University of California
Santa Barbara

Improving Reinforcement Learning for Robotics with Control and Dynamical Systems Theory

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Sean M Gillen

March 2022

The Dissertation of Sean M Gillen is approved.

Professor Joao Hespanha

Professor Linda Petzold

Professor Yon Visell

Professor Katie Byl, Committee Chair

March 2022

Improving Reinforcement Learning for Robotics with Control and Dynamical Systems Theory

Acknowledgements

First and foremost I would like to acknowledge my advisor, Katie Byl, for her support and mentorship throughout my PhD. She welcomed me into her lab, offered me the guidance to get started, and the freedom to explore the many different research topics that I was interested in. I would also like to extend my thanks to the rest of my committee, Joao Hespanha, Linda Petzold, and Yon Visell, for their feedback and support.

I would also like to thank the other members of the UCSB robotics lab that I had the privilege of working beside. Guillaume Bellegarda, Nihar Talele, Asutay Ozmen, Thomas Ibbetson, Roman Aguillera, Sean Anderson, and Chris Cheney. Guillaume and Nihar in particular were the senior members of the lab when I first joined, and they were role models and offered invaluable guidance. Additionally, I would like to acknowledge the visiting students that I had to privilege to work with and mentor, Marco Molnar, Nina Grabka, and Min Dai.

I also need to thank all of my friends that supported me throughout this time. Both my old friends: Fraser Hood, Haris Godil, Colin Briber, Holden Smith, and Stephanie Dang; and the many new friends I have met during my time in this PhD program: Nathan Tucker, Rahul Chandan, Chris Salls, Chani Jindal, Bryce Ferguson.

My Mother, Father, and brother Kyle all deserve extensive thanks for their unending support and care. Lastly I thank my partner, Jennifer Chan, for her love, support, and encouragement during final stage of my PhD.

SEAN M GILLEN

sgillen@ucsb.edu | github.com/sgillen | linkedin.com/in/sgillen

EDUCATION

University of California Santa Barbara September 2017 - March 2022 (Expected)
MS/PhD Electrical And Computer Engineering

University of Maryland College Park September 2013 - May 2017
BS Electrical Engineering

WORK HISTORY

University of California Santa Barbara September 2017 - Present
Graduate Student Researcher

- Researching the application of reinforcement learning to robotic control (**Python, C++, Matlab**). Focusing on the control of under-actuated dynamic systems and legged locomotion in particular.

Veoneer June 2019 - February 2020
Robotics Consultant

- Delivered a **C++** device driver and **ROS** node for a new automotive camera system designed for Daimler.
- Implemented camera authentication and security on a low power embedded system (**C**).

Northrop Grumman June 2016 - September 2016
Electrical Engineer

- System level integration for prototype sensor payloads in an unmanned underwater vehicle (UUV).

Naval Air Systems Command (NAVAIR) June 2015 - September 2015
Electrical Engineering Intern

- Developed a prototype vision system capable of imaging in extremely turbid underwater environments.
- Designed and constructed an underwater vessel to house and control the visioning system.

University Of Maryland September 2015 - June 2016
Undergraduate Student Researcher

- Implemented computer simulation of nematic liquid crystals using **Matlab** and **Mathematica**.
- Gave a talk at the American Physics Society March Meeting.

Horn Point Laboratory June 2014 - June 2015
Software Engineer

- Wrote software (**C++**) for a Remus 600 UUV to provide altitude control, user-defined acoustic modem messages, augmented data simulation, and adaptive mission planning to the vehicle.

PERSONAL PROJECTS

Seagul: Utility Library for Reinforcement Learning

- Wrote a **Python** package implementing state-of-the art reinforcement learning algorithms. Now used internally at the UCSB Robotics Lab for research in robotic control.

Qubo: Underwater Octocopter

- Led a team of around 8 engineers to design and build an autonomous underwater octocoptor from scratch. The vehicle can autonomously navigate to targets using computer vision. Semifinalists at Robosub 2017.
- Wrote majority of the vision (**OpenCV**), control (**C++**), autonomy (**Python**), and embedded (**Arduino**) software, using **ROS** as a middleware.

Aq Bars: Tactile, Fist Bumping, Robot Arm

- Wrote software connecting a Novint Falcon controller to a low-cost robot arm, allowing users to control the arm and "feel" objects it interacts with.
- Wrote Arduino code for the arm to allow for interactive use.

Compressed Air Engine

- Manufactured a small compressed air engine, using manual and CNC Machine tools.

PUBLICATIONS

Sean Gillen and Katie Byl. Leveraging reward gradients for reinforcement learning in differentiable physics simulations, 2022 (Prepint, under review)

Sean Gillen, Asutay Ozmen, and Katie Byl. Direct random search for fine tuning of deep reinforcement learning policies, 2021 (Preprint, under review)

Sean Gillen and Katie Byl. Mesh based analysis of low fractal dimension reinforcement learning policies. *International Conference on Robotics and Automation*, 2021

Sean Gillen and Katie Byl. Explicitly encouraging low fractional dimensional trajectories via reinforcement learning. *4th Conference of Robotic Learning*, 2020

Sean Gillen, Marco Molnar, and Katie Byl. Combining deep reinforcement learning and local control for the acrobot swing-up and balance task. *59th IEEE Conference on Decision and Control*, 2020

TEACHING

Intermediate C programming (Spring & Fall 2016)
Introduction to Circuits (Fall 2017)
Digital Control (Winter 2017)
Introduction to Probability & Statistics (Spring 2017)
Robotic Modeling & Control (Fall 2018)
Operating Systems (Spring 2019)
Compilers (Fall 2020)
Algorithms & Data Structures (Winter 2021, Spring 2021)

Abstract

Improving Reinforcement Learning for Robotics with Control and Dynamical Systems Theory

by

Sean M Gillen

Recent advances in machine learning, simulation, algorithm design, and computer hardware have allowed reinforcement learning (RL) to become a powerful tool which can solve a variety of challenging problems that have been difficult or impossible to solve with other approaches. One of the most promising applications of RL is robotic control, in which researchers have demonstrated success on a number of challenging tasks, from rough terrain locomotion to complex object manipulation. Despite this, there remain many limitations that prevent RL from seeing wider adoption. Among these are a lack of any stability or robustness guarantees, and a lack of any way to incorporate domain knowledge into RL algorithms.

In this thesis we address these limitations by leveraging insights from other fields. We show that a model-based local controller can be combined with a learned policy to solve a difficult nonlinear control problem that modern RL struggles with. In addition, we show that gradients in new, differentiable simulators can be leveraged by RL algorithms to better control the same class of nonlinear systems.

We also build on prior work that approximates dynamical systems as discrete Markov chains. This representation allow us to analyze stability and robustness properties of a system. We show that we can modify RL reward functions to encourage locomotion policies that have a smaller Markov chain representation, allowing us to expand the scope of systems that this type of analysis can be applied to. We then use a hopping robot simulation as a case study for this type of analysis. Finally, we show that the same tools that can shrink the Markov chain size can also be used for more generic fine tuning of RL policies, improving performance and consistency of learned policies across a wide range of benchmarking tasks.

Contents

Chapter 1

Introduction and Literature Review

The availability of computation as a resource has been growing exponentially since at least the 1970s, and there is every indication that this resource will, even if not exponentially, continue to become cheaper and more available well into the foreseeable future. The massive amount of computing power available today has been leveraged by scientists and engineers across a vast number of domains, from medical researchers predicting protein folding at a molecular level, to environmental scientists modeling the entire global climate.

Perhaps the application of these powerful computers with the biggest potential for impact in the long term, and certainly the one that has gotten the most media attention over the last decade, is Artificial Intelligence (AI) and specifically Machine Learning (ML). Machine learning can be defined as any algorithm that can improve itself automatically from data or experience, without being explicitly programmed to do so. Over the last decade, the number of research projects and commercial applications using machine learning have exploded. Image classification, natural language processing, targeted advertising, and even the next generation of art and poetry are just some of the many applications of machine learning in use today.

The particular interest in this thesis is a subfield of machine learning called reinforcement learning. In the language of reinforcement learning, we task an artificial agent to learn through trial and error how to accomplish some goal that we specify. The agent's actions are reinforced by a reward function, which we as scientists and engineers provide. Reinforcement learning has been responsible for many of the examples of machine learning that we see today, from beating the human world champion of Go, to learning to control a nuclear fusion reaction, to controlling complex robotic systems.

The control of robotic systems is a particularly promising application of reinforcement learning. Robots are ever increasing in complexity, making use of high dimensional sensors such as cameras and Lidar, and being asked to interact with the real world, outside of the laboratories and factories where they have historically been used. These developments present significant challenges to traditional control methods. Reinforcement learning shows promise in its ability to solve problems for systems that are hard to model, and/or that the user doesn't know how to solve themselves. There has been much work applying RL to robot locomotion in particular, which could enable contact free delivery during a pandemic, emergency work after an environmental disaster, or use as a logistical tool in military applications.

However, despite the promise and early successes, there remain many limitations that prevent RL from seeing wider adoption. Among these are a lack of any stability or robustness guarantees, and a lack of any way to incorporate domain knowledge into RL algorithms. In this thesis we will address these limitations by leveraging insights and tools from control and dynamical systems theory. We show that controllers from model-based optimal control can be combined with learned controllers in an ad hoc fashion to solve a difficult nonlinear control problem. We also show that gradients from a new class of differentiable physics simulation can be leveraged to solve this same class of problem with a more general approach.

Additionally, we build on prior work that approximates dynamical systems as discrete Markov chains. This representation allow us to analyze stability and robustness properties of a system. We show that we can modify RL reward functions to encourage locomotion policies that have a smaller Markov chain representation, allowing us to expand the scope of systems that this type of analysis can be applied to. We then use a hopping robot simulation as a case study for this type of analysis. Finally, we show that the same tools that can shrink the Markov chain size can also be used for more generic fine tuning of RL policies, improving performance and consistency of learned policies across a wide range of benchmarking tasks.

In this chapter, we will introduce relevant literature as a chronological history, expand on the current limitations of RL for control, and finally present an outline for the remainder of this thesis.

1.1 Literature Review

1.1.1 Early Reinforcement Learning

In their classic textbook on the subject, Sutton and Barto [1] argue that reinforcement learning can trace its roots to theories of animal cognition developed (independently) by Alexander Bain and Lloyd Morgan in the 1800s. The artificial intelligence community would start to consider these ideas in the 1950s, with works from Minsky [2], and Farley and Clark [3] who developed neural network machines that could learn by trial and error. Over the next decade we would see the first examples of what would become classic problems for reinforcement learning. In 1959 Samuel would introduce the first computer program that could play checkers [4], in 1963 a system called MENACE learned to play a game of Tic-Tac-Toe [5], and in 1968 a system called BOXES was used to balance a cart-pole pendulum [6].

Independently, during this same period, the theory of optimal control was developed. Although, even today, optimal control and reinforcement learning are seen as separate fields with distinct histories, the two are very closely related. Rather than maximizing a reward, optimal control seeks to minimize a cost function. Obviously these two objectives differ only by a minus sign, and some [1] even consider optimal control to a subset of reinforcement learning. In either case, during this time, Bellman formulated the Markov Decision Process (MDP) [7] which to this day underpins the theoretical framework of reinforcement learning. In 1960 an algorithm called value iteration was introduced as a method to solve generic MDPs [8]. Although value iteration and dynamic programming, more generally, can theoretically solve any MDP, these methods suffer from what Bellman called the “curse of dimensionality”. Essentially, the time to solve a problem with these methods grows exponentially with the dimensionality of the problem. This curse still plagues us today, and is something we will come back to many times in later chapters.

From here we can jump ahead to the 1980s, which is arguably when reinforcement learning as we know it today would emerge. In 1983, Barto et.al, would introduce the influential actor-critic architecture, which was used to solve a cart-pole balancing problem [9]. In Watkin’s 1989 PhD thesis, he presents Q-learning, which is the basis for many of the *off*-policy deep reinforcement learning algorithms used today [10]. Later in 1992, Williams would introduce REINFORCE, a classic example of a policy gradient algorithm, which has become the the basis for many of the *on*-policy reinforcement algorithms used today [11]. Finally we will mention TD-Gammon, in which a reinforcement algorithm was able to play Backgammon at the level of the top humans of the day [12]. TD Gammon received much attention from the press, and generated much excitement around the field.

1.1.2 Modern Reinforcement Learning

Starting in 2012, neural networks saw a resurgence of interest in machine learning. This is usually attributed to the work of Alex Krizhevsky, who showed that deep convolutional neural networks were extremely effective at image classification, a form of supervised learning [13]. The key insight was that deep neural networks were well-suited to leverage the large data sets that modern computing hardware was enabling. Unique among the other function approximators that were popular at the time, DNNs continued to improve their accuracy as they were trained on more and more data. These results set off an explosion of interest in machine learning, with deep neural networks at the center. This trend would quickly spread to reinforcement learning as well.[1]

Just one year after AlexNet[2], Mnih et al. introduced learning with Deep Q-Networks (DQN), where they used a similar deep convolutional network to approximate Q functions [14]. They used this framework to train agents to play a suite of Atari 2600 video games. The inputs were raw pixels, the output a button on the controller, and the reward the score in whatever game they were playing. (All Atari 2600 games included a score, always visible, at the top of the screen). They would go on to extend those results and in 2015 published results demonstrating human-level performance on these same tasks [15]. These results were extremely impressive at the time. The algorithm they presented simultaneously solves a difficult computer vision and artificial intelligence task and is extremely general. With a single network architecture and set of hyperparameters, they were able to achieve human level performance on 49 different games.

[1]In the context of reinforcement learning for control, the "deep" neural networks are often multi-layer perceptrons with two or three layers. Despite this, the term deep reinforcement learning is often used to describe any reinforcement learning algorithm developed after 2012 that uses neural networks. An additional complication is that some of the breakthroughs in the space during this time do not use neural networks at all. Some refer to these gradient free algorithms as alternatives to reinforcement learning, but for our purposes we will refer to algorithms from this period that solve MDPS as modern reinforcement learning.

[2]The specific architecture used in Alex's original paper is now called AlexNet.

2015 would also see RL in continuous domains. Lilicrap et al. introduced Deep Determinstic Policy Gradients (DDPG) [16] which was inspired by DQN but could be used in continuous domains. Also that year, Schulman would introduce Trust Region Policy Optimization (TRPO) [17], which is a policy gradient method that utilizes a trust region to avoid large drops in performance between policy updates. These algorithms were able to control simulated robotic systems with continuous action spaces.

In 2016 the authors of the DQN paper would publish Asynchronous Advantage Actor-Critic (A3C), a policy gradient algorithm that outperformed DQN on the Atari tasks. Later that year, Alpha Go achieved superhuman performance in the game of Go using a policy gradient reinforcement algorithm combined with supervised learning [18]. Alpha Go in particular would receive a huge amount of press, and drive even more attention towards the field.

Also in 2016, OpenAI gym [19] was made public. Gym is an API (application programming interface) and a set of benchmarking problems which would become very influential. The Deepmind control suite was also released, performing a similar role, but focusing only on continuous control [20]. These standardized environments allowed for more direct comparisons between the many different algorithms being developed at this time.

2017 and 2018 would see many refinements and improvements of these previously introduced algorithms. Proximal Policy Optimization (PPO) [21] was introduced which can be seen as a spiritual successor and more efficient version of TRPO. To this day, PPO remains the go-to reinforcement algorithm for OpenAI [22], because of its ease of use and good performance. Twin-Delayed Deep Deterministic Policy Gradients (TD3) introduced an algorithmic improvement to DDPG that significantly stabilized the training [23]. Soft Actor-Critic (SAC) was also introduced here, which is similar to DDPG and TD3 in that they each use off-policy learning [24]. SAC uses the so-called "maximum en-

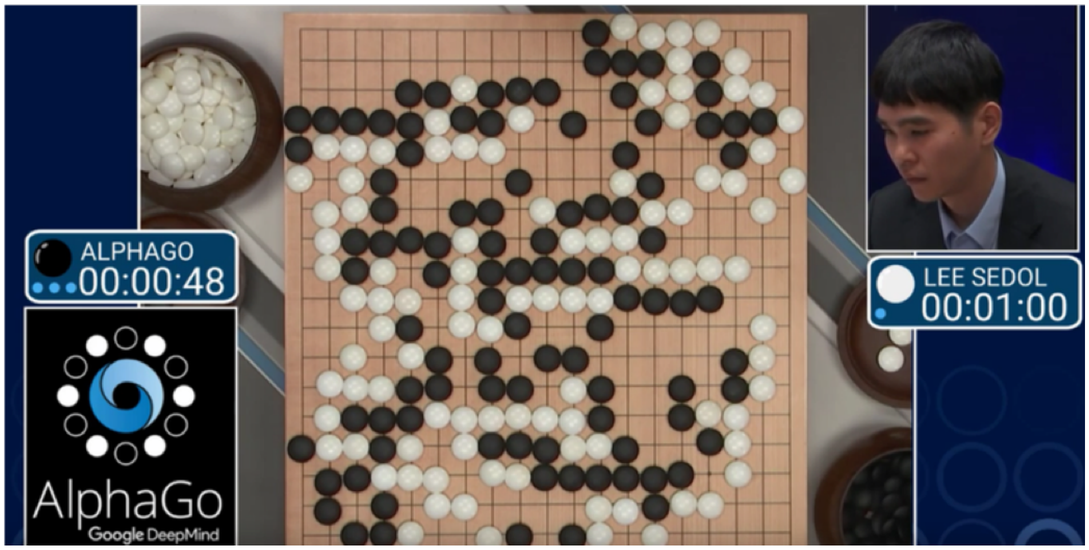


Figure 1.1: Alpha Go [18] playing a game of Go with the grand-master Lee Sedol. Alpha Go would win this match, and drive much attention from the general public to reinforcement learning.

tropy framework", which encourages exploration and was effective at finding locomotion policies in the openAI gym locomotion environments. Furthermore, Salimans et al. with OpenAI showed that Evolutionary Strategies (ES) could offer a very scalable gradient free method for reinforcement learning [25]. Though the authors advertise their work as an alternative to reinforcement learning, meaning an alternative to PPO, TD3 and the like, for our purposes we will consider ES to be a gradient-free, modern reinforcement learning algorithm.

These new algorithmic developments were used by researchers across a wide variety of domains. Perhaps the most impressive was in the realm of esports. In 2017 Open AI unveiled agents trained with PPO that could play the Dota 2 [26]. The next year, a team of these agents was able to play at a professional level against human opponents – a feat no other AI developed for Dota can claim. In 2019, DeepMind revealed their own agents capable of playing another popular esport, Starcraft II [27]. Using a combination of supervised learning and their own reinforcement learning method similar to A3C, they were able to train agents that could play at a grand master level, which is competitive with professional human players. Again, no other method for developing AI for Starcraft can make this claim.

In 2020, the same team that released Alpha Go released Mu Zero, which utilized self play to surpass Alpha Go and can additionally play shogi, chess, and a suite of Atari games, all at superhuman levels [28].

1.1.3 Modern Reinforcement Learning for Robotic Control

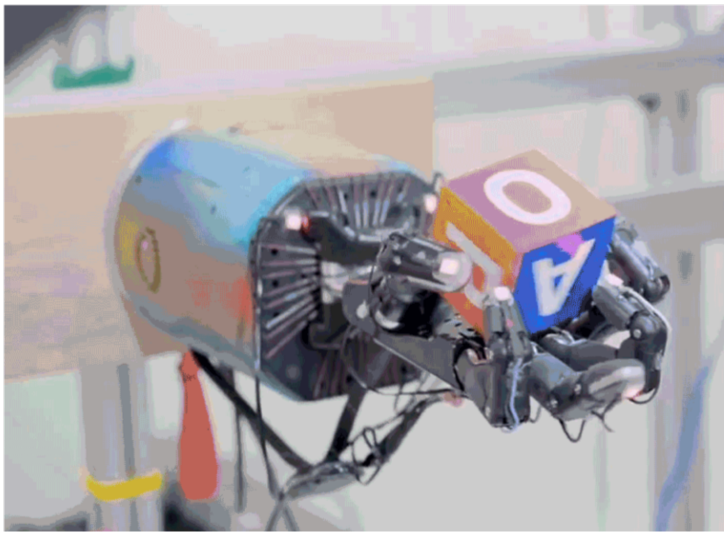


Figure 1.2: A robotic hand trained with RL to dexterously manipulate a cube [29]. Learning was done “end-to-end”, meaning the inputs were raw camera images, and the outputs were motor commands.

Though reinforcement learning has been applied to robotics in a variety of contexts, for example path planning and trajectory optimization, in this thesis we are primarily interested in low-level robotic control. There have been many works applying modern reinforcement learning techniques in this context. For example, [30] and [31] each applied an actor-critic algorithm to control an autonomous underwater vehicle, and [32] applies a policy gradient algorithm to quadcopter control.

Though it is undoubtedly interesting that the same algorithms that can play Go or Starcraft are able to control these systems, RL has not replaced traditional control techniques for these sorts of systems. Existing, well-understood, easily designed, and battle tested controllers, like the workhorse PID (proportional-integral-derivative) control framework, already meet the requirements we have for these systems.

Where then might RL be of use for control? We should look to areas where controller design is very difficult, or where there simply is no technique to solve the task. Robotic locomotion is one such area, which we will cover in its own section. Another is manipulation, particularly in unstructured environments or with high-dimensional manipulators.

Reinforcement learning has been applied productively to manipulation tasks. In 2018, [29] unveiled an agent that could control a humanoid robotic hand to dexterously manipulate various objects, and would later unveil a similar agent that could solve Rubik's cubes [33]. This training was done "end-to-end" meaning that the inputs were raw camera pixels, and the outputs were motor commands to the robot. Similar to the earlier example with Atari games, these algorithms are able to simultaneously solve computer vision and manipulation problems. Either one by itself present challenges to traditional approaches.

1.1.4 Legged Locomotion

Legged robots have clear potential to play an important role in our society in the near future. Examples include contact-free delivery during a pandemic, emergency work after an environmental disaster, or as a logistical tool for the military. Legged robots simply expand the reach of robotics when compared to wheeled systems. However, compared to wheeled systems, designing control policies for legged systems is a much more complex task, especially in the presence of disturbances, noise, and unstructured environments.

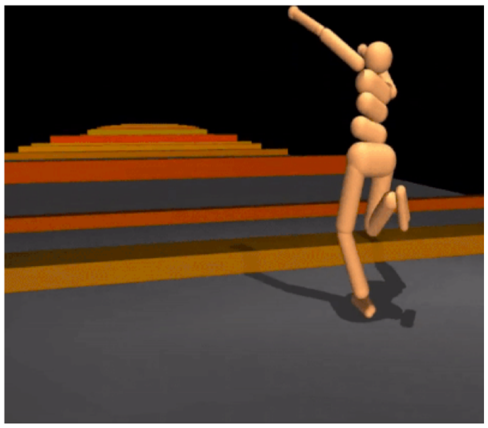


Figure 1.3: A simulated humanoid trained with RL to navigate an obstacle course [34]

Clearly, RL can be used to tackle these complexities. However despite this, the most famous and arguably most capable legged robots belong to Boston Dynamics, who seeming use no learning at all for their gait control. While Boston Dynamics has not published a paper since introducing Big Dog in 2008 [35], we can reasonably guess from that work and from press releases etc. that they make heavy use of trajectory optimization and other model-based approaches. The results are undeniably impressive, but they require a significant amount expert human labor for each new robot, or even for a new behavior for a robot.

How has RL been applied in this context? Let us first mention physics based character animation. Animating characters for movies and video games is also a very labor intensive task. Using reinforcement learning to train policies that move these characters in virtual spaces is one way to automate this [36] [37]. Although the focus and goals are different, there is much overlap with reinforcement learning for robot locomotion. And indeed, Michiel van de Panne's group has demonstrated transferring gaits developed in simulation for the purposes of animation to a real Cassie robot [38].

Additionally, much of the work done by researchers focused on RL more generally have used legged locomotion in simulation as a test problem. We have already mentioned openAI Gym [19], which includes a set of locomotion environments, and many of the algorithms introduced were originally tested on locomotion environments. Another work worth mentioning was done by Heess et al. [34], who demonstrated that RL could learn rough terrain locomotion policies for a variety of morphologies.

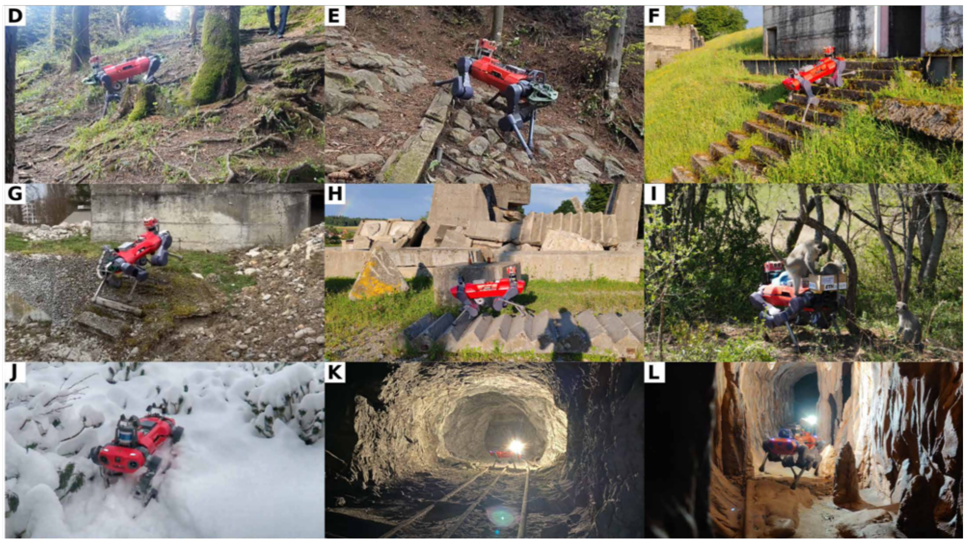


Figure 1.4: A series of examples showing an quadruped robot trained with RL navigating various challenging obstacles and terrains [39].

In terms of real robotic systems being trained with RL, there is often a significant "sim-to-real" gap between the closed-loop dynamics of a real world robot, as compared to a simulated version. As one notable success here, Google Brain and DeepMind showed in 2018 that a control policy developed for the Minotaur quadruped robot, trained in simulation with PPO, could also successfully be used when transferred to the real robot [40].

The bipedal Cassie robot, designed and built by Agility Robotics, has been used by researchers at the University of Oregon to demonstrate impressive results using RL for locomotion [41]. They have also demonstrated that PPO can be used to train a robot to blindly but reliably traverse stairs [42]. The Ohio State University has a Digit robot, a similar biped also developed at Agility Robotics, and in [43] they demonstrated that ES (Evolutionary Strategies) could be used to generate trajectories to enable locomotion.

