Reinforcement Learning for Robots through Efficient Simulator Sampling

by

Mark Johnson Cutler

S.M., Massachusetts Institute of Technology (2012)
B.S., Brigham Young University (2010)

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author .....
Department of Aeronautics and Astronautics
August 20, 2015

Certified by .....
Jonathan P. How
R. C. Maclaurin Professor of Aeronautics and Astonautics
Thesis Supervisor

Certified by .....
Leslie P. Kaelbling
Panasonic Professor of Computer Science and Engineering

Certified by .....
Andrea Censi
Research Scientist in LIDS

Accepted by .....
Paulo C. Lozano
Associate Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Reinforcement Learning for Robots through Efficient Simulator Sampling

by

Mark Johnson Cutler

Submitted to the Department of Aeronautics and Astronautics
on August 20, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Reinforcement learning (RL) has great potential in robotic systems as a tool for developing policies and controllers in novel situations. However, the cost of real-world samples remains prohibitive as most RL algorithms require a large number of samples before learning near-optimal or even useful policies. Simulators are one way to decrease the number of required real-world samples, but imperfect models make deciding when and how to trust samples from a simulator difficult. Two frameworks are presented for efficient RL through the use of simulators. The first framework considers scenarios where multiple simulators of a target task are available, each with varying levels of fidelity. It is designed to limit the number of samples used in each successively higher-fidelity/cost simulator by allowing a learning agent to choose to run trajectories at the lowest level simulator that will still provide it with useful information. Theoretical proofs of this framework's sample complexity are given and empirical results are demonstrated on a robotic car with multiple simulators. The second framework focuses on problems represented with continuous states and actions, as are common in many robotics domains. Using probabilistic model-based policy search algorithms and principles of optimal control, this second framework uses data from simulators as prior information for the real-world learning. The framework is tested on a propeller-driven inverted pendulum and on a drifting robotic car. These novel frameworks enable RL algorithms to find near-optimal policies in physical robot domains with fewer expensive real-world samples than previous transfer approaches or learning without simulators.

Thesis Supervisor: Jonathan P. How
Title: R. C. Maclaurin Professor of Aeronautics and Astonautics

Acknowledgments

Despite my best efforts, I've come to the realization over the past five years that PhD theses don't write themselves. Fortunately, though, I have also realized that this thesis is not my work alone, but rather the culmination of efforts, advice, and mentoring from countless people. I'd like to thank those who helped my along this path.

First off, I'm grateful to my advisor, Professor Jonathan How. At several points during my first year in the lab, I was absolutely convinced that I wouldn't pass the qualifying exams and that I was unfit for PhD research. While I'm still not certain about the later, I am grateful to Jon for helping me through the process, for keeping me in the lab, for sharing a passion for cool hardware videos, and for overall being an excellent advisor.

I would also like to thank my thesis committee, Professor Leslie Kaelbling and Dr. Andrea Censi, for their invaluable input, feedback, and support. I would also like to thank my thesis readers Dr. Shih-Yuan Liu and Dr. Miao Liu for taking time from their busy schedules to give me feedback on this document and the work.

I am incredibly grateful to the amazing group of graduate students both past and present in the ACL. From helping me with problem sets to running endless experiments to sharing a laugh about the PhD experience, they have been there every step of the way. In particular, I want to thank Thomas Walsh for his invaluable RL expertise and help with this thesis, Buddy Michini and Josh Redding for teaching me what a microcontroller is, Tuna Toksoz for teaching me to program, Kemal Ure for answering all of my math and machine learning questions, Steven Chen for patiently helping me through many, many problem sets, Luke Johnson for undoubtedly being the smartest student in the lab, Matt Graham for helping me study for and pass quals, Shayegan Omidshafiei and Justin Miller for their video editing skills, Mike Klinker for building my hardware, and Brett Lopez for (somewhat unknowingly) volunteering to take over the hardware duties in the lab. I couldn't have asked for a better group of lab mates, mentors, and friends.

I would also like to thank my funding sources, the National Science Foundation Graduate Research Fellowship (grant No. 0645960) and Boeing Research & Technology for their support of the facility in which the experiments were conducted.

Finally, none of this would have happened without the incredible support I received these many years from my loving wife, Kristen. She has been with me through thick and thin, encouraged and motivated me, and tirelessly worked to maintain order amid the chaos of kids, school, and work. I love her dearly. I'm also grateful to my three children, Hailey, Rachel, and Aaron. They are a sure source of joy when my experiments refuse to cooperate or the math problems are difficult.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real robots are often physically more capable than the software that controls them. For instance, expert human pilots are able to control remote-controlled (RC) airplanes and helicopters through very aggressive aerobatics and flight maneuvers, often pushing the aircraft to the limits of its physical structure. However, with few exceptions (e.g., using expert demonstrations to facilitate learning [1, 2]), computer algorithms have been unable to exploit the full capabilities of RC airplanes and helicopters to create flight performances that rival that of expert pilots. While significant limitations in achieving this goal can be attributed to limitations in sensors and state estimation capabilities, developing robust, aggressive flight controllers remains difficult, particularly because high-fidelity models of dynamic robots are both difficult to develop and difficult to use, resulting in using simplified models for control design and the real robots for experimentation and verification.

Reinforcement Learning (RL) [97] provides a framework that can help robots autonomously develop control laws and policies that enable them to achieve complicated motion in difficult-to-model environments. By allowing the robot to explore the environment and perceived rewards, learning agents discover policies and controllers, often through a trial-and-error methodology. In addition to freeing the design engineer from having to hand-code controllers, RL can discover new policies and controllers and adapt to novel situations without requiring reprogramming.

Over the past 20 years RL has demonstrated impressive results in many com-

plicated domains such as robot locomotion [55], helicopter flight [9, 76], backgammon [107], and elevator control [13]. However, many limitations exist when implementing RL on real robots [52]. In particular, real robots operate in a continuous world and often have relatively high-dimensional state and action spaces, thus suffering from the "curse of dimensionality" [14]. Also, real robots have real costs associate with running them, thus also suffering from the "curse of real-world samples" [52]. These costs can be financial (robots are often expensive), temporal (robots run in the real world and therefore are limited to running in real-time), and occasionally emotional (watching your robot crash for the n'th time is demoralizing). Because classical RL typically requires a large number of samples before converging to optimal or near-optimal policies [49], using RL to learn optimal behavior from scratch on a real robot is often ineffective and impractical. This thesis attempts to improve the efficiency and practicality of RL algorithms for robots through the use of simulations of the robot.

1.1 Motivation: Using Simulators Efficiently to Increase Learning Speed

Simulators often accompany any work with real robots. While it is difficult, if not impossible, to perfectly model something as complicated as a robot interacting with the world, simplified simulations are typically either already available or easy to construct for most robots. These simulations provide valuable insight for design engineers as they are used to effectively "weed-out" algorithms, code bugs, and ideas that would certainly be not useful and possibly even detrimental to implement on the real robot. In many cases, the simulator is a sanity check to ensure that, at least in the simplified setting of the simulation, the robot is operating correctly.

As Figure 1-1 indicates, simulations of robots often vary in quality. For example, a robot helicopter can be simulated by a simple point-mass model simulation of the helicopter that captures the general effects of different actions, but ignores much

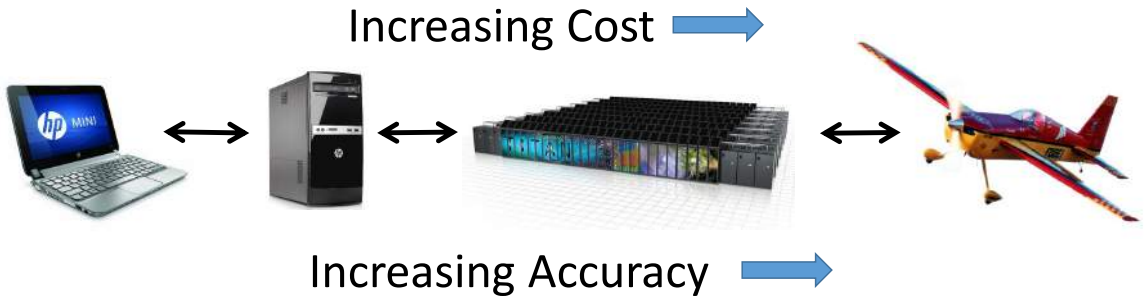


Figure 1-1: Many robotic domains (like the airplane on the right) use simulators for initial code development, hardware design, etc. Often, a range of simulators are used, varying from simple, low-cost (time and/or computing power) simulators on the left to complex, high-cost simulators on the right. The main goal of this thesis is to develop algorithms that use a series of simulators to efficiently learn useful behavior on real robots.

of the dynamics. Alternatively, a complicated physics simulation that accounts for aerodynamic forces such as blade flapping, motor dynamics, battery wear, etc can be used. This more complicated simulation may be more accurate in predicting how the actual robot will behave, but also may take significantly more time and computation power to run, although still being less accurate and less costly than flying the actual helicopter.

The key idea of this thesis is to efficiently utilize the simulators available for a robotic task to both decrease the time and real-world samples required for learning and increase the performance of a target task. In particular, I develop frameworks that use simulations when possible and only learn on the real robot that which cannot be learned in the simulator. The key challenge is choosing both *when* and *how* to use simulated data versus real-world data when learning a novel task.

1.2 Problem Statement

The primary goal of this work is to develop algorithms that increase the efficiency of real-world RL by transferring sampling complexity from the robot to simulations of the robot. Loosely speaking, the simulations should be used for what they are good at—generating lots and lots of samples, while the robot should be used to generate a few very good samples. Together, these ideas lead to algorithms that only learn in

the real world the things that the simulators do not accurately predict.

There are several challenges associated with this problem, primarily dealing with the type of information that is passed between simulators during a learning process.

- *Passing Information Between Simulators:*

  For any robotic task, the real robot performance is ultimately the most important metric. Thus, by definition, data generated by the physical robot are more accurate than data generated by the simulators. Similarly, data observed in a higher-fidelity simulator are assumed to be more accurate than data collected in the same state-action space in lower-fidelity simulators. Thus, the accuracy of lower-fidelity simulators (with respect to the real world) can be improved by appropriately using this observed information. This can decrease the samples required in the real world by allowing learning agents to rely more on lower-cost simulators; however, the benefits will depend on the type of information passed (e.g. observed reward and transition values or learned policy parameters) and on how that information is used by the lower-fidelity simulators. The primary challenge of this work is effectively combining information from inexpensive, potentially inaccurate simulators with expensive real-world samples to generate optimal behavior in the real world.

- *Generalization Between Simulators:* Function approximation [87] has been critical for scaling RL problems to real-world domains, as tabular representations of transition dynamics and value functions are impractical for large, high-dimensional problems. Function approximation has also allowed data collected in a specific state-action pair to be generalized to neighboring states and actions. A similar challenge exists when using multi-fidelity simulators to learn in a real-world target task. Specifically, the challenge is to generalize data across simulators in order to maximize the utility of observing samples in high-fidelity simulators and the real world.

- *Avoiding Negative Transfer:* A major concern in Transfer Learning (TL) [104] is the notion of *negative transfer*, or transferring information between different

domains that causes worse behavior in the target domain than had no information been passed at all. When passing information between simulators, both up and down the multi-fidelity chain, ensuring that passed information will not negatively affect the learning is a major challenge.

1.3 Literature Review

There is a vast existing literature regarding RL and robotics. This section attempts to highlight the major contributions relevant to using simulations to solve RL, control, and optimization problems.

1.3.1 Simulators in Robotics and Reinforcement Learning

Many researchers have used simulators to speed learning in RL. For instance, prioritized sweeping [72] and the Dyna architecture [98] perform computation between real-world samples to direct learning. In adaptive control [85], forward models of dynamic systems are learned from data [71] and are used to compute controllers. However, these learned forward models will never perfectly model the outcomes of a complicated real-world system, often resulting in policies learned in the model that do not perform well in the real world [7].

While a few researchers have shown that policy transfer from a simulation to a real robot is possible [10, 31, 32, 45, 100], as noted in [52], policy transfer from simulation to the real world will probably only work if the system is inherently stable. It almost certainly won't work if the transferred policy is needed for high-rate stabilization because of discrepancies between the real robot and the simulation. Also, policy transfer is limited in performance by the accuracy of the simulator. Unless the real world is perfectly simulated, it is very unlikely that a policy which is optimal in the simulator will be optimal in the real world. Therefore, some real-world experience is needed to find optimal policies on real robots.

To account for model uncertainty, some researchers have used methods that explicitly account for variance in the learned forward models [9, 91], yielding methods

that are cautious on the real robot by noting where the forward model is uncertain of outcomes. These methods, though, do not incorporate specific simulators of the robot, but instead just apply the model uncertainty across learned models.

As an alternative to directly transferring learned optimal policies from a simulator to the real robot, simulators are often used to train learning agents, with real-world experience used later to update the simulator or the agent's policy (e.g. [3]). However, such systems do not guarantee an efficient collection of samples from different models, require practitioners to decide when to run policies in the simulator/real world, and do not guarantee efficient exploration. Another approach is to always execute actions in the real world but use a low-fidelity simulator to help compute policy search gradients [3, 26, 29, 53, 56, 81]. However, these approaches are specific to policy search algorithms and, again, do not provide exploration guarantees. A similar idea in the controls community is know as Iterative Learning Control (ILC) [5, 6, 18], where simple models of the system are used to improve closed-loop control. ILC has shown impressive results on dynamic robots when the task to be performed is repetitive [36, 73, 92, 109]. However, these methods work only for explicitly repetitive tasks.

Simulators and demonstrations have also been used in model-based RL as a way to gain additional samples for the learning agents without running the real hardware [11, 75]. These approaches, though, do not allow for multiple levels of simulation nor do they guarantee efficient exploration. Recently (and after this initial work was already published), an algorithm for decreasing samples needed from hardware experiments was published [40]. The authors use an iterative, model-free RL algorithm with neither formal guarantees nor hardware results.

Significant recent advances in RL for robotics have also come from model-based policy search methods [28]. Real-world data are used to construct a model of the world, from which trajectory roll-outs can be used to update policy parameters [25]. These methods search locally for improvements to the current policy and so scale well as the problem dimensionality increases. However, due to the local nature of the search, they typically lack any formal guarantees as to global optimality and

are subject to local solutions. As shown in Chapter 5, problems exist for which these methods get stuck in very poor local solutions, particularly when multiple local solutions exist.

Methods such as trajectory optimization for exploration [62] and skill generalization [58] have shown impressive results in simulated and real robot domains, albeit often without convergence or sample complexity guarantees. By contrast, the frameworks developed in this thesis use data not just from a target domain but also from available simulators to build *multiple* models. Furthermore, to make provably efficient use of data, the framework developed in Chapter 3 uses KWIK-Rmax-style learning and planning [65] rather than policy search techniques.

Multi-fidelity simulator models have been considered for a single agent that learns different policies for varying observability conditions [115], but these models are not actively selected as they depend strictly on the observed environment. Multi-fidelity models have been used in the multi-agent context to combine data from people performing tasks in different simulators [90], but these policies were learned from traces through supervised learning, not RL.

1.3.2 Simulators and Models in Control

Many systems utilize models of varying fidelity for design, validation, and control [22, 23, 35, 108]. Simple models, such as linearized equations of motion, are typically used for control design, while more complex, data based or finite element based simulations are used for control and design verification. Finally, by assumption, the real world is the highest fidelity simulator and is used to verify the controllers and designs from the simulators, ultimately determining value.

Simple models are needed in the control design process as many traditional methods such as classical, adaptive, and optimal control rely on models with explicit equations of motion to develop the required controllers. These simple models are often deterministic and use closed-form, expressible equations of motion. Particularly in the case of optimal control, resulting policies are sometimes open-loop and depend entirely on the model used to compute them. These restrictions on simple

models used for control design lead to models that sometimes neglect parts of the true system, either because they are non-linear or because they are just not well-enough understood to be expressed in equations.

On the other hand, simple models for control are advantageous in that the policies designed for them consider the entire state-action space during the design process. The controllers are designed considering the equations of motion as a whole, rather than by sampling a complex simulator or the physical environment, leading to "global" policies, or policies that have considered the state-action space globally.

More complex (and hopefully more accurate) simulations are then used to verify and validate the controllers developed using the simple models [37, 46]. These complex simulations are typically stochastic, either through an attempt to model the stochasticity inherent in the physical system, or as a way to compensate for and capture unmodeled dynamics [3]. Complex simulators can also be "black-box" in the sense that the equations of motion can not be easily written down (e.g. when using physics simulators such as Gazebo [54]) or that the simulation engine is proprietary or otherwise unavailable (e.g. a commercial video game).

1.3.3 Transfer Learning

Similar to approaches in RL that attempt to generalize observed transition and reward information to nearby states and actions, transfer learning (TL) tries to generalize learned behavior across different tasks and domains. In TL, an RL algorithm is applied in a source task. Pertinent information is then transferred to a target task of interest, with the hope that the transferred information will increase the performance and/or decrease the learning time in the target task when compared to not using the transferred information. Significant results have been obtained through TL, such as learning a cart-pole task with changing parameters [93], reusing a set of learned policies [33], and transferring data between tasks with different state and action spaces [106].

In the TL framework, algorithms are designed to maximize utility when the domains being transferred to and the time at which transfer occurs are not selected

by the algorithm. Also, information transfer in TL is typically unidirectional, from a source task to a target task, but not the other way. Recent work in TL has considered sample complexity bounds in a series of environments under both bandit [8] and full MDP conditions [20]. However, both of these works assume environments are chosen i.i.d., not by the agent itself.

Representation transfer [105] applies the ideas of TL to transferring information between different representations of the same task. By changing the representation the RL algorithm is using (e.g. the current function approximation form), representation transfer may be able to decrease learning time and increase performance.

Significant information can be transferred from a source task to a target task through the use of *region transfer* [60,61]. State transition and reward values observed in a source task are transferred directly to the target task, but only from regions that are known to be similar to the target task. Thus, if the source task is similar to the target task in one half of the state-action space, observed transition values from that half would be used in the transferred domain, but not from the other half. This idea has interesting potential if applied to robotic domains where information is transferred from a simulator, but only in those parts of the state-action space that accurately model the real-world robot.

Transferred Delayed Q-Learning (TDQL) [67] transfers the value function *unidirectionally* from a source learning task as a heuristic initialization for the target learning task. Because TDQL uses Delayed Q-learning [95] as its base learner, it can only be used with tabular representations, not more general structured representations such as those covered by the "knows what it knows" (KWIK) framework [65]. Also, the TDQL algorithm does not support transferring learned model parameters to lower-fidelity simulators from higher-fidelity simulators, returning to lower-fidelity simulators when useful information might be gained.

Finally, TL with *partially constrained models* [34] learns in a computationally complex environment by using an approximation to the true simulation. The approximate model is created by removing some of the constraints that exist in the original problem. As with TDQL, information is passed only from the approximate model to

the true model, and not the other direction.

1.3.4 Multi-Fidelity Optimization

While not directly related to RL, the field of multi-fidelity optimization (MFO) [83, 88, 89] uses the idea of multiple levels of simulation to solve optimization problems. Sometimes, the objective function that needs to be optimized requires significant computation and time to return each function evaluation. In these cases, MFO uses approximations to the true objective function to perform a global search, finding a region that contains the optimal solution. Then, a final local search is performed using the original objective function to reach the true optimum.

Approximations to the true objective function, called surrogate or meta-models, fit into three broad categories [88]: data-fit models, reduced-order models, and hierarchical models. Data-fit surrogate models [48, 83, 112] are statistical representations of the true objective function created using sampled points of the true function. The statistical models used range from parametric methods such as polynomial regression to non-parametric methods such as radial basis functions [83]. Reduced-order models [86] approximate the objective function by constructing a simpler function using a set of basis functions and data from the target function, often using proper orthogonal decomposition [44]. Finally, hierarchical models use simpler, physics-based models as an approximation to the true objective function. These simpler functions could be simpler engineering models [4] or the original objective function solved with a coarser grid [63]. The simple models can be used as valid approximation models through the use of additive corrective terms, where the output of the simple model is corrected to predict observed outcomes from the actual objective function. These additive corrections can be described in the original objective function's state space which is not necessarily the same as the approximate model's state space [88].

Techniques in MFO include learning model disparities [113] and constraining search based on results from lower-fidelity models [70]. However, MFO does not consider sequential decision making tasks. One of the key contributions of this thesis is borrowing lessons from MFO and applying them to RL problems in order to

minimize the number of needed real-world samples to learn a given task.

1.3.5 Evolutionary Robotics

The field of *evolutionary robotics* commonly uses a combination of simulated and real data to learn new controllers [47]. Genetic algorithms are used to evaluate candidate controllers, with most of the evaluations occurring in simulation. Several papers have investigated the "reality gap" that occurs when controllers that work well in simulated environments perform poorly on the physical hardware, and develop robust ways of making simulators that can be used in conjunction with the physical hardware [57, 69, 116]. While the ideas are similar to the current work, the first half of this thesis focuses on the RL problem where efficient exploration can be guaranteed.

Very recent work shows an evolutionary robotics algorithm that combines a heavy preprocessing phase in simulation that allows robots to intelligently update predicted real-world performance based on sparse samples [21]. The researchers use Bayesian optimization and Gaussian processes to select new test parameters in the presence of hardware failures. In principle, these ideas mirror closely those presented in this thesis: use simulated data where possible. However, the algorithms presented later are specifically designed to avoid exhaustively sampling the simulated state-space and learn closed-loop feedback controllers with continuous states and actions, rather than open-loop gait patterns.

1.4 Summary of Contributions

This thesis focuses on the development of algorithms and methods that reduce the sample complexity of learning in the real world, as an attempt to make RL practical for physical robots. As such, the goal is *not* to make novel, sample efficient RL algorithms, but rather to develop methods that use existing algorithms in novel ways to transfer sampling burden from the real world to the simulator(s). In this respect, a number of contributions to the state of the art have been made.

First, a new framework, *Multi-Fidelity Reinforcement Learning* (MFRL), for per-

forming RL with a heterogeneous set of simulators (including the real world) is introduced, analyzed, and empirically demonstrated. MFRL not only chooses actions for an agent to execute, but also chooses in which simulator to perform them. The result is an agent that both

- uses information from lower-fidelity simulators to perform limited exploration in its current simulator, and
- updates the learned models of lower-fidelity agents with higher-fidelity data.

