Mobile Robot Learning

Gregory Kahn

Mobile Robot Learning

by

Gregory Kahn

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Sergey Levine, Co-chair
Pieter Abbeel, Co-chair
Ken Goldberg

Fall 2020

Mobile Robot Learning

Abstract

Mobile Robot Learning

by

Gregory Kahn

Doctor of Philosophy in Computer Science

University of California, Berkeley

Sergey Levine, Co-chair

Pieter Abbeel, Co-chair

In order to create mobile robots that can autonomously navigate real-world environments, we need generalizable perception and control systems that can reason about the outcomes of navigational decisions. Learning-based methods, in which the robot learns to navigate by observing the outcomes of navigational decisions in the real world, offer considerable promise for obtaining these intelligent navigation systems. However, there are many challenges impeding mobile robots from autonomously learning to act in the real-world, in particular (1) sample-efficiency—how to learn using a limited amount of data? (2) supervision—how to tell the robot what to do? and (3) safety—how to ensure the robot and environment are not damaged or destroyed during learning?

In this thesis, we will present deep reinforcement learning methods for addressing these real world mobile robot learning challenges. At the core of these methods is a predictive model, which takes as input the current robot sensors and predicts future navigational outcomes; this predictive model can then be used for planning and control. We will show how this framework can address the challenges of sample-efficiency, supervision, and safety to enable ground and aerial robots to navigate in complex indoor and outdoor environments.

Acknowledgments

The past five years have been some of the most gratifying and arduous times in my life, and I would not be here without the support of many people.

First and foremost, I would like to thank my advisers Sergey Levine and Pieter Abbeel for providing continual guidance and mentorship, giving me the freedom to pursue my passions, and never giving up on me, even when I pushed back. Thank you to my committee members Ken Goldberg and Claire Tomlin for their support and guidance throughout the years. And thank you to my undergraduate research mentors Sachin Patil and Ben Kehoe, who were critical in my decision to pursue a PhD.

I would like to thank all my collaborators, without whom my PhD research would not have been possible: Adam Villaflor, Anusha Nagabandi, Rowan McAllister, Katie Kang, Suneel Belkhale, Hayk Martiros, Abraham Bachrach, Rachel Li, Jeff Clune, Ron Fearing, Vitchyr Pong, Roberto Calandra, Bosen Ding, and Tianhao Zhang. I would also like to thank everyone in my labs (RAIL and RLL) and the greater BAIR community for creating a scholarly, collaborative, and fun environment in which to do research. In particular, I'd like to thank Adam Stooke, Alex Lee, Ashvin Nair, Avi Singh, Aviv Tamar, Carlos Florensa, Chelsea Finn, Dylan Hadfield-Manell, Frederik Ebert, Ignasi Clavera, Justin Fu, Kate Rakelly, Marvin Zhang, Michael Janner, Nick Rhinehart, Rowan McAllister, Sandy Huang, Somil Bansal, Tuomas Haarnoja, Vitchyr Pong, and many others. And a special shout out to Abhishek Gupta, Anusha Nagabandi, and Coline Devin: there were ups and there were downs, but we made it!

I would like to thank Ben Zipkin, Caitlin Simpson, Connor Davidge, Emi Goldstone, Michael Thornton, Nicolette Landucci, and Will Brogan for being friends that I can be myself around and can always count on, and to have a tiny bit of fun with. I would like to thank my family for their love and support. I would especially like to thank my parents Philip Kahn and Marilyn Elperin: they nourished my independence and worked tirelessly to enable me to pursue my passions, and continue to do so to this day. Lastly, thank you Olivia Solomon – to the Death Star and back.

Contents

Contents ii

1 Introduction **1**
1.1 Related Work and Contributions 2

I Sample-efficiency **5**

2 Interpolating between Model-Based and Model-Free Reinforcement Learning **6**
2.1 Related Work 7
2.2 Preliminaries 7
2.3 Generalized Computation Graphs 9
2.4 Learning Navigation Policies 11
2.5 Experiments 13
2.6 Discussion 18

3 Transfer Learning **19**
3.1 Related Work 20
3.2 Problem Formulation 21
3.3 Generalization through Simulation 21
3.4 Experiments 24
3.5 Discussion 29

II Supervision **30**

4 Self-Supervision **31**
4.1 Related Work 32
4.2 The Berkeley Autonomous Driving Ground Robot 33
4.3 Experiments 39
4.4 Discussion 44

5 Model Supervision **45**

5.1 Related Work 46
5.2 Composable Action-Conditioned Predictors 46
5.3 CAPs for Robot Navigation 49
5.4 Experiments 49
5.5 Discussion 54

6 Human Supervision 55
6.1 Related Work 57
6.2 Learning to Navigate from Disengagements 57
6.3 Experiments 62
6.4 Discussion 64

III Safety 66

7 Safe Learning using Expert Supervision 67
7.1 Preliminaries and Overview 68
7.2 Policy Learning using Adaptive Trajectory Optimization 69
7.3 Theoretical Analysis 72
7.4 Experiments 74
7.5 Discussion 79

8 Conservative Reinforcement Learning 80
8.1 Related Work 81
8.2 Preliminaries 82
8.3 Uncertainty-aware Collision Prediction 84
8.4 Experiments 88
8.5 Discussion 94

9 Conclusion 95

Bibliography 97

Chapter 1

Introduction

The goal of this thesis is to provide technical contributions which enable any person to take a robot, throw it into any environment, walk away, and have that robot autonomously learn how to perform desired tasks.

The two main components of this goal are robots and learning. While there are many motivations for developing autonomous robots, the primary perspective we take is that the purpose of autonomous robots is to perform tasks that humans do not want to perform, or that humans do want to perform but are otherwise unable to. These tasks can range from food delivery and warehouse package fulfillment to office sanitization and search-and-rescue.

But while these use cases for autonomous robots are typically noncontentious, the use of learning-based methods is currently a common source of tension and debate. One of the primary arguments made by learning-based advocates is that the robot is able to continually improve as it acts in the environment, while one of the main arguments against learning-based methods is their inability to generalize. Rather than attempt to argue for learning-based methods or for hand-engineered methods, this thesis takes a reconciliatory perspective: learning is a tool which enables engineers to design algorithms at different layers of abstraction.

But what are the current challenges preventing an autonomous learning-based robot from being deployed? This thesis examines three main challenges: (1) sample-efficiency—how to learn using a limited amount of data? (2) supervision—how to tell the robot what to do? and (3) safety—how to ensure the robot and environment are not damaged or destroyed during the learning process? By investigating these challenges, this thesis hopes to provide fundamental contributions towards the goal of enabling robots, such as the mobile robots shown in Fig. 1.1, to autonomously learn to act in real-world environments.

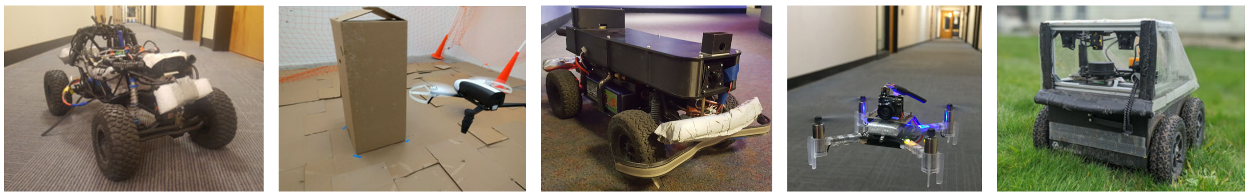


Figure 1.1: The goal of this thesis is to enable mobile robots to autonomously learn to act in real-world environments.

1.1 Related Work and Contributions

In order to create robots that can autonomously navigate complex and unstructured environments, such as roads, buildings, or forests, we need generalizable perception and control systems that can reason about the outcomes of navigational decisions. Although methods based on geometric reconstruction and mapping [1] have proven effective in a range of navigation and collision avoidance domains [2]–[14], these approaches still have limitations, such as performance degradation in textureless scenes, requiring expensive sensors, and—most importantly—do not get better as the robot acts in the world [15].

Learning offers considerable promise for mobile robotic systems: by observing the outcomes of navigational decisions in the real world, mobile robots can continuously improve their proficiency and adapt to the statistics of natural environments. Not only can learning-based systems lift some of the assumptions of geometric reconstruction methods, but they offer two major advantages that are not present in analytic approaches: (1) learning-based methods adapt to the statistics of the environments in which they are trained and (2) learning-based systems can learn from their mistakes. The first advantage means that a learning-based navigation system may be able to act more intelligently even under partial observation by exploiting its knowledge of statistical patterns. The second advantage means that, when a learning-based system does make a mistake that results in a failure, the resulting data can be used to improve the system to prevent such a failure from occurring in the future.

Sample-efficiency

One of the main challenges for deploying robot learning algorithms is sample-efficiency: how to enable a robot to quickly learn from a limited amount of data. We investigate two main approaches for addressing sample-efficiency: approaches that require little data in the target domain to effectively generalize and approaches that leverage other large and easily obtainable sources of data.

Prior work has studied how to learn generalizable control policies from limited data. These algorithms can generally be categorized as either supervised learning or reinforcement learning. In supervised learning, a human or computational expert labels the robot data, and a control policy is trained that maps the robot sensory observations to these expert labels. In reinforcment learning, the robot autonomously learns a control policy that maximizes reward via trial-and-error. While both supervised learning [16]–[21] and reinforcement learning [22] have successfully learned real-world control policies, they both have limitations regarding sample-efficiency. Supervised learning, which is typically based on human supervision, is inherently limited by the amount of human data available, which can be prohibitively expensive to collect. Reinforcement learning often requires choosing between model-free algorithms that can solve complex tasks, but are sample-inefficient [23]–[25] and model-based algorithms that are sample-efficient, but have difficulty with high-bandwidth sensors, such as cameras, and complex environments [22], [26]. In **Chapter 2**, we develop a sample-efficient reinforcement learning framework based on generalized computation graphs that subsumes value-based model-free methods and model-based methods.

Prior work has also investigated transfer learning for learning robotic control policies [27], with a particular effort on leveraging simulators due to the ease of data collection. These works have sought to transfer policies by combining simulated and real-world data [28]–[34], minimizing the gap between simulation and reality [35]–[44], and more generally investigating how to learn generalizable models when the training and test distributions differ [45]–[49]. In **Chapter 3**, we develop a generalization through simulation transfer learning algorithm that learns about the physical properties of the robot and its dynamics in the real world, while learning visual invariances and patterns from simulation.

Supervision

Another key challenge for robot learning is supervision: what signal does the robot use to learn how to successfully accomplish relevant tasks. We investigate three main methods for providing sources of supervision: self-supervision, model supervision, and human supervision.

Self-supervision, in which the robot learns directly from its own sensory observations, is perhaps the most ideal form of supervision due to its cheapness and reliability. In comparison, prior methods which learn using human labels [50]–[54] or demonstrations [55], [56] can be prohibitively expensive, while prior methods which leverage existing autonomy systems [18], [21], [57]–[60] and simulators [39], [61] can be brittle and unreliable. In **Chapter 4**, we design an end-to-end reinforcement learning approach that directly learns to predict relevant navigation cues with a sample-efficient, off-policy algorithm, and can continue to improve with additional experience via a self-supervised data labelling mechanism that does not depend on humans or SLAM algorithms.

However, certain navigation cues, such as learning to drive on the correct side of the road, are difficult to learn through pure self-supervision because there is no off-the-shelf sensor that produces the required training signal. In **Chapter 5**, we develop a flexible, multi-task reinforcement learning algorithm which learns directly from real-world events that are detected using learned models, such as existing modern computer vision systems, and is cheaper [16]–[18], [20], [62]–[64] and more flexible [56], [65]–[67] than prior approaches.

Nevertheless, leveraging learned models for supervision can be fallible because these models may be inaccurate. Perhaps the only source of truly accurate supervision is supervision directly from humans. However, the standard approaches for human supervision, such as labels [16], [50]–[52], [57], [68] or demonstrations [20], [55], [56], [62], [69]–[71], are prohibitively expensive. In **Chapter 6**, we develop a reinforcement learning algorithm that learns from disengagements, a supervision signal from human safety drivers which is already being collected in many commonly used real-world robotic testing pipelines.

Safety

The last key challenge for robot learning we investigate is safety: how to enable a robot to learn from experience without experiencing catastrophic failure. We investigate two main approaches for achieving safe robot learning: leveraging expert supervision and having the robot act conservatively in unknown situations.

In addition to providing a source of supervision for learning, having an expert collect demonstration data ensures the robot does not experience catastrophic failures. However, using these purely expert demonstration datasets for training control policies is challenging due to the distribution mismatch between the expert training data and the distribution the robot will actually visit at test time [63]. Prior work has proposed algorithms to address this data distribution mismatch [55], [63], [72], but are not safe because these algorithms must execute the learned policy during training, which can lead to catastrophic failures. In **Chapter 7**, we develop an imitation learning algorithm which adapts a computational expert to the learned policy, but does not actually execute the learned policy in the real world until training is completed; this ensures that the training and test distributions are similar, while ensuring the robot's safety during training.

Although an expert can ensure the robot does not catastrophically fail during training, requiring an expert for data collection can be prohibitively expensive. Prior work has investigated how to enable safe robot learning without assuming access to an expert, in particular by having the robot act conservatively in scenarios the robot has not seen before. However, these prior methods typically do not scale to high-bandwidth sensors [73]–[77] or only learn to detect—and not avoid—unsafe scenarios [78]. In **Chapter 8**, we develop a model-based reinforcement learning algorithm that trains an uncertainty-aware collision prediction model; this model enables the robot to plan and execute actions that avoid catastrophic failures, and act more conservatively in scenarios in which the model is uncertain.

Part I

Sample-efficiency

Chapter 2

Interpolating between Model-Based and Model-Free Reinforcement Learning

Reinforcement learning methods are typically classified as either model-free or model-based. Value-based model-free approaches learn a function that takes as input a state and action, and outputs the value (i.e., the expected sum of future rewards). Policy extraction is then performed by selecting the action that maximizes the value function. Model-based approaches learn a predictive function that takes as input a state and a sequence of actions, and output future states. Policy extraction is then performed by selecting the action sequence that maximizes the future rewards using the predicted future states. In general, model-free algorithms can learn complex tasks [24], [25], but are usually sample-inefficient, while model-based algorithms are typically sample-efficient [22], but have difficulty scaling to complex, high-dimensional tasks [26].

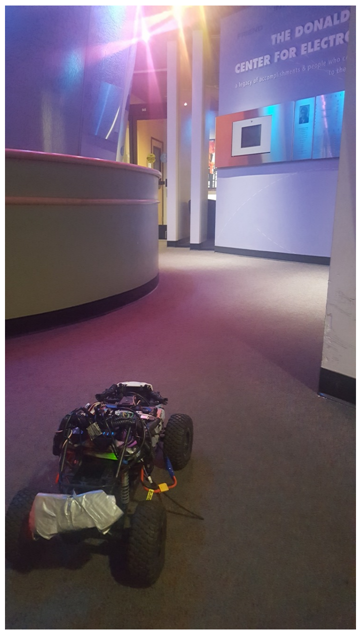


Figure 2.1: Our RC car navigating Cory Hall from raw monocular camera images using our learned navigation policy trained using 4 hours of fully autonomous reinforcement learning.

We explore the intersection between value-based model-free algorithms and model-based algorithms in the context of learning robot navigation policies. This chapter has three primary contributions. The first contribution is a generalized computation graph for reinforcement learning that subsumes value-based model-free methods and model-based methods. The second contribution is instantiations of the generalized computation graph for the task of robot navigation, resulting in a suite of hybrid model-free, model-based algorithms. The third contribution is an extensive empirical evaluation in which we: (a) investigate and discover design decisions regarding our robot navigation computation graph that are important for stable and high performing policies; (b) show our approach learns better policies than state-of-the-art reinforcement learning methods in a simulated environment; and (c) show our approach can learn a successful navigation policy on an RC car in a real-world, indoor environment from scratch using raw monocular images (Fig. 2.1).

2.1 Related Work

Our generalized computation graph allows for model instantiations that combine model-free and model-based approaches. Combining model-free and model-based methods has been investigated in a number of prior works [79]. Prior work has explored value function estimators that take in multiple actions [80], [81], in the context of simulated tasks such as playing Atari games. In contrast to this prior work, we examine a variety of multi-action prediction models, trained both with supervised learning and Q-learning style methods, and demonstrate effective learning in complex real-world environments. Our empirical results show that the design presented in prior work [80], [81] is often not the best one for real-world continuous learning tasks, and shed light on the tradeoffs involved with single- and multi- action Q-learning, as well as purely prediction-based control

2.2 Preliminaries

Our goal is to learn collision avoidance policies for mobile robots. We formalize this task as a reinforcement learning problem, where the robot is rewarded for collision-free navigation.

In reinforcement learning, the goal is to learn a policy that chooses actions $\mathbf{a}_t \in \mathcal{A}$ at each time step t in response to the current state $\mathbf{s}_t \in \mathcal{S}$, such that the total expected sum of discounted rewards is maximized over all time. At each time step, the system transitions from $\mathbf{s}_t$ to $\mathbf{s}_{t+1}$ in response to the chosen action $\mathbf{a}_t$ and the transition probability $T(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, collecting a reward r_t according to the reward function $R(\mathbf{s}_t, \mathbf{a}_t)$. The expected sum of discounted rewards is then defined as $\mathrm{E}_{\pi,T}[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} | \mathbf{s}_t, \mathbf{a}_t]$, where $\gamma \in [0, 1]$ is a discount factor that prioritizes near-term rewards over distant rewards, and the expectation is taken under the transition function T and a policy π. Algorithms that solve the reinforcement learning problem are typically either model-free or model-based. The generalized computation graph we introduce subsumes value-based model-free methods and model-based, therefore we will first provide a brief overview of both these methods.

Value-based model-free reinforcement learning

Value-based model-free algorithms learn a value function in order to select which actions to take. In this work, we will focus specifically on algorithms that learn state-action value functions, also called Q-functions. The standard parametric Q-function, $Q_\theta(\mathbf{s}, \mathbf{a})$, is a function of the current state and a single action, and outputs the expected discounted sum of future rewards that will be received by the optimal policy after taking action $\mathbf{a}$ in state $\mathbf{s}$, where θ denotes the function parameters. If the Q-function can be learned, then the optimal policy can be recovered simply by taking that action $\mathbf{a}$ that maximizes $V_t = Q_\theta(\mathbf{s}_t, \mathbf{a})$. A standard method for learning the Q-function is to minimize the Bellman error, given by

$$\mathcal{E}_t(\theta) = \frac{1}{2}\mathrm{E}_{\mathbf{s}\sim T,\mathbf{a}\sim\pi}\left[\|r_t + \gamma V_{t+1} - Q_\theta(\mathbf{s}_t, \mathbf{a}_t)\|^2\right],$$

where the actions are sampled from $\pi(\cdot|\mathbf{s})$ and the V_{t+1} term is known as the bootstrap. The policy π can in principle be any policy, making Q-learning an off-policy algorithm.

Multi-step returns [82] can be incorporated into Q-learning and other TD-style algorithms by summing over the rewards over N steps, and then using the current or target Q-function to label the N+1th step. Multi-step returns can increase sample efficiency, but also make the algorithm on-policy. Defining the N-step value as $V_t^{(N)} = \sum_{n=0}^{N-1} \gamma^n r_{t+n} + \gamma^N V_{t+N}$, we can augment the standard Bellman error minimization objective by considering a weighted combination of Bellman errors from horizon length 1 to N:

$$\mathcal{E}_t(\theta) = \frac{1}{2} \mathrm{E}_{\mathbf{s}\sim T, \mathbf{a}\sim\pi} \left[\| \sum_{N'=1}^{N} w_{N'} V_t^{(N')} - Q_\theta(\mathbf{s}_t, \mathbf{a}_t) \|^2 \right] \quad : \quad \sum_{N'=1}^{N} w_{N'} = 1, w_{N'} \geq 0.$$

Model-based reinforcement learning

In contrast to model-free approaches, which avoid modelling the transition dynamics by learning a Q-value and using bootstrapping, model-based approaches explicitly learn a transition dynamics $\hat{T}_\theta(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ parameterized by vector θ. At time t in state $\mathbf{s}_t$, the next action $\mathbf{a}_t$ is selected by solving the finite-horizon control problem

$$\arg\max_{\mathbf{A}_t^H} \mathrm{E} \left[\sum_{h=0}^{H-1} \gamma^h R(\hat{\mathbf{s}}_{t+h}, \mathbf{a}_{t+h}) \right] \quad : \quad \hat{\mathbf{s}}_{t'+1} \sim \hat{T}_\theta(\hat{\mathbf{s}}_{t'+1}|\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}), \hat{\mathbf{s}}_t = \mathbf{s}_t,$$

in which H is the planning horizon and $\mathbf{A}_t^H = (\mathbf{a}_t, \mathbf{a}_{t+1}, ..., \mathbf{a}_{t+H-1})$ is the planned action sequence. Note that the reward function R must be known a priori.

Since planning for large H is expensive and often undesirable due to model inaccuracies, planning is typically done in a model predictive control (MPC) fashion in which the optimization is solved at each time step, the first action from the optimized action sequence is executed, and the process is repeated. Standard model-based algorithms then alternate between gathering samples using MPC and storing transitions $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$ into a dataset $\mathcal{D}$, and updating the transition model parameter vector θ to maximize the likelihood of the transitions stored in $\mathcal{D}$.

Comparing model-free and model-based methods

We now compare the advantages and disadvantages of both model-free and model-based methods for learning continuous robot navigation policies by evaluating three metrics: sample efficiency, stability, and final performance.

Model-free methods have empirically demonstrated state-of-the-art performance in many complex tasks [24], [25], including navigation [83]. However, model-free techniques are often sample inefficient. Specifically, for (N-step) Q-learning, bias from bootstrapping and high variance multi-step returns can lead to slow convergence [84]. Furthermore, Q-learning often requires experience replay buffers and target networks for stable learning, which also further decreases sample efficiency [24]. We empirically demonstrate that these stability issues are further exacerbated in continuous learning scenarios, such as robot navigation.

In contrast, model-based methods can be very sample efficient and stable, since learning the transition model reduces to supervised learning of dense time-series data [22]. However, final performance can be poor because maximizing the accuracy of the learned transition model is merely a surrogate objective, that is to say that an accurate transition model does not necessarily mean the policy will perform well. In addition, all three metrics suffer when the state space is high-dimensional, such as when learning from raw images [85].

In order to develop a sample efficient, stable, and high performing reinforcement learning algorithm for training robot navigation policies, we will leverage aspects of both model-free and model-based methods. Combining model-free and model-based methods has been investigated in a number of prior works [79]. Prior work has explored value function estimators that take in multiple actions [80], [81], in the context of simulated tasks such as playing Atari games. In contrast to this prior work, we examine a variety of multi-action prediction models, trained both with supervised learning and Q-learning style methods, and demonstrate effective learning in complex real-world environments. Our empirical results show that the design presented in prior work [80], [81] is often not the best one for real-world continuous learning tasks, and shed light on the tradeoffs involved with single- and multi- action Q-learning, as well as purely prediction-based control.

2.3 Generalized Computation Graphs

We will now introduce a generalized computation graph for reinforcement learning that subsumes both model-free value function-based methods and model-based algorithms. This generalized computation graph not only encompasses existing model-free and model-based methods, but also will allow us to formulate a sample-efficient, stable, and high-performing algorithm for training robot navigation policies.

Fig. 2.2 shows a generalized computation graph for reinforcement learning models. The computation graph $G_\theta(\mathbf{s}_t, \mathbf{A}_t^H)$ parameterized by vector θ takes as input the current state $\mathbf{s}_t$ and a sequence of H actions $\mathbf{A}_t^H = (\mathbf{a}_t, ..., \mathbf{a}_{t+H-1})$ and produces H sequential predicted outputs $\hat{Y}_t^H = (\hat{y}_t, ..., \hat{y}_{t+H-1})$ and a predicted terminal output $\hat{b}_{t+H}$. These predicted outputs $\hat{Y}_t^H$ and $\hat{b}_{t+H}$ are combined and compared with labels Y_t^N and b_{t+N} to form an error signal $\mathcal{E}_t(\theta)$ that is minimized using an optimizer.

We will now show how the generalized computation graph can be instantiated to be standard model-free value function-based methods and model-based methods. We first instantiate the computation graph for N-step Q-learning by letting y be reward and b be the future value estimate; setting the model horizon $H = 1$ and using N-step returns; and letting the error function be the Bellman error: $\mathcal{E}_t(\theta) = \|(\hat{y}_t + \gamma\hat{b}_{t+1}) - (\sum_{n=0}^{N-1} \gamma^n y_{t+n} + \gamma^N b_{t+N})\|_2^2$. Next, we instantiate the computation graph for standard model-based learning by ignoring y and letting b be the state; setting the model horizon $H = 1$ and label horizon $H = 1$; and letting the error function minimize the difference between the predicted and actual next state: $\mathcal{E}_t(\theta) = \|\hat{b}_{t+1} - b_{t+1}\|_2^2$.

In order to use the generalized computation graph in a reinforcement learning algorithm, we must be able to extract a policy from the generalized computation graph. We define $J(\mathbf{s}_t, \mathbf{A}_t^H)$ to be the generalized policy evaluation function, which is a scalar function such that $\pi(\mathbf{A}^H|\mathbf{s}_t) =$

Algorithm 1 Reinforcement learning with generalized computation graphs

```
1: input: computation graph $G_\theta(\mathbf{s}_t, \mathbf{A}_t^H)$, error function $\mathcal{E}_t(\theta)$, and policy evaluation function $J(\mathbf{s}_t, \mathbf{A}_t^H)$
2: initialize dataset $\mathcal{D} \leftarrow \emptyset$
3: for $t = 1$ to $T$ do
4:     get current state $\mathbf{s}_t$
5:     $\mathbf{A}_t^H \leftarrow \arg\max_{\mathbf{A}} J(\mathbf{s}_t, \mathbf{A})$
6:     execute first action $\mathbf{a}_t$
7:     receive labels $y_t$ and $b_t$
8:     add $(\mathbf{s}_t, \mathbf{a}_t, y_t, b_t)$ to dataset $\mathcal{D}$
9:     update $G_\theta$ by $\theta \leftarrow \arg\min_\theta \mathcal{E}_{t'}(\theta)$ using $\mathcal{D}$
10: end for
```

$\arg\max_{\mathbf{A}^H} J(\mathbf{s}_t, \mathbf{A}^H)$. Similar to before, we now instantiate J for standard model-free value function-based methods and model-based methods. For N-step Q-learning, $J(\mathbf{s}_t, \mathbf{a}_t) = \hat{y}_t + \gamma \hat{b}_{t+1}$ is the estimated future value. For standard model-based learning, $J(\mathbf{s}_t, \mathbf{a}_t) = R(\mathbf{s}_t, \mathbf{a}_t) + J(\hat{b}_{t+1}, \arg\max_{\mathbf{a}} J(\hat{b}_{t+1}, \mathbf{a}))$ is the reward function evaluated on the single-step dynamics model propagated from the current state for multiple timesteps.

Using the generalized computation graph G_θ, the graph error function $\mathcal{E}$, and the policy evaluation function J, we outline a general reinforcement learning algorithm in Alg. 1. This framework for general-purpose predictive learning can subsume both standard model-free value function-based methods and model-based algorithms.

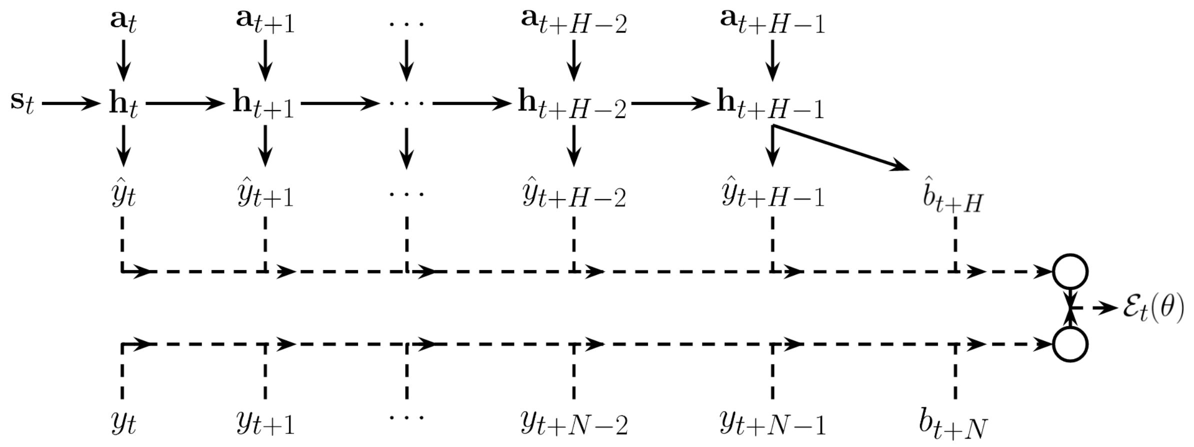


Figure 2.2: A generalized computation graph for model-free, model-based, and hybrid reinforcement learning algorithms. These algorithms train models that take as input the current state and a sequence of H actions and produce H outputs plus a final terminal output. These predicted outputs are then combined and compared with N labels to produce an error signal that is used by an optimizer in order to train the model. Solid lines indicate computations involving model parameters θ while dashed lines indicate signal flow.

2.4 Learning Navigation Policies

The computation graph outlined in the previous section can be instantiated to perform both fully model-free learning, where the model predicts the expected sum of future rewards, and fully model-based learning, where the model predicts future states. However, in practical robotic applications, especially in robotic mobility, we often have prior knowledge about our system. For example, the dynamics of a car could be identified in advance with considerable accuracy. Other aspects, such as the relationship between observed images and positions of obstacles, are exceptionally difficult to specify analytically, and could be learned by a model. The question then arises: which aspects of the system should we learn to predict, which aspects should we handle with analytic models, and which aspects should we ignore?

We will now explore the space of possible instantiations of the generalized computation graph for learning robot navigation policies. While some design decisions will remain constant, other design choices will have multiple options, which we will describe in detail and empirically evaluate in our experiments.

Model parameterization. While many function approximators could be used to instantiate our generalized computation graph, the function approximator needs to be able to cope with high-dimensional state inputs, such as images, and accurately model sequential data due to the nature of robot navigation. We therefore parameterize the computation graph as a deep recurrent neural network (RNN), depicted in Fig. 2.3.

Model outputs. While we have defined what our deep recurrent neural network model takes as input, namely the current state and a sequence of actions, we need to specify what quantities are the model outputs $\hat{y}$ and $\hat{b}$. We consider two quantities. The first quantity is the standard approach in the reinforcement literature: $\hat{Y}_t^H$ represent rewards and $\hat{b}_{t+H}$ represents the future value-to-go. For the task of collision-free navigation, we define the reward as the robot's speed, which is typically known using onboard sensors, and therefore the value is approximately the distance the robot travels before experiencing a collision. The advantage of outputting the value-to-go is that this is precisely what our agents want to maximize. However, the value representation does not leverage any prior knowledge about robot navigation.

The second quantity we propose is specific to collision avoidance, in which $\hat{Y}_t^H$ represents the

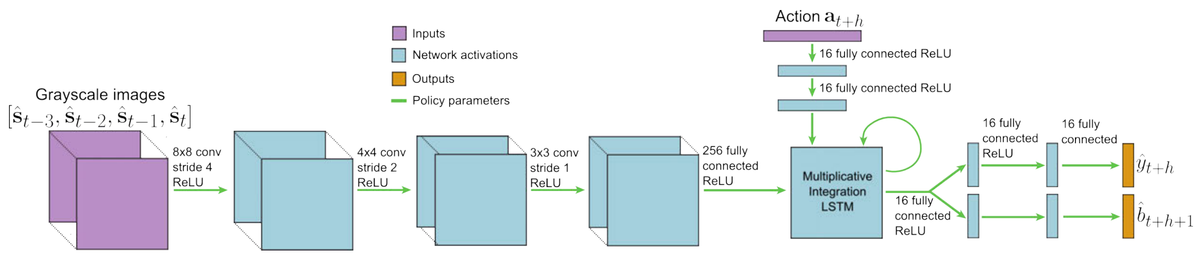


Figure 2.3: Recurrent neural network computation graph for robot navigation policies. The network takes as input the past four grayscale images, which are processed by convolutional layers to become the initial hidden state for the recurrent unit. The recurrent unit is a multiplicative integration LSTM [86]. From $h = 0$ to $H - 1$, the recurrent unit takes as input the processed action and the previous hidden state, and produces the next hidden state and the RNN output vector. The RNN output vector is passed through final layers to produce the model outputs $\hat{y}_{t+h}$ and $\hat{b}_{t+h+1}$.

probability of collision at or before each timestep—that is, $\hat{y}_{t+h}$ is the probability the robot will collide between time t and $t+h$—and $\hat{b}_{t+H}$ represents the best-case future likelihood of collision. One advantage of outputting collision probabilities is that this binary signal may be easier and faster to learn.

Policy evaluation function. Given the outputs of the navigation computation graph, we now need to define how the task of collision-free robot navigation is encoded into the policy evaluation function J.

If the model output quantities are values, which in our case is the expected distance-to-travel, then the policy evaluation function is simply the value $J(\mathbf{s}_t, \mathbf{A}_t^H) = \sum_{h=0}^{H-1} \gamma^h \hat{y}_{t+h} + \gamma^H \hat{b}_{t+H}$.

If the model output quantities are collision probabilities, then the policy evaluation function needs to somehow encourage the robot to move through the environment. We assume that the robot will be travelling at some fixed speed, and therefore the policy evaluation function needs to evaluate which actions are least likely to result in collisions $J(\mathbf{s}_t, \mathbf{A}_t^H) = \sum_{h=0}^{H-1} -\hat{y}_{t+h} - \hat{b}_{t+H}$.

Policy evaluation. Using the policy evaluation function, action selection is performed by solving the finite-horizon planning problem $\arg\max_{\mathbf{A}^H} J(\mathbf{s}_t, \mathbf{A}^H)$. Although we can use any optimal control or planning algorithm to perform the maximization, in our experiments we use a simple random shooting method, in which the K randomly sampled action sequences are evaluated with J and the action sequence with the largest value is chosen. We also evaluated action selection using the cross entropy method [87], but empirically found no difference in performance. However, exploring other methods could further improve performance.

Model horizon. An important design decision is the model horizon H. The value of the model horizon in effect determines the degree to which the model is model-free or model-based. For $H = 1$, the model is fully model-free because it does not model the dynamics of the output, while for horizon H that is the full length of a (possibly infinite) episode, the model is fully model-based in the sense that the model has learned the dynamics of the output. For intermediate values of H, the model is a hybrid of model-free and model-based methods. We empirically evaluate different horizon values in our experiments.

Label horizon. In addition to the model horizon, we must decide the label horizon N. The label horizon N can either be set to the model horizon H, or to some value $N > H$. Although setting the label horizon N to be larger than the model horizon H can increase learning speed, as N-step Q-learning often demonstrates, the learning algorithm then becomes on-policy. This is an undesirable property for robot navigation because we would like our policy to be able to be trained with any kind of data, including data gathered by old policies or by exploration policies. We therefore set the label horizon N to be the same as the model horizon H.

Bootstrapping. Because we chose the label horizon to be the same as the model horizon, the only way in which the model can learn about future outcomes is by increasing the model horizon or by using bootstrapping. An advantage of increasing the model horizon is that the model becomes more model-based, which has been shown to be sample efficient. However, increasing the model horizon increases the difficulty of policy evaluation because the search space grows exponentially in H. Bootstrapping can alleviate the planning problem by allowing for smaller H, but bootstrapping can cause bias and instability in the learning process. We evaluate the effect of bootstrapping in our experiments.

Training the model. Finally, to train our model using a dataset $\mathcal{D}$, we need to define the loss function between the model outputs and the labels. Using samples $(\mathbf{s}_t^H, \mathbf{A}_t^H, y_t^H) \in \mathcal{D}$ from the dataset, if the model outputs and labels are values, the loss function is the standard Bellman error $\mathcal{E}_t(\theta) = \| \sum_{n=0}^{N-1} \gamma^n y_{t+n} + \gamma^N b_{t+H} - J(\mathbf{s}_t, \mathbf{A}_t^H)\|_2^2$, in which $b_{t+H} = \max_{\mathbf{A}^H} J(\mathbf{s}_{t+H}, \mathbf{A}^H)$. If the model outputs are collision probabilities, the loss function is the cross entropy loss

$$\mathcal{E}_t(\theta) = -\Big[\sum_{h=0}^{H-1} y_{t+h} \log(\hat{y}_{t+h}) + (1 - y_{t+h}) \log(1 - \hat{y}_{t+h}) + b_{t+H} \log(\hat{b}_{t+H}) + (1 - b_{t+H}) \log(1 - \hat{b}_{t+H})\Big],$$

in which $b_{t+H} = \min_{\mathbf{A}^H} \frac{1}{H} \sum_{h=0}^{H} \hat{y}_{t+H+h}$ represents the lowest average probability of collision the robot can achieve at time $t + H$. We note that these probabilities can also be learned using a mean squared error loss, and we examine the effect of the loss function choice in our experiments.

2.5 Experiments

The navigation computation graphs discussed in the previous section can be instantiated in various ways, such as predicting full reward or only collision, and predicting with different horizons. In our experiments, we aim to evaluate the various design choices exposed by the generalized computation graph framework, including the special cases that correspond to standard algorithms such as Q-learning, and study their impact on both simulated and real-world robotic navigation. In our evaluations, we aim to answer the following questions:

Q1 How do the different design choices for our navigation computation graph affect performance?

Q2 Given the best design choices, how does our approach compare to prior methods?

Q3 Is our approach able to successfully learn a navigation policy on a real robot in a complex environment?

Experiment videos and code are provided on our website `github.com/gkahn13/gcg`.

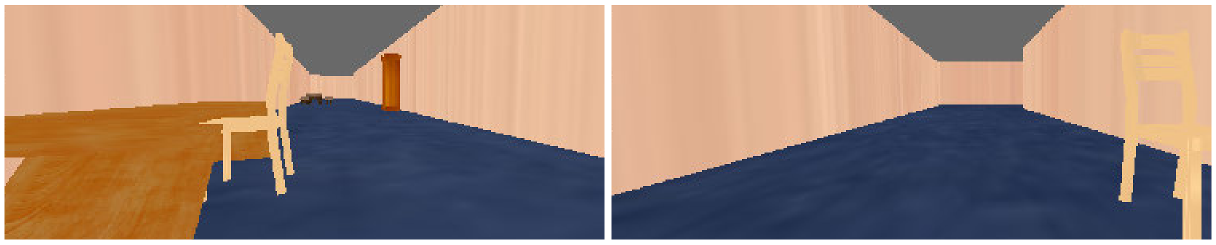


Figure 2.4: First-person view images from a simulated RC car learning to navigate in a cluttered hallway.