Some of the most impressive results have come from Marco Hutter's Robotic Systems Lab at ETH-Zurich. They have demonstrated learned policies that enable the Anymal quadruped robot to walk and to recover from falls, and these RL policies also outperform hand-designed controllers [44] [45]. More recently, they integrated these results with lidar sensing to enable rough terrain walking in outdoor environments that seemingly may rival what Boston Dynamics is able to do [39].

In 2018, Mania et. al. showed that a simple algorithm called Augmented Random Search (ARS) combined with static linear policies were competitive with deep reinforcement learning for locomotion tasks [46]. A parallel line of work [47] showed that radial basis functions and linear policies could be trained for these tasks using a natural policy gradient algorithm. More recently, [48] showed that these linear policies could be transferred to enable locomotion on a real Cassie robot.

Limitations of Reinforcement Learning

Despite these impressive results, there remain many limitations that prevent reinforcement learning from seeing wider adoption. For example, there is a lack of any stability or robustness guarantees for trained policies. One can perform Monte Carlo simulations, simply running tests with the trained policies and observing any failure modes. But this is unsatisfying and impractical, especially in cases where rare but catastrophic failures can occur, for example a self driving car. If we run the simulation 1000 times and there are no failures do we call our system safe? If not, then how many trials *do* we need? Are there other strategies we can potentially use, beyond brute force Monte Carlo trials, to quantify failure rates for such rare events?

In addition to this, RL algorithms treat their environments as a black box, assuming nothing about the system they are tasked with controlling. This is both a great strength of these algorithm, since it results in RL being extremely general, but is also unsatisfying in the many situation where we as engineers have a lot of domain knowledge that could be leveraged.

In this thesis, we address these limitations by leveraging insights from other fields, primarily control and dynamical systems theory. We show that a model-based local controller can be combined with a learned policy to solve a difficult nonlinear control problem that modern RL struggles with. In addition, we show that gradients in new, differentiable simulators can be leveraged by RL algorithms to better control the same class of nonlinear systems.

We also build on prior work that approximates dynamical systems as discrete Markov chains. This representation allow us to analyze stability and robustness properties of a system. We show that we can modify RL reward functions to encourage locomotion policies that have a smaller Markov chain representation, allowing us to expand the scope of systems that this type of analysis can be applied to. We then use a hopping robot simulation as a case study for this type of analysis. Finally, we show that the same tools that can shrink the Markov chain size can also be used for more generic fine tuning of RL policies, improving performance and consistency of learned policies across a wide range of benchmarking tasks.

1.2 Structure of This Thesis

We will now outline the structure of the rest of this thesis. The work presented in Chapters 3-7 has also been published as discrete papers and pre-prints [49] [50] [51] [52] [53].

- Chapter 2 introduces necessary background and the mathematical preliminaries for the thesis. We formally introduce reinforcement learning and define the Markov Decision Process. Supervised learning, neural networks, and parameterized functions in general are discussed. A more detailed, non-historic, outline of the modern reinforcement learning landscape is presented. Finally, we discuss the differences between on-policy, off-policy, and gradient-free algorithms, and introduce algorithms which are used in the text.

- Chapter 3 describes a novel algorithm that combined deep reinforcement learning with a local optimal controller. We then demonstrate this algorithm by solving a challenging nonlinear control problem, and show that it outperforms other state-of-the-art reinforcement learning algorithms in this setting.

- Chapter 4 introduces meshing, an algorithm to approximate a continuous dynamical system as a discrete-time Markov chain. We introduce a novel reward function for reinforcement learning that incorporates a notion of dimensionality for these Markov chains. We then show that ARS can learn policies that minimize this dimensionality, and that this also improves the robustness of the resulting policies.

- Chapter 5 discusses meshing the reachable state space of a system. Meshes of the reachable state space have been used previously for controller design and analysis of legged locomotion policies. Using a model hopping system as a case study, we show that the policies from Chapter 4 produce dramatically smaller meshes in this context.

- Chapter 6 extends the results from Chapters 4 and 5 to arbitrary neural network policies. We then demonstrate that using this method can successfully minimize the dimensionality across a wide range of benchmarking environments.

- Chapter 7 introduces differentiable physics simulators, a new development in the space of robotic simulation. We discuss the difficulties of using gradients from these simulators for control, and then immediately introduce an algorithm that does just that.

- Chapter 8 makes concluding remarks and discusses future research directions.

Chapter 2

Background

2.1 Reinforcement Learning

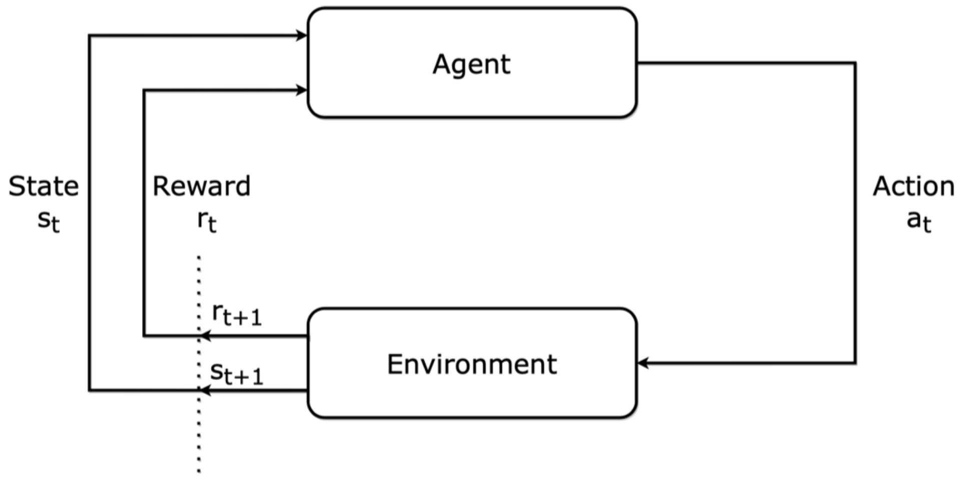


Figure 2.1: A diagram showing the essential parts of a Markov Decision Process (MDP).

Reinforcement learning concerns an agent taking actions in an environment in order to maximize some reward function. The environment is described by a state that can be observed by the agent. One example might be a chess playing program. The state is the position of the pieces on the board, the action is the move on the current turn, and their reward is plus one on any turn when they win the game, a minus one on a turn where they lost the game, or a zero otherwise. Or maybe the agent is a robot, the states are joint angles and velocities, the action a vector of motor commands, and the reward function is equal to their forward velocity. As these examples hopefully illustrate, reinforcement learning is extremely general, and can and has been applied to a myriad of problems across a number of domains.

The Markov Decision Process

More formally, the environment is a discrete time dynamical system described by state $s_t \in \mathbb{R}^n$ [1] and the current action $a_t \in \mathbb{R}^b$. We also introduce η, a parameter that captures any stochastic behavior in the environment and policy. An evolution function $f : \mathbb{R}^n \times \mathbb{R}^b \times \mathbb{R} \to \mathbb{R}^n$ takes as input the current state, action, and η, and outputs the state at time t+1:

$$s_{t+1} = f(s_t, a_t, \eta) \tag{2.1}$$

The policy, sometimes called the controller, is a function π parameterized by a vector θ. The policy is a function which maps states to actions $\pi : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^m$ such that:

$$a_t = \pi_\theta(s_t, \eta) \tag{2.2}$$

In general π may be stochastic, in which case the underlying object being parameterized is a probability distribution, which is sampled from at every time-step to get the current action.

We also define a scalar reward function $r : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$.

We use the term policy rollout to refer to initializing the system to some state s_0,

[1]Although in general a case one might have a discrete set for their states and actions, in the rest of this thesis, for simplicity and brevity, we will assume states, actions, and rewards are all real numbers.

and then letting it evolve under policy π_θ. We can define the return R, as the sum of rewards from a given rollout:

$$R_\theta = \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}). \tag{2.3}$$

The goal is to find a set of weights θ^* that maximize our expected reward:

$$\theta^* = \arg\max_\theta \mathop{\mathbb{E}}_\eta \left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \right]. \tag{2.4}$$

Taken together, the transition function, the reward function, and the sets of possible states and actions define a Markov Decision Process (MDP).

Value and Q Functions

In reinforcement learning it is often useful to examine so-called value functions. The value function $V_\pi : \mathbb{R}^n \to \mathbb{R}$ gives us the expected return from a given initial state assuming actions are taken according to the policy π. Thus

$$V^\theta(s) = \mathop{\mathbb{E}}_\eta \left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) | s_0 = s \right]. \tag{2.5}$$

Related to this is the optimal value function, V^*, which is the expected return if the system is initialized in state s_0 and then takes actions according to the optimal policy, π^*:

$$V^*(s) = \max_\theta \mathop{\mathbb{E}}_\eta \left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) | s_0 = s \right]. \tag{2.6}$$

The Q function is very similar. It describes the expected return if we initialize the system to state s_0, take action a_0, which may or may not come from the current policy, and then forever after take actions according to the policy described by θ:

$$Q^{\theta} = \mathop{\mathbb{E}}_{\eta}\left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a\right]. \tag{2.7}$$

The optimal Q function is the same thing but using the optimal policy rather than the current policy.

$$Q^{*}(s) = \max_{\theta} \mathop{\mathbb{E}}_{\eta}\left[\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a\right] \tag{2.8}$$

2.2 Parameterized Functions and Neural Networks

In the context of robot learning for control, we are usually studying so-called parameterized control policies. In addition to normal inputs and outputs, these functions also have an additional input θ, which is a vector of the that defines the function. Though strictly speaking a parameterized function is a function of both θ and x, we often put the parameters in the subscript. Let's look at a simple example, a third order polynomial with coefficients θ_0, θ_1, and θ_2:

$$y = f_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2. \tag{2.9}$$

At risk of belaboring the point, the equation above has an input (x), and an output (y) but also parameters (θ). When we speak of learning or training a policy, we are trying to find a set of parameters which produce a function that maximizes our expected return.

Static Linear Policies

The first and most straightforward policy class we refer to often in this thesis are "static linear policies". In this case the vector θ parameterizes an n x m weight matrix,

W:

$$W = \begin{bmatrix} \theta_1 & \theta_2 & \cdots & \theta_m \\ \theta_{m+1} & \theta_{m+2} & \cdots & \theta_{2m} \\ \vdots & & & \\ \theta_{m(n-1)} & \theta_{m(n-1)+1} & \cdots & \theta_{m*n} \end{bmatrix}. \tag{2.10}$$

This defines a deterministic policy, the action is computed with a simple matrix multiply with the weight matrix:

$$a_t = W^\top s_t. \tag{2.11}$$

This policy is static in the sense that the weight matrix does not change with each time step. It is linear because each element of the action vector is a linear combination of the state vector. As discussed in the literature review, these policies are surprisingly expressive, capable of producing complex gaits in legged robotic systems. In the context of modern reinforcement learning, these sorts of linear policies are usually trained with some type of direct policy search, like ARS or ES, however there is also absolutely nothing stopping you from applying back propagation to train a linear policy, just like we do for neural networks.

Neural Networks

One of the most successful and popular class of policies, especially in recent years, have been neural networks [2]. Neural networks can trace their history to the very dawn of computing, and were inspired by biological neurons. However for our purposes we can focus on the types of networks commonly used today, and consider them a class of

[2]You will often hear neural networks called function approximators, which was indeed their original purpose. In the context of function approximation, we have data from an unknown function, often sampled from a real world system, and we wish to solve for parameters such that the output of our function closely matches the output of the target function. We will use neural networks for this in Chapter 3, but otherwise use them as policies.

parameterized policies.

Modern neural networks come in many flavors, from the basic multi-layer perceptrons (MLPs), to the deep convolutional neural networks (CNNs) used for computer vision, to the long short-term memories (LSTM) and Gated Recurrent Units (GRUs) that are common in natural language processing. When these networks are large, we call them "deep" neural networks, although even networks with only two or three layers are often called deep.

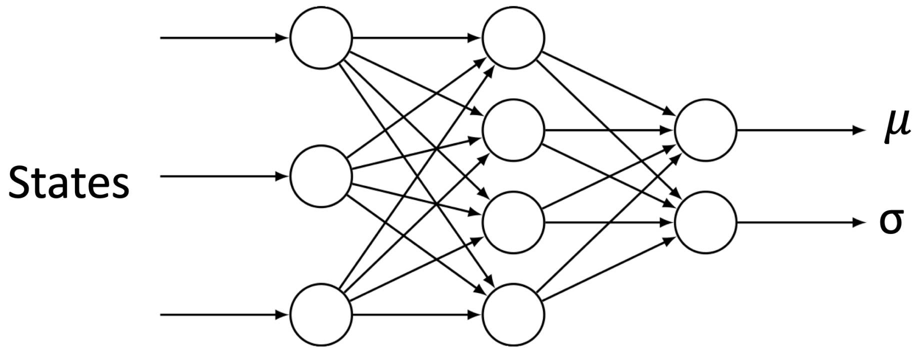


Figure 2.2: A stochastic MLP policy. The input is the state at time t, and the outputs are the mean and standard deviation for a Gaussian distribution. This distribution is then used from to generate the control action.

Let's examine the MLP in particular, which is the most common for control applications. We can see a diagram for an MLP in Figure 2.2. The MLP, as the name suggests, consists of multiple layers. The output for each layer is as follows:

$$x_l = \sigma(W_l^\top x_{l-1} + b_l), \tag{2.12}$$

with x_{l-1} being the output of the previous layer (with x_0 being the initial input), W_l being the weight matrix for layer l, b_l being a vector of bias terms for layer l, and σ being some nonlinear activation function.

For most but not all modern RL, the mathematical object we are parameterizing is actually a probability over actions rather than a direct mapping to actions. So if we have an action space of size three then our network may have 6 outputs, 3 means and 3 standard deviations which themselves parameterize a Gaussian distribution. When it is time to select an action, we sample from this distribution.

2.3 Deep Learning and Supervised Learning

If reinforcement learning is learning by trial and error, supervised learning is learning by example. The most famous examples are probably classifying images, for example deciding if a given image contains either a dog or cat. In the context of deep learning, we are training a deep neural network, typically with a process called backpropogation. A full explanation of backpropogation is outside the scope of this thesis, but it is sufficient to say that back-prop is an algorithm that allows for the efficient training of differentiable, parameterized functions.

In supervised learning, rather than maximizing a reward, we are attempting to minimize some loss function. This could be a simple L2 norm, but a more common loss in practice is the cross-entropy loss, which is used for classification tasks. This loss is designed to reduce classification error in supervised learning tasks. The binary cross-entropy loss, which will also be used in Chapter 3, is shown below. Here,

$$L^{\mathrm{G}} = \mathop{\mathbb{E}}_{\gamma} - \left[c_w y_i \log(G_\gamma(s_i)) + (1 - y_i)\log(1 - G_\gamma(s_i))\right], \tag{2.13}$$

where y_i is the class label for the ith sample, and c_w is a class weight for positive examples. We set $c_w = \frac{n_t}{n_p} w$ where n_t is the total number of samples, n_p is the number of positive examples, and w is a manually chosen weighting parameter.

2.4 Modern Reinforcement Learning

Reinforcement learning can be divided into model-based and model-free algorithms. Model-based algorithms typically learn a model of the system they are trying to control, and then use that model to do planning or model predictive control. A popular example is the PILCO algorithm [54]. These algorithms certainly have their place; however, they are typically limited to relatively low-dimensional systems, and they often require expensive planning at run time. This thesis focuses on model-free algorithms, which are more popular and have seen more development over the past decade.

Model-free algorithms, despite their name, can and do develop models of the system they are controlling, especially in the form of Q and value functions. The difference is that they do not attempt to use these models to do any sort of planning. Instead, they are directly optimizing a parameterized policy which will (indirectly) induce some trajectory, not planning a trajectory and then working backwards.

2.4.1 Off-Policy RL

We can further divide modern reinforcement learning into on-policy and off-policy algorithms. Off-policy algorithms train the current policy using data obtained from previous policies. This usually takes the form of a ”replay buffer” which stores state, action, and reward tuples obtained so far during training. These tuples can then be used to update the current policy. Off-policy algorithms are often a variant of Q learning, since Q functions can be learned with actions drawn from any distribution, not just the current policy. DQN, DDPG, TD3, and SAC are all examples of off-policy reinforcement learning [14] [16] [23] [55]. In general, off-policy algorithms tend to be more sample efficient, i.e., requiring fewer samples to obtain a similar reward level. At the same time they also tend to be less stable, meaning it is more likely for these algorithms to see big dips in performance while training, and for algorithms using different random seeds to get drastically different results.

Soft Actor-Critic

Soft actor-critic (SAC) is an off-policy deep reinforcement learning algorithm shown to do well on control tasks with continuous actions spaces [24]. To aid in exploration, rather than directly optimize the discounted sum of future rewards, SAC attempts to find a policy that optimizes a surrogate objective:

$$J^{\text{soft}} = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R_t + \alpha H(\pi(\cdot|s_t)) \Big)\right], \tag{2.14}$$

Where H is the entropy of the policy.

SAC introduces several neural networks for the training. We define a soft value function $V_\phi(s_t)$, a neural network defined by weights ϕ, which approximate J^{soft} given the current state. Next we define two soft Q functions, $Q_{\rho_1}(s_t, a_t)$ and $Q_{\rho_2}(s_t, a_t)$, which approximate J^{soft} given both the current state and the current action. Using two Q networks is a trick that aids the training by avoiding overestimating the Q function. We must also define a target soft value function $V_{\overline{\phi}}(s_t)$, which follows the value function via Polyak–Ruppert averaging:

$$V_{\overline{\phi}^+}(s_t) = c_{py} V_{\overline{\phi}}(s_t) + (1 - c_{py}) V_\phi, \tag{2.15}$$

with c_{py} being a fixed hyper parameter. We also define Π_θ, a neural network that outputs $\mu_\theta(s_t)$ and $\log(\sigma_\theta(s_t))$ which theen define the probability distribution of our policy π_θ. The action is given by:

$$a_t = \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t)\epsilon_t), \tag{2.16}$$

where ϵ_t is drawn from $N(0, 1)$.

SAC also makes use of a replay buffer D that stores the tuple (s_t, a_t, r_t) after policy rollouts. When it is time to update we sample randomly from this buffer, and use those samples to compute our losses and update our weights.

With this we can define the losses for each of these networks (originated from [24]). The loss for our two Q functions is:

$$L^Q = \mathbb{E}_{s_t,a_t\sim D}\left[\frac{1}{2}\left(Q_\rho(s_t,a_t) - \hat{Q}(s_t,a_t)\right)^2\right], \tag{2.17}$$

where

$$\hat{Q}(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}}\left[V_{\bar{\phi}}(s_{t+1})\right]. \tag{2.18}$$

Our policy seeks to minimize:

$$L^\pi = \mathbb{E}_{s_t\sim D,\epsilon_t\sim N(0,1)}\left[\log\pi_\theta(f_\theta(\epsilon_t,s_t)|s_t) - Q_{\rho_1}(s_t,f_\theta(\epsilon_t,s_t)\right], \tag{2.19}$$

and our value function is:

$$L^{\mathrm{V}} = \mathop{\mathbb{E}}_{s_t\sim D}\left[\frac{1}{2}\left(V_\phi(s_t) - \hat{V}_\phi(s_t)\right)^2\right], \tag{2.20}$$

Where

$$\hat{V}_\phi = \mathbb{E}_{a_t\sim\pi_\theta}\left[Q^{min}(s_t,a_t) - log\pi_\theta(a_t|s_t)\right], \tag{2.21}$$

and $Q^{min} = \min(Q_{\rho_1}(s_t,a_t), Q_{\rho_2}(s_t,a_t))$.

SAC starts by doing policy rollouts, recording the state, action, reward, and the active controller at each time step. It stores these experiences in the replay buffer. After enough trials have been run, we run our update step. We sample from the replay buffer, and we use these sampled states to compute the losses above. We then run one step of Adam [56] to update our network weights. We repeat this update n_u times with different samples. Finally, we copy our weights to our target network and repeat until convergence (or some other stopping metric).

2.4.2 On-Policy RL

On-policy algorithms use only data obtained with under the current policy to make updates. After an update, these methods essentially throw out the data obtained so far and start afresh. Although this causes on-policy algorithms to be less sample efficient, they also tend to be more stable. Additionally, they tend to be much faster at updating policies. In contexts where sampling from the environment is very fast, on policy algorithms are often faster than off policy, despite worse over all sample efficiency. Examples include the classic REINFORCE, A2C, TPRO and PPO [11] [57] [17] [21].

Policy Gradients

Many on-policy algorithms are a variant of the classic policy gradient algorithm. In general, we wish to use stochastic gradient descent to train our network by maximizing the gradient of our policy parameters with respect to our reward function:

$$\theta^{+} = \theta + \alpha \nabla_\theta R(\theta). \tag{2.22}$$

Typically, the gradient $\nabla_\theta R(\theta)$ is not available to us (though in Chapter 7 we will examine what happens when we *do* have such gradients), so that we must therefore approximate it in some way. Using the so-called log derivative trick, we can show that the reward gradient is equal to the following:

$$\nabla_\theta R(\theta) = \mathop{\mathbb{E}}_{\eta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) r_t \right]. \tag{2.23}$$

This expression can then be approximated by sampling from the environment. Any algorithm which is using this update rule can be considered a policy gradient algorithm.

2.4.3 Gradient-Free Algorithms

Although less common, there are some reinforcement learning algorithms which make no use of gradients at all. Examples include ES and ARS [25] [46]. These algorithms are essentially using the update rule from Equation 7.9, but rather than using the policy gradient, they instead approximate the reward gradient with finite differences. These methods have the advantage that they are easy to understand, and can scale easily to hundreds of CPUs/GPUs running in parallel.

Augmented Random Search

In contrast to other modern RL algorithms, Augmented Random Search (ARS) is an extremely minimalist algorithm. In [46], the authors showed that linear policies and a simple direct random search over policy parameters was sufficient to obtain state-of-the-art results for the OpenAI Mujoco locomotion tasks. In addition to this, work by [47] also showed that simpler policy classes, linear and radial basic function (RBF) policies, combined with a simple natural policy gradient algorithm, could also compete with state-of-the-art RL. There is also [48], which showed that linear policies could be applied to a complex locomotion task on a real robot.

A version of this algorithm is presented in Algorithm 8. We call this version Random Policy Search. It is very similar to ARS, but lacks the normalization of states. Essentially at each time step, we sample noise vectors from a normal distribution and add them to our current policy to get several candidate policies. We then do rollouts with the candidate policies, and collect their total rewards. These rewards are then used to update the current policy.

Algorithm 1 Random Policy Search

```
Require: Policy $\pi$ with trainable parameters $\theta$
Require: Hyper-parameters - $\alpha$ $\sigma$ $n$
1: Sample $\boldsymbol{\delta} = [\delta_1, ..., \delta_n]$ from $\mathcal{N}(0, \sigma)^{\mathrm{n\ x}\,|\theta|}$
2: $\theta^* = [\theta - \delta_1, ..., \theta - \delta_n, \theta + \delta_1, ..., \theta + \delta_n]$
3: for $\theta_i$ in $\theta^*$ do
4:     Do rollout with policy $\pi_{\theta_i}$
5:     Collect sum of rewards $R_i$.
6: $\theta^+ = \theta + \frac{\alpha}{n\sigma_R} \sum_{i=0}^{n} (R_i - R_{i+n})\delta_i$
```

Chapter 3

Switching

3.1 Introduction

As we have discussed, deep reinforcement learning has been used recently to solve very challenging problems in robotic control. Despite this, these same algorithms can struggle on certain low dimensional problems from the nonlinear control literature, e.g., the acrobot [58] and the cart-pole pendulum. These are both underactuated mechanical systems that have unstable fixed points in their unforced dynamics (see Section 3.2). Typically, the goal is to bring the system to an unstable fixed point and to keep it there. In this paper, we focus on the acrobot as we found fewer examples in the existing literature of model-free reinforcement learning to perform well on this task.

It is not uncommon to see some variation of these systems tackled in various reinforcement benchmarks, but we have found these problems have usually been artificially modified to make them easier. For example, the very popular OpenAI Gym benchmarks [19] includes an acrobot task, but the objective is only to get the system in the rough area of the unstable (fully upright) fixed point, and the dynamics are integrated with a fixed time-step of 0.2 seconds, which makes the problem much easier and unrepresentative of

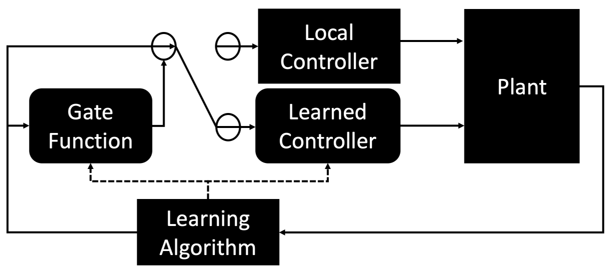


Figure 3.1: System diagram for the new technique proposed in this paper. Rounded boxes represent learned neural networks, square-cornered boxes represent static, hand-crafted functions. The local controller is a hand-designed LQR, the swing-up controller is obtained via reinforcement learning, and the gating function is trained as a neural network classifier.

a physical system. We have found that almost universally, modern model-free reinforcement learning algorithms fail to solve a more realistic version of the task. Notably, the DeepMind control suite [20] includes the full acrobot problem, and all but one algorithm that they tested (the exception being [59]) learned nothing, i.e., the average return after training was the same as before training.

Despite this, there are many traditional model-based solutions [58], [60], that can solve this problem well. In this, work we do not seek to improve upon the model-based solutions to this problem, but instead to extend the class of problems that model-free reinforcement learning methods can be used to solve. We believe the methods used here to solve the acrobot can be extended to other problems, such as making robust walking policies.

One of the primary reasons why this underactuated problem is difficult for RL is that the region of state space that can be brought to the unstable fixed point over a small time horizon is very small, even with generous torque limits. An untrained RL agent explores by taking random actions in the environment. Reaching the region of attraction is correspondingly rare. We found that for our system, random actions will reach the basin of attraction for a well-designed LQR (linear quadratic regulator) controller in about 1% of trials. However, an RL agent doesn't have access to a well-designed LQR at the start of training. Instead, in addition to reaching the region where stabilization via linearization of the system is possible, the agent must also then stabilize the acrobot for the agent to receive a strong reward signal. This results in successful trials in this environment being extremely rare, and therefore training is infeasibly slow and sample inefficient.

Our solution is to add a predesigned balancing controller into the system. This is comparatively easy to design, and can be done with a linear controller. Our contribution is a novel way to combine this balancing controller with an algorithm that is learning the swing-up behavior. We simultaneously learn the swing-up controller, and a function that switches between the two controllers.