The MFRL framework specifies rules for when the agent should move to a higher-fidelity simulator, as well as moving to a lower-fidelity level before over-exploring in a more expensive simulation. These rules and the transfer of values and data provide theoretical guarantees on convergence and sample efficiency. Specifically, given the assumptions in Section 3.1.2, the framework

- does not run actions at high-fidelity levels that have been proven to be suboptimal at lower-fidelity levels,
- minimizes (under certain conditions) the number of samples used in the real world, and
- polynomially limits the total number of samples used in all simulators.

In addition, in the worst case, MFRL provably uses no more real-world samples than unidirectional transfer approaches.

The main contributions relating to MFRL are

- introducing the MFRL framework for learning with multiple simulators, which is the first framework that dynamically chooses which simulator to perform actions in,
- a theoretical analysis of the framework's sample complexity, and
- several demonstrations of efficient learning on a robotic car with fewer real-world data points than unidirectional transfer or learning without simulators.

These results demonstrate MFRL is a provably and practically efficient manager of the low and high quality simulators often available for robotics tasks.

The second major set of contributions revolves around a framework tailored for domains with continuous representations of the states and actions. Policy search methods have gained a lot of traction in recent years in high-dimensional, continuous robotic domains [28]. By searching directly in the space of policies many of the computational complexities of the "curse of dimensionality" can be avoided, at the expense of theoretical guarantees (and sometimes practical performance).

In the second half of this thesis a framework that is similar in spirit to MFRL, but is centered around a model-based policy search algorithm called Probabilistic Inference for Learning Control (PILCO) [25], is developed. PILCO is applied to a simulator to learn a model of the simulated dynamics and a good policy for the simulated domain. The learned transition dynamics and policy are then used as a prior for a real-world learning agent. The simulated prior is used in a Gaussian process (GP) model of the transition dynamics in PILCO to infer about states that the real-world system has not yet sampled. Experiments demonstrate that, even when the simulator is inaccurate, using an informative simulated prior decreases the learning samples needed in the real world and increases the average performance of the achieved solution. This approach differs from previous work using priors in PILCO [15] since it is not limited to linear priors. Using a learned, nonlinear prior from a simulator allows for incorporating prior information from arbitrarily complex simulations without needing to make assumptions about the underlying dynamics of the system.

Finally, a method for transferring information gained from the physical system *back* to the simulator (referred to as *reverse transfer*) is developed. Combined with intelligent policy initializations based on optimal control, this reverse transfer helps avoid local minima and helps PILCO converge to good policies that a generic policy search algorithm would be very unlikely to find. Experiments demonstrate that using data observed in the real world, learning agents can revisit the simulator and sometimes find different policies that have lower cost in the real system.

The main contributions of this framework are

- a way to incorporate principles of optimal control for initializing policy parameters and avoiding local minima,
- a principled approach to incorporating data from any simulator into the PILCO learning algorithm,
- a derivation of propagating uncertain inputs through a Gaussian process with a nonparametric mean function,
- a derivation of mixing real and simulated data together in order to re-plan in a simulated environment, and
- simulated and hardware results empirically showing the benefits of using prior information in the learning process.

Using prior data from a simple simulator, convergence to a good policy on a physical inverted pendulum is shown with at least three times less data than is needed when a zero-mean prior is used. The framework is also applied to a robotic car that learns to drift by using optimal control to find an initial policy.

A summary of and links to the videos of the hardware experiments relating to this thesis can be found in Appendix A.

Chapter 2

Background

This section provides background on some of tools used in this thesis.

2.1 Reinforcement Learning

A simulator, Σ, is represented by a Markov Decision Process (MDP) [82], $M = \langle S, A, R, T, \gamma \rangle$ with states S and actions A. The bounded reward function is defined as $R(s,a) \mapsto [R_{\min}, R_{\max}]$ where $R_{\min}$ and $R_{\max}$ are real numbers. The transition function T encodes the probability of reaching some next state given the current state and action. Specifically, $T(s,a,s') = Pr(s'|s,a)$.

The optimal *state-action value function* is specified as

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')V^*(s'),$$

where $V^*(s) = \max_a Q^*(s,a)$. Intuitively, $Q^*(s,a)$ is the expected sum of discounted rewards when taking action a in s and then acting optimally thereafter. A deterministic policy $\pi : S \mapsto A$ is said to be optimal when $\pi^*(s) = \text{argmax}_a Q(s,a)$. Given an MDP (including T and R), planning methods such as value iteration [82] can be used to calculate the optimal value function and optimal policy.

In Chapters 4 and 5 both the states and the actions will be real-valued vectors. The reward function is also be defined as the negative of a user-defined cost function,

such that $R(s) = -c(s)$. The cost function is chosen to depend only on the current state and not the chosen action. The goal of the RL algorithm will be to find a policy $\pi^* : s \to a$ that minimizes the expected long-term cost

$$J = \sum_{t=0}^{H} \mathrm{E}_{s_t}[c(s_t)], \tag{2.1}$$

where the expected cost is a function of the state s_t, over some fixed horizon H.

In RL, an agent knows S, A, and γ but not T and R, which it learns from interaction with the environment. Many classes of algorithms exist for learning in this setting including classical algorithms such as Q-learning [114], which do not build explicit representations of T and R. Model-based reinforcement learning solutions, like classical R-max [17], explicitly build estimates $\hat{T}$ and $\hat{R}$ from data and then use a planner, such as value iteration, to determine the optimal policy. These model-based learners are generally more sample efficient than model-free approaches [102].

RL algorithms also differ as to how the optimal policy is found. In *policy search* methods, $\pi^*(\boldsymbol{\theta})$ is parameterized by a vector $\boldsymbol{\theta}$ and the RL algorithm searches for the optimal parameter values. In *value-function* methods, π^* is instead found by keeping track of the estimated long-term cost of each state. Policy search methods are often advantageous as expert knowledge can easily be incorporated by specifying the form of the policy. Also, the number of parameters needed to optimize are usually fewer in policy search methods than in corresponding value-function approaches [52].

2.2 Sample Complexity and the KWIK Framework

An RL algorithm searches for an optimal policy but does not know T and R a priori. This leads to an inherent tension between *exploration*, where an agent seeks out new experiences to update its model, and *exploitation* of its current knowledge where the agent attempts to complete the task. The exploration/exploitation trade-off has been studied extensively in RL [17, 65, 95, 101].

To judge the exploration efficiency of the algorithm, previous definitions [95] of

sample complexity for an RL agent are followed. Sample complexity analysis is used to bound the worst-case number of experiences needed by an agent to reach near-optimal behavior with high probability. Specifically, the following definition is used.

Definition 1. *The* sample complexity *of a reinforcement learning algorithm $\mathcal{A}$ that selects action a_t at state s_t on each timestep t is, with probability $1-\delta$, the maximum number of timesteps where $V^{\mathcal{A}_t}(s_t) < V^*(s_t) - \epsilon$.*

The KWIK framework [65] standardizes sample complexity analysis for model-based RL agents by measuring the number of times the learners of T and R are uncertain in making a prediction. Because samples for T and R have labels (the noisy sampled transition and the reward signal itself), these learners can be analyzed in the supervised learning setting. The KWIK framework defines sample complexity for supervised learners that initially only know the intended hypothesis class $H : X \mapsto Y$ and accuracy/confidence parameters ϵ and δ. Learning then follows the following protocol:

1. At each timestep t, the learner is given an input $x_t \in X$, potentially chosen adversarially. No distributional assumptions are made on the choice or order of inputs.

2. If the agent is certain of its prediction ($||\hat{y}_t - y_t|| < \epsilon$ with high probability), it predicts $\hat{y}_t$.

3. Otherwise, it must state "I don't know" (denoted $\perp$) and will view a noisy state measurement z_t, where $E[z_t] = y_t$.

The framework forbids (with high probability) the agent from making an inaccurate prediction of y_t. It must explicitly admit areas of the input space where it does not yet have enough data to make accurate predictions of T and/or R. A state becomes known once it has been observed sufficiently many times for the learner to be ϵ-confident of the outcomes. The *KWIK sample complexity* for such a supervised learner is just the number of times, with probability $1-\delta$, where it predicts $\perp$. A

Algorithm 1 KWIK-Rmax [65]

1: **Input:** Planner P, S (in some compact form), A, γ, ϵ, δ
2: **Initialize:** KWIK learners $L_T(\epsilon, \delta)$ and $L_R(\epsilon, \delta)$
3: **for** each timestep and state s_t **do**
4: **if** $t = 0 \vee \langle s_{t-1}, a_{t-1} \rangle$ produced $\perp$ but are now known **then**
5: Build $\hat{T}$ and $\hat{R}$ by filling in $\perp$ predictions from L_T and L_R with $R_{\max}$ heuristic
6: $\pi = P(\langle S, A, L_T.\hat{T}, L_R.\hat{R}, \gamma \rangle)$
7: $a_t = \pi(s_t)$
8: Execute a_t, Observe s_{t+1}, r_t.
9: **if** $L_T(s_t, a_t) = \perp$ **then**
10: $L_T.update(s_t, a_t, s_{t+1})$
11: **if** $L_R(s_t, a_t) = \perp$ **then**
12: $L_R.update(s_t, a_t, r_t)$

hypothesis class H is said to be *KWIK learnable* if an agent can guarantee, with probability $1 - \delta$, it will only predict $\perp$ a polynomial (in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, $|H|$) number of times.

The KWIK learnability of T and R can be used to induce a polynomially sample efficient RL algorithm, through the KWIK-Rmax RL framework [65], as illustrated in Algorithm 1. KWIK-Rmax maintains KWIK learners L_T and L_R with appropriate accuracy parameters (line 2) and queries these learners to create an approximate MDP with transition and reward functions $\hat{T}$ and $\hat{R}$. The KWIK learners are fed samples from the agent's experience. Whenever a state/action pair goes from "unknown" to "known," the algorithm replans with the learned MDP constructed from $\hat{T}$ and $\hat{R}$.

Because of its reliance on KWIK learners, the algorithm is explicitly aware of which areas of the state space are "known" (where L_R and L_T can make accurate predictions) or "unknown" (L_T or L_R predict $\perp$). In the unknown areas, $\perp$ predictions from the learners are replaced using the "optimism in the face of uncertainty" heuristic when constructing $\hat{T}$ and $\hat{R}$. Specifically, when L_R predicts $\perp$ for $\langle s, a \rangle$, $\hat{R}(s, a) = R_{\max}$. When L_T predicts $\perp$, a special transition to a state with a value of $V_{max} = \frac{R_{\max}}{1-\gamma}$ is inserted (line 5). This interpretation encourages exploration of unknown areas but not at the expense of already uncovered dominant policies. It has been shown that proving T and R are KWIK learnable and using them in the KWIK-Rmax framework guarantees polynomial sample complexity of the resulting RL agent [65]. The crucial

tenets of this relationship are the following properties of the KWIK-Rmax framework:

- **Optimism:** $V_t(s) \geq V^*(s) - \epsilon$ for all timesteps t
- **Accuracy:** $V_t(s) - V^{\pi_t}_{M_{k_t}}(s) \leq \epsilon$ for all timesteps t. That is, the values at known states must be accurate with respect to all other known states and the current policy on all timesteps.
- **Bounded number of changes:** The number of changes to the value function and visits to unknown states should be polynomial with respect to the accuracy and domain parameters.

The first property stems from the optimistic construction of $\hat{T}$ and $\hat{R}$ in areas where the learners predict $\perp$ and helps drive exploration to under-sampled areas. The second property follows from the KWIK accuracy requirement that predictions are made only when significant evidence is available. The final property is a result of the KWIK polynomial sample complexity of L_T and L_R.

In MFRL, the KWIK-Rmax framework is used at each level of simulation. Unknown and known areas at each level are cataloged by KWIK learners as specified above. However, the uninformed $R_{\max}$ heuristic is instead replaced by value functions from lower level simulators, filling in areas where learners at the current level predict $\perp$. Learned parameters of $\hat{T}$ and $\hat{R}$ from higher-fidelity levels are also shared to increase the accuracy of lower-fidelity simulations.

The same base KWIK learners are used for the parameters of the MDP in MFRL. This will allow the algorithm to determine which simulators have accurate models, and where the learned parameters of those models differ. Also, unlike single-environment KWIK-Rmax, the $\frac{R_{\max}}{1-\gamma}$ heuristic is replaced with a more informed heuristic based on interactions with other simulators.

Use of the KWIK framework also encourages the use of MFRL with compact (non-tabular) representations of T and R. That is, because the KWIK framework faciliates general learning algorithms for T and R, representations with far fewer parameters than the number of states can be used by an RL agent. For instance,

Algorithm 2 Probabilistic Inference for Learning COntrol (PILCO) [25]

1: **input:** Random controller parameters ($\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$). Apply random control signals and record data.
2: **while** task not learned **do**
3: Learn probabilistic (GP) dynamics model using observed data
4: **while** not converged **do**
5: Approximate inference for policy evaluation
6: Gradient-based policy improvement
7: Update parameters $\boldsymbol{\theta}$ (e.g., CG or L-BFGS)
8: **return** $\boldsymbol{\theta}^*$
9: Set $\pi^* \leftarrow \pi(\boldsymbol{\theta})^*$
10: Apply π^* to system and record data

despite a potentially infinite number of reachable states in a continuous environment, linear dynamics are KWIK learnable with only an $\tilde{O}(n^3)$ dependency on the number of dimensions n [65]. Thus, the sample complexity of KWIK-Rmax may be smaller than $|S|$ if T and R can be represented compactly. By building the MFRL framework over KWIK learners, the ability to efficiently explore in large domains is gained, a fact used in Section 3.4 and later in the robotic car experiments.

2.3 PILCO

Probabilistic Inference for Learning COntrol (PILCO) is a recently developed model-based policy search RL algorithm [25–27]. One of the key advantages of PILCO is a careful handling of uncertainty in the learned model dynamics that helps prevent negative effects of model bias. By explicitly accounting for uncertainty, the algorithm is able to determine where in the state space it can accurately predict policy performance and where more data are needed to be certain of future outcomes.

The PILCO algorithm is shown in Algorithm 2. Learning begins by randomly applying control signals and then using the observed data to build a probabilistic model of the transition dynamics. This model is then used to update the policy parameters by optimizing over long-term roll-outs of the learned model. Closed-form gradients of the optimization problem are available and so any gradient-based optimization algorithm can be applied. Once converged, the new policy parameters

are executed on the system and the process repeats until satisfactory performance is obtained.

In this thesis, as for the original PILCO algorithm, empirical simulations and hardware results are used to verify the utility of the proposed algorithm.

2.4 Gaussian Processes

Gaussian processes (GPs) [84] are a popular regression tool for modeling observed data while accounting for uncertainty in the predictions, and are used to model the transition dynamics in PILCO. Formally, a GP is a collection of random variables, of which any finite subset are Gaussian distributed. A GP can be thought of as a distribution over possible functions $f(\boldsymbol{x})$, $\boldsymbol{x} \in \mathcal{X}$ such that

$$f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}')), \tag{2.2}$$

where $m(\boldsymbol{x})$ is the mean function and $k(\boldsymbol{x}, \boldsymbol{x}')$ is the covariance function.

With a fixed mean function and data $\{X, \boldsymbol{y}\}$, where X and $\boldsymbol{y}$ are the input and output data, respectively, the predictive distribution for a deterministic input $\boldsymbol{x}_*$ is

$$\begin{aligned} f_* &\sim \mathcal{N}(\mu_*, \Sigma_*) \\ \mu_* &= m(\boldsymbol{x}_*) + k(\boldsymbol{x}_*, X)(K + \sigma_n^2 I)^{-1}(\boldsymbol{y} - m(X)) \\ &= m(\boldsymbol{x}_*) + k(\boldsymbol{x}_*, X)\boldsymbol{\beta} \\ \Sigma_* &= k(\boldsymbol{x}_*, \boldsymbol{x}_*) - k(\boldsymbol{x}_*, X)(K + \sigma_n^2 I)^{-1} k(X, \boldsymbol{x}_*) \end{aligned}$$

where $\boldsymbol{\beta} = (K + \sigma_n^2 I)^{-1}(\boldsymbol{y} - m(X))$, $K = k(X, X)$, and σ_n^2 is the noise variance parameter.

As in the PILCO algorithm [25], in this thesis the squared error kernel is used for its computational advantages. Thus, the kernel is

$$k(\boldsymbol{x}, \boldsymbol{x}') = \alpha^2 \exp(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}')^T \Lambda^{-1}(\boldsymbol{x} - \boldsymbol{x}')),$$

where α^2 is the signal variance and Λ is a diagonal matrix containing the square of the length scales for each input dimension. The hyperparmeters (σ_n^2, α^2, and Λ) are learned via evidence maximization [84].

2.5 Summary

This section provided some brief background material on the tools used later in the thesis. In particular, details regarding reinforcement learning, sample complexity, the "Knows What It Knows" RL framework, the Probabilistic Inference for Learning Control algorithm, and Gaussian processes were covered. These tools will be used in the following chapters to develop two frameworks to decrease the hardware sample complexity of RL problems with real robots.

Chapter 3

Multi-Fidelity Simulations

This chapter considers the problem of efficient real-world RL when multiple simulations are available, each with varying degrees of fidelity to the real world. Lower-fidelity simulators are assumed to be less expensive (typically less time to evaluate), but also less accurate with respect to the actual robot. Lower-fidelity simulators are also assumed to be optimistic with respect to the real world. While this assumption may not be valid in all domains, it is consistent with my observations of many simulators; policies that work in simulation *might* work on the real robot, but policies that perform poorly in simulation are unlikely to succeed in hardware. Machinery is also included to relax this optimistic assumption.

A new framework, *Multi-Fidelity Reinforcement Learning* (MFRL), for performing RL with a heterogeneous set of simulators (including the real world) is introduced, analyzed, and empirically demonstrated. MFRL, depicted in Fig. 4-1, not only chooses actions for an agent to execute, but also chooses which simulator to perform them in. The framework combines ideas from multi-fidelity optimization [88] and advances in model-based RL that have yielded efficient solutions to the exploration/exploitation dilemma. More specifically, heuristics from lower-fidelity simulators and adjustments from high-fidelity data (common techniques in multi-fidelity optimization) are instantiated in MFRL using the successful "optimism in the face of uncertainty" heuristic and the "Knows What It Knows" (KWIK) model-learning framework from RL [65]. The result is an agent that both

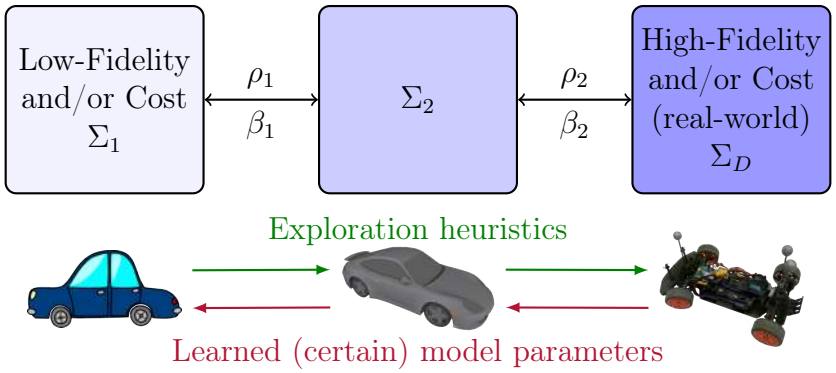


Figure 3-1: The MFRL architecture: a multi-fidelity chain of simulators and learning agents. Agents send exploration heuristics to higher-fidelity agents and learned model parameters to lower-fidelity agents. The environments are related by state mappings ρ_i and optimism bounds β_i. Control switches between learning agents, going to high-fidelity levels when an optimal policy is found, and to lower-fidelity levels when unexplored regions are encountered.

- uses information from lower-fidelity simulators to perform limited exploration in its current simulator, and
- updates the learned models of lower-fidelity agents with higher-fidelity data.

Unlike *unidirectional* methods that transfer heuristics only once to the real-world agent [67], the MFRL framework specifies rules for when the agent should move to a higher-fidelity simulator, as well as moving to a lower-fidelity level before over-exploring in a more expensive simulation. These rules and the transfer of values and data provide theoretical guarantees on convergence and sample efficiency. Specifically, the framework (1) does not run actions at high-fidelity levels that have been proven to be suboptimal at lower-fidelity levels, (2) minimizes (under certain conditions) the number of samples used in the real world and (3) polynomially limits the total number of samples used in all simulators. In addition, in the worst case, MFRL provably uses no more real-world samples than unidirectional transfer approaches.

The MFRL algorithm implements single-environment, model-based learners from the KWIK-Rmax family of algorithms at each level of simulation. The KWIK-Rmax algorithms explicitly track areas of the state space that are "known" or "unknown" and use an "optimism in the face of uncertainty" heuristic to fill in the unknown areas.

Theoretical results in this chapter are tied to the general KWIK framework and so apply not only to tabular models of environments, but also to a large class of representations such as linear and Gaussian-noise dynamics covered by the KWIK learning framework [65]. This theoretical link is illustrated by using a Dynamic Bayesian Network (DBN) representation [24] for model learning in two separate domains.

Section 3.1 formally introduces the multi-fidelity learning problem. Section 3.2 analyzes MFRL in the single-state, "bandit", setting, highlighting many of the core properties of the algorithm. MFRL is then presented and analyzed in the multi-state case in Section 3.3, with experiments in a multi-fidelity version of the puddle world domain [97]. Section 3.4 demonstrates learning using a DBN model of the transition dynamics and Section 3.5 discusses the use of generative models with random access to samples in the framework. The MFRL framework is demonstrated using robotic car experiments in Section 3.6, which are done with bandit, tabular, and DBN models. In all of these experiments, the car learns by utilizing a dynamics-free, input-based simulator and a reasonable-fidelity dynamics simulator, as illustrated in Fig. 4-1. The car experiments show that near-optimal driving policies can be found with fewer samples from the real car than unidirectional transfer methods or without using simulators.

