC does.

Overall, GreedyAC seems to be less sensitive to the entropy scale than SAC is on the classic control suite of environments. On Pendulum, the sensitivity regions of GreedyAC are narrower than the sensitivity regions of SAC. For many critic step-sizes on Mountain Car, GreedyAC also exhibits narrower sensitivity regions than SAC exhibits. On Acrobot, the sensitivity regions of GreedyAC are widest of all environments and are sometimes wider than those of SAC. Although SAC exhibits narrower sensitivity regions than GreedyAC under some critic step-sizes on both Mountain Car and Acrobot, these regions are generally near the bottom of the plot, indicating poor performance.

From Figure 7.1, one can see that overall, SAC exhibited poor sensitivity to the entropy scale. Its sensitivity regions are wide and average return generally low. On Pendulum, SAC achieved near-optimal performance, as can be seen at critic step-sizes of 10^{-3} and 10^{-2}; yet at these step-sizes, SAC also exhibited high sensitivity to the entropy scale. On both Mountain Car and Acrobot, SAC achieved nearly the lowest possible return for many critic step-sizes and entropy scales, causing its sensitivity to appear low. Likely, this is an optimistic estimate of sensitivity.

Greedy Actor-Critic generally exhibited narrow sensitivity regions, indicating that the critic step-size was the dominant factor to affect performance rather than the entropy scale on these environments. Overall, GreedyAC is somewhat less sensitive to the entropy scale than SAC is on this small classic control suite. Whether this is a general characteristic of the algorithm is uncertain.

7.2 Sample Size Sensitivity

Greedy Actor-Critic, and the CCEM in general, introduces two new hyperparameters: the number of samples to consider for each policy update, $N \in \mathbb{N}$, and the proportion of these samples which are effectively used in the gradient update, $\rho \in (0, 1)$. The time complexity of the CCEM linearly depends on N. At each policy update, N actions must be sampled and sorted by value. This part of the CCEM algorithm can be the longest, in terms of wall-clock time, given large enough N. A reasonable question is whether N can be reduced without negatively impacting performance.

To examine how the number of sampled actions N affects performance, we ran GreedyAC on the continuous action versions of Mountain Car, Pendulum, and Acrobot for 10 runs. All hyperparameters were swept from the same sets as mentioned in Chapter 5 except that a critic step-size of 10^{-5} was not swept nor was an actor step-size scale of 10. The hyperparameter N was swept in $\{10, 20, 30\}$. For each value of N, we set $\rho = \frac{3}{N}$ such that the algorithm always updates with the 3 actions of highest value. We tune hyperparameters on a per-environment basis.

Before considering the results, we can consider what to expect. In the ideal case, when we have access to the true action values, we would ideally like to sample every action. This would provide us with perfect information about the optimization landscape, but unfortunately is impossible. In reality,

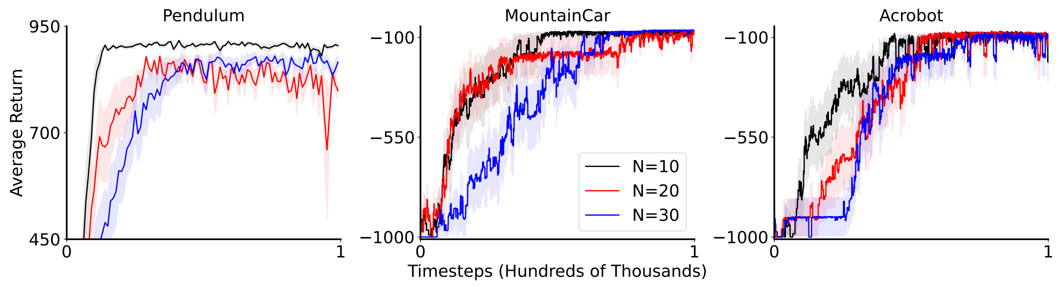

Figure 7.2: Mean learning curves with standard error of Greedy Actor-Critic over 10 runs. Each curve denotes the performance of Greedy Actor-Critic using a different value for the hyperparameter N, the number of actions sampled for the CCEM update. For each curve, ρ is set to be $\frac{3}{N}$ such that each CCEM update increases the density under the actor and proposal policies of the 3 actions of highest value from the set of actions sampled.

we optimize an action-value critic, which may be an inaccurate approximation to the true action values. To avoid rapid convergence to an inaccurate critic optimum, we may desire to limit the number of action samples. Rather than quickly converge to an optimum of the critic, we want to slowly track the critic's increasingly more accurate optima. We must therefore find a number of action samples to balance these two phenomena.

Figure 7.2 shows the mean learning curves over 10 runs, constructed in the same way as described in Section 6.1. On each environment, lowering the value of N did not significantly degrade performance, and in some cases it increased the speed of learning. This is most noticeable on Pendulum, where using $N = 10$ resulted in rapid early learning, high final performance, and stable performance on consecutive episodes.

We now return to the original question. Can we reduce the number of action samples used by the CCEM update? In some situations, lowering the value of this hyperparameter seems acceptable, if not desirable. By sampling fewer actions, the learned policy may be kept more exploratory. Varying the value of N between 10 and 30 (while keeping ρN fixed) does not seem to significantly impact the performance of GreedyAC on the classic control suite. Whether this is a general characteristic of the CCEM is unclear.

7.3 Conclusion

In this section, we studied the sensitivity of Greedy Actor-Critic to its hyperparameters on three classic control problems with continuous actions. In general, GreedyAC seems somewhat robust to its entropy scale hyperparameter. In particular, GreedyAC exhibits less sensitivity to the entropy scale than SAC does on the classic control suite. GreedyAC also seems to be somewhat insensitive to N, the number of action samples used in the CCEM update, on the classic control suite given that ρN is fixed.

Chapter 8

Conclusion and Future Work

This thesis has introduced both a new actor-critic algorithm, Greedy Actor-Critic, and a new policy improvement operator specific to actor-critic algorithms. Many current actor-critic algorithms use an improvement operator based on the Boltzmann policy which guarantees policy improvement; each consecutive policy learned is guaranteed to be an improvement over the previous policy under ideal conditions.

Two potential issues exist with using the Boltzmann policy in this fashion. The policy improvement guarantee is with respect to a different problem, an entropy regularized one. Therefore, algorithms which use the Boltzmann target policy in the policy improvement step may not find an optimal policy of the original problem. Furthermore, methods based on the Boltzmann policy are typically sensitive to the entropy scale hyperparameter (Chan et al., 2021; Haarnoja et al., 2018, 2019), as we have empirically seen in previous chapters.