Simulation results

We first present results on a simulated RC car in a cluttered, indoor environment (Fig. 2.4). The RC car was created using the Bullet physics simulator and images were rendered using the Panda3d graphics engine [88]. The robot state $\mathcal{S} \in \mathbb{R}^{2304}$ is a 64×36 grayscale image taken from an onboard forward-facing camera. The car navigates at a fixed speed of 2m/s, therefore the action space $\mathcal{A} \in \mathbb{R}^1$ consists solely of the steering angle. The car observes the current image and selects an action every $\text{dt} = 0.25$ seconds. We note that 1 hour of simulator time will result in $14,400$ datapoints. We define an episode as the car acting in the environment until it either crashes or travels 1000 meters. Because we are considering the setting of continuous learning, each episode continues from where the previous episode ended; if the previous episode ended in a collision, the car first executes a hard-coded backup maneuver before starting the next episode. All experiments were evaluated three times with different random seeds.

Evaluating design decisions for robot navigation learning. We will now explore and empirically evaluate the four design decisions (Sec. 2.4) of our navigation computation graph: the model output, loss function, model horizon, and bootstrapping. These decisions will be evaluated in terms of their effect on sample efficiency, stability, and final performance.

Model outputs and loss function

Fig. 2.5 shows learning curves for different model outputs and loss functions. "Value" corresponds to outputs that represent the expected sum of future rewards, while "collision" corresponds to outputs that represent probabilities of collision. Regression corresponds to using a mean squared error loss function, while classification corresponds to using a cross entropy loss function. For consistency, these models all use a long horizon ($H = 16$) and do not use bootstrapping.

These results show that, given the same mean squared error loss function, outputting collision probabilities leads to substantially more sample-efficient and higher-performing policies than outputting predicted values. Although one might be tempted to say that the improved performance is due to different policy evaluation functions J, we note that, because the car is always moving at a fixed speed of 2m/s and the reward function is the car's speed, the only rewards the value model trains on are either 2 (if no collision) or 0 (if collision), which is a binary signal. The major difference lies in the form of the mean squared error loss function: the value model loss is a single loss on the sum of the outputs, while the collision model is the sum of H separate losses on each of the outputs. The

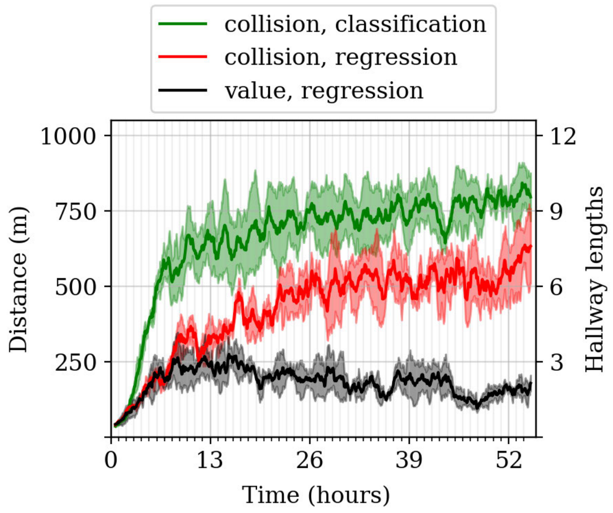


Figure 2.5: Evaluation of our approach with different model outputs (values or collision probabilities) and training methods (regression or classification).

collision model therefore has additional supervision about when the collision labels occur in time. Additionally, training with a cross entropy loss is significantly better than training with a mean squared error loss in terms of both sample efficiency and final performance.

This comparison shows that predicting discrete future events can lead to faster and more stable learning than predicting continuous sums of discounted rewards. While we have only shown this finding in the context of robot navigation, this insight could lead to a new class of sample-efficient, stable, and high-performing reinforcement learning algorithms [89].

Model horizon

We next examine the effect of using short model horizons ($H = 1$, corresponding to a $\frac{1}{2}$m lookahead) versus long model horizons ($H = 16$, corresponding to an 8m lookahead). For consistency, and so that the model with the short horizon can learn about events beyond its planning horizon, all models use bootstrapping. Fig. 2.6 shows these results for models that output values and models that output collision probabilities. The models use regression for outputs that are values and classification for outputs that are collision probabilities.

For models that output values, training with a longer horizon is more stable and leads to a higher performing final policy. While the short horizon model initially has the same learning speed, its performance peaks early on and declines thereafter. While one might be inclined to attribute this decrease in performance to overfitting, we note that the long horizon model should be even more prone to overfitting, yet it performs much better. We therefore conclude the longer horizon model learns better because the long horizon mitigates the bias of the bootstrap due to the exponential weighting factor γ^H in front of the bootstrap term.

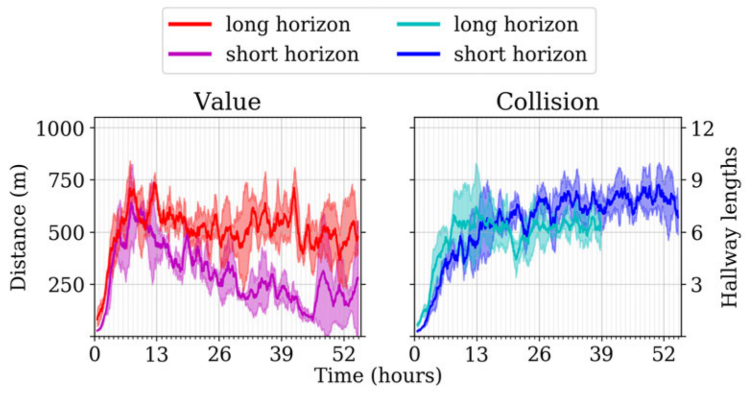


Figure 2.6: Evaluation of our approach with different model horizons. (Collision with long horizon was terminated early due to computational constraints.)

However, for models that output collision probabilities, we do not notice any change in performance when comparing short and long horizon models. This could be due to the fact that the probabilities are necessarily bounded between 0 and 1, which minimizes the bias from bootstrapping. Future work investigating the relationship between classification and bootstrapping could yield more stable and sample-efficient reinforcement learning algorithms.

Bootstrapping

Finally, we investigate the effect of bootstrapping. Fig. 2.7 shows these results for models that output values and models that output collision probabilities. The models all use long horizon prediction ($H = 16$) because short horizon models (e.g., $H = 1$) fail to learn anything when not using bootstrapping. For consistency, these models all use regression for outputs that are values and classification for outputs that are collision probabilities.

When not using bootstrapping, models that output values fail to learn, while models that output collision probabilities are extremely sample efficient, stable, and result in high-performing final policies. This dichotomy indicates that learning the dynamics of event probabilities, such as collisions, is easier than learning general, unbounded values. Future work investigating model-based reinforcement learning of domain-specific, discrete events could lead to a new class of sample-efficient model-based algorithms.

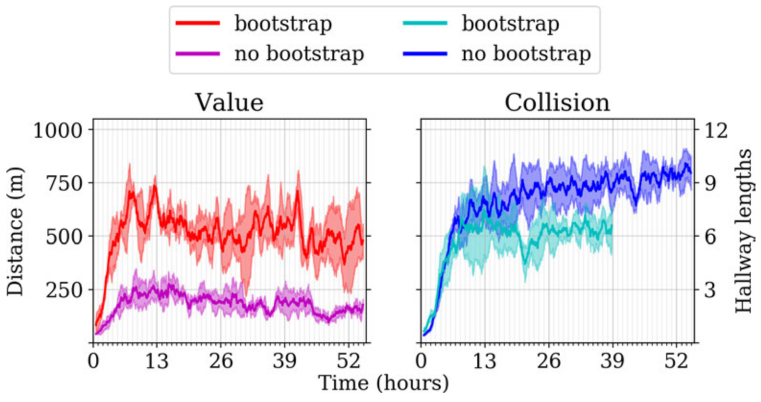


Figure 2.7: Evaluation of our approach with and without bootstrapping.

When using bootstrapping, models that output values perform worse than models that output collision probabilities. However, models that output values do benefit from using bootstrapping. In contrast, collision prediction models are not strongly affected by using or not using bootstrapping. These results indicate that if the task can be accomplished by looking H steps ahead, then not using bootstrapping can be advantageous.

Comparisons with prior work. Given the empirical evaluations of our design decisions, we choose the instantiation of our generalized computation graph for robot navigation to output collision probabilities, train with a classification loss, use a long model horizon ($H = 16$), and not use bootstrapping. To ensure a fair comparison with the prior methods (double Q-learning and N-step double Q-learning), we found the best settings for double Q-learning by performing a hyperparameter sweep over relevant parameters, such as exploration rates, learning rates, and target network update rates, and evaluating each set of hyperparameters on a simpler navigation task in an empty hallway. We used these best-performing hyperparameters for all methods in the cluttered hallway environment.

Fig. 2.8 shows results comparing our approach with double Q-learning and N-step double Q-learning; note that we do not compare with model-based approaches because they either assume knowledge of the ground truth state, or the model would have to learn to predict future images, which is sample-inefficient. Our approach is more stable and learns a final policy that is 50% better than the closest prior method. We believe this highlights that by viewing the problem through the generalized computation graph (Sec. 2.3) and incorporating domain knowledge

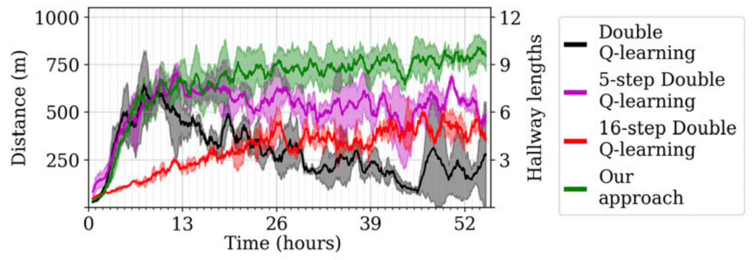


Figure 2.8: Comparison of our robot navigation learning approach to prior methods in a simulated cluttered hallway environment.

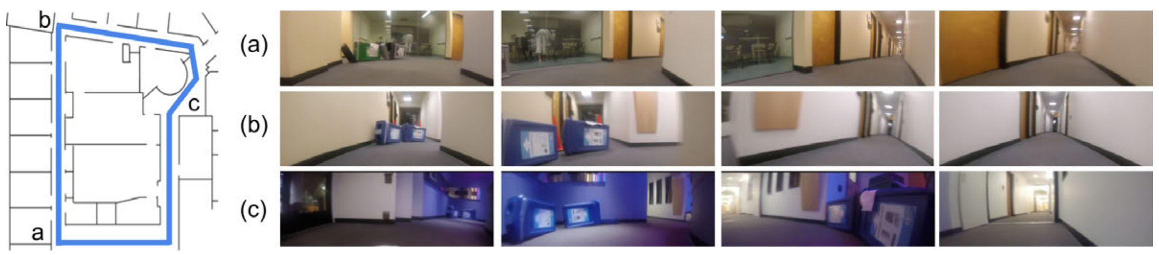


Figure 2.9: Real-world RC car experiments on the 5th floor of Cory hall. The path the robot can follow is drawn on the floor plan (left), however, this path is not provided to the RC car. Three example trajectories of the RC car navigating with our learned policy are shown.

for robot navigation (Sec. 2.4), we can achieve a sample-efficient, stable, and high performing learning algorithm.

Real-world results

We demonstrated the efficacy of our approach on a $\frac{1}{10}$th scale RC car in a real-world environment (Fig. 2.1). The task was to navigate the 5th floor of Cory Hall at 1.2m/s, which is challenging as these hallways contain tight turns, changing illumination, and glass walls (Fig. 2.9).

The RC car learning system was set up to maximize time spent gathering data, minimize computational burden for the car, and be fully autonomous. The computer onboard the RC car is an NVIDIA Jetson TX1, which is intended for embedded deep learning applications. However, all model training is performed offboard on a laptop for computational efficiency, while model inference is performed onboard. We therefore made the system fully asynchronous: the car continuously runs the reinforcement learning algorithm and sends data to the laptop, while the laptop continuously trains the model and periodically sends updated model parameters to the car. For full autonomy, the car automatically detects collisions using the onboard IMU and wheel encoder, and automatically backs up if a collision occurs. The only human intervention required is if the car flips over, which occurred approximately every 30 minutes.

In evaluating our approach, we chose the best design decisions from our simulation experiments: the model outputs are collision probabilities trained using classification, a large model horizon ($H = 12$, corresponding to 3.6m lookahead), and no bootstrapping. All other settings were the exact same as the simulation experiments.

Fig. 2.9 shows that, from training with only 4 hours of data in a complex, real-world environment using only raw camera images and no prior knowledge, the car could navigate significant portions of the environment. For example, the best trajectory travelled 197m, corresponding to nearly 2 loops (and 8 hallway lengths). However, sometimes the policy fails (Fig. 2.10); additional training should correct these mistakes.

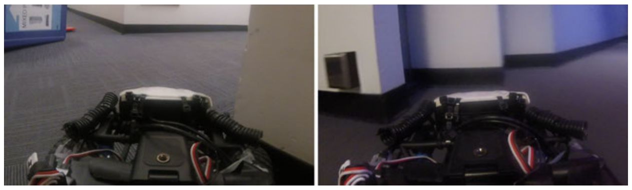


Figure 2.10: Example failure cases in which our approach turns too early (left) and turns too late (right).

Distance until crash (m)	Random policy	Double Q-learning with off-policy data	Our approach
Mean	3.4	7.2	52.4
Median	2.8	6.1	29.3
Max	8.0	21.5	197.0

Table 2.1: Evaluation of our learned policy navigating at 1.2m/s using only monocular images in a real-world indoor environment after 4 hours of self-supervised training, compared to a random policy and double Q-learning trained with the same data gathered by our approach.

Table 2.1 compares our method with a random policy and double Q-learning trained using the data gathered during our approach's learning. We trained double Q-learning in this way in order to compare the performance of the algorithms given the same state distribution. Our approach travels $17\times$ farther than the random policy and $7\times$ farther than double Q-learning. Qualitatively, our approach was smoothly driving straight when possible, while double Q-learning was exhibiting bang-bang control in that its steering angle is always at the limits.

2.6 Discussion

In this chapter, we presented a sample-efficient, stable, and high-performing reinforcement learning algorithm for learning robot navigation policies. By formalizing a generalized computation graph that subsumes value-based model-free and model-based learning, we subsequently instantiated this graph to form a suite of hybrid algorithms for robot navigation. Our simulated experiments evaluate which design decisions were important for sample-efficient and stable learning of robot navigation policies, and show our approach outperforms prior Q-learning based methods. Our real-world experiments on an RC car in a complex real-world environment show that our approach can learn to navigate significant portions of the environment using only monocular images with only 4 hours of training in a completely self-supervised manner.

Chapter 3

Transfer Learning

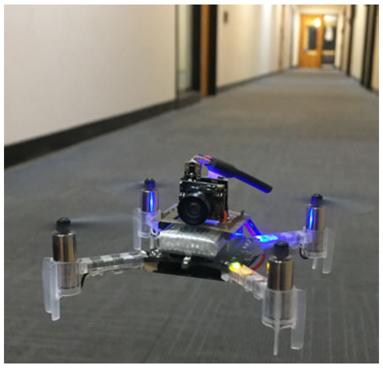


Figure 3.1: Our autonomous quadrotor navigating a building from raw monocular images using a learned collision avoidance policy trained with a simulator and one hour of real-world data.

In the previous chapter, we demonstrated our reinforcement learning algorithm based on generalized computation graphs can successfully learn from purely real-world data. However, as with all learning-based systems, the capacity of learned policies to generalize to new situations is determined in large part by the quantity and variety of the data that is available for training. While in principle autonomous robots could gather their own data directly in the real world, generalization is so strongly dependent on dataset size and diversity that it can almost always be improved simply by adding more experience, especially for fragile and safety-critical systems such as quadrotors, for which large datasets may be difficult to collect. It is therefore highly advantageous to integrate other, more plentiful sources of data into the training process. In this chapter, we investigate how a combination of simulated and real-world data can enable effective generalization for collision avoidance on a real-world nano aerial vehicle (NAV), shown in Fig. 3.1, using only an onboard monocular camera.

While transferring simulated experience into the real world has received considerable attention in recent years [38], [39], [90], [91], deployment of policies trained in simulation onto real-world robots poses a major challenge: complex real-world physical and visual phenomena are difficult to simulate accurately, and the systematic differences between simulation and reality are typically impossible to eliminate. Many of the prior works in this area have focused on differences that are irrelevant to the task; nuisance factors, such as variations in visual appearance, to which the optimal policy should be *invariant*. Such nuisance factors can be eliminated by regularizing for invariance. However, some aspects of the simulation, particularly in terms of the dynamics of the robot, differ from reality in systematic ways that cannot be ignored. This is especially important for small-scale aerial vehicles, where air currents, drift, and turbulence are significant. In principle, this mismatch can be addressed by fine-tuning models trained in simulation on real-world data. However, as we will show in our experiments, naïve fine-tuning with small, real-world datasets can result in catastrophic overfitting.

In this chapter, we instead aim to devise a transfer learning algorithm where the physical behavior of the vehicle is learned mostly from real-world data, while simulated experience provides for a visual perception system that generalizes to new environments. In essence, real-world experience is used to learn how to fly, while simulated experience is used to learn how to generalize. Rather than simply fine-tuning a deep neural network policy using real-world data, we separate our model into a perception and control subsystem. The perception subsystem transfers visual features from simulation, while the control subsystem is trained with real-world data. This enables our approach to transfer knowledge from simulation and generalize to new real-world environments more effectively than alternative techniques. We further evaluate several choices for training the visual system in simulation, and observe that visual features that are task-oriented, such as models trained with reinforcement learning specifically for navigation and collision avoidance, transfer substantially better than task-agnostic feature learners trained with unsupervised learning [92] or standard supervised pre-training techniques, such as pre-training on large image recognition datasets [93].

The main contribution of this chapter is a method for combining large amounts of simulated data with small amounts of real-world experience to train real-world collision avoidance policies for autonomous flight with deep reinforcement learning. The principle underlying our method is to learn about the physical properties of the vehicle and its dynamics in the real world, while learning visual invariances and patterns from simulation. We compare a variety of methods for learning the visual features, and find that reinforcement learning in simulation leads to the most transferable representations when compared to unsupervised and supervised alternatives. On a real-world nano aerial vehicle (NAV) collision avoidance task, our method can fly $4\times$ further compared to alternative methods, and can navigate through hallways with various lighting conditions and geometry.

3.1 Related Work

There has been much work on transfer learning for control policies [27], including from simulation to reality. Prior works have sought to transfer policies by combining simulated and real-world data, including techniques such as domain adaptation [28], [29] and feature space learning [30], [31]. These methods learn task-agnostic perception models, primarily in order to avoid requiring labels in the real-world. In contrast, our approach uses a task-specific perception model—specifically, the perceptual neural network layers from a policy learned in simulation—for transfer, which we demonstrate in our experiments is crucial for success.

Other approaches have sought to improve transfer by minimizing the gap between simulation and reality, either by bringing the simulator closer to reality [35], [36], making reality closer to the simulator [37], [38], or randomizing visual [39], [40] or physical [41]–[44] properties of the simulator. These approaches seek to reduce the policy overfitting to systemic or irrelevant differences between simulation and reality, while our approach is complementary in that it seeks to adapt models learned in simulation to the real-world using a limited amount of real-world data. Some prior works also acknowledge the existence of the reality gap and learn to adapt these imperfect models using real-world data [32]–[34]. In contrast to these methods, which are either evaluated on either low-dimensional or simple dynamical systems, our approach scales to raw image inputs and can cope with the highly nonlinear dynamics of a nano aerial vehicle by learning a scalable, sample-efficient, end-to-end latent dynamics model.

The general problem of addressing differences between training and test distributions has been extensively studied in the machine learning community [45]. With the recent advent of large datasets, prior work has shown that deep neural networks trained on these large datasets can enable easy transfer to new tasks via fine-tuning [46]–[49]. In our experiments, we attempted a similar transfer by using the perception layers from a neural network model trained on Imagenet [93] and fine-tuning; however, we found that this approach performed poorly compared to our method for the task of NAV collision avoidance.

There is extensive prior work on autonomous aerial flight, including approaches that use geometric mapping and path planning [13], [14], imitate an expert pilot or leverage expert labelled data [55], [70], [94], and learn from experience using large real-world datasets [60], [71]. In contrast to these works, our work focuses on developing a method for adapting to the real-world using a limited amount of real-world data, which is particularly important for the SWaP-constrained NAV platform used in this work.

3.2 Problem Formulation

Our goal is to learn a real-world control policy by leveraging data gathered in simulation in conjunction with a limited amount of real-world data. At each time step t, the robot selects an action $\mathbf{a}_t \in \mathcal{A}$ in state $\mathbf{s}_t \in \mathcal{S}$, proceeds to the next state $\mathbf{s}_{t+1}$ according to an unknown transition distribution $T(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, and receives a task-specific reward r_t. The objective of the robot is to learn the parameter vector θ of a policy distribution $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ such that the expected sum of discounted future rewards $\mathbb{E}_{\pi_\theta,T}[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}]$ is maximized, in which the discount factor $\gamma \in [0, 1)$ determines to what degree the robot cares about rewards in the distant future.

In order to train this policy π_θ, we assume we have access to both a simulator and a small dataset collected by the robot acting in the real world. The goal is therefore to learn a real-world policy π_θ using a real-world dataset $(\mathbf{s}_t, \mathbf{a}_t, r_t) \in \mathcal{D}^{\text{RW}}$, in combination with a simulated dataset $\mathcal{D}^{\text{SIM}}$, such that π_θ generalizes well in the real world.

3.3 Generalization through Simulation

We will now describe our approach for real-world robot learning using generalization through simulation. Our key insight is that the real world and simulation can serve complementary functions for robot learning: data gathered in the real world provides accurate signals about the dynamics of the robot, but suffers from a lack of visual diversity due to the difficulty of gathering experience in the real-world, while simulation provides an easy way to gather large amounts of visually diverse data, but suffers from unrealistic dynamics. Our approach therefore uses the real world data for learning the dynamics of the robot, while leveraging the simulation data to learn a generalizable visual perception system. We will first describe our real-world control policy learning approach,

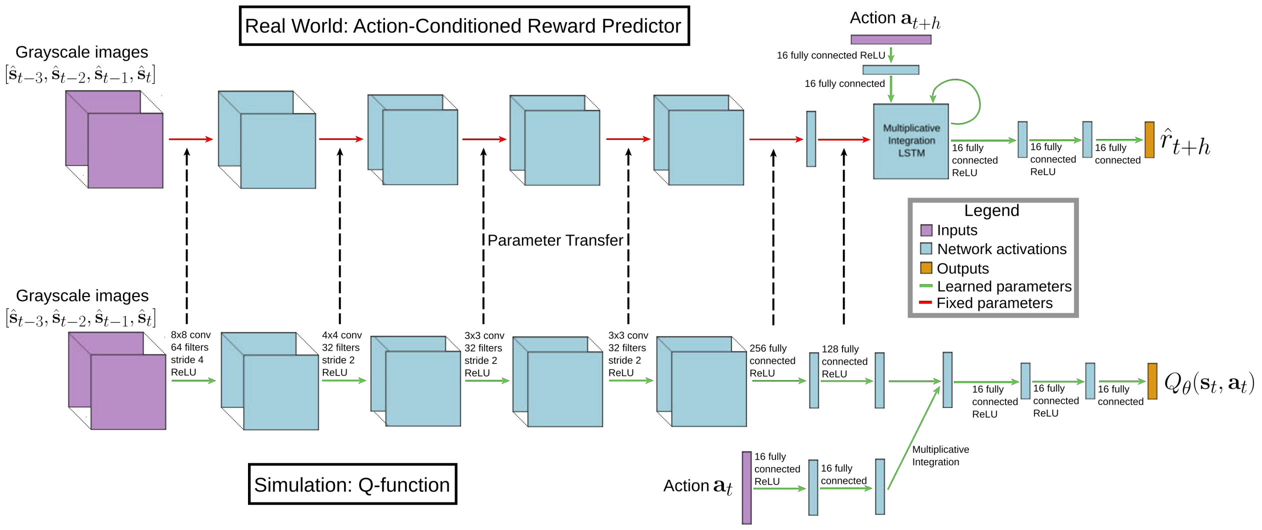

Real World: Action-Conditioned Reward Predictor
Grayscale images $[\hat{\mathbf{s}}_{t-3}, \hat{\mathbf{s}}_{t-2}, \hat{\mathbf{s}}_{t-1}, \hat{\mathbf{s}}_t]$
Action $\mathbf{a}_{t+h}$
16 fully connected ReLU
16 fully connected
Multiplicative Integration LSTM
16 fully connected ReLU
16 fully connected ReLU
16 fully connected
$\hat{r}_{t+h}$
Parameter Transfer
Legend
Inputs
Network activations
Outputs
Learned parameters
Fixed parameters
Grayscale images $[\hat{\mathbf{s}}_{t-3}, \hat{\mathbf{s}}_{t-2}, \hat{\mathbf{s}}_{t-1}, \hat{\mathbf{s}}_t]$
8x8 conv 64 filters stride 4 ReLU
4x4 conv 32 filters stride 2 ReLU
3x3 conv 32 filters stride 2 ReLU
3x3 conv 32 filters stride 2 ReLU
256 fully connected ReLU
128 fully connected ReLU
16 fully connected ReLU
16 fully connected ReLU
16 fully connected
$Q_\theta(\mathbf{s}_t, \mathbf{a}_t)$
Multiplicative Integration
Action $\mathbf{a}_t$
16 fully connected ReLU
16 fully connected
Simulation: Q-function

Figure 3.2: Our approach for leveraging both a simulator and real-world data. In simulation, we run reinforcement learning in order to learn a task-specific deep neural network Q-function model. Using real-world data from running the robot, we learn a deep neural network model that predicts future rewards given the current state and a future sequence of actions; this model can be used to form a control policy by selecting actions that maximize future rewards. In order to learn a generalizable reward prediction model with only an hours worth of real-world data, we transfer the perception neural network layers from the Q-function trained in simulation to be the perception module for the reward predictor. Our experiments demonstrate that (1) fine-tuning the Q-function on real-world data does not lead to good performance, (2) the reward predictor is better suited for real-world learning due to the limited amount of real-world data, and (3) learning a task-specific model in simulation improves transfer of the perception module.

and then discuss how to transfer a visual perception system learned in simulation to enable the real-world policy to generalize.

Real-World Policy Learning

Given that we will only have access to a small amount of real-world data, we require a policy learning algorithm that is sample-efficient. We therefore build off of the generalized computation graph from Chapter 2. In this chapter, we will instantiate the graph as an action-conditioned reward predictor $G_\theta(\mathbf{s}_t, \mathbf{A}_t^H)$ that takes as input the current state $\mathbf{s}_t$ and a sequence of H future planned actions $\mathbf{A}_t^H = (\mathbf{a}_t, ..., \mathbf{a}_{t+H-1})$, and outputs the predicted rewards $\hat{R}_t^H = (\hat{r}_t, ..., \hat{r}_{t+H-1})$ at each time step in the future.

At training time, the model parameters are updated using the real-world dataset to minimize the reward prediction error

$$\theta^* = \arg\min_\theta \sum_{(\mathbf{s}_t, \mathbf{A}_t^H, R_t^H) \in \mathcal{D}^{\text{RW}}} \| G_\theta(\mathbf{s}_t, \mathbf{A}_t^H) - R_t^H \|^2, \tag{3.1}$$

while at test time, the learned action-conditioned reward predictor is used by a finite-horizon optimal controller to select an action sequence that maximizes the predicted future rewards

$$\mathbf{A}^* = \arg\max_{\mathbf{A}} \sum_{h=0}^{H-1} \gamma^h \hat{r}_{t+h}. \tag{3.2}$$

At each time step, the controller solves for the optimal action sequence by solving Eqn. 3.2, executing the first action of the resulting action sequence, proceeding to the subsequent state, and then repeating this process in a receding horizon model predictive control (MPC) fashion. In order to actually find the optimal action sequence in Eqn. 3.2, we resort to approximate optimization methods. In particular, we use the cross entropy method (CEM) [87], which is a zeroth order stochastic optimization procedure.

We now instantiate the action-conditioned reward predictor as a deep neural network. Fig. 3.2 depicts the neural network architecture. The image state $\mathbf{s}_t$ is provided as input to a convolutional neural network, which outputs a latent representation of the state. This latent state then serves as the initial state of a latent dynamical system module, implemented as a recurrent neural network, which updates the latent state H times using the action sequence $\mathbf{A}_t^H$. Each of the H latent states is then passed through fully connected layers to produce the final reward predictions $\hat{R}_t^H$.

Transferring Visual Perception Systems from Simulation

Although the action-conditioned reward predictor is a sample-efficient policy learning algorithm, the policy is still prone to overfitting to the training data and may therefore fail to generalize to novel real-world environments due to the immense visual diversity of the real-world. We therefore seek to leverage simulation data in order to enable better real-world policy generalization.

In deciding how to leverage our simulator, we make two key observations: (1) current state-of-the-art simulators are good at providing realistic and diverse visual scenes [37], but do not accurately model the complex, real-world dynamics of NAVs and (2) the model learned in simulation should be task-specific and align with the real-world robot task in order for the model to learn to distill task-relevant features. Our approach will therefore learn a task-specific model in simulation, and then transfer the visual perception system part of the model to the real-world policy.

Learning a task-specific model. The task-specific model we learn is a deep neural network Q-function $Q_\theta(\mathbf{s}_t, \mathbf{a}_t)$ that is trained using Q-learning [95]; this Q-function represents the expected sum of future rewards an agent would achieve in state $\mathbf{s}_t$, executing action $\mathbf{a}_t$, and acting optimally thereafter. We use the Q-learning algorithm, as opposed to the action-conditioned reward predictor used for real-world policy learning, because (1) we have access to large amounts of data in simulation, which is a requirement for deep Q-learning, and (2) Q-learning can learn long-horizon tasks, which may improve the visual features that it learns.

Q-learning updates the parameters of the Q-function by minimizing the Bellman error for all state, action, reward, next state tuples in the (simulation-gathered) dataset:

$$\theta^* = \arg\min_{\theta} \sum_{\mathcal{D}^{\text{SIM}}} \|Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - [r_t + \gamma \max_{\mathbf{a}'} Q_\theta(\mathbf{s}_{t+1}, \mathbf{a}')]\|^2.$$

Using the Q-function, optimal actions can then be selected by finding the action that maximizes the Q-function:

$$\mathbf{a}^* = \arg\max_{\mathbf{a}} Q_\theta(\mathbf{s}_t, \mathbf{a}).$$

In deep Q-learning algorithms with discrete action spaces, this maximization can be performed optimally. However, for deep Q-functions that take as input continuous actions, this maximization can be approximated using stochastic optimization techniques [96], [97].

The Q-function neural network model is shown in Fig. 3.2. The model consists of three distinct neural network modules: a perception module consisting of a convolutional neural network for processing the input image state, an action module consisting of a fully connected neural network for processing the action, and a value module consisting of a fully connected neural network for combining the processed state and action to produce the resulting Q-value. We note that Q-function and action-conditioned reward predictor have a very similar structure, which illustrates the purpose of the generalized computation graph: both approaches have the same underlying mechanisms, but are trained slightly differently [96].

Visual perception system transfer. We will use the visual perception neural network layers trained when learning the task-specific Q-function in order to transfer the visual perception system from simulation to the real-world. Concretely, we will initialize the weights of the real-world policy's visual perception layers (Fig. 3.2 top) to the values of the visual perception layers from the Q-function learned in simulation (Fig. 3.2 bottom), and hold these perception layers fixed during real-world policy training. Although these layers could be further fine-tuned using the real-world data, we decided to hold these layers fixed to prevent the real-world policy from overfitting to the training data.

Algorithm Overview

We now provide a brief summarizing overview of our approach. First, we train a deep neural network Q-function using deep reinforcement learning in a visually diverse set of simulated environments. Then, we create the deep neural network action-conditioned reward prediction model, in which we use the perception layers from the simulation-trained Q-function to process the input image state. Next, we train the action-conditioned reward prediction model using real-world data gathered by the robot; however, when training the model, we do not update the parameters of the perception layers. Using this action-conditioned prediction model trained on real-world data, but leveraging a task-specific visual perception system trained in simulation, our real-world policy will be better able to generalize to novel environments.

3.4 Experiments

We evaluate our approach on a collision avoidance task with a nano aerial vehicle (NAV). This platform is well-suited for testing our transfer learning approach because it is SWaP constrained. The NAV we use is the Crazyflie 2.0 nano quadcopter [98], shown in Fig. 3.1. The Crazyflie has

	Simulation Model	Perception System Transferred	Real-World Learned Model	Uses Real-World Data	Perception Layers Trained with Real-World Data	Time Until Collision (seconds, max 86)	Percentage Hallway Traversed
sim only	Task-specific	N/A	N/A	✗	N/A	16.5 (0.5)	19
sim fine-tuned	Task-specific	✗	Q-function	✓	✓	6.0 (28.5)	7
sim fine-tuned perception fixed	Task-specific	✗	Q-function	✓	✗	6.5 (66.5)	8
real-world only	N/A	✗	ACRP	✓	✓	7.8 (30.0)	9
supervised (ImageNet) transfer	N/A	✓	ACRP	✓	✗	9.5 (4.5)	11
unsupervised (VAE) transfer	Task-agnostic	✓	ACRP	✓	✗	21.0 (19.3)	24
GtS (ours)	Task-specific	✓	ACRP	✓	✗	**85.8 (2.5)**	**100**

Table 3.1: Comparison of our generalization through simulation (GtS) approach with prior methods for the task of flying down a straight hallway. Note that this hallway was not in the real-world training data. Each approach and baseline can be characterized with five properties: (1) is the simulation model used for transfer task-specific or task-agnostic? (2) is the real-world perception module transferred from another model? (3) is the model trained in the real world a neural network Q-function or a neural network action-conditioned reward predictor? (4) is real-world data used for training? and (5) are the perception layers trained with the real-world data or held fixed? Each approach attempted the task 5 times, and the time to collision (median and interquartile range) was recorded. Our approach was able to reliably fly down the entire hallway without colliding, consistently reaching the maximum flight time.

dimensions 92x92x29mm and weighs 27 grams. The action space consists of forward speed, yaw rate, and height, which is enabled by a downward-facing optical flow and height sensor. To allow for perceptual navigation, we added a 3.4 gram monocular camera to the Crazyflie. With the added weight, the maximum flight duration is approximately four minutes. Communication with the Crazyflie is done via a radio-to-USB dongle connected to a nearby laptop. All action selection using the learned policies is performed on this laptop, but could be deployed on the NAV in future work [99].

For training the simulation policy, we used the Gibson simulator [37], which contains a large variety of 3D scanned environments (Fig. 3.3). We modelled the quadrotor as a camera with simple point mass dynamics, meaning that the actions directly control the pose of the robot camera. Although these dynamics are a severe oversimplification of real-world NAV dynamics, the goal of the simulator is not to accurately simulate the NAV, but rather to enable the collection of a visually diverse set of data that can then be used to train a task-specific model for the purpose of visual transfer. We will show in our experiments that even with this oversimplified dynamics model, we are still able to successfully transfer the visual perception system from our simulation-trained model.

Simulation data was gathered by running separate instances of Q-learning in 16 different environments. The reward function for Q-learning was 0 for no collision, and -1 for collision. After all the instances of Q-learning finished training, we trained a single Q-function on all of the 17 million simulation-gathered data points. Real-world data was gathered by running the simulation-trained policy in a single hallway on the 5th floor of Cory Hall at UC Berkeley for one hour, resulting in 14,000 data points.

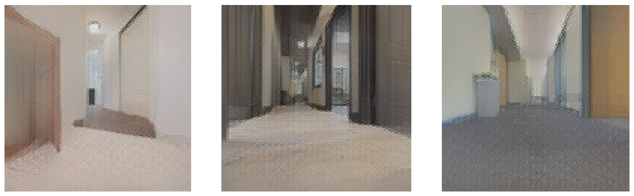


Figure 3.3: A subset of the environments used for simulation training.

For both simulation and the real world, the state consisted of the four most recent camera image converted to grayscale and downsized to a resolution of 72×96, resulting in the state space $\mathcal{S} \in \mathbb{R}^{4 \times 72 \times 96}$. The action consisted solely of the yaw angular velocity $\mathcal{A} \in \mathbb{R}^1$ because the height and speed were held constant at 0.4 meters and 0.3 m/s, respectively. Data was gathered at 4 Hz, the discount factor γ was set to 1, and the action-conditioned reward prediction model had a horizon of $H = 12$, which corresponds to predicting 3 seconds into the future.

In our experimental evaluation, we seek to answer the following questions:

Q1 Does including real-world data improve performance?

Q2 Does the action-conditioned reward predictor lead to better real-world policies compared to Q-learning?

Q3 Is a task-specific or task-agnostic simulation-trained model better for real-world transfer?

Q4 Does transferring the perception module from the simulation-trained model improve real-world performance?

We compare our approach to the following methods:

- Sim only: The Q-function policy trained on all of the simulated data.
- Sim fine-tuned: The Q-function policy trained on all of the simulated data, and then fine-tuned solely on the real-world data.
- Sim fine-tuned perception fixed: The Q-function policy trained on all of the simulated data, and then fine-tune only the non-perception layers on the real-world data.
- Real-world only: The action-conditioned reward predictor trained solely on the real-world data.
- Supervised (ImageNet) transfer: Using pre-trained convolutional features from a model [100] trained on Imagenet [93] for the perception module, and training the action-conditioned reward predictor using the real-world data with the perception layers held fixed.
- Unsupervised (VAE) transfer: A variational autoencoder [92] generative model is trained on the simulated image data. The encoder, which maps input images to a concise latent state, is then used as the perception module for the action-conditioned reward predictor, which is trained on the real-world data.

