Towards Safe Reinforcement Learning in the Real World

Edward Ahn

CMU-RI-TR-19-56

July 18, 2019

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
David Held, *chair*
John Dolan
Lerrel Pinto

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

Keywords: Robotics, Reinforcement Learning, Safety, Control, Sim-to-Real

To my family

Abstract

Control for mobile robots in slippery, rough terrain at high speeds is difficult. One approach to designing controllers for complex, non-uniform dynamics in unstructured environments is to use model-free learning-based methods. However, these methods often lack the necessary notion of safety which is needed to deploy these controllers without danger, and hence have rarely been tested successfully on real world tasks. In this work, we present methods and techniques that allow model-free learning-based methods to learn low-level controllers for mobile robot navigation while minimizing violations to user-defined safety constraints. We show these learned controllers working robustly in both simulation and in the real world on a 1:10 scale RC car, as well as a full-size vehicle called the MRZR.

Acknowledgments

First, I would like to thank my advisor, Dave Held, for his mentorship and guidance, as well as an opportunity to work on an interesting and important problem in robotics. I would also like to thank my committee members, John Dolan and Lerrel Pinto, for providing invaluable advice for my thesis. I am also sincerely thankful for everyone in the entire R-PAD lab for their support and help; I am honored to have been a member of this lab.

Finally, thank you to all of my family and friends. I could not have accomplished as much without the moral support I received especially during stressful, sleepless nights. This memorable, unique journey has only been possible because of these people.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Tremendous advances in autonomy have been made in the past few decades. Driven by innovations in computing, autonomy is increasingly being incorporated in our everyday lives, from facial recognition technology for security purposes to the budding self-driving car industry. While autonomy has shown great promise, factors impeding the full adoption of autonomous systems can be summarized as a lack of a concrete notion of safety, especially with the use of data-driven techniques that are highly unpredictable and lack guarantees on correctness but at the same time are necessary for generalization in uncertain, unstructured and dynamic environments. Historically, when the human was in the loop, assurance in autonomous systems was maintained via rigorous design processes and compliance through testing. These approaches assume, unfortunately, that the system does not learn and evolve over time.

This thesis addresses this lack of formal safety in autonomous systems today by investigating means of incorporating safe learning in systems, combined with offline policy verification as well as runtime verification via rigorous safety monitors during test time. Through this integration of safety measures, we look to show that it is possible to deploy systems that are both formally and empirically safe. Though applicable in a wide range of areas in robotics, we narrow our focus to developing safe algorithms for control of autonomous vehicle systems.

For a deeper insight into the problems of addressing safety in autonomous vehicles,

consider control for a mobile robot in a non-uniform terrain, slippery environment. While vehicle dynamics of mobile robots moving at low speeds is relatively simple, moving at high speeds in these environments often induces non-trivial lateral forces, causing slip that can be hard to model, particularly when environments change over time. Even though these problems can be mitigated with accurate system identification, dealing with dynamic environments relies on a lot of engineering that is not generalizable to other applications.

An attractive solution is to use model-free control. Without relying on the accuracy of a model of the environment, model-free control methods can essentially learn the environment dynamics, which can be powerful for planning in uncertain environments. Of course, learning control often warrants the need for the use of function approximators like neural networks, which are hard to verify. The work in this thesis revolves around making this approach safe through safe methods of learning, verifying trained networks and runtime monitoring during test time.

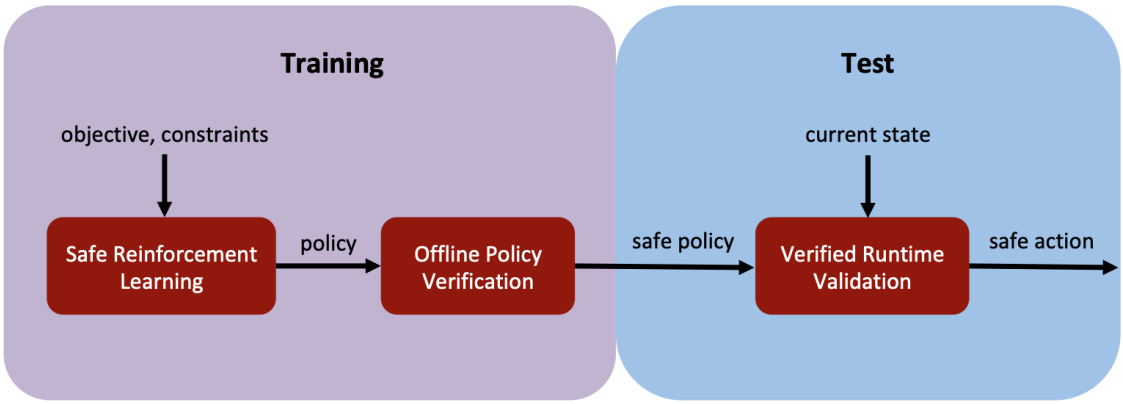


Figure 1.1: Safe control strategy overview. In the training phase, we use safe reinforcement learning algorithms that we investigate in this thesis combined with an offline neural network verification tool to produce a safe policy. During test time, a safety monitor based on verified runtime verification uses this safe policy as well as the current state to either pass on the action that the policy computed if deemed safe, or if unsafe, produce a fallback action.

1.2 Current Challenges and Scope

In this thesis, we specifically focus on low-level control, where we control the agent to follow lines to a given waypoint. Waypoints are computed via a high-level planner, which is just an off-the-shelf path planner such as RRT*. For our low-level controller, we use model-free control via model-free reinforcement learning, to remove the dependency on having an accurate model of the environment. This hierarchy of planners is a common approach used in robotic systems, such as the winners of the DARPA Grand Challenges, Stanley [21] and Tartan Racing [23].

The overarching goal of this thesis is to show that model-free control methods can be safe to deploy in the real world. Our solution to designing safe methods for model-free control in the real world is three-pronged: we use safe reinforcement learning methods, neural network verification to verify our trained policy networks, and formal runtime verification to monitor our runtime performance. See Figure 1.1 for an overview of our strategy. All three are challenging for the following reasons:

- Safe reinforcement learning: limited success in developing safe model-free control methods in the real world
- Neural network verification: larger networks take prohibitively long to verify
- Runtime verification: difficult to formally model complex dynamical systems; most runtime verification uses models that assume the agent to be a point mass

Figure 1.2: The MRZR 4. This highly mobile off-road vehicle developed by Polaris Industries is a military-grade vehicle used for deployment in missions with difficult terrains and a need for ultra-light tactical mobility. We test our algorithms on this vehicle.

In this work, we reduce our scope to developing safe reinforcement learning methods that are robust enough to train and run in the real world. Preliminary work has been attempted to integrate our trained policies with neural network verification tools and safety monitors at runtime, but experiments to see the effectiveness of doing so is reserved for future work; see Chapter 4 for more information.

Because this work is in conjunction with the DARPA Assured Autonomy initiative, we develop these control methods to deploy on the MRZR 4 (see Figure 1.2). While we were successfully able to deploy trained policies onto the MRZR, results on the MRZR are not mentioned in this document because of legal matters. Instead, all data and results in this thesis were collected from a custom 1:10 scale RC car (see Figure 1.3). Its specs are outlined in Appendix B.

Figure 1.3: The "FFAST" vehicle. This vehicle is a 1:10 scale rear-wheel drive front steering RC car designed as a test platform for testing dynamic motion planning and control algorithms.

1.3 Contributions and Organization

The main contributions of this thesis are listed as follows:

- Implementation of a high-fidelity vehicle model for simulation purposes, proven to robustly train policies using model-free reinforcement learning algorithms that transfer directly to the MRZR
- Use of a combination of techniques that we outline in this work to aid sim-to-real transfer and minimize fine-tuning

- To the best of our knowledge, the first use of CPO outside of toy simulation environments
- The ability to safely learn optimal drifting behaviors when following circles at high speeds that are unattainable without drifting

This thesis is organized as follows. Chapter 2 discusses relevant background and related work to this thesis, which includes explanation of vehicle models and reinforcement learning. Then, Chapter 3 provides an explanation of how model-free control was used to train policies that transferred well into the real world with minimal fine-tuning. We delve into both simulation and real world results for following arbitrary trajectories at a target velocity and following circles at high speeds in this chapter. Finally, in Chapter 5, we summarize the thesis contributions and present the ongoing future work. Note that all metrics in this thesis are in standard SI units (such as meters, seconds).

Chapter 2

Background and Related Work

In this chapter, we review background and related work for vehicle models needed to build a simulation pipeline, autonomous driving in slippery environments, safe policy learning and sim-to-real approaches to make it possible to deploy trained policies in simulation directly to the real world.

2.1 Vehicle Models

In reinforcement learning, a model-free algorithm is an algorithm that does not assume having a model of the dynamics of the robot interacting with the world. Instead, this class of algorithms simply use sampled trajectory data to perform optimization. However, for our purposes, we want to train policies that take a state input and output an action that are robust to different environments to allow direct sim-to-real transfer in order to minimize expensive fine-tuning processes. Therefore, an accurate model of how our agent interacts with the world is needed, so that our simulation is able to simulate what the agent may see in the real world.

To implement a simulator accurate enough to train robust policies, a suitable vehicle model must be chosen. Many vehicle models with varying fidelity exist. The trade-off between using a high-fidelity model versus a lower-fidelity model, lies in the increasing nonlinearity of the resultant dynamic equations. While nonlinearity prevents linear feedback controllers from being implemented, for our case this does not matter since we do not use linear controllers, but rather neural network controllers.

In this section, we examine the dynamic bicycle model with a nonlinear brush tire model, which we used to build a simulation for training policies.

Note that we investigate these models under the assumption that the control inputs to the model are the front wheel steering angle δ_f and commanded velocity $v = R\omega$, where R is the radius of the wheel and ω is the angular wheel velocity. Formally, our state X and our control inputs U are parametrized as follows:

$$X = \begin{bmatrix} x & y & \psi & \dot{x} & \dot{y} & \dot{\psi} \end{bmatrix}^T \tag{2.1}$$

$$U = \begin{bmatrix} R\omega & \delta_f \end{bmatrix}^T \tag{2.2}$$

where x and y are the coordinates of the center of mass in the inertial frame (X, Y), ψ is the inertial heading of the center of mass and $\dot{x}$, $\dot{y}$ and $\dot{\psi}$ are the corresponding velocities of those components.

2.1.1 Kinematic Bicycle Model

The bicycle model at its core is relatively simple; the tires on each axle are lumped together to form a 'bicycle' with two wheels that have twice the cornering stiffness and force capability. While this means that load transfer effects are ignored, this approach has been shown to be sufficient when modeling drifting [16].