The Conditional Cross-Entropy Method (CCEM) is a policy improvement algorithm based on the Cross-Entropy Method for optimization (CEM) and attempts to address these two problems by decoupling entropy regularization from the actor policy. The CCEM algorithm is both simple and intuitive; it works by keeping a separate, entropy regularized proposal policy which selects the actions to reason about. Given this set of actions, the CCEM then performs a maximum likelihood update on the top percentile of actions, ordered according to action-values produced by a critic. In this way, the CCEM uses an iterative, multi-level approach which slowly settles the learned policy on an optimal one.

Unlike the Boltzmann policy, the CCEM provides guaranteed policy improvement on whichever problem it is used in. If the MDP is entropy regularized, then each consecutively learned policy will be a guaranteed improvement in terms of the soft action-values. If the MDP is not entropy regularized, this guarantee is with respect to the action values. In this way, the CCEM will theoretically recover an optimal policy in a finite number of updates, given ideal conditions. The Boltzmann policy does not provide such guarantees.

Whereas the CEM attempts to find an optimum of a single function, the CCEM attempts to find an optimum of an action-value critic, which can be problematic. Since this critic is both an inaccurate approximation of the true function we wish to maximize (the action values of the current policy) and a function which changes with time, the theoretical results of the CEM do not immediately apply to the CCEM. Theoretical results demonstrate how the CCEM tracks the expected solution of the CEM across states, given certain assumptions. These theoretical results outline how the updates performed by the CCEM are reasonable and asymptotically unaffected by the stochastic noise in the update.

Greedy Actor-Critic (GreedyAC) is an actor-critic algorithm which uses the CCEM for policy improvement. We compared Greedy Actor-Critic to two baselines: Soft Actor-Critic (SAC) and Vanilla Actor-Critic (VanillaAC). We saw how GreedyAC was able to match the best-case performance of both baselines algorithms on a small suite of classic control environments. We also demonstrated that a single hyperparameter setting exists for GreedyAC which induces satisfactory performance on nearly every environment in the suite. The baseline algorithms were not able to perform as well as GreedyAC when forced to choose a single hyperparameter setting. This indicates that Greedy Actor-Critic may be less sensitive to its hyperparameters than the two baseline algorithms.

We then studied the sensitivity of GreedyAC to its hyperparameters. In particular, we saw that GreedyAC is somewhat less sensitive to the entropy scale hyperparameter than SAC is on the classic control suite. Such a result demonstrates the utility of isolating entropy regularization to a separate policy, not involved with action selection. We also studied the effects of hyperparameters specific to the CCEM on GreedyAC, in particular the number of action samples for the CCEM update. We saw that the algorithm may be insensitive to this hyperparameter on the classic control suite.

We hope that future research will take the foundation of the CCEM as introduced in this thesis and build upon it to further increase the effectiveness of this policy improvement operator. In particular, we hope that future research will consider theoretical characterizations of the CCEM algorithm actually used in practice. The theory we have presented is a first-step in this direction, but requires assumptions which do not hold in practice. We also encourage future research to explore different distribution matching techniques for the CCEM. The CCEM minimizes a forward KL divergence between the percentile-greedy policy and the learned policy. Because the forward KL is well-known to be mean seeking, this procedure could be problematic in environments with multiple global optima. Implementing a CCEM algorithm that reduces a reverse KL divergence could circumvent this. Finally, a promising direction to explore is the construction of the empirical percentile distribution for the CCEM update. The CCEM, as we have presented it, uses a repeated random sampling procedure to construct this set of actions, but future research should consider more effective ways of constructing this set. Since the quality of actions in this set directly affects the CCEM update, better constructing this set could improve the CCEM as a whole.

References

Abdolmaleki, A., J. T. Springenberg, Y. Tassa, R. Munos, N. Heess, and M. Riedmiller (2018). “Maximum a Posteriori Policy Optimisation.” In: *Proceedings of the 6th International Conference on Learning Representations*.

Ahmed, Z., N. Le Roux, M. Norouzi, and D. Schuurmans (2019). “Understanding the Impact of Entropy on Policy Optimization.” In: *International Conference on Machine Learning*.

Barto, A. G., R. S. Sutton, and C. W. Anderson (1983). “Neuronlike Adaptive Elements that can Solve Difficult Learning Control Problems.” In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5, pp. 834–846.

Boer, P., D. Kroese, S. Mannor, and R. Y. Rubinstein (2005). “A Tutorial on the Cross-Entropy Method.” In: *Annals of Operations Research* 134, pp. 19–67.

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba (2016). “OpenAI Gym.” In: *arXiv:1606.01540*.

Chan, A., H. Silva, S. Lim, T. Kozuno, A. R. Mahmood, and M. White (2021). “Greedification Operators for Policy Optimization: Investigating Forward and Reverse KL Divergences.” In: *arXiv:2107.08285*.

Ciosek, K., Q. Vuong, R. Loftin, and K. Hofmann (2019). “Better Exploration with Optimistic Actor-Critic.” In: *Advances in Neural Information Processing Systems*.

Ciosek, K. and S. Whiteson (2018). “Expected Policy Gradients.” In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

— (2020). “Expected Policy Gradients for Reinforcement Learning.” In: *Journal of Machine Learning Research* 21.52, pp. 1–51.

Clary, K., E. Tosch, J. Foley, and D. D. Jensen (2018). “Let’s Play Again: Variability of Deep Reinforcement Learning Agents in Atari Environments.” In.

Degris, T., P. Pilarski, and R. S. Sutton (2012a). “Model-Free Reinforcement Learning with Continuous Action in Practice.” In: *American Control Conference*.

Degris, T., M. White, and R. S. Sutton (2012b). “Off-Policy Actor-Critic.” In: *International Conference on Machine Learning*.

Duan, Y., X. Chen, R. Houthooft, J. Schulman, and P. Abbeel (2016). “Benchmarking Deep Reinforcement Learning for Continuous Control.” In: *International Conference on Machine Learning*.