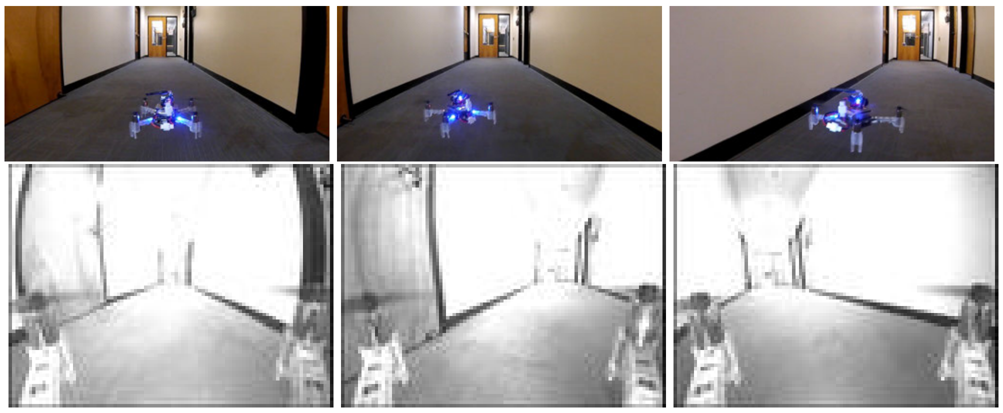
In order to evaluate the generalization capabilities of our approach, we present results in hallways not present in the real-world training dataset. Table 3.1 compares our generalization through simulation approach with all of the considered prior methods in a novel straight hallway, and Fig. 3.4a shows first- and third-person images of the NAV flying using our approach. Our method consistently flew the full length of the hallway without colliding, while the best prior method could only fly down a quarter of the hallway before colliding. Although flying down a straight hallway appears to be an easy task, the NAV drifts substantially due to imprecise sensors and environmental disturbances, and therefore avoiding collisions is non-trivial.

The sim only approach did not perform well and typically collided at doors. The sim fine-tuned and sim fine-tuned perception fixed models had trials that were indeed better than the sim only model, however their performances were inconsistent, indicating that Q-learning methods have difficulty fine-tuning on a limited amount of real-world data. Meanwhile, the policy trained solely on real-world data made some progress, but likely did not perform well due to overfitting.

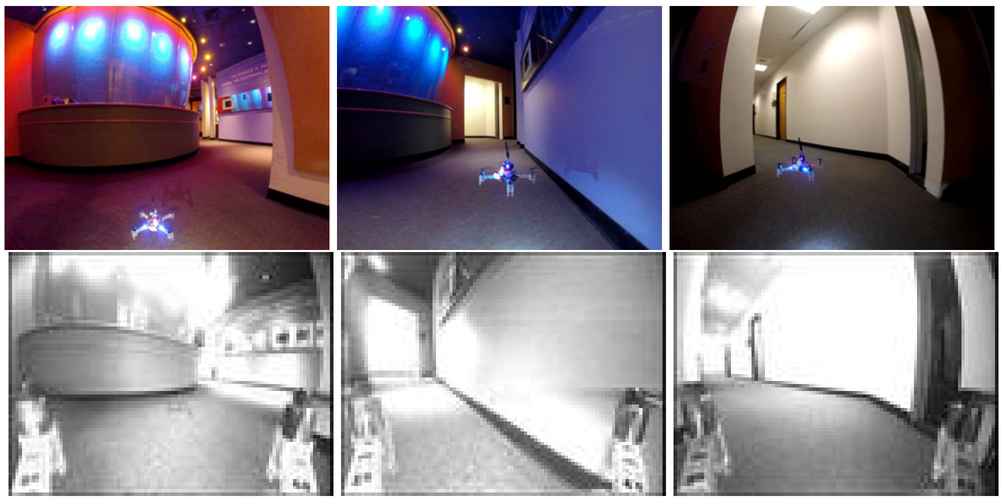
% Successful Trials (out of 5)	Straight Hallway with Tilted Camera	Curved Hallway	Zig-zag Hallway
Sim only	0	0	0
Unsupervised (VAE) transfer	0	0	0
GtS (ours)	80	60	80

Table 3.2: Comparison of our generalization through simulation approach with the best two prior methods from Table 3.1 on three more difficult tasks. Our approach succeeds in the majority of the trials, while the prior methods fail.

(a) Straight Hallway



(b) Curved Hallway



(c) Zig-zag Hallway

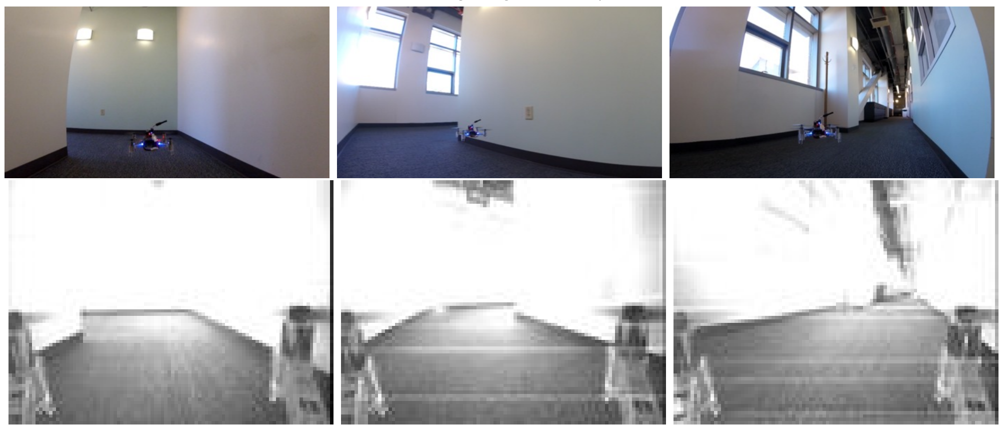


Figure 3.4: Our learning-based approach, using only the onboard, grayscale, 72 × 96 resolution camera images, flying through a straight, curved, and zig-zag hallway.

In contrast, our approach, which combines learning with real-world data with simulation model pre-training, results in improved real-world performance (**Q1** and **Q4**). Additionally, compared to the sim fine-tuned and sim fine-tuned perception fixed approaches, our method is able to leverage both a simulation-trained model and real-world data, indicating that the action-conditioned reward predictor is crucial for sample-efficient and stable learning (**Q2**). Lastly, our approach outperforms methods that transfer perception modules from task-agnostic models, showing that training task-specific models in simulation is beneficial for transfer (**Q3**).

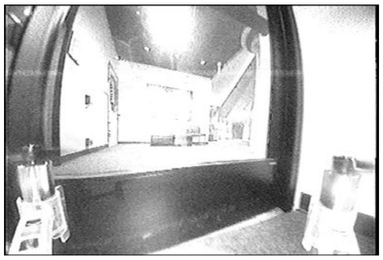


Figure 3.5: Example failure: collision with a glass door.

We also ran three additional experiments comparing our approach to the two best approaches in the straight hallway—sim only and VAE transfer—in the same straight hallway, but with the camera angle tilted down by 20 degrees, a curved hallway with varying lighting, and a zig-zag hallway. Table 3.2 summarizes the results, and Fig. 3.4 shows first- and third-person images of our approach flying. Our approach was able to fly through these difficult environments the majority of the trials, while the best prior methods were entirely unsuccessful. When our approach did fail, it was oftentimes reasonable; for example, in the curved hallways, in 30% of the trials the NAV collided with a glass door (Fig. 3.5). This is not surprising: the real-world data never included glass doors, and furthermore, many glass doors in simulation were actually traversable in simulation. However, with additional real-world data, the NAV would hopefully learn from these errors, which is the foundation of learning-based approaches.

3.5 Discussion

In this chapter, we presented an approach for learning generalizable real-world control policies using a simulator and a limited amount of real-world data. Our generalization through simulation approach uses the simulator to learn a task-specific model, and then transfers the perception layers to a sample-efficient, action-conditioned reward predictor that is trained on real-world data. Our experiments evaluate the design decisions of our method and show that our approach enables a nano aerial vehicle to fly through novel, complex hallway environments.

Part II

Supervision

Chapter 4

Self-Supervision

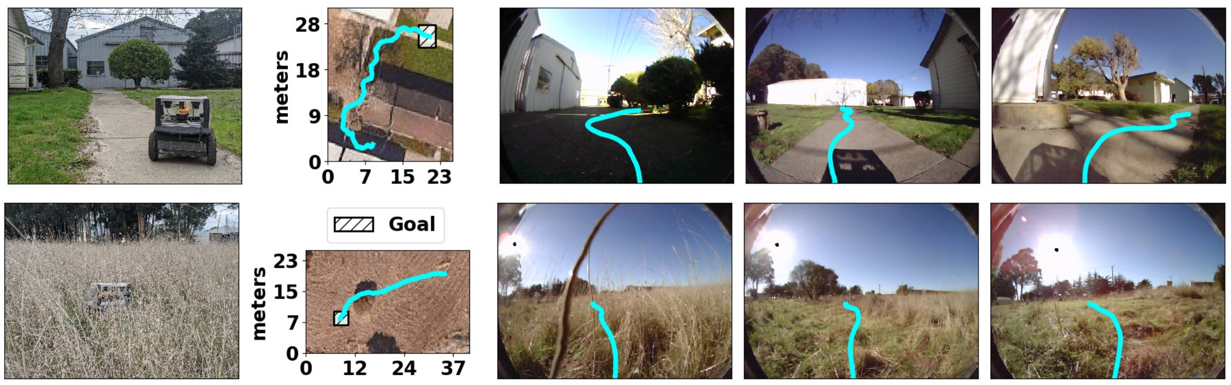


Figure 4.1: BADGR is an end-to-end learning-based mobile robot navigation system that can be trained with self-supervised off-policy data gathered in real-world environments, without any simulation or human supervision. Using only RGB images and GPS, BADGR can follow sparse GPS waypoints without colliding while (top row) preferring smooth concrete paths and (bottom row) ignoring geometrically distracting obstacles such as tall grass.

In part I, we developed reinforcement algorithms that quickly learned from a limited amount of data. However, these algorithms assumed the learning signal was known. While this assumption was valid for the given task—collision avoidance—we will need to develop new algorithms that do not assume the learning signal is given in order to enable mobile robot navigation beyond pure collision avoidance. In this chapter, we study how autonomous, self-supervised learning from experience can enable a robot to learn about the physical attributes—also known as *affordances*—of its environment using raw visual perception and without human-provided labels or geometric maps.

We investigate how the robot's own past experience can provide *retrospective* self-supervision: for many physically salient navigational objectives, such as avoiding collisions or preferring smooth over bumpy terrains, the robot can autonomously measure how well it has fulfilled its objective, and then retrospectively label the preceding experience so as to learn a *predictive* model for these objectives. For example, by experiencing collisions and bumpy terrain, the robot can learn, given an observation and a candidate plan of future actions, which actions might lead to bumpy or smooth to terrain, and which actions may lead to collision. This in effect constitutes a self-supervised multi-task reinforcement learning problem.

Based on this idea, we present a fully autonomous, self-improving, end-to-end learning-based system for mobile robot navigation, which we call BADGR—the Berkeley Autonomous Driving Ground Robot. BADGR works by gathering off-policy data—such as from a random control policy—in real-world environments, and uses this data to train a model that predicts future relevant events—such as collision, position, or terrain properties—given the current sensor readings and the recorded executed future actions. Using this model, BADGR can then plan into the future and execute actions that avoid certain events, such as collision, and actively seek out other events, such as smooth terrain. BADGR constitutes a fully autonomous self-improving system because it gathers data, labels the data in a self-supervised fashion, trains the predictive model in order to plan and act, and can autonomously gather additional data to further improve itself.

The primary contribution of this chapter is an end-to-end learning-based mobile robot navigation system that can be trained entirely with self-supervised off-policy data gathered in real-world environments, without any simulation or human supervision. Our results demonstrate that our BADGR system can learn to navigate in real-world environments with geometrically distracting obstacles, such as tall grass, and can readily incorporate terrain preferences, such as avoiding bumpy terrain, using only 42 hours of autonomously collected data. Our experiments show that our method can outperform a LIDAR policy in complex real-world settings, generalize to novel environments, and can improve as it gathers more data.

4.1 Related Work

Learning-based methods have shown promise in addressing these limitations by learning from data. One approach to improve upon SLAM methods is to directly estimate the geometry of the scene [50]–[52]. However, these methods are limited in that the geometry is only a partial description of the environment. Only learning about geometry can lead to unintended consequences, such as believing that a field of tall grass is untraversable. Semantic-based learning approaches attempt to address the limitations of purely geometric methods by associating the input sensory data with semantically meaningful labels, such as which pixels in an image correspond to traversable or bumpy terrain. However, these methods typically depend on existing SLAM approaches [18], [21], [57]–[59] or humans [53], [54] in order to provide the semantic labels, which consequently means these approaches either inherit the limitations of geometric approaches or are not autonomously self-improving. Methods based on imitation learning have been demonstrated on real-world robots [20], [55], [56], but again depend on humans for expert demonstrations, which does not constitute a continuously self-improving system. End-to-end reinforcement learning approaches have shown promise in automating the entire navigation pipeline. However, these methods have typically focused on pure geometric reasoning, require on-policy data, and often utilize simulation due to constraints such as sample efficiency [39], [96], [101]–[104]. Prior works have investigated learning navigation policies directly from real-world experience, but typically require a person [71], [102], [105] or SLAM algorithm [60] to gather the data, assume access to the ground-truth robot state [106], learn using low-bandwidth sensors [107], or only perform collision avoidance [96], [108]. Our approach overcomes the limitations of these prior works by designing an end-to-end reinforcement learning approach that directly learns to predict relevant navigation cues with a sample-efficient, off-policy algorithm, and can continue to improve with additional experience via a self-supervised data labelling mechanism that does not depend on humans or SLAM algorithms.

4.2 The Berkeley Autonomous Driving Ground Robot

Our goal is to enable a mobile robot to navigate in real-world environments. Our method, BADGR, is an end-to-end learning-based navigation system that can be trained entirely with self-supervised data gathered autonomously in real-world environments, without any simulation or human supervision, and can improve as it gathers more data.

BADGR autonomously gathers large amounts of off-policy data in real-world environments. Using this data, BADGR labels relevant events—such as collisions or bumpy terrain—in a self-supervised manner, and adds these labelled events back into the dataset. BADGR then trains a predictive model that takes as input the current camera images and a future sequence of actions, which correspond to linear and angular velocity commands, and predicts the relevant future events. When deploying the trained BADGR system, the user specifies a reward function that encodes the task they want the robot to accomplish in terms of these relevant events—such as to reach a goal while avoiding collisions and bumpy terrain—and the robot autonomously plans and executes actions that maximize this reward.

In order to build a self-supervised learning-based navigational system, BADGR uses *retrospective* self-supervision. This means that the robot must be able to experience events, such as collisions, and then learn to avoid (or seek out) such events in the future. In order to learn using retrospective self-supervision, the robot can only learn about events it has experienced and that can be measured using the onboard sensors, and that experiencing these events, even undesirable events such as colliding, is acceptable. We believe this approach to autonomous self-supervised robot learning is realistic for many real-world autonomous mobile robot applications.

In the following sections, we will describe the robot, data collection and labelling, model training, and planning components, followed by a summarizing overview of the entire system.

Mobile Robot Platform

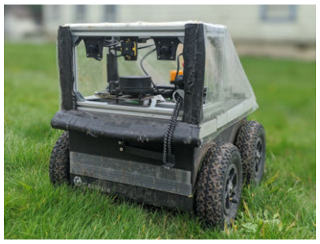


Figure 4.2: The mobile robot.

The specific design considerations for the robotic platform focus on enabling long-term autonomy with minimal human intervention. The robot we use is the Clearpath Jackal, shown in Fig. 4.2. The Jackal measures $508\text{mm} \times 430\text{mm} \times 250\text{mm}$ and weighs 17kg, making it ideal for navigating in both urban and off-road environments. The Jackal is controlled by specifying the desired linear and angular velocity, which are used as setpoints for the low-level differential drive controllers. The default sensor suite consists of a 6-DOF IMU, which measures linear

acceleration and angular velocity, a GPS unit for approximate global
position estimates, and encoders to measure the wheel velocity. In addition, we added the following sensors on top of the Jackal: two forward-facing $170°$ field-of-view 640×480 cameras, a 2D LIDAR, and a compass.

Inside the Jackal is an NVIDIA Jetson TX2 computer, which is ideal for running deep learning applications, in addition to interfacing with the sensors and low-level microcontrollers. Data is saved to an external SSD, which must be large and fast enough to store 1.3GB per minute streaming in from the sensors. Experiments were monitored remotely via a 4G smartphone mounted on top of the Jackal, which allowed for video streaming and, if needed, teleoperation.

The robot was designed primarily for robustness, with a relatively minimal and robust sensor suite, in order to enable long-term autonomous operation for large-scale data collection.

Data Collection

We design the data collection methodology to enable gathering large amounts of diverse data for training with minimal human intervention. The first consideration is the mobile robot platform (Fig. 4.2), which was designed primarily for robustness, with a relatively minimal and robust sensor suite, in order to enable long-term autonomous operation for large-scale data collection.

The first consideration when designing the data collection policy is whether the learning algorithm requires on-policy data. On-policy data collection entails alternating between gathering data using the current policy, and retraining the policy using the most recently gathered data. On-policy data collection is highly undesirable because only the most recently gathered data can be used for training; all previously gathered data must be thrown out. In contrast, off-policy learning algorithms can train policies using data gathered by any control policy. Due to the high cost of gathering data with real-world robotic systems, we choose to use an off-policy learning algorithm in order to be able to gather data using any control policy and train on all of the gathered data.

The second consideration when designing the data collection policy is to ensure the environment is sufficiently explored, while also ensuring that the robot execute action sequences it will realistically wish to execute at test time. A naïve uniform random control policy is inadequate because the robot will primarily drive straight due to the linear and angular velocity action interface of the robot, which will result in both insufficient exploration and unrealistic test time action sequences. We therefore use a time-correlated random walk control policy to gather data, which is visualized in Fig. 4.3.

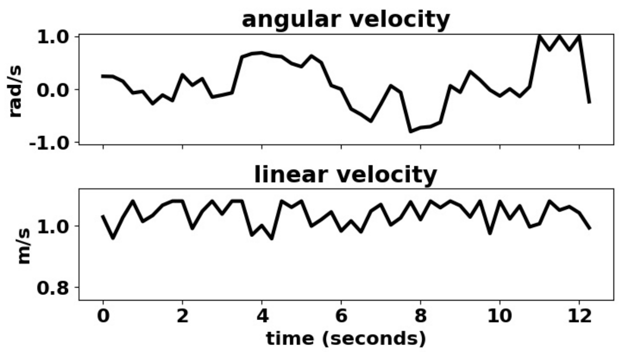


Figure 4.3: An example plot of the commanded angular and linear velocities from our time-correlated random control policy that is used to gather data.

As the robot is gathering data using the random control policy, it will require a mechanism to detect if it is in collision or stuck, and an automated controller to reset itself in order to continue gathering data.

We detect collisions in one of two ways, either using the LIDAR to detect when an obstacle is near or the IMU to detect when the robot is stuck due to an obstacle. We used the LIDAR collision detector in urban environments in order to avoid damaging property, and the IMU collision detector in off-road environments because the LIDAR detector was overly pessimistic, such as detecting grass as an obstacle. Once a collision is detected, a simple reset policy commands the robot to back up and rotate. However, sometimes the reset policy is insufficient, for example if the robot flips over (Fig. 4.4), and a person must manually reset the robot.

As the robot collects data, all the raw sensory data is saved onboard. After data collection for the day is complete, the data is copied to a desktop machine and subsampled down to 4Hz.

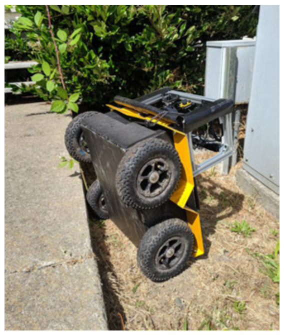


Figure 4.4: While collecting data, the robot will periodically require a manual intervention to reset from catastrophic failures, though recovery is usually automatic.

Self-Supervised Data Labelling

BADGR then goes through the raw, subsampled data and calculates labels for specific navigational events. These events consist of anything pertinent to navigation that can be extracted from the data in a self-supervised fashion.

In our experiments, we consider three different events: collision, bumpiness, and position. A collision event is calculated as occurring when, in urban environments, the LIDAR measures an obstacle to be close or, in off-road environments, when the IMU detects a sudden drop in linear acceleration and angular velocity magnitudes. A bumpiness event is calculated as occurring when the angular velocity magnitudes measured by the IMU are above a certain threshold. The position is determined by an onboard state estimator that fuses wheel odometry and the IMU to form a local position estimate.

Importantly, while these events are calculated using human-engineered functions, we still call this labelling scheme self-supervised because, once these simple labelling functions are created, all current and future data can be labelled with zero additional human effort. The labeling is automatic, and does not require any complex models, only simple evaluation with on-board sensors. Our approach stands in contrast to more standard methods for off-road navigation, which might require manual labeling of images to, for example, segment out traversable and impassable areas in an image [53].

After BADGR has iterated through the data, calculated the event labels at each time step, and added these event labels back into the dataset, BADGR can then train a model to predict which actions lead to which navigational events.

Predictive Model

The model at the core of BADGR is based on the generalized computation graph from Chapter 2. The learned predictive model takes as input the current sensor observations and a sequence of future intended actions, and predicts the future navigational events. We denote this model as

$f_\theta(\mathbf{o}_t, \mathbf{a}_{t:t+H}) \rightarrow \hat{\mathbf{e}}^{0:K}_{t:t+H}$, which defines a function f parameterized by vector θ that takes as input the current observation $\mathbf{o}_t$ and a sequence of H future actions $\mathbf{a}_{t:t+H} = (\mathbf{a}_t, \mathbf{a}_{t+1}, ..., \mathbf{a}_{t+H-1})$, and predicts K different future events $\hat{\mathbf{e}}^k_{t:t+H} = (\hat{\mathbf{e}}^k_t, \hat{\mathbf{e}}^k_{t+1}, ..., \hat{\mathbf{e}}^k_{t+H-1})\ \forall k \in \{0, ..., K-1\}$.

The model we learn is an image-based, action-conditioned predictive deep neural network, shown in Fig. 4.5. The network first processes the input image observations using convolutional and fully connected layers. The final output of the these layers serves as the initialization for a recurrent neural network, which sequentially processes each of the H future actions $\mathbf{a}_{t+h}$ and outputs the corresponding predicted future events $\hat{\mathbf{e}}^{0:K}_{t+h}$.

The model is trained—using the observations, actions, and event labels from the collected dataset—to minimize a loss function that penalizes the distance between the predicted and ground truth events

$$\mathcal{L}(\theta, \mathcal{D}) = \sum_{(\mathbf{o}_t, \mathbf{a}_{t:t+H}) \in \mathcal{D}} \sum_{k=0}^{K-1} \mathcal{L}^k(\hat{\mathbf{e}}^k_{t:t+H}, \mathbf{e}^k_{t:t+H}) \quad : \quad \hat{\mathbf{e}}^{0:K}_{t:t+H} = f_\theta(\mathbf{o}_t, \mathbf{a}_{t:t+H}). \tag{4.1}$$

The individual losses $\mathcal{L}^k$ for each event are either cross entropy if the event is discrete, or mean squared error if the event is continuous. The neural network parameter vector θ is trained by performing minibatch gradient descent on the loss in Eqn. 4.1.

Planning

Given the trained neural network predictive model, this model can then be used at test time to plan and execute desirable actions.

We first define a reward function $R(\hat{\mathbf{e}}^{0:K}_{t:t+H})$ that encodes what we want the robot to do in terms of the model's predicted future events. For example, the reward function could encourage driving towards a goal while discouraging collisions or driving over bumpy terrain. The specific reward function we use is specified in the experiments section.

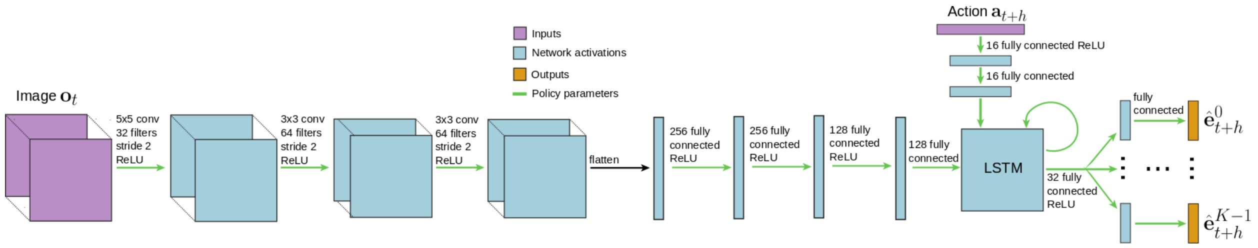


Figure 4.5: Illustration of the deep neural network predictive model at the core of our learning-based navigation policy. The neural network takes as input the current RGB image and processes it with convolutional and fully connected layers to form the initial hidden state of a recurrent LSTM unit [109]. This recurrent unit takes as input H actions in a sequential fashion, and produces H outputs. These outputs of the recurrent unit are then passed through additional fully connected layers to predict all K events for all H future time steps. These predicted future events, such as position, if the robot collided, and if the robot drove over bumpy terrain, enable a planner to select actions that achieve desirable events, such reaching a goal, and avoid undesirable events, such as collisions and bumpy terrain.

Using this reward function and the learned predictive model, we solve the following planning problem at each time step

$$\mathbf{a}^*_{t:t+H} = \arg\max_{\mathbf{a}_{t:t+H}} R(f_\theta(\mathbf{o}_t, \mathbf{a}_{t:t+H})), \tag{4.2}$$

execute the first action, and continue to plan and execute following the framework of model predictive control.

We solve Eqn. 6.3 using the zeroth order stochastic optimizer from [110]. This optimizer works by maintaining a running estimate $\hat{\mathbf{a}}_{0:H}$ of the optimal action sequence. Each time the planner is called, N action sequences $\tilde{\mathbf{a}}^{0:N}_{0:H}$ are sampled that are time-correlated and centered around this running action sequence estimate

$$\begin{aligned} &\epsilon^n_h \sim \mathcal{N}(0, \sigma \cdot \mathbf{I}) \;\forall n \in \{0...N-1\}, h \in \{0...H-1\} \\ &\tilde{\mathbf{a}}^n_h = \beta \cdot (\hat{\mathbf{a}}_{h+1} + \epsilon^n_h) + (1-\beta) \cdot \tilde{\mathbf{a}}^n_{h-1} \text{ where } \tilde{\mathbf{a}}_{h<0} = 0, \end{aligned} \tag{4.3}$$

in which the parameter σ determines how close the sampled action sequences should be to the running action sequence estimate, and the parameter $\beta \in [0,1]$ determines the degree to which the sampled action sequences are correlated in time.

Each action sequence is then propagated through the predictive model in order to calculate the reward $\tilde{R}^n = R(f_\theta(\mathbf{o}_t, \tilde{\mathbf{a}}^n_{0:H}))$. Given each action sequence and its corresponding reward, we update the running estimate of the optimal action sequence via a reward-weighted average

$$\hat{\mathbf{a}}_{0:H} = \frac{\sum_{n=0}^{N} \exp(\gamma \cdot R^n) \cdot \tilde{\mathbf{a}}^n_{0:H}}{\sum_{n'=0}^{N} \exp(\gamma \cdot R^{n'})}, \tag{4.4}$$

in which $\gamma \in \mathrm{R}^+$ is a parameter that determines how much weight should be given to high-reward action sequences.

Each time the planner is called, new action sequences are sampled according to Eqn. 4.3, these action sequences are propagated through the learned predictive model in order to calculate the reward of each sequence, the running estimate of the optimal action sequence is updated using Eqn. 4.4, and the robot executes the first action $\hat{\mathbf{a}}_0$.

This optimizer is more powerful than other zeroth order stochastic optimizers, such as random shooting or the cross-entropy method [111], because it warm-starts the optimization using the solution from the previous time step, uses a soft update rule for the new sampling distribution in order to leverage all of the sampled action sequences, and considers the correlations between time steps. In our experiments, we found this more powerful optimizer was necessary to achieve good planning.

Algorithm Summary

We now provide a brief summary of how our BADGR system operates during training (Alg. 2) and deployment (Alg. 3).

During training, BADGR gathers data by executing actions according to the data collection policy and records the onboard sensory observations and executed actions. Next, BADGR uses the gathered dataset to self-supervise the event labels, which are added back into the dataset. This dataset is then used to train the learned predictive model.

When deploying BADGR, the user first defines a reward function that encodes the specific task they want the robot to accomplish. BADGR then uses the trained predictive model, current observation, and reward function to plan a sequence of actions that maximize the reward function. The robot executes the first action in this plan, and BADGR continues to alternate between planning and executing until the task is complete.

Algorithm 2 Training BADGR

```
 1: initialize dataset $\mathcal{D} \leftarrow \emptyset$
 2: **while** not done collecting data **do**
 3:     get current observation $\mathbf{o}_t$ from sensors
 4:     get action $\mathbf{a}_t$ from data collection policy
 5:     add $(\mathbf{o}_t, \mathbf{a}_t)$ to $\mathcal{D}$
 6:     execute $\mathbf{a}_t$
 7:     **if** in collision **then**
 8:         execute reset maneuver
 9:     **end if**
10: **end while**
11: **for each** $(\mathbf{o}_t, \mathbf{a}_t) \in \mathcal{D}$ **do**
12:     calculate event labels $\mathbf{e}_t^{0:K}$ using self-supervision
13:     add $\mathbf{e}_t^{0:K}$ to $\mathcal{D}$
14: **end for**
15: use $\mathcal{D}$ to train predictive model $f_\theta$ by minimizing Eqn. 4.1
```

Algorithm 3 Deploying BADGR

```
1: **input**: trained predictive model $f_\theta$, reward function $R$
2: **while** task is not complete **do**
3:     get current observation $\mathbf{o}_t$ from sensors
4:     solve Eqn. 6.3 using $f_\theta, \mathbf{o}_t$, and $R$
       to get the planned action sequence $\mathbf{a}^*_{t:t+H}$
5:     execute the first action $\mathbf{a}^*_t$
6: **end while**
```

4.3 Experiments

In our experimental evaluation, we study how BADGR can autonomously learn successful navigation strategies in real-world environments, improve as it gathers more data, and generalize to unseen environments. We also compare BADGR to purely geometric approaches. Videos, code, and additional material are available on our website[1].

We performed our evaluation in a real-world outdoor environment consisting of both urban and off-road terrain. BADGR autonomously gathered **34 hours** of data in the urban terrain and **8 hours** in the off-road terrain. Although the amount of data gathered may seem significant, the total dataset consisted of 720,000 off-policy datapoints, which is smaller than currently used datasets in computer vision [112] and significantly smaller than the number of samples often used by deep reinforcement learning algorithms [113].

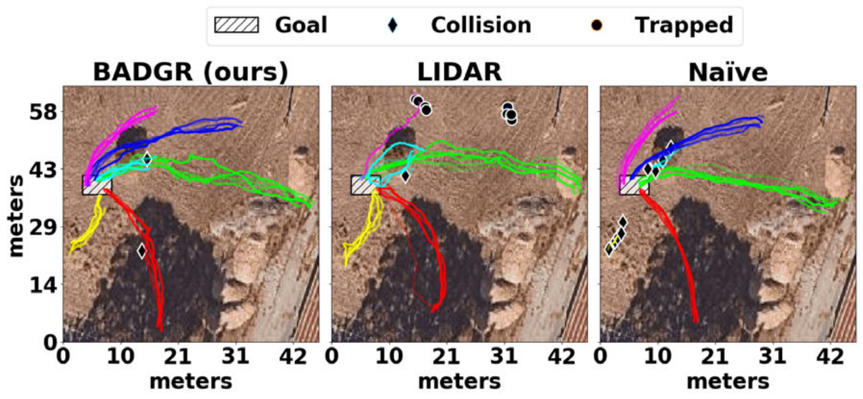
Our evaluations consist of tasks that involve reaching a goal GPS location, avoiding collisions, and preferring smooth over bumpy terrain. In order for BADGR to accomplish these tasks, we design the reward function that BADGR uses for planning as such

$$\begin{aligned} R(\hat{\mathbf{e}}^{0:K}_{t:t+H}) &= -\sum_{t'=t}^{t+H-1} R^{\text{COLL}}(\hat{\mathbf{e}}^{0:K}_{t'}) + \\ &\qquad \alpha^{\text{POS}} \cdot R^{\text{POS}}(\hat{\mathbf{e}}^{0:K}_{t'}) + \alpha^{\text{BUM}} \cdot R^{\text{BUM}}(\hat{\mathbf{e}}^{0:K}_{t'}) \\ R^{\text{COLL}}(\hat{\mathbf{e}}^{0:K}_{t'}) &= \hat{\mathbf{e}}^{\text{COLL}}_{t'} \\ R^{\text{POS}}(\hat{\mathbf{e}}^{0:K}_{t'}) &= (1 - \hat{\mathbf{e}}^{coll}_{t'}) \cdot \frac{1}{\pi} \angle(\hat{\mathbf{e}}^{\text{POS}}_{t'}, \mathbf{p}^{\text{GOAL}}) + \hat{\mathbf{e}}^{coll}_{t'} \\ R^{\text{BUM}}(\hat{\mathbf{e}}^{0:K}_{t'}) &= (1 - \hat{\mathbf{e}}^{coll}_{t'}) \cdot \hat{\mathbf{e}}^{\text{BUM}}_{t'} + \hat{\mathbf{e}}^{coll}_{t'}, \end{aligned} \tag{4.5}$$

where α^{POS} and α^{BUM} are user-defined scalars that weight how much the robot should care about reaching the goal and avoiding bumpy terrain. An important design consideration for the reward function was how to encode this multi-objective task. First, we ensured each of the individual rewards were in the range of $[0, 1]$, which made it easier to weight the individual rewards. Second, we ensured the collision reward always dominated the other rewards: if the robot predicted it was going to collide, all of the individual rewards were assigned to their maximum value of 1; conversely, if the robot predicted it was not going to collide, all of the individual rewards were assigned to their respective values.

We evaluated BADGR against two other methods: a hand-designed navigation strategy that uses LIDAR to detect obstacles, and a naïve policy that simply drives straight towards the specified goal. We compared against the LIDAR-based strategy, which is a common geometric-based approach for designing navigation policies, in order to demonstrate the advantages of our learning-based approach, while the purpose of the naïve policy is to provide a lower bound baseline and calibrate the difficulty of the task. To our knowledge, there are no other end-to-end learning-based methods that we could compare to that do not require either access to a simulator, a map of the environment,

[1]`https://sites.google.com/view/badgr`



Method	Successfully Reached Goal
BADGR (ours)	**23/25 (92%)**
LIDAR	15/25 (60%)
Naïve	15/25 (60%)

Figure 4.6: Experimental evaluation in an off-road environment for the task of reaching a specified goal location while avoiding collisions. Each approach was evaluated from 5 different start locations—each color corresponding to a different start location—with 5 runs per each start location. Each run terminated when the robot collided, failed to make progress and was trapped, or successfully reached the goal. Our BADGR policy is the only approach which can consistently reach the goal without colliding or getting trapped.

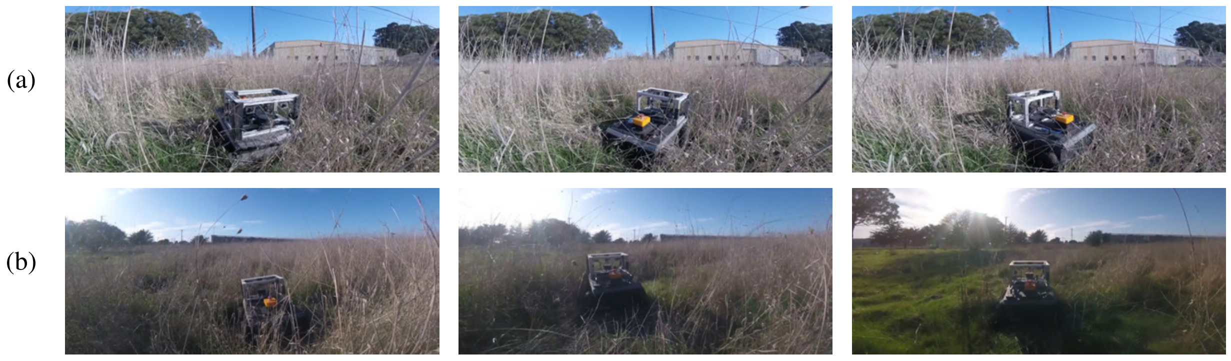


Figure 4.7: Comparison of LIDAR (top) versus our BADGR approach (bottom) in a tall grass portion of the off-road environment. The LIDAR policy incorrectly labels the grass as untraversable, and therefore rotates in-place in an attempt to find a traversable path; after completing a full rotation and failing to detect any traversable path, the LIDAR policy determines the robot is trapped. In contrast, our BADGR approach has learned from experience that some tall grass is indeed traversable, and is therefore able to successfully navigate the robot towards the goal.

demonstration data, or hand-labelled data. We therefore include comparisons to these hand-designed navigation strategies to provide a point of reference for our results.

Note that for all tasks, only a single GPS coordinate—the location of the goal—is given to the robot. However, this GPS coordinate alone does not tell the robot the locations of obstacles or impassable terrain, and therefore a successful navigation strategy must use the onboard sensors.

Off-road environment. We first evaluated all the approaches for the task of reaching a goal location while avoiding collisions and avoiding getting stuck in an off-road environment. Fig. 4.6 shows the resulting paths that BADGR, LIDAR, and the naïve policies followed. The naïve policy sometimes succeeded, but oftentimes collided with obstacles such as trees and became stuck on thick patches of grass. The LIDAR policy nearly never crashed or became stuck on grass, but sometimes refused to move because it was surrounded by grass which it incorrectly labelled as

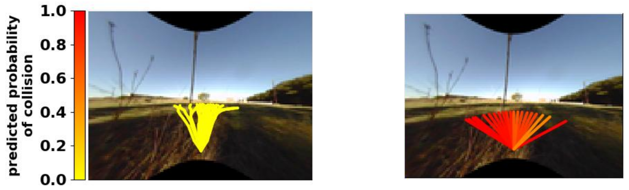
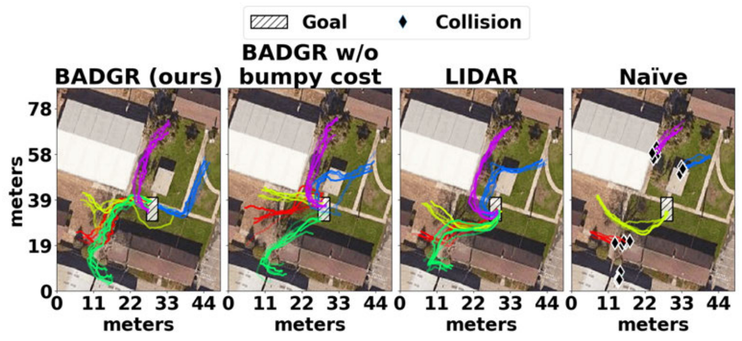


Figure 4.8: Comparison of our BADGR policy (left) versus the LIDAR policy (right). Each image shows the candidate paths each policy considered during planning, and the color of each path indicates if the policy predicts the path will result in a collision. The LIDAR policy falsely predicts the paths driving left or straight will result in a collision with the few strands of tall grass. In contrast, our BADGR policy correctly predicts that the grass is traversable and will therefore drive over the grass, which will result in BADGR reaching the goal $1.5\times$ faster.

untraversable obstacles (Fig. 4.7a). BADGR almost always succeeded in reaching the goal by avoiding collisions and getting stuck, while not falsely predicting that all grass was an obstacle (Fig. 4.7b).