Here, we discuss the kinematic bicycle model which is necessary to explain the dynamic bicycle model. See Figure 2.1. Let v be the speed of the vehicle, l_r and l_f be the distances from the center of mass of the vehicle to the rear and front axles, respectively, and β be the angle of the velocity with respect to the longitudinal axis of the car. Note that δ_f and δ_r are the steering angles of the front and rear wheels, but for our robot, since there is no rear steering, $\delta_r = 0$.

Then, based on geometric relations, we can deduce the following nonlinear contin-

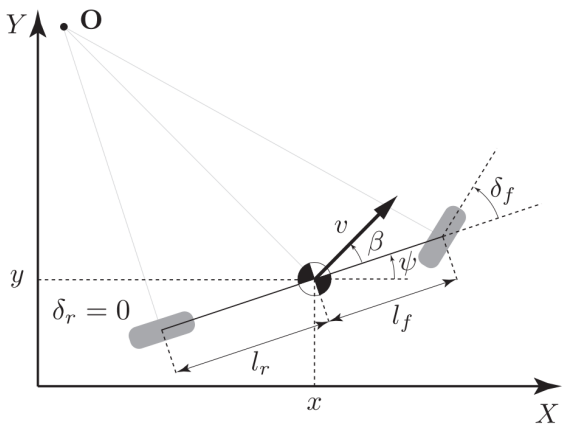


Figure 2.1: Kinematic Bicycle Model

uous time equations that describe the kinematic bicycle model [15]:

$$\dot{x} = v\cos(\psi + \beta) \tag{2.3}$$

$$\dot{y} = v\sin(\psi + \beta) \tag{2.4}$$

$$\dot{\psi} = \frac{v}{l_r}\sin(\beta) \tag{2.5}$$

$$\dot{v}_x = \dot{\psi} v_y \tag{2.6}$$

$$\dot{v}_y = -\dot{\psi} v_x \tag{2.7}$$

$$\ddot{\psi} = 0 \tag{2.8}$$

$$\beta = \tan^{-1}\left(\frac{l_r}{l_f + l_r}\tan(\delta_f)\right) \tag{2.9}$$

Evidently, the system identification on the kinematic bicycle model is simple - only two parameters need to be estimated, l_r and l_f. These parameters are also easily measured. However, this model is limiting because it assumes that the vehicle experiences small lateral forces that are characterized by $\frac{mv^2}{R}$, where m is the mass of the vehicle and $\frac{1}{R}$ is the curvature of the road the vehicle is following. In an intuitive sense, the model assumes that the velocity vectors at the wheels are in the direction of the orientation of the wheels, implying that all slip angles are assumed to be zero. Again, this holds when lateral forces are small with small values of v. Furthermore, the curvature must also be changing slowly, or else larger lateral forces can be induced.

A dynamic bicycle model is necessary to the extent that these assumptions do not hold.

2.1.2 Dynamic Bicycle Model

In a dynamic bicycle model, lateral forces are taken into account by building a model that takes into account the forces that are exerted on the vehicle (not just geometric relations). See Figure 2.2. The following equations describe the dynamic bicycle model [15]:

$$\dot{x} = v\cos(\psi + \beta) \tag{2.10}$$

$$\dot{y} = v\sin(\psi + \beta) \tag{2.11}$$

$$\dot{\psi} = \frac{v}{l_r}\sin(\beta) \tag{2.12}$$

$$\dot{v}_x = \dot{\psi}\dot{y} + \frac{1}{m}(F_{xr} - F_{yf}\sin\delta_f) \tag{2.13}$$

$$\dot{v}_y = -\dot{\psi}v_x + \frac{1}{m}(F_{xr}\cos\delta_f + F_{yr}) \tag{2.14}$$

$$\ddot{\psi} = \frac{1}{I_z}(l_f F_{yf} - l_r F_{yr}) \tag{2.15}$$

The first three equations stem from the kinematic bicycle model. Note that $\dot{x} \neq v_x$ and $\dot{y} \neq v_y$; due to an unfortunate abuse of notation, $\dot{x}$ and $\dot{y}$ refer to the velocity of the center of mass in the inertial frame (X, Y), whereas v_x and v_y are the velocities of the center of mass with respect to the vehicle's longitudinal and lateral axes, respectively. In addition, $\dot{\psi}$ is the yaw rate, m the vehicle's mass and I_z the moment of inertia for the vehicle. Finally, F_{yr} and F_{yf} are the lateral forces on the rear and front tires, respectively, and F_{xr} is the longitudinal force on the rear tire, which is generated by the rear motor. The lateral and longitudinal forces on the tire can be computed in a variety of different ways using different tire models. We investigate the linear and brush tire models below.

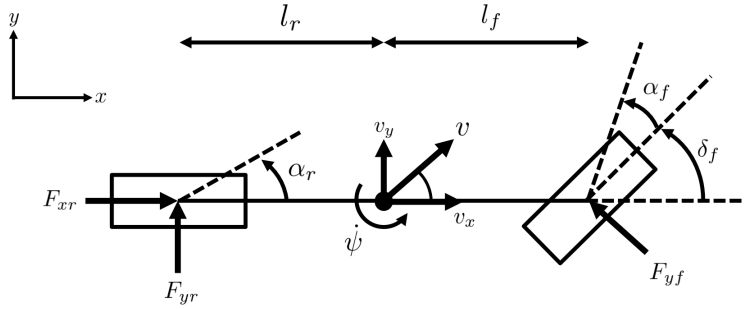


Figure 2.2: Dynamic Bicycle Model

Linear Tire Model

In the linear tire model, the longitudinal forces and lateral forces on the tire are simply defined as

$$F_{xi} = C_x \kappa \tag{2.16}$$

$$F_{yi} = -C_{\alpha_i} \alpha_i \tag{2.17}$$

where $i \in \{f, r\}$, α_i is the tire slip angle, C_x is the tire stiffness and C_{α_i} is the tire cornering stiffness. For our purposes, since we use the same tires, we let $C_\alpha := C_{\alpha_r} = C_{\alpha_f}$. κ is the slip value, which is defined to be

$$\kappa = \frac{R\omega - v_x}{v_x} \tag{2.18}$$

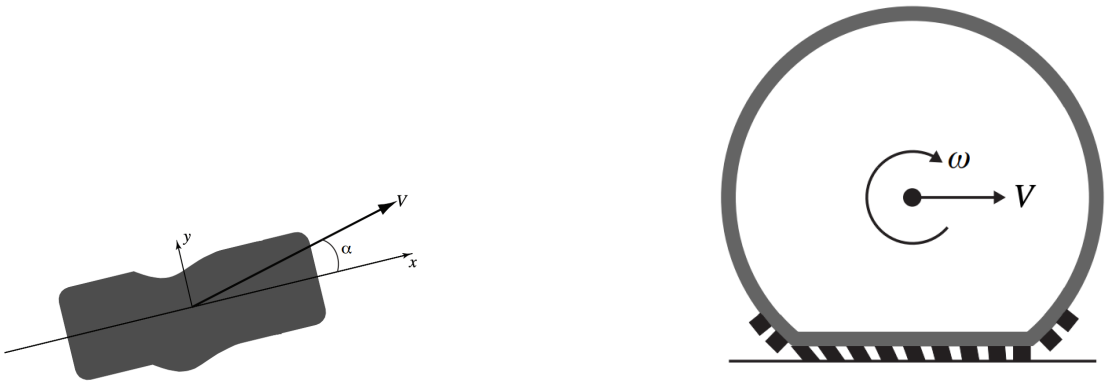
where v_x is the velocity of the vehicle with respect to its longitudinal axis, R the radius of the wheel and ω the angular velocity of the wheel. We can see that to control the vehicle's acceleration, the linear velocity of the motor $R\omega$ needs to be increased or decreased by the motors. In addition, we can deduce that if $\kappa = -1$ the vehicle is slipping, if $\kappa = \infty$ the vehicle is skidding and if $\kappa = 0$ the vehicle has no slip or skid.

Finally, the values of α_f and α_r can be found geometrically:

$$\alpha_f = \arctan2(v_y + l_f\dot{\psi},\ v_x) - \delta_f \tag{2.19}$$

$$\alpha_r = \arctan2(v_y - l_f\dot{\psi},\ v_x) \tag{2.20}$$

These values are the slip angles for each wheel axle, as denoted in Figure 2.2.



(a) Lateral deformation

(b) Longitudinal deformation

Figure 2.3: Tire deformation that is captured by the brush tire model to accurately compute tire forces.

Brush Tire Model

Unlike the linear approximation above, the brush tire model is a more sophisticated tire dynamics model that incorporates nonlinear relationships between lateral forces and slip angles. At a high level, the brush tire model assumes that the part of the tire in contact with the road consists of independent springs called brushes that undergo deformation and resist with a constant stiffness.

The equations below summarize the formulas that compute the tire forces, where F_{zi} is the load on the corresponding wheel axle with $i \in \{f, r\}$ and μ and μ_s are the coefficients of kinetic and static friction, respectively. For the full derivation, see Rami's doctoral thesis [16]. For the front wheels,

$$F_{yf} = \begin{cases} -C_\alpha \tan\alpha_f + \frac{{C_\alpha}^2}{3\mu F_{zf}} |\tan\alpha_f| \tan\alpha_f - \frac{{C_\alpha}^3}{27\mu^2 {F_{zf}}^2} \tan^3\alpha_f & |\alpha_f| \leq \alpha_{sl} \\ -\mu F_{zf} \ \text{sign}(\alpha_f) & |\alpha_f| > \alpha_{sl} \end{cases} \tag{2.21}$$

where the slip angle α_{sl} equals

$$\alpha_{sl} = \tan^{-1} \frac{3\mu F_{zf}}{C_\alpha} \tag{2.22}$$

Note that because the vehicle is rear-wheel drive, no forces are exerted on the front tires in the longitudinal direction, meaning $F_{xf} = 0$. For the rear wheels we have

$$F_{xr} = \frac{C_x}{\gamma}\frac{\kappa}{1+\kappa}F \tag{2.23}$$

$$F_{yr} = -\frac{C_\alpha}{\gamma}\frac{\tan\alpha_r}{1+\kappa}F \tag{2.24}$$

where γ, the combined slip value, and F equal

$$F = \begin{cases} \gamma - \frac{1}{3\mu F_{zr}}\gamma^2 + \frac{1}{27\mu^2 {F_{zr}}^2}\gamma^3 & \gamma \leq 3\mu F_{zr} \\ \mu_s F_{zr} & \gamma > 3\mu F_{zr} \end{cases} \tag{2.25}$$

$$\gamma = \sqrt{{C_x}^2\left(\frac{\kappa}{1+\kappa}\right)^2 + {C_\alpha}^2\left(\frac{\tan\alpha_r}{1+\kappa}\right)^2} \tag{2.26}$$

Conveniently, the brush tire model only adds two parameters, F_{zr} and F_{zf}, that need to be estimated. As a result, there is not much cost added to using a brush tire model as opposed to a linear tire model.