The main contributions of this chapter are (1) introducing the MFRL framework for learning with multiple simulators, which is the first framework that dynamically chooses which simulator to perform actions in, (2) a theoretical analysis of the framework's sample complexity, and (3) several demonstrations of efficient learning on a robotic car with fewer real-world data points than unidirectional transfer or learning without simulators. These results demonstrate MFRL is a provably and practically efficient manager of the low and high quality simulators often available for robotics tasks.

MFRL is most applicable in situations where simulators are already available or where the cost of coding the simulator is outweighed by the increased learning speed. In my experience, most robotics domains construct simulators for reasons other than learning, such as during the initial hardware and software develop phases, and so

simulators are often readily available [12, 94].

This chapter is based on previously published work in [22, 23].

3.1 Assumptions

This section provides background on the relationship between the KWIK framework and MFRL. The assumptions made about the multi-fidelity simulators to ensure the discovery of a near-optimal policy are also described.

3.1.1 The KWIK Framework

The KWIK framework was formally introduced in Section 2.2. In MFRL, the KWIK-Rmax framework is used at each level of simulation. Unknown and known areas at each level are cataloged by KWIK learners as specified above. However, the uninformed $R_{\max}$ heuristic is instead replaced by value functions from lower-level simulators, filling in areas where learners at the current level predict $\perp$. Learned parameters of $\hat{T}$ and $\hat{R}$ from higher-fidelity levels are shared to increase the accuracy of lower-fidelity simulations.

The same base KWIK learners are used for the parameters of the MDP in MFRL. This allows MFRL to determine which simulators have accurate models for, and where the learned parameters of those models differ. Also, unlike single-environment KWIK-Rmax, the $\frac{R_{\max}}{1-\gamma}$ heuristic is replaced with a more informed heuristic based on interactions with other simulators.

The use of the KWIK framework also encourages the use of MFRL with compact (non-tabular) representations of T and R. That is, because the KWIK framework faciliates general learning algorithms for T and R, representations with far fewer parameters than the number of states can be used by an RL agent. For instance, despite a potentially infinite number of reachable states in a continuous environment, linear dynamics are KWIK learnable with only an $\tilde{O}(n^3)$ dependency on the number of dimensions n [65]. Thus, the sample complexity of KWIK-Rmax may be smaller than $|S|$ if T and R can be represented compactly. By building the MFRL framework

over KWIK learners, this ability to efficiently explore in large domains is gained, a fact used in Section 3.4 and later in the robotic car experiments.

3.1.2 Simulator Assumptions and Objectives

The notion of a *multi-fidelity simulator chain* used throughout this chapter is now defined and the assumptions made for using them in the MFRL architecture. The objectives of this work are then described in terms of minimizing sampling costs throughout the chain of simulators.

In this work, a *simulator* Σ is defined as any environment that can be modeled as an MDP. The complexity of such domains, $|\Sigma|$, is defined as the number of parameters of their corresponding T and R representations [65], which may be far smaller than the number of parameters need to represent S. Also, since S may differ between Σ_i and a higher-fidelity Σ_j (some variables may be absent in Σ_i), prior work in TL [106] is followed and a *transfer mapping* $\rho_i : S_i \mapsto S_j$ is assumed to exist. Specifically, it is assumed that $S_i \subseteq S_j$, that is, that the states available in lower simulators are a subset of those available at a higher level, and that ρ_i maps states in S_i to states in S_j, setting data uniformly across variables that exist in S_j, but not in S_i. For instance, in the robotic car simulations, the lowest-fidelity simulator (Σ_1) does not model rotational rate $\dot{\psi}$, so states in S_1 map to all states in S_2 with the same variable values except for $\dot{\psi}$. The reverse mapping ρ_i^{-1} only applies to states in S_j with a single *default* value of the missing variable ($\dot{\psi} = 0$ for the car). The set of available actions is also assumed to be the same in all simulators.

Fidelity is defined based on how much Σ_i *overvalues* the state/actions of Σ_j. Specifically, the fidelity f of Σ_i to Σ_j, with associated mapping ρ_i and tolerance β_i, is

$$f(\Sigma_i, \Sigma_j, \rho_i, \beta_i) = \begin{cases} -\max_{s,a} |Q^*_{\Sigma_i}(s,a) - Q^*_{\Sigma_j}(\rho_i(s),a)|, \\ \qquad \text{if } \forall s,a, [Q^*_{\Sigma_j}(s,a) - Q^*_{\Sigma_i}(\rho_i(s),a) \leq \beta_i] \\ -\infty, \quad \text{otherwise} \end{cases} \tag{3.1}$$

where $s \in S_i$. Intuitively, the fidelity of Σ_i to Σ_j is inversely proportional to the

maximum error in the optimal value function, given that Σ_i never undervalues a state/action pair by more than β_i. Otherwise, Σ_i is considered to have no fidelity to Σ_j. While there are many other possible definitions of fidelity (for instance based on T and R), this definition fits natural chains of simulators and facilitates efficient learning through the MFRL architecture. Note that this definition is not a distance metric (for instance, it is not symmetric), but rather describes the relationship between simulators based on optimistic differences in their value functions.

While it may seem restrictive, this relationship is fairly common in real-life simulators. For instance, in the car simulators used later, the lowest-fidelity simulator assumes that actions will have perfect outcomes, so aggressive maneuvers achieve their desired results. In higher-fidelity simulators, and eventually the real world, these optimistic values are replaced with more realistic outcomes/values. Hence, the simulators form an *optimistic chain*, formally defined as follows:

Definition 2. *An optimistic* multi-fidelity simulator chain *is a series of D simulators ordered $\Sigma_1, \ldots, \Sigma_D$, with Σ_D being the target task (real-world model) and $f(\Sigma_i, \Sigma_{i+1}, \rho_i, \beta_i) \neq -\infty$ for specified ρ_i and β_i.*

Intuitively, each simulator overvalues the optimal value function of the next higher simulator, with β compensating for any undervalues.

This ordering can be calculated recursively starting from level D (the target task, known a priori) and simply maximizing the fidelity of the remaining simulators to the simulator above and using the minimum β_i to maintain optimism at each level. To make use of this simulator chain and minimize the number of samples used, the following assumptions about the cost and accessibility of the simulators are also used.

Assumption 1. *A single step from simulator Σ_i has a larger (but polynomially bounded in $|\Sigma_{i-1}|$)* cost *than a sample from simulator Σ_{i-1}.*

Assumption 2. *Access to each simulator may be limited to running contiguous* trajectories *rather than having random access to a generative model or the model parameters.*

The first assumption states that each successively higher-fidelity simulator costs more to run per step than the one below it, but it is potentially not worth sampling every $\langle s, a \rangle$ at the lower level. For instance, an inverse kinematics model of a single robot takes less time to run than an ODE simulator of the robot and its environment, which is less costly than trials in the real world. The polynomial relationship enforces the fact that samples should be limited at the lower level. The second restriction states that access may not be available to the simulator parameters or the ability to sample state/action outcomes generatively. This is the case in the real world and in certain simulators (e.g. most commercial video games). Section 3.5 provides an algorithm for MFRL without Assumption 2 restricting generative access to the simulators, but first the algorithm is presented and efficiency results are shown in the more restricted trajectory-only case.

Given such simulators, the objectives are the following:

1. Minimize the number of suboptimal learning samples (with respect to Q^*) taken in Σ_D.

2. Ensure that, for any run of the agent with simulator Σ_i, only a polynomial number of steps (in $|\Sigma_i|$) are taken before near-optimal behavior (given constraints from higher-fidelity simulators) is achieved or control is passed to a lower-fidelity simulator.

3. Guarantee that there are only a polynomial (in $|\Sigma_{1,\ldots,D}|$ and D) number of switches between simulators.

Objective 1 skews the sampling burden to lower-fidelity simulators while objective 2 limits the sample complexity of the algorithm as a whole. Objective 3 is included to prevent excessive switching between the real-world robot and a simulator, as there may be significant start-up costs associated with the robot. This is particularly pertinent with dynamic robots that can not pause their actions to wait for a learning agent to update a model or policy based on simulated outcomes. These objectives are achieved by leveraging properties of KWIK algorithms at each level of simulation.

3.2 Multi-Fidelity Bandit Optimization

One of the simplest RL settings where exploration is studied is the *k-armed bandit* case, an episodic MDP with a single state, k actions (called *arms*), and $\gamma = 0$. A learner must choose actions to explore the rewards, eventually settling on the best arm, which it then exploits. A Multi-Fidelity RL algorithm for the bandit setting, which has several features of the full MFRL algorithm, is now presented.

3.2.1 A MF-Reward Learning Algorithm

Consider a chain of bandit simulators: at each level $d \in \{1, \ldots, D\}$ there are $|A|$ actions with expected rewards $R_d(a) \leq R_{d-1}(a) + \beta_d$. For a single simulator, a *base learner* is utilized that can update the estimates of each arm's reward. Here, KWIK reward learners $L_{R,d,a}$ are used with parameter m based on the required accuracy parameters for a single-arm learner: $\bar{\epsilon}$ and $\bar{\delta}$. These quantities are related to the overall accuracy parameters in the learning scenario and are defined later in Theorem 1. Specifically, a KWIK learner for a single bandit arm a at a single level d can be created by keeping track of the number of times $c_{d,a}$ that arm has been pulled. The algorithm then predicts the learned reward $L_{R,d,a}$ as

$$L_{R,d,a} = \begin{cases} \hat{R}_d(a) & \text{if } c_{d,a} \geq m = \frac{1}{2\bar{\epsilon}^2} \log(\frac{2}{\bar{\delta}}) \\ \perp & \text{otherwise} \end{cases} \tag{3.2}$$

where the value of m is set based on Hoeffding's inequality [42], assuming that the rewards are bounded on $[0, 1]$.[1] When $\perp$ is predicted, a loose upper bound for the possible payout of the action is $R_{\max}$, which will be used in the algorithm below.

Algorithm 3 presents the Multi-Fidelity Bandit Framework (MF-Bandit) for accomplishing objectives 1-3 using a KWIK learner that keeps track of empirical reward means $\hat{R}_d(a)$, number of pulls $c_{d,a}$, and upper bounds on the rewards $\hat{U}_{d,a}$ at fidelity level d. MF-Bandit also tracks the *informed* upper bound $U_{d,a}$, which is the minimum

[1] This assumption is made for notational convenience. An extra term is added if the range is larger than 1.

of $\hat{U}_{d,a}$ and the heuristic from the lower level: $U_{d-1,a} + \beta_{d-1}$ (lines 20 and 25). The algorithm also keeps track of whether the value of each action has converged ($con_{d,a}$), whether an optimal action has been identified ($closed_d$), and if the learned model has changed ($change_d$) at simulator level d.

Starting in Σ_1, the algorithm selects an action a^* greedily based on U_d and checks if learning at the current level is complete (line 7). Before executing the action, it checks to make sure the action has been tried sufficiently at the simulator below (line 8). If not, control is returned to level $d-1$ where actions are selected using values from d that are converged (lines 11-15). Otherwise, if learning at d is not finished, the action is taken and L_R and $\hat{U}$ are updated (lines 17-22). Once the optimal action has been identified, the algorithm moves up to the level $d+1$ (lines 23-26).

Algorithm 3 differs from unidirectional heuristic transfer [67] because it can backtrack to a lower-fidelity simulator when a previously identified optimal action performs poorly. In unidirectional transfer, information is only transferred from lower- to higher-fidelity levels, with no option for the learning agent to return to lower-fidelity levels to continue exploration. Effectively, backtracking asks the lower-fidelity learner to find a new optimal policy given additional knowledge from higher-fidelity simulators.

An example where this behavior is beneficial on the robotic car is when an optimal configuration of parameters in the simulator generates a path with tight turns, but data in the real world prove such settings cause the car to spin out. In such a scenario there is still information to be gleaned from the lower-level simulator by exploring policies *given* the knowledge of spinning out from above, which is exactly what transferring the learned parameters accomplishes. As shown below, this behavior is crucial in minimizing the number of actions taken at the highest level.

3.2.2 Bandit Examples

Examples are now presented to showcase various features of Algorithm 3. First, MF-Bandit is shown to find an optimal policy for Σ_D with far fewer samples in Σ_D than an algorithm without multiple simulators. Consider a Bandit problem with $|A| = 5$

Algorithm 3 Multi-Fidelity Bandit Framework

Input: A bandit simulator chain $\langle \Sigma, \beta \rangle$, Actions A, $R_{\max}$, Accuracy requirements ϵ and δ
Initialize: $\text{con}_{d,a}$, $\text{change}_d \coloneqq \text{false}, \forall a, d$
Initialize: KWIK learners $L_{R,d}(a, \bar{\epsilon}, \bar{\delta})$
Initialize: $d \coloneqq 1$, $\hat{U}_{d,a}$, $U_{1,a} \coloneqq R_{\max} \forall a$
for each timestep **do**
    Select $a^* \coloneqq \text{argmax}_a \, U_{d,a}$
    $\text{closed}_d \coloneqq \text{con}_{d,a^*} \vee a^*$ is definitely near optimal
    if $d > 1 \wedge \neg\text{con}_{d-1,a^*} \wedge \text{change}_d$ **then**
        // Return to level $d-1$
        $\text{change}_{d-1} \coloneqq \text{false}$
        for $a \in A$ **do**
            if $\text{con}_{d,a}$ **then**
                // Updated learner using $\hat{R}_d$ and $\hat{U}_d$
                Set $L_{R,d-1,a}$ based on $L_{R,d,a}$
                $\text{con}_{d-1,a}, \text{change}_{d-1} \coloneqq \text{true}$
        $d \coloneqq d - 1$
    else if $\neg\text{closed}_d$ **then**
        Execute a^* in Σ_d, Observe r.
        Update L_{R,d,a^*} // Update $\hat{R}_d(a^*)$, $\hat{U}_{d,a^*}$
        $U_{d,a^*} \coloneqq \min(U_{d,a^*}, \hat{U}_{d,a^*})$
        if L_{R,d,a^*} switched from $\perp$ to "known" **then**
            con_{d,a^*}, $\text{change}_d \coloneqq \text{true}$
    else if $d < D \wedge \text{closed}_d$ **then**
        // Chosen action already converged, go up
        Where $\neg\text{con}_{d+1,a}$: $U_{d+1,a} \coloneqq U_{d,a} + \beta_d$
        $\text{change}_{d+1} \coloneqq \text{false}$, $d \coloneqq d + 1$

arms and $D = 3$ simulators with bounded reward $[0, 1]$. The rewards for each of the 5 actions in each simulator are

$$\Sigma_1 = \{0.8, 0.8, 0.8, 0.8, 0.1\}$$

$$\Sigma_2 = \{0.8, 0.8, 0.6, 0.6, 0.1\}$$

$$\Sigma_3 = \{0.8, 0.6, 0.6, 0.6, 0.1\},$$

all with uniform random noise up to 0.1. Table 3.1 shows the results of running Algorithm 3 in this scenario with the KWIK bandit learner with $m = 20$. Results

Table 3.1: Samples used from the simulators when the optimal action in Σ_3 is also optimal in Σ_1 and Σ_2.

Sims Used	$\mathbf{\Sigma_1}$	$\mathbf{\Sigma_2}$	$\mathbf{\Sigma_3}$
Σ_1, Σ_2, Σ_3	100	80	40
Σ_2, Σ_3	–	100	40
Σ_3	–	–	100

Table 3.2: Samples used from the simulators when the optimal action in Σ_3 is not optimal in Σ_2.

Sims Used	$\mathbf{\Sigma_1}$	$\mathbf{\Sigma_2}$	$\mathbf{\Sigma_{3'}}$
Σ_1, Σ_2, $\Sigma_{3'}$	100	80	60
Uni-directional	100	60	80
$\Sigma_{3'}$	–	–	100

with only $\langle \Sigma_2, \Sigma_3 \rangle$ and only Σ_3 are also shown. Using both simulators or just Σ_2 produces a significant reduction in samples from Σ_3, and having Σ_1 helps limit the samples needed from Σ_2.

In the scenario above, the algorithm could potentially avoid backtracking because one of the optimal actions always remained the same at each level. But, consider the same scenario except with an alternate top level, $\Sigma_{3'} = \{0.4, 0.4, 0.6, 0.6, 0.1\}$. Now, neither of the optimal actions in Σ_2 are optimal in $\Sigma_{3'}$. Table 3.2 shows the results of Algorithm 3 in this case along with a version that does no transfer and a version that only performs unidirectional transfer (UNI) [67]. By allowing the algorithm to return to lower-fidelity simulators once the previously considered optimal action has been disproved at a higher level, valuable exploration steps in Σ_D are saved and the cost in terms of samples from the highest level is minimized.

3.2.3 Theoretical Analysis

The examples above show the importance of using lower-fidelity simulators to generate heuristics and using data from higher-fidelity simulators to make lower learning agents explore policies that are potentially more useful for the target task. This intuition

is now formalized in theoretical guarantees for the bandit case. Throughout these theoretical results the rewards of each arm are assumed, without loss of generality, to be bounded on $[0, 1]$. The base learner is also assumed to be the KWIK bandit learner as described earlier that predicts $\perp$ as the output of an action where it does not have m samples and otherwise predicts the empirical mean $\hat{R}(a)$. Multi-state versions of most of these guarantees are presented in later sections.

The theoretical results begin by focusing on objectives 2 and 3 from Section 3.1.2: limiting the number of suboptimal actions at each level and the number of samples overall. The following theorem provides these sample complexity results as well as guidelines for setting $\bar{\epsilon}$ and $\bar{\delta}$ in (3.2).

Theorem 1. *Algorithm 3 uses only a polynomial number of samples over all the levels, specifically using only* $O(\frac{|A|D^2}{\epsilon^2}\log(\frac{|A|^2D}{\delta}))$ *samples per run at level* d *and only changing* d *a maximum of* $|A|D$ *times.*

Proof. First, $\bar{\epsilon}$ and $\bar{\delta}$, the accuracy parameters for a learner of an individual action's mean reward at a particular level d are set. The following settings will be sufficient for the theorem:

$$\bar{\epsilon} = \frac{\epsilon}{2D} \tag{3.3}$$

$$\bar{\delta} = \frac{\delta}{|A|^2 D} \tag{3.4}$$

Instantiating m based on (3.2) with these values and applying Hoeffding's inequality [42] produces a sample complexity bound of $O(\frac{D^2}{\epsilon^2}\log(\frac{|A|^2D}{\delta}))$. There are $|A|$ arms at the given level, thus yielding the bound given in the theorem statement. This bound guarantees that, at each level d, if an arm is pulled m times, the rewards are learned with $\bar{\epsilon}$ accuracy with probability $1 - \bar{\delta}$. Applying a Union bound across actions gives a probability of failure at a specific level d of $\frac{\delta}{|A|D}$.

Each execution at a level must learn a new action's reward before moving up or down, and once an arm's value is set from above it cannot be sampled at the current level. Therefore, by the Pigeonhole principle, there can be at most $|A|D$ level changes (changes to d). Applying a Union bound across these $|A|D$ possible runs, each with

failure probability $\frac{\delta}{|A|D}$, the total probability of failure in the algorithm is δ, satisfying the overall failure bound. Since each level achieves accuracy of $\frac{\epsilon}{2D}$, the potential errors across levels are added to determine the total error due to pruning out actions based on rewards from below. This quantity is $2D * \frac{\epsilon}{2D} = \epsilon$, so the accuracy requirements are fulfilled with the sample efficiency described in the theorem statement. □

Next, objective 1 from Section 3.1.2 is considered, minimizing the number of samples used in Σ_D. This investigation begins with the following lemma, which is similar to Lemma 1 of [67], stating that no action is tried at a level beyond which it is dominated by the value of a^* in Σ_D.

Lemma 1. *In the bandit setting described above with actions $a \in A$ and levels $1, \ldots, D$, consider action a at level d. If a has been executed m times at level d, let $\mu_d = \hat{R}_d(a)$. Otherwise, set $\mu_d = U_d(a)$. If $\mu_d < R_D(a^*_D) - \sum_{\bar{d}=d}^{D-1} \beta_{\bar{d}} - \epsilon$, where a^*_D is the optimal action in Σ_D, then, with probability $1 - \delta$, a will not be attempted at or above level d.*

Proof. Set $\bar{\delta}$ and $\bar{\epsilon}$ based on (3.3) and (3.4) and again instantiate each individual arm's KWIK Learner with m based on (3.2). As above, Hoeffding's inequality ensures that, with probability $1 - \delta$, the learners will introduce no more than ϵ error throughout the process.

Now consider action a as above. At each level $d' \geq d$, by Definition 2, the expectation on the reward of a^*_D will satisfy

$$R_{d'}(a^*_D) \geq R_D(a^*_D) - \sum_{\bar{d}=d}^{D-1} \beta_{\bar{d}}. \tag{3.5}$$

Also, from the lemma's assumption,

$$\mu_d < R_D(a^*_D) - \sum_{\bar{d}=d}^{D-1} \beta_{\bar{d}} - \epsilon. \tag{3.6}$$

Combining these two inequalities shows that at level d', $R_{d'}(a^*_D) > \mu_d$. Since the upper bound $U_{d'a^*} \geq R_{d'}(a^*)$ with high probability, $U_{d'a^*} > \mu_d$. This means whenever the

algorithm enters level d', action a^*_D will be used before a. By Hoeffding's inequality, pulling arm a^*_D m times gives a mean reward estimate $\hat{R}_{d'}(a^*_D) \geq R_{d'}(a^*_D) - \bar{\epsilon}$ with high probability, so there will also be no need to pull arm a at level d' after collecting these m samples. □

It is now shown that only actions that *must* be tested in Σ_D are used there (objective 1 from Section 3.1.2).

Theorem 2. *With probability $1 - \delta$, any action a attempted in simulator Σ_D (the real world) by Algorithm 3 is either near optimal (within ϵ of $R_D(a^*)$) or could only be shown to be suboptimal in Σ_D.*

Proof. Consider any action a executed in Σ_D and its associated μ_{D-1} values at the next lower-fidelity simulator as defined in Lemma 1. From Lemma 1,

$$\mu_{D-1} \geq R_D(a^*_D) - \beta_{D-1} - \bar{\epsilon}. \tag{3.7}$$

Otherwise, with high probability, a would have been pruned and not executed in Σ_D. If a is near optimal then the problem is complete. If not, then by line 8 of Algorithm 3 the algorithm must have taken action a at level $D-1$ and, with high probability, found

$$U_D(a) = \hat{R}_{D-1}(a) \geq R_D(a^*_D) - \beta_{D-1} - \bar{\epsilon}. \tag{3.8}$$

Therefore, the only way to determine that a is sub-optimal is to execute it in Σ_D. □

Thus, Algorithm 3 never tries actions that can be pruned by lower-fidelity simulators. A corollary of this theorem is that, in the *worst case*, MF-Bandit uses no more samples in Σ_D than unidirectional transfer methods use.