Additionally, even when the LIDAR-based strategy succeeded in reaching the goal, the path it took was sometimes suboptimal. Fig. 4.8 shows an example where the LIDAR-based model labelled a few strands of grass as untraversable obstacles, and therefore decided to take a roundabout path to the goal; in contrast, BADGR accurately predicted these few strands of grass were traversable, and therefore took a more optimal path. BADGR reached the goal $1.2\times$ faster on average compared to the LIDAR policy.

Urban environment. Next, we evaluated all the approaches for the task of reaching a goal GPS location while avoiding collisions and bumpy terrain in an urban environment. Fig. 4.9 shows the resulting paths that BADGR, LIDAR, and the naïve strategies followed. The naïve policy almost always crashed, which illustrates the urban environment contains many obstacles. The LIDAR policy always succeeded in reaching the goal, but failed to avoid the bumpy grass terrain. BADGR



Method	Successfully Reached Goal	Average Bumpiness (|rad/s|)
BADGR (ours)	**25/25 (100%)**	**8.7 ± 4.4**
BADGR w/o bumpy cost	25/25 (100%)	15.0 ± 3.4
LIDAR	25/25 (100%)	13.3 ± 2.9
Naïve	5/25 (20%)	N/A

Figure 4.9: Experimental evaluation in an urban environment for the task of reaching a specified goal position while avoiding collisions and bumpy terrain. Each approach was evaluated from 5 different start locations—each color corresponding to a different start location—with 5 runs per each start location. The figures show the paths of each run, and whether the run successfully reached the goal or ended in a collision. The table shows the success rate and average bumpiness for each method. Our BADGR approach is better able to reach the goal and avoid bumpy terrain compared to the other methods.

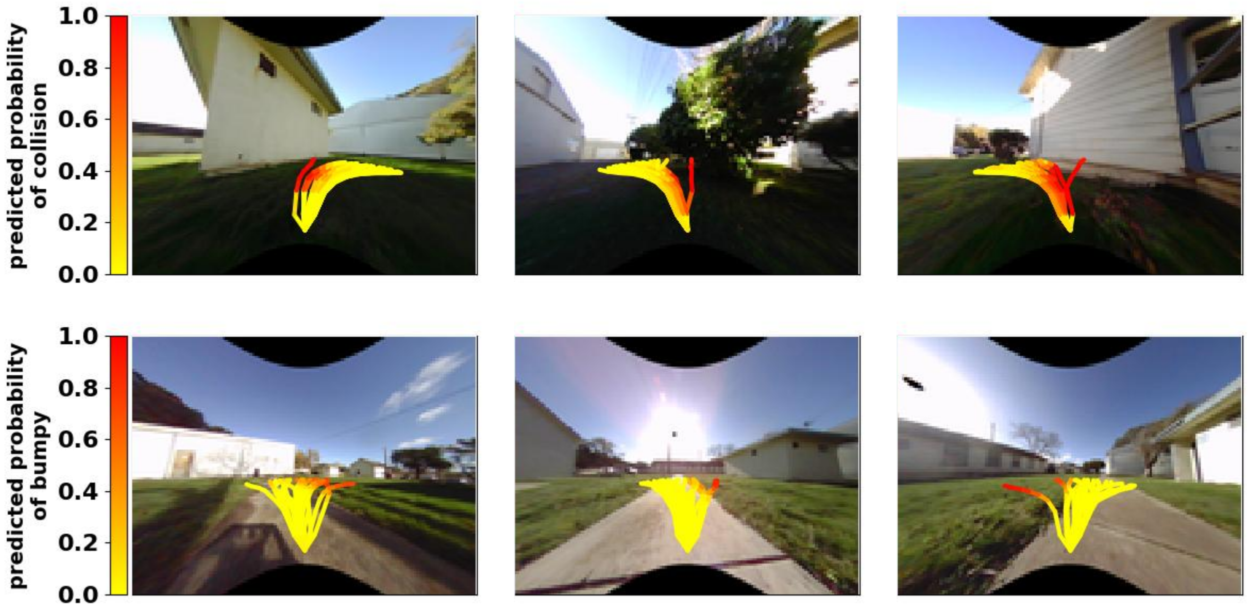

predicted probability of collision
1.0
0.8
0.6
0.4
0.2
0.0
predicted probability of bumpy
1.0
0.8
0.6
0.4
0.2
0.0

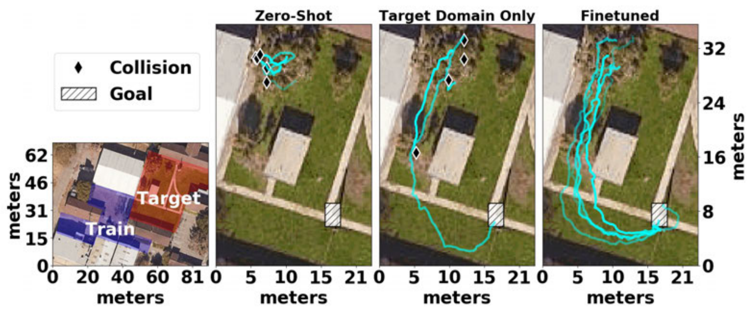
Figure 4.10: Visualization of BADGR's predictive model in the urban environment. Each image shows the candidate paths that BADGR considers during planning. These paths are color coded according to either their probability of collision (top row) or probability of experiencing bumpy terrain (bottom row) according to BADGR's learned predictive model. These visualizations show the learned model can accurately predict that action sequences which would drive into buildings or bushes will result in a collision, and that action sequences which drive on concrete paths are smoother than driving on grass.

also always succeeded in reaching the goal, and—as also shown by Fig. 4.9—succeeded in avoiding bumpy terrain by driving on the paved paths. Note that we never told the robot to drive on paths; BADGR automatically learned from the onboard camera images that driving on concrete paths is smoother than driving on the grass.

While a sufficiently high-resolution 3D LIDAR could in principle identify the bumpiness of the terrain and detect the paved paths automatically, 3D geometry is not a perfect indicator of the terrain properties. For example, let us compare tall grass versus gravel terrain. Geometrically, the tall grass is bumpier than the gravel, but when actually driving over these terrains, the tall grass will result in a smoother ride. This example underscores the idea that there is not a clear mapping between geometry and physically salient properties such as whether terrain is smooth or bumpy.

BADGR overcomes this limitation by directly learning about physically salient properties of the environment using the raw onboard observations—in this case, the IMU readings—to determine if the terrain is bumpy. Our approach does not make assumptions about geometry, but rather lets the predictive model learn correlations from the onboard sensors; Fig. 4.10 shows our predictive model successfully learns which image and action sequences lead to collisions and bumpy terrain and which do not.

Self-supervised improvement. A practical deployment of BADGR would be able to continually self-supervise and improve the model as the robot gathers more data. To provide an evaluation of how additional data enables adaptation to new circumstances, we conducted a controlled study—



Method	Zero-shot	Target Domain Only	Finetuned (ours)
Successfully reached goal	0/5	1/5	5/5

Figure 4.11: The robot's task was to reach a goal in the new target area without colliding. The zero-shot policy trained with only the initial training data always failed. The finetuned policy trained only using data from the target domain travelled farther before colliding, but still predominantly failed. The policy trained using both the initial training data and three hours of autonomously gathered, self-supervised-supervised data in the target domain always succeeded in reaching the goal. This result demonstrates that BADGR improves as it gathers more data, and also that previously gathered data from other areas can actually accelerate learning.

using a small subset of the collected data—in which BADGR gathers and trains on data from one area, moves to a new target area, fails at navigating in this area, but then eventually succeeds in the target area after gathering and training on additional data from that area.

In this experiment, we first evaluate the performance of the original model trained only in the initial training domain, labeled as 'zero-shot' in Figure 4.11. The zero-shot policy fails on every trial due to a collision. We then evaluate the performance of a policy that is finetuned after collecting three more hours of data with autonomous self-supervision in the target domain, which we label as 'finetuned.' This model succeeds at reaching the goal on every trial. For completeness, we also evaluate a model trained *only* on the data from the target domain, without using the data from the original training domain, which we label as 'target domain only.' This model is better than the zero-shot model, but still fails much more frequently than the finetuned model that uses both sources of experience.

This experiment not only demonstrates that BADGR can improve as it gathers more data, but also that previously gathered experience can actually accelerate policy learning when BADGR encounters a new environment. From these results, we might reasonably extrapolate that as BADGR gathers data in more and more environments, it should take less and less time to successfully learn to navigate in each new environment; we hope that future work will evaluate these truly continual and lifelong learning capabilities.

Generalization. We also evaluated how well BADGR—when trained on the full 42 hours of collected data—navigates in novel environments not seen in the training data. Fig. 4.12 shows our BADGR policy successfully navigating 230 meters total in three novel environments, ranging from a forest to urban buildings. This result demonstrates that BADGR can generalize to novel environments if it gathers and trains on a sufficiently large and diverse dataset.

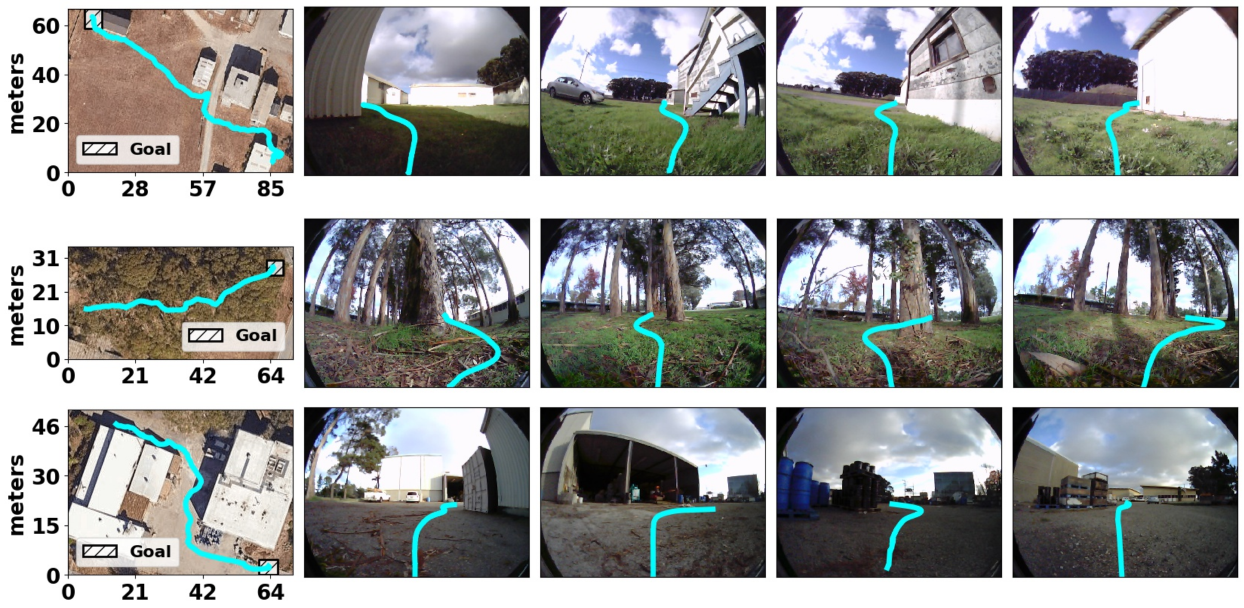


Figure 4.12: Our BADGR policy can generalize to novel environments not seen in the training data. Each row shows the BADGR policy executing in a different novel environment. The first column shows the approximate path followed by the BADGR policy. The remaining columns show sampled images from the onboard camera while the robot is navigating, with the future path of the robot overlaid onto the image.

4.4 Discussion

We presented BADGR, an end-to-end learning-based mobile robot navigation system that can be trained entirely with self-supervised, off-policy data gathered in real-world environments, without any simulation or human supervision, and can improve as it gathers more data. We demonstrated that our approach can learn to navigate in real-world environments with geometrically distracting obstacles, such as tall grass, and can readily incorporate terrain preferences, such as avoiding bumpy terrain, using 42 hours of autonomously collected and self-supervised training data. Our experiments show that BADGR can outperform a LIDAR-based strategy in complex real-world settings, generalize to novel environments, and can improve as it gathers more data. Our experiments suggest that self-supervised end-to-end learning methods can provide a promising alternative to hand-designed, geometric navigation strategies, due to their ability to discover navigational affordances (e.g., traversable or impassable grass) and improve from data.

Chapter 5

Model Supervision

In the previous chapter, we showed how a mobile robot can learn to navigate using self-supervised training signals. However, certain navigation cues, such as learning to drive on the correct side of the road, are difficult to learn through pure self-supervision because there is no off-the-shelf sensor that produces the required training signal.

In this chapter, we propose a generalization of the reinforcement learning framework that combines flexible multi-task learning, off-policy training, and the ability to learn directly from real-world events that can be detected automatically, for example with modern computer vision systems. The main idea behind our approach is to "back up" a set of event cues such that future values of those cues can be predicted based on the current observation (e.g., camera image) and actions. At test time, these predictors can be recombined to achieve user-specified goals. The user can flexibly specify the desired combination of events, such as what speed to move and which lane to drive in, and the robot evaluates candidate actions to determine which actions will maximize the likelihood of those events occurring in the future. We call our models composable action-conditioned predictors (CAPs). From the standpoint of reinforcement learning, our method generalizes value function methods to predict multiple events, and then uses these event predictors to flexibly solve various tasks at test time. From the standpoint of robotic perception, our method can be viewed as a way to turn a vision system into a prediction system: the output of a vision system that *detects* an event can be used to train a model that *predicts* that event, allowing the robot to plan how to cause (or avoid) this event.

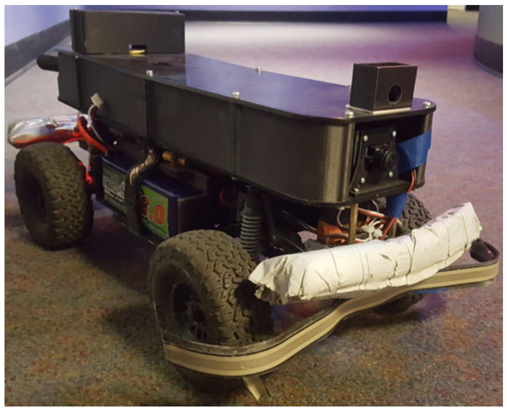


Figure 5.1: Our method can learn flexible mobility skills off-policy for a real-world RC car.

The main contribution of this chapter is CAPs, a general framework for multi-objective learning-based control that scales to large datasets, deep neural network function approximators, and off-policy training. We demonstrate that this framework can be used to train a robot to accomplish a variety of user-specified goals at test time. The individual event cues we consider include those that can be trivially labeled by the robot itself—such as collisions, speed, and heading—as well as event cues labelled by a learned detection system, such as road lanes and doorways. The framework is general and can accommodate any event cue. Our experimental evaluation consists of training these predictors for both a simulated car and real-world RC car (Fig. 5.1) using data collected entirely autonomously, without any human-provided labels beyond those needed to train the object detector. At test-time, we illustrate behaviors such as collision avoidance; path, road, and heading following; and speed control.

5.1 Related Work

Our composable reinforcement learning framework makes a conceptual contribution in how computer vision systems can be used within a learning-based control framework: we propose that, for commonly attainable visual recognition signals (such as object detections), reinforcement learning can effectively convert these *detections* into *predictions* by using them in place of reward signals in a multi-event prediction framework. Prior work has incorporated detectors into reinforcement learning-based settings [67], but for training a model-free policy that is only designed to generalize to new instances of the same event cues. Prediction has been an active area of research in computer vision, including prediction of future images [114]–[116], motion [117], and even physical events [96], [118], [119]. We study action-conditioned prediction, which enables us to control a robot to bring about desired combinations of predicted events at test-time.

Much of the prior work on learning-based robot navigation has focused on imitation learning [16], [20], [62]–[64], which requires demonstrations. Our method does not assume access to demonstrations, and can learn from off-policy data. Supervised learning for navigation using off-policy data has been investigated, including learning drivable routes [17] and near-to-far obstacle detectors [18]. Our approach is similar in that we also predict future events, but prior works typically rely on a hand-engineered control policy, while our control policy using CAPs can improve with more data because it is conditioned on the robot's intended actions. RL approaches [82] are designed to learn and improve control policies from data, including from off-policy data [95]. Much of the reinforcement learning work for robot navigation has focused on collision avoidance [21], [96], [120], while our work addresses goal-directed, multi-objective navigation. RL-based approaches for learning goal-directed navigation have been proposed [39], [121], but typically learn using hand-crafted reward signals that are difficult to calculate in the real-world. In contrast, our approach directly addresses the issue of defining the reward signals in the real-world by turning deployable vision-based detectors into predictors using our CAPs.

5.2 Composable Action-Conditioned Predictors

We now present our composable action-conditioned predictors (CAPs) framework. CAPs learns to "back up" a set of event cues into the past using a predictive model. This model takes as input the state and a sequence of planned future actions, and outputs predictions of these event cues, which can consist of anything relevant to the robot's task, such as collision, speed, and road lane positions.

Using this model, the user can then specify the reward function they wish the robot to maximize in terms of the event cues.

The CAPs Model

Formally, CAPs is an instantiation of the generalized computation graph from Chapter 2, corresponding to a model $f_\theta(\mathbf{s}_t, \mathbf{A}_t^H) \rightarrow \hat{E}_t^{(H,I)}$, which is a function parameterized by parameters θ that maps the state $\mathbf{s}_t$ at time t and a sequence of H intended actions $\mathbf{A}_t^H = (\mathbf{a}_t, ..., \mathbf{a}_{t+H-1})$ to predicted future event cues $\hat{E}_t^{(H,I)}$. The event cues $\hat{E}_t^{(H,I)}$ consist of events $\hat{e}_{t+h}^{(i)}$, where $h \in \{0, ..., H-1\}$ indexes the prediction time length and $i \in \{0, ..., I-1\}$ indexes the ith event cue. This model can be viewed as an extension of [96] to multiple event cues. The model is trained on a dataset of state-action-event tuples $\mathcal{D} := \{\mathbf{s}_t, \mathbf{A}_t^H, E_t^{(H,I)}\}$ such that

$$\theta^* = \arg\min_\theta \sum_{(\mathbf{s}_t, \mathbf{A}_t^H, E_t^{(H,I)}) \in \mathcal{D}} \sum_h \sum_i \|\hat{e}_{t+h}^{(i)} - e_{t+h}^{(i)}\|. \tag{5.1}$$

An important distinction between our approach and that of [66] is that our model is conditioned on a sequence of actions, which is critical for off-policy training. [66] predicts events far in the future conditioned on the single current action, which implicitly makes the predictor conditional on a policy, necessitating on-policy data collection. Training the CAPs is completely off-policy: all data collected by the robot can be used for training, which is advantageous for real-world robot learning in which gathering data is laborious and expensive.

Autonomous Labeling of Event Cues

Although the training dataset could be generated by letting the robot act in the environment and hand-labeling the event cues $E_t^{(H,I)}$, the amount of human supervision needed would become prohibitively expensive. We instead opt for automated labeling by leveraging existing detection systems to label these event cues. These detection systems, including modern computer vision systems, enable our approach to predict cues about the environment that would have otherwise remained unknown. Consequently, by having access to these labeled event cues and training our CAPs to predict these event cues, CAPs can be used to achieve tasks that would have otherwise required *a priori* knowledge of the environment. In short, the detection system provides "what" the robot wants to learn about by labelling the event cues, while CAPs provide "how" the robot can take actions to achieve these events. We discuss specific event cues and their respective learned detection systems used for labeling in the context of autonomous robot navigation in Section 5.3.

In addition to leveraging existing detection systems, we also can label a subset of the cues using self-supervision. For example, if we want the robot to move at a desired speed, and its state includes speed, it can self-label. Although few event cues can be self-supervised, these self-supervised signals are robust.

Action Selection

Using the trained CAPs model f_θ and following the approach of [96], the user can encode a task that the robot must accomplish by defining a reward function $R(\hat{E}_t^{(H,I)})$ on the predicted future event cues. The robot can then calculate the action sequence that maximizes this reward function by solving the following optimization:

$$\mathbf{A}_t^* = \arg\max_{\mathbf{A}^H} R(\hat{E}_t^{(H,I)}) : \hat{E}_t^{(H,I)} = f_\theta(\mathbf{s}_t, \mathbf{A}^H). \tag{5.2}$$

We use stochastic optimization based on the cross entropy method [122] to solve this maximization, although in principle any optimizer could be used.

Although the procedure in Eqn. 5.2 enables the robot to plan H time steps into the future, we would like our robot to be able to accomplish tasks beyond this horizon. We therefore adopt a model predictive control (MPC) approach, in which the robot solves for the optimal action sequence at each time step using Eqn. 5.2, executes the first action, proceeds to the next state, and repeats the planning procedure. Although this MPC approach is not equivalent to planning for the entire horizon of the task, it has been shown to be an effective and robust method for robot control [123].

Off-Policy Learning and Policy Deployment

We now describe the full off-policy learning algorithm using CAPs. A dataset of state-action pairs $(\mathbf{s}_t, \mathbf{a}_t)$ is gathered by having the robot act in the environment according to some policy, such as a random exploration policy or the latest trained CAPs policy (Eqn. 5.2). The robot does not need data gathered for the tasks it will need to accomplish at test time, but it does require data in which the relevant event cues are present. The dataset is then used by the detection systems to label the event cues $(e_t^{(0)}, ..., e_t^{(I-1)})$, which are then added back into the dataset. Finally, the CAPs model is trained using this dataset (Eqn. 5.1). After running off-policy learning, the user can encode a desired task for the robot to accomplish (Sec. 5.2). Importantly, the user can change the task without having to re-train CAPs, so long as the task is expressed in terms of the event cues that CAPs has learned to predict.

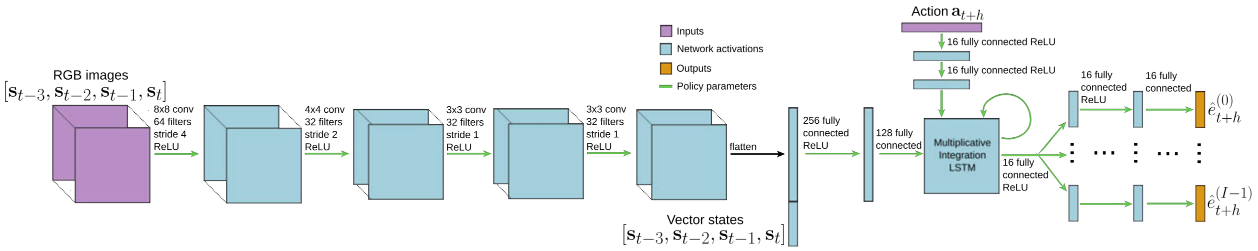

Action $\mathbf{a}_{t+h}$
Inputs
Network activations
Outputs
Policy parameters
16 fully connected ReLU
16 fully connected ReLU
RGB images $[\mathbf{s}_{t-3}, \mathbf{s}_{t-2}, \mathbf{s}_{t-1}, \mathbf{s}_t]$
8x8 conv 64 filters stride 4 ReLU
4x4 conv 32 filters stride 2 ReLU
3x3 conv 32 filters stride 1 ReLU
3x3 conv 32 filters stride 1 ReLU
flatten
256 fully connected ReLU
128 fully connected
Multiplicative Integration LSTM
16 fully connected ReLU
16 fully connected ReLU
16 fully connected
$\hat{e}_{t+h}^{(0)}$
$\hat{e}_{t+h}^{(I-1)}$
Vector states $[\mathbf{s}_{t-3}, \mathbf{s}_{t-2}, \mathbf{s}_{t-1}, \mathbf{s}_t]$

Figure 5.2: The composable action-conditioned predictions (CAPs) network architecture. The network first processes the past four RGB images with convolutional layers, concatenates the result with the past four vector states, and then passes the concatenation through fully connected layers to form the initial hidden state of the recurrent neural newtork (RNN). The RNN, which is a multiplicative integration LSTM [86], sequentially processes each action, and the resulting hidden layer is used to predict each event cue.

5.3 CAPs for Robot Navigation

We now instantiate our CAPs algorithm for robot navigation. Robot navigation is an ideal testbed for CAPs because navigation is a multi-objective task in which many of the event cues—such as road lane and object locations—are unknown *a priori*. To instantiate CAPs, we must instantiate the CAPs model (Sec. 5.2), event cue labellers (Sec. 5.2), and the task reward function (Sec. 5.2).

The CAPs model. We instantiate the CAPs model by defining its inputs, outputs, and parameterization. The inputs to the CAPs model are state observations $\mathbf{s}$ and actions $\mathbf{a}$. The state observations include all readings from sensors on board the robot, such as cameras, wheel encoders, collision bumpers, and inertial measurements. For a ground robot, the actions consist of the desired steering angle and speed. The outputs of the predictive model consist of event cues relevant to goal-oriented navigation, such as road lanes, collisions, speeds, and headings.

We parameterize the model using a deep neural network in order to make it feasible to input high-dimensional observations, such as images. This model is depicted in Fig. 5.2. The network first passes the input images through convolutional layers, concatenating the result with the input state vectors, and processes the concatenation through additional fully connected layers. The final layer serves as the initial hidden state for a recurrent network, which sequentially processes each planned action. The event cues at each time step are then predicted by processing the RNN hidden state with separate fully connected layers.

Event cue labeling. For the vision-based event cues, such as road lanes and doorways, we use modern computer vision models. Specifically, we train FCNs [124] to segment relevant event cues from the onboard camera images. The data used to train these models is a subset of the data used to train the CAPs model, and was either labeled by the simulator for simulation experiments, or by a human for real-world experiments. The non-vision-based event cues are simple functions of the robot state, such as if the robot collided or not, and are extracted automatically.

Task reward function. We encode desired tasks as linear combinations of reward functions of each individual event cue: $R(\hat{E}_t^{(H,I)}) = \sum_{t'=t}^{t+H-1} \sum_i \alpha^{(i)} \cdot R^{(i)}(\hat{e}_{t'}^{(i)})$, where $\alpha^{(i)}$ indicates the relative importance of each reward function. For example, the collision reward function will be highly weighted to ensure the robot does not collide. With this formulation, the task we want the robot to accomplish can be specified and altered by setting the weights $\alpha^{(i)}$.

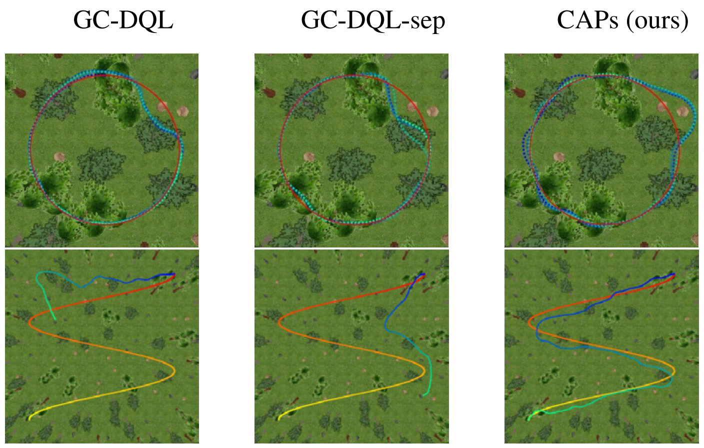
5.4 Experiments

We now present results evaluating our approach on simulated and real-world ground robot navigation tasks. In our evaluation, we aim to answer the following questions:

Q1 Does our event cue prediction approach enable flexible behavior at test time?

Q2 Is our approach able to learn from off-policy data?

Q3 Are we able to learn event predictors using learned detection models, such as modern computer vision systems?



	Avg % Success	Avg Dist. From Path
GC-DQL	0.0%	5.01 ± 3.76
GC-DQL-sep	8.3%	6.03 ± 3.35
CAPs (ours)	**75.0%**	**2.02** ± **1.60**

(a) Evaluation on novel sinusoidal paths.

Figure 5.3: Comparison of our CAPs approach against goal-conditioned deep Q-learning (GC-DQL) and separated goal-conditioned deep Q-learning (GC-DQL-sep) on the task of path following. Each approach was trained via reinforcement learning to following a circular path and avoid collisions (top row images), which shows a birds-eye view of the desired trajectory in red hues and actual trajectory in blue hues. Each approach was then evaluated on a sinusoidal path (bottom row images). Although all approaches successfully learned on the training task, only our approach (Table 5.3a) was able to generalize to a different task at test-time.

In evaluating CAPs, we compare with prior methods that can scale to image observations, learn from off-policy data, accomplish various tasks at test time, and are sample-efficient. We therefore compare with goal-conditioned deep Q-learning (GC-DQL) [65], [125], [126], in which a neural network Q-function is learned that also takes as input the desired goal. We also compared with goal-conditioned deep Q-learning in which each subreward is learned separately (GC-DQL-sep), which is similar to [127], but without entropy maximization. We do not compare with model-based methods because they either assume access to the ground-truth robot and environment state, or require large amounts of data to learn the dynamics of raw image observations.

We consider three experiments: a simulated path following task in a forest, a simulated goal-directed navigation task in a city environment, and a real-world indoor navigation task with an RC car. For each task, in order to instantiate CAPs, we must define which event cues $\hat{e}^{(i)}$ the model will be predicting, how the event cue labels $e^{(i)}$ will be generated, and what is the task reward function $R(\hat{E}_t^{(H,I)})$. Code, videos, and additional details are available at `github.com/gkahn13/CAPs`.

Simulated forest. The first task is a path following task in a forest-like environment (Fig. 5.3) built on the Bullet physics engine [128] with Panda3d [88] for rendering. The goal is to stay close to the path while avoiding collisions.

We define two event cues: the probability of collision and heading. The labels come from self-supervision, because collision and heading can be labelled directly from the robot's sensor observations. The reward function at test-time is:

$$R(\hat{E}_t^{(H,I)}) = \sum_{t'=t}^{t+H-1} 500 \cdot (1 - \hat{e}_{t'}^{(coll)}) + (\cos(\hat{e}_{t'}^{(heading)} - \text{GOAL\_HEADING}) - 1).$$

In order to gather data, we let our approach and the prior works each run RL on the task of following a circular path. Fig. 5.3 shows that all approaches successfully learn to follow the circular path without colliding. We chose to gather data in this way because Q-learning approaches, while technically off-policy, are still sensitive to the data distributions when using neural networks. We therefore wanted to give the prior methods the best chance of success by giving them access to data that was partially on-policy.

After training, we evaluated each approach on following sinusoidal paths in different parts of the forest environment. Note that these paths never occurred in the training data. Fig. 5.3 shows qualitative results for each method, while Table 5.3a provides a quantitative comparison. Our approach avoids collisions and closely follows the sinusoidal path, only deviating when obstacles are on the path itself, while the prior methods crash often and do not closely follow the path. This result shows that even though the prior methods were successful on the circular path they trained on during reinforcement learning, they fail to generalize at test time to different tasks. In contrast, CAPs can flexibly accomplish different tasks at test time (**Q1**).

Simulated city. The second task we evaluate our approach on is goal-directed navigation in a city environment using CARLA [129], a driving simulator with realistic renderings and physics. The objective is to reach the goal while avoiding collisions, driving at a desired speed, and staying in the lane. We train CAPs with five event cues: probability of a collision, heading, speed, is the right lane visible, and the pixel distance of the center of the right lane to the center of the camera image. The labels for the collision, heading, and speed event cues come from self-supervision because these values come directly from the robot's sensor observations. However, we use a learned computer vision system—specifically, a segmentation model [124]—to label the lane event cues; Fig. 5.4c shows an example input image, the ground truth segmentation label, and the model's predicted segmentation and resulting prediction of the center of the right lane. The reward function at test-time is:

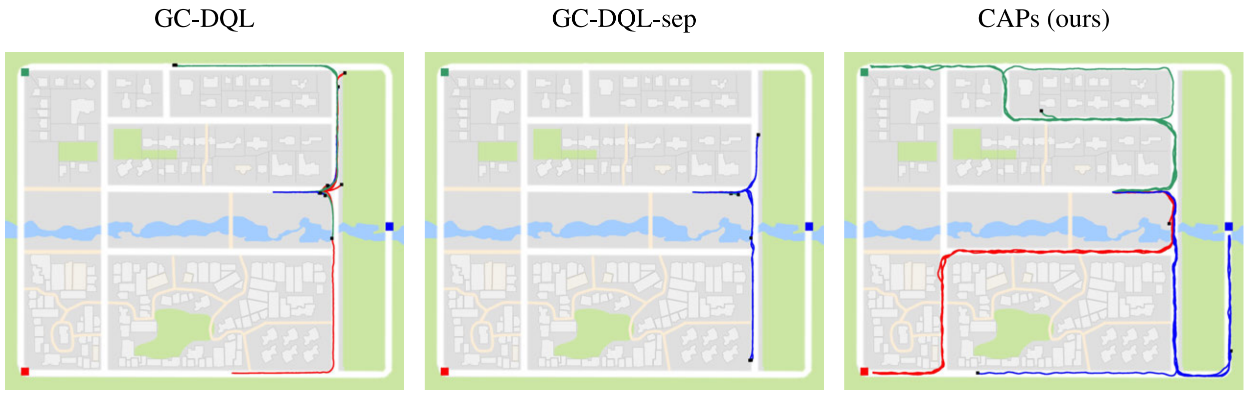
$$R(\hat{E}_t^{(H,I)}) = \sum_{t'=t}^{t+H-1} 50 \cdot (1 - \hat{e}_{t'}^{(coll)}) - 3 \cdot \frac{|\hat{e}_{t'}^{(speed)} - \text{GOAL\_SPEED}|}{\text{GOAL\_SPEED}} + 5 \cdot \hat{e}_{t'}^{(lane\_seen)}(1 - |\hat{e}_{t'}^{(lane\_diff)}|)$$
$$- \frac{5}{\pi} \cdot |\hat{e}_{t'}^{(heading)} - \text{GOAL\_HEADING}| - 0.15 \cdot \|\mathbf{a}_{t'}^{(steer)}\|_2^2.$$

For training data, we ran DQL for the task of collision avoidance, resulting in 800,000 points (2.3 days worth) of data. We chose to gather data using DQL to give the prior Q-learning-based methods the best chance at success, and to demonstrate that CAPs can learn from off-policy data (**Q2**). We then trained all methods on this data for the combined task of collision avoidance, driving at a desired speed, and staying in the right lane.

We evaluated all approaches on driving at 7 m/s and trying to reach one of three destination locations. Fig. 5.4a shows a birds-eye view of the city map and the resulting trajectories of each method's policy, while Table 5.4b provides the corresponding quantitative results. Our CAPs policy is able to successfully reach each goal the majority of the time while driving in the right lane at the desired speed, while the prior methods are never able to reach the goal.

The key aspect of CAPs that enables its success is its flexibility at test time (**Q1**): we can train the CAPs model once using off-policy data, and then define reward function at test time to achieve the desired robot behavior for the considered task. Although defining this reward function is non-trivial, it is significantly easier and less time consuming than designing the reward function for standard RL algorithms. For example, we spent only two hours tuning the CAPs reward function for this task, while running GC-DQL or GC-DQL-sep just once takes 12 hours; considering that we tried dozens of reward functions during the two hours of tuning with CAPs, attempting to do this amount of tuning with a standard RL algorithm could take days or weeks, and possibly still not result in a successful policy.

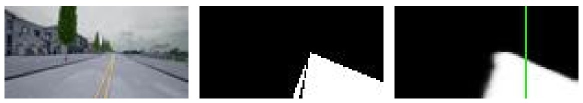
Another key aspect of our approach is using learned detection models to autonomously label the event cues (**Q3**). Although using a learned segmentation model (Fig. 5.4c) in simulation is not strictly necessary because we have access to the ground truth labels, learned event cue labellers will be crucial in real-world experiments in which ground truth labels from humans is prohibitively expensive to obtain. We further demonstrate the importance of autonomous labelling in the following real-world experiments.



(a) Birds-eye view for each approach of the resulting trajectories when attempting to go to either the red, green, or blue goal location (five attempts per goal location). Our CAPs approach is able to reach the goal the majority of the time, while the prior methods crash before reaching the goal.

	GC-DQL	GC-DQL-sep	CAPs (ours)
% did not crash	7	0	**73**
% reached goal	0	0	**67**
speed (m/s)	3.6 (1.3)	5.6 (0.2)	**6.7 (0.2)**
% in right lane	98 (3)	**100 (6)**	74 (8)

(b) Corresponding quantitative results for (a).



(c) To enable our CAPs approach to learn to stay in the right lane, we trained a model to take as input the onboard RGB image (left) and predict where the road is (right) using labels (center). Once trained the learned segmentation model autonomously labels where the center of the right lane is (green vertical line), the CAPs model can learn to keep the center of the right lane in the middle of its camera view.

Figure 5.4: Comparison of our CAPs approach against prior methods on the task of reaching a target goal position while avoiding collisions, driving at 7m/s, and staying in the right lane in the CARLA simulator [129].

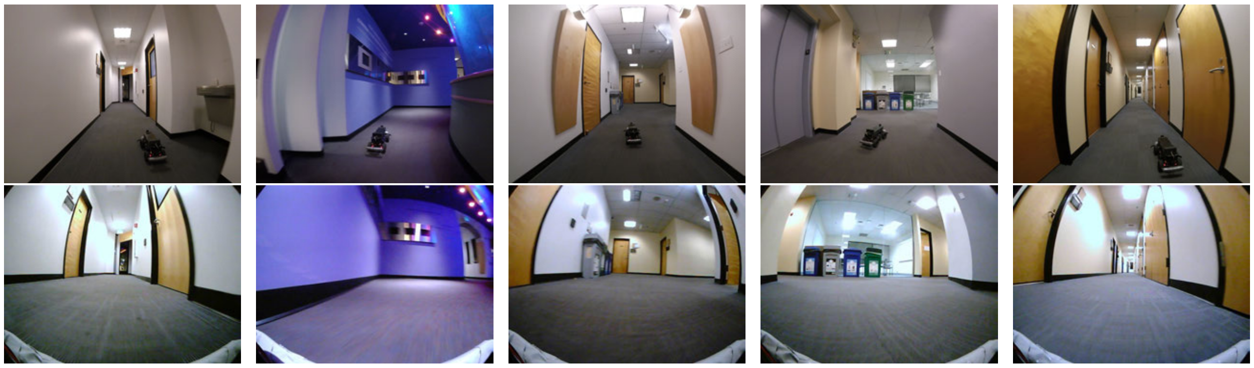


Figure 5.5: Third-person images (top row) and first-person images from the onboard camera (bottom row) from the RC car navigating through the 5th floor of Cory Hall at UC Berkeley for the task of avoiding collisions, following desired goal headings, and going towards doors using our CAPs approach.