Engstrom, L., A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry (2020). "Implementation Matters in Deep RL: A Case Study on PPO and TRPO." In: *International Conference on Learning Representations.*

Fujimoto, S., H. van Hoof, and D. Meger (2018). "Addressing Function Approximation Error in Actor-Critic Methods." In: *International Conference on Machine Learning.*

Ghiassian, S., B. Rafiee, Y. L. Lo, and A. White (2020). "Improving Performance in Reinforcement Learning by Breaking Generalization in Neural Networks." In: *International Conference on Autonomous Agents and Multiagent Systems.*

Ghosh, D., M. Machado, and N. Le Roux (2020). "An Operator View of Policy Gradient Methods." In: *Advances in Neural Information Processing Systems.*

Graves, E., E. Imani, R. Kumaraswamy, and M. White (2021). "Off-Policy Policy Actor-Critic using Emphatic Weightings." In: *arXiv:2111.08172v1.*

Haarnoja, T., A. Zhou, P. Abbeel, and A. Levine (2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." In: *International Conference on Machine Learning.*

Haarnoja, T., A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine (2019). "Soft Actor-Critic: Algorithms and Applications." In: *arXiv:1812.05905.*

Hasselt, H. van (2010). "Double Q-learning." In: *Advances in Neural Information Processing Systems.*

Henderson, P., R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger (2018). "Deep Reinforcement Learning that Matters." In: *AAAI Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference and AAAI Symposium on Educational Advances in Artificial Intelligence.*

Imani, E., E. Graves, and M. White (2018). "An Off-Policy Policy Gradient Theorem using Emphatic Weightings." In: *Advances in Neural Information Processing Systems.*

Islam, R., P. Henderson, M. Gomrokchi, and D. Precup (2017). "Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control." In: *Reproducibility in Machine Learning Workshop (ICML).*

Jaakkola, T., M. Jordan, and S. Singh (1994). "On the Convergence of Stochastic Iterative Dynamic Programming Algorithms." In: *Neural computation* 6.6, pp. 1185–1201.

Kalashnikov, D., A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine (2018). "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation." In: *Conference on Robot Learning.*

Khan, Arbaaz, Ekaterina Tolstaya, Alejandro Ribeiro, and Vijay Kumar (2020). "Graph Policy Gradients for Large Scale Robot Control." In: *Proceedings of the Conference on Robot Learning*.

Lazić, N., B. Hao, Y. Abbasi-Yadkori, D. Schuurmans, and C. Szepesvári (2021). "Optimization Issues in KL-Constrained Approximate Policy Iteration." In: *arXiv:2102.06234*.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2016). "Continuous Control with Deep Reinforcement Learning." In: *International Conference on Learning Representations*.

Lin, L. (1992). "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching." In: *Machine Learning* 8.3–4, pp. 293–321.

Liu, H., Y. Feng, Y. Mao, D. Zhou, J. Peng, and Q. Liu (2018). "Action-Depedent Control Variates for Policy Optimization via Stein's Identity." In: *International Conference on Learning Representations*.

Mannor, S., R. Rubinstein, and Y. Gat (2003). "The Cross Entropy Method for Fast Policy Search." In: *International Conference on Machine Learning*.

Mei, J., C. Xiao, R. Huang, D. Schuurmans, and M. Müller (2019). "On Principled Entropy Exploration in Policy Optimization." In: *International Joint Conference on Artificial Intelligence*.

Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu (2016). "Asynchronous Methods for Deep Reinforcement Learning." In: *International Conference on Machine Learning*.

Neumann, S., S. Lim, A. Joseph, Y. Pan, A. White, and M. White (2022). "Greedy Actor-Critic: A New Conditional Cross-Entropy Method for Policy Improvement." In: *arXiv:1810.09103*.

Nota, C. and P. S. Thomas (2020). "Is the Policy Gradient a Gradient?" In: *International Conference on Autonomous Agents and Multiagent Systems*.

Obando-Ceron, J. S. and P. S. Castro (2021). "Revisiting Rainbow: Promoting More Insightful and Inclusive Deep Reinforcement Learning Research." In: *International Conference on Machine Learning*.

Pourchot, A. and O. Sigaud (2019). "CEM-RL: Combining Evolutionary and Gradient-Based Methods for Policy Search." In: *International Conference on Learning Representations*.

Rubinstein, R. Y. (1997). "Optimization of Computer Simulation Models with Rare Events." In: *European Journal of Operational Research* 99, pp. 89–112.

— (1999). "The Cross-Entropy Method for Combinatorial and Continuous Optimization." In: *Methodology and Computing in Applied Probability* 1, pp. 127–190.

— (2001). "Combinatorial Optimization, Cross-Entropy, Ants and Rare Events." In: *Stochastic Optimization: Algorithms and Applications*. Vol. 54. Springer US, pp. 303–363.

Rummery, G. and M. Niranjan (1994). *On-Line Q-Learning Using Connectionist Systems*.

Schulman, J., S. Levine, P. Abbeel, M. Jordan, and P. Moritz (2015). “Trust Region Policy Optimization.” In: *International Conference on Machine Learning.*

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O.Klimov (2017). “Proximal Policy Optimization Algorithms.” In: *arXiv:1707.06347.*

Shani, L., Y. Efroni, and S. Mannor (2020). “Adaptive Trust Region Policy Optimization: Global Convergence and Faster Rates for Regularized MDPs.” In: *AAAI Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference and AAAI Symposium on Educational Advances in Artificial Intelligence.*

Silver, D., G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller (2014). “Deterministic Policy Gradient Algorithms.” In: *International Conference on Machine Learning.*

Sutton, R. S. (1984). “Temporal Credit Assignment in Reinforcement Learning.” PhD thesis. University of Massachusetts.

— (1988). “Learning to Predict by the Method of Temporal Differences.” In: *Machine Learning.*

Sutton, R. S. and A. G. Barto (2018). *Reinforcement learning: An Introduction.* MIT Press Ltd.

Sutton, R. S., H. Maei, and C. Szepesvári (2008). “A Convergent O(n) Temporal-difference Algorithm for Off-policy Learning with Linear Function Approximation.” In: *Advances in Neural Information Processing Systems.*

Sutton, R. S., H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora (2009). “Fast Gradient-Descent Methods for Temporal-Difference Learning with Linear Function Approximation.” In: *International Conference on Machine Learning.*

Sutton, R. S., A. R. Mahmood, and M. White (2016). “An Emphatic Approach to the Problem of Off-Policy Temporal-Difference Learning.” In: *Journal of Machine Learning Research.* ISSN: 1532-4435.