2.2 Autonomous Rallying

Autonomous rallying, involving controlling a vehicle under slippery conditions with non-trivial lateral forces, has been studied extensively. Approaches largely boil down to model-based approaches such as model-predictive control and trajectory optimization, but research in model-free control in these environments is lacking.

GeorgiaTech's AutoRally group [5] has used MPC in a variety of different ways to follow a circular dirt track at extremely high speeds using a 1:5 scale RC car. Their initial work with this platform involved using stochastic sampling of trajectories to control the vehicle aggressively [25]. Other teams have also used MPC successfully, such as Keivan's [10]. However, MPC is computationally expensive since it replans at each step online.

To mitigate this issue, some researchers have tried to learn parameterized control policies with expert demonstrations. The AutoRally group recently tried using imitation learning to learn end-to-end control with visual inputs [13]. For learning

control, they used DAgger [17] with an MPC expert, as opposed to a human expert who may not be able to provide stable and consistent high-frequency feedbacks while the learner is driving the car. Though this approach is successful, it does not easily generalize to new environments because MPC is model-based. Lau [11] showed that it is possible to leverage a single demonstration to learn a linear control policy using policy gradients to do aggressive maneuvers. However, having a strictly linear policy is a drawback even though a model-free learning method was used.

Finally, the most similar work to ours is work by Cutler [4], who initializes a policy with a simple model and fine-tunes the policy with PILCO, a model-based algorithm that uses Gaussian processes to model uncertainty in environment dynamics. Instead of using a model-based algorithm which depends on accurate modeling of uncertainty as well as a handcrafted, complex reward function, we use a model-free reinforcement learning algorithm with a simple objective.

2.3 Safe Policy Learning

Another goal of this project is to guarantee some sort of safety in our learning process. Though safe but aggressive driving has not been studied well to the best of our knowledge, safety in robotics is a very well studied area. Safety is usually achieved by motivating the agent to act safely, making sure a policy never leaves a safe subset of parameters or some constrained optimization with safety constraints.

Intrinsic fear [12] is a technique that can be used to motivate (or guard) agents from periodic catastrophes in training. Agents with intrinsic fear possess a fear model trained to predict probability of imminent catastrophe. The resulting score then used to penalize Q-learning objective. We do not use this approach because there is no formal notion of safety here.

Bayesian optimization with safety constraints [3] is another approach to find model parameters under a safe subset of parameters, by using Gaussian processes to explore parameter space. However, an accurate model of the environment is required, and in this work we want to mitigate that dependency. Similarly, Held [6] developed a probabilistic framework to have a probabilistically safe policy transfer during learning, in which expected return is maximized while containing expected damage within some limit. Unfortunately, assumptions that are made to allow this technique to

work for manipulation do not apply to our aggressive driving domain.

In our work, we use Constrained Policy Optimization [1], which is a model-free reinforcement learning algorithm with probabilistic guarantees on the number of user-defined safety constraint violations. This approach is explained in more detail in Chapter 3. Because this algorithm is model-free, we do not need to have an accurate model of the environment. In addition, the ability to define safety constraints allows our reward function to be very simple, with minimal reward engineering.

2.4 Sim-to-Real

In our work, we investigate ways to perform sim-to-real so that we can reduce expensive real world fine-tuning and leverage a lightweight simulation that we built. Sim-to-real is the process of transferring a policy trained in simulation directly into the real world, and is often difficult if state distributions seen in simulation differ from the real world. We outline relevant techniques to truncate this 'reality' gap in this section.

While the best approach is to perform better system identification in order to have a more realistic simulator, this is often difficult because the real world is challenging to model. Instead, we look at ways to make our policy more robust, so that it can handle a wide state distribution. A popular and simple approach to do this is domain randomization [22]. Domain randomization involves training a network with noise added to its input. With enough variability while training in the simulator, the real world may appear to the model as just another variation. Similarly, Peng adds noise to dynamics parameters [14], though a recurrent policy is also used so that a history of the past states and actions can be used to infer the dynamics of the system. The best demonstration of these techniques working to produce robust policies is work by OpenAI that involves learning dextrous in-hand manipulation policies using a physical Shadow Dextrous Hand [2]. The authors used a wide variety of randomization techniques to achieve accurate in-hand object reorientation. In our work we also use a variety of randomization techniques to produce a robust policy.

Rather than learning a more robust policy, some approaches simply learn dynamics values online. Yu introduced one such approach that involves learning a policy that takes in state and environment parameters, and a function for online system

identification, which takes in a recent history of state and action tuples to predict values of environment parameters [26]. We do not use this approach yet, but mentioned in future work in Chapter 4.

Chapter 3

Model-free Control

3.1 Introduction

Traditional methods of planning and control often suffer when under continuous, complex robotic tasks. Oftentimes engineering a cost function or generating a good nominal trajectory is difficult. In our application of mobile robot driving, this is also true. Since our vehicle will run in non-uniform, slippery terrain at high speeds, coming up with a robust, optimal control policy can become intractable.

Through success in robot simulation or benchmarks like Atari, model-free reinforcement learning has emerged as a popular choice for researchers to develop control policies for high-dimensional sequential decision making problems like this. These methods formulate the problem as a Markov Decision Process and attempt to optimize some reward function [7], which is ideally simple to prevent finding local optima. However, a critical challenge to model-free reinforcement learning is making sure policies are safe. Because reinforcement learning inherently uses a trial-and-error approach to finding successful control policies, we cannot formally guarantee anything about the agent's behavior during test-time. This is a risk for anyone performing robotic tasks in the real world.

While approaches to perform safe reinforcement learning have been developed, most do not transfer well into the real world and are only safe in simulation. In this chapter, we attempt to formulate our problem in a way that will allow us to use Constrained Policy Optimization (CPO) [1] to safely control the robot to drive in

a circular trajectory, either at a target velocity or as fast as possible. Doing so will allow us to place some guarantees on the safety of our policy, an important step in the direction of assured autonomy.

For faster convergence in training, we use the simulation that we developed in the previous chapter, which we know will accurately simulate the vehicle dynamics of our robot. Then, we either perform sim-to-real transfer and directly test policies on our robot, or we fine-tune policies by training in the real world first before testing.

3.2 Related Work

3.2.1 Preliminaries

In reinforcement learning, a Markov Decision Process is a widely-used framework for solving sequential decision making problems. Specifically, the framework allows an optimal policy to be computed which maximizes long-term reward over a sequence of decisions. An MDP is defined simply as a tuple $(\mathcal{S}, \mathcal{A}, r, P, \rho, \gamma)$ where:

- $\mathcal{S}$ is the set of all states
- $\mathcal{A}$ is the set of all actions, which may be continuous or discrete
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function (ex. $P(s'|s, a)$ is the probability of transitioning to state s' given that the agent was previously at state s and took action a at state s)
- $\rho : \mathcal{S} \to [0, 1]$ is the initial state distribution
- $\gamma \in (0, 1)$ is the discount factor.

Note that an MDP makes the control problem tractable by using the Markov Property, which assumes that a state s_{t+1} is only dependent on its predecessor state s_t and the action a_t that was taken to reach state s_{t+1}. That is, put formally,

$$p(s_{t+1}|s_t, a_t, \ldots, s_0, a_0) = p(s_{t+1}|s_t, a_t)$$

Finally, a stochastic policy is defined to be a function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\pi(a|s)$ denotes the probability of selecting action a in the state s. The set of all policies is

denoted as Π.

Reinforcement learning algorithms aim to select a policy π which maximizes some performance measure, $J(\pi)$, which is typically defined to be the infinite (or finite) horizon expected discounted return. In other words,

$$J(\pi) := \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t, a_t)\right] \tag{3.1}$$

where $\gamma \in [0, 1)$ is the discount factor and $\tau = (s_0, a_0, s_1, \dots)$ is a trajectory. $\tau \sim \pi$ means that the trajectory was sampled from a distribution of trajectories defined by π; in other words, $s_0 \sim \rho$, $a_t \sim \pi(\cdot, s_t)$ and $s_{t+1} \sim P(\cdot|s_t, a_t)$. Also of interest is the discounted future state distribution, $d^\pi(s)$, defined by $d^\pi(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_t = s|\pi)$.

We use the following standard definitions of the state-action value function Q_π, the value function V_π and the advantage function A_π:

$$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1},a_{t+1},\dots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l}, a_{t+l})\right] \tag{3.2}$$

$$V_\pi(s_t) = \mathbb{E}_{a_t,s_{t+1},\dots}\left[\sum_{l=0}^{\infty}\gamma^l r(s_{t+l}, a_{t+l})\right] \tag{3.3}$$

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{3.4}$$

where $t \geq 0$.

3.2.2 Policy Gradient Algorithms

In policy gradient algorithms with function approximation, a parametrized policy $\pi(a|s, \boldsymbol{\theta})$ is learned that can select actions without consulting a value function, where $\boldsymbol{\theta}$ is the policy's parameter vector. The policy parameter is learned based on the gradient of some performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter, which allows gradient ascent to be performed (since we want to maximize performance).

Using policy gradient algorithms has advantages, such as being able to approach an optimal stochastic policy [19]. In addition, value-based methods such as Q-learning [24] require a discretization of the action space, while policy gradient methods do not.

Sutton proved in the policy gradient theorem that the policy gradient does not depend on the gradient of the changing state distribution [20]. In addition, he showed that the policy gradient could be simplified into an analytic expression:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s} \mu(s) \sum_{a} Q_{\pi}(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}) \tag{3.5}$$

where μ is the on-policy distribution under π.

3.2.3 Trust Region Policy Optimization

Trust region methods for reinforcement learning is one approach that has shown great promise for optimizing neural network policies that often suffer from performance collapse after bad updates [8]. To guarantee monotonic policy improvement, trust regions are used to optimize an analytically computed lower bound on policy performance. This approach has shown to reach state-of-the-art performance in conventional deep RL benchmarks via Trust Region Policy Optimization (TRPO) [18].