Real-world indoor navigation. The third task we evaluate our approach on is a real-world goal-directed indoor navigation task with an RC car (Fig. 5.1). The task objective is to avoid collisions, follow the desired goal headings, and go near doors, which simulates a package delivery application. We therefore define three event cues: probability of a collision, heading, and the fraction of the image that is a door. The labels for the collision and heading come directly from the robot's sensor observations, while we use a learned computer vision segmentation model to label the door fraction. We trained the segmentation model by labelling 345 (0.2%) of the images from the RL-gathered dataset. The reward function is:

$$R(\hat{E}_t^{(H,I)}) = \sum_{t'=t}^{t+H-1} (1 - \hat{e}_{t'}^{(coll)}) \cdot \left[1 - \frac{0.1}{\pi} \cdot |\hat{e}_{t'}^{(heading)} - \text{GOAL\_HEADING}| + 0.05 \cdot \hat{e}_{t'}^{(door\_frac)}\right] - 0.01 \cdot \|\mathbf{a}_{t'}\|_2^2.$$

For training data, we ran RL with CAPs on the 5th floor of Cory Hall at UC Berkeley for 11 hours, gathering 158,400 data points. The RL objective for gathering this data was purely collision avoidance, and therefore the data is off-policy (**Q2**). We then evaluated all approaches in this same environment. Although this evaluation does not test our learned policy's ability to generalize to new environments, our approach does not need to rely on policy generalization; instead, we rely on our sample-efficient and flexible real-world CAPs learning algorithm to quickly learn in new environments.

We evaluated our approach on navigating a 75m loop in the environment, which contained five intersections. Two seconds before each intersection, a heading command was given to indicate which way to turn at the intersection. Fig. 5.5 shows images of our approach navigating the loop, while Table 5.1 provides the corresponding quantitative results, with five trials per experiment. Our CAPs policy is able to successfully navigate this loop, while GC-DQL consistently crashes early on. However, DQL is able to successfully perform collision avoidance down a straight hallway (Table 5.1), indicating that its failure at the full task is due to its difficulty in learning from a multi-objective, goal-conditioned reward function.

In addition, we evaluated CAPs without the door subreward to determine if our approach could learn to predict and act on visual event cues. Note that we used the same CAPs model, but simply

Experiment	Method	% of Route Traversed	% Did Not Crash	% Successful Turns	Avg. Door % Seen
Collision	DQL	100 (0)	100	N/A	N/A
	CAPs (ours)	100 (0)	100	N/A	N/A
Collision, Heading	GC-DQL	4.5 (7.5)	0	0	6.9 (1.0)
	CAPs (ours)	**98 (0)**	**80**	**80**	6.9 (0.3)
Collision, Heading, Door	GC-DQL	5.5 (3.1)	0	0	7.0 (2.0)
	CAPs (ours)	**98 (0)**	**80**	**80**	**8.1 (0.2)**

Table 5.1: Comparison of our CAPs approach on a real-world RC car for the task of navigating a 75m loop and going near doors, which simulates a package delivery application. Our approach is able to successfully accomplish the task (bottom row), while the prior method does not succeed and can only accomplish a simple collision-avoidance task (top row). Note that only one model was trained with CAPs, and each experiment merely required changing the reward function at test time; in contrast, the prior method had to be retrained for each task. Additionally, our approach tasked with going towards doors (bottom row), compared to our approach not tasked with going towards doors (middle row), is indeed able to navigate while moving towards doors, showing our approach can learn from visual detection systems.

changed the reward function at test time. Table 5.1 shows that the car sees 17% more doors when the door subreward is included, which is a significant increase considering that the majority of the floor does not contain doors. This evaluation highlights the flexibility (**Q1**) of our approach: using the same CAPs model, we can perform different tasks at test time.

5.5 Discussion

We presented CAPs, a general framework for flexible learning-based control. CAPs predicts event cues, which can be trained from off-policy data and flexibly combined at test time to accomplish various tasks. These event cues are automatically labeled using learned detection models, such as computer vision systems, which enable CAPs to be learned fully autonomously. We demonstrated CAPs on simulated and real-world robot navigation tasks, showing that it was indeed able to learn from these automatically labelled event cues and could flexibly accomplish various tasks at test time.

Chapter 6

Human Supervision

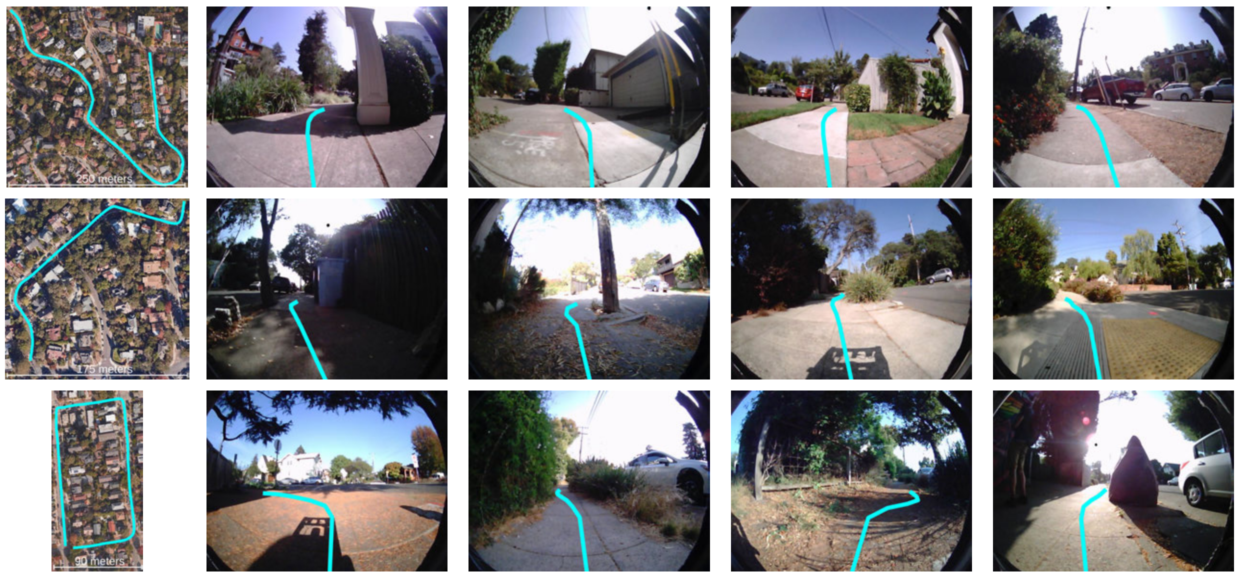


Figure 6.1: LaND is a learning-based approach for autonomous mobile robot navigation that directly learns from disengagements—any time a human monitor disengages the robot's autonomy. These disengagement datasets are ubiquitous because they are naturally collected during the process of testing these autonomous systems. LaND is able to navigate in a diverse set of sidewalk environments, including parked bicycles, dense foliage, parked cars, sun glare, sharp turns, and unexpected obstacles.

In the previous chapter, we developed an algorithm that can learn using signals from learned models. However, training using signals from learned models can be expensive—because these models must be trained in the first place—and fallible—because these models may be inaccurate. In this chapter, we investigate how already-existing signals from humans can enable a robot to learn to navigate.

One of the primary metrics for measuring progress has been the notion of average distance travelled before disengagement: how far can the robot travel before the robot fails and a human must intervene? Although using these disengagement numbers as a metric for comparing progress is contentious [130], the general consensus is clear: the better the autonomous system, the less disengagements. However, we believe a shift in perspective is needed. Not only do these disengagements show where the existing system fails, which is useful for troubleshooting, but also that these disengagements provide a *direct learning signal* by which the robot can learn how to navigate. With this perspective, we believe disengagement data is severely underutilized, and in this work we investigate how to learn to navigate using disengagements as a reinforcement signal.

Disengagements are typically used as a tool to debug and improve autonomous mobile robots: run the robot, discover the failure modes, and then rectify the system such that those failure modes are removed. While the first two steps are fairly similar for most developers, the last step—figuring out how to fix the autonomy failure modes—is highly nontrivial and system dependent. This is especially true for the learning-based modules, which are a key component of modern autonomous mobile robots. Improving these learning-based components is a complex process, which could involve designing neural network architectures, hyperparameter tuning, and data collection and labelling; this is a time-consuming, expensive, and technically challenging endeavor that has to be done largely through trial and error.

In this chapter, we propose a method for learning to navigate from disengagements, or LaND, which sidesteps this laborious process by directly learning from disengagements. Our key insight is that if the robot can successfully learn to execute actions that avoid disengagement, then the robot will successfully perform the desired task. Crucially, unlike conventional reinforcement learning algorithms, which use task-specific reward functions, our approach does not even need to *know* the task – the task is specified implicitly through the disengagement signal. However, similar to standard reinforcement learning algorithms, our approach continuously improves because our learning algorithm reinforces actions that avoid disengagements.

Our approach works by leveraging a ubiquitous dataset: the robot's sensory observations (e.g., camera images), commanded actions (e.g., steering angle), and whether the robot autonomy mode was engaged or disengaged. Using this dataset, we then learn a predictive model that takes as input the current observation and a sequence of future commanded actions, and predicts whether the robot will be engaged or disengaged in the future. At test time, we can then use this predictive model to plan and execute actions that avoid disengagement while navigating towards a desired goal location.

This chapter has three primary contributions. First, we propose that disengagements provide a strong, direct supervision signal for autonomous mobile robots, which is already being collected in many commonly used real-world pipelines. Second, we introduce our learning to navigate from disengagements algorithm, or LaND, which is a simple, effective, and scalable framework for creating autonomous mobile robots. Third, we demonstrate our approach on a real world ground robot in diverse and complex sidewalk navigation environments (Fig. 6.1), and show our method outperforms state-of-the-art imitation learning and reinforcement learning approaches.

6.1 Related Work

Learning-based methods are a promising approach for robot navigation. One common class of these learning-based methods aims to directly learn to predict navigational cues about the environment — such as depth, object detection, and road segmentation—directly from the robot's onboard sensors[16], [50]–[52], [57], [68]. Although these approaches have been successfully demonstrated, they are exceedingly expensive to train due to the labelling cost, and additional labels do not necessarily lead to improved performance. In contrast, our LaND approach has zero additional cost beyond the already-required human safety driver, and our experiments show LaND directly improves as more data is gathered.

Another class of learning-based methods is imitation learning, in which a policy is trained to mimic an expert. These approaches have been successfully demonstrated for both ground robots [20], [56], [62], [69] and aerial robots [55], [70], [71]. However, these demonstrations are typically in visually simplistic environments—such as lane following [20], [56], [71], hallways [71], and race courses [69], require injecting noise into the expert's policy [55], [56], [69]—which can be dangerous, and ignore disengagement data [20], [55], [62], [69], [70]. In contrast, our approach is able to navigate on visually diverse sidewalks, does not require injecting noise during data collection, and can directly leverage disengagement data. Additionally, our experiments demonstrate that our LaND method outperforms behavioral cloning, a standard approach for imitation learning.

In broader terms, learning from disengagements can be viewed as reinforcement learning [131], in which a robot learns from trial and error. One class of reinforcement learning methods for robot learning is sim-to-real, in which a control policy is learned in simulation and then executed in the real world [39], [105], [132]. These sim-to-real approaches are complementary to our LaND approach; the simulation policy can be used to initalize the real-world policy, while our method continues to finetune by learning from disengagements. Other reinforcement learning methods, including ours, learn directly from the robot's experiences [21], [60], [106], [108], [133]–[136]. However, these methods typically assume catastrophic failures are acceptable [60], [136] or access to a safe controller [21], [134], the robot gathers data in a single area over multiple traversals [60], [106], [108], on-policy data collection [108], access to a reward signal beyond disengagement [106], perform their evaluations in the training environment [106], [108], or are only demonstrated in simulation [133], [135]. In contrast, our LaND method is safe because it leverages the existing human-safety driver, learns from off-policy data, does not require retraversing an area multiple times, learns directly from whether the robot is engaged or disengaged, and evaluate in novel, never-before-seen real-world environments. Additionally, we show in our experiments that LaND outperforms [108], a state-of-the-art real world reinforcement learning method for autonomous driving.

6.2 Learning to Navigate from Disengagements

Our goal is to develop an algorithm, which we call LaND, that enables a mobile robot to autonomously navigate in diverse, real-world environments. An overview of LaND is shown in

Fig. 6.2. LaND leverages datasets that are naturally collected while testing autonomous mobile robot systems: the robot's sensor observations—such as camera images, commanded actions—such as the steering angle, and whether the robot autonomy mode was engaged or disengaged. Using this dataset, we then train a convolutional recurrent neural network that takes as input the current observation and a sequence of future commanded actions, and predicts the probability that the robot will be engaged or disengaged in the future. At test time, we can then use this model to plan and execute actions that minimize the probability of disengagement while navigating towards a desired goal location.

LaND has several desirable properties. First, our method does not require any additional data beyond what is already collected from testing the autonomous system with a human safety overseer. Second, LaND learns directly from the disengagement data, as opposed to prior methods that use disengagements as an indirect debugging tool. Third, we make minimal assumptions about the robot—just access to the robot's onboard sensors, commanded actions, and whether the autonomy mode was engaged or disengaged; we do not require access to nontrivial information, such as high-definition maps. And lastly, our method continuously improves as the robot is tested, which we demonstrate in our experiments.

In the following sections, we will describe the data collection process, model training, and planning and control, and conclude with a summarizing overview of LaND.

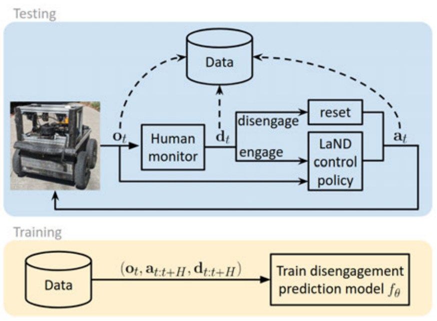


Figure 6.2: System diagram of our LaND algorithm. In the testing phase, the robot produces sensory observations, such as camera images. Based on these observations, the human monitor determines whether to disengage or engage the robot's autonomy system. If the robot is engaged, our LaND control policy determines which action to execute using the current observation; if the robot is disengaged, the human determines which actions the robot should execute in order to reset. This action is then executed by the robot. While testing, the observations, actions, and disengagements are added to a dataset. In the training phase, this dataset is used to train the disengagement prediction model at the core of the LaND control policy. LaND alternates between training and testing until the control policy reaches satisfactory performance.

Data Collection

We start by describing the robot platform used to both collect data and for autonomous navigation. The robot is defined by observation $\mathbf{o}_t$ gathered by its onboard sensors, action $\mathbf{a}_t$ which commands the robot, and a binary signal $\mathbf{d}_t$ indicating if the autonomy mode is disengaged. In our experiments, we use a Clearpath Jackal robot, as shown in Fig. 6.3. The observation $\mathbf{o}$ is a 96×192 RGB image from

Figure 6.3: The mobile robot.

a front-facing 170° field-of-view monocular camera, the action $\mathbf{a}$ is the desired heading change, and the disengagement signal $\mathbf{d}$ is conveyed by a human following nearby with a remote control.

Data collection proceeds by having the robot execute an autonomous control policy, such as the LaND planning-based controller described in Sec. 6.2. A person monitors the robot, and if the robot is in a failure mode or approaching a failure mode, the person disengages autonomy mode. The person then repositions the robot back into a valid state and then re-engages autonomy mode.

Figure 6.4: Person monitoring data collection.

An example of the data collection process for our sidewalk experiment evaluations is shown in Fig. 6.5. The robot proceeds by autonomously driving on the sidewalk. However, if the robot enters one of the three possible failure modes—colliding with an obstacle, driving into the street, or driving into a house's driveway—the person disengages autonomy mode, uses the remote control to reposition the robot onto the sidewalk, and then re-enables autonomy mode.

As the robot collects data, the observations, actions, and disengagements $(\mathbf{o}_t, \mathbf{a}_t, \mathbf{d}_t)$ at each time step t are saved every $\Delta\mathbf{x}$ meters into the dataset $\mathcal{D}$.

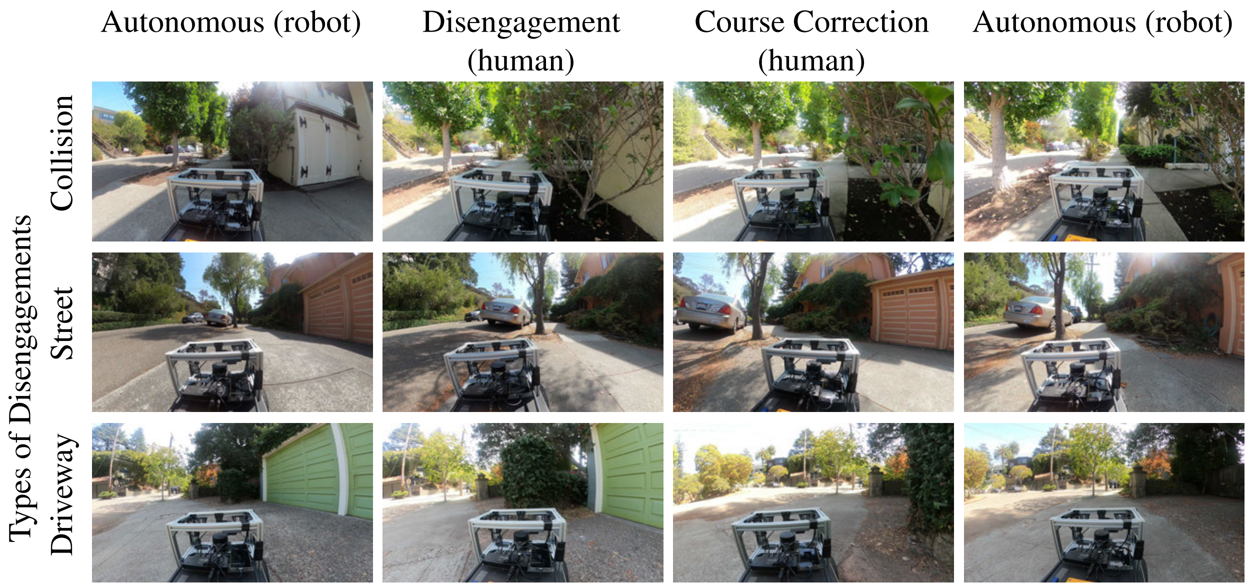


Figure 6.5: Our LaND approach learns to navigate from disengagements. For the sidewalk navigation task studied in our experiments, there are three types of scenarios that will cause the human overseer to disengage the robot's autonomy mode: colliding with an obstacle, driving into the street, and driving into a driveway. After the robot is disengaged, the human repositions the robot onto the sidewalk and then re-engages autonomy.

Predictive Model

The model at the core of LaND is an instantiation of the generalized computation graph from Chapter 2. The learned predictive model takes as input the current observation and a sequence of future actions, and outputs a sequence of future disengagement probabilities. We define this model as

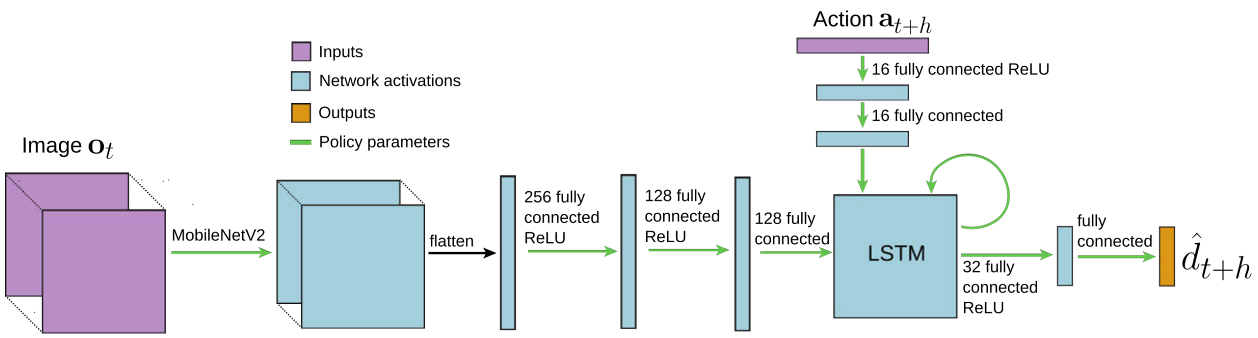


Figure 6.6: Illustration of the image-based, action-conditioned convolutional recurrent deep neural network at the core of LaND. The network first processes the input image observation using the MobileNetV2 [137] convolutional neural network, followed by a series of fully connected layers. The output of these image layers serves as the initial hidden state for an LSTM recurrent neural network [109], which sequentially processes each of the H future actions $\mathbf{a}_{t+h}$ and outputs the corresponding predicted probability of disengagement $\hat{\mathbf{d}}_{t+h}$. When this model is deployed, these predicted disengagement probabilities enable a planner to plan and execute actions that avoid disengagements.

$f_\theta(\mathbf{o}_t, \mathbf{a}_{t:t+H}) \to \hat{\mathbf{d}}_{t:t+H}$, which is a function f parameterized by vector θ that takes as input the current observation $\mathbf{o}_t$ and a sequence of H future actions $\mathbf{a}_{t:t+H} = (\mathbf{a}_t, \mathbf{a}_{t+1}, ..., \mathbf{a}_{t+H-1})$, and outputs a sequence of H future predicted disengagement probabilities $\hat{\mathbf{d}}_{t:t+H} = (\hat{\mathbf{d}}_t, \hat{\mathbf{d}}_{t+1}, ..., \hat{\mathbf{d}}_{t+H-1})$.

We instantiate the model as an image-based, action-conditioned convolutional recurrent neural network, as shown in Fig. 6.6. The network first processes the input image observation using the MobileNetV2 [137] convolutional neural network, followed by a series of fully connected layers. The output of these image layers serves as the initial hidden state for an LSTM recurrent neural network [109], which sequentially processes each of the H future actions $\mathbf{a}_{t+h}$ and outputs the corresponding predicted probability of disengagement $\hat{\mathbf{d}}_{t+h}$.

The model is trained using the collected dataset to minimize the cross entropy loss between the predicted and ground truth disengagement probabilities

$$\mathcal{L}(\theta, \mathcal{D}) = \sum_{(\mathbf{o}_t, \mathbf{a}_{t:t+H}, \mathbf{d}_{t:t+H})} \sum_{h=0}^{H-1} \mathcal{L}^{\text{CE}}(\hat{\mathbf{d}}_{t+h}, \mathbf{d}_{t+h}) \quad : \hat{\mathbf{d}}_{t:t+H} = f_\theta(\mathbf{o}_t, \mathbf{a}_{t:t+H}), \tag{6.1}$$

in which $\mathcal{L}^{\text{CE}}$ is the cross-entropy loss.

The neural network parameters θ are trained by performing minibatch gradient descent on Eqn. 6.1. However, we modify the standard minibatch training procedure in two important ways. First, we ensure that half the minibatch contain sequences ending in a disengagement and the other half contain sequences with no disengagements. This rebalancing ensures that disengagement data—which is often a small proportion of the total dataset—is seen often during training. Second, if the sampled time step t is less than H steps from a disengagement, we still sample a full sequence of actions $\mathbf{a}_{t:t+H}$ and disengagements $\mathbf{d}_{t:t+H}$ of length H by padding the sequence with (a) actions randomly sampled from the dataset and (b) additional disengagement signals. This artificial disengagement extension scheme assumes that once the robot is disengaged, any action it takes will cause the robot to remain in a disengaged state. This extension scheme is important because it ensures that (a) the model is trained on observations close to disengagements and (b) the training procedure matches the planning procedure—which always plans over a sequence of H actions.

Planning and Control

Using the trained neural network disengagement prediction model, the robot can plan and execute actions at test time that avoid disengagements while navigating towards a desired goal location. We encode this planning objective with the following cost function

$$\mathcal{C}(\hat{\mathbf{d}}_{t:t+H}, \mathbf{a}_{t:t+H}) = \sum_{h=0}^{H-1} \hat{\mathbf{d}}_{t+h} + \alpha \cdot \|\mathbf{a}_{t+h} - \mathbf{g}\|_2^2. \tag{6.2}$$

The first term encourages the robot to avoid disengagements, while the second term encourages the robot to navigate towards a desired goal location; the scalar α is a user-defined weighting between these two terms. The goal location is conveyed through the desired heading vector g; for example, in our experiments the action is the steering angle, and the goal heading vector is either to turn left, right, or continue straight when the robot encounters a junction. Note that this goal heading does not tell the robot how to navigate, and only provides high-level guidance at junctures; this level of supervision is similar to smartphone driving directions. In our experiments, because there were no junctures, we set $\alpha = 0$, but we maintain this general formulation to show our approach is goal-conditioned.

Using this cost function, the robot solves solves the following planning problem at each time step

$$\mathbf{a}^*_{t:t+H} = \arg\min_{\mathbf{a}_{t:t+H}} \mathcal{C}(\hat{\mathbf{d}}_{t:t+H}, \mathbf{a}_{t:t+H}) \quad : \quad \hat{\mathbf{d}}_{t:t+H} = f_\theta(\mathbf{o}_t, \mathbf{a}_{t:t+H}), \tag{6.3}$$

executes the first action, and continues to plan and execute following the framework of model predictive control [123]. We use the same optimizer as in Chapter 4, with hyperparameters $N = 8192, \sigma = 1, \beta = 0.5, \gamma = 50$.

Algorithm Summary

We now provide a brief summary of our LaND algorithm (Alg. 4). LaND alternates between two phases: collecting data and training the predictive model.

In the data collection phase, the robot executes actions according to the planning procedure from Eqn. 6.3. A person monitors the robot, and disengages the robot if it enters a failure mode; if the person does disengage the robot, they then reposition the robot and subsequently re-engage autonomous execution. While collecting data, the current observation, action, and disengagement are saved into the training dataset. In the training phase, the collected dataset is used to train the predicted model by minimizing Eqn. 6.3.

Although Alg. 4 uses our LaND control policy to collect data, we note that any control policy can be used to gather data; in fact, in our experiments we used both LaND and the imitation learning

Algorithm 4 Learning to Navigate from Disengagements

```
1:  initialize dataset $\mathcal{D} \leftarrow \emptyset$
2:  randomly initialize learned parameter $\theta$
3:  while not done do
4:      while collecting data do
5:          get current observation $\mathbf{o}_t$ from sensors
6:          solve Eqn. 6.3 using $f_\theta$ and $\mathbf{o}_t$ to get the
            planned action sequence $\mathbf{a}^*_{t:t+H}$
7:          execute the first action $\mathbf{a}^*_t$
8:          get current disengagement $\mathbf{d}_t$
9:          add $(\mathbf{o}_t, \mathbf{a}^*_t, \mathbf{d}_t)$ to $\mathcal{D}$
10:         if $\mathbf{d}_t$ is disengaged then
11:             let human execute reset maneuver and re-engage autonomy
12:         end if
13:     end while
14:     use $\mathcal{D}$ to train predictive model $f_\theta$
        by minimizing Eqn. 6.1
15: end while
```

and reinforcement learning comparison methods to gather data. However, we note that the ideal policy for data collection is the LaND policy because this ensures that the collected disengagements are from the failure modes of the LaND policy.

6.3 Experiments

In our experimental evaluation, we study how LaND can learn to navigate from disengagement data in the context of a sidewalk navigation task, and compare our approach to state-of-the-art imitation learning and reinforcement learning approaches. Videos, code, and other supplemental material are available on our website [1]

Our dataset consists of 17.4 km of sidewalks gathered over 6 hours. Data was saved every $\Delta\mathbf{x} = 0.5$ meters, and therefore the dataset has 34,800 data points, and contains 1,926 disengagements. Although the amount of data gathered may seem significant, (1) this data is already being collected while testing the robot, (2) the robot requires less human disengagements as it gathers more data and trains, and (3) this dataset is significantly smaller than those typically used in computer vision [112] and reinforcement learning [113] algorithms.

We evaluated LaND in comparison to two other methods:

1. *Behavioral cloning*: a common imitation learning approach used by many state-of-the-art navigation methods [20], [56], [71].

[1] https://sites.google.com/view/sidewalk-learning

2. *Kendall et. al. [108]*: a reinforcement learning algorithm which first learns a compressed representation of the training images using a VAE [92], and learns a control policy from this compressed representation using the DDPG reinforcement learning algorithm [25]. Kendall et al. [108] did not provide source code, so we therefore used existing VAE [2] and DDPG [3] implementations.

All methods, including ours, were trained on the same dataset. For behavioral cloning, data within 2 meters of a disengagement was not used for training. For Kendall et. al. [108], the reward was -1 for a disengagement, and 0 otherwise.

We compare against these methods because, to the best of our knowledge, they are representative of state-of-the-art methods in imitation learning and reinforcement learning for such tasks. We note, however, that we were unable to perfectly replicate these algorithms because they contained assumptions that violated our problem statement; for example, many of the algorithms injected exploration noise into the data collection policy. Nevertheless, we believe our evaluation accurately reflects current work in the context of our realistic problem statement and real world experimental evaluation.

[2]`www.tensorflow.org/tutorials/generative/cvae`
[3]`www.github.com/rail-berkeley/d4rl_evaluations`

Method	Avg. distance until disengagement (meters)
Behavioral cloning (e.g., [20], [56], [71])	13.4
Kendall et. al. [108]	2.0
LaND (ours)	**87.5**

Table 6.1: Experimental evaluation on 2.3 km of never-before-seen sidewalks (Fig. 6.1). Our LaND approach is better able to navigate these sidewalks, travelling 6.5× further before disengagement compared to the next best method.

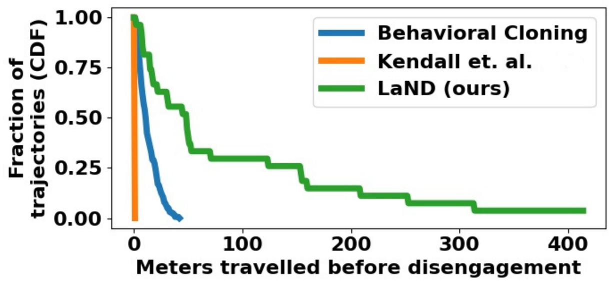


Figure 6.7: Experimental evaluation on 2.3 km of never-before-seen sidewalks (Fig. 6.1). The plot shows the fraction of trajectories—defined as a continuous episode of engaged autonomy—which travelled a certain distance before a disengagement. Methods closer to the top right are better because this indicates a longer distance travelled before disengagement. Our LaND approach is able to travel farther before disengagement: 33% of the trajectories travelled further than 50 meters, including a trajectory of over 400 meters. In contrast, none of the prior methods were able to travel more than 50 meters before disengagement.

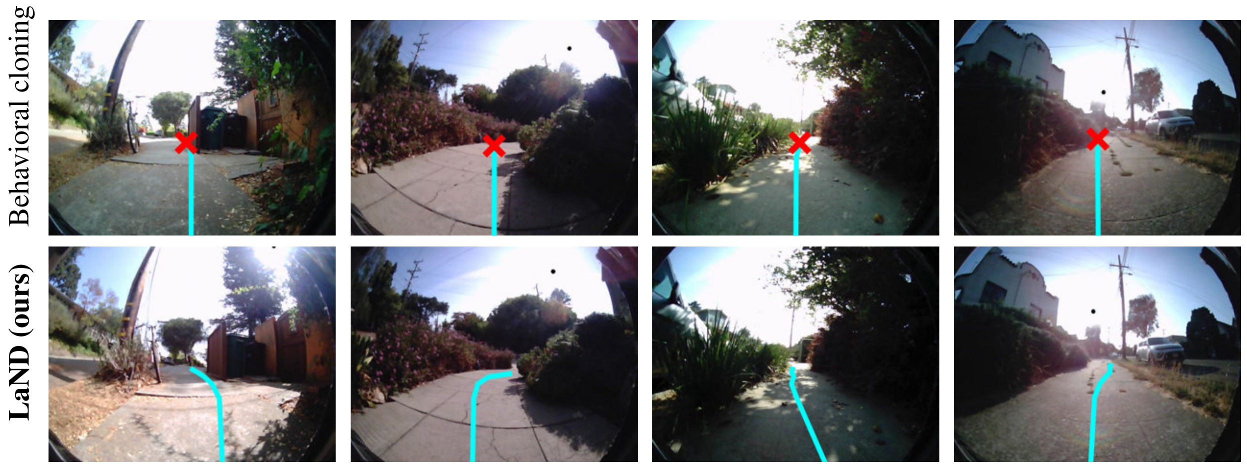


Figure 6.8: Qualitative comparison of our LaND method versus the best performing prior approach (behavioral cloning) in scenarios containing parked bicycles, dense foliage, sun glare, and sharp turns. Our approach successfully navigates these scenarios, while imitation learning is unable to and crashes.

We evaluated all approaches on 2.3 km of sidewalks not present in the training data, as shown in Fig. 6.1. Table 6.1 shows that LaND is better able to navigate sidewalks compared to the other approaches, traveling $6.5\times$ further on average before disengagement compared to the next best approach. Fig. 6.7 shows a more detailed per-trajectory analysis. None of the methods besides LaND were able to travel further than 50 meters before disengagement. LaND was able to travel further than 50 meters before disengagement for 33% of the trajectories, and could sometimes travel up to 400 meters. Fig. 6.8 shows example challenge scenarios—including a parked bicycle, dense foliage, sun glare, and sharp turns—in which LaND successfully navigated, while the best performing comparative approach (imitation learning) failed.

To demonstrate that our approach is able to learn which action sequences lead to disengagements and which do not, we visualized the planner in Fig. 6.9. This visualization shows the model has learned that actions which lead to collisions, driving into the street, or driving into a driveway will lead to a disengagement.

We also investigated the ability of LaND to continuously improve as more data is gathered. In this controlled experiment, we first ran our method on 1.3 km of sidewalk. We then trained our method on this additional data—which included mostly engagements, but also a few disengagements—and evaluated this finetuned model on the same 1.3 km of sidewalk. Table 6.2 shows that our method improves by over $2\times$ when finetuned on the collected data, showing that our approach is able to continue to improve as more data is collected.

6.4 Discussion

We presented LaND, a method for learning navigation policies from disengagements. LaND directly leverages disengagement data by learning a model that predicts which actions lead to disengagements given the current sensory observation. This predictive disengagement model can then be used for planning and control to avoid disengagements. Our results demonstrate LaND

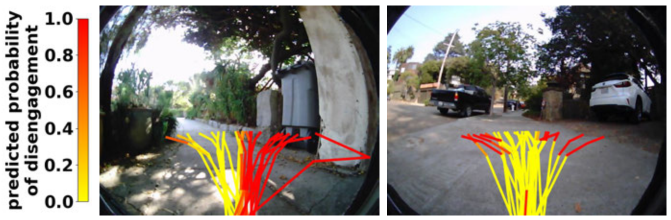


Figure 6.9: Visualization of planning with the disengagement predictive model. Each image shows the candidate paths considered during planning, and are color coded according to their predicted probability of disengagement. These visualizations show the learned model can accurately predict which action sequences would lead to disengagements, including driving into obstacles (left) or streets or driveways (right).

Method	Avg. distance until disengagement (meters)
LaND	101.2
LaND with finetuning	218.3

Table 6.2: Experimental demonstration of our LaND improving as more data is gathered. We first collected data using our LaND control policy on 1.3 km of never-before-seen sidewalk. We then finetuned our method on this additional data, and evaluated this finetuned model on the same 1.3 km of sidewalk. Our method improves by over 2$\times$ when finetuned on the collected data, showing that our approach is able to continue to improve as more data is gathered.

can successfully learn to navigate in diverse, real world sidewalk environments, demonstrating that disengagements are not only useful as a tool for testing, but also directly for learning to navigate.

Part III

Safety

Chapter 7

Safe Learning using Expert Supervision

In Part II, we studied different sources of supervision that a robot can learn from. However, some of these learning signals—such as collision—require the robot to experience catastrophic failure. This requirement is unacceptable for safety-critical mobile robot applications.

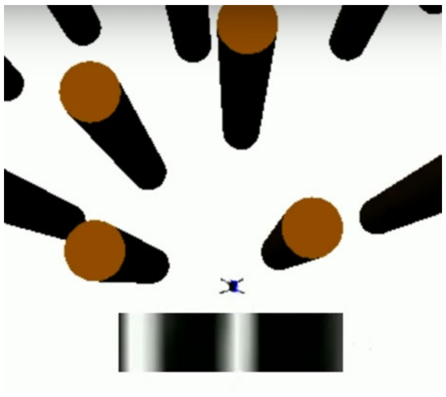


Figure 7.1: **Policy Learning using Adaptive Trajectory Optimization**: A neural network control policy trained by PLATO navigates through a forest using camera images. During training, the adaptive MPC teacher policy chooses actions to achieve good long-horizon task performance while matching the learner policy distribution. The policy learned by PLATO converges with bounded cost.

In this chapter, we study how to leverage computational experts to enable safe robot learning. We propose PLATO (Policy Learning using Adaptive Trajectory Optimization), a reset-free method for training complex policies that combine perception and control by using a trajectory optimization teacher in the form of model-predictive control (MPC). At training time, MPC chooses actions that make a tradeoff between succeeding at the task and matching the behavior of the current policy. By gradually adapting to the policy, MPC ensures that the states visited during training will allow the policy to learn good long-horizon performance. MPC makes use of full state information, which could be obtained, for example, by instrumenting the environment at training time. The final policy, however, is trained to mimic the MPC actions using only the observations available to the robot, which makes it possible to run the resulting policy at test time without any instrumentation. The algorithm requires access to at least a rough model of the system dynamics in order to run MPC during training, but does not require any knowledge of the observation model, making it feasible to use with complex, raw observation signals, such as images and depth scans. Since MPC is used to select all actions at training time, the algorithm never requires running a partially trained and potentially unsafe policy.