3.3 Multi-Fidelity Reinforcement Learning

This section instantiates the principles of generating heuristics from lower-fidelity simulators and sending learned model data down from higher-fidelity simulators in the multi-state, cumulative discounted reward RL case.

Algorithm 4 MFRL (MF-KWIK-Rmax)

Input: A simulator chain $\langle \Sigma, \beta, \rho \rangle$, $R_{\max}$, Planner P, accuracy parameters $\langle \epsilon, \delta, m_{known} \rangle$
Initialize: $\text{change}_d \coloneqq \text{false}, \forall d$
Initialize: $2D$ KWIK learners $L_{R,d}(\bar{\epsilon}, \bar{\delta})$ and $L_{T,d}(\bar{\epsilon}, \bar{\delta})$
Initialize: $\hat{Q}_0 \coloneqq \frac{R_{\max}}{1-\gamma}$, $\hat{Q}_1(s,a) \coloneqq \textsc{Plan}(1)$
Initialize: $d \coloneqq 1$, $m_k \coloneqq 0$
for each timestep and state s **do**
    Select $a^* \coloneqq \text{argmax}_a \hat{Q}_d(s,a)$
    if $d > 1 \wedge \text{change}_d \wedge (L_{T,d-1}(\rho_{d-1}^{-1}(s), a^*) = \perp \vee L_{R,d-1}(\rho_{d-1}^{-1}(s), a^*) = \perp)$ **then**
        // Return to level $d-1$
        $\hat{Q}_{d-1} \coloneqq \textsc{Plan}(d-1)$
        $m_k \coloneqq 0$, $d \coloneqq d-1$
    else
        Execute a^* in Σ_d, Observe r, s'.
        if $L_{R,d}(s,a^*) = \perp \vee L_{T,d}(s,a^*) = \perp$ **then**
            $m_k \coloneqq 0$
            Update $L_{R,d}$ and/or $L_{T,d}$ that predict $\perp$
        else
            $m_k \coloneqq m_k + 1$
        if $L_{R,d}(s,a^*)$ or $L_{T,d}(s,a^*)$ is now *known* **then**
            $\hat{Q}_d \coloneqq \textsc{Plan}(d)$, $\text{change}_d \coloneqq \text{true}$
    if $d < D \wedge m_k = m_{known}$ **then**
        // Go up to level $d+1$
        $\hat{Q}_{d+1}(s,a) \coloneqq \textsc{Plan}(d+1)$
        $m_k \coloneqq 0$, $\text{change}_d \coloneqq \text{false}$, $d \coloneqq d+1$
procedure $\textsc{Plan}(d)$
    // Call planner P using highest-fidelity data available
    For any (s,a) let $d^*(s,a)$ be largest d such that $L_{R,d^*}(s,a) \neq \perp \wedge L_{T,d^*}(s,a) \neq \perp \wedge d^* \geq d$
    if d^* does not exist **then**
        $d^* \coloneqq d$
    $\hat{Q}_d \coloneqq P(\langle S_d, A, L_{R,d^*}.\hat{R}, L_{T,d^*}.\hat{T}, \gamma \rangle, Q_{d-1} + \beta_{d-1})$
end procedure

3.3.1 The MFRL Algorithm

Algorithm 4 shows the MFRL framework, which takes as input a simulator chain, the maximum reward $R_{\max}$, a state-mapping between simulators $\rho_1, \ldots, \rho_{D-1}$, a planner P, and accuracy requirements ϵ, δ, and m_{known}. MFRL is similar to MF-Bandit but now the heuristic passed to higher-fidelity simulators is the Q-function, and both the

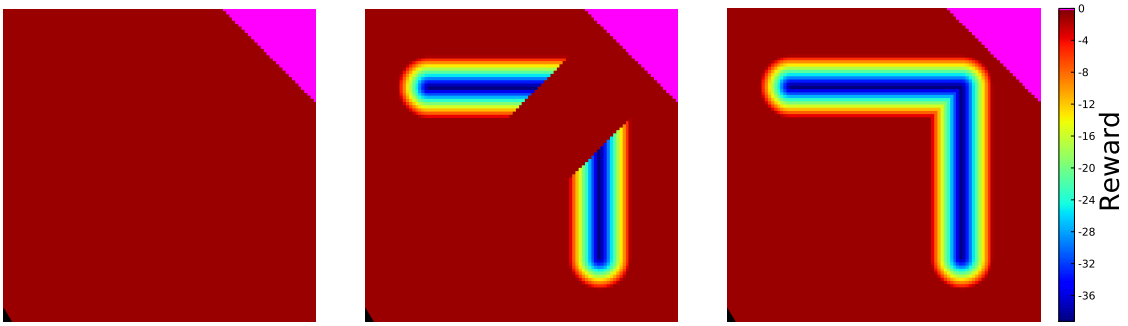
reward and transition functions are passed to lower-fidelity simulators.

The algorithm begins by initializing the variables d, m_k and $change_d$ and the base KWIK learners $L_{T,d}$ and $L_{R,d}$, parametrized by $\bar{\epsilon}$ and $\bar{\delta}$. These KWIK learners are proxies for the learned transition and reward functions at level d, which are denoted $L_{T,d}.\hat{T}$ and $L_{R,d}.\hat{R}$, respectively. More specifically, whenever a state/action pair is "known" (neither L_R nor L_T predict $\perp$), $\hat{T}$ and $\hat{R}$ model the predictions of the KWIK learners. If one of the learners predicts $\perp$, the Q-values from the previous level will be inserted as a heuristic to encourage exploration. The Q-values for the lowest-fidelity simulator are set optimistically using $\frac{R_{\max}}{1-\gamma}$, and the agent begins choosing actions at that level.

The agent chooses actions for the current state greedily. If, according to the KWIK model learners, the selected state/action pair is not known at level $d-1$, and a change has been made at the current level, the algorithm backtracks one layer of fidelity (lines 8-11). Otherwise, the action is executed and L_T and L_R at the current level are updated. If control returns to the lower-fidelity simulator, this forces the corresponding agent to explore policies that might be optimal given the dynamics of the higher-fidelity simulator. Note that while backtracking after seeing a single state/action pair that is not known at level $d-1$ is theoretically correct, in the experiments the learning agent typically waits until $m_{unknown}$ such "unknown" state/action pairs are encountered, which helps control sampling at lower-fidelity simulators.

If the model parameters change, the $change_d$ flag is also set and the planner recalculates the Q-values using the PLAN subroutine (lines 25-31). Information from the highest-fidelity level that does not predict $\perp$ is used in the planner. If no such level exists, then the planner uses the current heuristic value (passed from a lower-fidelity level).

Finally, the convergence check (line 21) determines if MFRL should move to a higher-fidelity simulator. In the multi-state case, simply encountering a known state does not indicate convergence, as states that are driving exploration may be multiple steps away. Instead, Algorithm 4 checks if the last m_{known} states encountered at the



(a) No puddle or noise (Σ_1) (b) Some puddle and noise (Σ_2) (c) Full puddle and noise (Σ_3)

Figure 3-2: $\Sigma_1, \ldots, \Sigma_3$ for the puddle world. The learning agent starts in the lower left corner, taking actions until it reaches the pink goal region at the top right. Σ_1 has no puddle. Σ_2 has most of the puddle, but the optimal policy in Σ_1 can bypass these puddle portions. Thus, the optimal policy in Σ_1 is still optimal in Σ_2, but not in Σ_3.

current level were known according to the base learners. For theoretical purposes, m_{known} can be set to the following quantity, which is the number of steps needed to show that an MDP comprised only of the "known" states sufficiently models the environment (see Theorem 4 of [64]).

$$m_{known} = \frac{1}{1-\gamma} \ln \left(\frac{4\left(R_{\max} - R_{\min}\right)}{\bar{\epsilon}(1-\gamma)} \right) \tag{3.9}$$

This quantity guarantees that, if the true value function is significantly different from the value of the current policy in the "known" MDP, with high probability an unknown state will be encountered during the run of m_{known} states. Further details about the theoretical properties are given in Theorem 3 below. In practice, a smaller value is usually adequate to check for convergence and move to a higher-fidelity simulator, but (3.9) can be used to ensure theoretical correctness.

3.3.2 Puddle World with MFRL

The behavior of MFRL is illustrated in a variant of the puddle world domain [99] with multi-fidelity simulators, shown in Fig. 3-2. A puddle world agent moves in one of four diagonal directions with a step cost of -1 (0 at the goal) and high negative

Table 3.3: Parameters used for the puddle world results.

Sim	$\mathbf{m_{L_R}}$	$\mathbf{m_{L_T}}$	$\boldsymbol{\sigma}$	$\mathbf{m_{known}}$	$\mathbf{m_{unknown}}$	$\boldsymbol{\beta}$
Σ_1	1	3	0	75	–	0
Σ_2	1	4	0.01	75	20	0
Σ_3	1	5	0.02	75	20	–

rewards in the puddle. The puddle world is implemented with diagonal actions and $\gamma = 0.95$, so the optimal policy in Σ_3 is generally to skirt along the outer edges of the puddle, while the optimal policy in Σ_2 is to move diagonally between the puddles. The puddle world is 1 unit square and each step moves 0.1 units plus some zero mean Gaussian noise. For learning and policy evaluation, the world is discretized in the two dimensions into a 10 by 10 grid array, yielding a state-action space of size $10 \times 10 \times 4 = 400$.

MFRL is tested in the presence of two lower-fidelity simulators with respect to the "real" puddle world. The base level Σ_1 contains no puddle and has deterministic actions. The middle level Σ_2 contains some of the puddle and has noisy actions, but has an opening in the worst reward region from Σ_3. This creates a scenario where an optimal policy in the low-fidelity simulator is poor in Σ_3 but still contains significant useful information, such as the puddle portions in Σ_2 and the goal location. The top level Σ_3 contains the full puddle and the full action noise.

Fig. 3-3 shows learning curves from this experiment with Table 3.3 showing the parameters used during the experiments. Here, m_{L_T} and m_{L_R} denote the number of times a state/action pair must be observed before the transition and reward functions, respectively, are known. MFRL is compared with unidirectional transfer (UNI), no-transfer (RMAX) and prioritized sweeping (PS) [97]. The m_{L_T} and m_{L_R} parameters were the same in each of the three Rmax-based algorithms, ensuring a consistent comparison. In PS, the agent explores using an ϵ-greedy policy ($\epsilon = 0.1$) and optimistic initialization while evaluation occurs greedily.

MFRL performed the best, with some negative transfer at the beginning from the "shortcut" in Σ_2. As the learning agent encounters the real puddle in Σ_3, it

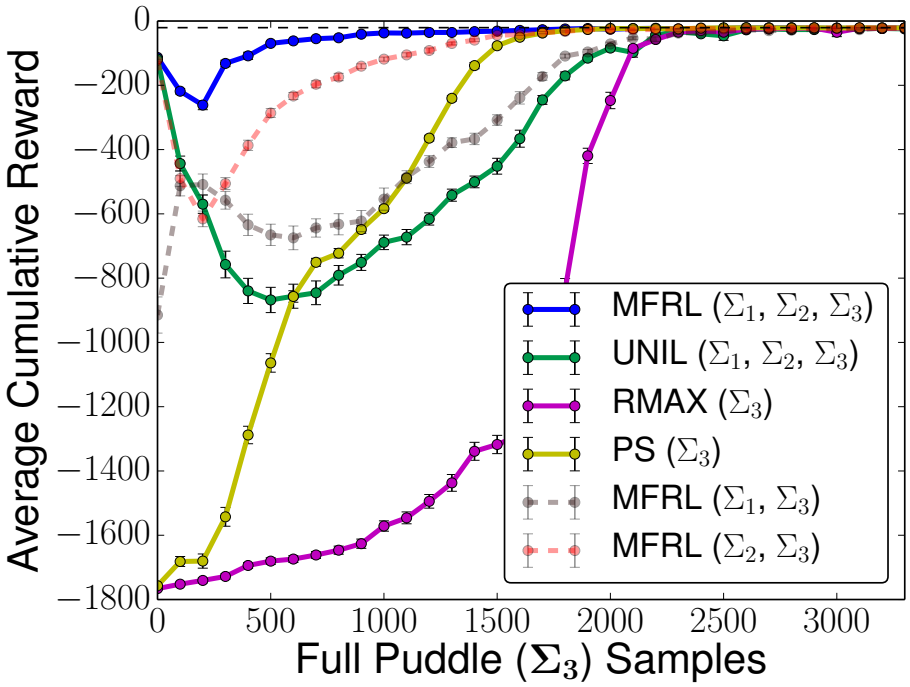
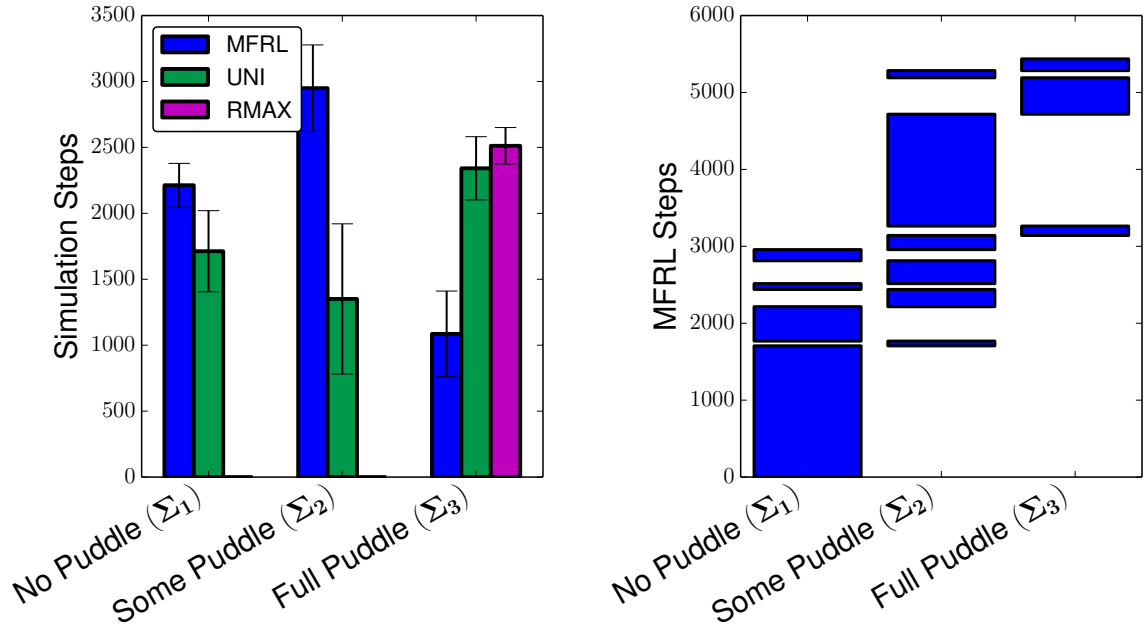


Figure 3-3: During learning, MFRL consistently outperforms unidirectional transfer (UNI), no-transfer Rmax (RMAX), and prioritized sweeping (PS) at Σ_3. Note that these are only the samples from Σ_3. Each point is an average of 1000 learning runs (standard errors shown). Greedy policies are evaluated 60 times, each capped at 600 steps.

starts exploring areas of the state-action space that where not explored in Σ_2. This exploration results in several level changes where the learning agent in Σ_2 plans using information from Σ_3, causing that lower-fidelity agent to find a way around the puddle. The result is a consistent and significant improvement over unidirectional transfer throughout learning. Note that backtracking from higher- to lower-fidelity levels occurs only when there is still uncertainty at the lower-fidelity levels. If no uncertainty exists at the lower-fidelity levels, then returning to those levels will not yield better Q-values to guide exploration at the higher-fidelity levels.

MFRL also outperforms naïve model-based RL algorithms such as PS despite PS performing better than Rmax alone. Fig. 3-3 also shows that even in the presence of only one simulator (MFRL (Σ_1, Σ_3) and MFRL (Σ_2, Σ_3)), MFRL still outperforms standard Rmax.



(a) Average Samples

(b) Example MFRL algorithm run

Figure 3-4: (a) At the top level, MFRL requires fewer than half the samples needed by both unidirectional transfer (UNI) and no-transfer Rmax (RMAX). This is accomplished by transferring some learning burden to the other simulators. Each bar represents an average of 1000 learning runs with standard deviations shown. (b) An example run of the MFRL algorithm. After initially exploring the top level, the algorithm spends a significant number of samples exploring the rest of the middle level, thus decreasing the required samples at the top level.

If Σ_2 and Σ_3 were switched so that Σ_2 is the "real" world, the optimistic chain assumption would be violated. In this case, the learning agent would explore all of the puddle in Σ_3, converging to a policy that moves around the puddle towards the goal. When the agent transitioned to Σ_2, it would not explore and find the "shortcut" between the puddles, but instead continue with the optimal policy from Σ_3. Despite not converging to the true optimum, a good policy in Σ_2 would be found with very few steps needed from that level.

The improvement of MFRL over unidirectional transfer is accomplished while using more than 50% fewer samples in the top level as seen in Fig. 3-4(a). In fact, unidirectional transfer takes almost as long as the no-transfer case to consistently find the optimal policy, primarily because even with unidirectional transfer, the learning agent still needs to explore the majority of the state/action space in Σ_3 before finding

the path around the puddle. Fig. 3-4(b) shows a single run of MFRL with bars showing samples in each of the 3 simulators. The MFRL agent relied heavily on Σ_1 initially and swapped back to Σ_2 several times while learning at Σ_3, gathering crucial information through this lower cost simulator.

A video showing the progression of the MFRL framework in the puddle world is available at `https://youtu.be/-UYu0cGER-s`.

3.3.3 Theoretical Analysis

Similar to the bandit examples earlier, the puddle world examples above enhance intuitions about the behavior of Algorithm 4. The behavior of Algorithm 4 is now formally quantified by extending the theoretical results from the bandit case. Throughout this section the standard assumption is made, without loss of generality, that the rewards of the MDP are in the range $[0, R_{\max}]$. The parameter m_{known} is also set by (3.9) and instantiate the accuracy parameters of the KWIK MDP learners as

$$\begin{aligned} \bar{\epsilon} &= \frac{\epsilon}{4(D+1)} \\ \bar{\delta} &= \frac{\delta}{4(D+2D|\Sigma|)} \end{aligned} \tag{3.10}$$

where $|\Sigma|$ is the maximum number of parameters used to describe a simulator in the optimistic chain. Typically, $|\Sigma|$ is the number of parameters representing the transition and reward functions in the real world MDP. Furthermore, throughout this section the variables in each Σ_i are assumed to be the same; that is, ρ_i is the identity mapping. This assumption is revisited at the end of the section, where it is shown that β_i may need to be increased to preserve the stated properties when some simulators have missing variables.

It should be noted that, like all algorithms in the Rmax family, the sample complexity bounds presented here are meant to show the scalability of the algorithm with respect to problem size, not for actually setting the known-ness parameters in practice. That is, the analysis shows that as problems grow larger (with respect to $|S|$, $|A|$, ϵ, or δ), the number of samples needed increases only polynomially in those

terms. However, because of the loose bounds used, practitioners almost always choose far lower values for known-ness based on the noisiness and risk in their environments. It is likely that setting known-ness parameters lower in this way invalidates the theoretical worst-case sample complexity, but in most problems these algorithms have proven remarkably robust to these lower values. Indeed, recent theoretical results [66] indicate that tighter bounds can be achieved by considering such features, supporting these more aggressive known-ness parameters.

The analysis begins by analyzing the sample complexity of a run at each level of simulation and the total number of level changes in the MFRL algorithm. As with Theorem 1, this Theorem covers objectives 2 and 3 from Section 3.1.2.

Theorem 3. *Algorithm 4 uses only a polynomial number of samples in* $\langle |\Sigma|, \frac{1}{\epsilon}, \frac{1}{\delta}, \frac{1}{1-\gamma} \rangle$ *per run at level* d *and only changes* d *a maximum of* $(D + 2D|\Sigma|)$ *times.*

Proof. For the first portion, note that, when entering a level, the KWIK-Rmax algorithm at each instantiation makes only a polynomial number of suboptimal steps (where $V^{\pi}(s) < V^{*}(s) - \epsilon$). This bound on the number of suboptimal steps is a known property of the KWIK-Rmax algorithm [64, 65]. Also note that the derivation of this property shows that "unknown" states are only encountered a polynomial number of times. In fact, this is exactly $B(\epsilon, \delta)$ where B is the KWIK bound for learning the transition and reward functions.

Since the number of samples used at each level should be limited and not just the number of suboptimal steps, the number of *optimal* steps taken at level d should also be bounded. For this, m_{known} is utilized as described in (3.9). There can be at most $m_{known} - 1$ steps between encounters with "unknown" states, and, since the latter can only occur $B(\bar{\epsilon}, \bar{\delta})$ times, this results in at most $B(\bar{\epsilon}, \bar{\delta})(m_{known} - 1)$ samples.

The number of steps before moving to another level also need to be limited. For moving down to level $d - 1$, a state/action pair at $d - 1$ can only be unknown if it is unknown at level d. Therefore, the bound on the number of steps before this might happen is the same as the number of steps before learning all the unknown states: $B(\bar{\epsilon}, \bar{\delta})(m_{known} - 1)$. For moving up to level $d + 1$, a maximum of $B(\bar{\epsilon}, \bar{\delta})(m_{known} -$

$1) + m_{known}$ steps is needed to reach the conditions for going to the next higher level of fidelity.