3.1.1 Related Work

Work done by Randolov et al. [61] is closely related to our own. In that work, the authors construct a local controller, an LQR, and combine it with a learned controller to swing-up and balance an inverted double pendulum. Their choice of system is similar to the acrobot we study, but, importantly, it is fully actuated, with actuators at both joints. Another difference between our work and theirs is that they hard code the transition between their two controllers. In contrast, we learn our transition function online and in parallel with our swing-up controller.

Work done by Yoshimoto et al. [62], like ours, learns the transition function between controllers in order to swing-up and balance an acrobot. However, unlike our work they limit the controllers to switch between two pre-computed linear functions. In contrast, our work simultaneously learns a nonlinear swing-up controller and the transition between a learned and pre-computed balance controller.

Wiklendt et al. [63] also achieve swing-up and balance an acrobot using a combined neural network and LQR. However, they only learn to swing-up from a single initial condition, whereas our method learns a full control *policy*, to solve the task from any initial configuration of θ_1 and θ_2 (at zero velocity).

Doya [64] also learns many controllers using reinforcement learning, and then adaptively switches between them. However, unlike our work, the switching function is not learned using reinforcement learning, but is instead selected according to which of the controllers currently makes the best prediction of the state at the current point in state space. We believe our model free updates will avoid the model bias that can be associated with such approaches. Furthermore, our work allows for combining learned controllers with hand-designed controllers, such as LQR.

3.2 The Acrobot System

The acrobot is described in Figure 3.2. It is a double inverted pendulum with a motor only at the elbow. We use the following parameters, from Spong [58]:

Parameter	Value	Units
m_1, m_2	1	Kg
l_1, l_2	1	m
l_{c1}, l_{c2}	.5	m
I_1	.2	Kg*m^2
I_2	1.0	Kg*m^2

The state of this system is $s_t = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]^\intercal$. The action $a_t = \tau_t$, is the torque at the elbow joint. The goal we wish to achieve is to take this system from any random initial

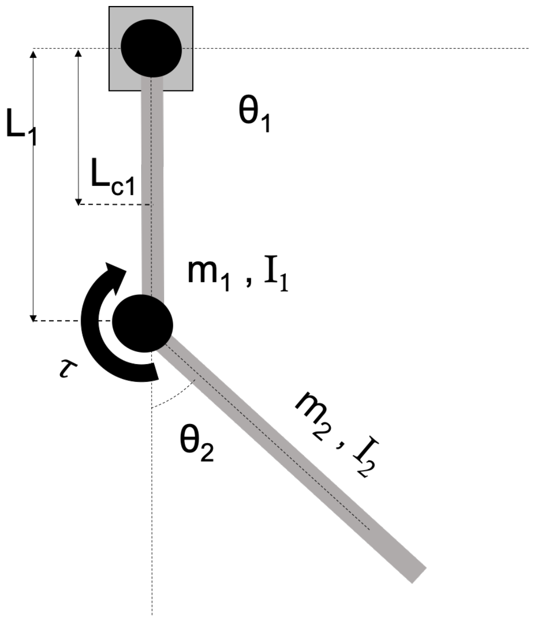


Figure 3.2: Diagram for the acrobot system

state to the upright state $gs = [\pi/2, 0, 0, 0]^{\mathsf{T}}$, which we will refer to as the goal state. To achieve this goal, we seek to maximize the following reward function:

$$r_t = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2), \tag{3.1}$$

which is, geometrically, just the vertical height of the tip of distal link. This choice was motivated by the popular Acrobot-v1 environment [65]. Empirically, using this reward signal for our algorithm led to the same solutions as did the more typical $r_t = -\|s_t - gs\|$. However, some of the other algorithms we against which we benchmarked performance (see Table 3.1) perform better with the reward function in Eq. 3.1.

We implement the system in Python (all source code is publicly available[1]), the dynamics are implemented using Euler integration with a time-step of 0.01 seconds, and the control is updated every 0.2 seconds. We experimented with smaller time steps and higher-order integrators, as well. In general, these modifications made the balancing task easier, but made the wall clock time for the learning much slower.

[1]Source code for this work can be found here: https://github.com/sgillen/ssac

3.3 Switched Soft Actor Critic

Our primary contribution is to extend SAC in two key ways. We call our modified algorithm switched soft actor critic (SSAC). The first modification is a change to the structure of the learned controller in order to inject our domain knowledge into the learning. Our controller consists of three distinct components: the gate function, the swing-up controller, and the balancing controller. The gate, $G_\gamma : S \to [0, 1]$, is a neural network parameterized by weights γ which takes the observations at each time step and outputs a number $g_t \in [0, 1]$ representing which controller it thinks should be active. $g_t \approx 1$ implies high confidence that the balancing controller should be active, and $g_t \approx 0$ implies the swing-up controller is active. This output is fed through a standard switching hysteresis function, to avoid rapidly switching on the class boundary. Switching parameters are given in the appendix on page 42. The swing-up controller can be seen as the policy network from vanilla SAC, with the action then determined by Equation 2.16. (The parameters for these networks are also given in the same appendix.) The balancing controller is a linear quadratic regulator (LQR), $C : S \to A$ about the acrobot's unstable equilibrium. We use the LQR designed by Spong [58], i.e., with

$$Q = \begin{pmatrix} 1000 & -500 & 0 & 0 \\ -500 & 1000 & 0 & 0 \\ 0 & 0 & 1000 & -500 \\ 0 & 0 & -500 & 1000 \end{pmatrix}, \qquad R = \begin{pmatrix} .5 \end{pmatrix}.$$

The resulting state feedback control law is of the form

$$u = -Ks,$$

with

$$K = [-1649.8, -460.2, -716.1, -278.2].$$

These three functions (gate function, swing-up controller, and balancing controller) together form our policy, π_θ. Algorithm 3 demonstrates how the action is computed at each time step.

We learn the basin of attraction for the regulator by framing it as a classification problem: our neural network takes as input the current state, and it outputs a class prediction between 0 and 1. A one implies that the LQR is able to stabilize the system, and a zero implying that it cannot. We then define a threshold function $T(s)$, as a criteria for what we consider a successful trial:

$$T(s) \; = \|s_t - gs\| < \epsilon_{thr} \quad \forall t \in \{N_e - b, ..., N_e\}. \tag{3.2}$$

Here, s is understood to be an entire trajectory of states, N_e is the length of each episode, e_{thr} and b are hyperparameters with values given in the appendix. We are following the convention of a programming language here, where (3.2) returns one when the inequality holds and zero otherwise. To gather data, we sample a random initial condition, do a policy rollout using the LQR, and record the value of 3.2 as the class label.

To train the gating network we minimize the binary cross entropy loss:

$$L^{\mathrm{G}} = \mathop{\mathbb{E}}_{\gamma} - \left[c_w y_i \log(G_\gamma(s_i)) + (1 - y_i)\log(1 - G_\gamma(s_i)) \right], \tag{3.3}$$

Where y_i is the class label for the ith sample, and c_w is a class weight for positive examples. We set $c_w = \frac{n_t}{n_p} w$ where n_t is the total number of samples, n_p is the number of positive examples, and w is a manually chosen weighting parameter to encourage learning a conservative basin of attraction. We found that the learned basin was very sensitive to this parameter; a value of 0.01 empirically works well. Note that unlike the other losses above, the data here are not computed over a single sample but are instead computed over the entire replay buffer. We found that the gate was prone to "forgetting" the basin of attraction early in the training, otherwise. This also allows us to update the gate infrequently, when compared to the other networks, so that the total impact on wall clock time is modest.

The second extension is a modification of the replay buffer D. We do this by constructing D from two separate buffers, D_n and D_r. Only rollouts that ended in a successful balance (as defined by Equation (3.2)) are stored in D_r. The other buffer stores all trials, the same as the unmodified replay buffer. Whenever we draw experience from D, with probability p_d we sample from D_n, and with probability $(1 - p_d)$ we sample from D_r. We found that this sped up learning dramatically, as even with the LQR and a decent gating function in place, the swing-up controller finds the basin of attraction only in a tiny fraction of all trials.

Algorithm 2 Warm Start Gate Data Generation

```
1: Initialize network weights $\gamma$
2: for $i \in \{1, ..., N_d\}$ do
3:     sample initial state $s_0$ from observation space
4:     if $s_i \in B$ then
5:         $y_i = 0$
6:     else
7:         $y_i = 1$
8: set $\alpha(0) = 1$, and $\alpha(1) = \frac{\text{sum(y)}}{\text{len(y}}$
9: update $G_\gamma$ with $n_w$ steps of Adam to minimize $L^{ws}$
```

3.4 Results

3.4.1 Training

To train SSAC we first start by training the gate, exclusively, using the supervised learning procedure outlined in Section 3.3. This allows us to form an estimate of the basin of attraction before we try to learn to reach it. We trained the gate for 1e6 time

Algorithm 3 Do-Rollout(G_γ, Π_θ, K)

```
1:  $s = r = a = g = r = \{\}$
2:  Reset environment, collect $s_0$
3:  for $t \in \{0, ..., T\}$ do
4:      $g_t = hyst(G_\gamma(s_t))$
5:      if $(g_t) == 1$ then
6:          $a_t = -Ks_t$
7:      else
8:          Sample $\epsilon_t$ from $N(0,1)$
9:          $a_t = \beta \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t) * \epsilon_t)$
10:     Take one step using $a_t$, collect $\{s_{t+1}, r_t\}$
11:     $s = s \bigcup s_t$, $r = r \bigcup r_t$
12:     $a = a \bigcup a_t$, $g = g \bigcup g_t$
13: return $s, a, r, g$
```

Algorithm 4 Switched Soft Actor Critic

```
1:  Initialize network weights $\theta, \phi, \gamma, \rho_1, \rho_2$ randomly
2:  set $\overline{\phi} = \phi$
3:  for $n \in \{0, ..., N_e\}$ do
4:      $s, r, a, g =$ Do-Rollout$(G_\gamma, \Pi_\theta, K)$
5:      if $T(s)$ then
6:          Store $s, r, a$ in $D_n$
7:      Store $s, r, a$ in $D_r$
8:      Store $s, g, T(s)$ in $D_g$
9:      if Time to update policy then
10:         sample $s^r, a^r, r^r$ from $D$
11:         $\hat{Q} \approx R + \gamma V_{\overline{\phi}}(S)$
12:         $Q^{min} = \min(Q_{\rho_1}(s^r, a^r), Q_{\rho_2}(s^r, a^r))$
13:         $\hat{V} \approx Q^{min} - \alpha H(\pi_\theta(A|S))$
14:         Run one step of Adam on $L^Q(s^r, q^r, r^r)$
15:         Run one step of Adam on $L^\pi(s^r)$
16:         Run one step of Adam on $L^V(s^r)$
17:         $\overline{\phi} = q\overline{\phi} + (1-q)\phi$
18:     if Time to update gate then
19:         Run one step of Adam on $L^G$ using all samples in $D_g$
```

steps, and then trained both, together in parallel, using Algorithm 2 for another 1e6 time steps. The policy, value, and Q functions are updated every 10 episodes, and the gate every 1000. The disparity is because, as mentioned earlier, the gate is updated using the entire replay buffer, while all the other losses are updated with one sample batch from the buffer. Hyperparameters were selected by picking the best performing values from a manual search, which are reported in the appendix.

In addition to training on our own version of SAC and Switched SAC we also examined the performance of several algorithms written by OpenAI and further refined by various contributors on Github [66]. We examine PPO and TRPO, two popular trust region methods. A2C was included to compare to a non-trust-region, modern policy gradient algorithm. We also include TD3, which has been shown in the literature to do well on the acrobot and cart-pole problems [16].

Stable baselines includes hyperparameters that were algorithmically tuned for each environment. For algorithms where parameters for Acrobot-v1 were available, we chose those. Some algorithms did not yet have parameters tuned for Acrobot-v1, and for those we used parameters for Pendulum-v0, simply because it is another continuous, low-dimensional task. Note that we do not expect the hyperparameters to impact the learned policy's score in this case, but instead only how fast learning occurs. Reported rewards are averaged over 4 random seeds. Every algorithm makes 2e6 interactions with the environment. Also note that this project was in fact largely inspired by previously having spent a large amount of time *manually* tuning these parameters to work on this task[2], previous to developing the SSAC approach described here. Figure 3.3 shows the reward curve for our algorithm and the algorithms from stable baselines. Table 3.1 shows the mean and standard deviation for the final rewards obtained by all algorithms.

As we can see, for this environment, and with the number of steps we have allotted,

[2]...and with no success better than what we see here!

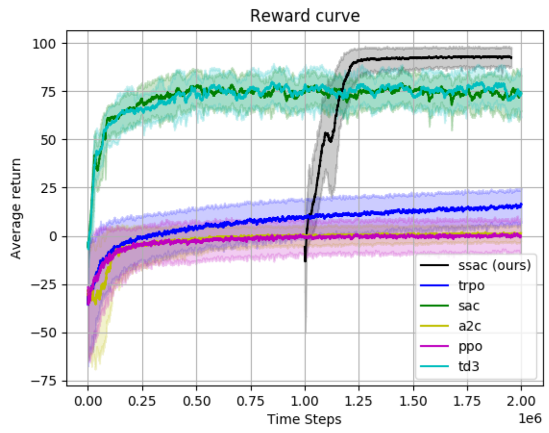


Figure 3.3: Reward curve for SSAC and the other algorithms against which performance is compared. The solid line is the smoothed average of episode reward, averaged over four random seeds. The shaded area indicates the best and worst rewards at each epoch across the four seeds. SSAC is shown starting later to account for the time training the gating function alone.

Algorithm (implementation)	Mean Reward ± Standard Deviation
SSAC (Ours)	**92.12 ± 2.35**
SAC	73.01 ± 11.41
PPO	0.43 ± 8.89
TD3	78.67 ± 61.85
TRPO	17.63 ± 3.39
A2C	2.57 ± 3.63

Table 3.1: Rewards across learning algorithms, after 2 million environment interactions

our approach outperforms the other tested algorithms, with TD3 making it the closest to our performance. This is a necessarily flawed comparison. These algorithms are meant to be general purpose, so it is unfair to compare them to something designed for a particular problem. But that is, in fact, part of the point we are making, i.e., that adding just a small amount of domain knowledge can improve performance dramatically.

3.4.2 Analyzing performance

To qualitatively (and, perhaps, more intuitively) evaluate the performance of our learned agent, beyond just the scalar reward function, we examine the behavior during individual episodes. SSAC also gives us a deterministic controller, as we can set ϵ_t from 2.16 to zero. We did so, chose the initial condition $s_0 = (-\pi/2, 0, 0, 0)$ and recorded a rollout. The actions are displayed in Figure 3.4, and the positions in Fig. 3.5.

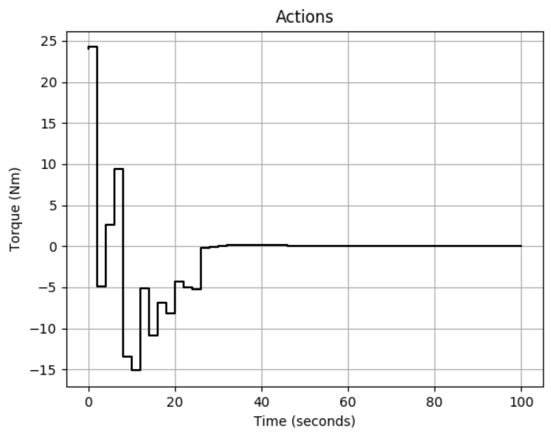


Figure 3.4: Torque exerted during the sampled episode

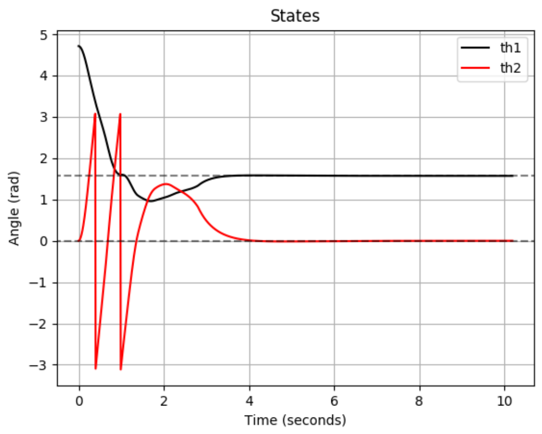


Figure 3.5: Observations during the sampled episode

By comparison, we notes that despite achieving relatively high rewards, the other algorithms we compare to often fail to meet the balance criteria (Eq. 3.2). We often see solutions where the first link is constantly rotating, with the second link constantly vertical. To demonstrate this, as well as to demonstrate the robustness of our SSAC algorithm, we ran rollouts with the trained agents across a grid of initial conditions, recording if the trajectory unltimately satisfies Eq. 3.2 or not. For brevity, we compare our method only with TD3 here, as this was the best performing model-free method we could find on this task. Figure 3.6 show the results for TD3. **When these initial conditions were run for SSAC, it satisfied Eq. 3.2 for *every* initial condition**.

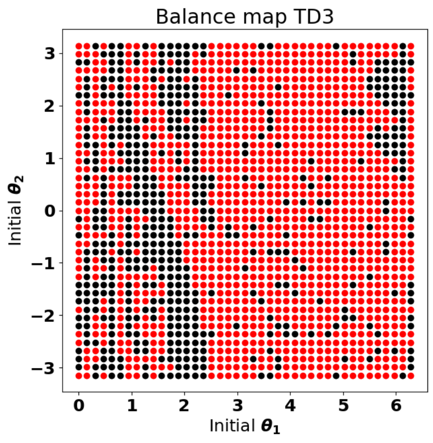


Figure 3.6: Balance map for TD3, X and Y indicate the initial position for the trial, a black dot indicates that the trial started from that point satisfies Equation (3.2), and red indicates the converse. **When these initial conditions were run for SSAC, the balancing condition was met for every initial condition.**

3.5 Conclusions

We have presented a novel control design methodology that allows engineers to leverage their domain knowledge while simultaneously reaping many of the benefits from recent advances in deep reinforcement learning. In our case study, we constructed a policy for swing-up and balance of an acrobot while only needing to manually design a linear controller for the balancing task (in terms of domain knowledge). We believe this method of control will be straightforward to apply to the double or triple cart-pole problems, for which, to our knowledge, no model-free algorithm is reported as solving. We also think that this general methodology can be extended to more complex problems, such as legged locomotion. In particularly, legged locomotion also involves multi-link systems that are underactuated and need to balance upright. In such a case, the baseline controller here could be a nominal walking controller using partial feedback linearization to track references obtained via trajectory optimization, and the learned controller could be a recovery controller to return to the basin of attraction of this nominal controller.

APPENDIX

Hyperparameters

Hyperparameter	Value
Episode length (N_e)	50
Exploration steps	5e4
Initial policy/value learning rate	1e-3
Steps per update	500
Replay batch size	4096
Policy/value minibatch size	128
Initial gate learning rate	1e-5
Win criteria lookback (b)	10
Win criteria threshold (ϵ_{thr})	.1
Discount (γ)	.95
Policy/value updates per epoch	4
Gate update frequency	5e4
Needle lookup probability p_n	.5
Entropy coefficient (α)	.05
Polyak constant (c_{py})	.995
Hysteresis on threshold	.9
Hysteresis off threshold	.5

Network Architechture. The policy, value, and Q networks are each made of four fully connected layers, with 32 hidden nodes and ReLU activations. The gate network is composed of two hidden layers with 32 nodes each, also with ReLU activations, the last output is fed through a sigmoid to keep the result between 0-1.

Chapter 4

Reward Post-Processing and Mesh Dimensions

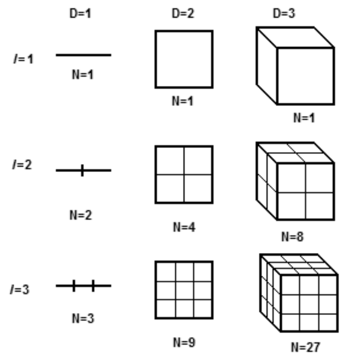
4.1 Introduction

As we have discussed, modern RL algorithms have a serious draw back in that they are mostly black boxes. It is an open challenge to figure out what exactly it is that your RL agent has learned. If all you know is that one of your agents achieved very a high reward, it is not clear how to verify that this system is safe and sensible in all the regions of state space it will visit during its life. Nor can we necessarily say anything about the stability or robustness properties of the system. Recent work [67] has used so-called mesh-based tools to examine precisely these questions by approximating the nonlinear dynamics within state space with a mesh of discrete points and a set of mappings between them over time.

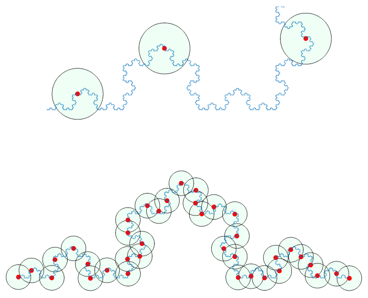
However, utility of any mesh-based tools to accurately discretize a state space is limited, due to the curse of dimensionality. In practice, these methods are only able to work on relatively high dimensional systems if the *reachable* state space grows at a rate that is much smaller than the exponential growth of the full state space the system within which it is embedded. To expand these methods to higher dimensional systems, we will need to find ways to keep the "volume" of visited states from expanding commensurately. One way to quantify this rate of growth is by using one of the several notions of "fractional dimensionality" from fractal geometry.

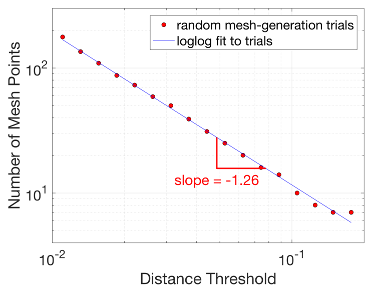
In this chapter, we discuss an efficient meshing algorithm, which we call box meshing. We show that this approach makes calculating the so called mesh dimension feasible in the context of reinforcement learning. We also propose using other notions of fractional dimension from the literature as a proxy for the property we care about. We then show that reinforcement learning agents can be trained to shrink these measures by post-processing their reward function. We present the results of this training, and also present some brief analysis of the resulting structure for select policies.

4.2 Meshing & Fractional Dimensions



(a) Mesh scaling in different dimensions


(b) A non-uniform mesh of a fractal structure



(c) Estimation of the mesh dimension

Figure 4.1: Image credit for sub-figure a: [68], image credit for sub-figures b and c: [69]

Let's say we have a continuous set S that we want to approximate by selecting a discrete set M composed of regions in S. We will call this set M a mesh of our space.

Figure 4.1(a) shows some examples of this: a line is broken into (1D) segments, a square into (2D) grid spaces, and so on. The question is: as we increase the resolution of these regions, how many more regions N do we need? Again, Figure 4.1(a) shows us some very simple examples. For a D dimensional system, if we go from regions of size d to d/k, then we would expect the number of mesh points to scale as $N \propto k^D$. But not all systems will scale like this, as 4.1(b) and 4.1(c) illustrate. Figure 4.1(b) is an example of a curve embedded in a two dimensional space"; namely, this depicts part of the Koch snowflake fractal pattern. The question of how many mesh points are required must be answered empirically. Going backwards, we can use this relationship to assign a notion of "dimension" to the curve.

$$D_f = -\lim_{k \to 0} \frac{\log N(k)}{\log k} \tag{4.1}$$

What we are talking about is called the Minkowski–Bouligand dimension, also known as the box counting dimension. This dimension need not be an integer, hence the name "fractional dimension". As a practical matter, we can use the slope of the log-log plot of mesh sizes over d to estimate this value. This is one of many measures of "fractional dimension" that that emerged from the study of fractal geometry. Although these measures were invented to study fractals, they can still be usefully applied to non-fractal sets.

In [70], Saglam and Byl introduced a technique that is able to both build a non-uniform mesh of a reachable state space and develop robust policies for a bipedal walker on rough terrain. Having a discrete mesh allows for the use of value iteration, to select among a set of candidate controllers, toward finding a robust switching control policy. In addition, this mesh allows for the construction of a state transition matrix, which can be used to calculate the **mean first passage time** [71], a.k.a the mean time to failure, as a metric that quantifies the expected number of steps a metastable system can take before falling.

Since its introduction, in improving and quantifying robustness of biped walking to terrain height variation, meshing in this fashion has also been used for designing walking controllers robust to push disturbances [69], to design agile gaits for a quadruped [72], and to analyze hybrid zero dynamics (HZD) controllers [73]. There has also been recent work to use these tools to analyze policies trained by deep reinforcement learning [67]. A long term goal and motivation for this work is to take a high performance controller obtained via reinforcement learning, and to extract from it a mesh-based policy that is both explainable and amenable to robustness analyses.

4.2.1 Box Meshing

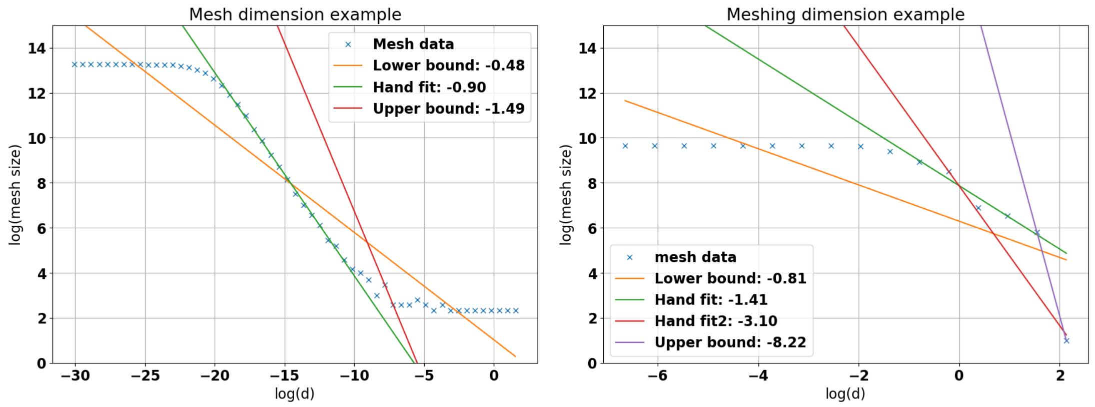
Our primary improvement to the prior work on meshing itself is to introduce something we call **box meshing**. Prior, a new mesh point could take any, arbitrary value in the state space, based on detection of any new reachable state that was not currently "close enough" to existing mesh points, given tolerance settings. To determine if a new state is already in the mesh, we would compute a distance metric to every point in the mesh, and check if the minimum was below our threshold. Thus, building the mesh was an $O(n^2)$ algorithm. By contrast, in box meshing we a priori divide the space uniformly into boxes with side length d. We identify any state s with an indexing key. This key is obtained by first normalizing each element of s by the standard deviation of that dimension, based on all data points, to create $\bar{s}$. Then: key $= \text{round}(\frac{\bar{s}}{d})d$, where round performs an element-wise rounding to the nearest integer. (See Algorithm 5.) We can then use these keys to store mesh points in a hash table. Using this data structure, we can still store the mesh compactly, only keeping the points we come across. However, insertion and search are now $O(1)$, and so building the mesh is $O(n)$. This is very similar to non-hierarchical bucket methods, which are well-studied spatial data structures [74], although we are using them for data compression here. In the prior meshing work, this sort of speedup would be minor, as the run-time is dominated by the simulator or robot. However, this speedup does open some new possibilities: most poignantly, it makes calculating the mesh dimension during reinforcement learning plausible.