We prove that the policy learned by PLATO converges to a policy with bounded cost. Our empirical results further demonstrate that PLATO can learn complex policies for simulated quadrotor flight with laser rangefinder observations and camera observations in cluttered environments and at high speeds. We show that PLATO outperforms a number of previous approaches in terms of both the performance of the final neural network policy and the robustness to catastrophic failure during training. In comparisons with MPC-guided policy search [138], the DAgger algorithm [63], DAgger with coaching [139] and supervised learning, our approach experiences substantially fewer catastrophic failures both during training time and at test time.

7.1 Preliminaries and Overview

We address the problem of learning control policies for dynamical systems, such as robots and autonomous vehicles. The system is defined by states $\mathbf{x}$ and actions $\mathbf{u}$. The policy must control the system from observations $\mathbf{o}$, which are in general insufficient for determining the full state $\mathbf{x}$. The policy is a conditional distribution over actions $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$, parametrized by θ. At test time, the agent chooses actions according to $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ at each time step t, and experiences a loss $c(\mathbf{x}_t, \mathbf{u}_t)$. We assume without loss of generality that $c(\mathbf{x}_t, \mathbf{u}_t)$ is in the interval $[0, 1]$. The next state is distributed according to the dynamics $p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$. The goal is to learn a policy $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ that minimizes the total cost $J(\pi) = E_\pi\big[\sum_{t=1}^T c(\mathbf{x}_t, \mathbf{u}_t)\big]$. We will use $J_t(\pi|\mathbf{x}_t) = E_\pi\big[\sum_{t'=t}^T c(\mathbf{x}_{t'}, \mathbf{u}_{t'})|\mathbf{x}_t\big]$ as shorthand for the expected cost from state $\mathbf{x}_t$ at time t, such that $J(\pi) = E_{\mathbf{x}_1\sim p(\mathbf{x}_1)}\big[J_1(\pi|\mathbf{x}_1)\big]$.

In this work, we further assume that during training, our algorithm has access to the true underlying states $\mathbf{x}$. This additional assumption allows us to use simple and efficient model-predictive control (MPC) methods to generate training actions. We do not require knowing the true states $\mathbf{x}$ at test time, since the learned policy $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ only requires observations. This training setup could be implemented in various ways in practice, including instrumenting the training environment (e.g. using motion capture to track a mobile robot) or using more effective hardware at training time (such as a more accurate GPS system), while only having access to cheaper and more practical hardware at test time. While this assumption does introduce some restrictions, we will show that it enables very efficient and relatively safe training, making it an appealing option for safety-critical systems.

We will train the policy $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ by mimicking a computational "teacher," rather than attempting to learn the policy directly with reinforcement learning. There are three key advantages to this approach: first, the teacher can exploit the true state $\mathbf{x}$, while the final policy π_θ is only trained on the observations $\mathbf{o}$; second, we can choose a teacher that will remain safe and stable, avoiding dangerous actions during training; third, we can train the final policy π_θ using standard, robust supervised learning algorithms, which will allow us to construct a simple and highly data-efficient algorithm that scales easily to complex, high-dimensional policy parametrization. Specifically, we will use MPC as the teacher. MPC uses the true state $\mathbf{x}$ and a model of the system dynamics (which we assume to be known in advance, but which in general could also be learned from experience). MPC plans locally optimal trajectories with respect to the dynamics, and by replanning every time step, is able to achieve considerable robustness to unexpected perturbations and model errors [140], making it an excellent choice for sample-efficient learning.

7.2 Policy Learning using Adaptive Trajectory Optimization

One naïve approach to learn a policy from a computational teacher such as MPC would be to generate a training set with MPC, and then train the policy with supervised learning to maximize the log-likelihood of this dataset. The teacher can safely choose robust, near-optimal trajectories. However, this type of supervision ignores the fact that the state distribution for the teacher and that of the learner are different [63]. Formally, the distribution of states at test time will not match the distribution at training time, and we therefore cannot expect good long-horizon performance from the learned policy.

In order to overcome this challenge, PLATO uses an adaptive MPC teacher that modifies its actions in order to bring the state distribution in the training data closer to that of the learned policy, while still producing robust trajectories and reacting intelligently to unexpected perturbations that cannot be handled by a partially trained policy. To that end, the teacher generates actions at each time step t from a controller obtained by optimizing the following objective:

$$\pi_\lambda^t(\mathbf{u}|\mathbf{x}_t, \theta) \leftarrow \arg\min_\pi J_t(\pi|\mathbf{x}_t) + \lambda D_{\mathrm{KL}}\big(\pi(\mathbf{u}|\mathbf{x}_t)||\pi_\theta(\mathbf{u}|\mathbf{o}_t)\big), \tag{7.1}$$

where λ determines the relative importance of matching the learner π_θ versus optimizing the expected return $J(\cdot)$. Since the teacher uses an MPC algorithm, this objective is reoptimized at each time step to obtain a locally optimal controller for the current state. The only difference from a standard MPC algorithm is the inclusion of the KL-divergence term. The particular MPC algorithm we use is based on iterative LQG (iLQG) [141], using a maximum entropy variant that produces linear-Gaussian stochastic controllers of the form $\pi_\lambda(\mathbf{u}|\mathbf{x}_t) = \mathcal{N}(\mathbf{K}_t\mathbf{x}_t + \mathbf{k}_t, \Sigma_t)$ [142]. The details of this maximum entropy variant of iLQG may be found in prior work [141], [143], [144]. We describe the details of PLATO and its relation to prior methods in Sec. 7.2 and show that PLATO produces a good learned policy in Sec. 7.3.

Algorithm 5 PLATO algorithm

```
1:  Initialize data $\mathcal{D} \leftarrow \emptyset$
2:  for $i = 1$ to $N$ do
3:      for $t = 1$ to $T$ do
4:          Optimize $\pi_\lambda^t$ with respect to Equation (7.1)
5:          Sample $\mathbf{u}_t \sim \pi_\lambda^t(\mathbf{u}|\mathbf{x}_t, \theta)$
6:          Optimize $\pi^*$ with respect to Equation (7.2)
7:          Sample $\mathbf{u}_t^* \sim \pi^*(\mathbf{u}|\mathbf{x}_t)$
8:          Append $(\mathbf{o}_t, \mathbf{u}_t^*)$ to the dataset $\mathcal{D}$
9:          State evolves $\mathbf{x}_{t+1} \sim p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$
10:     end for
11:     Train $\pi_{\theta_{i+1}}$ on $\mathcal{D}$
12: end for
```

Algorithm Description

Algorithm 5 outlines PLATO. We collect training trajectories by choosing actions $\mathbf{u}_t$ according to an adaptive teacher policy $\pi_\lambda^t(\mathbf{u}|\mathbf{x}_t, \theta)$, which is generated by optimizing the objective in Equation 7.1 at each time step via iLQG. We then update the learner policy $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ with supervised learning at the observations $\mathbf{o}_t$ corresponding to the visited states $\mathbf{x}_t$ to minimize the difference between $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ and the locally optimal policy

$$\pi^*(\mathbf{u}|\mathbf{x}_t) \leftarrow \arg\min_\pi J(\pi), \tag{7.2}$$

which is also obtained via MPC, but without considering the KL-divergence term. This approach ensures the teacher visits states that are similar to those that would be visited by the learner policy π_θ, while still providing supervision from a near-optimal policy. Note that the MPC policy is conditioned on the state of the system $\mathbf{x}_t$, while the learned policy $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ is only conditioned on the observations. MPC requires access to at least a rough model of the system dynamics, as well as the system state, in order to robustly choose near-optimal actions. However, by training π_θ on the corresponding observations, instead of the true states, π_θ can learn to process raw sensory inputs without requiring true state observations, making it possible to run the learned policy with only the raw observations at test time. In the rest of this section, we describe the MPC teacher and the supervised learning procedure in detail.

Adaptive MPC teacher: The teacher's policy π_λ^t must take reasonable, robust actions while visiting states that are similar to those that would be seen by the learner policy π_θ. However, we do not know the state distribution of π_θ in advance, since although we have some approximate knowledge of the system dynamics, we do not assume a model of the observation function that produces observations $\mathbf{o}_t$ from states $\mathbf{x}_t$, making it impossible to simulate the policy π_θ into the future. Instead, we choose the actions at each time step according to an MPC policy π_λ^t that minimizes the expected long-term sum of costs $J_t(\pi_\lambda^t|\mathbf{x}_t)$, but only greedily minimizes the KL-divergence against π_θ at the current time step t, where the observation $\mathbf{o}_t$ is already available, resulting in the objective in Equation 7.1. Since MPC reoptimizes the local policy at each time step, this method produces a sequence of policies $\pi_\lambda^{1:T}$, each of which is optimized with respect to its long-horizon cost and immediate disagreement with π_θ.

As discussed previously, our iLQG-based MPC algorithm produces linear-Gaussian local controllers $\pi_\lambda^t(\mathbf{u}|\mathbf{x}_t) = \mathcal{N}(\mu_\lambda(\mathbf{x}_t), \Sigma_t)$ where $\mu_\lambda(\mathbf{x}_t) = \mathbf{K}_t\mathbf{x}_t + \mathbf{k}_t$. We will further assume that our learner policy is conditionally Gaussian (but nonlinear), though other parametric distributions are also possible. The policy therefore has the form $\pi_\theta(\mathbf{u}|\mathbf{o}_t) = \mathcal{N}(\mu_\theta(\mathbf{o}_t), \Sigma_{\pi_\theta})$ where $\mu_\theta(\mathbf{o}_t)$ is the output of a nonlinear function, such as a neural network, and covariance Σ_{π_θ} can be either learned or deterministic. Then the MPC objective can be expressed in closed form:

$$\min_\pi \; J_t(\pi|\mathbf{x}_t) + \frac{1}{2}\lambda\Big[\ln\left(\frac{|\Sigma_{\pi_\theta}|}{|\Sigma_t|}\right) + \text{tr}\big(\Sigma_{\pi_\theta}^{-1}\Sigma_t\big) + \big(\mu_\theta(\mathbf{o}_t) - \mu_\lambda(\mathbf{x}_t)\big)^\intercal \Sigma_{\pi_\theta}^{-1}\big(\mu_\theta(\mathbf{o}_t) - \mu_\lambda(\mathbf{x}_t)\big) + \text{const}\Big].$$

The KL-divergence term in this objective is quadratic in $\mathbf{u}_t$ and linear in the covariance Σ_t, with an entropy maximization term $-\ln|\Sigma_t|$. This is precisely the objective that is optimized by the maximum entropy variant of iLQG [144], and optimization requires us only to expand the cost-to-go J_t to second order, which is a standard procedure in iLQG.

Training the learner's policy: We want the learner's policy π_θ to approach the optimal policy $\pi^*(\mathbf{u}|\mathbf{x}_t)$. We can estimate a (locally) optimal policy π^* at each state $\mathbf{x}_t$ with iLQG, simply by repeating the optimization at each time step but excluding the KL-divergence term. During the supervised learning phase, we minimize the KL-divergence between the learner π_θ and the precomputed near-optimal policies π^* at the observations stored in the dataset $\mathcal{D}$:

$$\theta \leftarrow \arg\min_\theta \sum_{(\mathbf{x}_t,\mathbf{o}_t)\in\mathcal{D}} D_{\mathrm{KL}}\big(\pi_\theta(\mathbf{u}|\mathbf{o}_t)||\pi^*(\mathbf{u}|\mathbf{x}_t)\big). \tag{7.3}$$

Since both π_θ and π^* are conditionally Gaussian, the KL-divergence can be expressed in closed-form:

$$\min_\theta \frac{1}{2} \sum_{(\mathbf{x}_t,\mathbf{o}_t)\in\mathcal{D}} \big(\mu^*(\mathbf{x}_t) - \mu_\theta(\mathbf{o}_t)\big)^\intercal \Sigma_{\pi^*}^{-1} \big(\mu^*(\mathbf{x}_t) - \mu_\theta(\mathbf{o}_t)\big) + \mathrm{tr}\big(\Sigma_{\pi^*}^{-1}\Sigma_{\pi_\theta}\big) + \ln\left(\frac{|\Sigma_{\pi^*}|}{|\Sigma_{\pi_\theta}|}\right) + \text{const.}$$

Ignoring the terms that do not involve the learner policy mean $\mu_\theta(\mathbf{o}_t)$, the objective function can be rewritten in the form of a weighted Euclidean loss:

$$\min_\theta \sum_{(\mathbf{x}_t,\mathbf{o}_t)\in\mathcal{D}} ||\mu^*(\mathbf{x}_t) - \mu_\theta(\mathbf{o}_t)||^2_{\Sigma_{\pi^*}^{-1/2}}.$$

This optimization can then be solved using standard regression methods. In our experiments, μ_θ is represented by a neural network, and the above optimization problem corresponds to standard neural network regression, solvable by stochastic gradient descent. The covariance of π_θ can be solved for in closed form, and corresponds to the inverse of the average precisions of π^* at the training points [142].

Relationship to previous work

The motivation behind PLATO is most similar to the MPC variant of guided policy search (MPC-GPS) [138]. However, PLATO lifts a major limitation of MPC-GPS. MPC-GPS requires the ability

approach	teacher policy	supervision policy
supervised learning	π^*	π^*
DAgger	π_{MIX}	π^*
DAgger + coaching	π_{MIX}	π_{COACH}
PLATO	π_λ	π^*

Table 7.1: **Overview of teacher-based policy optimization methods**: For PLATO and each prior approach, we list which teacher policy is used for sampling trajectories and which supervision policy is used for generating training actions from the sampled trajectories. Note that the prior methods execute the mixture policy π_{MIX}, which requires running the learned policy π_θ, potentially executing dangerous actions when π_θ is not fully trained.

to deterministically reset the environment into one of a small set of initial states. MPC-GPS requires deterministic resets because the KL-divergence term is evaluated using a linearization around each rollout. Deterministic episodic resets can be complex, time-consuming, or even impossible in the real world. For example, imagine a robot learning to navigate a human crowd; deterministic resets would require having the crowd walk through the same paths in each episode. Not requiring such resets is a major advantage. Furthermore, even when deterministic resets are feasible, PLATO empirically outperforms MPC-GPS (Section 8.4).

Formally, PLATO can also be viewed as a generalization of the Dataset Aggregation (DAgger) algorithm [63], which samples trajectories according to the mixture policy $\pi_{\text{MIX}i} = \beta_i\pi^* + (1-\beta_i)\pi_{\theta i}$. The training data is generated from the observations sampled by executing $\pi_{\text{MIX}i}$ but labelled with actions from π^*. DAgger converges if $\frac{1}{N}\sum_{i=1}^{N}\beta_i \to 0$ as $N \to \infty$. Coaching [139], a related extension to DAgger, modifies the supervision policy π^* to adapt to the learned policy π_θ by labelling the training data with a coach policy π_{COACH} that encourages the action training labels to be similar to the actions $\pi_{\theta i}$ would choose. Our empirical evaluation shows that PLATO outperforms coaching.

Another distinction of PLATO is the use of an adaptive MPC policy $\pi_\lambda^{1:T}$ to select the actions at each time step, rather than the mixture policy π_{MIX} used in the prior methods. As demonstrated in our evaluation, this adaptive MPC policy allows PLATO to robustly avoid catastrophic failure during training, which is particularly important in safety-critical domains. Our experiments also demonstrate that policies trained using PLATO empirically outperform policies trained by either DAgger or coaching. Table 7.1 summarizes the teacher and supervision policies used by PLATO and prior work.

7.3 Theoretical Analysis

In this section, we present a proof that the policy π_θ learned by PLATO converges to a policy with bounded cost. This proof extends the result by Ross et al. [63], which only admits mixture policies, to our adaptive MPC policy $\pi_\lambda^{1:T}$.

Given a policy π, we denote d_π^t as the state distribution at time t when executing policy π from time 1 to $t-1$. Define the cost function $c(\mathbf{x}_t, \mathbf{u}_t)$ as a function of state $\mathbf{x}_t$ and control $\mathbf{u}_t$, with $c(\mathbf{x}_t, \mathbf{u}_t) \in [0, 1]$ without loss of generality. We wish to learn a policy $\pi_\theta(\mathbf{u}|\mathbf{o}_t)$ that minimizes the total expected cost over time horizon T:

$$J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{\mathbf{x}_t \sim d_{\pi_\theta}^t}[\mathbb{E}_{\mathbf{u}_t \sim \pi_\theta(\mathbf{u}|\mathbf{o}_t)}[c(\mathbf{x}_t, \mathbf{u}_t)|\mathbf{x}_t]].$$

Let $J_t(\pi, \tilde{\pi})$ denote the expected cost of executing π for t time steps, and then executing $\tilde{\pi}$ for the remaining $T-t$ time steps, and let $Q_t(\mathbf{x}, \pi, \tilde{\pi})$ denote the cost of executing π for one time step starting from initial state $\mathbf{x}$, and then executing $\tilde{\pi}$ for the remaining $t-1$ time steps. We assume the cost-to-go difference between the learned policy and the optimal policy is bounded:

$Q_t(\mathbf{x}, \pi_\theta, \pi^*) - Q_t(\mathbf{x}, \pi^*, \pi^*) \leq \delta$. In the worst case, δ is $O(T)$ and PLATO (as well as similar methods such as DAgger) will not outperform supervised learning. However, if π^* is able to quickly recover from mistakes made by π_θ, δ will be $O(1)$ [63].

When optimizing Equation 7.1 to obtain the teacher policy π_λ, we choose λ such that $D_{\text{KL}}(\pi_\lambda(\mathbf{u}|\mathbf{x})||\pi_\theta(\mathbf{u}|\mathbf{o})) \leq \epsilon_{\lambda\theta}$ for all state-observation pairs $(\mathbf{x}, \mathbf{o})$. We can always guarantee this bound when optimizing Equation 7.1 because $D_{\text{KL}}(\pi_\lambda(\mathbf{u}|\mathbf{x})||\pi_\theta(\mathbf{u}|\mathbf{o})) \to 0$ as $\lambda \to \infty$.

When optimizing the supervised learning objective in Equation 7.3 to obtain the learner policy π_θ, we assume the supervised learning objective function error is bounded by a constant $D_{\text{KL}}(\pi_\theta(\mathbf{u}|\mathbf{o})||\pi^*(\mathbf{u}|\mathbf{x})) \leq \epsilon_{\theta *}$ for all states $\mathbf{x}$ (and corresponding observations $\mathbf{o}$) in the dataset, which were sampled from the teacher policy distribution d_{π_λ}. Since the policy π_θ is trained with supervised learning precisely on these states $\mathbf{x} \sim d_{\pi_\lambda}$, this bound $\epsilon_{\theta *}$ corresponds to assuming that the learner policy π_θ attains bounded training error.

Let $l(\mathbf{x}, \pi_\theta, \pi^*)$ denote the expected 0-1 loss of π_θ with respect to π^* in state $\mathbf{x}$: $\mathbb{E}_{\mathbf{u}_\theta \sim \pi_\theta(\mathbf{u}|\mathbf{o}), \mathbf{u}^* \sim \pi^*(\mathbf{u}|\mathbf{x})}[\mathbf{1}[\mathbf{u}_\theta \neq \mathbf{u}^*]]$. We note that the total variation divergence is an upper bound on the 0-1 loss [145] and the KL-divergence is an upper bound on the total variation divergence [146]. Therefore for all states $\mathbf{x} \sim d_{\pi_\lambda}$ in the dataset used for supervised learning, the 0-1 loss can be upper bounded:

$$\begin{aligned} l(\mathbf{x}, \pi_\theta, \pi^*) &= \mathbb{E}_{\mathbf{u}_\theta \sim \pi_\theta(\mathbf{u}|\mathbf{o}), \mathbf{u}^* \sim \pi^*(\mathbf{u}|\mathbf{x})}[\mathbf{1}[\mathbf{u}_\theta \neq \mathbf{u}^*]] \\ &\leq D_{\text{TV}}(\pi_\theta(\mathbf{u}|\mathbf{o})||\pi^*(\mathbf{u}|\mathbf{x})) \\ &\leq \sqrt{D_{\text{KL}}(\pi_\theta(\mathbf{u}|\mathbf{o})||\pi^*(\mathbf{u}|\mathbf{x}))} \\ &\leq \sqrt{\epsilon_{\theta *}}. \end{aligned}$$

We also note the state distribution bound $||d_\pi^t - d_{\tilde{\pi}}^t||_1 \leq 2t\sqrt{D_{\text{KL}}^{\max}(\pi, \tilde{\pi})}$ proven in [147]. This lemma implies that for an arbitrary function $f(\mathbf{x})$, $E_{\mathbf{x} \sim d_\pi^t}[f(\mathbf{x})] \leq E_{\mathbf{x} \sim d_{\tilde{\pi}}^t}[f(\mathbf{x})] + 2f^{\max}t\sqrt{D_{\text{KL}}^{\max}(\pi, \tilde{\pi})}$

We can then prove the following theorem:

Theorem 7.3.1 *Let the cost-to-go* $Q_t(\mathbf{x}, \pi_\theta, \pi^*) - Q_t(\mathbf{x}, \pi^*, \pi^*) \leq \delta$ *for all* $t \in \{1, ..., T\}$. *Then for* PLATO, $J(\pi_\theta) \leq J(\pi^*) + \delta\sqrt{\epsilon_{\theta *}}O(T) + O(1)$.

Proof:

$$\begin{aligned}
J(\pi_\theta) &= J(\pi^*) + \sum_{t=0}^{T-1} J_{t+1}(\pi_\theta, \pi^*) - J_t(\pi_\theta, \pi^*) \\
&= J(\pi^*) + \sum_{t=1}^{T} \mathbb{E}_{\mathbf{x}\sim d^t_{\pi_\theta}}[Q_t(\mathbf{x}, \pi_\theta, \pi^*) - Q_t(\mathbf{x}, \pi^*, \pi^*)] \\
&\leq J(\pi^*) + \delta \sum_{t=1}^{T} \mathbb{E}_{\mathbf{x}\sim d^t_{\pi_\theta}}[l(\mathbf{x}, \pi_\theta, \pi^*)] && \text{(7.4a)} \\
&\leq J(\pi^*) + \delta \sum_{t=1}^{T} \mathbb{E}_{\mathbf{x}\sim d^t_{\pi_\lambda}}[l(\mathbf{x}, \pi_\theta, \pi^*)] + 2l^{\max} t\sqrt{\epsilon_{\lambda\theta}} && \text{(7.4b)} \\
&\leq J(\pi^*) + \delta \sum_{t=1}^{T} \sqrt{\epsilon_{\theta *}} + 2t\sqrt{\epsilon_{\theta *}}\sqrt{\epsilon_{\lambda\theta}} && \text{(7.4c)} \\
&= J(\pi^*) + \delta T\sqrt{\epsilon_{\theta *}} + \delta T(T+1)\sqrt{\epsilon_{\theta *}}\sqrt{\epsilon_{\lambda\theta}}
\end{aligned}$$

Equation 7.4a follows from the fact that the expected 0-1 loss of π_θ with respect to π^* is the probability that π_θ and π^* pick different actions in $\mathbf{x}$; when they choose different actions, the cost-to-go increases by $\leq \delta$. Equation 7.4b follows from the state distribution bound proven in [147]. Equation 7.4c follows from the upper bound on the 0-1 loss.

Although we do not get to choose $\epsilon_{\theta *}$ because that is a property of the supervised learning algorithm and the data, we are able to choose $\epsilon_{\lambda\theta}$ by varying parameter λ. If we choose λ such that $\epsilon_{\lambda\theta} = O(\frac{1}{T^2})$. We therefore have

$$J(\pi_\theta) \leq J(\pi^*) + \delta\sqrt{\epsilon_{\theta *}}O(T) + O(1) \quad . \qquad \square$$

As with DAgger, in the worst case $\delta = O(T)$. However, in many cases $\delta = O(1)$ or is sub-linear in T, for instance if π^* is able to quickly recover from mistakes made by π_θ. We also note that this bound, $O(T)$, is the same as the bound obtained by DAgger, but without actually needing to directly execute π_θ at training time. Compared to supervised learning with bound $O(T^2)$ [63], PLATO trains the policy at states closer to those induced under its own distribution.

7.4 Experiments

We evaluate PLATO on a series of simulated quadrotor navigation tasks. MPC is a standard choice for quadrotor control [148] because approximate models are typically known in advance from standard rigid body physics and the vehicle specifications. However, effective use of MPC requires explicit state estimation and can be computationally intensive. It is therefore very appealing to be able to train an entirely feedforward, reactive policy to control a quadrotor performing navigation in obstacle-rich environments, directly in response to raw sensor inputs. During training, the vehicle might be placed in a known, instrumented training environment to collect data using MPC, while at test time, the learned feedforward policy could control the aircraft directly from raw observations. This makes simulated quadrotor navigation an ideal domain in which to compare PLATO to prior work.

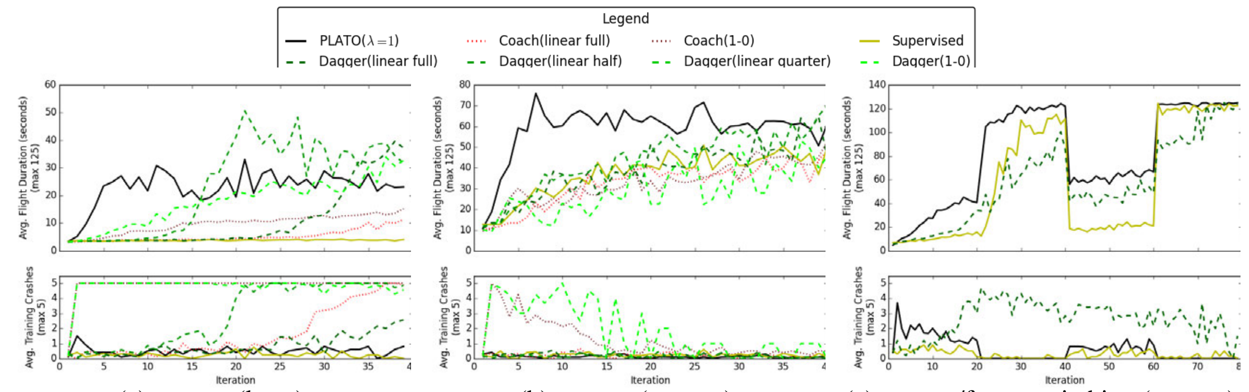
Prior methods and baselines: We compare PLATO to four methods. The first method is DAgger, which, as discussed in Section 7.2, executes a mixture of the learned policy and teacher policy, which in this case is MPC (without a KL-divergence term). DAgger has previously been used for learning quadrotor control policies from human demonstrations [55]. While DAgger carries the same convergence guarantees as PLATO, successful use of DAgger requires the learned policy to be executed at training time, before the policy has converged to a near-optimal behavior. The second method is the coaching algorithm of [139] which, like DAgger, executes a mixture of the learned and teacher policies, but supervises the learner using the adapted policy. In these experiments, we chose the coaching policy π_{COACH} to be the teacher policy π_λ from PLATO. For both DAgger and coaching, we must choose the mixing parameter β_i at each iteration i. Since the performance of these algorithms is quite sensitive to the schedule of the β_i parameter, we include four schedules for comparison: three linear schedules that interpolate β_i from 1 at the first iteration to 0 at the last iteration ("linear full"), the halfway iteration ("linear half"), and the quarter-way iteration ("linear quarter"), as well as the more standard "1-0" schedule that sets $\beta_i = 1[i = 1]$. The third method is MPC-GPS [138], which, unlike PLATO, DAgger and coaching, requires deterministic resets during training (Figure 7.4b). In addition to these prior methods, we also compare our approach to a standard supervised learning baseline, which always executes the MPC policy without any adaptation. For all experiments, we assume additive Gaussian noise is applied to both controls and observations.

Policy representation: For all of the methods, we represent π_θ as a conditional Gaussian policy, with a constant covariance and a mean given by a neural network function of the observation $\mathbf{o}_t$. The network has two fully connected hidden layers of size 40 with ReLU activations [149]. The loss function is the weighted euclidean loss (see Section 7.2). We used the Caffe [150] framework and the ADAM solver [151]. Each iteration was trained using the final weights from the previous iteration.

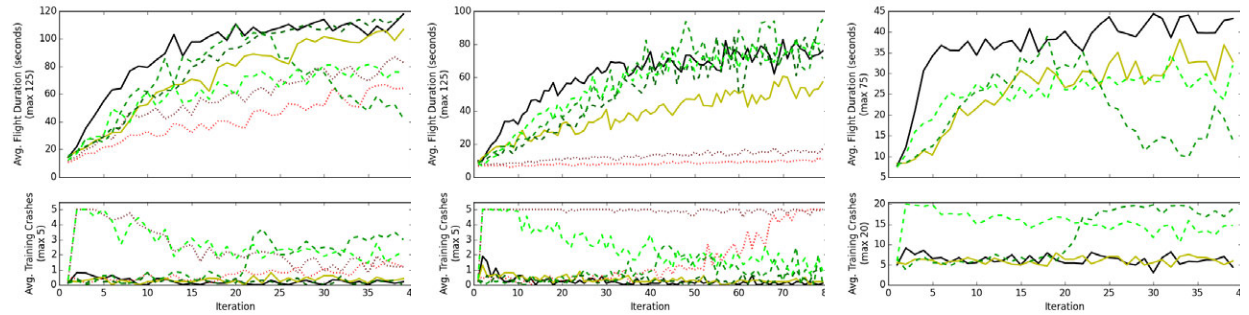
Experimental domains: The comparisons are conducted on two test environments: a winding canyon with randomized turns, and a dense forest of cylindrical trees with randomized positions. An example environment is shown in Figure 7.1. The canyon changes direction up to $\frac{\pi}{4}$ radians every 0.5m. The forest is composed of 0.5m radius cylinders with an average spacing of 2.5m. The target velocity is 6m/s in the canyon and 2m/s in the forest.

The dynamical system is a quadrotor with dynamics described by [152]. The state of the vehicle $\mathbf{x} \in \mathbb{R}^{13}$ consists of the position and orientation, as well as their time derivatives, and the control $\mathbf{u} \in \mathbb{R}^4$ consists of motor velocities. The observations $\mathbf{o}$ consist of orientation, linear velocity, angular velocity and either (i) a set of 30 equally spaced 1-d laser depth scanners arranged in 180 degree fan in front of the vehicle ($\mathbf{o} \in \mathbb{R}^{40}$) or (ii) a 5×20 grayscale camera image ($\mathbf{o} \in \mathbb{R}^{110}$). Learning neural network policies with these observations forces the policies to perform both perception and control, since success on each of the domains requires avoiding obstacles using only raw sensory input.

The cost function for the MPC teacher encourages the quadrotor to fly at a specific linear velocity



(a) canyon (laser) (b) canyon (camera) (c) canyon/forest switching (camera)



(d) forest (laser) (e) forest (camera) (f) velocity commands in forest (laser)

Figure 7.2: **Experiments**: We compare PLATO to baseline methods in a winding canyon, a dense forest, and an alternating canyon/forest. For each scenario and learning method, we trained 10 different policies using different random seeds. Each iteration required 2 minutes of flight time. Then for each policy, we evaluated the neural network policy trained at each iteration by flying through the scenario 20 times. Therefore each datapoint corresponds to 200 samples.

and orientation while minimizing control effort and avoiding collisions:

$$\begin{aligned}L(\mathbf{x},\mathbf{u}) =& 10^3||\mathbf{x}_{\text{LINVEL}} - \mathbf{x}^*_{\text{LINVEL}}||_2^2 + 10^3||\mathbf{x}_{\text{HEIGHT}} - \mathbf{x}^*_{\text{HEIGHT}}||_2^2 + \\ & 10^4||\mathbf{x}_{\text{QUAT}} - \mathbf{x}^*_{\text{QUAT}}||_2^2 + 250||\mathbf{x}_{\text{ANGVEL}}||_2^2 + \\ & 5^{-3}||\mathbf{u} - \mathbf{u}_{\text{HOVER}}||_2^2 + \\ & 10^3 \max(d_{\text{SAFE}} - \text{signed-distance}(\mathbf{x}), 0),\end{aligned}$$

where $\mathbf{x}_{\text{LINVEL}}, \mathbf{x}_{\text{HEIGHT}}, \mathbf{x}_{\text{QUAT}}, \mathbf{x}_{\text{ANGVEL}}$ are the linear velocity, height, orientation, and angular velocity of the state $\mathbf{x}$, respectively; $\mathbf{x}^*_{\text{LINVEL}}, \mathbf{x}^*_{\text{HEIGHT}}, \mathbf{x}^*_{\text{QUAT}}$ are the target linear velocity, height, and orientation, respectively; and $\mathbf{u}_{\text{HOVER}}$ is the rotor velocity when the quadrotor is hovering. The final term is a hinge loss on the distance of the quadrotor to the nearest obstacle; there is no penalty if the nearest obstacle is further than d_{SAFE}.

Performance of learned policies: In Figures 7.2a, 7.2b, 7.2d, and 7.2e, we present the mean time to failure (MTTF) of the learned policy π_θ on the canyon and forest environments using the laser or camera sensors. The graphs show the MTTF of each policy at each iteration of the learning process, averaged over 10 training runs of each method with 20 repetitions each. Failure occurs when the quadrotor crashes into an obstacle, with the maximum flight time for each domain listed on the graphs. The results indicate that the PLATO algorithm is able to learn effective policies faster, and converges to a solution that is better than or comparable to the baseline methods. For some choices of β schedule and supervision scheme, some DAgger variants achieve similar final MTTFs, but always at a slower rate and, as discussed next, with significantly more training crashes.

Robustness during training: In Figures 7.2a, 7.2b, 7.2d, and 7.2e, we show the number of crashes experienced during training at each iteration. PLATO on average experiences less than one crash per iteration, comparable in performance to the baseline MPC method (supervised learning), indicating that mimicking the learner with a KL-divergence penalty does not substantially degrade the robustness of MPC. In contrast, both DAgger and coaching begin to experience a substantial number of failures when the mixing constant β drops. By carefully selecting the schedule for β, the number of crashes can be reduced.

However, even with a carefully chosen schedule, the prior methods are vulnerable to non-stationary training environments, as illustrated in Figure 7.2c. In this experiment, the vehicle switches from the canyon to the forest halfway through training, and then switches back to the forest. Prior methods that directly execute π_θ during training experience many crashes because a policy trained only on the canyon cannot succeed on the forest without additional training. However, PLATO experiences on average less than one crash per episode because PLATO is able to automatically switch to more off-policy behavior when encountering novel scenarios. While this example might appear pathological, it is in fact a plausible training setup for a real quadrotor exploring a varied environment, such as different floors of a building. If the walls on one floor are painted, e.g., a different color than the rest, the learned policy could easily experience a catastrophic failure when entering the floor for the first time, even if it was consistently successful on preceding floors.

Policies with user velocity commands: Figure 7.2f shows the performance of PLATO when learning policies that take an additional input to simulate high-level user control in the form of the desired velocity of the quadrotor. These policies are useful because instead of training multiple policies for different target velocities, we can train one generalizable policy. This input modifies the cost function used by MPC, producing command-aware supervision. During training, the commands vary in the range of ± 1 m/s sideways and 1 to 2.5 m/s forward. At test time, we sample velocity commands uniformly at random; the velocity commands are re-sampled whenever the quadrotor reaches the current sampled velocity. The results indicate that PLATO can successfully learn such policies, outperforming prior methods and again minimizing the number of crashes during training.

Sensitivity to KL-divergence weight: Recall that λ determines the degree to which MPC prioritizes following the learner π_θ versus performing the desired task. As $\lambda \to 0$, PLATO approaches standard supervised learning and is thus safe, while as $\lambda \to \infty$, PLATO approaches DAgger 0-1. In practice, to choose lambda, we start with $\lambda = 0$, and then increase λ until the cost of the behavior starts to increase. Figure 7.3 compares different non-limiting settings of λ, while we refer to Figure 7.2 for limiting cases for $\lambda = 0$ and $\lambda = \infty$. The results suggest that a relatively broad range of λ values produces successful policies.

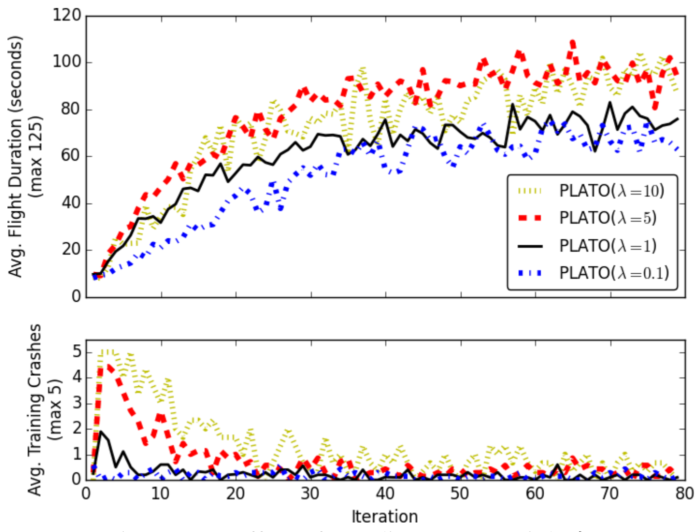
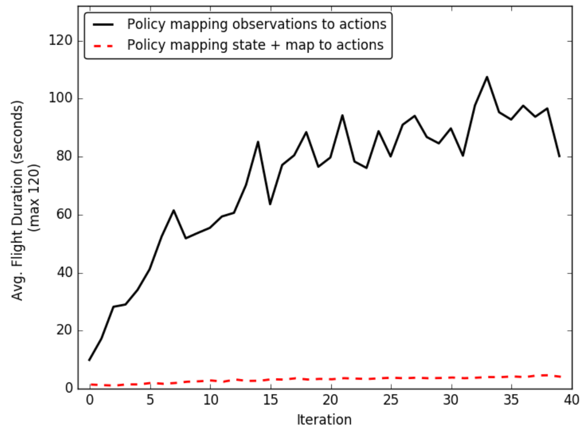


Figure 7.3: Effect of KL-divergence weight λ.