Sutton, R. S., D. McAllester, S. Singh, and Y. Mansour (1999). “Policy Gradient Methods for Reinforcement Learning with Function Approximation.” In: *Advances in Neural Information Processing Systems.*

Szita, I. and A. Lörincz (2006). “Learning Tetris Using the Noisy Cross-Entropy Method.” In: *Neural Computation* 18.12.

Thomas, P. (2014). “Bias in Natural Actor-Critic Algorithms.” In: *International Conference on Machine Learning.*

Tomar, M., L. Shani, Y. Efroni, and M. Ghavamzadeh (2020). “Mirror Descent Policy Optimization.” In: *arXiv:2005.09814.*

Tucker, G., S. Bhupatiraju, S. Gu, R. E. Turner, Z. Ghahramani, and S. Levine (2018). “The Mirage of Action-Dependent Baselines in Reinforcement Learning.” In: *International Conference on Machine Learning.*

Uhlenbeck, G. E. and L. S. Ornstein (1930). “On the Theory of the Brownian Motion.” In: *Physical Review* 36 (5), pp. 823–841.

Vieillard, N., T. Kozuno, B. Scherrer, O. Pietquin, R. Munos, and M. Geist (2020). "Leverage the Average: an Analysis of KL Regularization in Reinforcement Learning." In: *Advances in Neural Information Processing Systems.*

Wang, Z., V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas (2017). "Sample Efficient Actor-Critic with Experience Replay." In: *International Conference on Learning Representations.*

Williams, R. J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." In: *Machine Learning* 8.3-4.

Williams, R. J. and J. Peng (1991). "Function Optimization using Connectionist Reinforcement Learning Algorithms." In: *Connection Science* 3.3, pp. 241–268.

Witten, I. H. (1977). "An Adaptive Optimal Controller for Discrete-Time Markov Environments." In: *Information and Control* 34.4, pp. 286–295.

Wu, C., A. Rajeswaran, Y. Duan, V. Kumar, A. M. Bayen, S. Kakade, I. Mordatch, and P. Abbeel (2018). "Variance Reduction for Policy Gradient with Action-Dependent Factorized Baselines." In: *International Conference on Learning Representations.*

Yu, H., W. Xu, and H. Zhang (2021). "TAAC: Temporally Abstract Actor-Critic for Continuous Control." In: *Advances in Neural Information Processing Systems.*

Ziebart, B. D., A. L. Maas, J. A. Bagnell, and A. K. Dey (2008). "Maximum Entropy Inverse Reinforcement Learning." In: *AAAI Conference on Artificial Intelligence.*

Appendix A

Hyperparameter Settings

In this section, we outline the tuned hyperparameters for each algorithm on each environment in our experiments in Chapter 6. For each algorithm, hyperparameters were tuned over an initial 10 runs with different random seeds. Each algorithm saw the same 10 initial random seeds. For a list of all hyperparameters swept, see Chapter 5. For a description of the exact tuning procedure on a per-environment basis, see Section 6.1. For a description of the exact tuning procedure on an across-environment basis, see Section 6.2. In Table A.1, we list the tuned hyperparameters for each algorithm when tuning across environments. In Tables A.2, A.3, and A.4, we list the tuned hyperparameters when tuning per-environment for GreedyAC, VanillaAC, and SAC respectively.

Hyperparameter	κ	α_{critic}	τ
Greedy Actor-Critic	2.0	1e-3	1e-2
Vanilla Actor-Critic	2.0	1e-3	1e-3
Soft Actor-Critic	10.0	1e-5	10.0

Table A.1: Hyperparameters tuned across-environments for GreedyAC, VanillaAC, and SAC.

Hyperparameter	κ	α_{critic}	τ
Acrobot-CA	1e-1	1e-3	1e-2
Acrobot-DA	1e-1	1e-2	-
Mountain Car-CA	1.0	1e-3	10.0
Mountain Car-DA	2.0	1e-3	-
Pendulum-CA	1e-1	1e-2	10.0
Pendulum-DA	1.0	1e-3	-

Table A.2: Hyperparameters tuned per-environment for GreedyAC.

Hyperparameter	κ	α_{critic}	τ
Acrobot-CA	2.0	1e-3	1e-3
Acrobot-DA	1e-1	1e-2	1e-2
Mountain Car-CA	2.0	1e-3	1e-3
Mountain Car-DA	1.0	1e-3	1e-2
Pendulum-CA	1e-2	1e-2	1e-2
Pendulum-DA	2.0	1e-3	1.0

Table A.3: Hyperparameters tuned per-environment for VanillaAC.

Hyperparameter	κ	α_{critic}	τ
Acrobot-CA	10.0	1e-5	10.0
Acrobot-DA	2.0	1e-5	10.0
Mountain Car-CA	1.0	1e-3	1e-3
Mountain Car-DA	1.0	1e-3	1e-2
Pendulum-CA	1e-1	1e-2	1e-1
Pendulum-DA	1.0	1e-3	1.0

Table A.4: Hyperparameters tuned per-environment for SAC.

Appendix B

The Cross-Entropy Method: A Derivation

In this section, we discuss an algorithm upon which we build our novel policy greedification operator. The cross-entropy method (CEM) is both an optimization algorithm and an estimation algorithm, initially developed to estimate the probability of rare events (Boer et al., 2005; Rubinstein, 1997, 1999, 2001). The optimization algorithm is a global, 0-order optimization algorithm which makes it specifically designed to optimize non-concave and even non-differentiable functions. This makes it a desirable optimization algorithm for reinforcement learning where it can be utilized to optimize an action-value function which may not be differentiable.

In this section, we focus on the case of continuous random variables. The results presented here are trivially extended to the discrete case.

B.1 The Cross-Entropy Method for Estimation

In this section, we discuss how the CEM can be applied to estimate the probability of rare events. The problem is as follows: given some probability distribution with density function f, we want to find the probability of some rare event $E(\boldsymbol{Y})$, where $\boldsymbol{Y} \sim f$[1]. In theory, we could use Monte Carlo simulation:

[1]For notational convenience, we may sometimes refer to a random variable being drawn from a density function. In reality, we mean that the random variable is drawn from the distribution which induces the density.

draw many samples from f and compute the relative frequency of $E(\boldsymbol{Y})$. Unfortunately, such a method is problematic: gathering a sufficient number of samples to accurately estimate the probability of $E(\boldsymbol{Y})$ is prohibitively expensive. The CEM works by finding some other distribution $g \neq f$ for which E has high probability. Using importance sampling, the probability of E under f can then be computed using an expectation:

$$\mathbb{P}_f(E(\boldsymbol{Y})) = \mathbb{E}_f[\mathbb{I}(E(\boldsymbol{Y}))] = \mathbb{E}_g\left[\frac{f(\boldsymbol{Y})}{g(\boldsymbol{Y})}\mathbb{I}(E(\boldsymbol{Y}))\right]$$

where $\mathbb{P}$ indicates the probability of some event, $\mathbb{I}$ is the indicator function which is 1 when its argument is true and 0 otherwise.