Algorithm 5 Create Box Mesh, see Sections 4.2.1-4.2.3

```
1: **Input:** State set $S$, box size d.
2: **Output:** Mesh size m.
3: **Initialize:** Empty hash table M.
4: **for** s $\in$ S **do**
5:     $\bar{s}$ = Normalize(s)
6:     key = round($\bar{s}$ / d)d
7:     **if** s $\in$ M **then**
8:         M[key]++
9:     **else**
10:        M[key]=1
11: **Return:** M
```

4.2.2 Algorithmic Box Mesh Dimension

The "box mesh dimension" is the quantity extracted from the slope of a log-log plot of mesh size vs d value, as depicted in Fig. 4.1(c). In our work, we estimate this slope based on analyzing a large but finite number of states visited over time by the agent during learning. The log-log plot for a very large data set tends to take on a particular form, illustrated in Figure 4.2(a). As the box edge size d gets very small, nearly every point falls into its own, unique box. This results is the flat, saturated portion of the curve at the upper left of the subplot. As d gets very large, the curve tends to flatten out at the bottom right, as well. We hypothesize this second saturation can happen when, for a learned gait-like locomotion behavior, all the points at a particular time step in



(a) Curve of a well behaved policy (b) Curve of a typical run-time policy

Figure 4.2: Mesh curve and mesh dimension examples

the gait tend to fall within the same box with high probability, yielding nearly the same number of mesh boxes across a range of values of d. Correspondingly, these two flat areas are expected numerical artifacts of the curve that we hope to “ignore”, and the mesh dimension should ideally be based only on the well-behaved flat region near the middle of the curve. This has been done by hand in Figure 4.2(a) by selected only a subset of the range of d and fitting the log-log data to a line.

As one might expect, automatically computing the mesh dimension of an arbitrary data set generated from a learning agent can be quite challenging in terms of speed and accuracy. A single trajectory provides a relatively small amount of data, essentially making mesh size calculation a noisy estimation process. Agents attempting to learn legged locomotion also might fall over, generating extremely short trajectories, or they might learn a trajectory that ”stands in place”, which could lead to numerical errors.

Finally, there is a clear trade-off *between* accuracy and speed. Model-free RL is predicated on having a huge number of rollouts to learn from, and we would therefore like for any mesh-dimension quantification algorithm to be fast enough so as to not dominate the total learning time. With these factors in mind, we introduce two box mesh dimensions. The *lower mesh dimension* is based on the linear fit of all of the log-

log curve data, so that it intentionally errs on the side of including any flat parts of the graph, therefore tending to underestimate the true mesh dimension. We also calculate the *upper mesh dimension*, which takes the largest slope of any two neighboring points in the log-log relationship, thus tending to overestimate the true mesh dimension. Neither of these measures are correct, but taken together they provide goods bounds on the mesh dimension, and as we will see they can be useful on their own.

4.2.3 Mesh Dimension Examples

Figure 4.2 illustrates two examples of the curves used to compute the mesh dimension. Recall that to compute the mesh dimension, we choose several values for d, the box length, and for each d construct a mesh using that box size. The x axis of these plots represents the log of the box length used, the y axis represents the log of size of the mesh created. For each curve, we display the lower bound and upper bound for the dimension as computed by Algorithm 6, as well as example attempts to "hand fit" the data. Subplot 4.2a illustrates what a close-to-ideal situation looks like, in addition to providing intuition as to why the upper and lower mesh dimension bound the quantity we are trying to measure. Subplot 4.2b serves to illustrate some of the problems with making an algorithmic measure of the dimension. Here, there is much less data to work with, as illustrated by the difference in magnitude of the maximum log(mesh size) at the upper left of each subplot. This performance constraint in turn causes the estimate of the mesh dimension to be noisy. Indeed, even fitting this data by hand becomes a challenge! We provide two fits which can both be argued to be "plausibly correct". The true value we would have in the limit for a larger data set remains unknown, and the range between upper and lower mesh dimensions also becomes more dramatic here, indicative of a large uncertainty.

Algorithm 6 Compute Box Mesh Dimension, see Sections 4.2.2-4.2.3

```
1: Input: State set $S$
2: Output: Mesh M.
3: Hyperparameters: scaling factor f, initial box size $d_0$.
4: Initialize: Empty list of mesh sizes H, empty list of d values D.
5: m = Size(CreateBoxMesh(S, $d_0$))
6: d = $d_0$
7: Append m to H, append d to D.
8: while m < size(S) do
9:     d = d/f
10:    m = Size(CreateBoxMesh(S, d))
11:    Prepend m to H, prepend d to D.
12: while m $\neq$ 1 do
13:    d = d*f
14:    m = Size(CreateBoxMesh(S,d))
15:    Append m to H, append d to D.
16: X = log d
17: Y = -log m
      Lower Mesh Dim: fit Y = gX + b, Return: g
18: Upper Mesh Dim: w = greatest slope in Y over X Return: w
```

4.2.4 Variation Estimators

Consider the more general case of a set of data, X, collected at equally-spaced moments in time. For example in our work, $X = S$ (i.e., the "state set"), with each particular element X_i representing the state vector **s** in state space at time stamp i, and with a total of n points recorded at equal spacing over time. As discussed, computing the mesh dimension (which, as previously discussed, is our hash-table inspired version of the *box dimension*) in an automatic way is fraught with peril, in many practical scenarios. Fortunately, there are also various other metrics one might consider to give various approximations to the fractional dimension we seek to estimate. Gneiting et al. [75] compare a number of these estimators and propose that the **variation estimator** [76] offers a very good trade off between speed and robustness. To obtain this estimator, first define the *power variation* estimate of order p, $\hat{V}_p$ as:

$$\hat{V}_p(X,l) = \frac{1}{2}\mathbb{E}\left|X_i - X_{i-l}\right|^p = \frac{1}{2(n-l)}\sum_{i=l}^{n}\left|X_i - X_{i-l}\right|^p. \tag{4.2}$$

Then, the *variation estimator* of order p is:

$$Dv_p(X) = 2 - \frac{1}{p}\frac{\log V_p(X,2) - \log V_p(X,1)}{\log 2}, \tag{4.3}$$

The **madogram** estimator is the special case of (4.3) where $p = 1$, and for the **variogram**, $p = 2$.

4.2.5 Post-processing Rewards

In order to influence the dimensionality of the resulting policies, we introduce various post-processors, which act on the reward signals before passing them to the agent. These obviously modify the problem: in some sense the post-processed environment is a completely different problem from the original. However our meta-goal is to train agents that achieve reasonable rewards in the base environment, while simultaneously exhibiting the reduced dimensionality we are looking for. These post-processors take the form:

$$R_*(\mathbf{s},\mathbf{a}) = \frac{1}{D_*(\mathbf{s})}\sum_{t=0}^{T} r(s_t, a_t, s_{t+1}), \tag{4.4}$$

where $(\mathbf{s},\mathbf{a})$ are understood to be an entire trajectory of state action pairs, and D_* is some measure of fractional dimension. Various measures of dimensionality can be inserted here directly (e.g., see 4.2.4).

However, the mesh dimensions computed by algorithm 6 require a little more care.

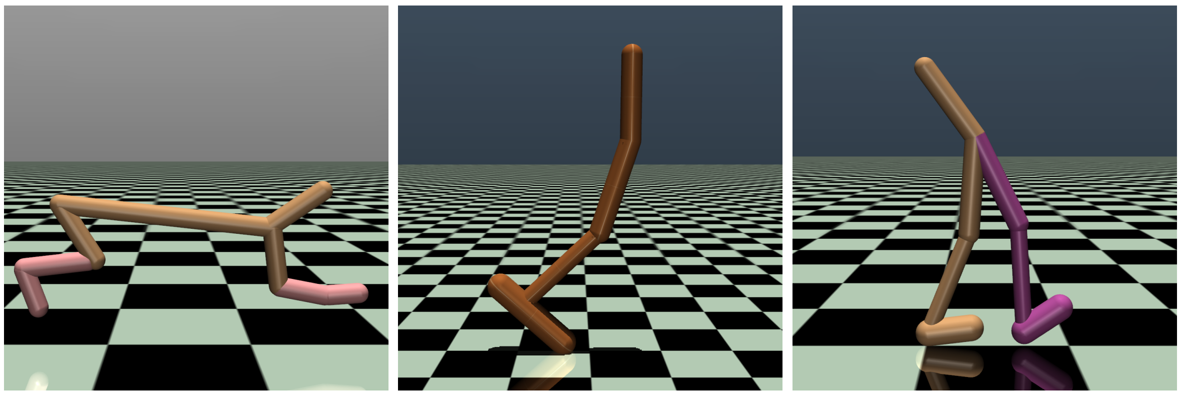
We must first define a *clipped dimension*:

$$D_*^c = \text{clip}[D_*(\mathbf{s}_{t>Tr}), 1, D_t/2)], \tag{4.5}$$

where D_t is the topological dimension, equal to the number of states (i.e., for our work, the position and velocity variables) in the system. T_r is a fixed number of timesteps chosen to exclude the initial transients resulting from a system moving from rest to into a quasi-cyclical "gait". In this paper we set $T_r = 200$ for all experiments. For comparison, the nominal episode length is 1000 timesteps. The clipping is intended to minimize the influence of pathological trajectories the RL agent might generate while not interfering dramatically with the overall training. Additionally, it also weeds out trajectories that terminate very early, to prevent agents learning to fall over immediately to "game the system". Using half of the topological dimension (i.e., $\frac{1}{2}D_*$) proved to be a decent upper bound for the worst case dimensionality of each system in practice. The **mesh dimension post-processors** use the clipped dimension. Finally, in order to benchmark against a "no post-processing" comparison, we additionally train with a fictitious value of $D_* = 1$, and we call those non-post-processed results the **identity post-processor**, since in this case the total reward is completely unchanged.

4.2.6 Environments

We examine a subset of the popular OpenAI Mujoco locomotion environments introduced in [19]. In particular, we evaluate our work on HalfCheetah-v2, Hopper-v2, and Walker2d-v2. These environments were chosen because they have a relatively high dimensionality, i.e., 11-17 degrees of freedom (DOF), with twice that number of states (including both position and velocity of each DOF), since our goal is to demonstrate that mesh-based approaches are feasible even as dimensionality grows. The state space



(a) HalfCheetah-v2 (b) Hopper-v2 (c) Walker2d-v2

Figure 4.3: The Mujoco locomotion environments

consists of all joint / base positions and velocities, with the x (the ”forward”) position being held out, because we want a policy that is invariant along that dimension.

4.2.7 Augmented Random Search

In [46], Mania et al. introduce Augmented Random Search (ARS), which proved to be efficient and effective on simulated locomotion tasks. Rather than a neural network, ARS uses static linear policies, and compared to most modern reinforcement learning, the algorithm is very straightforward. The algorithm operates directly on the policy weights. In each epoch, the agent perturbs its current policy N times and collects $2N$ rollouts (i.e., using both positive and negative policy deviation by this perturbation vector) of modified policies. The rewards from these rollouts are then used to update the current policy weights, with the process repeating until arriving adequately close to some locally optimal solution. The algorithm is known to have high variance, so that not all seeds obtain high rewards, but to our knowledge this work in many ways represents the state of the art on the benchmark environments described in Sec.4.2.6. Mania et al. introduce several small modifications of the algorithm in their paper, and our implementation corresponds to the version they call ARS-V2t.

4.2.8 Training

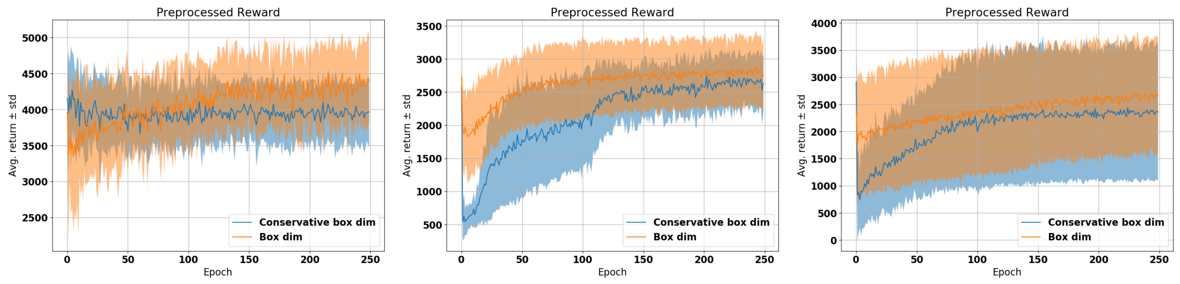
We chose parameters that were found to work well across all environments, using the parameters reported in Table 9 from [46] as a starting point. We then tuned until our unprocessed learning achieved satisfactory results across all tasks. Again, ARS is known to have high variance between random seeds, indicating high variance among local minima for ARS policies, so that some seeds never learn to gather a large reward. The parameters we found are able to consistently solve the Cheetah and Walker environments. For the hopper, the algorithm learns a policy with high reward[1] roughly half the time. This seems consistent with the performance reported in [46]. We train each post-processor on 10 random seeds. The evaluation metrics are averages over 5 rollouts from each seed, and for the dimension metrics we use extended episodes of length 10,000 to get more accurate estimates of dimensionality. The reported returns, and the training, both use the normal 1,000 timestep episodes. We found that the mesh post-processors were getting very poor performance when trained from a random policy. However, we found that we saw good results when these trials were initialized with a working policy. Therefore we trained agents for 750 epochs without post-processing, and used that to initialize the policies with mesh dimension post-processing. The mesh policies were then trained for an additional 250 epochs, with the results reported below.

4.3 Results

4.3.1 Mesh Dimension Post-processors

For all environments, the mesh post-processors had a significant impact in reducing the mesh dimensions. It's important to emphasize here that the dimensions reported rep-

[1]The MuJoCo environments reward "forward progress" over time, with a subtracted penalty for squared torque inputs.



(a) HalfCheetah (b) Hopper (c) Walker

Figure 4.4: Reward curves for mesh dimension postproccesor runs.

Environment	Postprocessor	Lower Mesh Dim.	Upper Mesh Dim.	Return
HalfCheetah-v2	Identity	2.31 ± 0.71	7.34 ± 1.56	5469 ± 823
	Lower Mesh Dim	**0.66 ± 0.51**	**2.55 ± 1.52**	4962 ± 598
	Upper Mesh Dim.	**1.06 ± 1.13**	**2.83 ± 1.27**	4432 ± 539
Hopper-v2	Madogram*	1.62 ± .27	4.68 ± 0.82	3461 ± 119
	Lower Mesh Dim.	1.13 ± .02	3.54 ± 0.96	2941 ± 538
	Upper Mesh Dim.	1.27 ± .50	2.98 ± 1.48	3020 ± 337
Walker2d-v2 (walking seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Lower Mesh Dim.	1.21 ± 0.06	4.09 ± 1.03	3339 ± 887
	Upper Mesh Dim.	1.89 ± 0.42	3.10 ± 0.93	3359 ± 903
Walker2d-v2 (all seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Lower Mesh Dim.	1.04 ± 0.53	4.45 ± 1.19	3034 ± 1086
	Upper Mesh Dim.	1.48 ± 0.67	2.27 ± 0.95	2556 ± 1378

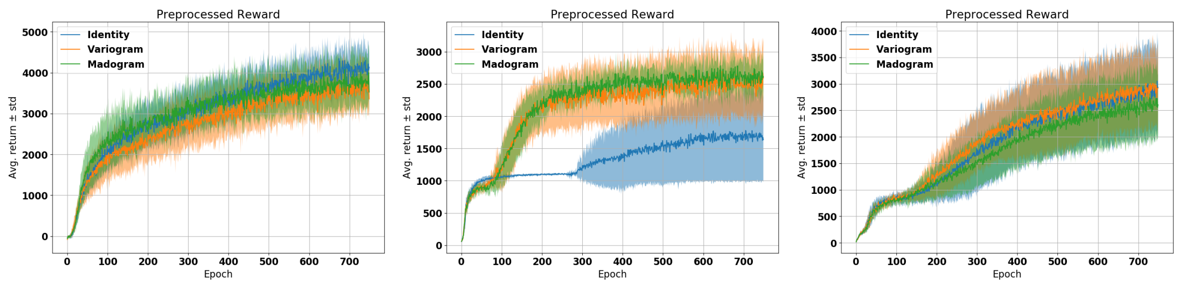
Table 4.1: Mesh dimensions and returns for trajectories after training. See 4.2.8 for details

resent lower- and upper-bound estimates (see Sec. 4.2.2) for the actual mesh dimensions. Although mesh dimension was successfully reduced, there was also a corresponding and statistically significant decrease in the unprocessed reward returns.

However, this work stands as a building block for broader, future aims of addressing the "curse of dimensionality", lowering dimensionality of the reachable state space for a controlled system toward enabling a variety of other numerical techniques to quantify long-term robustness (which is not measured directly by modern-day reward functions). Given our primary goal in this work is to train agents to have an acceptable reward while being more amenable to meshing, we argue the trade-offs between mesh dimensionality and reward are quite good here, overall.

Of note, several seeds (4 for the upper dim., 3 for the lower mesh dim.) for the Walker system “forget how to walk”, instead learning a policy that stands in place. Although this behavior would arguably be likely to have a low dimensionality[2], it is certainly not a very useful behavior for locomotion! For completeness, we include the Walker statistics both from the seeds that learned a gait, and for all 10 seeds including the standing policies.

4.3.2 Variational Post-processors



(a) HalfCheetah (b) Hopper (c) Walker

Figure 4.5: Reward curves for the variation postprocessors

The variational post-processors had a modest effect the variational metrics of dimension, but that did not seem to correlate to a smaller mesh dimension in our experiments, despite what our preliminary tests had led us to hypothesize. The Hopper and Walker had remarkable consistency in the variation dimensions they found. Without running many more trials and hyperparameter sweeps, it is challenging to make broad generalizations; however, our experiments show that 1) measures for fractional dimension can be influenced without adversely effecting the reward, and 2) that it is possible for an agent to reduce the variogram and madogram dimensions of observed trajectories without having

[2]However, this is not guaranteed to result in a lower dimensionality since states are first normalized in performing our “box meshing”.

Env.	Post-proc.	Variogram	Madogram	Lower Mesh Dim.	Return
H-C	Identity	1.71 ± .03	1.42 ± .05	2.36 ± .61	5545 ± 593
	Variogram	1.68 ± .01	1.36 ± .02	2.06 ± .60	5136 ± 851
	Madogram	1.65 ± .02	1.31 ± .04	2.09 ± .64	5234 ± 950
Hop	Identity*	1.61 ± .14	1.22 ± .28	1.03* ± .71	2063 ± 1052
	Variogram	**1.51 ± .02**	**1.03 ± .04**	1.58 ± .54	3299 ± 711
	Madogram	**1.51 ± .002**	**1.02 ± .004**	1.57 ± .36	3449 ± 146
Walk	Identity	1.68 ± .35	1.36 ± .71	2.14 ± .29	3742 ± 1038
	Variogram	**1.54 ± .07**	**1.07 ± .01**	1.85 ± .54	3779 ± 894
	Madogram	**1.53 ± .01**	**1.06 ± .02**	1.99 ± .53	3414 ± 1025

Table 4.2: Mesh dimensions and returns for trajectories after training. Here, the environments ("Env.") are H-Ch: HalfCheetah-v2, Hop: Hopper-v2, and Walk: Walker2d-v2. See 4.2.8 for details
* This includes policies which learned to "stand still", which lowers the average mesh dimension considerably see discussion

a significant impact on its box mesh dimension.

4.4 Analysis

We now examine the learned behavior for one of the more notable policies. The most dramatic effects in Table 4.1 above were for mesh dimension post-processors on the Cheetah. Using either the upper and lower measures of dimension shrunk by 2-4 times. Figure 4.6 presents data for this case.

Toward more intuitively understanding this data, a few comments are worth making, first. We have discussed the mesh dimension rather abstractly so far. In visualizing what this really means, imagine two different gait cycles. In one case, there is a general pattern to the motion, but it wanders in a noisy-looking way, like a "signature" that does not quite match up, cycle after cycle. As motions become closer to being exact limit cycles, there is a more clear pattern of repetition, exactly analogous to re-tracing the same path, again and again, within the state space. Such a more tightly-structured limit cycle nature in turn results in a significantly lower-dimensional set of states being visited.

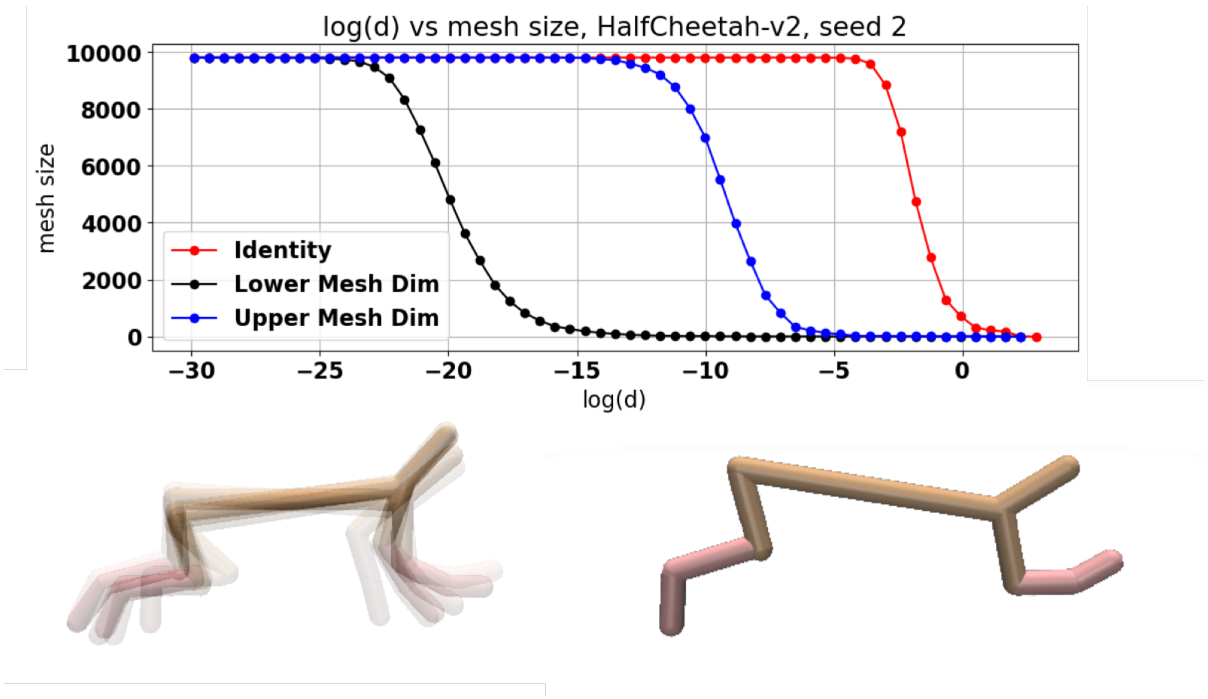


Figure 4.6: Top: mesh sizes vs log of the box size for the cheetah environment. Lower Left: Every five frames overlaid for the an identity policy on the cheetah. Lower Right: Every five frames of cheetah after the lower mesh dimension training.

We can see from the data in Figure 4.6 that there is an overwhelming difference in the mesh sizes between the lower mesh dimension post-processor and the other two. Notice that the axes are semilog (in x) here. The curve for the lower mesh dimension is furthest to the left, so that overall fewer mesh points are needed for values of d. And it transitions more gently in slope, so that the box dimension is also lower.

To put this in perspective, before the extra 250 epochs of training, if given a box size of $d = 0.01$ ($\log(d) \approx -4.6$), the agent would need a unique mesh point for every single point in the 10,000 state trajectory. After the additional training, however, the agent can represent all 10,000 points with just 5 mesh points! In this case it appears both agents learned a quasi-periodic gait that, with the additional training, converged to be almost exactly period-5. In Figure 4.6, we present an overlay of the agents rendered every 5 steps. The results show us that the mesh agent has learned an extremely tight limit

Environment	Postprocessor	Lower Mesh Dim.	Upper Mesh Dim.	Return
HalfCheetah-v2	Identity	2.38 ± 0.43	6.65 ± 1.90	5404 ± 1015
	Lower Mesh Dim	1.51 ± 0.13	3.03 ± 1.09	4952 ± 572
	Upper Mesh Dim.	1.76 ± 0.53	3.54 ± 1.27	4222 ± 803
Hopper-v2	Madogram*	$1.63 \pm .14$	4.49 ± 0.75	3438 ± 185
	Lower Mesh Dim.	$1.67 \pm .22$	3.71 ± 0.89	2943 ± 535
	Upper Mesh Dim.	$1.64 \pm .16$	3.01 ± 1.36	3019 ± 337
Walker2d-v2 (walking seeds)**	Identity	2.13 ± 0.31	4.62 ± 1.03	3758 ± 1037
	Lower Mesh Dim.	1.83 ± 0.34	2.73 ± 0.75	3511 ± 872
	Upper Mesh Dim.	1.60 ± 0.33	4.01 ± 1.18	3384 ± 903
Walker2d-v2 (all seeds)**	Identity	2.10 ± 0.34	4.42 ± 1.00	3743 ± 1034
	Lower Mesh Dim.	1.68 ± 0.70	4.19 ± 1.25	3048 ± 1071
	Upper Mesh Dim.	1.48 ± 0.38	2.98 ± 0.86	2558 ± 1373

Table 4.3: Mesh dimensions and returns for trajectories subject to zero-mean Gaussian noise. Standard deviation of 0.001 and 0.01 was added to all actions and observations respectively. See Sec. 4.2.8 for details.
Because ARS with our chosen hyperparameters does not consistently produce 10 seeds that perform well on the hopper, we instead use madodiv (see the 4.2.4) for the seed policies.
** See Sec. 4.3.2

cycle. It's a bit of a strange limit cycle, being only 5 time steps long, but nonetheless we think this is interesting and surprising behavior.