Next, it is shown that if m_{known} known states in a row are encountered, then, with high probability, the optimal policy has been identified at the current level. This property has been shown previously in Theorem 4 of [64], which is briefly recounted here. First, an escape probability $Pr(W)$ of encountering an unknown state is defined. From this follows

$$V^{\pi}(s_t, m_{known}) \geq V^{\pi}_{M_k}(s_t, m_{known}) - Pr(W)V_{\max}, \tag{3.11}$$

where $V^{\pi}(s_t, m_{known})$ is the value of running policy π from s for m_{known} steps. By using the closeness of this finite-horizon sum to the infinite discounted sum and several properties of the KWIK-Rmax algorithm, Li showed that

$$V^{\pi}(s_t, m_{known}) \geq V^{*}(s_t) - \frac{3\epsilon}{4} - Pr(W)V_{\max}. \tag{3.12}$$

If $Pr(W) < \frac{\epsilon}{4V_{\max}}$, then there is a very low probability of reaching an unknown state, and it can be shown that the current policy is near-optimal. Otherwise, $Pr(W) \geq \frac{\epsilon}{4V_{\max}}$, which means that the probability of reaching an unknown state is high, and, with high probability, an unknown state will be encountered before m_{known} known states are seen in a row.

For the number of possible changes to d, $D + 2D|\Sigma|$ is an upper bound on the number of level changes because each backtrack can only occur when at least one parameter is learned, and the number of parameters in the system is $|\Sigma|D$. The number of "up" entries can only be D more than he number of down entries, giving us $D + 2D|\Sigma|$ level changes. □

Because Theorem 3 uses the KWIK framework to derive its sample complexity guarantees and m_{known} in Equation 3.9 depends only on γ, ϵ and δ, these results apply to MDP learning algorithms using a large number of representations. That is, while the bounds apply to the tabular MDP case where each $\langle s, a \rangle$ is learned independently,

they also apply to any KWIK learnable class, including Dynamic Bayesian Networks and linear transitions [65]. Such generalizations will be used in Section 3.4.

While Theorem 3 covers the convergence and sample complexity of MFRL, the worst case bounds in the theorem above may require an agent to do the same amount of learning in Σ_D as an agent without any simulators. This is necessary because in the worst case the simulators provide no useful information to the real world (for instance, assigning the same Q-value to each state-action pair). To understand the more general theoretical benefits of MFRL and its relationship to the unidirectional transfer method, Lemma 1 and Theorem 2 are expanded to the multi-state case, covering objective 1: limiting steps in Σ_D.

The following extension of Lemma 1 shows that, for a given state s, if an action's Q-value is definitely dominated by the optimal action at s, with high probability the action will not be attempted at the current or higher-fidelity levels.

Lemma 2. *Consider state s and action a at level d and let $\mu_d = \hat{Q}_d(s,a)$ if $\hat{T}(s,a)$ and $\hat{R}(s,a)$ are known. Otherwise, $\mu_d = Q_{d-1}(s,a)$. If $\mu_d < Q_D(s,a_D^*) - \sum_{\bar{d}=d'}^{D-1} \beta_{\bar{d}} - \epsilon$ where a_D^* is the optimal action for s in Σ_D, then, with probability $1-\delta$, a will not be attempted in s at or above level d.*

Proof. The proof of this property is similar to the proof of Lemma 1 so the argument is merely outlined here.

The parameters $\bar{\delta}$ and $\bar{\epsilon}$ are set based on (3.10) to ensure the accuracy of the known model parameters throughout the learning process. Through the simulation lemma (Lemma 4 of [50]) it is known that a learned MDP with ϵ-accurate parameters will model the value function with comparable loss (adding some additional, but still polynomial, terms).

Therefore, a chain of inequalities similar to those in Lemma 1 can be formed, but now replacing R with Q from the lower-fidelity simulators. The situation in each state becomes exactly the same as in the single-state bandit case, where an action will not be chosen if its optimistic value has fallen beyond the optimal action's Q-value as stated in the lemma. □

Unlike the bandit case, the lemma above does not translate directly into a guarantee on the *necessity* of an action in Σ_D because the KWIK-Rmax algorithm is not guaranteed to be the most efficient exploration algorithm possible. It does, however, guarantee that every suboptimal step is, with high probability, leading towards a learning experience (see Lemma 13 of [65] and Theorem 4 of [64]). Therefore, the following property of Algorithm 4 can be stated based on the lemma above and this guarantee of a future learning experience.

Property 1. *With probability $1 - \delta$, any action a attempted in state s of simulator Σ_D (the real environment) by Algorithm 4 is either near optimal (within ϵ of $V_D^*(s)$) or will eventually lead to an unknown state that is either not learned about in levels below or that needs to be learned about in Σ_D.*

Thus, Property 1 means that MFRL tries actions in the real world that either lead to needed exploration or backtracking to a lower-fidelity level. Property 1 also means that MFRL will, with high probability, enter no more unknown states in Σ_D than a unidirectional transfer method with the same base learner and architecture. Both approaches will be drawn to areas that fit the two cases above. However, by returning to Σ_{D-1}, MFRL can potentially learn about areas that were not visited earlier in Σ_{D-1} and thereby prune actions as in Lemma 2. By contrast, the unidirectional case can only learn about such areas in Σ_D. Because of the optimistic chain assumption, obtaining data from the lower-fidelity simulator can only strengthen the heuristic and prune more actions. Therefore, in the case where MFRL returns to Σ_D in the exact state it was in before it decided to use the simulators, MFRL will make no more (worst case) sub-optimal steps than the unidirectional approach with the same base learners.

However, in cases where the agent "resets" to the start state of Σ_D upon returning from the lower-fidelity simulators, it is possible for MFRL to make more suboptimal steps than the unidirectional algorithm because it needs to retake potentially suboptimal steps to reach the state it was in when it decided to return to the lower-fidelity simulators. However, this increase in suboptimal steps is at most a multiple of the

polynomial number of possible entries into the simulator (covered in Theorem 3) and will usually be offset by better information gained in the lower simulators.

3.3.4 MFRL Properties with Missing Variables

The analysis now returns to the assumption, made in Section 3.3.3, that each Σ_i contains the same set of variables. In cases where a variable v exists in Σ_i but is missing in Σ_{i-1}, one value of the variable is designated as the *default* value and only parameters learned in states with this value can be used by the planner in Σ_{i-1}. For instance, in the robotic car, if a simulator is missing the "wheel slip" variable, only non-slip dynamics should be used by the planner in that simulator. However, because ρ_i is potentially one-to-many from Σ_{i-1} to Σ_i, the Q-values passed *up* to Σ_i could cause an undervaluation of some states in Σ_i.

Consider the wheel-slip example with states s_0 and s_1 in Σ_i where s_0 has no slip and s_1 has slip. Suppose $V^*(s_1) = 1$ and $V^*(s_0) = 0$; that is, the slipping state is more valuable. If the agent experiences s_0 first and then returns to Σ_{i-1}, s_0 in Σ_{i-1} may set $V^* = 0$, the value in the simulator above. Now, when the agent returns to Σ_i, 0 will be used as the heuristic value for s_1. Unfortunately, this is an *underestimate* of the value function in the slippery state, invalidating the KWIK-Rmax assumptions.

However, MFRL has machinery to compensate for such undervaluation. Specifically, β_{i-1} can be used to increase the transferred heuristics in this case. All of the properties described above hold in the case of missing variables as long as β_{i-i} is set high enough at each level to guarantee both Definition 2 *and* that $Q_i^*(s_1, a) + \beta_{i-1} > Q_{i-1}^*(s_0, a)$ where s_1 and s_0 are states in Σ_i that were aliased in Σ_{i-1}.

In summary, missing variables in some of the simulators add complications to the algorithm's exploration strategy, but they can be overcome by utilizing the existing β_i parameters already built into the framework. Future work could investigate the effects of a non-unity ρ_i mapping in greater detail.

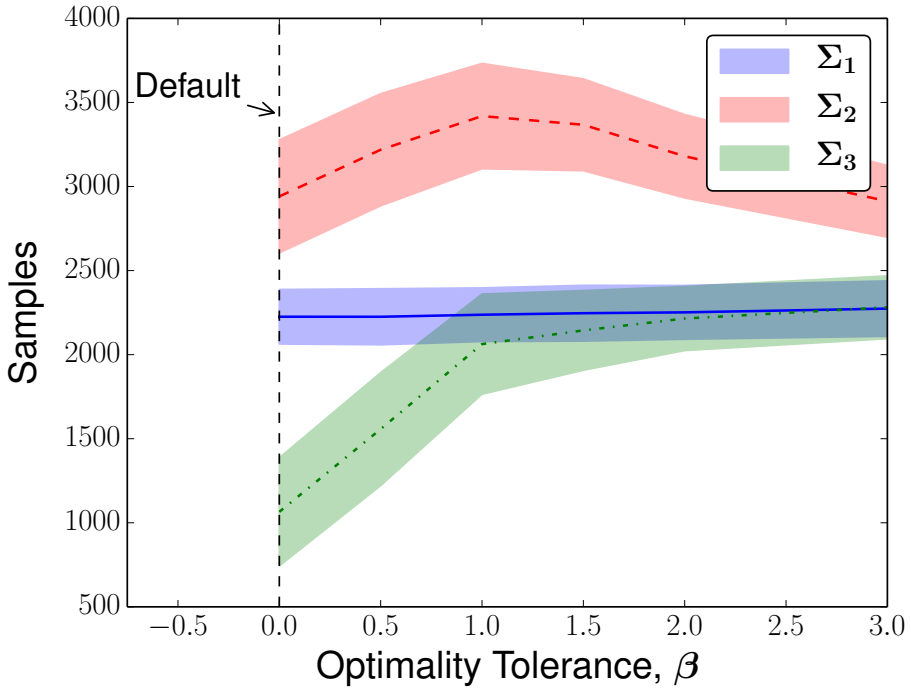


Figure 3-5: The true optimality tolerance for the puddle world example is $\beta = 0$. As this value is increased, the top level learns less and less from the other levels. Eventually, no useful information is passed to the top level and the algorithm needs as many samples in the full puddle world (Σ_3) as it needed in the no puddle grid world (Σ_1). Bold lines denote means and shaded areas show one standard deviation.

3.3.5 Sensitivity of MFRL Parameters

This section demonstrates how the input parameters β_i, m_{known}, and $m_{unknown}$ affect the performance of Algorithm 4. Fig. 3-5-3-7 show the performance of the algorithm in the puddle world domain as each of the three parameters is varied from its nominal value. The nominal values used in the domain are $\beta = \beta_1 = \beta_2 = 0$, $m_{known} = 75$, and $m_{unknown} = 20$. Each data point in the graphs is an average of 1000 runs of the MFRL algorithm in the puddle world domain, with the shaded regions denoting one standard deviation.

In Fig. 3-5, the simulator tolerance β is increased significantly from the true value of $\beta = 0$. The algorithm converges to the optimal policy for all values of β; however, as the tolerance is increased, the number of simulator steps at Σ_D increases since the algorithm is unable to prune as many suboptimal state/action pairs. Eventually, Σ_3 no longer gains any information from the simulators below it and the number of

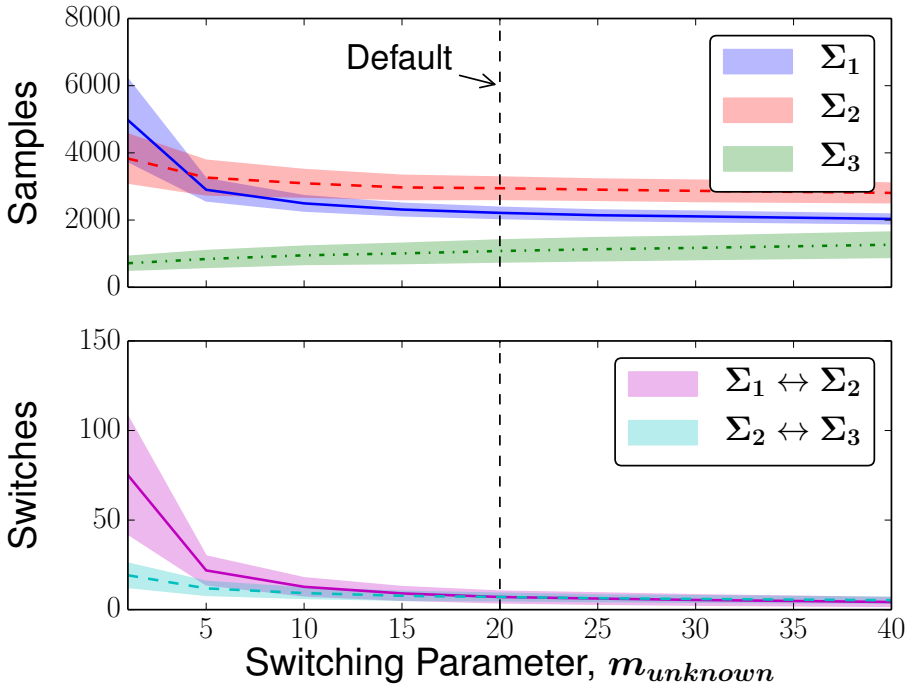


Figure 3-6: The parameter $m_{unknown}$ controls how many unknown states must be seen at Σ_i before moving down to Σ_{i-1}. The number of samples used at the top level monotonically increases with increasing $m_{unknown}$. However, the number of switches between simulator levels decreases with increasing $m_{unknown}$. Thus, setting $m_{unknown}$ is determined by the trade-off between the cost of obtaining samples at the top level versus the cost of switching environments, as certain robots may have substantial initialization costs. The default value of $m_{unknown}$ used in the experiments is $m_{unknown} = 20$. Bold lines denote means and shaded areas show one standard deviation.

samples needed to converge at Σ_3 quickly approaches the number needed to converge at Σ_1. However, even with an inaccurate β value, the MFRL algorithm converges to the correct policy using no more top-level samples than had learning been performed at the top level alone.

Fig. 3-6 shows how the number of samples used at the top level increases with increasing $m_{unknown}$. As with changing β, the MFRL algorithm converged to the optimal policy for all values of $m_{unknown}$. For the theoretically correct value of $m_{unknown} = 1$ (see Section 3.3.1), the number of samples used at Σ_3 is minimized, but at the expense of a high number of simulator switches during the progression of the algorithm and a large number of samples used at the other simulator levels. Setting $m_{unknown}$

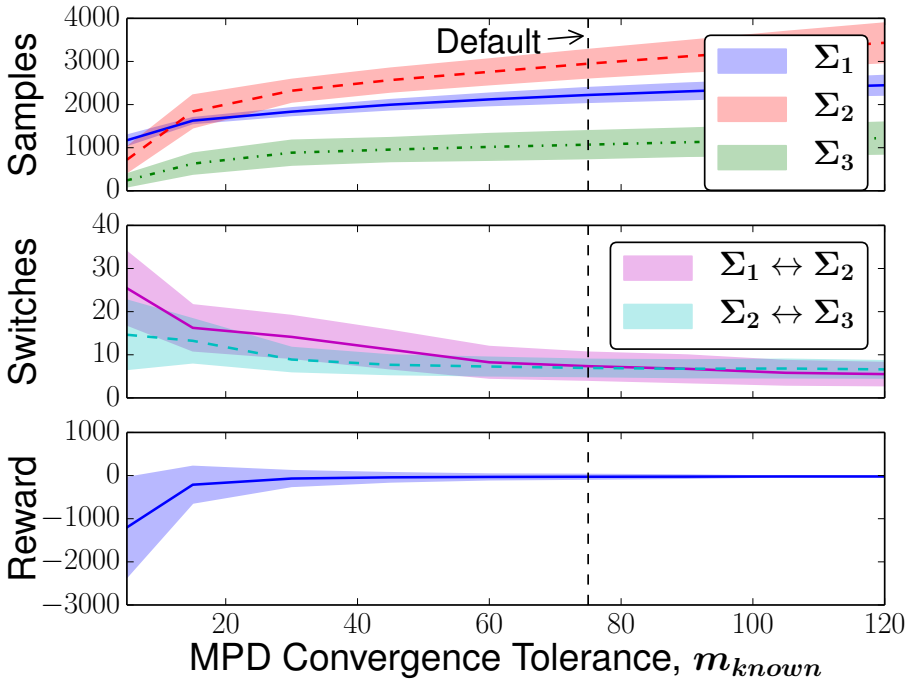


Figure 3-7: The parameter m_{known} controls how many known states must be seen in a row at Σ_i before moving up to Σ_{i+1}. The number of samples used at the top level monotonically increases with increasing m_{known}. As seen by the average reward, selecting m_{known} too low can lead to under-exploration and suboptimal policies. The default value of m_{known} used in the experiments is $m_{known} = 75$. Bold lines denote means and shaded areas show one standard deviation.

determines a trade-off between minimizing the samples at Σ_3 and minimizing the number of simulator switches. For many robotic scenarios it may be advantageous to set $m_{unknown} > 1$ if the start-up costs of running the robot are significant. Also, setting $m_{unknown}$ low for a particular expensive simulator can help limit the number of samples needed from that level.

Finally, Fig. 3-7 shows the algorithm performance as the convergence parameter m_{known} is varied. Notice that setting m_{known} too low causes the algorithm to converge prematurely to a suboptimal policy, while setting the value too high wastes unnecessary samples converging to the same policy.

Empirically, the MFRL algorithm is robust to variations in the tuning parameters of β, $m_{unknown}$, and m_{known}. The algorithm converges to the optimal policy at the top level for all values of the tried parameters, except for when m_{known} is set artificially

too low. While no firm guidelines exist yet for determining these parameters for new domains, in general, β should be set based on approximately how optimistic two adjacent simulators are, m_{known} based on how many successive "known" samples should be observed before being confident that further exploration is not needed, and $m_{unknown}$ based on how costly it is to backtrack to a lower-fidelity simulator.

3.4 More General Representations through KWIK

Thus far, while the analysis of MFRL's sample complexity has been done through the general KWIK framework, the focus has been mostly on tabular representations of T and R, where each state/action pair's transition and reward dynamics are completely independent of other state/action pairs. However, the KWIK framework allows model-based RL agents to employ a number of more general representations that scale polynomially efficient learning to larger environments.

The mechanism for performing such scaling is to use a representation of T and R with far fewer parameters than $|S|$. For instance, in a continuous domain, T or R might be represented as a linear combination of the n state factors with n unknown weights. In that case, only n parameters need to be learned by the base learners, and hence polynomial sample efficiency can ultimately be guaranteed. Many such representation classes have been analyzed within the KWIK framework [65], including linear systems [96], "typed" Gaussian-offset dynamics [19], Dynamic Bayesian Networks (DBNs) [65], and Gaussian processes [39]. Complex robotic motion may also benefit from stronger relational representations such as Object Oriented MDPs [30]. Since the theoretical analysis was done for any KWIK learnable representation, the algorithm and efficiency guarantees hold for these representations as well. When continuous representations are used, Q is approximated with a function approximator. The approximator parameters are passed from lower- to higher-fidelity levels to construct optimistic values. Note that not all systems are KWIK learnable [65] (e.g., the conjunction of n terms).

The following sections describe one of these general representations, a DBN, in

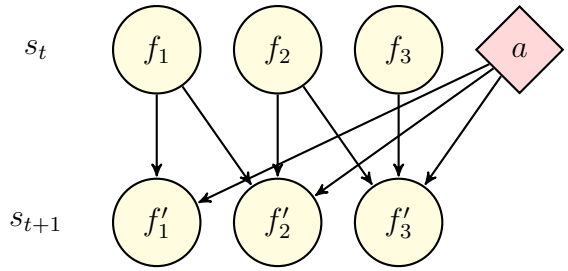


Figure 3-8: An example Dynamic Bayesian Network (DBN) representation of an MDP transition function where the state has 3 factors. DBN representations are most useful when the network is not fully connected, as shown here.

more depth and illustrate how it can be used in the MFRL framework in the puddle world simulators from Section 3.3.2. Later, coupling MFRL with a DBN representation allows the robotic car experiments to scale to much larger state spaces.

3.4.1 KWIK Learning a Dynamic Bayesian Network

A Dynamic Bayesian Network [16, 24] represents the evolution of a set of factors F from one step to another. A DBN can be viewed as a graphical model containing $2|F|$ nodes representing the value of each factor at time t and time $t+1$. Each factor f at level $t+1$ is connected to a set of *parent* nodes $\Pi(f)$ at level t.[2] The assumption leveraged in a DBN is that the probability distribution over f's value is independent of the probability of any other factor value given $\Pi(f)$. Thus, the probability of f taking on a given value can be encoded in a small probability table that grows exponentially only in the number of parents. In this work tabular representations of the conditional probability distributions are used, but any non-tabular, KWIK compatible representation could also be used.

DBNs can naturally be used to represent an MDP transition function for a factored state space by representing the factors of s and a in the top level and the factors of s' in the bottom level (with dependencies on the variables of s and a). Fig. 3-8 shows

[2]Cross-Edges within the $t+1$ level are also allowed as long as they do not create cyclic dependencies.

an example DBN. Based on the independence assumption of the DBN,

$$T(s, a, s') = \prod_{f \in F} Pr(f(s')|\Pi(f|s, a)).$$

A DBN over n factors, each with v values, contains only $n(v-1)v^{|\Pi|}$ parameters, where $|\Pi|$ is the maximum number of parents of any node. Thus, the DBN is KWIK learnable given the structure Π using the KWIK-Rmax framework.[3] Specifically, a parameter

$$m = \frac{2v}{\epsilon^2} \ln(\frac{2v}{\delta})$$

is set, and, for a given $\langle s, a \rangle$, if any factor induces a $\Pi(f|s, a)$ with fewer than m experiences, $L_T(s, a) = \perp$, otherwise the maximum likelihood distribution given the current data is used for T. Combining this partitioning of known and unknown factor combinations with the $R_{\max}$ heuristic yields agents that efficiently target factor combinations that are unknown but may lead to better behavior, without exploring the entire ground state space.

In MFRL, the generic Algorithm 4 with the base KWIK-DBN learners are used as described above, but instead of the $R_{\max}$ heuristic, $\hat{Q}_i(s, a)$ is set for any s that induces an unknown parent configuration (that is, $L_T(s, a) = \perp$) to $\hat{Q}_{i-1}(s, a)$.

An example of how the DBN representation can speed up MFRL's learning is now provided. Later sections show how such a representation allows MFRL to scale to state spaces on the robotic car that would be prohibitive to learn with a tabular representation, even in the presence of high-fidelity simulators.