Specifically, TRPO, an on-policy, model-free reinforcement learning algorithm, has policy updates of the form

$$\pi_{k+1} = \arg\max_{\pi \in \Pi_{\theta}} \mathbb{E}_{s \sim d^{\pi_k}, a \sim \pi} \left[A^{\pi_k}(s,a) \right]$$
$$\text{s.t. } \bar{D}_{KL}(\pi || \pi_k) \leq \delta$$

where $\bar{D}_{KL}(\pi||\pi_k) = \mathbb{E}_{s \sim \pi_k}[D_{KL}(\pi||\pi_k)[s]]$ and $\delta > 0$ is the optimization step size. The set $\pi_{\theta} \in \Pi_{\theta} : \bar{D}_{KL}(\pi||\pi_k) \leq \delta$ is called the trust region.

In implementation, TRPO is simply a truncated natural gradient descent with a line search to enforce the trust region. Natural gradient descent [9] is a form of gradient descent that is invariant to model parameterizations because steps in the gradient descent are taken with respect to the KL divergence (or, in practice, the second order approximation of the KL divergence at $\pi_k = \pi$, called the Fisher information matrix, since for small step sizes this approximation is sufficient). Because natural gradient descent involves inverting this Fisher information matrix which can be prohibitively expensive, truncated natural gradient descent approximates this

inversion via the conjugate gradient method.

Note that due to approximations from theory, policy improvements are not strictly monotonic. However, TRPO is a very stable learning algorithm.

3.2.4 Constrained Policy Optimization

Because these trust region methods guarantee monotonic policy performance, this approach can be adapted to Constrained MDPs (CMDPs), which are MDPs augmented with constraints that restrict the allowable policies for these MDPs. Specifically, the MDP is augmented with a set C of auxiliary cost functions (constraints), $C_1, \ldots, C_m$, where each function $C_i : S \times A \times S \to \mathbb{R}$ is, much like the reward function, a mapping between transition tuples to costs. The MDP is also augmented with limits $d_1, \ldots, d_m$ that correspond to their respective cost functions.

Let $J_{C_i}(\pi)$ denote the expected discounted return of a policy π with respect to the cost function C_i; in other words, let $J_{C_i}(\pi) = \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t C_i(s_t, a_t, s_{t+1})$. Constrained Policy Optimization (CPO) [1], is a local policy search method that attempts to find the optimal policy $\pi^* = \arg\max_{\pi\in\Pi_C} J(\pi)$, where $\Pi_C = \pi \in \Pi_\theta : \forall i,\ J_{C_i}(\pi) \leq d_i$ is the set of policies that satisfy all constraints in the CMDP.

Like TRPO, CPO is an on-policy model-free reinforcement learning algorithm that has policy updates of the form

$$\begin{aligned}\pi_{k+1} = \arg\max_{\pi\in\Pi_\theta} & \mathbb{E}_{s\sim d^{\pi_k}, a\sim\pi}\left[A^{\pi_k}(s,a)\right] \\ \text{s.t. } & J_{C_i}(\pi_k) + \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi_k}, a\sim\pi}\left[A^{\pi_k}_{C_i}(s,a)\right] \\ & \bar{D}_{KL}(\pi||\pi_k) \leq \delta\end{aligned}$$

where $A^{\pi_k}_{C_i}(s,a)$ is the advantage function for policy π_k with respect to the constraint C_i, similar to the normal advantage function A^π but replacing the reward function R with the constraint C_i.

In implementation, assuming small step sizes, the objective and cost constraints are well-approximated by linearizing around π_k, and, like TRPO, the second order approximation of KL divergence at $\pi_k = \pi$ (ie. the Fisher information matrix) is used. Since the Fisher information matrix is always positive semi-definite, this optimization

problem is convex and can be solved using duality. In the current implementation, only one constraint is handled, but is possible to optimize over an arbitrary number of constraints.

3.3 Simulation Environment

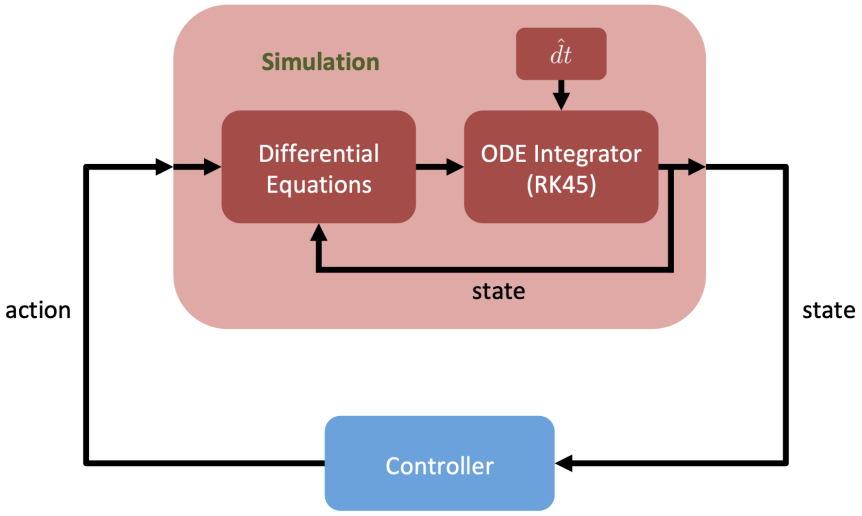


Figure 3.1: Simulation pipeline. The simulation takes action inputs to produce new states using the differential equations provided by the vehicle model, and the planner takes new states and computes new actions.

To train policies with minimal real world training to mitigate issues of sample efficiency, we built a simulation based on the dynamic bicycle model with a brush tire model mentioned in Chapter 2. In this section, we briefly outline the simulation pipeline that was built. Note that (also in Chapter 2) the control inputs to our model are the commanded steering angle δ_f and the commanded linear velocity of the rear wheels, v_x. For simplicity in notation, define $\delta := \delta_f$. Note that the linear velocity $v = R\omega$.

The simulation pipeline can be seen visually in Figure 3.1. At a high level, the planner takes the current state as an input and outputs a corresponding action. The action and the current state are fed into differential equations from the vehicle dynamics model to produce derivatives of the state, which the ODE integrator uses

to compute the next state. The ODE integrator also takes in a time step dt, which simulates control frequency of the planner (how fast the controller can output actions).

Table B.2 in Appendix B.3 shows the parameters we used to build the simulation, for training policies for both the RC car and the MRZR. Figure 3.2 shows a typical screen display from the simulation.

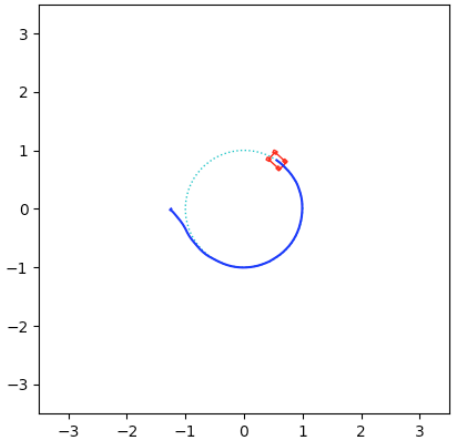


Figure 3.2: Simulation rendering. We see the robot following a circle of radius 1 meter at a velocity of 1.0 m/s with a randomized initial state. This policy was trained using TRPO.

3.4 Problem Formulation

In both our simulation and on the actual robot, the planner receives the state in the form $\left(x, y, \psi, \dot{x}, \dot{y}, \dot{\psi}\right)$, where (x, y) are the Cartesian coordinates of the robot's position with respect to the inertial frame, ψ is the yaw (heading) of the robot, $\dot{x}$ and $\dot{y}$ are the velocities of the robot with respect to the longitudinal and lateral axes of the vehicle and $\dot{\psi}$ is the angular velocity of the robot in the robot frame. We want to develop a controller that will take states as an input and output controls of the form (v_x, δ) where v_x is the commanded wheel velocity and δ is the commanded steering angle. We work under the assumption that a high-level path planner exists, and the controller's job is to simply follow an already planned trajectory.

Intuitively, a trajectory can be broken down into a sequence of line segments

and arcs of varying curvature. Therefore, to simplify the problem, we design the controller to follow a straight line and circles of arbitrary curvature. Note that in all of these experiments, we fixed the curvature to 1 (i.e., follow a circle of radius 1). Our work naturally extends to following arcs of varying curvature. In addition, our agents are only trained to follow the straight line $y = 0$ and a circle centered at $(0, 0)$. To follow arbitrary lines and arcs, we use coordinate transformations to transform these arbitrary lines and arcs into $y = 0$ and circles centered at $(0, 0)$, respectively.

A key insight into training policies is to use relative state and not absolute state. Since we parameterize the policy with a neural network as our function approximator, we want the agent to recognize that being on one part of the straight line or the arc is the same as being on another part of the straight line or the arc. Put differently, the control inputs should not depend on the absolute state of the robot, since absolute states depend on the reference frame. Below, we explain how we transformed absolute states into the relative states that our planners were trained with by defining a function T that maps absolute states to relative states.

Straight Line Following

Let the absolute state $s = \left(x, y, \psi, \dot{x}, \dot{y}, \dot{\psi}\right)$. For straight line following, we define the transformation $T_S : \mathbb{R}^6 \to \mathbb{R}^5$ as follows:

$$T_S(s) = \left(y, \psi, \dot{x}, \dot{y}, \dot{\psi}\right) \tag{3.6}$$

The transformation T simply discards the parameter x, so that the input to the network does not depend on how far the agent has driven along the line $y = 0$.

Circle Following

Let the absolute state $s = \left(x, y, \psi, \dot{x}, \dot{y}, \dot{\psi}\right)$. For circle following, we define the transformation $T_C : \mathbb{R}^6 \to \mathbb{R}^4$ as follows:

$$T_C(s) = \left(\Delta x, \theta, \dot{\Delta x}, \dot{\theta}\right) \tag{3.7}$$

The derivations and formulas for computing the four relative state parameters can be found in Appendix A. These parameters were derived based on Figure 3.3, where

the parameters stay the same if the agent's yaw with respect to the tangent and the agent's distance to the closest point on the circle remain the same.