The behavior displayed in Figure 4.6 is clearly something that can only happen in a noiseless simulation, so we also measured the mesh dimensions of our policies when subjected to noise during rollouts. Table 4.3 shows these results. The difference in the fractional dimension is less pronounced than for the "no noise" case, but there is still a clear improvement for the post-processor cases. Furthermore, we anticipate (intuitively) that if we were also to add noise at training time, the learned policies might have been able to lower the mesh dimension more significantly for these sorts of post-training trials with noise.

It's worth noting at this point that in practice the lower mesh dimension seems to work better than the upper one. We found, when computing the mesh dimension by hand (by hand fitting a line to a set of carefully obtained mesh size data), that the hand

Case	Env.	Identity	Lower mesh dim.	Action std
(a)	Cheetah	0.24	0.05	.05
	Hopper	0.19	0.10	.05
	Walker	0.28	0.03	.15
		Identity	Lower mesh dim.	Observation std
(b)	Cheetah	0.20	0.02	.005
	Hopper	0.20	0.25	.02
	Walker	0.18	0.10	.03
		Identity	Lower mesh dim.	Magnitude, Rate
(c)	Cheetah	0.21	0.03	3, .2
	Hopper	0.17	0.10	1, .2
	Walker	0.20	0.00	1, .2

Table 4.4: Failure rates for agents under various noise and push disturbances

picked value was generally much closer to the lower mesh dimension, at least for the three systems we studied. Training with the lower mesh dimension also resulted in agents that were more robust and achieved higher reward compared to the upper dimension.

To quantify robust with versus without post-processing, we tested three different cases, adding zero-mean Gaussian noise either (a) to the actions, (b) to the observations, or (c) via an external force disturbance directed at the center of mass of the agents during their rollouts. Table 4.3 shows the fraction of of runs resulting in a “failure”, e.g., early termination of a 10,000 time step episode, for example due to tripping and falling. For the push disturbances we have two parameters, the rate of disturbances, and the magnitude of the force applied. At every step we sample uniformly from [0,1], if the result is less than the rate parameter, then a force is applied at that time step. The force is applied at a random angle in the xz plane (each environment is a planar system), with the fixed magnitude from the magnitude parameter. For each type of disturbance, we did a grid search over the parameters and report the parameter for which the identity post processor failed in roughly 20 percent of cases, so that values are close to 0.2, by design.

4.5 Conclusion

In this work, we introduced a technique to influence the fractional dimension of the closed-loop dynamics of a system through the use of novel, dimensionality-based modifications to the cost functions for reinforcement learning policies. We demonstrated this technique on several benchmark tasks, and we briefly analyzed a resulting policy to verify the outcome, demonstrating a much smaller mesh dimension without a large loss in reward or function.

Hyper Parameters

ARS: For all environments $\alpha = .02$, $\sigma = .025$, $N = 50$, $b = 20$.

MeshDim: f = 1.5, d_0 = 1e-2

4.5.1 Implementation Details

For performance reasons, the mesh dimension algorithm does not actually create meshes until the mesh size equals the total data size, but rather until the mesh size is 4/5 the total data size. Figure 4.6 shows a typical mesh curve, and we can see the long tail of values, at the upper left portion of each curve, with mesh sizes close to the maximum value. Not much useful information is gained from this and it wastes time, so we stop early. We do not place the same limitation on the lower size of the mesh, since typically the mesh size hits one much more rapidly. Figure 4.6 illustrates this, too. In addition, we set a minimum size for $d_{min} = 1e-9$ in this work to avoid numerical errors.

The normalization done during box creation (see Alg. 5) uses a running mean and standard deviation of all states seen so far during training. These stats are saved and used for evaluation as well. We found that the upper mesh dimension is very sensitive to the normalization used, but that the other metrics where not.

Chapter 5

Reachable State Space Mesh for the Hopper

We have already discussed some prior work using mesh-based techniques to analyze control policies for robotic systems [67]. Broadly, these techniques take a dynamic system with a continuous state space and approximate it with a discrete set of states. This set forms a non-uniform "mesh" within the full state space. The underlying dynamics might be deterministic or stochastic, yielding either deterministic or probabilistic mappings between the discrete states as an approximate model of the true system dynamics.

In other words, this technique allows us to model the dynamics system as a Markov chain, or, in a case where we have a discrete number of controllers or control actions possible, as a Markov Decision Process (MDP). In either case, this opens up a new box of tools we can bring to bear on the problem. For example, we can use value iteration to switch between several controllers to improve the robustness [69] or agility [72] of the system. We can also perform eigen-analysis on the Markov chain's transition matrix, which provides us insights on the stability of the system [71]. These techniques could both be used for policy refinement, and/or for verification and analysis of existing policies.

Unlike the previous section, where we considered meshes of a single trajectory, this prior work that studies a systems reachable state space mesh. That is, the set of states that can be reached with a fixed controller, and a known set of disturbances. Although in the previous section, each trajectory was encouraged to have a smaller fractal dimension, this does not obviously extend to properties of the entire reachable state space for the system, which is what was used for the previous mesh based analysis of RL policies.

In this work we take the next step and construct reachable state space meshes of agents trained with and without our modified reward. Our primary contribution is showing that these modified policies result in significantly smaller reachable meshes for a given box size, and in smaller fractal dimensions for the reachable state space. We then use the modified policies to construct a much finer mesh than would be possible otherwise. We use this mesh to compute a quantity called the mean first passage time (MFPT), and validate the obtained MFPT with Monte Carlo trials. Finally we use our mesh to produce interesting visualizations of failure states, which motivates future work.

5.1 The Hopper Benchmark System

Our model system is openAI gym's Hopper-v2 environment introduced in [19]. This environment is part of a popular and standardized set of benchmarking tasks for reinforcement learning algorithms. The system is a 4 link, 6 DOF hopper constrained to travel in the XZ plane, seen in Figure 5.1. The observation space for the agent has 11 states, the position in the direction of motion is held out, since we seek a policy that is invariant to forward progress. The actions in this case are commanded joint torques. The reward function for this environment is simply forward velocity minus a small penalty to actions. Successful controllers in this environment must execute a dynamic hopping motion to move robot along the x axis as quickly as possible. This is clearly a toy problem,

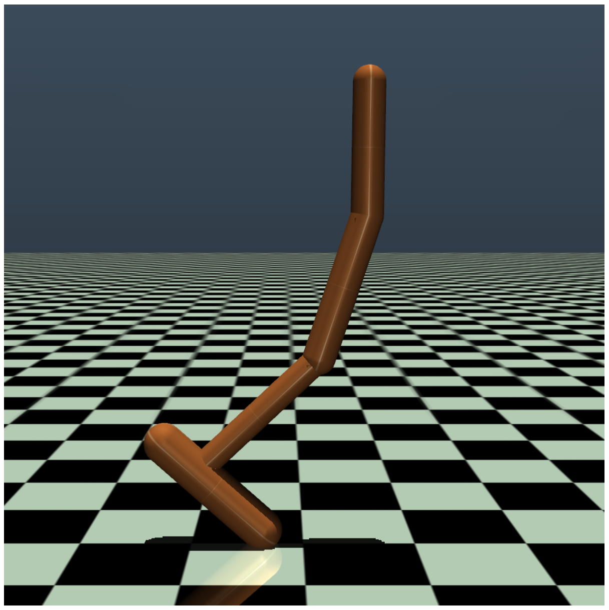


Figure 5.1: A render of the hopper system studied in this work.

but it captures many of the challenges of legged locomotion. The system is highly non-linear, under-actuated, and must interact with friction and ground contacts to maximize it's reward.

5.2 Meshing the Reachable State Space

We are interested in the set of states that our system can transition to with a fixed policy and a given set of push disturbances. We first introduce a failure state to the mesh. The failure state is assumed to be absorbing, once the robot falls it is assumed to stay that way. For our hopper, any state where the COM falls below .7m is considered to have failed, which works well in practice. This is also the failure condition of the environment during training, and therefore the agent is never trained in regions of the state space that satisfy the failure condition.

In addition to the reachable set of states, we want to construct a state to state transition map. That is, for a given initial state, we wish to know which state we transition to for every disturbance in our disturbance set. It's worth emphasizing that this map is completely deterministic.

To make this concrete, recall that we manifest our mesh as a hash table. The key for any given state is obtained by **??**. When we insert a new key into our hash table, the value we place is a pair with a unique state ID (which is simply the number of keys in the table at the time of insertion), and an initially empty list of all mesh states which are reachable after one step from the key state. This data structure will provide both the reachable set, and the transition mapping.

For the hopper in particular, the system transitions from its initial standing position to a stable long term hopping gait. After letting the system enter its gait, we start detecting states on the Poincaré section by selecting the state corresponding to the peak of the base link's height in every ballistic phase. These states are then collected as the initial states to seed the mesh with. Throughout this paper, we seed the mesh with trajectories from 10 initial conditions.

For each snapshot, we initialize the system in the snap-shotted state. For each disturbance in our fixed disturbance set, we simulate the system forward subject to that disturbance. If the system does not fail, then the next Poincaré snapshot is captured, this state is then checked for membership in our mesh. If the new state is already in our mesh, then we simply append the new state to the list of states that the initial state can transition to. If the new state is not already in our mesh, then we expand our mesh to include the new state, and append this new state to the transition list of the initial state. If the system does fail, then we simply append the failure state to the transition list of the initial state, and no new state is added to the mesh.

For every new state added to the mesh, we repeat this process until every state has been explored. Algorithm 7 details this process in pseudo code.

Algorithm 7 createMesh

```
1: Input: Initial states $S_i$, Disturbance set $D$
2: Output: Mesh M.
3: Q ← $S_i$ (excluding the failure state)
4: while Q not empty do
5:     pop q from Q
6:     for d ∈ D do
7:         Initialize system in state q
8:         Run system for one step subject to disturbance d
9:         Obtain final state x
10:        if x ∉ M then
11:            M[x] = List()
12:            Push x onto Q
13:        Append x to M[q]
14: Return: M
```

5.2.1 Stochastic Transition Matrix

The stochastic transition matrix **T** is defined as follows:

$$\mathbf{T}_{ij} = \Pr(id[n+1] = j \mid id[n] = i) \tag{5.1}$$

where $id[n]$ is the index in our mesh data structure of the state at step n. For some intuition, consider the transition matrix as the adjacency matrix for a graph. There is one row/column for every state in our mesh, for a given row i, each entry j is the probability of transitioning from state i to state j. Every row will sum to one, but the sum for each column has no such constraint. After constructing a mesh using algorithm 7, it is straightforward to create the stochastic transition matrix by iterating through every transition list in our mesh.

5.2.2 Mean First Passage Time

We wish to use our mesh based methods to quantify the stability of our system. To do this we estimate the average number of steps the agent will take before falling, subject to a given distribution of disturbances. To do this we will use the so called Mean First Passage Time (MFPT) which in this case will describe expected number of footsteps, rather than the number of timesteps to failure. First recall that our assumption is that our failure state is an absorbing state in our Markov chain approximation, and this implies that the largest eigenvalue of $\mathbf{T}$ will always be $\lambda_1 = 1$. In [71] Byl showed that when the second largest eigenvalue λ_2 is close to unity, the MFPT is approximately equal to:

$$MFPT \approx \frac{1}{(1 - \lambda_2)} \tag{5.2}$$

5.3 Training the Hopper

In [46] Mania et al introduce Augmented Random Search (ARS) which proved to be efficient and effective on the locomotion tasks. Rather than a neural network, ARS used static linear policies, and compared to most modern reinforcement learning, the algorithm is very straightforward. The algorithm is known to have high variance; not all seeds obtain high rewards, but to our knowledge their work in many ways represents the state of the art on the Mujoco benchmarks. Mania et al introduce several small modifications of the algorithm in their paper, our implementation corresponds to the version they call ARS-V2t, hyper parameters are provided in the appendix.

The training process is done in episodes, each episode corresponds to 1000 policy evaluations played out in the simulator. At the start of each episode, the system is initialized in a nominal initial condition offset by a small amount of noise added to each state. During each episode we fix a static policy to let the the system evolve under, we collect the observed state, the resulting action, and the resulting reward at each timestep. This information is then used to update the policy for the next episode.

We compare four different sets of agents trained in different conditions, for each training condition we use training runs across 10 different random seeds. As mentioned ARS is a very high variance algorithm, so a common practice is to run many seeds in parallel and choose the highest performing one. The standard environment has two sources of randomness which are set by the random seed. The first is a small amount of noise added to a the nominal initial condition at the beginning of each episode. The second is noise added to the policy parameters as part of the normal ARS training procedure. Using ARS in the unmodified Hopper-v2 environment will be called the **standard** training procedure. In addition to this, we have a second set of agents which are initialized with the standard training, and then trained for another 250 epochs with the fractal reward function used in equation **??**, these are called the **fractal agents**. Using the standard training agents as the initial policies for the fractal reward was also used in [50], please see that manuscript for more details.

In addition to standard training, we repeat this standard / fractal setup but with the addition of a small amount of zero mean Gaussian noise added to both the states and actions at training time. For brevity we will call these the **Standard noise** and **Fractal noise** scenarios. Hyper parameters for ARS and noise values are reported in the appendix.

5.4 Results

5.4.1 Mesh Sizes Across All Seeds

First we wish to compare the reachable state space mesh sizes obtained for these four different training regiments. For this we assume a disturbance profile consisting of 25 pushes equally spaced between -15 and 15 Newtons, applied for 0.01 seconds along the x axis at the apex of each jump. The goal for this particular exercise is to get an idea of the relative mesh sizes among the different agents across box sizes. Table 5.1 shows these results. We can see that across all box sizes, adding noise at run time decreases the mesh sizes slightly, and that adding the fractal reward training decreases the mesh size even further. The combination of adding noise and the fractal reward seems to perform best at reducing the mesh size.

Training	$d_{thr} = .4$	$d_{thr} = .3$	$d_{thr} = .2$	$d_{thr} = .1$
Standard	64.9	129.0	289.2	2975.2
Standard Noise	40.7	73.3	231.6	2133.3
Fractal	26.0	41.8	67.7	684.4
Fractal Noise	15.1	24.6	45.1	297.2

Table 5.1: Mesh sizes across all seeds for a disturbance profile of 25 pushes. All values are the average mesh size across 10 agents trained with different seeds.

5.4.2 Larger Meshes

With the general trend established, we now take the best performing seed from the noisy training for further study. We chose the seed that had the smallest mesh size from both the standard noise and fractal noise agents.

For this next experiment, we consider a richer distribution of 100 randomly generated push disturbances. These disturbances have a magnitude drawn from a uniform distribution between 5-15 Newtons. This force is applied in the xz plane with an angle drawn from a uniform distribution between 0 and 2π. The number of forces was chosen by increasing the number of forces sampled until the mesh sizes between two random sets did not change. The magnitude of the pushes was chosen arbitrarily, in principle one can use these methods for any distribution of disturbance they expect their robot to encounter during operation.

We then construct meshes for different box sizes. For each agent we construct 10 meshes. We vary the box size between 0.1 and 0.01 for the fractal noise agent. For the standard noise agent we instead vary the box size between 0.1 and 0.02 because the mesh sizes for the standard agent were proving to be too large at the smaller box sizes. Figure 5.2 shows the comparison, We can see clearly that at the very least, the exponential blowup in mesh size starts at much more accurate mesh resolutions for the fractal agent.

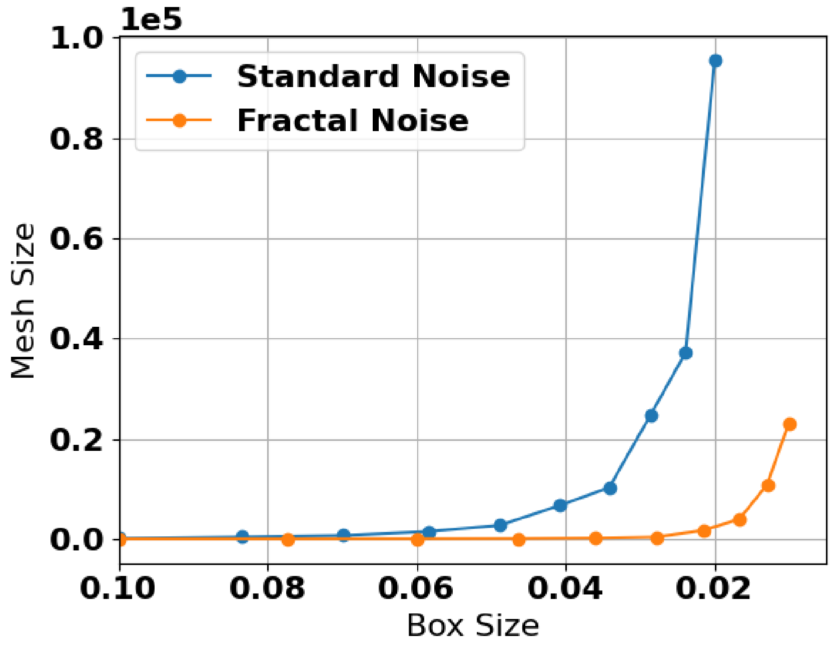


Figure 5.2: Mesh sizes for the top performing standard noise and fractal noise agents.

We are also interested in the exponential scaling factor in the mesh size as the box gets smaller, which is captured by the fractal dimension discussed in section **??**. As mentioned before, in previous work our modified reward signal resulted in agents with a smaller fractal dimension with respect to individual trajectories. We now ask if this carries over to meshes of the reachable state space obtained by the procedure from algorithm 7. Table 5.2 shows the results, we can see that indeed, the fractal training does seem to reduce the mesh dimensionality for the reachable state space meshes.

Training	Trajectory Mesh Dim.	Reachable Mesh Dim.
Standard Noise	1.38	3.83
Fractal Noise	1.16	3.16

Table 5.2: Mesh dimensions for the best performing seed from the standard with noise training, and the fractal with noise training, given the same disturbance profile of 100 pushes. For reference the state space for our system has 12 dimensions.

5.4.3 Validating the Mean First Passage Time

We emphasize that the reward function for the hopper environment is simply to move forward with the highest velocity possible, no attempts were made to make the system robust to disturbances. Perhaps because of this, the mean first passage time for these systems are relatively small, on the order of 100 foot steps. For this small number of steps, we can validate the mean first passage time with Monte Carlo trials. It's worth noting that the eigen estimate of the mean first passage time is much more valuable for more robust systems. This is because this estimate becomes more accurate as the system becomes more stable, and because the cost of calculating the MFPT with Monte Carlo trials grows much more expensive for more stable systems. In previous works [70] it was used to quantify robustness for systems with a MFPT as high as 10^{15}.

To do this, we compare the mean first passage time as estimated by equation 5.2 to the value computed by looking at many Monte Carlo rollouts. For the rollouts we apply a random action drawn from the same distribution described above. Instead of sampling 100 pushes though we sample a new push every time we need a new disturbance. During the rollouts we still apply the push at the apex height of the ballistic phase.

Figure 5.3 shows the convergence of the MFPT as we expand the size of the mesh, and compares it to the mean steps to failure obtained with Monte Carlo trials. We can see that it does look like the MFPT is converging to the Monte Carlo result. Although at the largest mesh we tried, the eigen analysis gives an estimate of 110.2 steps to failure, while the Monte Carlo trials tell us that an average of 85 steps are taken before failure. It's worth noting that the distribution of failure times has a large variance with a standard deviation of 80 steps.

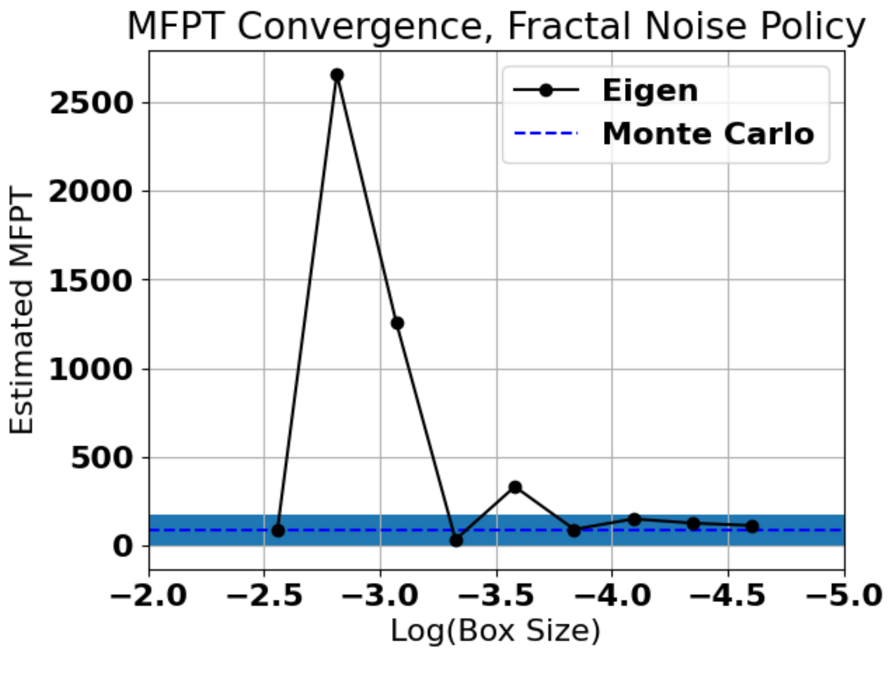


Figure 5.3: Estimated mean first passage time computed from 5.2 compared to a Monte Carlo estimate. The blue dashed line and shaded region are the mean and standard deviation of the steps to failure for 2500 Monte Carlo rollouts.

5.4.4 High Resolution Mesh

We now use the fractal agent and construct an even more accurate mesh. Figure 5.4 show the sparsity pattern for the state transition matrix for the fractal noise agent with a box size of 0.005. Recall that in the process for creating the mesh, we start with a small number initial seed states. After that every new state that we add is added in order we find them to the mesh. So if we are expanding state # 2, and there are currently 100 states in the mesh, if we transition to an unseen state, that state will be labeled # 101. So although it may seem like it is not possible for states in the top right quadrant to visit states later in the mesh, this is really an artifact of how we construct our mesh and label our points.

We note that there are a smaller set of states that make up most of the transitions. In fact we can see from Figure 5.5 that 20% of the states in our mesh account for about 90% of all transitions seen during the mesh construction.

One of the advantages of having a discrete set of states is that it opens up new tools and visualizations, for example we can apply Principle Component Analysis (PCA). Figure 5.6 shows a projection of our mesh states on the top 3 principle components. We note that these three states account for more than 97% of the variance, we also note that our analysis reveals that states in red are where 99% of all failures occur. The visualization reveals that at least in PCA space, all the trouble states are clustered in one spot. A promising direction for future work is to introduce a policy refinement step that attempts to avoid these states. Additionally, if we were designing a real robot this may give us insights into design changes that could be made.

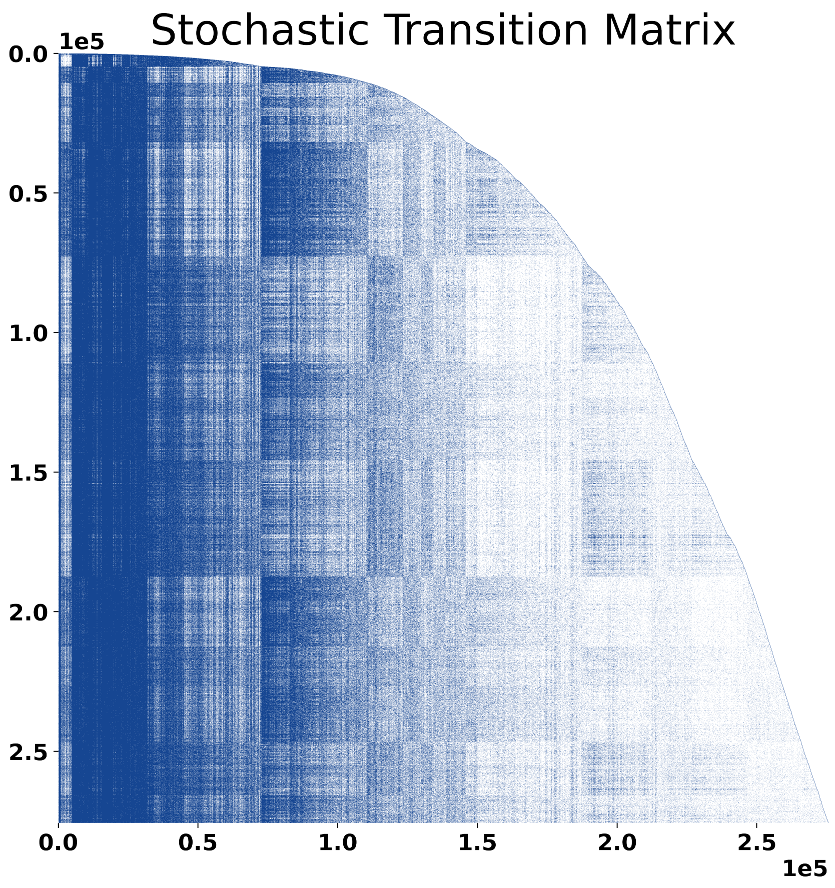


Figure 5.4: Visualization of the stochastic transition matrix for the top performing fractal noise agent. All non zero values are shown with equal size and coloration. Recall that each entry in T_{ij} tell us the probability of transitioning to state j after one step if we start in state i.

5.5 Conclusions

In this work, we apply previously developed tools that create discrete meshes for the reachable state space of a system. These tools were applied to policies obtained with a modified reinforcement learning reward function which was previously shown to encourage small mesh dimensions for individual trajectories not subject to any disturbances. We showed that these modified policies have a smaller average reachable mesh size across all random seeds for coarse meshes and a small number of disturbances. We then showed

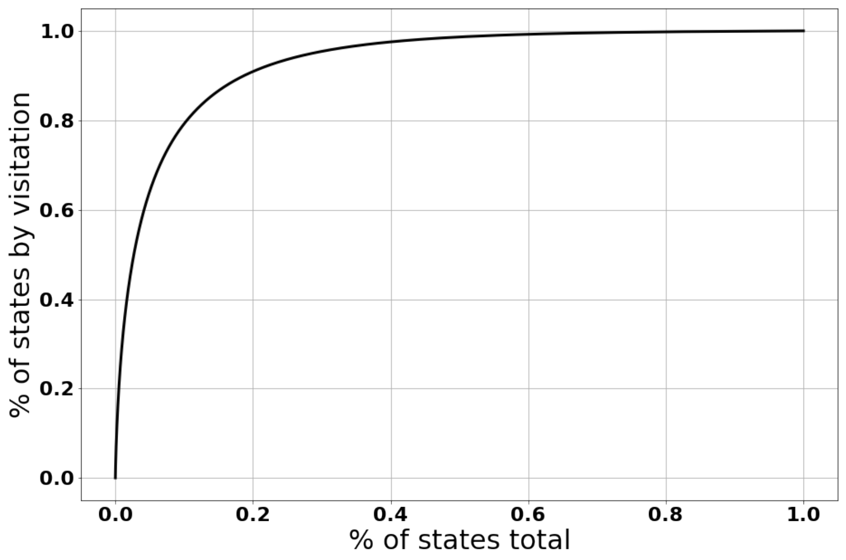


Figure 5.5: Cumulative sum of probability mass excluding the failure state. We take the sum of each column of T, and sort it in descending order, then report the cumulative sum of probability. Each point on the curve tells us that x% of states make up y% of all state transitions.

a clear difference in mesh sizes and mesh dimensions for the top performing seeds on a richer set of disturbances and finer mesh sizes. We also validated our use of the MFPT as a tool by comparing it to Monte Carlo trials. Finally, we constructed a high fidelity mesh at a resolution that would not have been feasible with standard ARS policies. In addition, we created visualizations with this mesh that revealed insights about the contracting nature of the policy, and which point to future applications of this approach. Taken together, these results show two things. First, it further validates the utility of the fractal dimension reward, which we have shown transfers it's desirable quality of having a more compact state space to a setting with external disturbances. These results are also a credit to the mesh based tools, because it shows that the fractal training can be used to extend the reach of these tools to higher dimensional systems or higher resolution meshes than would have otherwise been possible.