In a more formal mathematical sense, consider some function $H : \mathcal{Y} \to \mathbb{R}$ and some event $E_\zeta(\boldsymbol{Y}) = \{H(\boldsymbol{Y}) \geq \zeta\}$ for $\zeta \in \mathbb{R}$. Suppose that we wish to estimate the probability of this event with a non-zero estimate. The objective then is to evaluate the following expectation:

$$\mathscr{p} = \mathbb{P}_{f_{\boldsymbol{\xi}}}(E_\zeta(\boldsymbol{Y})) = \mathbb{E}_{f_{\boldsymbol{\xi}}}[\mathbb{I}(E_\zeta(\boldsymbol{Y}))] \tag{B.1}$$

To estimate the probability $\mathscr{p}$ using the CEM, we construct an importance sampling distribution g for which E_ζ is known a priori to be likely. We can then estimate $\mathscr{p}$ with the stochastic estimate:

$$\hat{\mathscr{p}} \doteq \frac{1}{K}\sum_{i=1}^{K}\frac{f_{\boldsymbol{\xi}}(\boldsymbol{Y}_i)}{g(\boldsymbol{Y}_i)}\mathbb{I}(E_\zeta(\boldsymbol{Y}_i)) \approx \mathbb{E}_g\left[\frac{f_{\boldsymbol{\xi}}(\boldsymbol{Y})}{g(\boldsymbol{Y})}\mathbb{I}(E_\zeta(\boldsymbol{Y}))\right] = \mathscr{p} \tag{B.2}$$

with samples $\boldsymbol{Y}_i$ drawn from g and for some $K \in \mathbb{N}$. If g is chosen such that the probability of E_ζ is large enough, then this computation is cheap. In fact, g can be chosen such that the estimator $\hat{\mathscr{p}}$ has zero variance and a single sample from g can be used to recover $\mathscr{p}$ exactly. This is known as the optimal importance sampling distribution and is defined as:

$$g^*(\boldsymbol{Y}) \doteq \frac{\mathbb{I}(E_\zeta(\boldsymbol{Y}))\ f_{\boldsymbol{\xi}}(\boldsymbol{Y})}{\mathscr{p}} \tag{B.3}$$

Evaluating Equation B.2 with a single sample from g^* will result in exactly $\mathscr{p}$. If we are able to construct this distribution, then estimating $\mathscr{p}$ is simple and easy. The difficulty is that generally we cannot compute g^* at all since it requires that we know the solution $\mathcal{p}$ beforehand. It seems that we have not made any progress in finding an estimate of Equation B.1.

Instead of computing g^*, we can use an additional parametric distribution $h_{\boldsymbol{\psi}}$ with parameters $\boldsymbol{\psi} \in \mathbb{R}^{|\boldsymbol{\psi}|}$. For convenience, we often choose this distribution to be in the same class as $f_{\boldsymbol{\xi}}$, in which case we refer to $\boldsymbol{\psi}$ as the *reference parameters*. The idea is to make $h_{\boldsymbol{\psi}}$ as close to g^* as possible and then use $h_{\boldsymbol{\psi}}$ in place of g^* to compute $\hat{\mathcal{p}}$. As long as $h_{\boldsymbol{\psi}}$ places sufficient density on the event E_ζ, then we will be able to estimate $\mathcal{p}$ efficiently using Equation B.2 with $h_{\boldsymbol{\psi}}$ in place of g. To match $h_{\boldsymbol{\psi}}$ to g^*, we can minimize the KL divergence between the two densities:

$$\begin{aligned}
\boldsymbol{\psi}^* &= \arg\min_{\boldsymbol{\psi}} \mathbb{KL}(g^* \mid\mid h_{\boldsymbol{\psi}}) && \text{(B.4)}\\
&= \arg\min_{\boldsymbol{\psi}} \mathbb{E}_{g^*}\left[\ln\left(\frac{g^*(\boldsymbol{Y})}{h_{\boldsymbol{\psi}}(\boldsymbol{Y})}\right)\right] && \text{(B.5)}\\
&= \arg\min_{\boldsymbol{\psi}} \left(\mathbb{E}_{g^*}[\ln g^*(\boldsymbol{Y})] - \mathbb{E}_{g^*}[\ln h_{\boldsymbol{\psi}}(\boldsymbol{Y})]\right) && \text{(B.6)}\\
&= \arg\min_{\boldsymbol{\psi}} \left(\int\limits_{\mathcal{Y}} g^*(\boldsymbol{Y}) \ln g^*(\boldsymbol{Y})\ d\boldsymbol{Y} - \int\limits_{\mathcal{Y}} g^*(\boldsymbol{Y}) \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y})\ d\boldsymbol{Y}\right) && \text{(B.7)}\\
&= \arg\min_{\boldsymbol{\psi}} \left(-\int\limits_{\mathcal{Y}} g^*(\boldsymbol{Y}) \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y})\ d\boldsymbol{Y}\right) && \text{(B.8)}\\
&= \arg\min_{\boldsymbol{\psi}} \left(-\int\limits_{\mathcal{Y}} \frac{\mathbb{I}(E_\zeta(\boldsymbol{Y}))\ f_{\boldsymbol{\xi}}(\boldsymbol{Y})}{\mathcal{p}} \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y})\ d\boldsymbol{Y}\right) && \text{(B.9)}\\
&= \arg\max_{\boldsymbol{\psi}} \mathbb{E}_{f_{\boldsymbol{\xi}}}\left[\mathbb{I}(E_\zeta(\boldsymbol{Y})) \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y})\right] && \text{(B.10)}
\end{aligned}$$

In Equation B.8 we have used the fact that the first term of Equation B.7 does not depend on $\boldsymbol{\psi}$ and therefore will not change the argument at which the minimum is attained. In Equation B.10 we have used the fact that $\mathcal{p}$ is constant and therefore does not change the argument at which the maximum is attained. At last, we have developed some quantity to optimize which will produce the parameters $\boldsymbol{\psi}^*$ to compute an approximation $h_{\boldsymbol{\psi}^*} \approx g^*$, allowing us to compute the rare event probability $\mathcal{p}$. A careful reader will note that we still have a problem. The expectation in Equation B.10 is with respect to the objective density $f_{\boldsymbol{\xi}}$, and the argument to the expectation contains an indicator to the rare event which should be 0 almost always when sampling from $f_{\boldsymbol{\xi}}$. Optimizing Equation B.10 using Monte Carlo estimates will be infeasible. We once again turn to importance sampling.