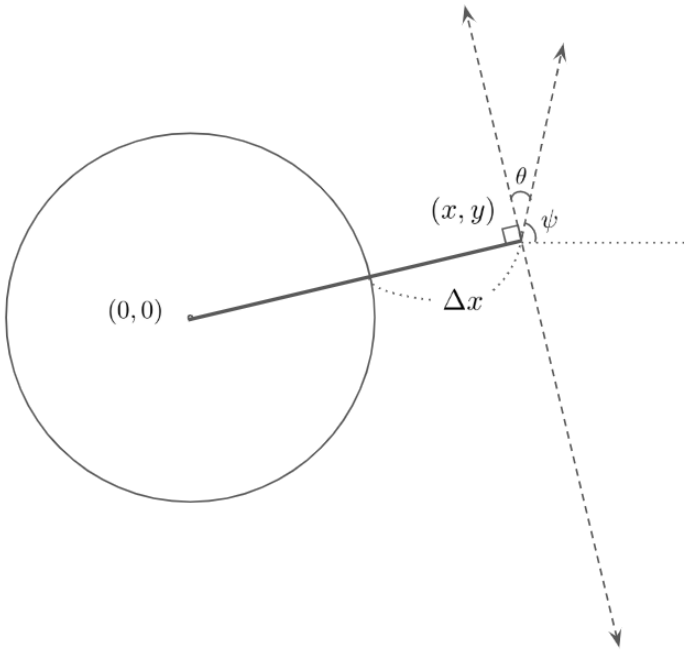


Figure 3.3: State space for circle following.

To train this neural network policy, we also need to define a reward function for our reinforcement learning algorithms to optimize. The key idea is to make reward functions simple to avoid reaching local optima during training. In all of the experiments conducted, we want our vehicle to follow a straight line or circle at a target velocity or, or follow a circle as fast as possible to see if the agent will potentially learn drifting behavior.

For straight line following at a target velocity, we used the reward function

$$r(s, a) = -|y| - \lambda_1(v - v_f)^2 \tag{3.8}$$

and for circle following at a target velocity, we used the reward function

$$r(s, a) = -|\Delta x| - \lambda_1(v - v_f)^2 - \lambda_2 \max\left(0, |\theta| - \frac{\pi}{2}\right)^2 \tag{3.9}$$

where λ_1 and λ_2 are hyperparameters and v_f is the target velocity. The first terms in both functions are distance penalties, and the second terms in both functions are velocity penalties. The third term in Equation 3.9 is the direction penalty, designed to force the agent to move either counter-clockwise or clockwise depending on the initial state. Note that we use the L1 penalty for distance and L2 for velocity since L1 penalizes more for smaller deviations than L2 penalties, and we would like distance error to be minimized over velocity error.

Finally, for circle following as fast as possible, we used the reward function

$$r(s,a) = v^2 - C \cdot \mathbb{1}[\Delta x \geq \epsilon] - \lambda_2 \max\left(0, |\theta| - \frac{\pi}{2}\right)^2 \tag{3.10}$$

for training an agent using TRPO, where C is a constant used to penalize the agent for drifting too far from the circle. Since CPO has the ability to set a constraint, the following reward function was used for CPO training instead:

$$r(s,a) = v^2 - \lambda_2 \max\left(0, |\theta| - \frac{\pi}{2}\right)^2 \tag{3.11}$$

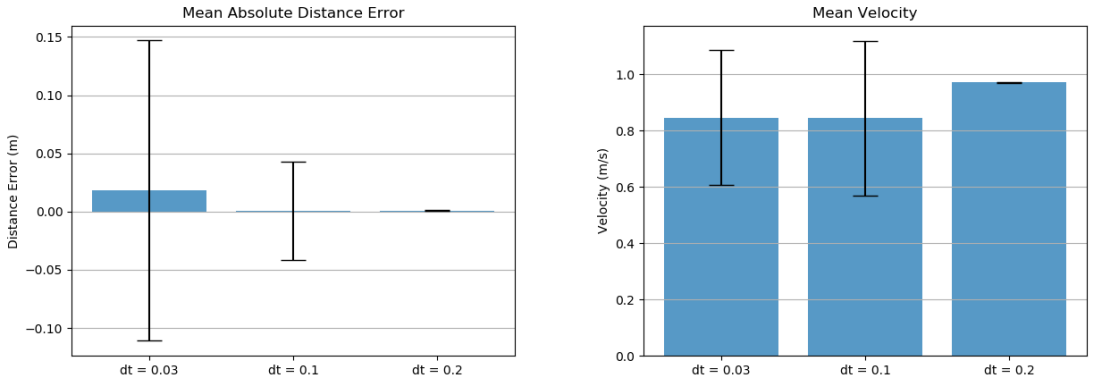
In our experiments, we had $\lambda_1 = \lambda_2 = 0.25$ for all of the reward functions.

3.5 Varying Simulation Sensor Delay for Stability

In a perfect simulation, trained policies can transfer directly onto the real robot, and these policies will be optimal. However, in our experiments, we found that policies trained in our current simulator did not directly transfer to the real world. One failure case that we observed is the following: for a robot trained to follow a straight line, the policy in simulation would output extreme changes in steering angle to correct any errors and stay close to the line. This policy performs well in simulation, but it does not transfer well to the real world; such a policy is sensitive to variations in sensor and actuator delays, control frequency, and other variables.

One solution to this problem is to perform better system identification to model the sensor and actuator delays, control frequency, and other relevant system parameters. However, these variables can be difficult to measure. As an alternative, we tried

significantly reducing the control frequency (or increasing dt) during training, i.e., we reduce the frequency at which our policy can choose a new action. In the original simulation, the policy can choose a new action every 35 ms; in the updated version, the policy can choose a new action every 100 ms. We found that such a change results in the policy learning much smoother actions, rather than choosing actions that alternate between the action limits as we previously observed. At test time, we do not alter the control frequency; we allow the robot at test time to change the actions as frequently as possible. Nonetheless, reducing the control frequency at training time results in a smoother learned policy that significantly improves the performance when transferred to the real world. Another way to think about this change is as follows: by reducing the control frequency during training, we encourage the model to learn more conservative actions, since there is less feedback in the system. On a high level, we are training more of an open-loop controller to find stable actions.



(a) Average Distance Error

(b) Average Velocity

Figure 3.4: Average distance error and average velocity for varying control frequencies for driving in a straight line at a target velocity of 1.0 m/s. Four policies using random seeds were trained for each of the dt values 0.03, 0.1 and 0.2. Then, the best-performing policies for each dt in terms of mean distance error were chosen. 50 different trajectories were collected from each best-performing policy and averaged to compute performance metrics above. We see from these results that using a longer sensor delay results in a marginally more optimal policy with respect to mean distance error (in meters) and mean velocity (in meters per second).

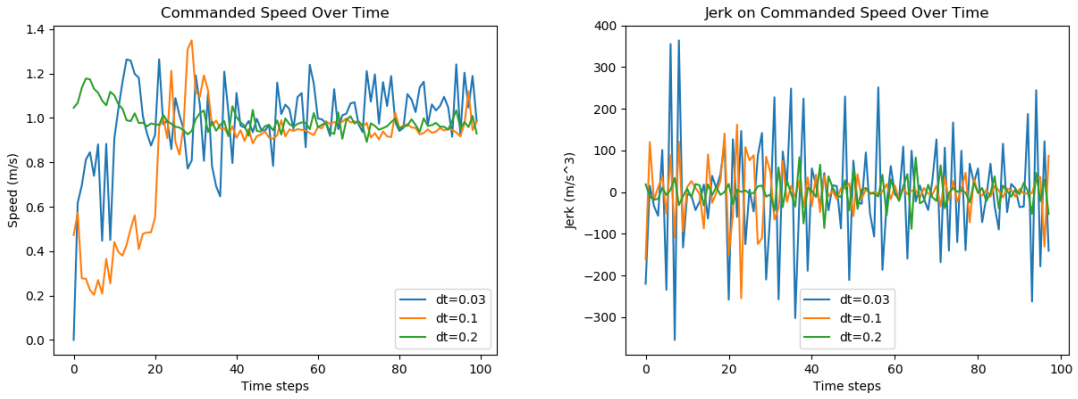
To test how policies trained with higher values of dt perform empirically, we ran

an experiment in simulation where the objective was to follow a straight line at a target velocity of 1.0 m/s. In Figure 3.4, we see that reducing the control frequency allows distance error to be minimized, while the average velocity hugs the target velocity more tightly.

To further see the benefit of using a higher dt in training, we examine jerk. Jerk is simply defined to be the rate of change of acceleration, i.e., the time derivative of acceleration:

$$\mathbf{j}(t) = \frac{d}{dt}\mathbf{a}(t) \tag{3.12}$$

Intuitively, if the rate of change of acceleration is high inside a vehicle, one would experience 'jerky' motion rather than smooth behavior. We therefore want jerk to be relatively low to exert less force on the vehicle and its passengers and (for our application) sensors.



(a) Commanded Speed

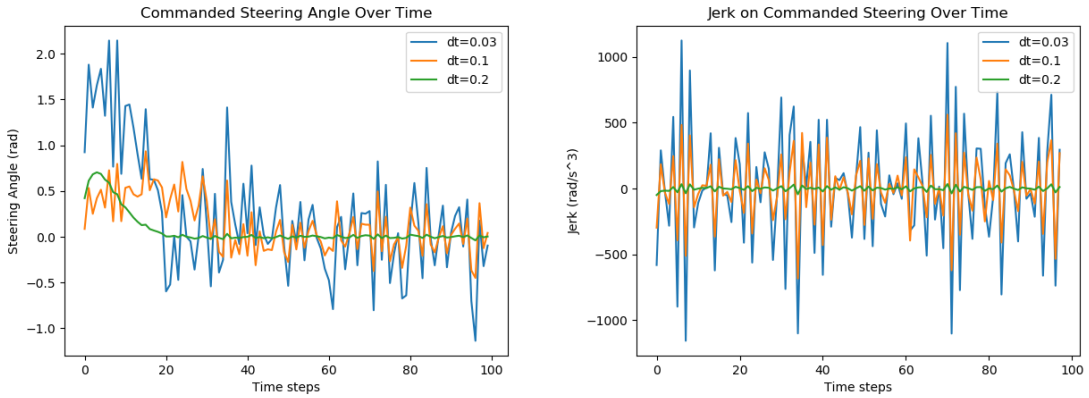
(b) Jerk on Commanded Speed

Figure 3.5: Commanded speed and the corresponding jerk induced by the commanded speed from a policy rollout. We see in **(a)** that $dt = 0.2$ has the least variance in commanded speeds. **(b)** shows this further, as training against higher dt values clearly induces policies with significantly less jerk in commanded speeds.

In Figure 3.5, **(a)** shows the commanded speed produced by the best trained policies for each value of dt for one rollout, and **(b)** shows the jerk induced by the commanded speeds by each policy. We see that the commanded speed is significantly less jerky and much smoother with policies that were trained with higher dt. Again,

this makes sense because essentially we are training more and more of an open-loop controller as dt increases, which yields stable actions.