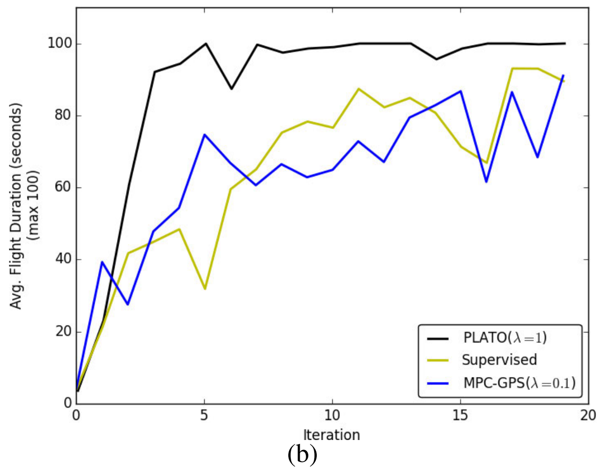
Comparison with training on full state: Figure 7.4a shows a comparison where the policy maps state to action using an oracle SLAM algorithm that provides perfect state information and a local 2D distance map of the obstacles. The observation-based policy substantially outperforms the policy that learns to map the state to action, even with an oracle SLAM algorithm. Although the state and obstacle map are sufficient to choose good actions, this mapping is much harder to learn. Of course, alternative full state representations that are carefully engineered to the task may perform better, but this experiment demonstrates that, at least in some cases, mapping observations directly to actions without going through full state estimation can lead to better performance.

Comparison with MPC-GPS: MPC-GPS [138] cannot directly be evaluated on the domains described above, because training must occur in episodes with deterministic resets (see Section 7.2). We constructed a fixed-length episodic variant of the forest task where MPC-GPS was allowed to use deterministic resets. Besides not requiring an episodic formulation or deterministic resets, the comparison in Figure 7.4b shows that PLATO substantially outperforms the policy learned by MPC-GPS in terms of MTTF.

Supplementary material, including a video, can be viewed online: `sites.google.com/site/platopolicy`.



(a)



(b)

Figure 7.4: Comparisons with (a) training on full state and (b) MPC-GPS.

7.5 Discussion

In this chapter, we presented PLATO, a continuous, reset-free algorithm for learning complex, high-dimensional policies that combine perception and control into a single expressive function approximator, such as a deep neural network. PLATO uses a trajectory optimization teacher to provide supervision to a standard supervised learning algorithm, allowing for a simple and data-efficient learning method. The teacher adapts to the behavior of the neural network policy to ensure that the distribution over states and observations is sufficiently close to the learned policy, allowing for a bound on the long-term performance of the learned policy. Our empirical evaluation on a simulated quadrotor demonstrates that PLATO outperforms a number of previous methods, both in terms of the robustness and effectiveness of the final policy, and in terms of the safety of the training procedure.

PLATO has two key advantages that make it well-suited for learning control policies for real-world robotic systems. First, since the learned neural network policy does not need to be executed at training time, the method benefits from the robustness of model-predictive control (MPC), minimizing catastrophic failures at training time. This is particularly important when the training state distribution is non-stationary. Methods that execute the learned policy, such as DAgger, can suffer a catastrophic failure when the agent encounters novel observations. Mitigating these issues typically requires hand-designed safety mechanisms, while PLATO automatically switches to a more off-policy behavior.

The second advantage of PLATO is that the learned policy can use a different set of observations than MPC. Effective use of MPC requires observing or inferring the full state of the system, which might be accomplished, for instance, by instrumenting the environment with motion capture, or using a known map with relocalization [153]. The policy, however, can be trained directly on raw input from onboard sensors, forcing it to perform both perception and control. Once trained, such a policy can be used in uninstrumented natural environments.

Chapter 8

Conservative Reinforcement Learning

In the previous chapter, we demonstrated how a robot can safely learn to navigate by leveraging a computational expert. However, the computational expert requires complete state and environment knowledge at training time, which severely restricts the feasibility of real-world deployment. In this chapter, we investigate reinforcement learning algorithms for safety-critical systems that reason about perception and control in unknown environments, understand uncertainty, and explore safely.

One of the central challenges in reinforcement learning is that a robot can only learn the outcome of an action by executing the action itself. Consider a robot learning to navigate an unknown environment while avoiding collisions. This scenario seemingly presents a quandary: the robot needs to learn how to avoid collisions in order to achieve the desired task, but to learn how to avoid collisions, the robot must experience (possibly catastrophic) collisions during training. The robot can overcome this quandary by first experiencing gentle collisions in order to learn about the environment; once the robot is confident about the environment, the robot can avoid catastrophic failures in the future. Central to this approach is that the robot must be able to reason about its own uncertainty because these catastrophic failures are likely to occur in novel scenarios.

Consider an example scenario in which an autonomous drone is learning to fly in an obstacle-rich building. If the drone encounters a novel scenario, the drone will likely crash because the novel scenario is not contained within the training distribution of the reinforcement learning algorithm policy. However, by reasoning about its own policy's uncertainty, the drone can safely interact with the environment and avoid catastrophic failures while also increasing the diversity of its training distribution.

To realize this kind of safe, uncertainty-aware navigation in unknown environments, we propose a model-based learning approach in which the robot learns a collision prediction model and uses estimates of the model's uncertainty to adjust its navigation strategy. By using a speed-dependent collision cost together with uncertainty-aware collision estimates, our navigation strategy naturally chooses to move cautiously when uncertainty is high so as to experience only harmless low-speed collisions, and increases speed only in regions where the confidence of the prediction model is high.

Our main contribution is an uncertainty-aware collision prediction model that enables a robot to learn how to accomplish a desired task in an unknown environment while only experiencing gentle collisions. The collision prediction model takes as input the current robot observation and a

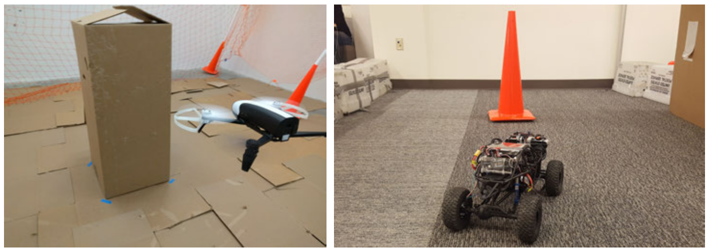


Figure 8.1: **Uncertainty-aware collision prediction model for collision avoidance**: A quadrotor and an RC car are tasked with navigating in an unknown environment. How should the robots navigate while avoiding collisions? We propose a model-based reinforcement learning approach in which the robot learns a collision prediction model by experiencing collisions at low speed, which is unlikely to damage the vehicle. We formulate a velocity-dependent collision cost that uses collision prediction estimates and their associated uncertainties to enable the robot to only experience safe collisions during training while still approaching the desired task performance.

sequence of controls, computes the probability of a collision occurring along with an estimate of its uncertainty, and outputs a speed-dependent collision cost. The speed-dependent collision cost is a function of the model and its uncertainty, which enables the robot to automatically avoid catastrophic high-speed collisions by acting cautiously in novel situations. We use a deep neural network for the collision prediction model, which allows the model to cope with raw, high-dimensional sensory inputs. To obtain uncertainty estimates from the neural network, we leverage uncertainty estimation methods for discriminatively trained neural networks based on a combination of bootstrapping [154] and dropout [155], [156]. A model-based reinforcement learning algorithm then gathers samples using the neural network collision prediction model, which are aggregated and used to further improve the collision prediction model. Our empirical results demonstrate that a robot equipped with our uncertainty-aware neural network collision prediction model experiences substantially fewer dangerous collisions during training while still learning to achieve the desired task. We present an evaluation of our method with various parameter settings for both a simulated and real-world quadrotor, and a real-world RC car (Fig. 8.1), and demonstrate that our method offers a favorable tradeoff between training-time collisions and final task performance compared to baseline approaches that do not explicitly reason about uncertainty.

8.1 Related Work

In this chapter, we investigate how model-based reinforcement learning for robot collision avoidance can be made safe and reliable at both training and test time. Reinforcement learning has been applied to a wide range of robotic problems, ranging from locomotion and manipulation to autonomous helicopter flight [157], [158]. Model-free methods have been particularly popular due to their simplicity and favorable computational properties [23]. However, model-based methods are generally known to be more sample-efficient [22]. In this chapter, we adopt a model-based approach and learn an uncertainty-aware collision avoidance model; however, similar uncertainty estimation techniques could be extended also to model-free methods.

Several model-based robotic learning algorithms have been proposed that explicitly reason about uncertainty [22], [159]. Uncertainty estimates have been used to perform both risk-averse and risk-seeking, optimistic exploration [160]. The role of uncertainty estimation in this chapter is to avoid unsafe actions at training time until the model has gained sufficient confidence, which is largely orthogonal and complementary to prior work that seeks to improve exploration in order to accelerate learning. Combining these two directions is a promising direction for future work.

Uncertainty-aware model-based reinforcement learning has been explored in previous work using Bayesian models [73], [74]. While our work is similar in the overall aim, one of the central goals of our method is to directly process raw inputs from high-bandwidth sensors such as cameras, which necessitates the use of rich and expressive models, such as deep neural networks. Uncertainty estimation for deep neural networks is substantially more challenging, since these models are inherently discriminative. Recent work has proposed to use a Bayesian formulation of neural networks based on dropout [161], as well as to use the bootstrap for exploration [162] (but not, to the best of our knowledge, for uncertainty estimation for the purpose of safety). In this chapter, we demonstrate that combining both dropout and bootstrap can yield substantially improved uncertainty estimates for reinforcement learning tasks.

There is much prior work on safe robot control for safety-critical systems such as autonomous cars [6], legged robots [163], and quadrotors [164]–[166]. A number of recent works have sought to address the question of safety for learning-based robotic systems. Methods based on reachability provide appealing theoretical guarantees, but cannot cope with rich sensory input and are often difficult to scale to high-dimensional systems [75]–[77]. Several works have suggested using discriminative models, including neural networks, to learn safety predictors [78]. These methods generally take the approach of training a model to predict whether an unsafe action will occur, and reverting to a hand-designed safety controller if such a potential failure is detected. Our method offers two advantages over this approach. First, by directly estimating model uncertainty, we do not rely on a discriminative safety estimator. This approach is preferred in environments where the model might encounter previously unseen inputs because a discriminative safety estimator cannot provide meaningful predictions for completely novel inputs; in short, the discriminative safety estimator may erroneously conclude that an unsafe environment is safe. In contrast, a statistical uncertainty prediction such as bootstrapping is more likely to estimate high uncertainty in novel environments. Secondly, our approach does not assume the existence of a manually designed safety control, but instead naturally reverts to more cautious exploratory behavior in the presence of uncertainty. This makes the approach more automated, and does not require a safety mechanism that can recover from arbitrary unsafe situations.

8.2 Preliminaries

Our goal is to control a mobile robot, such as a quadrotor or a car, attempting to navigate an unknown environment. The task may be formally defined in terms of states $\mathbf{x}$, actions $\mathbf{u}$, dynamics $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t)$, and observations $\mathbf{o}$. We use $\mathcal{M}$ to represent the environment, including any potential obstacles. We assume the robot's objective is encoded as a scalar cost function of the form

$$\mathcal{C}(\mathbf{x}_t, \mathbf{u}_t, \mathcal{M}) = \mathcal{C}_{\text{TASK}}(\mathbf{x}_t, \mathbf{u}_t) + \mathbb{1}_{\text{COLL}(\mathbf{x}_t, \mathcal{M})}\mathcal{C}_{\text{COLL}}(\mathbf{x}_t).$$

That is, the cost consists of an obstacle-independent task term $\mathcal{C}_{\text{TASK}}(\mathbf{x}_t, \mathbf{u}_t)$, which might include, for instance, flying to a desired position or in a desired direction, as well as an obstacle-dependent collision cost, which is given by the product of an indicator for collision $\mathbb{1}_{\text{COLL}(\mathbf{x}_t, \mathcal{M})}$, which is the only term that depends on the environment, and a collision cost $\mathcal{C}_{\text{COLL}}(\mathbf{x}_t)$ that may, for instance, penalize high-speed collisions more than relatively harmless low-speed collisions.

In a fully observable environment where $\mathcal{M}$ is known, the collision indicator can be evaluated exactly, and the problem can be solved by a standard optimal control method, such as the receding-horizon model-predictive control (MPC) approach we use in this work. In receding-horizon MPC, the robot solves an optimal control problem of the form

$$\min_{\mathbf{u}_t, \ldots, \mathbf{u}_{t+H}} \sum_{h=0}^{H} \mathcal{C}(\mathbf{x}_{t+h}, \mathbf{u}_{t+h}, \mathcal{M}) \text{ s.t. } \mathbf{x}_{t+h+1} = f(\mathbf{x}_{t+h}, \mathbf{u}_{t+h})$$

at each time step, it executes the action $\mathbf{u}_t$, advances to time step $t+1$, and repeats the optimization, effectively performing replanning at each time step. In this work, we assume that the dynamics, which might correspond, for instance, to the equations of motion of a quadrotor, are known at least approximately in advance. We instead focus on estimation of the cost, which depends on the unknown environment $\mathcal{M}$. If the environment is unknown and the indicator $\mathbb{1}_{\text{COLL}(\mathbf{x}_t, \mathcal{M})}$ cannot be estimated exactly, we can attempt instead to evaluate the probability of a collision using sensor observations, such as LIDAR or camera images. In this case, we can approximate the collision indicator according to

$$\mathbb{1}_{\text{COLL}(\mathbf{x}_{t+h}, \mathcal{M})} \approx P(\text{COLL}_{t+h} | \mathbf{x}_t, \mathbf{u}_{t:t+h} \mathbf{o}_t).$$

That is, we can estimate the probability of collision at a future time step $t+h$ based on the current state $\mathbf{x}_t$, the sequence of actions $\mathbf{u}_{t:t+h}$ that we intend to take, and the current observation $\mathbf{o}_t$, which might be used to deduce where the obstacles are located and thereby estimate the probability of collision, without prior knowledge about the environment.

In practice, we will slightly simplify the problem by predicting the probability of a collision at *any* time step h within the MPC horizon H. This approximation is not required, but yields a somewhat simpler model that we found performed equally well in practice, especially for relatively short-horizon MPC problems where $\mathcal{C}_{\text{COLL}}(\mathbf{x}_{t+H})$ doesn't change much over the MPC horizon. In this case, the full approximate cost at time $t+h$ evaluated using observation at time step t is given by

$$\mathcal{C}(\mathbf{x}_{t+h}, \mathbf{u}_{t+h}) \approx \mathcal{C}_{\text{TASK}}(\mathbf{x}_{t+h}, \mathbf{u}_{t+h}) + P_\theta(\text{COLL} | \mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t) \mathcal{C}_{\text{COLL}}(\mathbf{x}_{t+H}),$$

where we parameterize the probability of collision by model parameters θ, which corresponds to a class of parameteric conditional models. In our case, we present $P_\theta(\text{COLL} | \mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$ with a neural network that outputs the parameter of a Bernoulli random variable, as we will discuss in Section 8.3. Our goal now is to learn the probability of collision model P_θ in such a way that avoids catastrophic failures (i.e., high-speed collisions) at both training and test time. However, for the robot to be able to act appropriately in novel situations, the robot must be able to reason about the uncertainty of the collision prediction model P_θ, as we will discuss in the next section.

8.3 Uncertainty-aware Collision Prediction

The core component of our approach is an uncertainty-aware collision prediction model P_θ. Training this collision prediction model from experience presents a dilemma: the robot must first experience collisions in order to learn how to avoid collisions. We formulate a speed-dependent collision cost that uses uncertainty-aware collision estimates, resulting in the robot exploring cautiously when uncertainty is high and moving faster when uncertainty is low. This naturally arising behavior enables the robot to learn about collisions without experiencing catastrophic failures, and subsequently use these safe collision experiences to act more aggressively in the future.

An example application domain and desired application of the uncertainty-aware collision prediction model is the following: consider a quadrotor navigation task in which the objective is to fly fast and avoid collisions in an unknown environment. The quadrotor seeks to learn a collision prediction model that takes as input an image and a sequence of velocity commands and outputs the probability of collision. Initially, the quadrotor flies conservatively because the speed-dependent collision cost favors low-speed actions due to high uncertainty estimates of the collision prediction model. While flying conservatively, the quadrotor experiences safe collisions. These safe collisions, coupled with the associated images, are used to train the collision prediction model; the collision prediction model then learns how to associate images and velocity commands with the likelihood of colliding. As the algorithm continues and the collision prediction model uncertainty becomes low enough, the speed-dependent collision cost will favor high-speed flight.

Collision Prediction with Uncertainty

The collision prediction model P_θ takes as input the current state $\mathbf{x}_t$ and observation $\mathbf{o}_t$, a sequence of H controls $\mathbf{u}_{t:t+H}$, and outputs the probability the robot experiences a collision within the horizon. We formulate P_θ as a discriminative model using the logistic function $\mathrm{L}(y) = 1/(1+\exp(-y))$, so that

$$P_\theta(\text{COLL}|\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t) = \mathrm{L}\big(\mathbb{E}[f_\theta(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)]\big).$$

Here, $f_\theta(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$ is a random variable that corresponds to the real-valued output of our stochastic discriminatively trained model, which in our case corresponds to a modified neural network model that can produce uncertainty estimates. In general a variety of alternative models, including stochastic Bayesian models, could be used. Under this model, we can also define a risk-averse collision estimator $\widetilde{P}_\theta(\text{COLL}|\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$, given by

$$\widetilde{P}_\theta(\text{COLL}|\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t) = \mathrm{L}\Big(\mathbb{E}[f_\theta(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)] + \lambda_{\text{STD}}\sqrt{\mathrm{Var}[f_\theta(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)]}\Big), \tag{8.1}$$

where λ_{STD} is a non-negative user-defined scalar and f_θ is scalar-valued function of the current state, observation, and a sequence of controls.

The risk-averse collision prediction model $\widetilde{P}_\theta$ accounts for uncertainty using the variance of the function f_θ: the larger the variance of f_θ, the less certain the underlying stochastic model is about the probability of collision. The standard model P_θ ignores this uncertainty, while the risk-averse model $\widetilde{P}_\theta$ uses the uncertainty to produce a conservative guess about the collision probability. Note that we use the variance of the sigmoid pre-activation value f_θ, since sigmoid probabilities are always in the range $[0,1]$. Our goal is to increase the conservative estimate of collision if the model f_θ is uncertain (has high variance). However, if we use the sigmoid values, we might systematically underestimate the uncertainty. For example, imagine that the expected value of f_θ is a large negative number. Then, even if the variance is very large, the sigmoid expectation will be zero, which means that the sigmoid variance will be low. This is because the tails of the sigmoid flatten any variance in the model, making it invisible in situations where the mean prediction is close to 0 or 1. The hyperparameter λ_{STD} allows us to set how conservative the risk-averse model $\widetilde{P}_\theta$ should be, which allows the user to make intuitive tradeoffs between safety and task completion.

Velocity-Dependent Collision Cost

Based on the previously defined risk-averse model, we can now formulate a collision cost that will naturally favor slow, cautious exploration in regions of high uncertainty. The particular cost that we use has the form

$$\mathcal{C}_{\text{COLL}}(\mathbf{x}_t) = \lambda_{\text{COLL}} \|\text{VEL}_t\|^2, \tag{8.2}$$

where VEL_t is the robot velocity at time t and λ_{COLL} is a non-negative user-defined scalar that weights the relative importance of $\mathcal{C}_{\text{COLL}}$ versus $\mathcal{C}_{\text{TASK}}$. The full cost is then approximated using the risk-averse collision prediction model, according to

$$\mathcal{C}(\mathbf{x}_{t+H}, \mathbf{u}_{t+H}) \approx \mathcal{C}_{\text{TASK}}(\mathbf{x}_{t+H}, \mathbf{u}_{t+H}) + \widetilde{P}_\theta(\text{COLL}|\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)\mathcal{C}_{\text{COLL}}(\mathbf{x}_{t+H}), \tag{8.3}$$

With $\widetilde{P}_\theta$ and $\mathcal{C}_{\text{COLL}}$ defined, let us now confirm that Eqn. 8.3 will naturally favor cautious behavior when the collision prediction model is uncertain, and favor more aggressive behavior when the collision prediction model is confident. If the risk-averse collision prediction probability $\widetilde{P}_\theta$ is large, the robot is encouraged to move slowly in order to minimize $\mathcal{C}_{\text{COLL}}$. The collision prediction probability P_θ is large when $\mathbb{E}[f_\theta] + \sqrt{\text{Var}[f_\theta]}$ is large, which occurs whenever the model predicts a collision (i.e., $\mathbb{E}[f_\theta] \gg 0$) or when the model is uncertain (i.e., $\text{Var}[f_\theta] \gg 0$). On the other hand, if the risk-averse collision prediction probability is small, corresponding to a confident no-collision prediction, the robot can focus on minimizing $\mathcal{C}_{\text{TASK}}$ and move at fast speeds. The collision prediction probability $\widetilde{P}_\theta$ is small when $\mathbb{E}[f_\theta] + \sqrt{\text{Var}[f_\theta]}$ is small, which occurs when the model predicts no collision (i.e., $\mathbb{E}[f_\theta] \ll 0$) and the model is certain (i.e., $\text{Var}[f_\theta] \approx 0$).

Neural Network Collision Prediction Model

In order to be able to predict collisions from rich, high-dimensional sensory inputs, such as cameras or LIDAR measurements, we will use deep neural networks to estimate the probability of a collision. In the case of a standard deterministic, discriminatively trained neural network, f_θ would represent the pre-activation values in the network at the last layer, while P_θ is obtained by applying a sigmoidal nonlinearity to the pre-activations. Such a network can be trained on prior trajectories experienced by the robot simply by slicing all prior data into subsequences of length H, and inputting the states $\mathbf{x}_t$, observations $\mathbf{o}_t$, and the concatenated sequence of controls $\mathbf{u}_{t:t+H}$ into the model. The probability of collision labels are binary values recorded by the robot indicating whether a collision occurred, and we can obtain the label for each subsequence simply by checking whether a collision occurred between time steps t and $t + H$. The network can then be trained using standard stochastic gradient descent (SGD) with a cross-entropy loss on the final sigmoid output.

While such a model can provide accurate predictions about collision probability in regions of the environment close to the training data, it is inherently discriminative and deterministic. Such a deterministic model does not provide an estimate of its variance, and therefore is not by itself suitable for risk-averse collision prediction.

Estimating Uncertainty with Neural Networks

Standard predictive neural network models are trained discriminatively, which means that, even though the network might achieve a high accuracy on samples drawn from the same distribution as the training data, it is very difficult to predict how the network would behave on data drawn from a different distribution. While it is possible to train a neural network model that outputs a mean and a variance as its prediction [78], this model is not in general guaranteed to output high variances for unfamiliar inputs because the network is by definition trained only on the datapoints that are in the training set. Indeed, such a method for estimating variance is only effective at estimating the inherent noise in the data, and the variance estimates are not a meaningful indication of the model's own uncertainty about its predictions. To produce accurate uncertainty estimates for data that is outside of the training distribution, we must explore techniques that go beyond direct discriminative training. In order to obtain accurate uncertainty estimates from our model, we use two techniques: bootstrapping and dropout.

Bootstrapping: Bootstrapping [154], [167] is a simple and effective method of estimating model uncertainty using resampling that can be used with any discriminatively trained model. Given a dataset $\mathcal{D}$, B new datasets $\mathcal{D}^{(b)}$ are sampled with replacement from $\mathcal{D}$ such that $|\mathcal{D}^{(b)}| = |\mathcal{D}|$. Then, instead of training a single model $\mathcal{M}$ on the entire dataset $\mathcal{D}$, B different models $\mathcal{M}^{(b)}$ are trained on the datasets $\mathcal{D}^{(b)}$. The output prediction and uncertainty estimates are the sample mean and standard deviation of the outputs from the population of models.

The intuition behind bootstrapping is that, by generating multiple populations (using sampling with replacement) and training one model per population, the models will agree in high-density areas of the population (i.e., low uncertainty regions) and disagree in low-density areas of the population (i.e., high uncertainty regions). This intuition is backed with theoretical guarantees [168]. However, for time- and resource-constrained applications such as robotics, usually only a limited number of bootstraps can be used, which often leads to inaccurate estimates of the model uncertainty.

Dropout: Dropout [156] is, by comparison, a computationally cheap method to improve uncertainty estimates. Dropout is commonly used to reduce overfitting in neural networks by randomly dropping units from the neural network during training [155]. Specifically, a given unit with dropout is set to 0 with probability p and left as its original value with probability $1 - p$ during training. Dropout prevents units from co-adapting (and thus overfitting) too much because different units are sampled for each forward pass, which effectively samples a new, but related, network during each step of training. Given a neural network NN$^{(b)}$, dropout in effect constructs a new

Algorithm 6 Neural net training with bootstrapping and dropout

```
1: input: dataset $\mathcal{D} = \{\mathbf{x}_t^{(i)}, \mathbf{u}_{t:t+H}^{(i)}, \mathbf{o}_t^{(i)}\}$, neural network model NN
2: for $b = 1$ to $B$ do
3:     Sample a dataset of subsequences $\mathcal{D}^{(b)}$ from the full dataset $\mathcal{D}$ with replacement
4:     Initialize neural network NN$^{(b)}$ with random weights
5:     for number of SGD iterations do
6:         Sample datapoint $(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$ from $\mathcal{D}^{(b)}$
7:         Sample NN$_d^{(b)}$ by masking the units in NN$^{(b)}$ using dropout
8:         Run forward pass on NN$_d^{(b)}$ using $(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$
9:         Run backward pass on NN$_d^{(b)}$ to get gradient $g_d^{(b)}$
10:        Update model NN$^{(b)}$ parameters using $g_d^{(b)}$
11:    end for
12: end for
```

randomized version of this network NN$_d^{(b)}$ by sampling independent Bernoulli random variables to act as masks on each neuron.

When dropout is used to reduce overfitting, it is only applied during training in order to force the units in the network to cope with stochastic removal of other units. In order to achieve high accuracy at test time, the dropout regularization is removed and all network weights are scaled by p to compensate for the increased level of activation. However, Gal and Ghahramani [156] showed that dropout can be used to obtain uncertainty estimates at test time by calculating the sample mean and standard deviation of multiple stochastic forward passes of the neural network *using dropout*. In this way, dropout can be viewed as an economical approximation to an ensemble method (such as bootstrapping) in which each sampled dropout mask corresponds to a different model. However, dropout underestimates the uncertainy because it acts roughly as a variational lower bound [156].

Neural Networks with Bootstrapping and Dropout: Alg. 6 provides an overview of training neural networks with bootstrapping and dropout. From an initial dataset, multiple datasets are resampled with replacement, along with corresponding neural network model instantiations. While performing stochastic gradient descent on each bootstrap, different units are dropped each time a forward pass occurs; the gradient calculated by backpropagation is then used to update that specific bootstrap model's parameters.

At test time, we can evaluate the mean and variance of the ensemble by performing multiple forward passes on each network NN$^{(b)}$ using multiple instantiations of the dropout process, corresponding to NN$_d^{(b)}$. The random function $f_\theta(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$ then corresponds to sampling a network, sampling a dropout process, and evaluating the output. Thus, using neural networks with bootstrapping and dropout, we can estimate $\mathbb{E}[f_\theta(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)]$ and $\mathrm{Var}[f_\theta(\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)]$ for use in the risk-averse model $\widetilde{P}_\theta(\textsc{coll}|\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$.

Algorithm 7 RL with Risk-Averse Collision Estimates

```
1: Initialize empty dataset $\mathcal{D}$
2: Initialize collision prediction model $\widetilde{P}_\theta$
3: for iter=1 to max_iter do
4:     Sample trajectories $\{\tau_i\}$ using MPC with cost $\mathcal{C}$
5:     Add samples $\{\tau_i\}$ to $\mathcal{D}$
6:     Train $\widetilde{P}_\theta$ using $\mathcal{D}$ (Alg. 6)
7: end for
```

Reinforcement Learning with Risk-Averse Collision Estimation

Alg. 7 provides an overview of how the uncertainty-aware collision prediction model is used in a model-based reinforcement learning algorithm. Each iteration of the algorithm, the cost function $\mathcal{C}$ is formed using the current uncertainty-aware collision prediction model $\widetilde{P}_\theta$. The model predictive controller then samples trajectories using cost $\mathcal{C}$. These sample trajectories are aggregated into a dataset containing all previous sampled trajectories. Then $\widetilde{P}_\theta$ is trained on the dataset according to Alg. 6 and the next iteration begins.

8.4 Experiments

We present simulated and real-world experiments to evaluate our uncertainty-aware collision prediction model, as well as our proposed model-based RL algorithm. We compare different settings for the parameters in our model, as well as evaluate its performance against a model-based approach that directly estimates the probability of collision, without explicitly accounting for uncertainty. Videos of the experiments can be found at `https://sites.google.com/site/probcoll/`.

Our collision prediction model $\widetilde{P}_\theta(\textsc{coll}|\mathbf{x}_t, \mathbf{u}_{t:t+H}, \mathbf{o}_t)$ is a fully connected neural network with two layers with 40 ReLU [149] hidden units each. The activation of the last layer, which outputs the collision probability, is a sigmoid (see Eqn. 8.1). The model inputs are the concatenation of $\mathbf{x}_t, \mathbf{u}_{t:t+H}$ and $\mathbf{o}_t$. We trained the network using ADAM [151] and a standard cross-entropy loss. For uncertainty estimation, the simulation experiments used 50 bootstraps and a dropout ratio of 0.2, while the real-world experiments used 5 bootstraps (due to real-time constraints) and a dropout ratio of 0.05.

At each time step, the receding-horizon MPC planner chooses among a set of fixed action sequences of horizon length H by evaluating cost $\mathcal{C}$ on each action sequence, and executes the first action of the minimal cost action sequence.

Quadrotor experiments

The simulated and real-world quadrotors have the same states, controls, and observations. We use a high-level representation of the quadrotor in which the control $\mathbf{u} \in \mathbb{R}^2$ is the commanded planar linear velocity, and therefore we assume the state $\mathbf{x}$ is estimated such that this level of control is feasible. However, we do not provide the state $\mathbf{x}$ as input to the collision prediction model. The observation $\mathbf{o} \in \mathbb{R}^{256}$ is a 16 by 16 grayscale image. The set of action sequences considered by the MPC planner at each time step consists of 190 straight-line, constant-velocity trajectories at various angles and speeds.

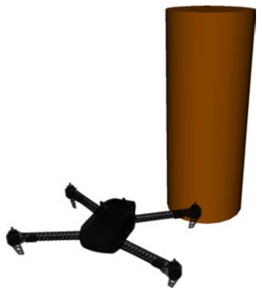


Figure 8.2: Simulation

Simulated quadrotor: We first evaluate our uncertainty-aware collision prediction model in a simulated environment consisting of a cylindrical obstacle of radius 0.2m (Fig. 8.2). The objective $\mathcal{C}_{\text{TASK}}$ is to fly forward at 0.5 m/s, which is encoded as an ℓ^2 norm. The time horizon is $H = 6$ and each discrete time step corresponds to $\delta = 0.2$ seconds, therefore the planning horizon is δH seconds. At each time step, the quadrotor must decide on the sequence of actions using only the observation from a simulated monocular camera.

Fig. 8.3 compares safety versus task performance for different variants of Alg. 7. All experiments consist of 20 training iterations, with each iteration consisting of 20 on-policy rollouts from start states drawn from the same distribution. Each experiment was run 5 times with different random seeds.

First, we investigate the benefits of incorporating uncertainty into the cost by evaluating different values for λ_{STD} (Eqn. 8.1). Fig. 8.3a shows that, when not accounting for uncertainty (i.e., $\lambda_{\text{STD}} = 0$), the final task performance approaches the desired speed of 0.5 m/s. However, the quadrotor experiences high-speed collisions during training, as shown by the vertical axis. By accounting for uncertainty (i.e., $\lambda_{\text{STD}} > 0$), the quadrotor experiences lower speed collisions during training. The final task performance decreases if λ_{STD} is increased too much, which is expected: the more conservative the vehicle behaves during training, the longer it takes to learn the task. These results

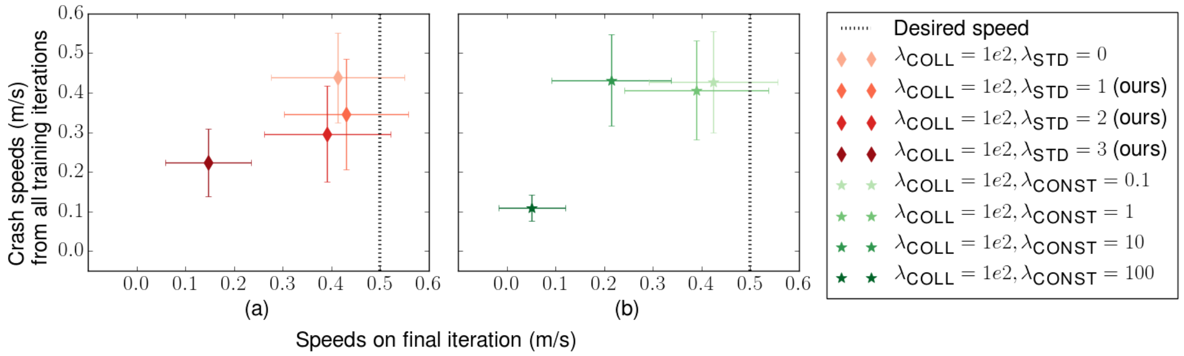


Figure 8.3: **Comparison of safety versus task performance in simulation**: We investigated the effect of changing parameters in Eqn. 8.1 on crash speeds experienced during all training iterations (y-axis) versus the desired objective of flying forward at 0.5 m/s (x-axis). (a) shows the effect of changing λ_{STD} for our uncertainty-aware approach. (b) shows the effect of changing λ_{CONST} for a conservative baseline, in which the uncertainty in Eqn. 8.1 is replaced by a constant. Compared to the conservative baseline approach in (b), (a) shows our uncertainty-aware approach and its parameters can effectively trade off between safety and performance.

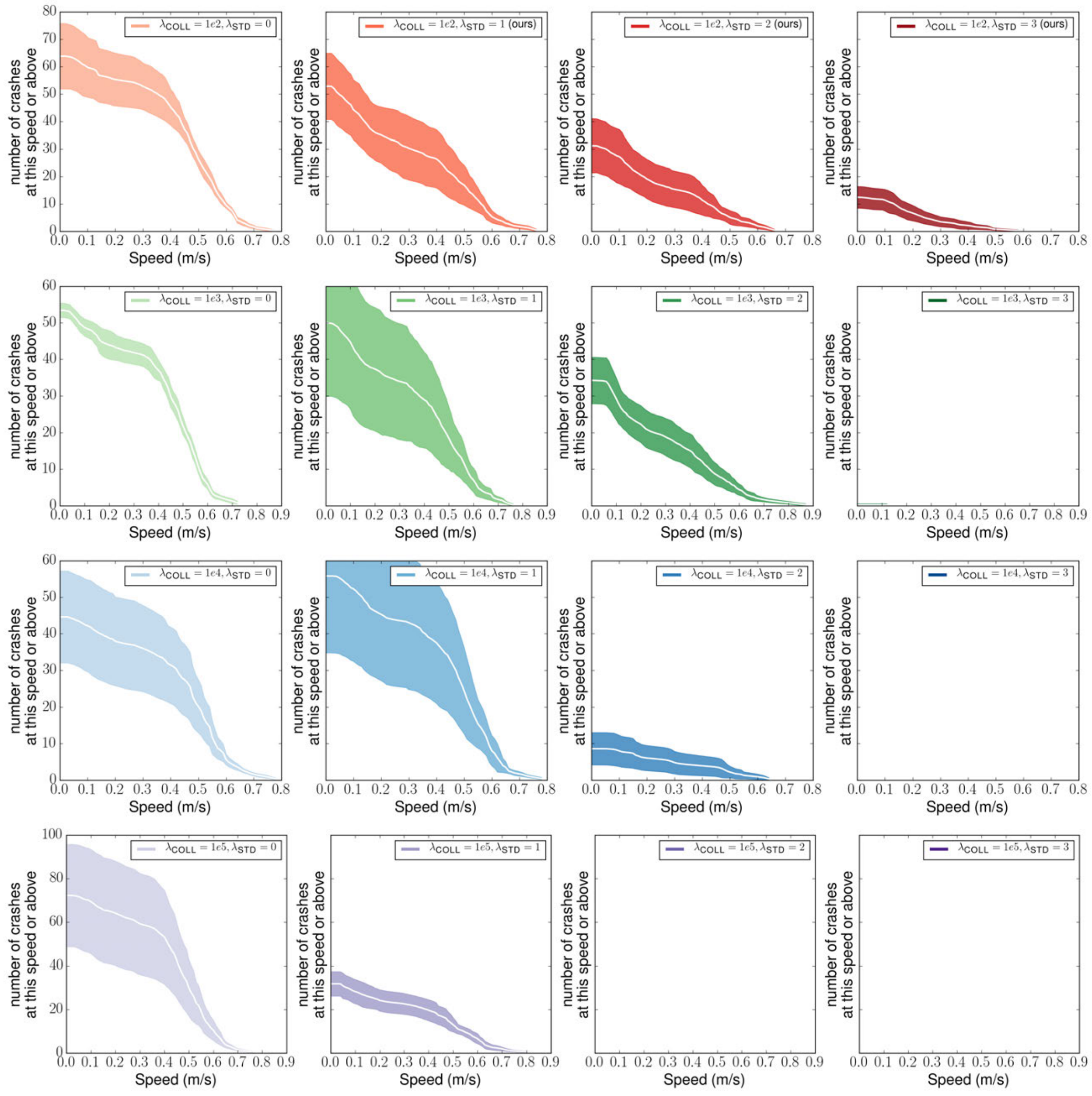


Figure 8.4: **Safety comparison for different values of λ_{COLL} and λ_{STD}**: Each plot shows the number of training crashes at a given speed or above for a specific setting of λ_{COLL} and λ_{STD} averaged over 5 experiments. Each row corresponds to a fixed value for λ_{COLL} and each column corresponds to a fixed value for λ_{STD}. Examining the rows show that increasing λ_{STD} leads to fewer collisions, and of the collisions that do occur they are at lower speed. Examining the first column shows that increasing λ_{COLL} and not accounting for uncertainty does not lead to fewer collisions.

show that λ_{STD} allows the user to control their desired degree of risk during training and trade off safety against learning efficiency.

One reasonable question is whether accounting for uncertainty improves safety due to good uncertainty estimates, or simply because adding uncertainty to the collision probability simply makes the vehicle more cautious by penalizing high speeds. To answer this question, we compare our uncertainty-aware approach against a conservative baseline that replaces the uncertainty in Eqn. 8.1 with a constant λ_{CONST} (Fig. 8.3b). The experiments for $\lambda_{\text{CONST}} = 0.1, 1$, and 10 show no safety improvement, and also show decreased task performance compared to the baseline $\lambda_{\text{CONST}} = 0 \equiv \lambda_{\text{STD}} = 0$. The experiment for $\lambda_{\text{CONST}} = 100$ shows substantial safety improvement, but task performance is also substantially diminished. Compared to our uncertainty-aware approach with different settings of λ_{STD}, the baseline constant penalty approach with λ_{CONST} is ineffective at trading off between safety and performance, and always produces overly conservative motions. This indicates that uncertainty estimation is in fact reasoning about the vehicle's surroundings, rather than uniformly encouraging slower flight.