Consider $h_{\boldsymbol{\chi}}$, a distribution from the same family as $h_{\boldsymbol{\psi}}$ with parameters $\boldsymbol{\chi} \in \mathbb{R}^{|\boldsymbol{\chi}|^2}$. We can rewrite Equation B.10 using importance sampling:

$$\boldsymbol{\psi}^* = \arg\max_{\boldsymbol{\psi}} \mathbb{E}_{h_{\boldsymbol{\chi}}} \left[\frac{f_{\boldsymbol{\xi}}(\boldsymbol{Y})}{h_{\boldsymbol{\chi}}(\boldsymbol{Y})} \, \mathbb{I}(E_\zeta(\boldsymbol{Y})) \; \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y}) \right] \tag{B.11}$$

By solving Equation B.11, we can compute the optimal reference parameters $\boldsymbol{\psi}^*$. We can then use $h_{\boldsymbol{\psi}^*}$ to estimate p, our original objective. Generally, the optimal reference parameters can be estimated using a stochastic estimate of Equation B.11:

$$\begin{aligned} \hat{\boldsymbol{\psi}}^* &= \arg\max_{\boldsymbol{\psi}} \frac{1}{K} \sum_{i=1}^{K} \frac{f_{\boldsymbol{\xi}}(\mathbf{Y}_i)}{h_{\boldsymbol{\chi}}(\mathbf{Y}_i)} \, \mathbb{I}(E_\zeta(\mathbf{Y}_i)) \ln h_{\boldsymbol{\psi}}(\mathbf{Y}_i) \quad \text{where } \mathbf{Y}_i \sim h_{\boldsymbol{\chi}} && \text{(B.12)} \\ &= \arg\max_{\boldsymbol{\psi}} L(\boldsymbol{\psi}) && \text{(B.13)} \end{aligned}$$

where $K \in \mathbb{N}$, and L is defined implicitly. If L is differentiable with respect to $\boldsymbol{\psi}$, we can approximately solve for $\hat{\boldsymbol{\psi}}^*$ using gradient-based optimization algorithms. If the reference distributions $h_{\boldsymbol{\psi}}$ and $h_{\boldsymbol{\chi}}$ are from the natural exponential family, then Equation B.12 can be solved analytically. Finally, we can estimate p as:

$$\hat{p} = \frac{1}{K} \sum_{i=1}^{K} \frac{f_{\boldsymbol{\xi}}(\mathbf{Y}_i)}{h_{\hat{\boldsymbol{\psi}}^*}(\mathbf{Y}_i)} \, \mathbb{I}(E_\zeta(\mathbf{Y}_i)) \quad \text{where } \mathbf{Y}_i \sim h_{\hat{\boldsymbol{\psi}}^*} \tag{B.14}$$

for some $K \in \mathbb{N}$.

Algorithm 8: Iterative CEM Algorithm for Estimation

```
Input: function $H : \mathcal{Y} \to \mathbb{R}$ defining event $E_\zeta$; $N, K \in \mathbb{N}$
$\boldsymbol{\psi}_0 \leftarrow \boldsymbol{\xi}$
$t \leftarrow 0$
do
    Set $\rho_t \in (0, 1)$
    $\boldsymbol{\chi}_t \leftarrow \boldsymbol{\psi}_t$
    Generate $N$ random samples from $h_{\boldsymbol{\chi}_t}$: $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \ldots, \boldsymbol{Y}_N$
    Sort the images of each $\boldsymbol{Y}_i$ under $H$:
        $H_1 = H(\boldsymbol{Y}_{i_1}) \leq \ldots \leq H_N = H(\boldsymbol{Y}_{i_N})$
    Compute $\hat{\zeta}_t$ using Equation B.17:
        $\hat{\zeta}_t = \min\left(\zeta,\ H_{\lceil(1-\rho_t)N\rceil}\right)$
    Solve Equation B.12 for event $E_{\hat{\zeta}_t}$:
        $\boldsymbol{\psi}_{t+1} = \arg\max_{\boldsymbol{\psi}} \frac{1}{N} \sum_{i=1}^{N} \frac{f_{\boldsymbol{\xi}}(\boldsymbol{Y}_i)}{h_{\boldsymbol{\chi}_t}(\boldsymbol{Y}_i)} \mathbb{I}(E_{\hat{\zeta}_t}(\boldsymbol{Y}_i))\ \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y}_i)$
    $t \leftarrow t + 1$
while $\hat{\zeta}_t \neq \zeta$
$T \leftarrow t$
Estimate $p$ using Equation B.14:
    $\hat{p} = \frac{1}{K} \sum_{i=1}^{K} \frac{f_{\boldsymbol{\xi}}(\boldsymbol{Y}_i)}{h_{\boldsymbol{\psi}_t}(\boldsymbol{Y}_i)} \mathbb{I}(E(\boldsymbol{Y}_i))$   where $\boldsymbol{Y}_i \sim h_{\boldsymbol{\psi}_T}$
```

B.2 The Two-Phase, Multi-Level Algorithm

The algorithm outlined in the previous section is only of practical utility when the probability of the event E_ζ is not too small under the distributions $h_{\boldsymbol{\psi}}$ and $h_{\boldsymbol{\chi}}$. Otherwise, the CEM may produce inaccurate estimates of $\boldsymbol{\psi}^*$ and p due to the indicator functions in Equations B.12 and B.14 respectively. In the worst case, an arbitrary value can be assigned to $\boldsymbol{\psi}^*$, which will produce an inaccurate estimate of p with high probability. This is further exacerbated by the fact that the distributional families of $h_{\boldsymbol{\psi}}$ and $h_{\boldsymbol{\chi}}$ as well as the parameters $\boldsymbol{\chi}$ must be specified beforehand. The parameters $\boldsymbol{\chi}$ should place sufficient probability on event E_ζ, but a priori knowledge of these parameters is difficult

[2]In reality, $h_{\boldsymbol{\chi}}$ need not be in the same family as $h_{\boldsymbol{\psi}}$ if using a single iteration of the CEM algorithm. If the full, iterative, multi-level algorithm is used, then these two distributions should be from the same family, otherwise the algorithm may become needlessly complex. For notational convenience, and because the CEM is generally used in its full, iterative, multi-level form, we consider both distributions to be from the same family and use the same symbols for both distributions. In fact this reflects the choices in our novel policy improvement operator.

to obtain. Instead, an iterative approach to the CEM algorithm is often useful. The idea is to generate a sequence of thresholds ζ_t and a sequence of parameters $\boldsymbol{\chi}_t$ and $\boldsymbol{\psi}_t$ such that $h_{\boldsymbol{\psi}_t}$ iteratively places more density on the event E_ζ.