Perhaps more importantly, we see a similar result in commanded steering angles, which is shown in Figure 3.6. It is not only undesirable but often impossible to have steering angles that are jerky since there is a physical limit on how fast the steering angle can change. Therefore, training with a higher dt helped stabilize the commanded steering angles, which helped the agent learn a more optimal policy.



(a) Commanded Steering Angle

(b) Jerk on Commanded Steering Angle

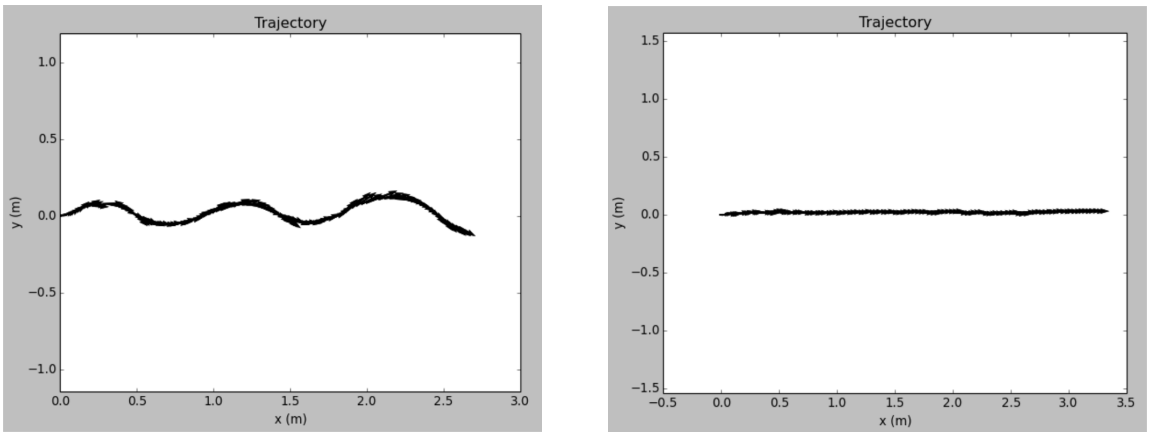
Figure 3.6: Commanded steering angles and the corresponding jerk induced by the commanded steering angles from a policy rollout. We see in **(a)** that $dt = 0.2$ has the least variance in commanded steering angles. **(b)** shows this further, as training against higher dt values clearly induces policies with significantly less jerk in commanded steering angles.

On the other hand, there are various alternate ways to mitigate drastic action changes, such as manually constraining the action changes, using action regularization via reward function engineering, inputting actions into the policy network to learn some sort of continuity, etc. However, we found that manipulating dt has been the simplest way to avoid local optima.

3.5.1 Real World Results

To test if this technique helps with sim-to-real, trained policies with varying control frequencies were directly transferred to the real world. Specifically, five seeds for

$dt = 0.035$ and $dt = 0.2$ were trained, and the best performing policy was chosen based on average distance error.



(a) $dt = 0.035$ (b) $dt = 0.2$

Figure 3.7: Trajectories collected from real world using policies trained with varying control frequencies. We can see that a longer dt, or a lower control frequency, performs significantly better than training with a higher control frequency.

From these experiments, we observed that policies trained with lower control frequencies are much more stable and exhibit less jerk. See Figure 3.7 for a visualizations of the real world trajectories. We found that, due to the gap between simulation and the real world, oscillations in the resulting trajectories made by policies were amplified in the real world. Hence, even though the policy trained in **(a)** is trying to follow a straight line, it follows more of a sinusoidal trajectory.

3.6 Learning Policies with CPO

While policies with TRPO can be trained to be robust, using CPO is more powerful since we can obtain probabilistic guarantees on the number of violations to constraints that we specify. In the sections below, we use TRPO as a baseline to compare performance and show that CPO not only trains policies to perform comparably against its TRPO counterparts, but also empirically does not violate the constraints in expectation.

3.6.1 Straight Line Following

The constraint that we want to satisfy at all times is

$$J_{C_{\text{straight}}}(s, a) = y < \epsilon \tag{3.13}$$

where $s = \left(x, y, \psi, \dot{x}, \dot{y}, \dot{\psi}\right)$ and $a = (v_x, \delta)$ are the current state and action taken by the planner. This constaint encourages the agent to never stray away further than some distance ϵ away from the target straight line.

Learning curves can be seen in Figure 3.8. Interestingly, we can see that TRPO converges at a faster rate than CPO, but CPO reaches a slightly higher average return at convergence.

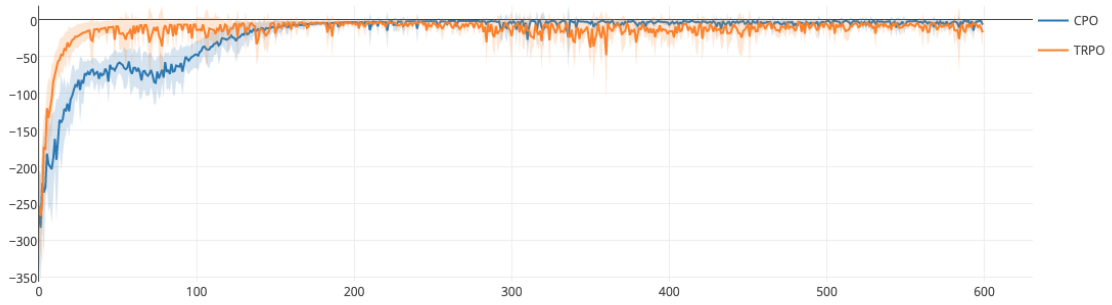


Figure 3.8: Average returns using TRPO and CPO to follow straight lines at a target velocity of 1.0 m/s. 5 random seeds were trained for each algorithm.

However, at test time TRPO- and CPO-trained policies perform similarly. Performance metrics can be seen in Figure 3.9. Both mean distance from the target trajectory (which should be close to 0) and mean velocity (which should be close to the target velocity, 1.0 m/s) measured from a trajectory rollout with a randomized initial state for TRPO- and CPO-trained policies do not show much difference in value. This is similar for constraint violations - see Figure 3.11. CPO marginally violates fewer constraints than TRPO at test time, but the difference is minimal and can be attributed to noise.

(a) Average Distance Error

(b) Average Velocity

Figure 3.9: Average distance error and average velocity for driving in a straight line at a target velocity of 1.0 m/s, driving in a circle at a target velocity of 1.0 m/s and driving in a circle as fast as possible. Four policies using random seeds were trained for each of the objective. Then, the best-performing policies for each objective in terms of mean distance error were chosen. 50 different trajectories were collected from each best-performing policy and averaged to compute performance metrics above.

3.6.2 Circle Following at Target Velocity

The constraint that we want to satisfy at all times is

$$J_{C_{\text{circle}}}(s, a) = \Delta x < \epsilon \tag{3.14}$$

where Δx is the closest distance from the circle to the agent (see Figure 3.3) and ϵ is a hyperparameter. This constraint encourages the agent to never venture more than a distance ϵ away from the circle. In our experiments, we set $\epsilon = 0.05$ meter.

Learning curves can be seen in Figure 3.10. We see that the constraint helps the agent converge to a policy marginally quicker than if an agent was trained with TRPO. This makes sense because the constraint guides the agent to learn a policy that is optimal, whereas vanilla TRPO would have to find the optimal policy through trial and error by searching through the state space.

See Figure 3.9 for performance metrics. Like straight line following at a target velocity, at test time TRPO- and CPO-trained policies performed similarly. While TRPO performed marginally better in both mean distance and mean velocity, the

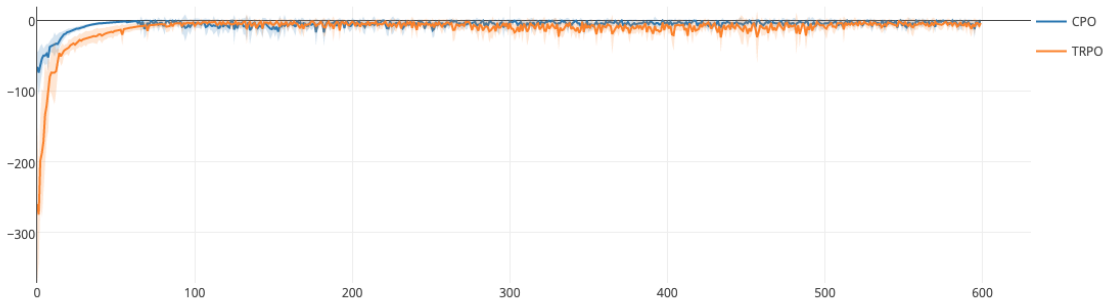


Figure 3.10: Average returns using TRPO and CPO to follow circles of radius 1 at a target velocity of 1.0 m/s. 5 random seeds were trained for each algorithm.

difference in the real world is not significant enough and can be attributed to noise. We also see in Table 3.11 that both TRPO- and CPO-trained policies commit a similar number of constraint violations. Again, results are not conclusive enough to show which of TRPO or CPO is preferred for training policies.

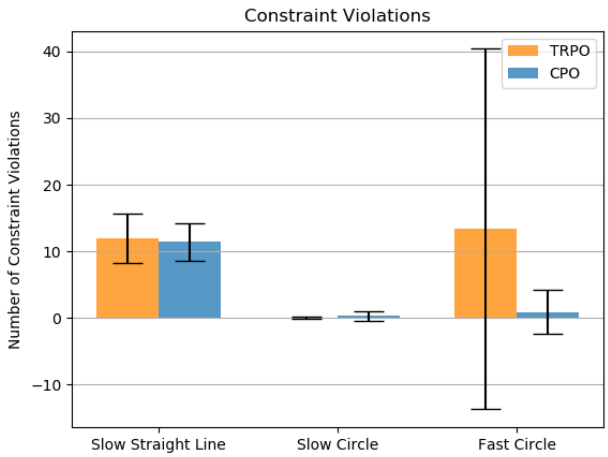


Figure 3.11: Number of constraint violations made by TRPO and CPO on different environments. The best policy with respect to average mean distance to the target trajectory from five random seeds was chosen to collect metrics displayed in this table. Metrics were averaged based on 50 rollouts. Note that metrics were collected after the first 30 time steps passed, so that the agent had sufficient time to converge to the line from different initial states.

3.6.3 Circle Following at High Velocities

The constraint for circle following at high velocities is the same constraint we used for circle following at a target velocity - see Equation 3.14. In these experiments, we again set $\epsilon = 0.05$.