3.4.2 MFRL DBNs for the Puddle World

States in the puddle world domain described in Section 3.3.2 consist of the x and y coordinates of the grid containing the current location of the agent. When the agent takes a step, the x coordinate of the new grid depends only on the previous x grid

[3] KWIK algorithms also exist for learning this structure. See [65].

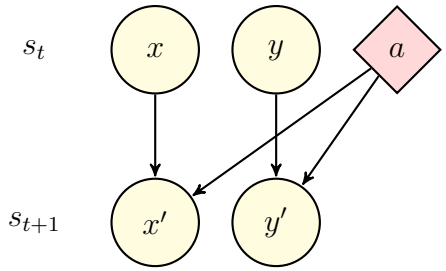
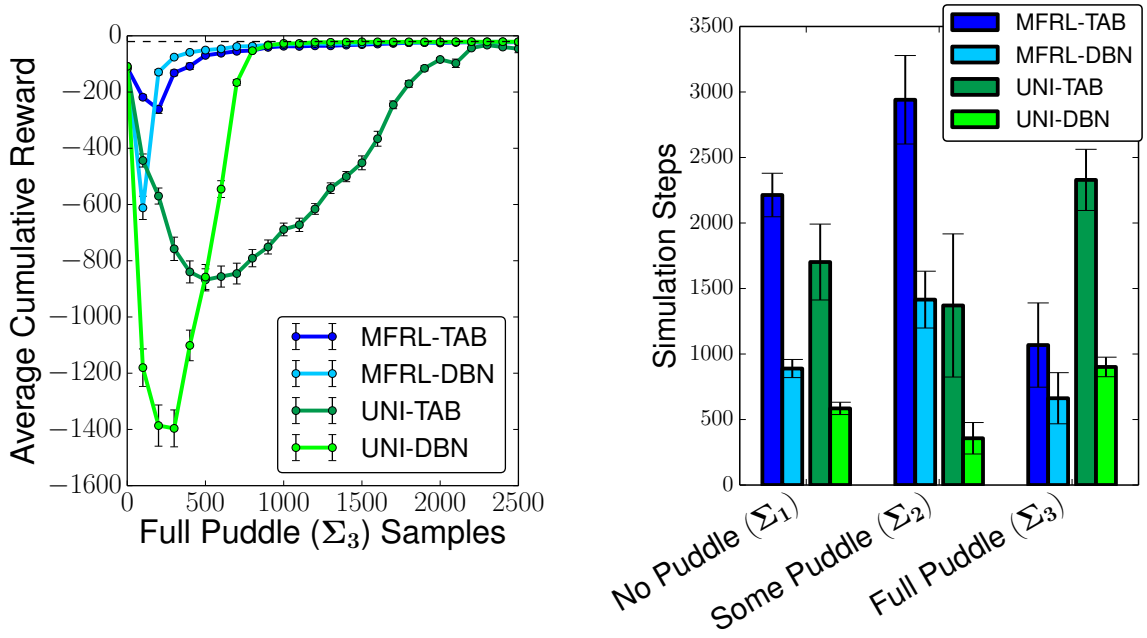


Figure 3-9: A Dynamic Bayesian Network representation of the transition dynamics for the puddle world domain introduced in Section 3.3.2. Each state consists of an x and a y grid location. These dimensions are independent given the current action, thus reducing the fully connected tabular representation to one with 2 factors.

coordinate, and not on the previous y grid coordinate. Similarly, transitions between y grid values depend only on the current y values. Therefore, the transition dynamics of the puddle world agent can be captured by the DBN model shown in Fig. 3-9, where the next x and y state values depend only on the previous action and previous x or y state value, respectively. It should be noted that this DBN representation is not the most efficient way to represent or solve the puddle world domain, but is merely an example of how DBN's can be used to increase the speed of learning.

Fig. 3-10(a) and 3-10(b) shows average learning curves at the top level and average samples at each of the three levels using Algorithm 4 with the DBN extensions described in Section 3.4.1. Improvement over the tabular transition function representation is evident as more than 50% fewer samples are needed in the three levels when using the DBN.

One unanticipated effect of using the more general representation is the increased negative transfer from Σ_1 and Σ_2 to Σ_3, as demonstrated by the more substantial initial decrease in average reward during learning shown in Fig. 3-10(a). The increased negative transfer is due to the increased generalization of the transition dynamics. When exploring Σ_3, the agent is less likely to initially reach states unobserved at Σ_2 because the DBN generalizes each experience to many more states than a tabular representation would. Thus, the DBN agent will explore more of the puddle in Σ_3 before moving back to Σ_2 and Σ_1. A detailed analysis of the effects of transition and reward dynamics generalization on the amount of negative transfer in MFRL is



(a) Learning curves (standard errors shown)

(b) Samples used (standard deviation shown)

Figure 3-10: Using the Dynamic Bayes Network (DBN) shown in Fig. 3-9 to represent the transition function for the puddle world decreases the top-level convergence time and overall samples used. (a) However, the more generalized representation function also transfers more negative information than the tabular representation (TAB) as seen by the initial dips in average reward. (b) The total samples used across the three simulator levels is decreased by an average of 52% and 66% for the MFRL and unidirectional transfer (UNI) cases, respectively.

outside the scope of this thesis and will be saved for future work.

3.5 Generative Simulators

Thus far, the analysis and experiments have been conducted under Assumption 2, which states samples can only be obtained by executing trajectories. While this is certainly true in the real world and may be true in many high-level simulators with little API access (such as a commercial video game or complex simulator), often this assumption may be too restrictive. Many simulators provide *generative* access to dynamics and reward information, where arbitrary state/action pairs can be queried. Many simulators provide *generative* access to dynamics and reward in-

formation. Specifically, many simulators can be queried with a $\langle s, a \rangle$ and will return a next s' and reward without the user having to perform all of the steps to reach the query state s.

However, generative access does not necessarily make choosing *where* to sample any easier. For instance, consider the case where an agent in Σ_d encounters a state that has not been visited in Σ_{d-1}. It is tempting to simply query Σ_{d-1} at that state. However, these samples might be ineffective. The state may have been reached because of a poor decision made much earlier in the trajectory. In fact, the information gathered in the Σ_d trajectory may have already updated the model to correct such a decision in the future. A concrete example of this phenomenon happens in the robotic car experiments in Section 3.6 where optimistic transitions in Σ_{D-1} cause the car to attempt aggressive maneuvers that lead to spinning out in Σ_D. But this does not mean the system should query the simulator to learn what happens *during* an unrecoverable spin. The more prudent course may be to replan in Σ_{D-1} given the transition dynamics actually encountered in Σ_D, which is what MFRL does.

Of course, there are situations where the opposite reaction would be better. For instance, if simulator Σ_{d-1} lacks a variable used in Σ_d, it may not support the trajectory in Σ_d that encountered an unknown state s. Thus, running greedy trajectories in Σ_{d-1} may not lead to s even though the simulator may have a perfect model of the dynamics in s itself.

Since both of these cases can occur, a simple extension is proposed to the MFRL algorithm when one or more simulators provide generative access. When moving to a lower-fidelity simulator based on a set $\bar{S}$ of states unknown at level $d-1$, if Σ_{d-1} is generative, the algorithm runs greedy trajectories starting directly at the states in $\bar{S}$ to gather samples for its model of Σ_{d-1}. Once all of the states in $\bar{S}$ are known, it performs the normal greedy trajectories in Σ_{d-1} as specified in the original MFRL algorithm.

Fig. 3-11 and 3-12 show the result of applying Algorithm 4 to the puddle world domain described in Section 3.3.2, where generative access is assumed at all levels of the domain. Generative access does little to improve the convergence of the algorithm

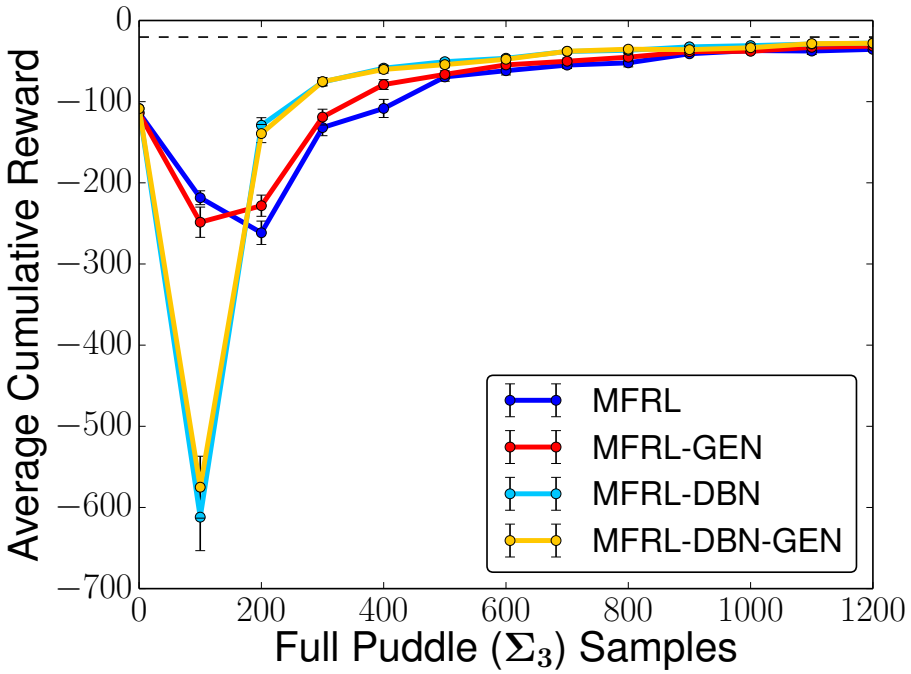


Figure 3-11: Using generative access in the puddle world domain doesn't significantly improve learning performance. Other simulation environments could benefit more from generative access to the simulation environments. Learning curves are the average of 1000 runs with standard errors shown.

in the real puddle world or with the robotic car in the next section. However, I believe other domains might benefit more from generative simulator access.

3.6 Robotic Car Results

Small robotic (or remote-controlled) cars have been popular testbeds for RL algorithms, with several results demonstrating improved trajectory tracking performance using various RL techniques [3, 56, 59], though none of these approaches chose which, if any, simulator to run trajectories in. Several bandit and multi-state experiments are run on a robotic car with two simulators of the car's dynamics. This section describes the car, simulators, experimental setup and results. A video showing an example of MFRL applied on the car is available at `https://youtu.be/c_d0Is3bxXA`.

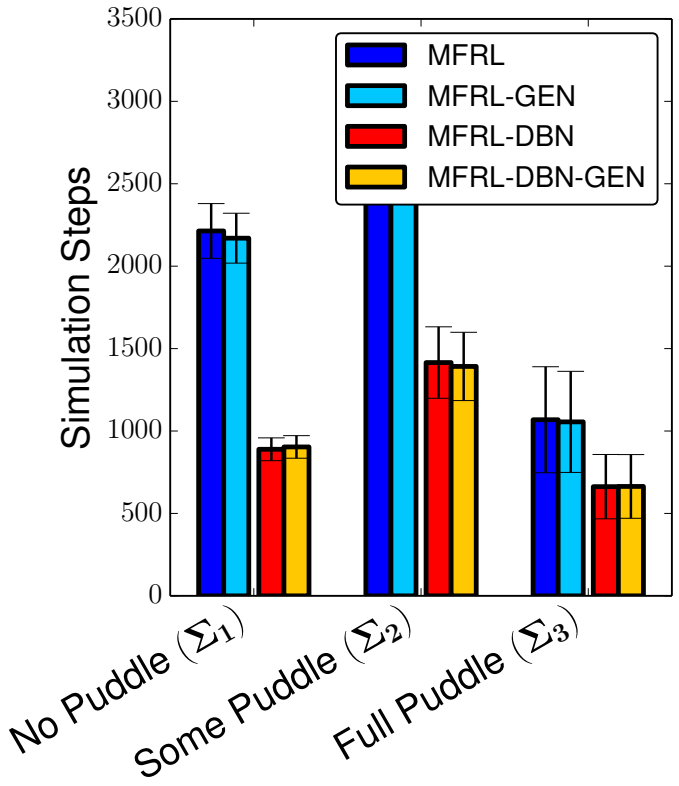


Figure 3-12: Using generative access in the puddle world does not significantly alter the average samples used in any of levels of learning. Bars show an average of 1000 runs with standard deviations shown.

3.6.1 Experimental Setup

The MFRL algorithms are experimentally verified using a robotic car driving on an indoor track. The car is an off-the-shelf 1/16 scale 4-wheel drive rally car shown in Fig. 3-14. The position, velocity, and heading angle of the vehicle are measured using an external motion capture system. The wheel velocity is measured and filtered using an optical encoder read by a microcontroller.

Fig. 3-13 shows the car task, which consists of selecting different radii and velocities to minimize lap-times on a track of fixed length. In this scenario, the track consists of straight and curved segments, each with an associated distance and velocity parameter, both of which remain constant during the length of each segment. As shown in the diagram, the virtual "cones" are fixed and denote the length of the

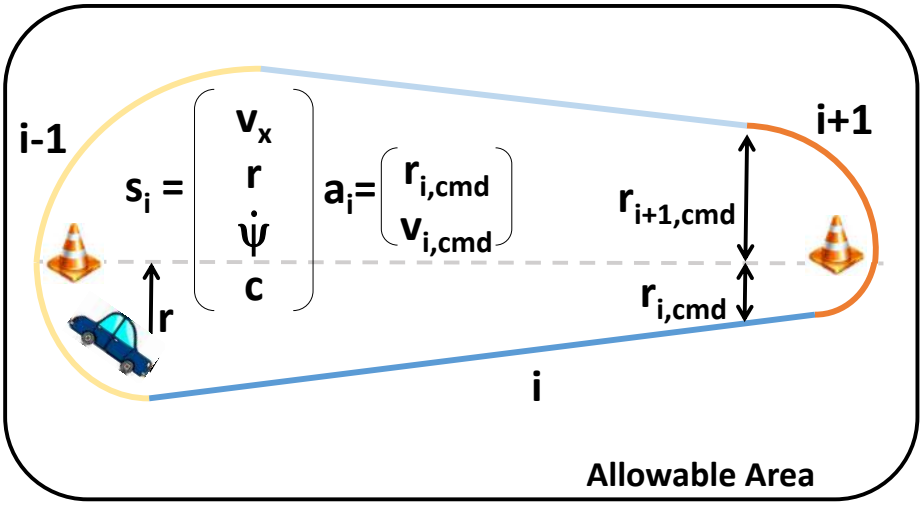


Figure 3-13: The robotic car task consists of driving around a fixed-length track (length determined by cone placement). Each lap is divided into four segments and the car must choose both the commanded velocity and commanded turn radius of each segment upon completion of the current segment. These decisions are made based on the state values at the end of each segment, namely the car forward velocity, current radius, rotational velocity, and segment type.

path. As the car finishes each segment, the commanded radius $r_{i,cmd}$ and velocity $v_{i,cmd}$ values for the next segment are chosen. The reward returned for each segment is $-t$ where t is the elapsed time required to drive that segment. If the car drives, or often slips, out of a virtual "drivable" area around the track (denoted by the large black box in Fig. 3-13), the car resets to a fixed initial condition and is given a large negative reward. The state variables in s_i (Fig. 3-13) are the body frame forward velocity V_x, rotational rate $\dot{\psi}$, distance from track center r, and the current segment type c (straight or curved).

Choosing the next radius and velocity fully defines the desired path for the next segment (note that straight and curved track segments are forced to alternate). The car follows this path using a pure pursuit controller where the look ahead control distance is a function of the commanded velocity [79]. Running at 50 Hz, the pure pursuit controller computes the desired forward velocity, rotational rate and heading angle required to keep the car on the specified trajectory. Steering angle commands and desired wheel velocities are computed using the closed-loop controllers from [43], where the cross track error term in the steering angle control law is omitted, as the

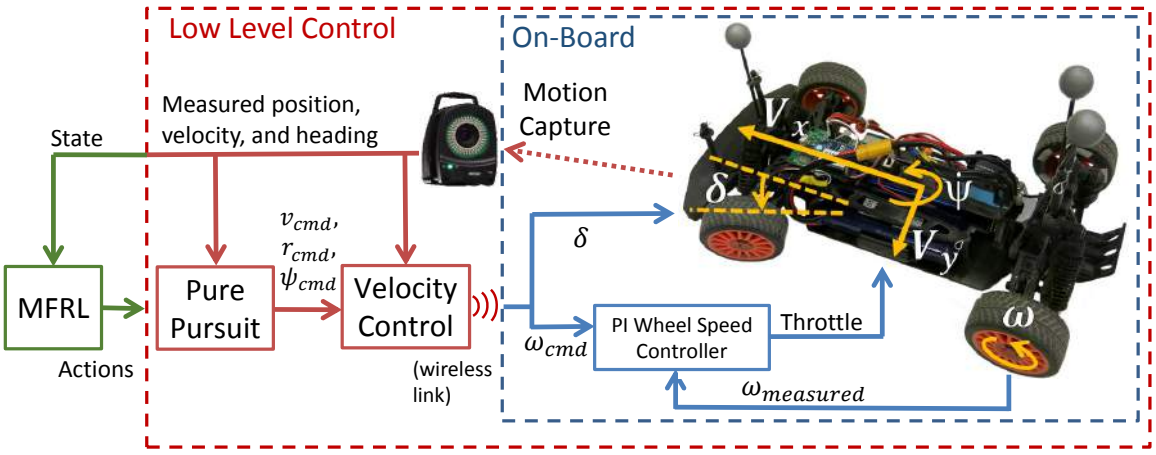


Figure 3-14: One of the benefits of the MFRL framework is the ability to wrap the learning around fixed lower-level controllers. The car is controlled using a series of closed-loop controllers that perform high-rate velocity and position tracking, while MFRL learns which high-level commands to issue at a much slower rate. The robotic car and associated state and control variables are also shown. The state variables consist of the body-frame linear velocities V_x and V_y, the wheel velocity ω, and the heading rate $\dot{\psi}$. The control inputs are the steering angle δ and the desired wheel velocity ω_{cmd}.

cross track error is reduced by the pure pursuit algorithm. The C_y parameter in this control law (see Equation 8 in [43]) is found by matching measured vehicle data to input commands.

The steering angle δ and commanded wheel speed ω_{cmd} are sent to the car's microcontroller over a wireless serial connection. Steering commands are sent directly to the servo. Commands to the motor come from a simple closed-loop controller around the commanded and measured wheel speed. This proportional-integral wheel speed controller is used to lessen the effects of the time-varying battery voltage on the velocity dynamics.

An overview of the control loops used to control the vehicle is shown in Fig. 3-14. One of the strengths of the MFRL framework is that it can be used around existing closed-loop controllers. These closed-loop controllers need not be the same at the different levels of simulation.

The simulation environments for the robotic car consist of a naïve simulator (Σ_1) and a dynamics-based simulator (Σ_2). The naïve simulator ignores the dynamic model

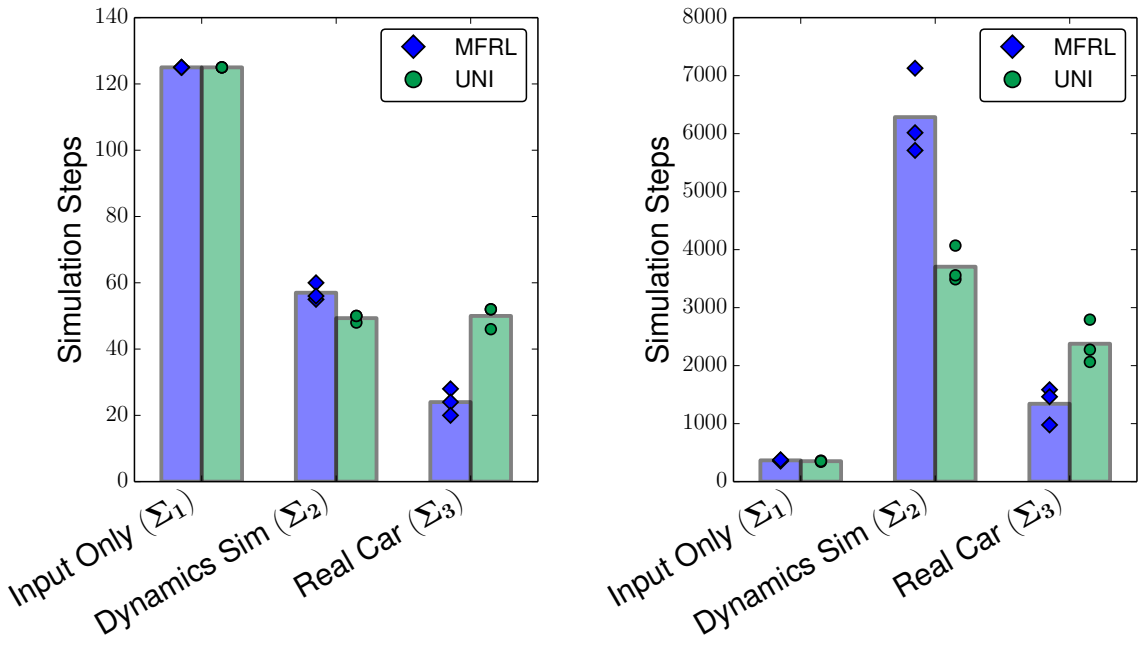
of the car and returns ideal segment times assuming the car followed the requested trajectory exactly. As mentioned in Section 3.1.2, this simulator does not model rotational rate $\dot{\psi}$ and so the state-space mapping is not 1-to-1 when going from Σ_1 to Σ_2. The higher-fidelity simulator models the basic dynamics of the car, including wheel slip, where model parameters such as the "Magic Tyre Parameters" [110] are estimated using test data collected on the car. This simulator captures much of the dynamic behavior of the car, although discrepancies in real-world data and simulator data become more significant at higher velocities (above about 2.0 m/s) and when the wheels slip significantly. Therefore, learning needs to be performed not only in the simulators, but also on the physical car. Descriptions of this simulator and its underlying equations are included in Appendix B.