Another reasonable question to ask is whether simply increasing the collision cost λ_{COLL} induces safer training behavior. Fig. 8.4 shows that increasing λ_{COLL} does not lead to safer training behavior, while increasing λ_{STD} does lead to safer training behavior.

Real-world quadrotor: We evaluated our approach in a real-world environment consisting of a single obstacle, in which the objective is to fly around the obstacle (Fig. 8.1). Although the task of avoiding a single static obstacle is relatively simple, it is worth noting that the vehicle must perform this task entirely using real-world training data and only monocular images, while minimizing the number of collisions experienced during training. As such, the task is in fact quite challenging.

We ran our experiments using a Parrot Bebop 2 quadrotor. We used the ROS bebop_autonomy package, which allows the laptop to send linear velocity commands and receive the onboard images in real-time. The quadrotor's objective $\mathcal{C}_{\text{TASK}}$ is to fly forward at 1.6 m/s, which is encoded as an ℓ^2

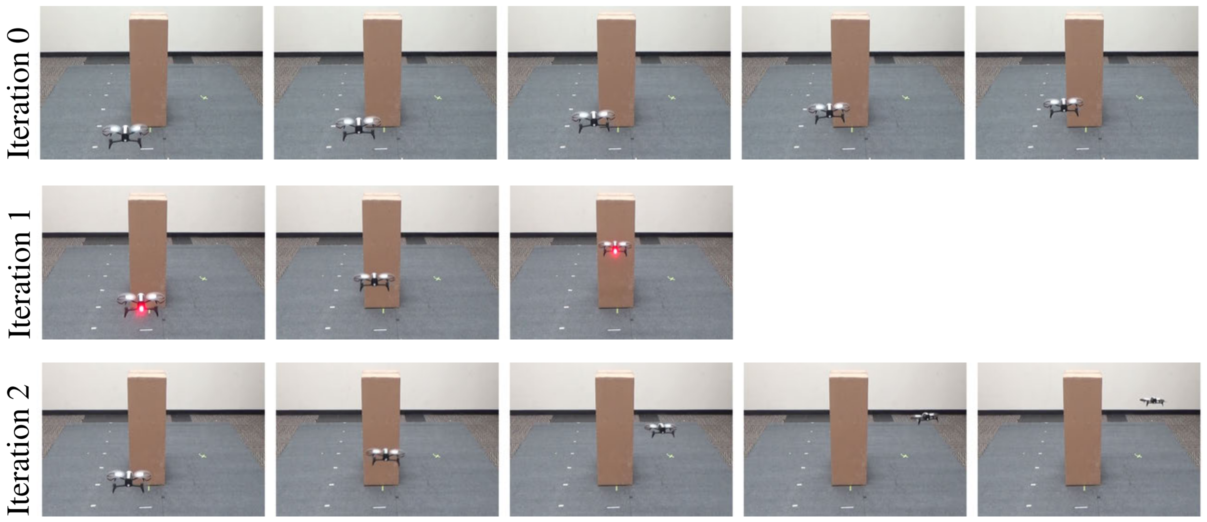


Figure 8.5: **Real-world quadrotor experiments**: A Parrot Bebop 2 quadrotor learns to fly while avoiding obstacles using our uncertainty-aware reinforcement learning for collision avoidance algorithm (Alg. 7). Sample trajectories from the RL algorithm are shown above. On iteration 0, the quadrotor does not collide with the obstacle, but flies slowly. On iteration 1, the quadrotor flies faster, but collides with the obstacle. On iteration 2, the quadrotor avoids the obstacle while flying at high speed.

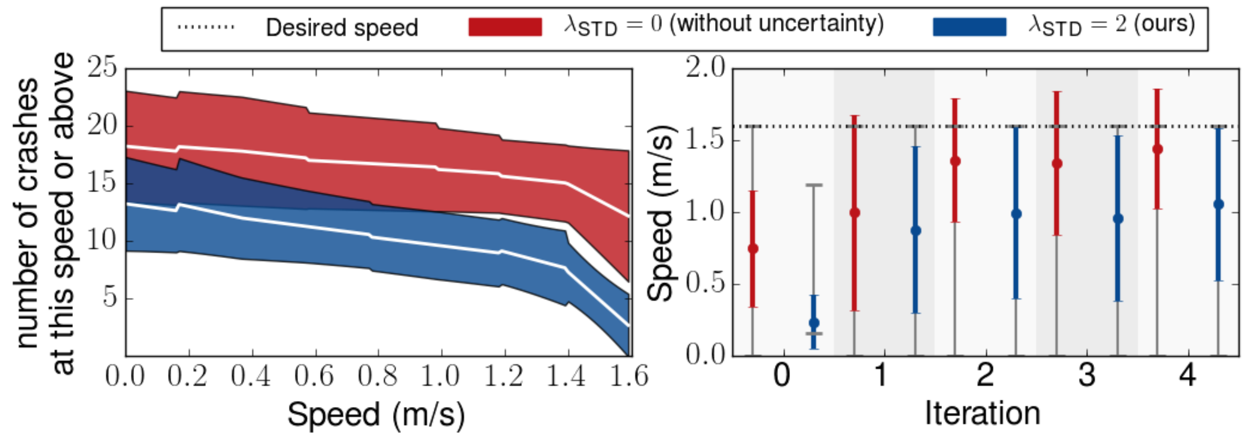


Figure 8.6: **Comparison of safety versus task performance on a real-world quadrotor**: We investigated the effect of changing λ_{STD} in Eqn. 8.1 on crash speeds experienced during all training iterations (left) versus the desired objective of flying forward at 1.6 m/s (right) on a Bebop 2 quadrotor. For each value of λ_{STD}, results are combined from 5 complete experiments with the left plot displaying the mean and std and the right plot displaying the mean, std, and max/min. Increasing λ_{STD} leads to fewer crashes (left), but suboptimal performance (right).

norm. The time horizon $H = 3$ and each time step corresponds to $\delta = 0.5$ seconds.

All experiments consist of 5 training iterations, with each iteration consisting of 5 rollouts from 4 different initial positions. This experimental setup can be viewed in the online video. After each rollout, the quadrotor was manually reset to the next initial state. Note that this reset was solely done for minimizing experimental confounds for the purpose of evaluation, and is not a requirement of our approach. In principle, the vehicle could simply continue flying around the room and collecting data until good performance is achieved. Each experiment was initialized with 6 flight demonstrations provided by a human pilot. These demonstrations were the exact same for all experiments and consisted of 2 crashes and 4 successful flights around the obstacle. To prevent damage to the quadrotor, particularly for the baselines, a human pilot intervened if a crash was imminent; the algorithm therefore treated each intervention as a collision. Each experiment was run 5 times.

Fig. 8.5 shows images of our approach during the training process for an example experiment. In the beginning iterations, the quadrotor makes little progress and experiences collisions. As the RL algorithm progresses, the quadrotor is eventually able to fly around the obstacle at high speed.

Fig. 8.6 compares safety versus task performance when running our model-based RL algorithm (Alg. 7) without uncertainty ($\lambda_{\text{STD}} = 0$) and with uncertainty ($\lambda_{\text{STD}} = 2$). When accounting for uncertainty, the quadrotor experiences substantially fewer collisions, especially at higher speeds, but takes longer to approach the desired task performance.

Real-world RC car experiments

We evaluated our approach on an RC car (Fig. 8.7) in a simple obstacle avoidance task (Fig. 8.1). The car is parameterized by control $\mathbf{u} \in \mathbb{R}^2$ consisting of speed and steering angle and observation $\mathbf{o} \in \mathbb{R}^{576}$ consisting of a 32 by 18 grayscale image. We do not assume access to any underlying state $\mathbf{x}$.

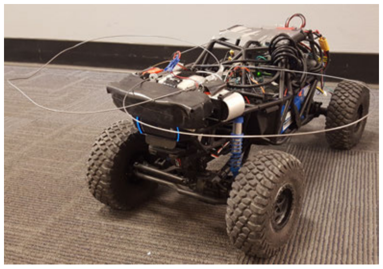


Figure 8.7: 1/10th scale RC car with a Logitech C920 Webcam and limit switch collision detectors.

The car's objective $\mathcal{C}_{\text{TASK}}$ is to drive at 1.2 m/s in any direction, which is encoded as an ℓ^2-norm. The time horizon was set to $H = 4$ and each discrete time step corresponds to $\delta = 0.5$ seconds. The set of action sequences considered by the MPC planner at each time step consists of 49 curving, constant-velocity trajectories at various steering angles and speeds.

All experiments consist of 10 training iterations, with each iteration consisting of 5 on-policy rollouts from 4 different initial states. Each rollout ended after either a collision or 10 time steps, therefore each experiment consists of approximately 15 minutes of real-world experience. After each rollout, the car was manually reset to the next initial state. No human demonstrations were used for initialization and each experiment was ran twice. Unlike in the quadrotor experiments, the car was allowed to collide at full speed and automatically registered collisions using limit switches mounted on the front of the car.

Fig. 8.8 shows images of our approach during the training process for an example experiment. Initially, the car is unable to avoid the obstacle and side walls, but eventually learns to avoid collisions.

Fig. 8.9 compares safety versus task performance when running our model-based RL algorithm (Alg. 7) without uncertainty ($\lambda_{\text{STD}} = 0$) and with uncertainty ($\lambda_{\text{STD}} = 1$). The final model-based planner for both approaches succeeds in navigating without colliding for almost 70% of the rollouts, which is a significant improvement over the initial policy. When accounting for uncertainty, the car experiences fewer high-speed collisions and achieves comparable speeds compared to when not accounting for uncertainty.

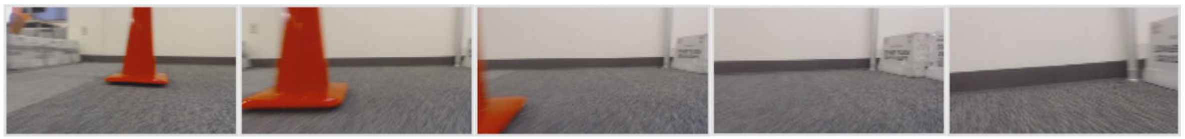


Figure 8.8: **Real-world RC car experiments**: An RC car learns to drive while avoiding obstacles using our uncertainty-aware reinforcement learning for collision avoidance algorithm (Alg. 7). A successful rollout is shown above.

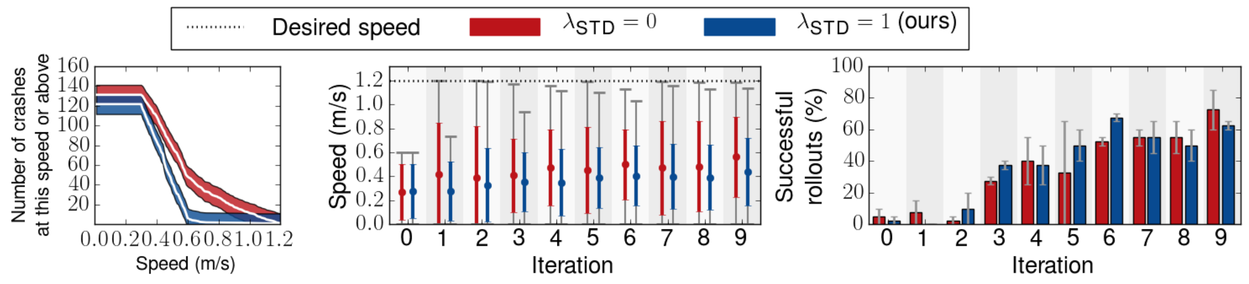


Figure 8.9: **Comparison of safety versus task performance on a real-world RC car**: We investigated the effect of changing λ_{STD} in Eqn. 8.1 on crash speeds experienced during all training iterations (left) versus the task objective of driving at 1.2 m/s (middle) and the percentage of rollouts in which the RC car reached the end of the track (right). For each value of λ_{STD}, results are combined from 2 complete experiments. Our uncertainty-aware approach ($\lambda_{\text{STD}} = 1$) experiences 13% fewer crashes at speeds above 0.6 m/s (left) and comparable task performance (middle/right) compared to the baseline approach which does not account for uncertainty.

8.5 Discussion

We presented a model-based combined perception and control method for learning obstacle avoidance strategies that uses uncertainty estimates to automatically generate safe strategies. Our method is based on predicting the probability of collision conditioned on raw sensory inputs and a sequence of actions, using deep neural networks. This predictor can be used within a model-predictive control pipeline to choose actions that avoid collisions with high probability. In regions of high uncertainty, our risk-averse cost function naturally causes the robot to revert to a cautious low-speed strategy, without any explicit manual engineering of safety controllers or fail-safe mechanisms. We demonstrate our approach is safer compared to methods without uncertainty estimates in both a simulated and real-world quadrotor obstacle avoidance task, as well as a real-world RC car task.

Chapter 9

Conclusion

In this thesis, we developed learning algorithms for mobile robots to address three main challenges: sample-efficiency, supervision, and safety. To address sample-efficiency, we designed a flexible framework for interpolating between model-based and model-free methods (Chapter 2) and investigated how to leverage simulation in order to accelerate real world learning (Chapter 3). To address supervision, we developed algorithms that provide supervision using self-supervision (Chapter 4), other learned models (Chapter 5), and weak human supervision (Chapter 6). To address safety, we investigated methods that leveraged experts (Chapter 7) and uncertainty (Chapter 8).

Although we believe the work in this thesis and other ongoing work has led to much progress on real-world mobile robot learning, there is still much more work to be done. We discuss some open questions and future directions below.

Sample-efficiency. While developing future algorithms that require even less data to successfully learn is a relevant and fruitful endeavor, the notion of sample-efficiency itself depends on the specific task. This is perhaps one of the most challenging aspects of robot learning: not knowing ahead of time how much data the robot will need to gather in order to succeed at a specific task. For robot learning practitioners, knowing how much and what type of data the robot will need to gather in order to succeed would drastically change how robot learning algorithms are developed and deployed.

A related challenge for future work is how to learn from long-tailed distributions. These long-tail distributions are particularly relevant for mobile robots due to the diversity of the open world. However, by the very nature of the rarity of these long-tail events, it is challenging to study and develop methods to address the long tail. This challenge is further exacerbated for academic researchers, who have limited resources. Creating a unified definition, framework, and benchmark for addressing the long tail could significantly advance mobile robot learning.

Supervision. Although we showed that self-supervision (Chapter 4), learned models (Chapter 5), and weak human supervision (Chapter 6) can all provide supervision for mobile robot learning, the methods presented in this thesis were all offline methods—the robot paused data collection in order to train a new model on the collected data. Online methods, which adapt the model during data collection, can be advantageous for many reasons, including for sample-efficiency and for safety, but also introduce additional challenges, such as instability and catastrophic forgetting.

While self-supervision (Chapter 4) and weak human supervision (Chapter 6) are powerful approaches for learning core mobile robot navigation policies, we believe that model supervision (Chapter 5) is the best path forward for learning more complex navigation strategies. However, the main challenge with leveraging learned models for supervision is that these models are fallible, and therefore the source of supervision is imperfect. We believe that investigating how to learn from imperfect models is an important avenue for future work.

Safety. Although we believe expert supervision (Chapter 7) is appropriate for some mobile robot learning, we believe leveraging uncertainty (Chapter 8) is typically a more general and viable option. However, conservative reinforcement learning methods are critically dependent on the accuracy of the uncertainty estimates, and while there has been much progress on improving uncertainty estimation for high-dimensional models such as neural networks, there is still much progress to be made.

One of the main challenges to achieve accurate uncertainty estimation is that not all sources of uncertainty are equivalent. For example, while a robot may be equally uncertain if placed near a never-before-seen field or cliff, the uncertainty estimate should encourage exploration in the field but discourage exploration near the cliff. We believe it is important to develop safe robot learning algorithms that can disambiguate these various forms of uncertainty.

* * *

We hope the work presented in this thesis serves as a stepping stone for continuing work on enabling mobile robots to learn and navigate in complex, real-world environments.

Bibliography

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2008.

[2] C Rosen and N Nilsson, “Application of intelligent automata to reconnaissance,” SRI, Tech. Rep., 1968.

[3] R. S. Wallace, A. Stentz, C. E. Thorpe, H. P. Moravec, W. Whittaker, and T. Kanade, “First Results in Robot Road-Following,” in *IJCAI*, 1985.

[4] C. Thorpe, M. H. Hebert, T. Kanade, and S. A. Shafer, “Vision and navigation for the Carnegie-Mellon Navlab,” *TPAMI*, 1988.

[5] J. Leonard and H. F Durrant-Whyte, “Simultaneous Map Building and Localization for an Autonomous Mobile Robot,” in *IROS*, 1991, pp. 1442–1447.

[6] C. Urmson and et. al., “Autonomous driving in urban environments: Boss and the urban challenge,” *Journal of Field Robotics*, 2008.

[7] J. P. How, B. Behihke, A. Frank, D. Dale, and J. Vian, “Real-time indoor autonomous vehicle test environment,” *IEEE Control Systems Magazine*, 2008.

[8] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, *et al.*, “Junior: The stanford entry in the urban challenge,” *JFR*, 2008.

[9] P. Furgale and T. D. Barfoot, “Visual teach and repeat for long-range rover autonomy,” *Journal of Field Robotics*, 2010.

[10] S. Shen, N. Michael, and V. Kumar, “Autonomous multi-floor indoor navigation with a computationally constrained MAV,” in *ICRA*, 2011.

[11] L. Paull, J. Tani, H. Ahn, J. Alonso-Mora, L. Carlone, M. Cap, Y. F. Chen, C. Choi, J. Dusek, Y. Fang, *et al.*, “Duckietown: an open, inexpensive and flexible platform for autonomy education and research,” in *ICRA*, 2017.

[12] T. Ort, L. Paull, and D. Rus, “Autonomous vehicle navigation in rural environments without detailed prior maps,” in *ICRA*, 2018.

[13] K. Mohta, K. Sun, S. Liu, M. Watterson, B. Pfrommer, J. Svacha, Y. Mulgaonkar, C. J. Taylor, and V. Kumar, “Experiments in fast, autonomous, gps-denied quadrotor flight,” in *ICRA*, 2018.

[14] A. J. Barry, P. R. Florence, and R. Tedrake, “High-speed autonomous obstacle avoidance with pushbroom stereo,” in *Journal of Field Robotics*, 2018.

[15] J. Fuentes-Pacheco, J. Ruiz-Ascencio, and J. M. Rendon-Mancha, “Visual simultaneous localization and mapping: a survey,” *Artificial Intelligence Review*, 2015.

[16] D. A. Pomerleau, “Alvinn: An autonomous land vehicle in a neural network,” in *NIPS*, 1989.

[17] D. Barnes, W. Maddern, and I. Posner, “Find your own way: Weakly-supervised segmentation of path proposals for urban autonomy,” in *ICRA*, 2017.

[18] R. Hadsell, P. Sermanet, J. Ben, A. Erkan, M. Scoffier, K. Kavukcuoglu, U. Muller, and Y. LeCun, “Learning long-range vision for autonomous off-road driving,” *Journal of Field Robotics*, 2009.

[19] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, “Deepdriving: Learning affordance for direct perception in autonomous driving,” in *ICCV*, 2015.

[20] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, “End to end learning for self-driving cars,” in *ArXiv:1604.07316*, 2016.

[21] C. Richter and N. Roy, “Safe Visual Navigation via Deep Learning and Novelty Detection,” in *RSS*, 2017.

[22] M. Deisenroth and C. Rasmussen, “A Model-Based and Data-Efficient Approach to Policy Search,” in *ICML*, 2011.

[23] J. Peters and S. Schaal, “Policy Gradient Methods for Robotics,” in *IROS*, 2006.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and R. M., “Playing Atari with Deep Reinforcement Learning,” in *Workshop on Deep Learning, NIPS*, 2013.

[25] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous Control with Deep Reinforcement Learning,” in *ICRL*, 2016.

[26] M. P. Deisenroth, R. Calandra, A. Seyfarth, and J. Peters, “Toward fast policy search for learning legged locomotion,” in *IROS*, 2012.

[27] M. E. Taylor and P. Stone, “Transfer learning for reinforcement learning domains: A survey,” *JMLR*, 2009.

[28] S. Daftry, J. A. Bagnell, and M. Hebert, “Learning transferable policies for monocular reactive mav control,” in *ISER*, 2016.

[29] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, *et al.*, “Using simulation and domain adaptation to improve efficiency of deep robotic grasping,” *ArXiv preprint arXiv:1709.07857*, 2017.

[30] F. Zhang, J. Leitner, M. Milford, and P. Corke, “Modular deep q networks for sim-to-real transfer of visuo-motor policies,” in *ACRA*, 2017.

[31] A. Ghadirzadeh, A. Maki, D. Kragic, and M. Björkman, "Deep predictive policy training using reinforcement learning," in *IROS*, 2017.

[32] J. Fu, S. Levine, and P. Abbeel, "One-shot learning of manipulation skills with online dynamics adaptation and neural network priors," in *IROS*, 2016.

[33] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-real robot learning from pixels with progressive nets," in *CoRL*, 2017.

[34] D. Rastogi, I. Koryakovskiy, and J. Kober, "Sample-efficient reinforcement learning via difference models," in *Machine Learning in Planning and Control of Robot Motion Workshop at ICRA*, 2018.

[35] J. P. Hanna and P. Stone, "Grounded action transformation for robot learning in simulation.," in *AAAI*, 2017.

[36] K. Lowrey, S. Kolev, J. Dao, A. Rajeswaran, and E. Todorov, "Reinforcement learning for non-prehensile manipulation: Transfer from simulation to physical system," in *SIMPAR*, 2018.

[37] F. Xia, A. R. Zamir, Z. He, A. Sax, J. Malik, and S. Savarese, "Gibson env: Real-world perception for embodied agents," in *CVPR*, 2018.

[38] J. Zhang, L. Tai, Y. Xiong, M. Liu, J. Boedecker, and W. Burgard, "Vr-goggles for robots: Real-to-sim domain adaptation for visual control," *ArXiv preprint arXiv:1802.00265*, 2018.

[39] F. Sadeghi and S. Levine, "(CAD)2 RL: Real Single-Image Flight without a Single Real Image," *RSS*, 2017.

[40] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel, "Asymmetric actor critic for image-based robot learning," in *RSS*, 2018.

[41] I. Mordatch, K. Lowrey, and E. Todorov, "Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids," in *IROS*, 2015.

[42] A. Rajeswaran, S. Ghotra, B. Ravindran, and S. Levine, "Epopt: Learning robust neural network policies using model ensembles," in *ICLR*, 2017.

[43] W. Yu, J. Tan, C. K. Liu, and G. Turk, "Preparing for the unknown: Learning a universal policy with online system identification," in *RSS*, 2017.

[44] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *ICRA*, 2018.

[45] S. J. Pan, Q. Yang, *et al.*, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, 2010.

[46] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "Decaf: A deep convolutional activation feature for generic visual recognition," in *ICML*, 2014.

[47] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: An astounding baseline for recognition," in *CVPR*, 2014.

[48] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" In *NIPS*, 2014.

[49] S. Hinterstoisser, V. Lepetit, P. Wohlhart, and K. Konolige, "On pre-trained image features and synthetic images for deep learning," *ArXiv preprint arXiv:1710.10710*, 2017.

[50] H. Fu, M. Gong, C. Wang, K. Batmanghelich, and D. Tao, "Deep ordinal regression network for monocular depth estimation," in *CVPR*, 2018.

[51] J.-R. Chang and Y.-S. Chen, "Pyramid stereo matching network," in *CVPR*, 2018.

[52] Y. Wang, W.-L. Chao, D. Garg, B. Hariharan, M. Campbell, and K. Q. Weinberger, "Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving," in *CVPR*, 2019.

[53] A. Valada, J. Vertens, A. Dhall, and W. Burgard, "Adapnet: Adaptive semantic segmentation in adverse environmental conditions," in *ICRA*, 2017.

[54] N. Hirose, A. Sadeghian, M. Vázquez, P. Goebel, and S. Savarese, "Gonet: A semi-supervised deep learning approach for traversability estimation," in *IROS*, 2018.

[55] S. Ross, N. Melik-Barkhudarov, K. S. Shankar, A. Wendel, D. Dey, J. A. Bagnell, and M. Hebert, "Learning monocular reactive uav control in cluttered natural environments," in *ICRA*, 2013.

[56] F. Codevilla, M. Müller, A. López, V. Koltun, and A. Dosovitskiy, "End-to-end Driving via Conditional Imitation Learning," in *ICRA*, 2018.

[57] J. Michels, A. Saxena, and A. Y. Ng, "High speed obstacle avoidance using monocular vision and reinforcement learning," in *ICML*, 2005.

[58] G. Kahn, A. Villaflor, P. Abbeel, and S. Levine, "Composable Action-Conditioned Predictors: Flexible Off-Policy Learning for Robot Navigation," in *CoRL*, 2018.

[59] L. Wellhausen, A. Dosovitskiy, R. Ranftl, K. Walas, C. Cadena, and M. Hutter, "Where should I walk? Predicting terrain properties from images via self-supervised learning," *RA-L*, 2019.

[60] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *IROS*, 2017.

[61] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena, "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots," in *ICRA*, 2017.

[62] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. L. Cun, "Off-road obstacle avoidance through end-to-end learning," in *NIPS*, 2006.

[63] S. Ross, G. J. Gordon, and J. A. Bagnell, "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning," in *Proceedings of the 14th AISTATS*, 2011.

[64] M. Wulfmeier, D. Z. Wang, and I. Posner, "Watch this: Scalable cost-function learning for path planning in urban environments," in *IROS*, 2016.

[65] T. Schaul, D. Horgan, K. Gregor, and D. Silver, “Universal value function approximators,” in *ICML*, 2015.

[66] A. Dosovitskiy and V. Koltun, “Learning to act by predicting the future,” in *ICLR*, 2017.

[67] C. Devin, P. Abbeel, T. Darrell, and S. Levine, “Deep Object-Centric Representations for Generalizable Robot Learning,” in *ICRA*, 2018.

[68] M. Teichmann, M. Weber, M. Zoellner, R. Cipolla, and R. Urtasun, “Multinet: Real-time joint semantic reasoning for autonomous driving,” in *IV*, 2018.

[69] Y. Pan, C.-A. Cheng, K. Saigol, K. Lee, X. Yan, E. Theodorou, and B. Boots, “Agile autonomous driving using end-to-end deep imitation learning,” in *RSS*, 2018.

[70] A. Giusti, J. Guzzi, D. C. Ciresan, F.-L. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, *et al.*, “A machine learning approach to visual perception of forest trails for mobile robots.,” in *RAL*, 2016.

[71] A. Loquercio, A. I. Maqueda, C. R. Del-Blanco, and D. Scaramuzza, “Dronet: Learning to fly by driving,” in *RA-L*, 2018.

[72] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, “Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning,” in *NIPS*, 2014.

[73] C. Richter, W. Vega-Brown, and N. Roy, “Bayesian Learning for Safe High-Speed Navigation in Unknown Environments,” in *ISRR*, 2015.

[74] F. Berkenkamp, A. Krause, and A. Schoellig, “Bayesian Optimization with Safety Constraints: Safe and Automatic Parameter Tuning in Robotics,” in *ArXiv:1602.04450*, 2016.

[75] T. Perkins and A. Barto, “Lyapunov Design for Safe Reinforcement Learning,” in *JMLR*, 2002.

[76] A. Majumdar and R. Tedrake, “Funnel Libraries for Real-Time Robust Feedback Motion Planning,” in *ArXiv:1601.04037*, 2016.

[77] J. Gillula and C. Tomlin, “Reducing Conservativeness in Safety Guarantees by Learning Disturbances Online: Iterated Guaranteed Safe Online Learning,” in *RSS*, 2012.

[78] S. Daftry, S. Zeng, J. A. Bagnell, and M. Hebert, “Introspective Perception: Learning to Predict Failures in Vision Systems,” in *IROS*, 2016.

[79] R. Sutton, “Dyna, an Integrated Architecture for Learning, Planning, and Reacting,” in *AAAI*, 1991.

[80] A. Mujika, “Multi-task learning with deep model based reinforcement learning,” *ArXiv:1611.01457*, 2016.

[81] J. Oh, S. Singh, and H. Lee, “Value Prediction Network,” in *NIPS*, 2017.

[82] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*.

[83] S. Gupta, J. Davidson, S. Levine, R. Sukthankar, and J. Malik, “Cognitive mapping and planning for visual navigation,” 2017.

[84] S. Thrun and A. Schwartz, "Issues in Using Function Approximation for Reinforcement Learning," in *Proceedings of the Fourth Connectionist Models Summer School*, 1993.

[85] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh, "Action-conditional video prediction using deep networks in atari games," in *NIPS*, 2015.

[86] Y. Wu, S. Zhang, Y. Zhang, Y. Bengio, and R. R. Salakhutdinov, "On multiplicative integration with recurrent neural networks," in *NIPS*, 2016.

[87] L.-Y. Deng, *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning*, 2006.

[88] M. Goslin and M. R. Mine, "The Panda3D graphics engine," *Computer*, 2004.

[89] M. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," in *ICML*, 2017.

[90] E. Tzeng, J. Hoffman, T. Darrell, and K. Saenko, "Simultaneous deep transfer across domains and tasks," in *ICCV*, 2015.

[91] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-adversarial training of neural networks," *JMLR*, 2016.

[92] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *ICLR*, 2014.

[93] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," in *IJCV*, 2015.

[94] E. Kaufmann, A. Loquercio, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Deep drone racing: Learning agile flight in dynamic environments," *ArXiv preprint arXiv:1806.08548*, 2018.

[95] C. Watkins and P. Dayan, "Q-Learning," in *Machine Learning*, 1992.

[96] G. Kahn, A. Villaflor, B. Ding, P. Abbeel, and S. Levine, "Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation," in *ICRA*, 2018.

[97] D. Quillen, E. Jang, O. Nachum, C. Finn, J. Ibarz, and S. Levine, "Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods," in *ICRA*, 2018.

[98] A. Bitcraze, *Crazyflie 2.0*, 2016.

[99] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "Ultra low power deep-learning-powered autonomous nano drones," *ArXiv preprint arXiv:1805.01831*, 2018.

[100] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *ArXiv preprint arXiv:1704.04861*, 2017.

[101] N. Savinov, A. Dosovitskiy, and V. Koltun, "Semi-parametric topological memory for navigation," in *ICLR*, 2018.

[102] J. Bruce, N. Sünderhauf, P. Mirowski, R. Hadsell, and M. Milford, “Learning deployable navigation policies at kilometer scale from a single traversal,” in *CoRL*, 2018.

[103] X. Meng, N. Ratliff, Y. Xiang, and D. Fox, “Neural Autonomous Navigation with Riemannian Motion Policy,” in *ICRA*, 2019.

[104] H.-T. L. Chiang, A. Faust, M. Fiser, and A. Francis, “Learning navigation behaviors end-to-end with autorl,” *RA-L*, 2019.

[105] N. Hirose, F. Xia, R. Martín-Martín, A. Sadeghian, and S. Savarese, “Deep Visual MPC-Policy Learning for Navigation,” in *RA-L*, 2019.

[106] M. Riedmiller, M. Montemerlo, and H. Dahlkamp, “Learning to drive a real car in 20 minutes,” in *FBIT*, 2007.

[107] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, “Benchmarking reinforcement learning algorithms on real-world robots,” in *CoRL*, 2018.

[108] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah, “Learning to drive in a day,” in *ICRA*, 2019.

[109] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, 1997.

[110] A. Nagabandi, K. Konoglie, S. Levine, and V. Kumar, “Deep Dynamics Models for Learning Dexterous Manipulation,” in *CoRL*, 2019.

[111] R. Y. Rubinstein and D. P. Kroese, *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

[112] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *CVPR*, 2009.

[113] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” in *AAAI*, 2018.

[114] M. Mathieu, C. Couprie, and Y. LeCun, “Deep multi-scale video prediction beyond mean square error,” 2016.

[115] C. Finn, I. Goodfellow, and S. Levine, “Unsupervised learning for physical interaction through video prediction,” in *NIPS*, 2016.

[116] E. Denton and V. Birodkar, “Unsupervised learning of disentangled representations from video,” in *NIPS*, 2017.

[117] C. Vondrick, H. Pirsiavash, and A. Torralba, “Anticipating the future by watching unlabeled video,” 2016.

[118] R. Mottaghi, M. Rastegari, A. Gupta, and A. Farhadi, ““What happens if...” Learning to Predict the Effect of Forces in Images,” in *ECCV*, 2016.

[119] L. Pinto and A. Gupta, “Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours,” in *ICRA*, 2016.

[120] Y. F. Chen, M. Everett, M. Liu, and J. How, "Socially Aware Motion Planning with Deep Reinforcement Learning," in *IROS*, 2017.

[121] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell, "Learning to navigate in complex environments," in *ICLR*, 2017.

[122] R. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization," *Methodology and computing in applied probability*, 1999.

[123] E. F. Camacho and C. B. Alba, *Model predictive control*. Springer Science & Business Media, 2013.

[124] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *CVPR*, 2015.

[125] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, and D. Precup, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," in *AAMAS*, 2011.

[126] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *NIPS*, 2017.

[127] T. Haarnoja, V. Pong, A. Zhou, M. Dalal, P. Abbeel, and S. Levine, "Composable Deep Reinforcement Learning for Robotic Manipulation," in *ICRA*, 2018.

[128] E. Coumans *et al.*, "Bullet physics library," *Open source: Bulletphysics. org*, 2013.

[129] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *CoRL*, 2017.

[130] T. Verge. (2020). Everyone hates california's self-driving car reports, [Online]. Available: `https://www.theverge.com/2020/2/26/21142685/california-dmv-self-driving-car-disengagement-report-data` (visited on 02/26/2020).

[131] R. S. Sutton, A. G. Barto, *et al.*, *Introduction to reinforcement learning*. 1998.

[132] M. Müller, A. Dosovitskiy, B. Ghanem, and V. Koltun, "Driving policy transfer via modularity and abstraction," in *CoRL*, 2018.

[133] Z. C. Lipton, J. Gao, L. Li, J. Chen, and L. Deng, "Combating reinforcement learning's sisyphean curse with intrinsic fear," *ArXiv:1611.01211*, 2016.

[134] G. Kahn, A. Villaflor, V. Pong, P. Abbeel, and S. Levine, "Uncertainty-aware reinforcement learning for collision avoidance," in *ArXiv:1702.01182*, 2017.

[135] W. Saunders, G. Sastry, A. Stuhlmueller, and O. Evans, "Trial without error: Towards safe reinforcement learning via human intervention," in *AAMAS*, 2018.

[136] G. Kahn, P. Abbeel, and S. Levine, "BADGR: An autonomous self-supervised learning-based navigation system," *ArXiv preprint arXiv:2002.05700*, 2020.

[137] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.

[138] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search," in *ICRA*, 2016.

[139] H. He, J. Eisner, and H. Daume, "Imitation Learning by Coaching," in *NIPS*, 2012.

[140] D. Mayne, M. M. Seron, and S. V. Rakovic, "Robust model predictive control of constrained linear systems with bounded disturbances," in *Automatica*, 2005.

[141] E. Todorov and W. Li, "A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *American Control Conference*, 2005.

[142] S. Levine and V. Koltun, "Guided policy search," in *ICML*, 2013.

[143] H. Kappen, V. Gomez, and M. Opper, "Optimal control as a graphical model inference problem," in *Machine Learning*, 2012.

[144] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in *NIPS*, 2014.

[145] X. Nguyen, M. J. Wainwright, and M. I. Jordan, "Divergences, surrogate loss functions and experimental design," in *NIPS*, 2005.

[146] D. Pollard, "Asymptopia: An exposition of statistical asymptotic theory," 2000. [Online]. Available: `stat.yale.edu/~pollard/Books/Asymptopia/`.

[147] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust Region Policy Optimization," in *ICML*, 2015.

[148] M. Mueller and R. D'Andrea, "A model predictive controller for quadrotor state interception," in *ECC*, 2013.

[149] V. Nair and G. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *ICML*, 2010.

[150] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *ArXiv preprint arXiv:1408.5093*, 2014.

[151] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2015.

[152] P. Martin and E. Salaun, "The true role of accelerometer feedback in quadrotor control," in *ICRA*, 2010.

[153] B. Williams, G. Klein, and I. Reid, "Real-time slam relocalisation," in *ICCV*, 2007.

[154] B. Efron and R. Tibshirani, "The Jackknife, the Bootstrap and Other Resampling Plans," in *SIAM*, 1982.

[155] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutidnov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," in *JMLR*, 2014.

[156] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," in *ICML*, 2016.

[157] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *IJRR*, 2013.

[158] M. P. Deisenroth, G. Neumann, and J. Peters, "A Survey on Policy Search for Robotics," in *Foundations and Trends in Robotics*, 2013.

[159] J. Schneider, "Exploiting Model Uncertainty Estimates for Safe Dynamic Control Learning," in *NIPS*, 1997.

[160] T. Moldovan, S. Levine, M. Jordan, and P. Abbeel, "Optimism-Driven Exploration for Nonlinear Systems," in *ICRA*, 2015.

[161] Y. Gal, R. Mcallister, and C. Rasmussen, "Improving PILCO with Bayesian Neural Network Dynamics Models," in *Data-Efficient Machine Learning workshop, ICML*, 2016.

[162] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, "Deep Exploration via Bootstrapped DQN," in *NIPS*, 2016.

[163] P. Wieber, "Viability and Predictive Control for Safe Locomotion," in *IROS*, 2008.

[164] M. Mueller and R. D'Andrea, "Relaxed hover solutions for multicopters: Application to algorithmic redundancy and novel vehicles," *The International Journal of Robotics Research*, p. 0 278 364 915 596 233, 2015.

[165] M. Watterson and V. Kumar, "Safe receding horizon control for aggressive MAV flight with limited range sensing," in *IROS*, 2015.

[166] J. Gillula and C. Tomlin, "Guaranteed Safe Online Learning via Reachability: Tracking a Ground Target using a Quadrotor," in *ICRA*, 2012.

[167] B. Efron and R. Tibshirani, *An introduction to the bootstrap*. CRC press, 1994.

[168] A. Kleiner, A. Talwalkar, P. Sarkar, and M. I. Jordan, "The Big Data Bootstrap," in *ICML*, 2012.