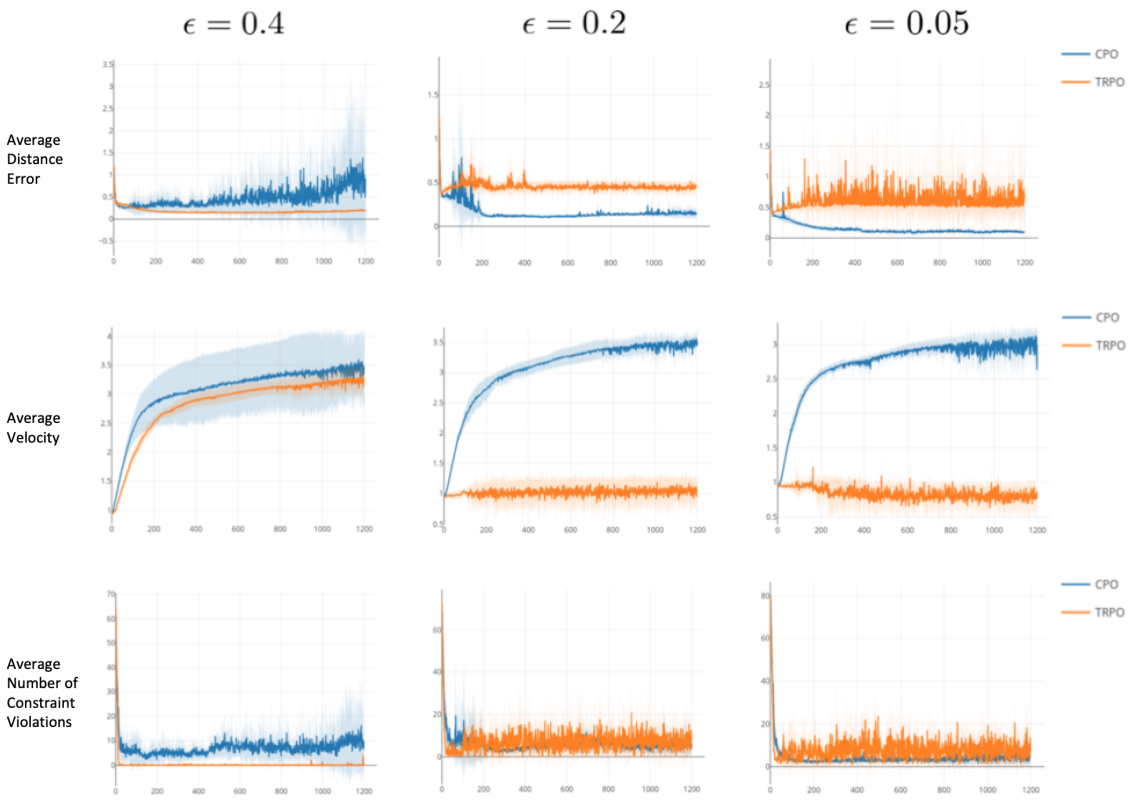


Figure 3.12: Average distance error, velocity and number of constraint violations over training using TRPO and CPO to follow circles of radius 1 as fast as possible. 5 random seeds were trained for each algorithm.

Because reward functions are defined differently for TRPO and CPO training, it does not make sense to compare learning curves for average return over training. However, in Figure 3.12, the learning curves for average mean distance error, average velocity and average number of constraint violations are plotted across training. Interestingly, we see that policies trained with CPO performs better relative to TRPO-trained policies with respect to all metrics as ϵ is decreased. Specifically, as ϵ is reduced, CPO-trained policies achieve significantly lower distance errors, while also

reaching high velocities. Furthermore, less constraint violations are made. Note that as ϵ is reduced, the constraint is harder to achieve (less distance tolerance to target trajectory).

We hypothesize that TRPO fails to find an optimal policy because the amount of high negative reward the agent obtains at initial states prevents it from obtaining any useful reward signals necessary for learning. On the other hand, training a policy with CPO is convenient because separating the constraint from the reward function allows learning to be done more efficiently by having a more dense reward signal guide the agent to learn policies that do not violate constraints.

Using the best performing policy from both TRPO and CPO with respect to distance error across 50 rollouts, performance metrics were collected; see Figure 3.9. There is a clear benefit to using CPO, as the mean distance error is significantly less and yet still achieves high velocities. We also see in Figure 3.11 the average number of constraint violations that were made with these policies over 50 rollouts. It is clear that CPO policies violate constraints considerably less than TRPO policies.

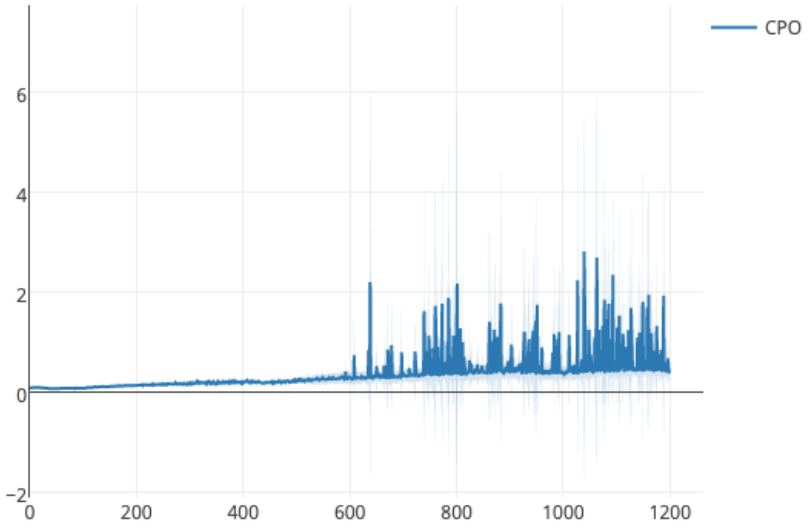


Figure 3.13: Mean wheel slip κ over training, averaged over 5 random seeds, with the objective of following a circle of radius 1.0 meter as fast as possible.

Interestingly, to achieve a high speed while attaining small distance error, the best performing CPO policy needed to learn to drift. We measure the amount of

drift by recording the value of wheel slip, or κ (see Equation 2.18). In Figure 3.13, we see that κ increases over training, which means the agent is leveraging non-trivial lateral forces to follow the circle at high speeds.

3.6.4 Real World Results

Unfortunately, due to the gap between simulation and the real world, drifting was not able to successfully be done in the real world. This is probably due to the inaccuracies of some dynamics parameters that we rely on in simulation such as moment of inertia and friction coefficients (see Chapter 4 for discussion on future work). See Figure 3.14 for a sample trajectory collected in the real world. Even though in simulation the policy performed well, in the real world the trajectory it followed deviated significantly from the target trajectory.

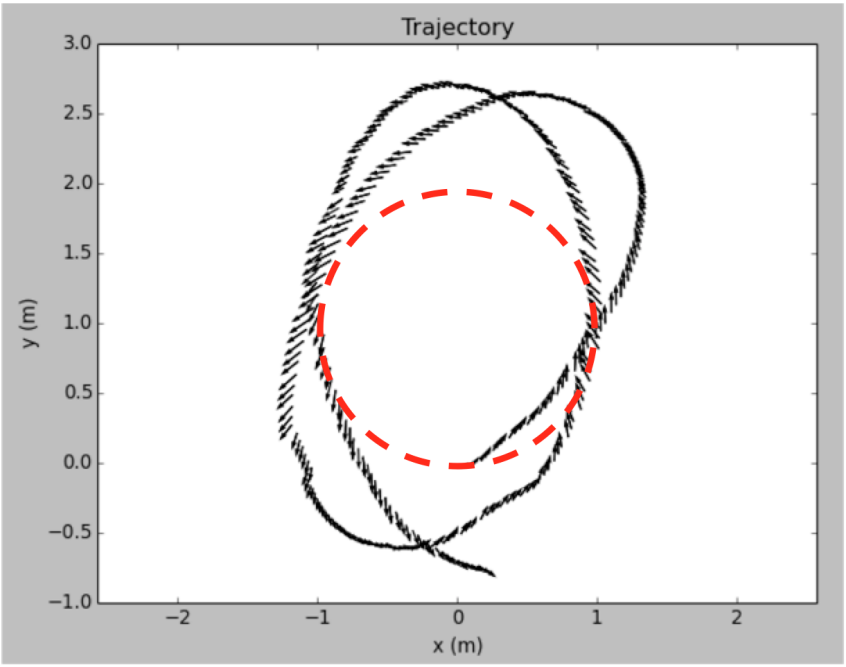


Figure 3.14: Real world trajectory. The black arrows show the position and orientation of the car over a trajectory. The red dotted circle shows the target trajectory.

Chapter 4

Conclusions

4.1 Contributions

Control for autonomous driving in slippery environments is difficult due to dynamics that are hard to model. While model-free learning removes this dependency on an accurate environment model, learning often involves black-box optimization that do not provide much interpretability, rendering policies unsafe. In this thesis, we introduced techniques we used to learn robust, probabilistically safe control policies using model-free methods for autonomous driving in slippery environments. This work is part of a larger effort to safely learn policies for autonomous drifting.

Our work involved manipulating the control frequency to produce more robust policies for sim-to-real transfer. Specifically, we observed that reducing the control frequency at which our simulation runs during training time while testing with a higher control frequency minimized jerk (first derivative to acceleration) in both simulation and the real world. This allowed policies to be successfully transferred to the real world.

In addition, to the best of our knowledge, we are the first to use Constrained Policy Optimization (CPO) to train control policies on real world tasks. We observed that the ability to create a distance constraint while rewarding high velocities allowed the agent to robustly learn drifting behaviors, which was not easily possible if the constraint was combined with the reward function with our baseline algorithm. Furthermore, CPO is a step into the direction of safe policy learning. CPO provides probabilistic

guarantees on the number of safety constraint violations that are made throughout training.

4.2 Future Work

The next step for this work is to show the viability of using CPO to learn drifting maneuvers in the real world. Drifting results in this thesis were limited to simulation results, and the gap between our simulation and the real world caused direct transfer of the policy to the real world to be unsuccessful. One approach is to conduct an investigation into model parameters such as moment of inertia and friction coefficients. An alternative is to take advantage of the model-free aspect of CPO and directly train in the real world.

Another aspect lacking in this work is comparison to other baseline methods. In the contribution minimizing jerk via reducing the control frequency in training, it would be helpful to compare this method against constraining actions during training, or penalizing large action changes during training. In the contribution towards drifting, it would be beneficial to compare performance with other methods mentioned in Chapter 2, most of which use model-based methods to learn drifting.