Both car simulators (Σ_1 and Σ_2) run many times faster than real time. Therefore, the total learning time in the experiments below is dominated by the time spent collecting samples from the real car. As mentioned earlier, the known-ness (m and m_{known}) parameters for the base learners in these experiments are set lower than the loose worst-case bounds from the theoretical section, which is standard practice with the Rmax family of algorithms.

3.6.2 Experiments for the Bandit Setting

The first robotic car experiment takes place in the bandit setting: choosing a single radius and two velocities (one for curves and one for straightaways) at the beginning of a 3-lap run. Five values were allowed for radii (between 0.5 and 1.2 m) and five values for velocities (between 2.0 and 3.5 m/s), yielding 125 actions/arms in the bandit scenario. Algorithm 3 is evaluated in this setting as well as unidirectional transfer of Q-value heuristics, that is, the transfer mechanism of [67] applied with the KWIK-Rmax framework.

The simulators are deterministic (Σ_2 has only a small amount of added artificial noise) and so the known-ness parameter is set as $m = 1$ at Σ_1 and Σ_2. To account for real-world noise, $m = 2$ is set at Σ_3, meaning 2 tries with a given parameter setting were needed to determine its value. Both MFRL and unidirectional transfer



(a) Bandit

(b) State-based

Figure 3-15: Samples used by MFRL and unidirectional transfer (UNI) at each level in the (a) bandit case and the (b) state-based case. In both experiments, the MFRL algorithm converges to an equally fast policy when compared to unidirectional transfer; however, MFRL uses over 50% fewer real-world samples in the bandit case and over 30% fewer in the state-based case. Each algorithm is tested 3 times with bars showing the average.

found a near-optimal policy within 60 steps on the real car. These results were not compared to a no-transfer algorithm since it would need at least 250 trials to identify the optimal policy[4].

Fig. 3-15(a) depicts the samples used in each simulator by the algorithms. MF-Bandit on average uses fewer than half as many samples in the real-world when compared to the unidirectional learner. Both MF-Bandit and unidirectional transfer converged to policies with lap times of about 3.7 seconds per lap and learned to use higher velocities on the straightaways than the curves. These lap times are similar to the ones found in the state-based setting described in the next section, although the state-based policy is more robust to disturbances and noise.

[4]While there are only 125 actions, each action must be tried $m = 2$ times in the real world before it is known.

Table 3.4: State and action discretizations for the state-based car MFRL results.

Value	**Min**	**Max**	**Disc. Small**	**Disc. Large**
c (segment)	–	–	2	2
$\dot{\psi}$ (rad/s)	-1.0	3.5	3	3
r (m)	0.5	1.2	3	4
v_x (m/s)	2.0	3.2	3	6
r_{cmd} (m)	0.5	1.2	3	4
$v_{x,cmd}$ (m/s)	2.0	3.2	3	6

Table 3.5: Parameters for the state-based car MFRL results.

Parameter	$\mathbf{\Sigma_1}$	$\mathbf{\Sigma_2}$	$\mathbf{\Sigma_3}$
m_{L_R}	1	1	1
m_{L_T}	1	3	3
m_{known}	100	100	50
$m_{unknown}$	–	10	10
β	0	0	–

3.6.3 Experiments for the State-Based Setting

In the multi-state case, the state space described in Section 3.6.1 and Fig. 3-13 was used, allowing the car to pick a radius and velocity at the beginning of every segment (4 per lap). Because closed-loop, state-based decisions (i.e. changing velocities and radii in short segments), are being made, the action-space can be reduced from the bandit setting, since $|\Pi| = O(|A|^{|S|})$. Here, 3 radii and 3 velocities are used ($|A| = 9$). The discretization of the state and action values is shown in Table 3.4 (small problem). Since the discretization in the state space makes the simulation results potentially noisier (from state aliasing), m_{L_T} is set to 3 in Σ_2 and the real car. The MFRL parameters used for these results are shown in Table 3.5.

In the experiment, both MFRL and unidirectional transfer converged to a policy with lap times of just under 3.7 seconds, slightly faster than the bandit results from the previous section. Fig. 3-15(b) shows the samples used in each level by MFRL and unidirectional transfer. The MFRL algorithm converges using an average of 35%

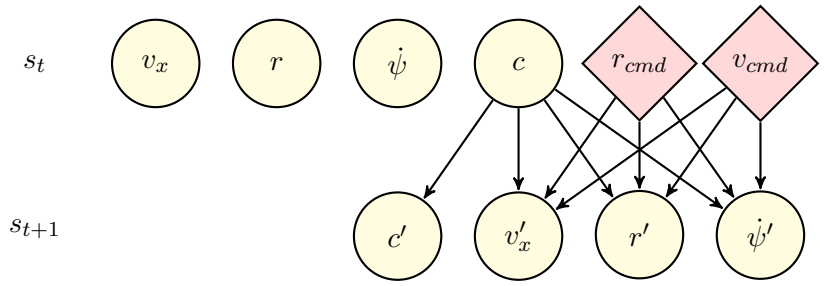


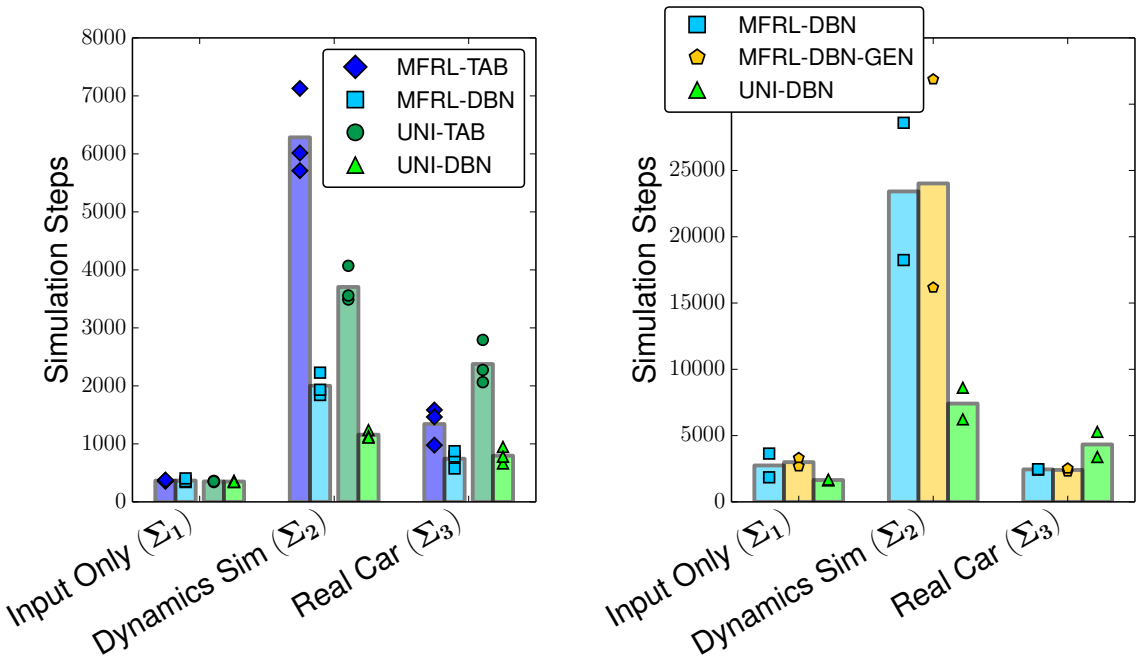
Figure 3-16: A Dynamic Bayesian Network representation of the transition dynamics for the robotic car. The next radius, forward velocity, and rotational rate depend primarily on the commanded radius, commanded velocity and the current segment. The next segment depends only on the current segment as the car is forced to go from straight to curved to straight and so forth. Note that this transition model decreases the maximum factor order from 6 to 3.

fewer samples in the real world when compared to unidirectional transfer.

The converged policy in the state-based experiments is different from the bandit case, due to the versatility of state-based control. Instead of an oval shape, MFRL chose different values for the radius entering a curve versus the radius exiting the curve. This led to initially wide turns towards the cones followed by a sharp turn towards the straightaway, maximizing the time the car could drive fast down the straight section. Reaching this fairly complicated policy with a reasonable number of real-world samples was made possible by MFRL's efficient use of samples from the previous levels. Particularly, experience in Σ_1 pruned policies that were too slow, while experience in Σ_2 pruned policies that were too fast on the curves. Finally, experience with the real car refined the policy under the actual, but noisier, conditions in the real environment.

3.6.4 Experiments using DBN Representation and Generative Access

This section uses the notion of a generalized transition function via a DBN introduced in Section 3.4.1 to decrease the required real-world samples needed to solve the track problem. The independence assumptions made are shown in Fig. 3-16. Because the closed-loop controllers guiding the car are fast relative to the length of the track



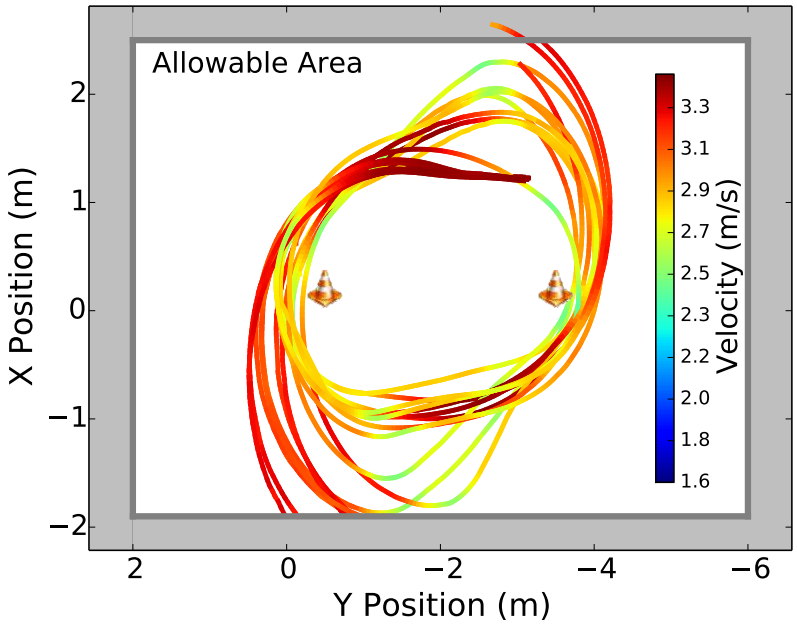
(a) DBN (3 runs)

(b) DBN Large (2 runs)

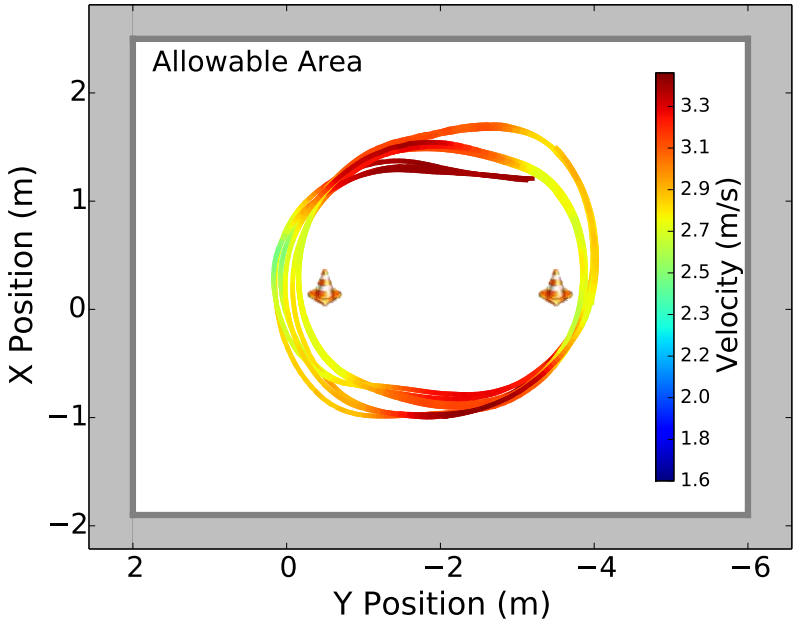
Figure 3-17: (a) Using the Dynamic Bayes Network (DBN) shown in Fig. 3-16 to represent the transition function of the car decreases the total samples used across the three levels of learning by an average of 63% for both the MFRL and unidirectional transfer cases. (b) With the larger domain specified in Table 3.4, the MFRL algorithm decreases the samples used on the real car by an average of 40% when compared to a unidirectional transfer case. Note that the size of the state/action space here is over 3400, but the MFRL algorithm converges in fewer than 2500 samples on the real car.

segments, the state values at the end of a segment are assumed to depend only on the segment type and the actions chosen for that segment, and not on the state values when the segment was started. There are cases when this assumption is violated, such as when the car starts sliding out of control in one segment before transferring to the next segment; however, for most of the state/action space and the testing performed, this assumption holds true.

Fig. 3-17(a) shows the samples used to solve the track problem using the small discretization values shown in Table 3.4. The DBN transition model significantly decreases the samples needed to converge at all levels of the learning, with both the MFRL and the unidirectional algorithms using more than 60% fewer samples than when using the nominal tabular transition function (results from Fig. 3-15(b)).



(a) Initial real-world policy



(b) Converged real-world policy

Figure 3-18: Actual trajectories driving by the car during one of the learning runs. (a) As the MFRL algorithm transitions to the real car for the first time, the attempted trajectories are too aggressive for the actual hardware, leading to the car frequently exiting the allowable region. (b) By the end of the learning process, the car consistently drives fast along the straight segments and slower around the turns, yielding fast lap times.

Comparing the MFRL algorithm using a DBN transition model (MFRL-DBN) to unidirectional transfer with a DBN (UNI-DBN), MFRL-DBN uses slightly (though not significantly) fewer samples in Σ_3 while using more samples in Σ_2. This trade-off is by design, as MFRL assumes (under Assumption 1) that using more samples in a lower-fidelity simulator is worth using fewer samples in the real world, since the real world samples are far more costly. Indeed, this is true in the robotic car experiments where actually running the car in the lab takes significantly more time and effort than the computer simulation. The combination of a DBN and the MFRL transfer framework significantly decreases the number of real world samples compared to the other representations/framework combinations studied here.

This decreased number of required samples now allows MFRL to solve the track problem using a larger state/action space. Table 3.4 shows state and action discretization values for the larger problem solved using MFRL with a DBN transition function representation. Note that, in this case, the state/action size is nearly an order of magnitude larger than the state/action size for the smaller problem in Table 3.4, the parameter settings used in the previous section.

Fig. 3-17(b) shows that with this larger state/action space, the MFRL algorithm uses on average more than 40% fewer samples in the real world than the unidirectional transfer algorithm. As in the puddle world examples, giving the algorithm generative access to the simulators does little to improve the convergence rate of the algorithm. In all of the tested cases, the algorithms converged to policies very similar to those found in the smaller state/action space problem described in Section 3.6.3. In these experiments, the car converged to this policy with fewer real-world samples than the size of the state/action space.

An example of the types of policies that the MFRL algorithm converges to in these experiments is shown in Fig. 3-18. Plots of the actual trajectory driven by the car for the first several and last several laps during the learning process are shown in Figs. 3-18(a) and 3-18(b), respectively. When the MFRL algorithm first transitions to the real world, the optimism assumption in the simulators is evident by the fact that the car often attempts to drive quickly around the turns, a strategy that works

well in simulation but causes the car to drive out of the feasible region in practice. Eventually, the algorithm discovers that driving quickly on the straight-aways and slower around the cones yields a safer and faster policy.

3.7 Summary

This chapter introduced MFRL, which extends lessons from the multi-fidelity optimization community to sequential decision making problems. MFRL transfers heuristics from lower- to higher-fidelity simulators to guide exploration. Unlike previous transfer learning techniques, this framework also allows agents in lower-fidelity simulators to plan using higher-fidelity learned model parameters, a tactic shown to be crucial for minimizing sub-optimal steps in the real world. Many robotics domains already use simulators of varying fidelity during the initial stages of hardware and software development. MFRL can leverage these existing simulators to decrease the dependence of RL algorithms on the physical hardware.

Throughout MFRL, the learning agents retain theoretical sample efficiency guarantees over the entire learning process because of the integration of the KWIK-Rmax framework, and the empirical results show these algorithms are also efficient in practice. Experiments with a robotic car show that, not only is the framework theoretically sound, but it is also a practical technique for scaling RL algorithms to real-world decision making problems.

Chapter 4

Using Informative Simulated Priors

This chapter introduces a method for using samples from a simulated robot to decrease the number of real-world samples that are needed to learn a good policy (see Figure 4-1). Specifically, a learning algorithm is applied in a simulator to learn a model of the simulated dynamics and a good policy for the simulated domain. The learned transition dynamics and policy are then used as a prior for real-world learning using the Probabilistic Inference for Learning Control (PILCO) algorithm [25]. The simulated prior is used in a GP model of the transition dynamics in PILCO to infer about states that the real-world system has not yet sampled. Empirical results show that, even when the simulator is inaccurate, using an informative simulated prior decreases the learning samples needed in the real world and increases the average performance of the achieved solution. This approach differs from previous work using priors in PILCO [15] in that it is not limited to linear priors. Using a learned, nonlinear prior from a simulator allows for incorporating prior information from arbitrarily complex simulations without needing to make assumptions about the underlying dynamics of the system.

There is significant existing work verifying the idea that using prior knowledge can increase the performance of learning algorithms. For instance, using a physics-based prior when learning inverse dynamics using a Gaussian process has been shown to yield superior performance when compared to using no prior knowledge [51, 77]. Also, several existing RL algorithms use simulators to augment real robot data [3, 22,

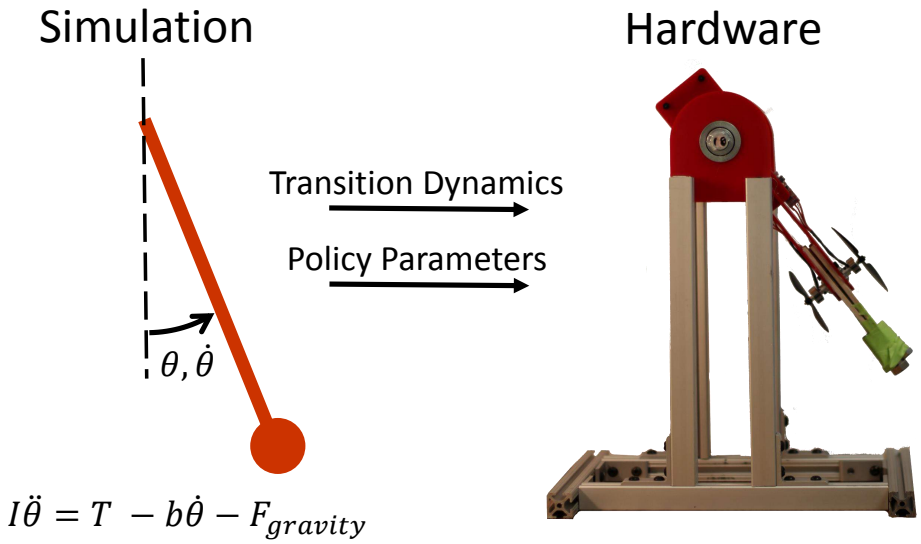


Figure 4-1: The proposed algorithm provides a method for transferring policies and learned transition dynamics from a simulated environment (left) to a real-world learning agent (right). The simulator captures the basic dynamics of the physical system but is not sufficiently accurate for performing learning in the simulator alone. Using policy parameters and transition dynamics learned in the simulator leads to faster learning on the physical robot. The algorithm is shown to decrease the number of samples needed by the real-world learning agent to achieve useful behavior.

56]. Likewise, the transfer learning community [106] has sought to more seamlessly transfer information from simulators to the real world [67]. However, the above work assumes either an explicit form for the simulator equations or a discrete state and action space. In contrast, this work leverages nonparametric Gaussian processes (GPs) to incorporate data from simulators in a principled way. The simulators can model continuous states and actions and be black-box codes such as proprietary robot simulators or based on finite element methods.

The main contributions of this chapter are (1) a principled approach to incorporating data from any simulator into the PILCO learning algorithm, (2) a derivation of propagating uncertain inputs through a Gaussian process with a nonparametric mean function, and (3) simulated and hardware results empirically showing the benefits of using prior information in the learning process. Using prior data from a simple simulator, experimental results show convergence to a good policy on a physical inverted pendulum with at least three times less data than is needed when a zero-mean prior

Algorithm 5 PILCO [25] with Prior Knowledge

input: Controller parameters, either random ($\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$) or from the simulator ($\boldsymbol{\theta}_p$). Apply random control signals and record data.
input: Observed simulator data $\{X_p, \boldsymbol{y}_p\}$
Learn dynamics model using simulator data
while task not learned **do**
    Learn probabilistic (GP) dynamics model using real-world data with the simulator data as a prior
    while not converged **do**
        Approximate inference for policy evaluation
        Gradient-based policy improvement
        Update parameters $\boldsymbol{\theta}$ (e.g., CG or L-BFGS)
    return $\boldsymbol{\theta}^*$
    Set $\pi^* \leftarrow \pi(\boldsymbol{\theta})^*$
    Apply π^* to system and record data

is used.

The remainder of this chapter is organized as follows. Section 4.1 details the derivation of using a Gaussian process with a nonlinear mean when the inputs are uncertain. Simulated and hardware results of the proposed algorithm are then shown in Section 4.2. Finally, the chapter concludes with a brief discussion in Section 4.3 and summary in Section 4.4.

4.1 PILCO using a Nonlinear Prior Mean

The generic PILCO algorithm (Algorithm 5, black text) assumes a zero-mean prior on the transition dynamics. This uninformative prior does not bias the model, giving the algorithm freedom to model arbitrary transition dynamics. However, the uninformative prior also means that the policy search algorithm cannot make informed decisions about areas of the state space from which no samples have yet been collected. In contrast, this chapter proposes using PILCO with an informative prior consisting of data from a simulator of the real domain. The informative prior gives the policy search phase information about what effect a policy will have on the system even in areas of the state space that have not yet been explored. The proposed algorithm is shown in Algorithm 5, with the additions to the original algorithm highlighted in blue on

lines 1-3 and 5.

The modified algorithm takes as inputs policy parameters (either randomly initialized or from a learning algorithm applied to the simulator) and observed simulated data. The simulated data are used to build a probabilistic model of the simulator's transition dynamics. The mean of this model is then used as a prior for the transition dynamics model learned in the target domain.