We begin by choosing a value $\rho_0 \in (0,1)$ and setting $\boldsymbol{\chi}_0 = \boldsymbol{\xi}$. We will solve a simpler problem by reducing the threshold of optimization $\zeta_0 \leq \zeta$, where the threshold ζ_0 is induced by ρ_0. Reducing the threshold of optimization results in a new event E_{ζ_0} which is less rare than the original event of interest E_ζ. Estimating the probability of E_{ζ_0} will be easier than estimating the probability of E_ζ. Let us denote

$$p_t \doteq \mathbb{E}_{h_{\boldsymbol{\chi}_t}}[\mathbb{I}(E_{\zeta_t}(\boldsymbol{Y}))] \leq \rho_t \tag{B.15}$$

where ζ_t is set such that the above equation holds. We then estimate p_0 using the CEM (that is, solving Equation B.12). Let $\boldsymbol{\psi}_1$ be the optimal reference parameters for estimating p_0. We then set $\boldsymbol{\chi}_1 = \boldsymbol{\psi}_1$, choose ρ_1, compute ζ_1, and solve for $\boldsymbol{\psi}_2$. We repeat this two-fold process iteratively to eventually estimate both p and $\boldsymbol{\psi}^*$. In the most common case, we fix $\rho_t = \rho \in (0,1)\ \forall t$, but for completeness we deal with the more general case where ρ changes at each iteration.

We must now address the question of setting ρ_t, which induces the threshold ζ_t. The goal is to start with a low threshold and iteratively increase it such that $\lim_{t\to\infty} \zeta_t \approx \zeta$. A simple way to set ζ_t is to use the $(1-\rho_t)$-percentile of H under $h_{\boldsymbol{\chi}_t}$:

$$\begin{aligned} p_t = \mathbb{E}_{h_{\boldsymbol{\chi}_t}}[\mathbb{I}(E_{\zeta_t}(\boldsymbol{Y}))] &= \rho_t \\ \mathbb{E}_{h_{\boldsymbol{\chi}_t}}[\mathbb{I}(E^{\complement}_{\zeta_t}(\boldsymbol{Y}))] &= 1-\rho_t \end{aligned} \tag{B.16}$$

where $E^{\complement}_{\zeta_t}$ is the complement of event E_{ζ_t}. A simple estimator $\hat{\zeta}_t \approx \zeta_t$ can be computed using repeated random sampling. We draw a random sample $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \ldots, \boldsymbol{Y}_N$ from $h_{\boldsymbol{\chi}_t}$, compute the images of each sample $H_j = H(\boldsymbol{Y}_{i_j})$, and sort these images from least to greatest such that $H_1 = H(\boldsymbol{Y}_{i_1}) \leq H_2 = H(\boldsymbol{Y}_{i_2}), \leq \ldots \leq H_N = H(\boldsymbol{Y}_{i_N})$. Finally, we compute $\hat{\zeta}_t$ as the minimum between ζ and the empirical $(1-\rho_t)$-percentile of H:

$$\hat{\zeta}_t = \min\left(\zeta,\ H_{\lceil (1-\rho_t)N \rceil}\right) \tag{B.17}$$

If we find ourselves in the situation where $\hat{\zeta}_t$ has been capped at ζ, then we have successfully recovered an importance sampling distribution with which we can efficiently estimate $\mathscr{p}$, and we can stop the iterative procedure. At this point the empirical $(1-\rho_t)$ percentile will be larger than ζ, indicating that the likelihood of the event of interest E_ζ is approximately larger than or equal to ρ_t under $h_{\boldsymbol{\chi}_t}$. The full iterative CEM algorithm for estimating the probability of a rare event is given in Algorithm 8.

B.3 The Cross-Entropy Method for Optimization

Soon after the CEM was introduced, it was realized that the algorithm could be extended for use in optimization problems (Rubinstein, 1999, 2001). In this section, we discuss this extension of the CEM algorithm.

Let $\mathcal{Y} \subseteq \mathbb{R}^n$ and $H : \mathcal{Y} \to \mathbb{R}$ be some performance function. We would like to discover the maximum attainable performance:

$$\zeta^* \doteq H(\boldsymbol{y}^*) = \max_{\boldsymbol{y}\in\mathcal{Y}} H(\boldsymbol{y}) \tag{B.18}$$

We can replace the optimization problem in Equation B.18 with an estimation problem, where we estimate the probability of H taking on high values under some distribution. If the threshold defining *high values* is sufficiently large, then we can obtain an approximate optima of H using the CEM for rare event probability estimation.

Let $f_{\boldsymbol{\xi}}$ be some distribution parameterized by $\boldsymbol{\xi} \in \mathbb{R}^{|\boldsymbol{\xi}|}$. Similar to the case of rare event probability estimation, we are interested in computing the following probability:

$$\mathscr{p}(\zeta) \doteq \mathbb{P}_{f_{\boldsymbol{\xi}}}(E_\zeta(\boldsymbol{Y})) \tag{B.19}$$

where $\mathscr{p}$ is a function of $\zeta \in \mathbb{R}$. Here $E_\zeta(\boldsymbol{Y}) = \{H(\boldsymbol{Y}) \geq \zeta\}$ as in the previous section. We can use the method of rare-event probability estimation to estimate this quantity; we will utilize the solution to Equation B.10, which we reiterate here:

$$\boldsymbol{\psi}^* = \arg\max_{\boldsymbol{\psi}} \mathbb{E}_{f_{\boldsymbol{\xi}}} \left[\mathbb{I}(E_\zeta(\boldsymbol{Y})) \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y}) \right]$$