Finally, towards the safety end, potential next steps include integrating neural network verification into training. This could involve either verifying the policy network at each step in training, or simply verifying the trained policy network after training. In addition, integrating a runtime safety monitor during test time (or even during training at each step to guide learning) could provide stronger guarantees on safety.

Appendix A

State Space Derivations

See Figure 3.3 for a visualization of the geometric relations between the input absolute state $\left(x, y, \psi, \dot{x}, \dot{y}, \dot{\psi}\right)$ and the output relative state $\left(\Delta x, \theta, \dot{\Delta x}, \dot{\theta}\right)$ for circle following.

Given that the current state is $\left(x, y, \psi, \dot{x}, \dot{y}, \dot{\psi}\right)$. Trivially, we see that

$$\Delta x = \sqrt{x^2 + y^2} - r \tag{A.1}$$

where r is the radius of the circle the agent is following. To determine an expression for θ, we use properties of perpendicular lines:

$$\begin{aligned}
\frac{y}{x} \cdot \tan(\psi + \theta) &= -1 \\
\tan(\psi + \theta) &= -\frac{x}{y} \\
\theta + \psi &= \tan^{-1}\left(-\frac{x}{y}\right) \\
\implies \theta &= \tan^{-1}\left(-\frac{x}{y}\right) - \psi
\end{aligned}$$

Note that due to the periodicity of the tangent function, the value of θ depends on

the direction the agent is moving. This can be summarized below:

$$\theta = \begin{cases} \text{atan2}(-x, y) - \psi & \text{if moving clockwise} \\ \text{atan2}(-x, y) + \pi - \psi & \text{if moving counter-clockwise} \end{cases} \tag{A.2}$$

To determine an expression for $\dot{\Delta x}$ and $\dot{\theta}$, we find the total derivative of each with respect to the time t.

$$\begin{aligned} \dot{\Delta x} &= \frac{d}{dx}\frac{dx}{dt}(\Delta x) + \frac{d}{dy}\frac{dy}{dt}(\Delta x) \\ &= \dot{x}\frac{d}{dx}(\Delta x) + \dot{y}\frac{d}{dy}(\Delta x) \\ &= \frac{x\dot{x}}{\sqrt{x^2+y^2}} + \frac{y\dot{y}}{\sqrt{x^2+y^2}} \end{aligned} \tag{A.3}$$

$$\begin{aligned} \dot{\theta} &= \frac{d}{dx}\frac{dx}{dt}(\theta) + \frac{d}{dy}\frac{dy}{dt}(\theta) + \frac{d}{d\psi}\frac{d\psi}{dt}(\theta) \\ &= \dot{x}\frac{d}{dx}(\theta) + \dot{y}\frac{d}{dy}(\theta) + \dot{\psi}\frac{d}{d\psi}(\theta) \\ &= -\frac{y\dot{x}}{x^2+y^2} + \frac{x\dot{y}}{x^2+y^2} - \dot{\psi} \end{aligned} \tag{A.4}$$

In summary, an absolute state is converted using the equations above to convert the state into a relative state, which is fed into the neural network policy.

Appendix B

Experiment Parameters

B.1 RC Car Specifications

The components used to build the RC car can be found in Table B.1. More information on the RC car can be found at https://github.com/jsford/FFAST.

Component	Manufacturer & Model No.
Car chassis & drivetrain	MST FXX-D
Sensored BLDC motor	MST XBLS 601012
Servo motor	Savox SAVSB2274SG
Battery	Multistar 3S 11.1V 5200mAh
Voltage regulator	Icstation LM2596 XL6009
Single-board computer	NVIDIA Jetson TX2
IMU	Variense VMU931
LIDAR	Hokuyo URG-04LX-UG01
ESC	VESC 4.12

Table B.1: Components of vehicle

B.2 Initial State Distribution

For each trajectory rollout, an initial state is randomly chosen so that the agent can learn to follow the target trajectory from a wide range of state distributions. The

ranges that we use for circle following and straight line following are different. We specify the ranges that we used for training both types of policies below.

B.2.1 Straight Line Following

For training policies to be run on the RC car:

$$
\begin{aligned}
x &= 0 \\
y &\in [-0.25, 0.25] \\
\psi &\in \left[-\frac{\pi}{3}, \frac{\pi}{3}\right] \\
\dot{x} &\in [0, 1.3] \\
\dot{y} &\in [-0.6, 0.6] \\
\dot{\psi} &\in [-2.0, 2.0]
\end{aligned}
$$

For training policies to be run on the MRZR:

$$
\begin{aligned}
x &= 0 \\
y &\in [-0.25, 0.25] \\
\psi &\in \left[-\frac{\pi}{3}, \frac{\pi}{3}\right] \\
\dot{x} &\in [0, 2.0] \\
\dot{y} &\in [-0.6, 0.6] \\
\dot{\psi} &\in [-0.3, 0.3]
\end{aligned}
$$

B.2.2 Circle Following

Note that for circle following, we place the agent around the point $(-R, 0)$, with the agent tasked with following a circle centered at $(0, 0)$ with radius R and target velocity v_f.

For training policies to be run on the RC car:

$$
\begin{aligned}
x &\in [-0.25, 0.25] - R \\
y &= 0 \\
\psi &\in \left[-\frac{\pi}{3}, \frac{\pi}{3}\right] + \frac{3\pi}{2} \\
\dot{x} &\in [0, 2v_f] \\
\dot{y} &\in [-0.6, 0.6] \\
\dot{\psi} &\in [-2.0, 2.0]
\end{aligned}
$$

Since it was infeasible to run these policies on the MRZR due to space constraints, no policies were trained with MRZR parameters.

B.3 Simulation Parameters

Parameter	RC Car	MRZR
m (kg)	2.5960	879.0
l_f (m)	0.1150	1.364
l_r (m)	0.1420	1.364
F_{zf} (N)	14.0711	4307.1
F_{zr} (N)	11.3956	4307.1
C_x (N)	103.9400	13782.0
C_α (N)	56.4000	68912
I_z	0.0558	1020.0
μ	1.3700	1.3700
μ_s	1.9600	1.9600

Table B.2: Estimated values used to model RC Car and MRZR in simulation. Values for the MRZR were not estimated rigorously, and all values can significantly improved via better system identification.

B.4 Hyperparameters

The hyperparameters were used to train all policies in this document can be seen in Table B.3.

Hyperparameter	Value
Non-linearity	tanh
Number of Hidden Layers	2
Number of Hidden Units	32
Batch Size	600
Horizon Length	100
Discount Factor	0.99
Step Size	0.01
λ (GAE)	0.95
Safety λ (GAE)	1
dt (for CPO)	0.2

Table B.3: Hyperparameters

Appendix C

Code

`rllab` (https://github.com/rll/rllab) was used as our reinforcement learning framework. High-quality implementations of both TRPO and CPO were found online. `ROS` (https://www.ros.org) was used to interface with our robot.

Use the following links to view robust code implementations used for this thesis:

- Code for our custom simulation environments as well as training scripts used to train policies: https://github.com/r-pad/aa_simulation
- Code for running trained policies on our robot using ROS: https://github.com/r-pad/aa_planner

Bibliography

[1] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 22–31. JMLR. org, 2017. 2.3, 3.1, 3.2.4

[2] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*, 2018. 2.4

[3] Felix Berkenkamp, Andreas Krause, and Angela P Schoellig. Bayesian optimization with safety constraints: safe and automatic parameter tuning in robotics. *arXiv preprint arXiv:1602.04450*, 2016. 2.3

[4] Mark Cutler and Jonathan P How. Autonomous drifting using simulation-aided reinforcement learning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5442–5448. IEEE, 2016. 2.2

[5] Brian Goldfain, Paul Drews, Changxi You, Matthew Barulic, Orlin Velev, Panagiotis Tsiotras, and James M Rehg. Autorally: An open platform for aggressive autonomous driving. *IEEE Control Systems Magazine*, 39(1):26–55, 2019. 2.2

[6] David Held, Zoe McCarthy, Michael Zhang, Fred Shentu, and Pieter Abbeel. Probabilistically safe policy transfer. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5798–5805. IEEE, 2017. 2.3

[7] Ronald A Howard. Dynamic programming and markov processes. 1960. 3.1

[8] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *ICML*, volume 2, pages 267–274, 2002. 3.2.3

[9] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002. 3.2.3

[10] Steven Lovegrove Nima Keivan and Gabe Sibley. A holistic framework for planning, real-time control and model learning for high-speed ground vehicle navigation over rough 3d terrain. In *IROS*, 2012. 2.2

[11] Tak Kit Lau and Yun-hui Liu. Stunt driving via policy search. In *2012 IEEE*

International Conference on Robotics and Automation, pages 4699–4704. IEEE, 2012. 2.2

[12] Zachary C Lipton, Kamyar Azizzadenesheli, Abhishek Kumar, Lihong Li, Jianfeng Gao, and Li Deng. Combating reinforcement learning's sisyphean curse with intrinsic fear. *arXiv preprint arXiv:1611.01211*, 2016. 2.3

[13] Yunpeng Pan, Ching-An Cheng, Kamil Saigol, Keuntaek Lee, Xinyan Yan, Evangelos Theodorou, and Byron Boots. Agile off-road autonomous driving using end-to-end deep imitation learning. *arXiv preprint arXiv:1709.07174*, 2017. 2.2

[14] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018. 2.4

[15] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011. 2.1.1, 2.1.2

[16] Y Hindiyeh Rami. Dynamics and control of drifting in automobiles. *Doctor degree thesis Stanford University*, 2013. 2.1.1, 2.1.2

[17] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011. 2.2

[18] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015. 3.2.3

[19] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 3.2.2

[20] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000. 3.2.2

[21] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006. 1.2

[22] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference*

on Intelligent Robots and Systems (IROS), pages 23–30. IEEE, 2017. 2.4

[23] Chris Urmson, J Andrew Bagnell, Christopher Baker, Martial Hebert, Alonzo Kelly, Raj Rajkumar, Paul E Rybski, Sebastian Scherer, Reid Simmons, Sanjiv Singh, et al. Tartan racing: A multi-modal approach to the darpa urban challenge. 2007. 1.2

[24] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989. 3.2.2

[25] Grady Williams, Paul Drews, Brian Goldfain, James M Rehg, and Evangelos A Theodorou. Aggressive driving with model predictive path integral control. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1433–1440. IEEE, 2016. 2.2

[26] Wenhao Yu, Jie Tan, C Karen Liu, and Greg Turk. Preparing for the unknown: Learning a universal policy with online system identification. *arXiv preprint arXiv:1702.02453*, 2017. 2.4