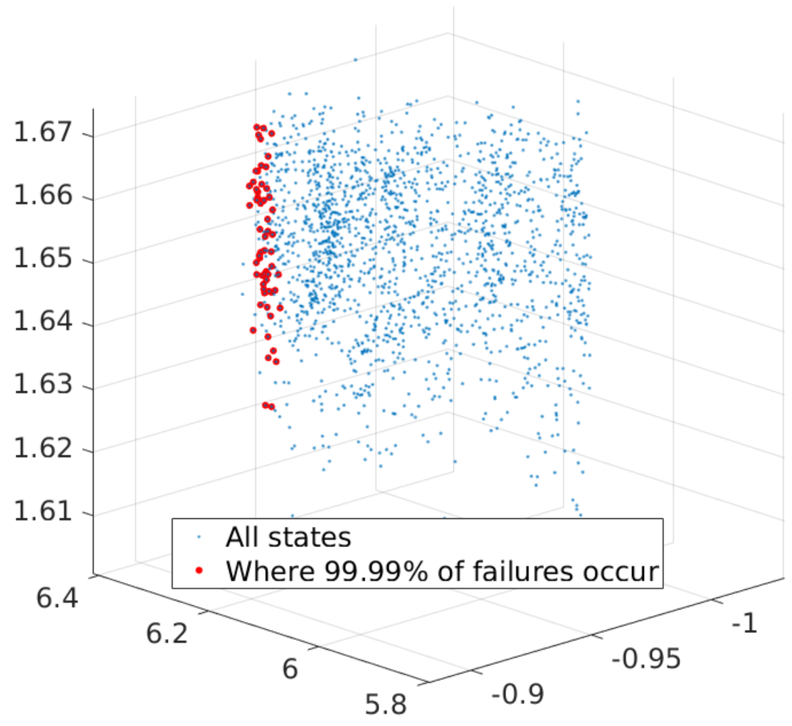


Figure 5.6: View of the first 3 principle components of the mesh for a fractal noise policy.

Noise During Training

Zero mean Gaussian noise with std = 0.01 added to policy actions before being passed to the environment, for reference all actions from the policy are between -1 and 1. Zero mean Gaussian noise with std = 0.001 added to observations before being passed to the policy.

Chapter 6

Fine Tuning of Reinforcement Learning Policies

6.1 Introduction

But, of course, there are significant drawbacks to these model-free approaches. While Deep Neural Networks (DNNs) are very powerful, they also need to acquire a lot of data during training. This contributes to DRL being very sample inefficient, meaning that many interactions with the environment are required in order to find a good policy. As a result, most training for robotic systems must be done in simulation, where the environment can be parellelized and run thousands of times faster than real time. Transfer learning is often required to adapt such policies so that they work for real-world hardware. Doing so effectively remains an important, open problem. Furthermore, modern DRL algorithms can be difficult to implement, as small implementation details can change performance dramatically [77], which motivates our additional focus in reducing the observed variability in performance of closed-loop policies from DRL.

DRL policies are almost always stochastic in nature. During training almost all the common DRL algorithms either add exploration noise to the actions, or learn a probability distribution from which to sample at training time. This might be, for example, a simple Gaussian distribution, the more sophisticated Ornstein-Uhlenbeck correlated noise process in the case of Deep Deterministic Policy Gradient DDPG [16]), or, in the case of DQN, a random selection of sub-optimal actions [15]. However, when policies are deployed or evaluated, one typically uses a deterministic policy by taking what we will call the Maximum Likelihood Action (MLA).

In [46], the authors show that a simple Augmented Random Search (ARS) over linear functions was competitive with deep reinforcement learning across a standard suite of benchmark tasks. Furthermore, this algorithm is simple and, in the cases the authors tested, around fifteen times more sample efficient than the best-performing DRL baseline. Despite these advantages, the simplicity of the policy class limits the environments to which it can currently be applied.

In this work, we show that a slightly modified version of this random search can be applied directly to DNNs for fine tuning, without any apparent loss in sample efficiency. The simplicity of this approach has several advantages. The first is that it does not appear to be very sensitive to hyper-parameter settings. We are able to use a single set of parameters for all the results obtained in this paper, across a dozen environments, and with most systems being tested for six different initial policies each obtained from a different DRL algorithm. Second, we avoid some of the previously mentioned problems stemming from the complexity and fragility of modern DRL algorithms. Finally, our data thus far indicate that we seem to achieve essentially the same sample efficiency seen in ARS, despite operating over much larger parameterizations.

We show that our proposed method of fine tuning leads to modest increases in reward and substantial improvements to consistency in performance for DRL agents across a large set of RL environments. In addition, we also show that we can also use this fine-tuning method to extend previous work of ours involving an extra dimensionality term in the reward [50].

The rest of this paper is laid out as follows. First, we introduce the problem statement, the algorithms and environments used, and implementation details for the training. We then present results obtained from using this policy refinement approach across a collection of continuous-control RL environments. We also perform some analysis on how often fall events occur for a benchmark bipedal walker where the existing DRL baselines are particularly prone to failure. After this, we present results of using this method to train with additional, dimensionality based reward terms in order to show that we are able to extend our previous work to DNN policy classes. Finally, we demonstrate the approach on a Panda arm simulation environment, where our approach leads to considerably smoother policies that avoid unwanted jitter, without any environment specific or algorithm specific tuning or reward shaping.

Environment		A2C	PPO	DDPG	TD3	SAC	TQC
MountainCar	Baseline Return	91 ± 0.2	88 ± 2.3	93 ± 0.0	93 ± 0.1	94 ± 1.3	**67 ± 43.8**
	Tuned Return	92 ± 0.1	96 ± 17.0	94 ± 0.4	94 ± 0.2	95 ± 1.1	**96 ± 0.9**
LunarLander	Baseline Return	61 ± 137.3	273 ± 30.5	216 ± 100.0	**205 ± 86.7**	259 ± 67.8	279 ± 28.6
	Tuned Return	160 ± 126.1	275 ± 32.4	249 ± 68.5	**257 ± 20.1**	283 ± 18.1	286 ± 17.7
BoxWalker	Baseline Return	**296 ± 27.0**	**220 ± 122.4**	**217 ± 127.4**	**302 ± 65.1**	**289 ± 66.0**	**326 ± 58.2**
	TunedReturn	**313 ± 0.7**	**325 ± 0.7**	**281 ± 54.1**	**334 ± 0.6**	**321 ± 1.0**	**344 ± 0.3**
BoxWalkerHard	Baseline Return	99 ± 129.3	137 ± 119.4	N/A	-92 ± 16.3	16 ± 104.2	238 ± 102.0
	Tuned Return	109 ± 121.0	137 ± 119.7	N/A	-23 ± 5.2	44 ± 86.4	242 ± 107.6
Walker2D	Baseline Return	785 ± 389.2	2108 ± 16.0	**1432 ± 720.1**	**2218 ± 194.6**	**2290 ± 34.8**	**2540 ± 557.6**
	Tuned Return	913 ± 269.3	2250 ± 194.1	**1896 ± 375.7**	**2411 ± 7.5**	**2413 ± 13.6**	**2812 ± 8.8**
HalfCheetah	Baseline Return	2109 ± 36.3	2938 ± 53.7	2064 ± 198.7	2820 ± 21.0	2792 ± 10.9	**3676 ± 16.7**
	Tuned Return	2211 ± 35.9	3000 ± 42.3	2264 ± 133.1	2928 ± 15.4	2883 ± 6.9	**3802 ± 11.9**
Hopper	Baseline Return	**834 ± 343.3**	**2523 ± 383.5**	**1179 ± 453.1**	2681 ± 27.2	2602 ± 205.2	**2631 ± 329.7**
	Tuned Return	**1643 ± 204.1**	**2633 ± 91.0**	**2379 ± 341.6**	2749 ± 337.1	2706 ± 96.7	**2782 ± 20.7**
Ant	Baseline Return	2502 ± 25.4	2869 ± 72.7	2365 ± 212.5	**3268 ± 288.8**	3096 ± 31.3	**3478 ± 24.0**
	Tuned Return	2679 ± 28.4	2897 ± 157.0	2424 ± 86.7	**3391 ± 24.8**	3206 ± 18.0	**3654 ± 21.7**

Table 6.1: Average Return ± standard deviation before and after fine tuning

6.2 Methods

6.2.1 Direct Policy Search

We start by noting that there are many names for what we are calling direct policy search, as it is at least 50 years old and has been rediscovered by a variety of different optimization communities. Algorithm 8 outlines our particular version of it. In essence the random search chooses 2n candidate policies at each step by adding zero-mean Gaussian noise to the current policy parameters. These candidate policies are used to perform rollouts, and the reward for each rollout is recorded. These rewards are then used in the update step for the policy.

In [46], the authors show that this direct policy search is competitive with DRL. Specifically, they add a number of ”tricks” to the basic algorithm and call their resulting approach the Augmented Random Search (ARS). We add our own set of tricks in this work. First, we keep ARS's update step, where the step size is divided by the standard deviation of returns obtained. Second, we maintain the normalization functions learned by the DRL algorithms we are tuning. This differs from algorithm to algorithm, but usually it involves normalization of the data using statistics of the observation that is seen during training, followed by a clipping operation. We also found that it is important to be careful with the random seeds used for rollouts, and so for each policy pair $\theta \pm \delta_i$ we

Env.	Algo.	Fail % Before	Fail % After
Walker	A2C	19.33	12.00
	PPO	0.00	0.33
	DDPG	42.67	4.00
	TD3	12.33	1.67
	SAC	2.33	0.67
	TQC	5.67	0.00

Table 6.2: Measured early termination events before and after the fine tuning process

ensured that the environment used the same seed. This was particularly important for environments with a wide distribution of initial conditions. Finally, we found performance was slightly improved by using a linear schedule for step size and exploration noise.

One advantage of this method that we have found is that it is not very sensitive to hyper-parameters. For every result presented in this paper, we deliberately used the same parameters: 200 update steps, $n = 64$, $\alpha = [0.02, .002]$, and $\sigma = [0.025, 0.0025]$, which were chosen using the parameters used in [46] as a starting point. Ignoring for a second that 200 update steps is in fact more than is necessary for most environments, this implies that our method takes 25600 rollouts to train. In simulation with parallel rollouts, this is completed in a matter of minutes using a Ryzen 3900x. For the Panda arm environments that we will discuss in more detail later, this would correspond to about 14 hours of real robot time, and we suspect this time could be brought down considerably by tuning the hyper-parameters specifically for sample efficiency.

We also note that we ran experiments where we train only a subset of the neural network parameters, which would make the number of trainable parameters comparable to the linear policies used in [46]. During these experiments we found the results were slightly inferior to training on the entire network, and that the sample efficiency, measured by number of updates required to reach a given reward threshold, was almost exactly the same.

Algorithm 8 Direct Policy Search

```
Require: Policy $\pi$ with trainable parameters $\theta$
Require: Hyper-parameters - $\alpha$ $\sigma$ $n$
1: Sample ffi = [ffi1, ..., ffin] from $\mathcal{N}(0, \sigma)^{\mathrm{n\ x}\,|\theta|}$
2: $\theta^* = [\theta - \delta_1, ..., \theta - \delta_n, \theta + \delta_1, ..., \theta + \delta_n]$
3: for $\theta_i$ in $\theta^*$ do
4:     Do rollout with policy $\pi_{\theta_i}$, using the MLA
5:     Collect sum of rewards $R_i$.
6: $\theta^+ = \theta + \frac{\alpha}{n\sigma_R} \sum_{i=0}^{n} (R_i - R_{i+n})\delta_i$
```

6.2.2 Environments

We examine a number of popular benchmarking environments from the RL community. The environments all conform to the OpenAI Gym API introduced in [19]. For ease of reference, we will refer to each environment by the ID it has in the Gym registry. MountainCarContinuous-v0, LunarLandarContinuous-v2, BipedalWalker-v3, and BipedalWalkerHardCore-v3 are all standard continuous control environments included with the base Gym environments. To the best of our knowledge these environments are not meant to be physically realistic. We also study a collection of locomotion environments implemented in PyBullet. The locomotion environments were created by [78] and are maintained by the Bullet Physics team [79]. In this work we study HalfCheetahBulletEnv-v0, HopperBulletEnv-v0, Walker2DBulletEnv-v0, and AntBulletEnv-v0. All of these environments are simulated legged robots. Agents take joint angles and velocities as input states, and compute joint torques as actions. The reward functions are designed to encourage agents to walk forward as fast as possible. It may be worth noting that these are inspired by OpenAI's popular Mujoco environments, though the Bullet versions are considerably heavier and impose more realistic torque limits, which makes them a bit more challenging for RL algorithms. In the second half of this paper, we study a set of environments based on a 7DOF Franka Emika Panda arm [78]. These environments are made difficult both by their complexity and the fact that they use a sparse reward structure. As an example of these aspects, consider the PandaPickAndPlace-v1 environment, in which the arm must pick up a block somewhere in its workplace and bring it to a randomized goal state. The agent recieves a reward of -1 everywhere except when the block has reached the goal state.

6.2.3 Pre Trained Agents

We use the Stable Baselines 3 Zoo [80] [81] for a collection of pretrained agents with tuned hyper parameters. The Zoo provides agents for Truncated Quantile Critics (TQC), Soft Actor Critic (SAC), Proximal Policy Optimization (PPO), Asynchronous Actor Critic (A2C), Deep Deterministic Policy Gradients (DDPG), and Twin Delayed Deep Deterministic policy gradient (TD3) [82] [55] [83] [84] [16] [23]. In all examples, the policies are deep neural networks and the exact architecture has been tuned by the Zoo maintainers to have reasonable performance for each environment algorithm pair. We use these policies to initialize the values of θ in Algorithm 8.

Environment		A2C	PPO	DDPG	TD3	SAC	TQC
Walker2D	Baseline Dim.	2.55 ± 0.6	3.45 ± 0.4	5.54 ± 0.5	**6.09 ± 1.6**	5.96 ± 1.6	**5.36 ± 0.5**
	Tuned Dim.	1.21 ± 0.3	2.35 ± 0.2	3.82 ± 0.3	**3.72 ± 0.3**	3.85 ± 0.5	**3.71 ± 0.2**
	Baseline Return	785 ± 389.2	2108 ± 16.0	1432 ± 720.1	**2218 ± 194.6**	2290 ± 34.8	**2540 ± 557.6**
	Tuned Return	997 ± 2.2	2024 ± 10.1	1961 ± 12.5	**2152 ± 27.6**	2269 ± 13.3	**2562 ± 12.6**
HalfCheetah	Baseline Dim.	3.19 ± 0.3	3.35 ± 0.2	**4.31 ± 0.4**	**5.17 ± 0.3**	4.83 ± 0.3	3.65 ± 0.2
	Tuned Dim.	2.4 ± 0.2	2.54 ± 0.2	**3.01 ± 0.3**	**2.76 ± 0.3**	3.46 ± 0.3	2.56 ± 0.2
	Baseline Return	2109 ± 36.3	2938 ± 53.7	**2064 ± 198.7**	2820 ± 21.0	2792 ± 10.9	3676 ± 16.7
	Tuned Return	2137 ± 22.3	2778 ± 27.5	**2594 ± 41.9**	2697 ± 13.1	2658 ± 12.1	3606 ± 7.2
Hopper	Baseline Dim.	2.85 ± 0.5	3.16 ± 0.5	3.67 ± 0.5	3.76 ± 0.4	**5.12 ± 0.3**	**5.12 ± 0.3**
	Tuned Dim.	2.24 ± 0.1	2.31 ± 0.2	3.12 ± 0.1	2.74 ± 0.1	**2.7 ± 0.2**	**2.3 ± 0.1**
	Baseline Return	**834 ± 343.3**	**2523 ± 383.5**	**1179 ± 453.1**	2681 ± 27.2	**2602 ± 205.2**	**2631 ± 329.7**
	Tuned Return	**2072 ± 12.4**	**2559 ± 26.0**	**2641 ± 39.2**	2763 ± 7.4	**2687 ± 8.1**	**2547 ± 10.4**
Ant	Baseline Dim.	2.65 ± 0.2	3.91 ± 0.6	7.14 ± 0.4	5.76 ± 0.2	7.17 ± 0.3	5.25 ± 0.3
	Tuned Dim.	2.15 ± 0.2	3.11 ± 0.1	6.87 ± 0.3	4.29 ± 0.4	3.35 ± 0.2	3.39 ± 0.2
	Baseline Return	2502 ± 25.4	2869 ± 72.7	**2365 ± 212.5**	3268 ± 288.8	3096 ± 31.3	3478 ± 24.0
	Tuned Return	2527 ± 13.5	2817 ± 26.8	**2498 ± 42.9**	3330 ± 100.1	2854 ± 8.0	3488 ± 3.4

Table 6.3: Returns and Dimensionality after Fine Tuning with an extra dimensionality reward term

6.3 Results

First we examine the results of using our direct policy search for policy fine-tuning of a large set of environments and initial policies, using the parameters from Section 6.2.1. We compare the mean and standard deviation of returns before and after our fine-tuning process. In both cases, the policies are evaluated deterministically by using the MLA at each step, the only randomness in the system is from the initial condition at the start of each episode, which is drawn from the same distribution seen during training. Each agent is evaluated with 100 Monte Carlo trials.

The results are presented in Table I. In almost all cases, we see at least a modest improvement to average return. Recalling that an even more fundamental goal in this work is to reduce variability, also note that many cases resulted in a substantial decrease in the variance of the return, as desired. This suggests that our fine-tuning process is effective both for squeezing extra performance out of a trained DRL agent and also for reducing the variability of those agents.

We also examine the robustness of these policies. We note that the baseline agents, even with no noise added and using the deterministic policy evaluation, will experience failure events from some particular initial conditions. Here we define failure as any "early termination" event from the environment. In the case of Walker2DBulletEnv-v0, the environment automatically terminates early if a non-foot link contacts the ground or if the simulation determines that a fall is imminent due to its center of mass location or body orientation. To test robustness, we sample 300 initial conditions and evaluate the policies both before and after our refinement step. We present the results from the Walker2D system because it had the highest failure rate across all baselines algorithms. We can see in Table 6.2 that DDPG, for example, failed in about 42% of cases before the refinement process and in around 4% afterwards. The other algorithms show improvement as well. In the case of TQC, we went from failing about 5% of the time to not detecting any failure events during the 300 trials.

6.4 Mesh Dimensions

In previous work [85], we introduced what we call a "mesh dimension" as an component to reward functions for reinforcement learning agents. Informally, agents typically operate in relatively high dimensional state spaces. However in practice they will often only move along a comparatively lower-dimensional manifold within that full space. That is, although motions are not completely synchronized over time, they demonstrate quite a bit of coordination among joints. By eye, such a gait-like coordination is often quite apparent. The mesh dimension attempts to identify this dimensionality reduction quantifiably. It estimates the dimensionality of the reachable state space of the closed-loop system, and, for those familiar with the term, it is very closely related to a "fractal dimension".

In another line of prior work [50], we showed that ARS was able to train linear policies on environments which were modified to include this mesh dimension reward. This had a number of desirable qualities including finding very precise periodic gaits in some cases, and it improved robustness to push disturbances and sensor noise. In that work training used a lower and upper bound of the estimated dimensionality; in this work we train on the average of those two bounds.

We experimented with several ways to incorporate this measure of dimensionality into the reward function, including both a linear and quadratic combination with the original reward. While these methods worked to some extent, they required fairly precise manual tuning of coefficients. Somewhat surprisingly, we found that simply taking the product of the original reward multiplied by the reciprocal of the dimension estimate D was an effective reward that required no manual tuning:

$$R^r = \frac{\sum_{t=0}^{T} r(s_t, a_t, s_{t+1})}{D}. \tag{6.1}$$

One caveat is that this only works for environments with positive rewards. For negative returns, however, as in the case of the Panda environments, we can simply take the product instead:

$$R^p = D \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}). \tag{6.2}$$

We found that these rewards successfully gave the agents a signal to optimize, leading to significant reductions in dimensionality without any significant degradation in performance over the original reward.

6.5 Mesh Dimension Results

6.5.1 Locomotion Environments

First, we present results for fine-tuning with the post-processed reward from Equation 6.2. In this work we train DNNs on the more difficult Bullet environments. Results are shown in Table III. All agents are evaluated deterministically here, using the MLA. Each entry in the Table for mean and standard deviation for the return and for the estimated dimensionality is calculated based on 100 Monte Carlo Trials. We see that the dimensionality ("Dim.") in most environments is decreased quite drastically, particularly for the off-policy algorithms (DDPG, TD3). As before, this process also seems to decrease the variability of the return, perhaps even more reliably than without the dimension reward. We believe this shows that our previous results can be extended to DNNs, which greatly expands the scope of problems they can be applied to.

6.5.2 Panda Arm Environments

We present data here for a set of environments utilizing a Panda arm, introduced earlier in Section 6.2.2. These environments present several challenging problems for an

Emika Franka Panda arm. The PandaReach task is the most straightforward. Here, the goal is for the arm to reach a given point in task space. For PandaPush, the arm mush push a block along the floor to a desired location. In PandaSlide, the robot must grab a block and and bring it to a desired point on the ground. Finally, PandaPickAndPlace requires the arm to pick up a block and keep its grip on it while attempting to reach a point in space. We found that merely fine-tuning the action network with our random search did not improve performance significantly, though to be fair, none of the algorithms in the Zoo are able to solve this environment without Hindsight Experience Replay (HER) [86].

Environment	Base Dim.	Our Dim.	Base. Return	Our Return
PandaReach	2.73 $\pm$ 0.7	2.28 $\pm$ 0.5	-2 $\pm$ 0.6	-1 $\pm$ 0.7
Pick&Place	1.63 $\pm$ 0.3	1.61 $\pm$ 0.5	-6 $\pm$ 2.6	-11 $\pm$ 13.3
PandaPush	1.91 $\pm$ 0.5	1.68 $\pm$ 0.3	-6 $\pm$ 2.7	-7 $\pm$ 3.0
PandaSlide	1.89 $\pm$ 0.4	1.53 $\pm$ 0.3	-22 $\pm$ 7.1	-41 $\pm$ 12.4

Table 6.4: Dimensionality and Returns before and after fine tuning for the panda environments

We did however also apply our dimensionality reward signal to this environment using our fine tuning process. We show the resulting dimensionality, and the returns in terms of the original reward function are shown in Table IV. We observed a modest decrease in the dimensionality, accompanied by some decrease in the original return. Again, this decrease in reward is not unexpected, as we are after all trained on modified reward function. In addition to this, despite the Table data that suggests perhaps only a small change in behavior was observed, we noticed a significant beneficial change in the qualitative behavior of the robot. Figure 6.1 shows a stark example of this. In the PandaReach environment, the baseline agent is able to get its end effector into the target region, however it exhibits undesirable shaking behavior which the reward function does not punish. Our agent is able to achieve the same effect with a smooth motion. Note that both agents received exactly the same reward for the episodes we show,i.e. that despite the jittering deviations in the end effector, it remain in the goal region.

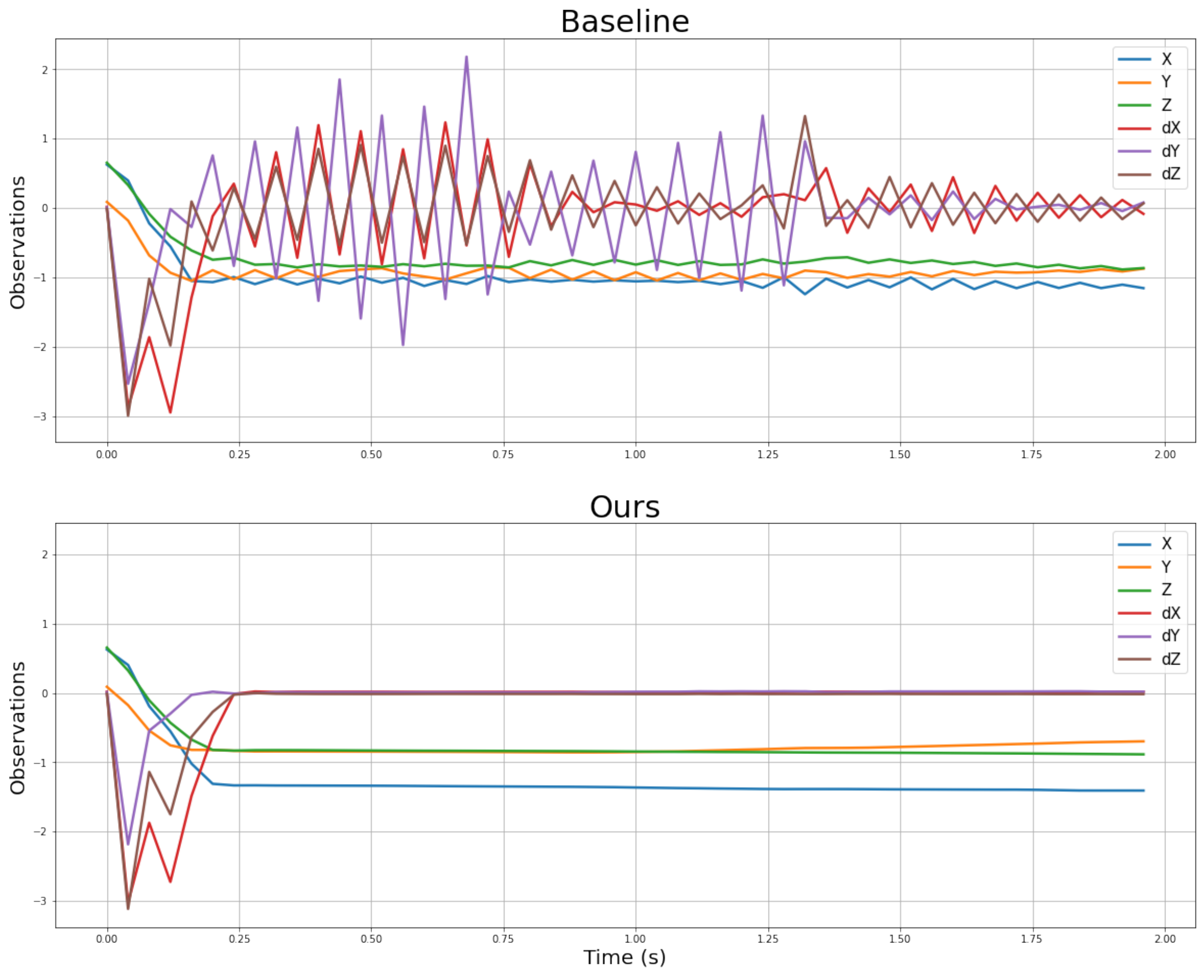


Figure 6.1: End effector positions and velocities for a policy roll out on PandaReach before and after fine tuning with the mesh dimension reward

6.6 Discussion and Related Work

It's worth discussing alternatives to our method for fine-tuning. The most similar work to ours that we have found is [38]. There, the authors take a similar approach in that they decouple the algorithms used for exploration versus exploitation. We agree with their conclusion that this decoupling brings advantages on its own, regardless of the methods used for the fine-tuning exploitation. In some sense their work is doing the opposite of our approach, however, in that after using a gradient-free evolutionary strategy for exploration, it then uses DRL for exploitation (rather than for exploration). While the approaches are quite distinct, they are also in fact likely compatible, in that they could actually be combined. It is easy to imagine a pipeline using their gradient-free method for broad exploration, followed by DRL for initial exploitation, with our random search added for the final fine-tuning stage of an algorithm.