where $h_{\boldsymbol{\psi}}$ is some distribution from the same family as $f_{\boldsymbol{\xi}}$ for mathematical convenience, similar to the previous section. We can estimate this quantity using a sample of the expectation:

$$\hat{\boldsymbol{\psi}}^* = \arg\max_{\boldsymbol{\psi}} \frac{1}{K} \sum_{i=1}^{K} \mathbb{I}(E_\zeta(\boldsymbol{Y}_i)) \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y}_i) \qquad \text{where } \boldsymbol{Y}_i \sim f_{\boldsymbol{\xi}} \tag{B.20}$$

where $K \in \mathbb{N}$. At this point, we should recall what these equations are specifically doing. By computing $\hat{\boldsymbol{\psi}}^*$, we find a distribution $h_{\hat{\boldsymbol{\psi}}^*}$ from which we should sample in order to approximate $p(\zeta)$ in Equation B.19 using importance sampling. In essence, what we have achieved at this point is a distribution $h_{\hat{\boldsymbol{\psi}}^*}$ which places higher density than $f_{\boldsymbol{\xi}}$ on values closer to a maximum of H.

Algorithm 9: Iterative CEM Algorithm for Optimization

```
1  Input: objective function $H : \mathcal{Y} \to \mathbb{R}$; $N, K, d \in \mathbb{N}$; $\delta \in \mathbb{R}$ small
2  $t \leftarrow 0$
3  $\boldsymbol{\psi}_0 \leftarrow \boldsymbol{\xi}$
4  do
5      $t \leftarrow t + 1$
6      Set $\rho_t \in (0, 1)$
7      Generate $N$ random samples from $h_{\boldsymbol{\psi}_{t-1}}$: $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \ldots, \boldsymbol{Y}_N$
8      Sort the images of each $\boldsymbol{Y}_i$ under $H$:
9          $H_1 = H(\boldsymbol{Y}_{i_1}) \leq \ldots \leq H_N = H(\boldsymbol{Y}_{i_N})$
10     Compute $\zeta_t$:
11         $\zeta_t = H_{\lceil (1-\rho_t)N \rceil}$
12     Solve Equation B.20 for event $E_{\zeta_t}$:
13         $\boldsymbol{\psi}_t = \arg\max_{\boldsymbol{\psi}} \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(E_{\zeta_t}(\boldsymbol{Y}_i)) \ln h_{\boldsymbol{\psi}}(\boldsymbol{Y}_i)$
   while $t \leq d$ or $|\zeta_t - \zeta_{t-1}|, \ldots, |\zeta_{t-d+1} - \zeta_{t-d}| \geq \delta$
14 $T \leftarrow t$
15 Estimate $\zeta^*$:
16     $\zeta^* \approx \mathbb{E}_{h_{\boldsymbol{\psi}_T}}[H(\boldsymbol{Y})] \approx \frac{1}{K} \sum_{i=1}^{K} H(\boldsymbol{Y}_i) = \hat{\zeta}^*$   where $\boldsymbol{Y}_i \sim h_{\boldsymbol{\psi}_T}$
```

A major difference exists between the optimization and estimation versions of the CEM. In the case of the latter, we are given a distribution with density $f_{\boldsymbol{\xi}}$ and a value ζ which defines an event $E_\zeta(\boldsymbol{Y}) = \{H(\boldsymbol{Y}) \geq \zeta\}$. We then approximate the probability of this given event under the given distribution $f_{\boldsymbol{\xi}}$. In the optimization case, we must ourselves specify ζ and $\boldsymbol{\xi}$. At a qualitative level, the utility of Equation B.20 is inversely related to magnitude of ζ. As the rarity of event E_ζ increases, $\mathbb{I}(E_\zeta(\boldsymbol{Y})) = 0$ for nearly all samples and $\hat{\boldsymbol{\psi}}^*$ may no longer be accurate if indeed useful at all. In the worst case, an arbitrary value for $\hat{\boldsymbol{\psi}}^*$ will satisfy Equation B.20. A trade off exists in setting both ζ and $\boldsymbol{\xi}$: ζ should be set close to ζ^* in order that $h_{\boldsymbol{\psi}^*}$ place sufficient density around ζ^* and is therefore useful for generating an approximate solution to Equation B.18. On the other hand, $\boldsymbol{\xi}$ should be set such that $\mathbb{P}_{f_{\boldsymbol{\xi}}}(E_\zeta)$ is not too low, otherwise the estimator $\hat{\boldsymbol{\psi}}^*$ may not be accurate. The issue is that in setting ζ close to ζ^*, we implicitly decrease the $\mathbb{P}_{f_{\boldsymbol{\xi}}}(E_\zeta)$.

Unsurprisingly, we find ourselves in a dilemma quite similar to that which we experienced when studying the CEM for estimation. The solution is also similar: use an iterative, two-step approach. In this approach, we start with a low value for threshold ζ_t and solve Equation B.20 with samples from $h_{\boldsymbol{\psi}_{t-1}}$ to find the parameters $\boldsymbol{\psi}_t$. We then increase the threshold $\zeta_{t+1} \geq \zeta_t$ and solve Equation B.20 with samples from $h_{\boldsymbol{\psi}_t}$ to find the parameters $\boldsymbol{\psi}_{t+1}$. Iteratively, we adjust our distribution parameters to place higher density on values for which the objective H is maximized while increasing the threshold over time. We repeat this process with the goal of eventually recovering both the maximum performance $\zeta^* = \max_{\boldsymbol{y} \in \mathcal{Y}} H(\boldsymbol{y})$ and the parameters $\boldsymbol{\psi}^*$ which assign density only to $\arg\max_{\boldsymbol{y} \in \mathcal{Y}} H(\boldsymbol{y})$. The final CEM algorithm for optimization is given in Algorithm 9.

The CEM algorithm for optimization is designed to find the maximum of a performance function $\hat{\zeta}^* \approx \max_{\boldsymbol{y} \in \mathcal{Y}} H(\boldsymbol{y})$. Given that the final iteration of Algorithm 9 is T, we can estimate the set $\arg\max_{\boldsymbol{y} \in \mathcal{Y}} H(\boldsymbol{y})$ in a few different ways. We can use the modes of $h_{\boldsymbol{\psi}_T}$. We can use a random sample drawn from $h_{\boldsymbol{\psi}_T}$. In the case of unimodal $h_{\boldsymbol{\psi}_T}$, we can use its mean to approximate a single argmax.