There are also many small tricks and improvements found in DRL algorithms that aim to achieve similar results to what we've shown. One example is to decrease the SGD/Adam/RMSProp step size as training goes on. This is an effective method, and indeed our own method uses a linear schedule for the step size. However, the algorithms we are using as a baseline were already using this approach as well, and we still saw improvements in performance with the additional of our fine tuning process.

Entropy regularization / penalties are another toolset available, which can also be put on a schedule. These can encourage an agent to use a wide distribution of actions initially and then gradually narrow this down as training continues. Again though, most of the algorithms used as a baseline (PPO, SAC, TQC) have some form of this already, and our method is still able to improve on them.

We could try curriculum learning, meaning that the reward function could change by design as training goes on. We believe this is likely most effective when one has a lot of domain knowledge of the task, and when being applied to tasks that are too difficult for the algorithm to learn initially. For example, the authors of [87] use this approach when controlling Cassie. This approach works well for them because they are able to engineer a reward that led to the desired behavior.

6.7 Conclusion

We have presented a method that can fine tune policies obtained from DRL algorithms by optimizing directly using the MLA. We showed that performance compared to a baseline was improved considerably on a large set of standard benchmarking tasks. Of more particular note, the variability of episode returns was decreased significantly on many of the environments we tested as well. For the system on which we also quantified failure rates (i.e., for the biped Walker), this lower variability was also accompanied by significantly fewer early termination events compared to the baseline. We hypothesize that this increased robustness is, quite plausibly, due to the dimensionality reduction. (That hypothesis is in fact why we performed these experiments, of course.) However, any conclusions on correspondence remain a topic for further investigation.

We also showed that this method allows us to expand our previous work on adding dimensionality metrics to the reward function of RL agents to DNNs as well, which greatly expands the scope of problems it can be applied to. We demonstrated this on a set of locomotion environments and also on a challenging set of Panda arm environments with sparse reward structures. We showed that for the case of the Panda our approach achieved significantly less jitter, and arguably more visually pleasing (and mechanically desirable) motion than the baseline, without any environment specific reward shaping, or manual adjustment of any parameters.

We believe versatility and simplicity are major strengths of this approach. Policies obtained from any kind of DRL algorithm can be tuned in this way, and the method seems to require very little manual tweaking. The potential applications for this method are broad. Engineers designing robots which are public facing or that interact with humans may find it useful to employ policies that make their robots motions smoothing and thereby easier for humans to predict. There are applications outside of robotics as well.

Physics-based character animation may also benefit from more consistently behaving policies, and DRL is also popular for video game AI, which is another area where the improved consistency of this method may prove desirable.

Finally, we will end with a discussion on the broader impacts and future directions of this work. DRL has been an exciting and promising paradigm for robotic control for some time now, but it has yet to be widely adopted by industry. This is largely because it is difficult to trust a DNN controller, and deploying a poorly understood controller can be expensive and dangerous. By itself, we think the fine tuning method we've introduced can help make DRL policies more effective and reliable, however we also think that the lower dimensional policies can unlock even more tools to aid with this. With lower dimensional polices, we open the possibility to develop methods to perform numerical estimates of a variety of controls-based metrics, such as rates of contraction (Lyapunov exponents), identification of dangerous regions in state space (outside a stochastic separatrix for a basin of attraction), and/or expected (conservative) distributions of failure rate. All of these are promising directions towards safer and more reliable DRL based control, and we anticipate that our method brings us closer to realizing them for useful, real world, robotic systems.

Chapter 7

Reinforcement Learning in Differentiable Simulators

Computer simulation has become an indispensable tool for researchers and engineers of robotic systems for design, control, and verification. Recent advances in model free deep reinforcement learning (DRL) have been able to leverage simulation and the continued exponential increase computer resources to solve a variety of challenging control and perception problems. Examples include dexterously manipulating objects with a 24 DOF robotic hand [29], and recent work from the Robotic Systems Lab at ETH Zurich demonstrates rough terrain quadruped robot arguably on part with Boston Dynamics [39]. In both cases, training was done in simulation, and then successfully transferred to the real world.

These simulations have largely been treated as black boxes by DRL algorithms. This is both a strength and a weakness of DRL, and this stems partially from the fact that in commonly used simulators like MuJoco [88] or Bullet [79], are not differentiable (nor are physical robots, for that matter). That is, we are unable to compute the gradient of the state at time t+1, with the action from time t. Therefore we are also unable to take the gradients of the reward function with respect to controller parameters. Thus, RL must thus rely on various approximations of the true gradients, like finite differences or various policy gradient algorithms, the classic example being REINFORCE [11]

However in recent years, fully differentiable physics simulators have started to emerge [89] [90] [91], which offer analytic gradients using automatic differentiation. These simulators already have a number of interesting applications. For example, it is possible to use data from a physical system as data for a learning algorithm to make simulation to better match. Furthermore the nature of some these simulators allow them to be run on hardware accelerators, which offers some obvious speed advantages for algorithms which can make use of them. And, most relevant to this work, they also provide analytic gradients for any differentiable function of simulation state variable. This means we can use stochastic gradient descent, which is the gold standard for training neural networks, directly using the negative sum of rewards as the loss function. We will call training policies in this way an analytic policy gradient algorithm.

However, in practice, these gradients have proven extremely challenging to use, for a number of reasons. Part of the problem is that to take the gradient through any iterated dynamical system required back propagation through time (BPTT). Long chains of BPTT have long been known to cause exploding or vanishing gradients, which naturally causes to difficulty in learning [92]. A recent paper by Metz et. al. [93] also offer some exposition on the challenges of using the analytic gradients offered by these new rigid body simulators. They highlight that in addition to problems inherent to BPTT, the naturally chaotic dynamics of many rigid body systems exasperate the problems with diverging gradients significantly.

Another difficulty are severe local maxima. Local maxima are a common problem in all of deep learning, but it is apparent that using APG for reinforcement learning with rigid body systems is especially prone to falling into these extrema. In [93] they also show the reward landscape of the Ant system in Brax. The Ant is a standard benchmark problem in the RL community, it consists of a quadruped robot who's objective is to move forward as fast as possible. They show that reward landscape and reward gradient for this system has extremely high variance, and fraught with local extrema. One may expect that this is due primarily to the fact that the Ant system is relatively high dimensional and subject to contact and frictional forces with the ground. However as we show in figure 7.1, an Acrobot system, which is a simple two DOF system that is not subject to any contact forces, suffers from many of the same problems.

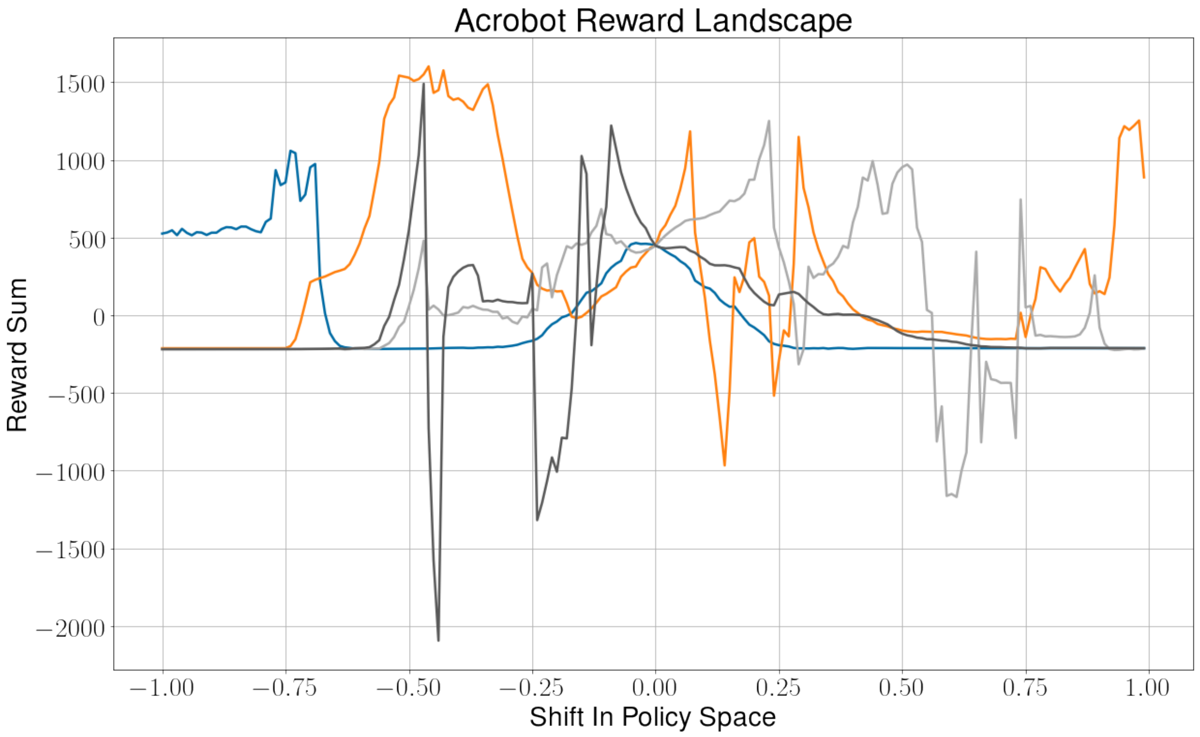


Figure 7.1: A visualization of the reward landscape for the Acrobot. We started with a random neural network policy, and then sampled a random vector from parameter space. We then added used this vector to shift the initial policy, and record the sum of rewards for a single rollout from the shifted policy . We use the same initial condition for each trial, the only difference between rollouts is that a shifted different policy is used.

Given these difficulties, it remains an open question what role, if any, analytic policy gradients have to play in reinforcement learning for robotic control. In this paper we present a novel algorithm, Cross Entropy Analytic Policy Gradients (CE-APG), which we have found is able to outperform both vanilla APG and state of the art DRL algorithms in at least one class of challenging nonlinear control problems. Specifically, we demonstrate that under-actuated planar kinematic chains, like the acrobat or inverted cart pole pendulum can be successfully controlled by this algorithm.

CE-APG uses APG as a local search, combining it with an outer loop cross entropy method to escape from local maxima in the reward landscape. This was inspired the the approach taken by the authors of Tiny Differentiable Simulator [90], which used an optimization technique called Parallel Basin Hopping combined with a gradient based algorithm for what was essentially system ID.

7.1 Background And Related Work

7.1.1 Deep Reinforcement Learning

Deep Reinforcement Learning has seen a lot of attention and impressive results over the last decade or so, including in the context of continuous control for robotic systems [34] [29] [45] [42]. These problems are all high dimensional, nonlinear, underactuated, and they all involve complex contact sequences with the environments, which makes them very challenging for more traditional control design.

DRL is usually divided into model based and model free control. Model based reinforcement learning learns a model of the system under control, and uses that to do planning, classic dynamic programming is an example, as is PILCO [54]. These approaches are typically much more sample efficient than model free RL, however these methods typically have a hard time scaling to higher dimensional problems, and can require expensive re planning at run time.

Model free RL on the other hand learns a policy directly to maximize a reward function. This implies solving a more difficult optimization problem, but is what has been used to achieve all the results we have discussed in this paper. Examples of model free algorithms include Soft Actor Critic (SAC), Proximal Policy Optimization (PPO), and Twin Delayed Deep Deterministic policy gradient (TD3) [55] [83] [23].

Although they do not strictly use nueral networks, gradient free methods like Evolutionary Strategies (ES) [25], or Augmented Random Search (ARS) [46] can also be considered modern model free reinforcement learning algorithms.

Our algorithm can be considered a model free reinforcement learning algorithm, and inherits many of their advantages and disadvantages. For example the algorithm we present should theoretically scale well to high dimensional tasks. However our method currently has sample efficiency on par with other DRL algorithms, and, as it requires a differentiable simulator to function, will require sim2real transfer to be applicable to physical systems. We do note that successful transfer of polices from simulation directly to hardware has been demonstrated many times in the literature [29] [39].

7.1.2 Policy Search in Differentiable Simulations

Many of the papers which introduce a differentiable simulator also include a basic example using analytic gradients for policy search. Brax [91] and the unnamed simulator developed by Degrave et. al. [94] both implement a basic version of APG. Brax's APG is used to command a fully actuated double pendulum to reach random targets, but is currently unable to solve most of the other problems in their benchmarking suite. Degrave Et. Al. manage to develop a walking gait for a quadruped, though their method requires a fairly significant amount of hand designed components.

In [95] They introduce a simulator and suggest something called "policy enhancement" whereby they augment a model based RL algorithm with the analytic gradients to control a fully actuated double pendulum.In [96] the authors present policy optimization via differentiable simulation. They don't directly use the analytic policy gradient, and instead develop an indirect second order method. They also demonstrate their system on under-actuated systems like the inverted pendulum, the difference is that they are balancing at the stable equilibrium. This is a deceptively difficult problem, however we show that our method works to swing-up and balance the system in their unstable equilibrium, which we would argue is more difficult.

7.1.3 Combining Local and Global Search

As already mentioned, our algorithm was inspired by basin hopping [97] and the extension of parallel basin hopping [98]. Tiny Differentiable Simulator [90] uses this method for parameter estimation in their own work to perform system ID. We instead use a cross entropy method, and are obviously tackling a different method.

There are also several methods that combine a zeroth order optimizer with a local gradient based optimizer for robot learning [99] [100] [101] [99]. However none of them are making use of analytic gradients, or doing direct policy optimization.

7.1.4 Back Propagation Through Time

In order to propagate gradients through an iterated system, we must use a technique called back propagation through time. As we have mentioned, this causes difficulty in exploding or vanishing gradients, especially in long chains of computation. The problem essentially is that the reward at time t depends not only on the state and action at time t-1, but on the state and action from t-2, t-3, back to the initial state, even if system itself is Markovian. For even modest length roll-outs this causes instability in the gradients that make learning with them challenging.

There are many other contexts that this problem arises, including in natural language processing (NLP). One of the tools used to combat this in the NLP community are specialized recurrent neural networks, which are specifically designed to stop gradients that pass through the network from diverging. In our case we use gated recurrent unit GRU [102] as our control policy, we outline this architecture with more detail in the methods section.

7.2 Problem Formulation

7.2.1 Reinforcement Learning

In reinforcement learning, the goal is to train an agent, acting in an environment, to maximize a scalar reward function. The environment is a discrete time dynamical system described by state $s_t \in \mathbb{R}^n$ and the current action $a_t \in \mathbb{R}^b$. An evolution function $f : \mathbb{R}^n \times \mathbb{R}^b \to \mathbb{R}^n$ takes as input the current state and action, and outputs the state at time t+1:

$$s_{t+1} = f(s_t, a_t) \tag{7.1}$$

The controller is a function parameterized by a vector $\theta \in \mathbb{R}^{|\theta|}$ that maps states to actions $g : \mathbb{R}^n \times \mathbb{R}^{||\theta||} \to \mathbb{R}^m$ such that:

$$a_t = g(s_t, \theta) \tag{7.2}$$

The goal is to maximize a scalar reward function $r : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$. We consider the finite time undiscounted reward case. The objective function then is:

$$R(\theta) = \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \tag{7.3}$$

7.2.2 Kinematic Chains and Simulation

We consider three systems, a classic cartpole pendulum, a double cartpole pendulum, and an Acrobot [58]. These are all under-actuated, kinematic chains, and are often used as benchmark problems for both reinforcement learning and nonlinear control. Their dynamics in general can be described with the following:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + T_g(\mathbf{q})g = \mathbf{Bu} \tag{7.4}$$

with M being the inertia matrix, C being the Coriolis matrix, T being a matrix capturing gravitational affects. For these systems u is the torque outputted at each motor, and q is the vector of state variables. In each of these systems, there is a single unstable equilibrium point. Our goal is to swing the system from an initial condition to it's unstable point and the maintain balance there.

It was demonstrated in [49] that most DRL struggles with the full version of the Acrobot in particular. It is worth elaborating on that point. Many benchmarking suites (for example, OpenAI's gym [19]) have underactuated systems like the acrobot or cartpole pendulum, however the tasks are typically either to swing the system up, or to balance it, asking one controller to do both becomes a much more challenging problem.

7.2.3 Acrobot

The Acrobat is kept relatively simple, the only state variables are the joint angles and velocities, the reward is just the negative squared distance between the goal state and the current state:

$$r_a = -\phi_1^2 - \phi_2^2$$

With $\phi_1 = \phi_2 = 0$ being the upright balanced position with both links straight up.

7.2.4 Cartpole Pendulum

The cart-pole pendulum is also kept simple, the states are simply the joint angles and velocities, and the reward function is simply the negative square of the pendulum angle, with $\phi_1 = 0$ corresponding the the upright position.

7.2.5 Inverted Double Cartpole Pendulum

The double cartpole pendulum is modified from Brax's existing benchmark environments, the only difference is that the initial condition is rotated 180 degrees from the upright. These environments use some reward shaping, adding an an alive bonus as well as some feature extraction. rather than directly feeding in joint angles, both the reward and state variables are fed in as the x,y coordinate for the end of each link. The reward function for the environment is:

$$r_{dp} = r_{alive} - r_{distance} - r_{velocity} \tag{7.5}$$

Where

$$r_{alive} = 10 \tag{7.6}$$

$$r_{distance} = 0.05x^2 + (y - y_{des})^2 \tag{7.7}$$

$$r_{velocity} = \dot{\phi}_1 + \dot{\phi}_2 \tag{7.8}$$

and y_{des} co-responds to the height of the second link when in the upright position. Put another way, the reward function is an alive bonus minus the euclidean distance between the end of the second link and the goal state.

7.2.6 The Brax Simulator

The above systems are simulated using Brax [91]. Brax is a differentiable physics engine that can simulate systems made up of rigid bodies, joint constraints, and actuators. The simulators primary advantage is that it can run massively parallel simulations very quickly on accelerator hardware, I.E. TPUs and GPUs. By virtue of being written entirely in Jax [?], we can also take arbitrary gradients through the simulator using autodiff.

7.3 Methods

7.3.1 Analytic policy gradients

Stochastic gradient descent and its variants (in our case adam [?]) are the gold standard for training deep neural networks. We seek to train our policy using the gradient of the sum of the reward function for a given episode.

$$\theta^{+} = \theta + \alpha \nabla_{\theta} R(\theta) \tag{7.9}$$

To be more specific, we perform N policy rollouts using the current parameters, take the gradient of the mean of the sum of rewards for each these roll outs, use that gradient to update the current policy, and repeat until convergence. Thus our update step is:

$$\theta^{+} = \theta + \alpha \nabla_{\theta} \frac{1}{N} \sum_{i=0}^{N} \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}) \tag{7.10}$$

As we've discussed, because we are using a differentiable simulation, the analytic gradient with respect to θ is available to us.

7.3.2 The Cross Entropy Method

The Cross Entropy Method (CEM [103]) is a well established algorithm for importance sampling and optimization. CEM maintains a probability distribution over its decision variables, in this case the decision variables are the parameters for our policy. The most common formulation is to use a normal Distribution, thus we must mantain a vector of means μ_{pi}, and a covariance matrix σ_{π}. $\mathcal{N}(\mu_{\pi}, \sigma_{\pi})$. At each step we sample candidate policies from this distribution, and use the following update rules:

$$\mu_{\pi}^{+} = \frac{1}{K_e} \sum_{i=0}^{K_e} \mu_i \tag{7.11}$$

$$\sigma_{\pi}^{2+} = \frac{1}{K_e} \sum_{i=0}^{K_e} (\mu_{\pi} - \mu_i)(\mu_{\pi} - \mu_i)^T \tag{7.12}$$

Howeve, the covariance matrix grows quadratically with the number of policy parameters, and neural networks can have thousands of parameters even for small systems. Thus, we make the following simplification to the variance:

$$\sigma_{\pi}^{2+} = \frac{1}{K_e} \sum_{i=0}^{K_e} (\mu_{\pi} - \mu_i)^2 \tag{7.13}$$

This implicitly ensures that our covariance "matrix" only has entries on the diagonal, and can thus be stored as a vector. This simplification is also made by [99].

7.3.3 Cross Entropy Analytic Policy Gradients

Algorithm 9 Cross Entropy Analytic Policy Gradients

```
Require: Policy $\pi$ with trainable parameters $\theta$
Require: Hyper-parameters - $\sigma_0$, $K_a$, $K_e$
  Sample $\boldsymbol{\theta_c} = [\theta_1, ..., \theta_n]$ from $\mathcal{N}(\theta, \sigma^2)^{K_a \times |\theta|}$
  for $\theta_i$ in $\boldsymbol{\theta_c}$ do
      Run APG with initial policy weights $\theta_i$
      Collect sum of rewards $R_i$ and final policy $\theta_i^*$.
  Sort $\theta^*$ values in descending order according to reward
  $\theta^+ = \frac{1}{K_e} \sum_{i=0}^{K_e} \theta_i^*$
  $\sigma^+ = \sqrt{\frac{1}{K_e} \sum_{i=0}^{K_e} (\theta - \theta_i^*)^2}$
```

We combine these two algorithms as follows. Start with initial policy weight θ, and an initial parameter variance σ_0. We then generate K_a candidate policies by sampling from $\mathcal{N}(\theta, \sigma_0)$. Using these policies as initial conditions, we run K_a analytic policy gradient algorithms in parallel, which gives us new weight vector $[\theta_0, \theta_1 ... \theta_{K_a}]$, and the final rewards for these new policy weight, $R_1, R_2 ... R_{K_a}$. We then sort the policy weights in descending order based on their associated final return. Finally we select the top K_e and use equations 7.11 and 7.13 to update our parameter and variance vector. This is repeated until some stopping criteria, for this paper we simply train for fixed number of steps.

7.3.4 Controller Architecture

As we have already mentioned, we employ a Gated Recurrent Unit (GRU) network as our control policy to help combat the exploding / vanishing gradient problem. For our CE-APG experiments, we used a GRU with two fully connected layers on the output, with ReLU hidden activations and a Tanh non-linearity on the final layer. Network sizes for each experiment can be found in the appendix. We found that by using the GRU we are able to train with episodes lengths of at least 500 steps.

In addition to this, we use deterministic policies, rather than the stochastic ones usually associated with deep reinforcement learning. Typically in DRL, the policy actually parameters a probability distribution over possible actions. At every time step, one generates a new distribution based on the current state, and then samples from that distribution to select an action. While our algorithm is compatible with stochastic policies of this nature, we instead compute the action directly. We believe this is especially advantageous for systems with unstable and highly sensitive dynamics (like the acrobot, cartpole etc).

7.3.5 Implementation Details

There are some notable differences between our implementation of APG (which we call PAGP for parallel apg) and the implementation of APG provided Brax. First, we use different controller architectures, our method uses a deterministic GRU, and theirs uses a stochastic multi layer perceptron. Furthermore the parellization characteristics are quite different, the Brax implementation of APG was designed for use with a TPU, and thus performs hundreds or thousands of rollouts in parallel for every update step. We note that we found the performance of CE-APG to be significantly faster on CPU compared to GPU/TPU.

7.3.6 Network Architecture

Each policy consists of GRU cell with two fully connected layers on the output. The fully connection network has ReLU hidden activations and a tanh non-linearity on the final layer.

Environment	GRU Size	FC size
Cartpole Pendulum	4	4x16x16x1
Acrobot	4	4x16x16x1
Cartpole Double Pendulum	6	6x32x32x1

7.3.7 Hyper Paramaters

In all cases we selected the best hyper parameters we could find using a manual search, using a coarse parameter sweep as a starting point.

CE-APG Hyperparameter	Value
APG Epochs	100
Total Epochs	200
Total Env Interactions	1e7
initial std	0.05
learning rate	1e-3 $\rightarrow$ 1e-6
batch size (N)	4
K_a	24
K_e	8

PPO Hyperparameter	Value
Total Timesteps	8e7
Minibatch Size	32
Batch Size	256
Unroll Length	50
N Update Epochs	8
Discounting	0.99
Learning Rate	3e-4
Entropy Cost	1e-3
N Envs	512

SAC Hyperparameter	Value
Total Timesteps	2e6
Discounting	.95
Learning Rate	1e-3
N Envs	64
Batch Size	128

Brax-APG Hyperparameter	Value
Total Env interactions	1e7
N Environments	24
learning rate	5e-4

7.4 Results

For each environment, we ran trials with 8 random seeds. The seeds affect all the sources of randomness during training, of which there are several. The initial value of the policy parameters, the noise added to the policy at the beginning of each iteration, and the initial condition of the simulator at the beginning of each episode. Figure 7.2. For each of these trials we used the GRU controller architecture discussed above, details on layer sizes etc. can be found in the appendix. We report the resulting rewards obtained at the end of in table 7.1. In addition to our own algorithm, we also run comparisons from PPO, SAC, and Brax's implementation of APG (which has some note-able difference from our own, discussed in the appendix).

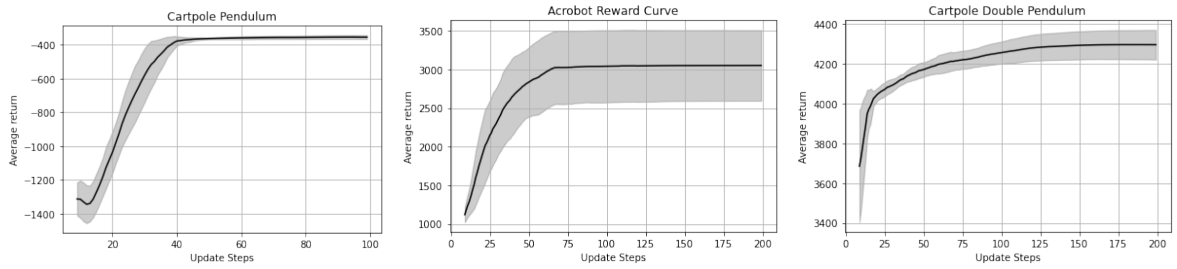


Figure 7.2: Reward curves for CE-APG

Environment	Pendulum	Acrobot	Double Pendulum
CE-APG (ours)	-355± 10	3053 ± 458	4295 ± 75
PPO	-464 ± 223	405 ± 927	4249 ± 549
SAC	-2274 ± 1000	359 ± 159	-3.9e5 ± 4.9e6
Brax Apg	-3485 ± 819	1949 ± 814	-1982 ± 4056

Table 7.1: Results of the training on our test environments, we report the mean and standard deviation of rewards obtained from training each algorithm with 8 random seeds

We can see that across all three environments, our method outperforms the other benchmark algorithms. On the double inverted pendulum we get the same final reward as PPO, exceed it slightly on the cartpole, and get significantly higher reward on the acrobot. In fact for the cart-pole in particular our algorithm significantly outperforms the others. It is worth putting this in context, as it is difficult to understand what a reward of 2000 vs. 3000 really means.

To do this we perform roll-outs with the best performing seeds for both CE-APG and the best performing benchmark, which was Brax's APG implementation. We perform 10 rollouts with these top performers. For completeness we report the mean and standard deviation of the resulting rewards, CE-APG: 3507 $\pm$ 5, Brax's APG: 2778 $\pm$ 89. However it is much more revealing to visualize one of these roll outs, which we do in figure 7.3. We can see that despite comparable total reward, APG does not actually stabilize the system at the equilibrium, likely because it has become stuck in a maxima of the reward landscape. By contrast, our algorithm, augmented by the cross entropy method to avoid such local maxima, manages to find a policy that does stabilize our system. Unsurprisingly given the rewards from table 7.1 none of the other systems manage to balance the acrobat either. Of course such stabilization is exactly what we as humans had in mind when defining the environment.

7.5 Ablation Studies

In addition to the comparisons done above, we also conducted several ablation studies, this isolates the effect of our implementation of APG from the results presented above. We conducted experiments using only the gradient free CEM method, with no analytic policy gradients being used. And also compared to our implementation of APG, which we are calling PAPG for parallel APG. This essentially means that we run APG N times in parallel, and picking the best result to return. In both cases we still used the GRU controller architecture as before. In the case of CEM we used 2000 iterations for every environment, which we found to be well past the point of convergence. For APG we used 200*100 iterations, which results in the same number of environment interactions to CE-APG. The results are shown in table 7.2.

As we can see, the performance of either method individually is generally poor, though

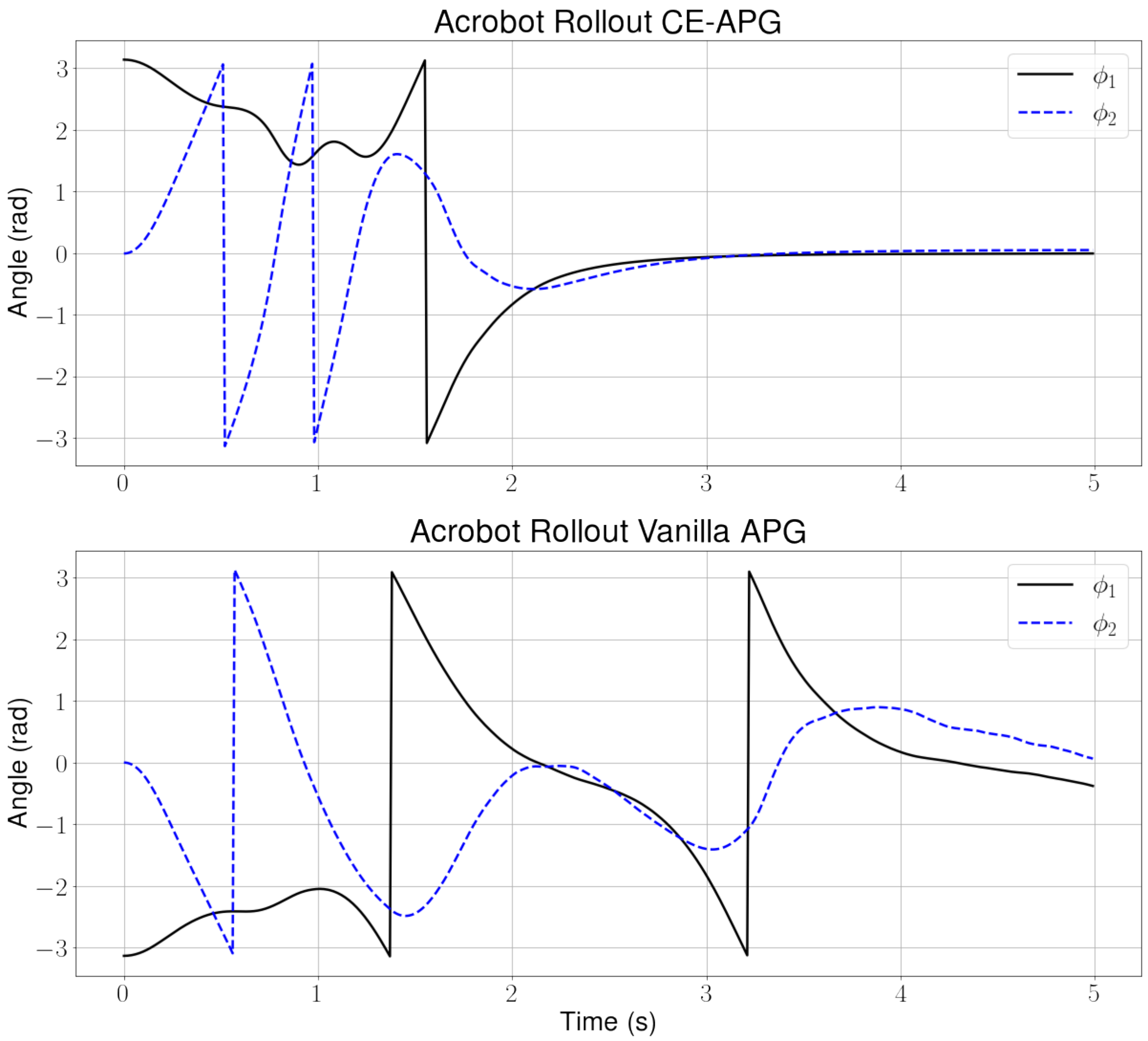


Figure 7.3: Best performing seeds for CE-APG vs APG on the Acrobot. Despite comparable total reward, APG does not actually stabilize the system at the equilibrium, likely because it has become stuck in a maxima of the reward landscape. By contrast our algorithm, augmented by the cross entropy method to avoid such local maxima, manages to find a policy that does stabilize our system

we do see that PAPG does about as well as the best agent from the APG results reported in table 7.1, however when we perform rollouts the resulting policies exhibit the same behavior shown in figure 7.3, that is they continuously spin, spending as much time as they can near the goal state, but never settling there.

Environment	Pendulum	Acrobot	Double Pendulum
CE-APG (ours)	-355± 10	3053 ± 458	4295 ± 75
PAPG	-1034 ± 208	2456 ± 647	2335 ± 431
CEM	-3019 ± 559	626 ± 160	-2.7e5 ± 7.6e5

Table 7.2: Results of our ablation studies. We report the mean and standard deviation of rewards obtained from training each algorithm with 8 random seeds

7.6 Discussion and future work

We have shown that analytic policy gradients can be leveraged effectively for at least one class of system, nonlinear underacted systems with unstable target states. This is a limited scope of problems, but it is interesting because other modern DRL algorithms struggle with such systems. However, we are as of yet unable to get our algorithm to perform well in contact rich environments, likely because the contacts introduce huge variance into the gradients. As of writing, there is active work in the community to make Brax friendlier to analytic gradient based algorithms, in particular adding soft contacts, which may very well help a lot.

In addition, the sample complexity of our algorithm is comparable to other on policy DRL algorithm, and is behind what we might expect from off policy algorithms. However there are many algorithmic improvements could be made, for example importance sampling has been found to improve the sample efficiency of CEM by up to a factor of 10. We are currently unable to effectively implement this due technical limitations in the way we implement parallelization.

7.7 Conclusion

In conclusion, we presented Cross Entropy Analytic Policy Gradients, an algorithm that can exploit analytic gradients. We covered some relevant background, introduced our method, and placed it in the broader context. We then presented our algorithm and the environments we used for testing. We presented results of our algorithm compared to state of the art baselines, and performed ablation studies. We then contextualized the rewards obtained on Acrobat in particular. This demonstrated that our algorithm was able to successfully stabilize the system, whereas the baseline algorithms where not. We think this algorithm shows that analytic policy gradients can be leveraged productively in at least context.

Chapter 8

Conclusions

To conclude, let us revisit the limitations of reinforcement learning that we outlined in chapter one, and examine how we have addressed them in this thesis. We claimed that modern RL does not have good ways to incorporate domain knowledge into their learning. To address this, we introduced two methods to do just this. In chapter 3 we introduced a method to learn a switching controller that can combine hand designed and learned controllers to solve a difficult nonlinear control problem. Later, in chapter 7 we introduced a method to allow reinforcement learning agents to make use of gradient information from a new class of differentiable simulator.

We also claimed that RL is difficult to trust and analyze, to address this we turned to previously developed mesh based tools. These tools allow us to analyze the robustness of locomotion policies under a given set of disturbances, but they suffer from the curse of dimensionality, which limits their use to lower dimensional systems. We introduced a method to train RL policies in such a way that lowers their mesh dimensionality, and analyzed these policies. We then showed that this reduced dimensionality also reduced the size of meshes for the reachable states of these systems, which is what is required to use meshing tools to analyze robustness. Finally we extended these results to fine tuning of arbitrary neural network policies. We show that a simple policy search can be used to shrink the mesh dimension of DNN policies, and that this sort of policy refinement is useful for increasing reward and lowering variance in policy rollouts across a wide variety of problems, even without the mesh dimension reward.

Reinforcement learning shows great promise for robotic control, and this work represents a step toward more reliable, safe, and performant controllers for the robotic systems of tomorrow.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* The MIT Press, second ed., 2018.

[2] M. Minsky, *Theory of neural-analog reinforcement systems and its application to the brain model problem.* PhD thesis, Princeton University, 1954.

[3] B. G. Farley and W. A. Clark, *Simulation of self-organizing systems by digital computer.*, *Trans. IRE Prof. Group Inf. Theory* **4** (1954) 76–84.

[4] A. L. Samuel, *Some studies in machine learning using the game of checkers*, *IBM Journal on Research and Development* **3** (1959), no. 3 210–229.

[5] D. Michie, *Experiments on the mechanization of game-learning part i. characterization of the model and its parameters*, *Comput. J.* **6** (1963) 232–236.

[6] D. Michie and R. A. Chambers, *BOXES: An experiment in adaptive control*, in *Machine Intelligence* (E. Dale and D. Michie, eds.). Oliver and Boyd, Edinburgh, UK, 1968.

[7] R. Bellman, *A markovian decision process*, *Journal of Mathematics and Mechanics* **6** (1957), no. 5 679–684.

[8] R. A. Howard, *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

[9] A. G. Barto, R. S. Sutton, and C. W. Anderson, *Neuronlike adaptive elements that can solve difficult learning control problems.*, *IEEE Trans. Syst. Man Cybern.* **13** (1983), no. 5 834–846.

[10] C. J. C. H. Watkins, *Learning from Delayed Rewards.* PhD thesis, King's College, Oxford, 1989.

[11] R. J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, *Machine learning* **8** (1992), no. 3 229–256.

[12] G. Tesauro, *Td-gammon, a self-teaching backgammon program, achieves master-level play.*, *Neural Comput.* **6** (1994), no. 2 215–219.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in *Advances in Neural Information Processing Systems* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing Atari with deep reinforcement learning*, 2013.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, *Human-level control through deep reinforcement learning*, *Nature* **518** (Feb., 2015) 529–533.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, *Continuous control with deep reinforcement learning*, *arXiv:1509.02971 [cs, stat]* (Sept., 2015). arXiv: 1509.02971.

[17] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, *Trust Region Policy Optimization*, *arXiv:1502.05477 [cs]* (Feb., 2015). arXiv: 1502.05477.

[18] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, *Mastering the game of Go without human knowledge*, **550** (Oct., 2017) 354–359.

[19] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016.

[20] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. de Las Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller, *DeepMind control suite*, tech. rep., DeepMind, Jan., 2018.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal Policy Optimization Algorithms*, *arXiv:1707.06347 [cs]* (July, 2017). arXiv: 1707.06347.

[22] J. Schulman, O. Klimov, F. Wolski, P. Dhariwal, and A. Radford, *openai.com*, Jul, 2017.

[23] S. Fujimoto, H. van Hoof, and D. Meger, *Addressing function approximation error in actor-critic methods*, 2018.

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*, *arXiv:1801.01290 [cs, stat]* (Aug., 2018). arXiv: 1801.01290.

[25] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, *Evolution strategies as a scalable alternative to reinforcement learning*, 2017.

[26] OpenAI, "Openai five." `https://blog.openai.com/openai-five/`, 2018.

[27] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. P. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, *Grandmaster level in starcraft ii using multi-agent reinforcement learning*, *Nature* (2019) 1–5.

[28] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, *Mastering Atari, go, chess and shogi by planning with a learned model*, *Nature* **588** (Dec, 2020) 604–609.

[29] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, *Learning Dexterous In-Hand Manipulation*, *arXiv:1808.00177 [cs, stat]* (Aug., 2018f). arXiv: 1808.00177.

[30] A. El-Fakdi and M. Carreras, *Two-step gradient-based reinforcement learning for underwater robotics behavior learning*, *Robot. Auton. Syst.* **61** (mar, 2013) 271–282.

[31] I. Carlucho, M. De Paula, S. Wang, Y. Petillot, and G. G. Acosta, *Adaptive low-level control of autonomous underwater vehicles using deep reinforcement learning*, *Robotics and Autonomous Systems* **107** (2018) 71–86.

[32] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, *Control of a quadrotor with reinforcement learning*, *IEEE Robotics and Automation Letters* **2** (2017), no. 4 2096–2103.

[33] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, *et. al.*, *Solving rubik's cube with a robot hand*, *arXiv preprint arXiv:1910.07113* (2019).

[34] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver, *Emergence of Locomotion Behaviours in Rich Environments*, *arXiv:1707.02286 [cs]* (July, 2017). arXiv: 1707.02286.

[35] M. Raibert, K. Blankespoor, G. Nelson, and R. Playter, *Bigdog, the rough-terrain quadruped robot*, *IFAC Proceedings Volumes* **41** (2008), no. 2 10822–10825.

[36] X. B. Peng, G. Berseth, and M. Van de Panne, *Terrain-adaptive locomotion skills using deep reinforcement learning*, *ACM Transactions on Graphics (TOG)* **35** (2016), no. 4 1–12.

[37] X. B. Peng, P. Abbeel, S. Levine, and M. van de Panne, *Deepmimic: Example-guided deep reinforcement learning of physics-based character skills*, *ACM Transactions on Graphics (TOG)* **37** (2018), no. 4 1–14.

[38] Z. Xie, P. Clary, J. Dao, P. Morais, J. Hurst, and M. van de Panne, *Learning locomotion skills for cassie: Iterative design and sim-to-real*, in *Proceedings of the Conference on Robot Learning* (L. P. Kaelbling, D. Kragic, and K. Sugiura, eds.), vol. 100 of *Proceedings of Machine Learning Research*, pp. 317–329, PMLR, Oct., 2020.

[39] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, *Learning robust perceptive locomotion for quadrupedal robots in the wild*, *Science Robotics* **7** (Jan, 2022).

[40] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, *Sim-to-real: Learning agile locomotion for quadruped robots*, *arXiv preprint arXiv:1804.10332* (2018).

[41] J. Siekmann, Y. Godse, A. Fern, and J. Hurst, *Sim-to-real learning of all common bipedal gaits via periodic reward composition*, in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7309–7315, IEEE, 2021.

[42] J. Siekmann, K. Green, J. Warila, A. Fern, and J. Hurst, *Blind bipedal stair traversal via sim-to-real reinforcement learning*, 2021.

[43] G. A. Castillo, B. Weng, W. Zhang, and A. Hereid, *Robust feedback motion policy design using reinforcement learning on a 3d digit bipedal robot*, in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5136–5143, IEEE, 2021.

[44] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, *Learning agile and dynamic motor skills for legged robots*, *Science Robotics* **4** (Jan., 2019) eaau5872.

[45] J. Lee, J. Hwangbo, and M. Hutter, *Robust Recovery Controller for a Quadrupedal Robot using Deep Reinforcement Learning*, *arXiv:1901.07517 [cs]* (Jan., 2019). arXiv: 1901.07517.

[46] H. Mania, A. Guy, and B. Recht, *Simple random search of static linear policies is competitive for reinforcement learning*, *Advances in Neural Information Processing Systems* **2018-December** (2018), no. NeurIPS 1800–1809.

[47] A. Rajeswaran, K. Lowrey, E. Todorov, and S. Kakade, *Towards Generalization and Simplicity in Continuous Control*, in *NIPS*, 2017.

[48] L. Krishna, G. A. Castillo, U. A. Mishra, A. Hereid, and S. Kolathaya, *Linear policies are sufficient to realize robust bipedal walking on challenging terrains*, *IEEE Robotics and Automation Letters* **7** (2022), no. 2 2047–2054.

[49] S. Gillen, M. Molnar, and K. Byl, *Combining deep reinforcement learning and local control for the acrobot swing-up and balance task*, *2020 59th IEEE Conference on Decision and Control (CDC)* (2020) 4129–4134.

[50] S. Gillen and K. Byl, *Explicitly encouraging low fractional dimensional trajectories via reinforcement learning*, *2020 4th Conference Of Robot Learning (CORL)* (2020).

[51] S. Gillen and K. Byl, *Mesh based analysis of low fractal dimension reinforcement learning policies*, in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3546–3552, 2021.

[52] S. Gillen, A. Ozmen, and K. Byl, *Direct random search for fine tuning of deep reinforcement learning policies*, *arXiv preprint arXiv:2109.05604* (2021).

[53] S. Gillen and K. Byl, *Leveraging reward gradients for reinforcement learning in differentiable physics simulations*, 2022.

[54] M. Deisenroth and C. E. Rasmussen, *Pilco: A model-based and data-efficient approach to policy search*, in *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472, Citeseer, 2011.

[55] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, 2018.

[56] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, *arXiv:1412.6980 [cs]* (Dec., 2014). arXiv: 1412.6980.

[57] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, *Asynchronous Methods for Deep Reinforcement Learning*, *arXiv:1602.01783 [cs]* (Feb., 2016). arXiv: 1602.01783.

[58] M. W. Spong, *Swing up control of the acrobot using partial feedback linearization **, *IFAC Proceedings Volumes* **27** (Sept., 1994) 833–838.

[59] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, *Distributed Distributional Deterministic Policy Gradients*, *arXiv:1804.08617 [cs, stat]* (Apr., 2018). arXiv: 1804.08617.

[60] M. W. Spong, *Energy Based Control of a Class of Underactuated Mechanical Systems*, *IFAC Proceedings Volumes* **29** (June, 1996) 2828–2832.

[61] J. RandlÃžv, A. G. Barto, and M. T. Rosenstein, *Combining Reinforcement Learning with a Local Control Algorithm*, in *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, (San Francisco, CA, USA), pp. 775–782, Morgan Kaufmann Publishers Inc., 2000.

[62] J. Yoshimoto, M. Nishimura, Y. Tokita, and S. Ishii, *Acrobot control by learning the switching of multiple controllers*, *Artificial Life and Robotics* **9** (May, 2005) 67–71.

[63] L. Wiklendt, S. Chalup, and R. Middleton, *A small spiking neural network with LQR control applied to the acrobot*, *Neural Computing and Applications* **18** (May, 2009) 369–375.

[64] K. Doya, K. Samejima, K.-i. Katagiri, and M. Kawato, *Multiple Model-Based Reinforcement Learning*, *Neural Computation* **14** (June, 2002) 1347–1369.

[65] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines." `https://github.com/openai/baselines`, 2017.

[66] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

[67] N. Talele and K. Byl, *Mesh-based Tools to Analyze Deep Reinforcement Learning Policies for Underactuated Biped Locomotion*, arXiv:1903.1231.

[68] Brendan Ryan / Public domain, *Fractal Dimension Example*, 2020.

[69] N. Talele and K. Byl, *Mesh-based methods for quantifying and improving robustness of a planar biped model to random push disturbances*, *Proceedings of the American Control Conference* **2019-July** (2019) 1860–1866.

[70] C. Oguz Saglam and K. Byl, *Robust Policies via Meshing for Metastable Rough Terrain Walking*, in *Robotics Science and Systems*, 2015.

[71] K. Byl and R. Tedrake, *Metastable walking machines*, *International Journal of Robotics Research* **28** (2009), no. 8 1040–1064.

[72] K. Byl, T. Strizic, and J. Pusey, *Mesh-based switching control for robust and agile dynamic gaits*, *Proceedings of the American Control Conference* (2017) 5449–5455.

[73] C. O. Saglam and K. Byl, *Meshing hybrid zero dynamics for rough terrain walking*, *Proceedings - IEEE International Conference on Robotics and Automation* **2015-June** (2015), no. June 5718–5725.

[74] H. Samet, *The design and analysis of spatial data structures.pdf.* Addison-Wesley, 1990.

[75] T. Gneiting, H. Ševčíková, and D. B. Percival, *Estimators of fractal dimension: Assessing the roughness of time series and spatial data*, *Statistical Science* **27** (2012), no. 2 247–277, [arXiv:1101.1444].

[76] X. Emery, *Variograms of order ω: A tool to validate a bivariate distribution model*, *Mathematical Geology* **37** (2005), no. 2 163–181.

[77] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, *Implementation matters in deep policy gradients: A case study on ppo and trpo*, 2020.

[78] Q. Gallouédec, N. Cazin, E. Dellandréa, and L. Chen, *Multi-goal reinforcement learning environments for simulated franka emika panda robot*, 2021.

[79] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning." `http://pybullet.org`, 2016–2020.

[80] A. Raffin, "Rl baselines3 zoo." `https://github.com/DLR-RM/rl-baselines3-zoo`, 2020.

[81] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3." `https://github.com/DLR-RM/stable-baselines3`, 2019.

[82] A. Kuznetsov, P. Shvechikov, A. Grishin, and D. Vetrov, *Controlling overestimation bias with truncated mixture of continuous distributional quantile critics*, 2020.

[83] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017.

[84] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, *Asynchronous methods for deep reinforcement learning*, 2016.

[85] C. O. Saglam and K. Byl, *Robust policies via meshing for metastable rough terrain walking*, in *Proceedings of Robotics: Science and Systems*, (Berkeley, USA), July, 2014.

[86] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, *Hindsight experience replay*, 2018.

[87] T. Xie, N. Jiang, H. Wang, C. Xiong, and Y. Bai, *Policy finetuning: Bridging sample-efficient offline and online reinforcement learning*, 2021.

[88] E. Todorov, T. Erez, and Y. Tassa, *Mujoco: A physics engine for model-based control*, in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, IEEE, 2012.

[89] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand, *Difftaichi: Differentiable programming for physical simulation*, 2020.

[90] E. Heiden, D. Millard, E. Coumans, Y. Sheng, and G. S. Sukhatme, *NeuralSim: Augmenting differentiable simulators with neural networks*, in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

[91] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem, *Brax - a differentiable physics engine for large scale rigid body simulation*, 2021.

[92] Y. Bengio, P. Simard, and P. Frasconi, *Learning long-term dependencies with gradient descent is difficult*, *IEEE Transactions on Neural Networks* **5** (1994), no. 2 157–166.

[93] L. Metz, C. D. Freeman, S. S. Schoenholz, and T. Kachman, *Gradients are not all you need*, *arXiv preprint arXiv:2111.05803* (2021).

[94] J. Degrave, M. Hermans, J. Dambre, and F. wyffels, *A differentiable physics engine for deep learning in robotics*, *Frontiers in Neurorobotics* **13** (2019).

[95] Y.-L. Qiao, J. Liang, V. Koltun, and M. C. Lin, *Efficient differentiable simulation of articulated bodies*, 2021.

[96] M. A. Z. Mora, M. Peychev, S. Ha, M. Vechev, and S. Coros, *Pods: Policy optimization via differentiable simulation*, in *Proceedings of the 38th International Conference on Machine Learning* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 7805–7817, PMLR, 18–24 Jul, 2021.

[97] D. J. Wales and J. P. K. Doye, *Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms*, *The Journal of Physical Chemistry A* **101** (1997), no. 28 5111–5116.

[98] S. L. McCarty, L. M. Burke, and M. McGuire, *Parallel Monotonic Basin Hopping for Low Thrust Trajectory Optimization.* https://arc.aiaa.org/doi/pdf/10.2514/6.2018-1452.

[99] A. Pourchot and O. Sigaud, *Cem-rl: Combining evolutionary and gradient-based methods for policy search*, 2019.

[100] H. Bharadhwaj, K. Xie, and F. Shkurti, *Model-predictive control via cross-entropy and gradient-based optimization*, 2020.

[101] K. Huang, S. Lale, U. Rosolia, Y. Shi, and A. Anandkumar, *Cem-gd: Cross-entropy method with gradient descent planner for model-based reinforcement learning*, 2021.

[102] R. Dey and F. M. Salem, *Gate-variants of gated recurrent unit (gru) neural networks*, in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1597–1600, 2017.

[103] R. Y. Rubinstein, *Optimization of computer simulation models with rare events*, *European Journal of Operational Research* **99** (1997), no. 1 89–112.