Given data from a simulation of the target domain, one way to incorporate the data into the learning algorithm is to train a single GP using both simulated and real data as inputs. Mixing simulated and real data has been shown to cause poor performance as the GP models of the real-world transition dynamics can become corrupted by incorrect simulation data [68]. In this approach, even with an incorrect simulator, real data from the target domain will eventually overcome the effects of the prior and converge to the true transition dynamics as the number of obtained data points increases.

To effectively use the learned GP dynamics model in PILCO (line 7 in Algorithm 5), the algorithm performs simulated roll-outs of the system dynamics using the learned model. This calculation requires machinery for correctly propagating the mean and covariance of uncertain inputs through the GP model of the transition dynamics. This section gives the required calculations to propagate uncertain inputs through a GP when the prior function is the mean of a different GP. This mean prior function is equivalent to a radial basis function (RBF) network.

Unlike deterministic inputs, mapping an uncertain Gaussian test input $\boldsymbol{x}_* \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ through a GP does not, in general, result in a Gaussian posterior distribution. However, the posterior can be approximated as a Gaussian distribution by computing the mean and covariance of the posterior distribution [38]. PILCO iteratively uses these Gaussian approximations when performing long-term predictions using the learned GP transition dynamics.

Next, the posterior mean and covariance equations are shown when the prior mean of the GP is an RBF network. When the learning domain has multiple target variables (such as angle and angle rate for the inverted pendulum), independent GPs

are learned for each output dimension. Where necessary, different output dimensions are differentiated with the subscripts a and b. In each equation, blue text denotes the terms that come from the prior. A full derivation of these equations is given in Appendix C.

The predictive mean μ_* is given by

$$\mu_* = \boldsymbol{\beta}^T \boldsymbol{q} + \boldsymbol{\beta}_p^T \boldsymbol{q}_p, \tag{4.1}$$

where $q_i = \alpha^2 |\Sigma\Lambda^{-1} + I|^{-1/2} \exp(-\frac{1}{2}\boldsymbol{\nu}_i^T(\Sigma+\Lambda)^{-1})\boldsymbol{\nu}_i)$ with $\boldsymbol{\nu}_i = \boldsymbol{x}_i - \boldsymbol{\mu}$. The subscript p denotes terms coming from the prior.

The predictive covariance Σ_* of the uncertain test inputs through the GP $f(\boldsymbol{x})$ is given element-wise as

$$\begin{aligned}\sigma_{ab}^2 = \delta_{ab}(\alpha_a^2 - \operatorname{tr}((K_a + \sigma_{\epsilon_a}^2 I)^{-1} Q)) + \boldsymbol{\beta}_a^T Q \boldsymbol{\beta}_b + \boldsymbol{\beta}_{p_a}^T Q_p \boldsymbol{\beta}_{p_b} + \boldsymbol{\beta}_{p_a}^T \hat{Q} \boldsymbol{\beta}_b + \boldsymbol{\beta}_a^T \hat{Q}_p \boldsymbol{\beta}_{p_b} - \\ \left(\boldsymbol{\beta}_a^T \boldsymbol{q}_a + \boldsymbol{\beta}_{p_a}^T \boldsymbol{q}_{p_a}\right)\left(\boldsymbol{\beta}_b^T \boldsymbol{q}_b + \boldsymbol{\beta}_{p_b}^T \boldsymbol{q}_{p_b}\right),\end{aligned} \tag{4.2}$$

where δ_{ab} is 1 when $a = b$ and 0 otherwise.

Finally, PILCO uses the covariance between the uncertain test input $\boldsymbol{x}_* \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ and the predicted output $f(\boldsymbol{x}_*) \sim \mathcal{N}(\mu_*, \Sigma_*)$ to compute the joint distribution $p(\boldsymbol{x}_*, f(\boldsymbol{x}_*))$. This covariance is calculated as

$$\Sigma_{\boldsymbol{x}_*, f_*} = \sum_{i=1}^{n} \beta_i q_i \Sigma(\Sigma + \Lambda)^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu}) + \sum_{i=1}^{n_p} \beta_{p_i} q_{p_i} \Sigma(\Sigma + \Lambda_p)^{-1}(\boldsymbol{x}_{p_i} - \boldsymbol{\mu}). \tag{4.3}$$

In summary, Eq. (4.1)–(4.3) are the true predictive mean, covariance, and input-output covariance of an uncertain input passed through a GP with a mean function modeled as an RBF network.

In addition to propagating uncertain inputs through the transition dynamics model, PILCO requires the calculation of closed-form derivatives of the predicted mean, covariance, and input-output covariance with respect to the input mean and covariance. These calculations are rather long, but not particularly difficult and so are not included here. A full derivation of the required derivatives is given in Appendix C.

Table 4.1: Default parameters used in the inverted pendulum and cart pole domains.

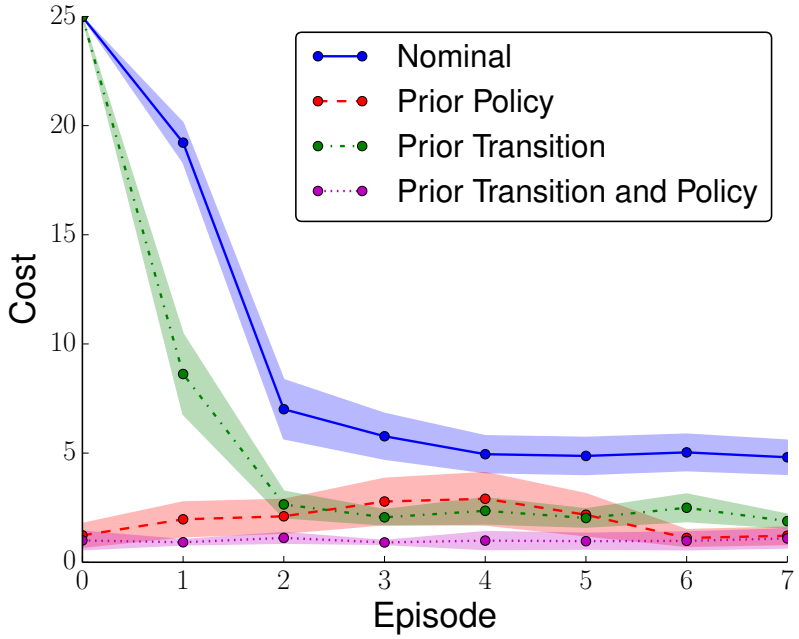
	Pendulum Mass	Pendulum Length	Friction Coefficient	Cart Mass	Max Force or Torque
Inverted Pendulum	1 Kg	1 m	0.1 N/m/s	-	2.5 Nm
Cart Pole	0.5 Kg	0.5 m	0.1 N/m/s	0.5 Kg	10 N

The computation of the predictive mean, covariance, and input-output covariance using a simulated prior requires approximately twice as much computation time as the zero-mean prior case (assuming the number of data points in the prior GP and the current GP are roughly equal), as most of the computations are merely repeated on the prior data. However, note that the size of the prior data is fixed and does not grow with iteration number, thus the additional computational complexity of the algorithm due to the nonlinear prior does not grow as new data points are observed.

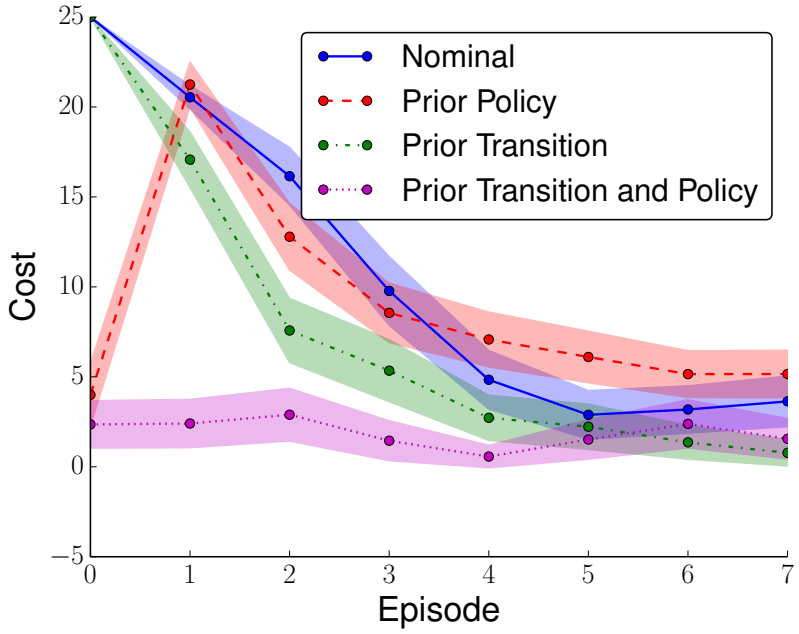
4.2 Results

Using the equations derived in the previous section, both simulated and hardware experiments are performed to identify how well the proposed alterations to PILCO work in practice. In all of these experiments the generic PILCO algorithm is run in the simulated domain and use the data collected during that learning exercise as the prior for the learning in the target domain. Note that any learning algorithm (including just randomly sampling the state-action space) could be applied in the simulator to obtain the required observations for the prior, but collecting data in the simulator using a good policy will more likely yield better results as data points will exist in the important areas of the state space.

As in [25], dynamics models are learned using tuples $(\boldsymbol{x}_t, \boldsymbol{\mu}_t)$ as inputs and differences $\Delta_t = \boldsymbol{x_{t+1}} - \boldsymbol{x_t}$ as training targets. In all the experiments the generalized binary saturating cost function from [25] is also used.



(a) Inverted Pendulum



(b) Cart-Pole

Figure 4-2: Learning curves for the 2-dimensional inverted pendulum (a) and the 4-dimensional cart-pole (b) domains when the prior comes from the same domain. Using an informative prior on the transition dynamics consistently improves performance regardless of the initial policy parameters. In the more complicated cart-pole domain using good initial policy settings does little to improve the algorithm performance unless prior transition information is used as well. Each line shows the mean of 20 independent learning samples each evaluated 5 times. The shaded regions show 95% confidence bounds on the standard error of the mean.

4.2.1 Using a Simulation Prior in a Simulated Domain

The first set of results explore the performance of Algorithm 5 using two different simulations of the same domain. This will demonstrate the benefits of the algorithm as a function of the difference between the domains. The two chosen domains are the benchmark RL domains of the inverted pendulum and the inverted pendulum on a cart, or the cart-pole domain. Unless otherwise stated, the default parameters used for simulation of these domains are giving in Table 4.1.

Using two identical instances of the target domains, the results first verify that, given a perfect simulation of a physical robot (assuming such a simulation exists), no learning needs to be performed in the real-world as a learning algorithm can be applied in the simulator and the resulting control policy used on the real robot. Figures 4-2(a) and 4-2(b) show the performance of Algorithm 5 under these idealized circumstances in the pendulum and cart-pole domains, respectively. In each figure the learning curves depict average cost as a function of learning episode. The following scenarios are compared: (a) the original PILCO algorithm with a random policy initialization, (b) the original PILCO algorithm using learned policy parameters from the simulation, (c) the proposed algorithm using a random policy initialization, and (d) the proposed algorithm using policy parameters from the simulation.

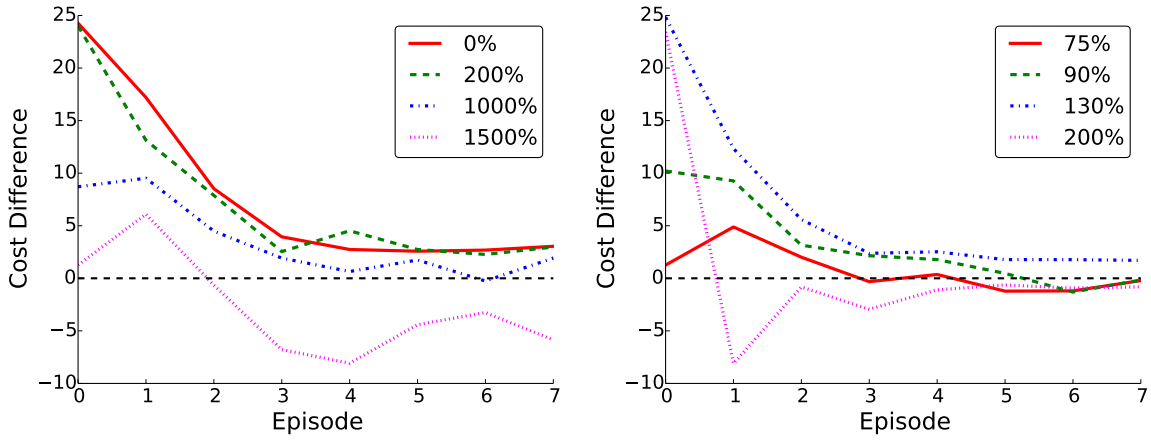
Note that, as expected, the learning curves either start with the performance of a random policy or with the performance of the converged policy, depending on the initial policy parameters used. When the prior policy is used and the prior for the transition dynamics comes from the same domain, the algorithm remains at the same performance level during the duration of the learning episodes as there is no incentive for the algorithm to explore different policy parameters. However, when a zero-mean prior is used (the original PILCO algorithm), even when initialized with near-optimal policy parameters, the performance actually gets worse for several learning episodes while the policy search explores different policy parameters before returning to the initial values. This transient learning phase is mild in the simple inverted pendulum domain. However, in the more complicated cart-pole domain,

using a good prior policy does little to increase the learning rate when compared a random policy initialization.

In both domains, even when the algorithms are initialized with random policy parameters, using an informative prior speeds the learning process and consistently leads to a better policy. The on-average worse performance of the original PILCO algorithm comes from the algorithm occasionally converging to a sub-optimal solution. For instance, in the inverted pendulum domain, the pendulum has sufficient torque to swing up to vertical with just one swing-back. However, the learning agent occasionally, depending on the random initial policy parameters, converges to a policy that consists of a double swing-back, taking longer than is necessary to get the pendulum to the inverted configuration. In these experiments, the prior data were generated using an instance of the PILCO algorithm that converged to a single swing-back policy. Thus, even when the policy parameters are randomly initialized, having an informative prior coming from a good policy consistently helps the algorithm to converge to a single swing-back policy.

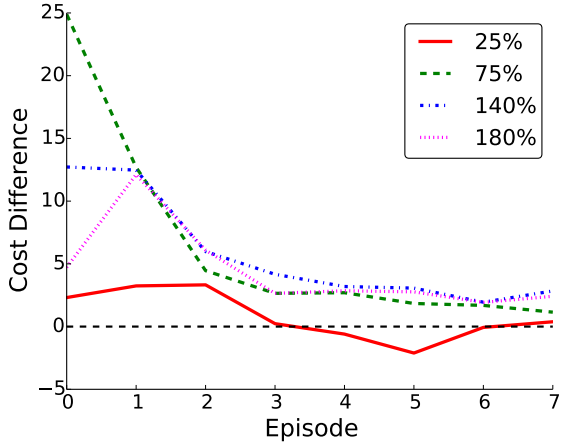
The next set of results demonstrates the performance of Algorithm 5 when the data for the prior come from a domain with different transition dynamics than the target domain. In Figures 4-3 and 4-4 the performance of PILCO using an informative prior (transition dynamics and policy parameters) is compared to using a uniform prior and random initial policy parameters. The plots show the difference between the uninformative and the informative prior as the domain parameters are varied between the prior domain and the target domain. For instance, in Figure 4-3(a) a prior learned in the default domain was used in a target domain where the friction coefficient was up to 15 times more than the default value. In each plot, the parameters not varied were kept at their default values. Positive cost differences show the modified algorithm performing better than the original. Except for extreme changes in the parameters of the target domain compared to the prior domain, using an informative prior from an imperfect simulator is still better than using no prior at all.

Due to the nature of continuous state-action RL being an optimization problem in an arbitrary non-convex space, it is difficult, if not impossible, to quantify or predict



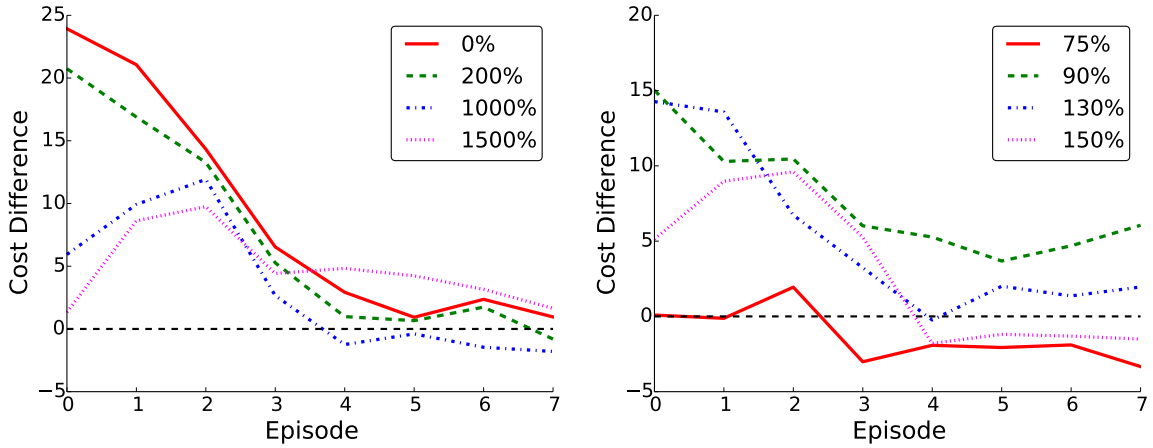
(a) Varying friction coefficient

(b) Varying thrust



(c) Varying pendulum inertia

Figure 4-3: Differences between the original PILCO algorithm and the algorithm when using an informative prior in the pendulum domain. In each case the informative prior was learned using the nominal pendulum parameters in Table 4.1 and then tested in a domain where the default parameters were varied. Positive cost differences show the modified algorithm performing better than the original during that episode. Thus, except when the target domain was extremely different from the prior domain, the modified algorithm performed better than the original. Each line is the average of 24 independent learning runs. The error bars have been left off for clarity, but are similar in magnitude to those in Figure 4-2.



(a) Varying friction coefficient

(b) Varying torque

Figure 4-4: Results when varying the friction coefficient and the actuator force from the prior to the target domain in the cart-pole domain. As in the inverted pendulum (see Figure 4-3), the modified algorithm is robust to significant difference between the prior and the target domain.

the expected improvement of using a simulated prior in a domain as opposed to using a zero-mean prior. Intuitively, data from an arbitrarily poor simulator should be expected not to be useful, and possibly even be harmful to the learning process if it causes the gradient-based policy optimizer to look in the wrong directions. For instance, in Figure 4-3(b), when the true domain thrust is 200% higher than the simulated domain thrust, the optimal policies are fundamentally different between the domains. The additional thrust allows the pendulum to go directly from hanging down to upright without a swing-back. The zero-mean prior algorithm discovers this simple policy after a single iteration; however, using the simulated prior, the learning is biased towards policies that use a swing-back to reach the goal position, temporarily causing worse performance. Further discussion on whether the proposed algorithm will improve the learning performance is contained in Section 4.3.

4.2.2 Using a Simulated Prior on an Inverted Pendulum

This section verifies the performance of Algorithm 5 on a physical implementation of the inverted pendulum, shown in Figure 4-1. The pendulum is actuated using two propellers driven by independent brushless motors. The propellers are facing opposite

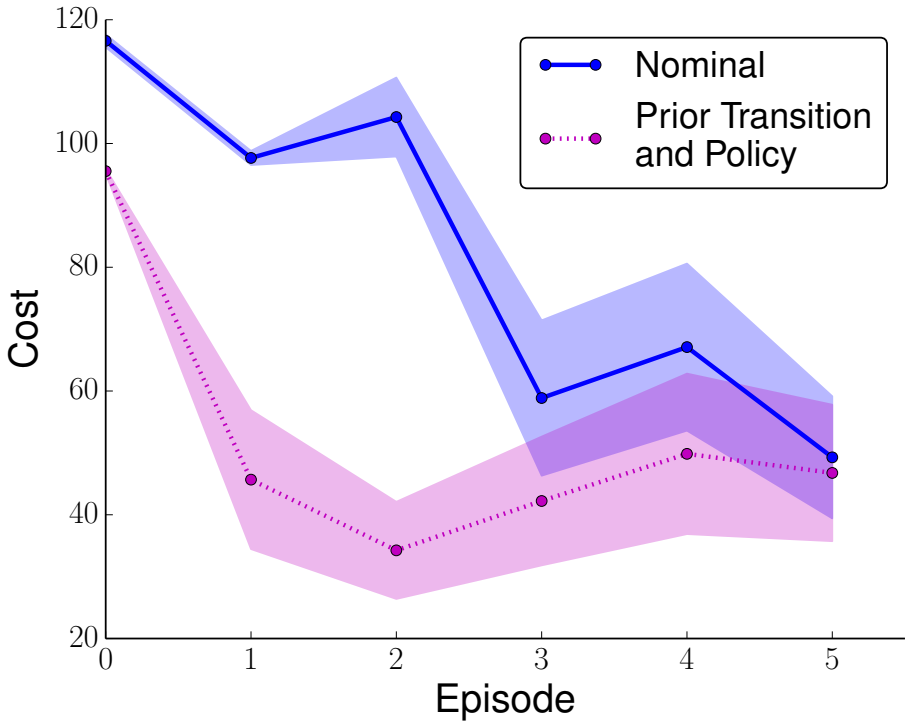


Figure 4-5: Difference between PILCO with a zero-mean prior and the proposed extension with a nonparametric prior coming from a simulator when implemented on a physical inverted pendulum. In the simulated prior both the dynamics and the policy are passed to the real domain. Each algorithm was run three times, with the policy at each episode evaluated 5 times. Error bars show a 95% confidence interval on the standard error of the mean. On average, the additional knowledge from the simulator led the physical system to converge to a good policy in just one episode, whereas without prior knowledge PILCO required at least three learning episodes before the pendulum could be stabilized.

directions, blowing air away from each other. The control input to the motors keep the propellers spinning at all times as brushless motors have a significant spin-up time when starting from rest. The model of the inverted pendulum consists of two states, the angle θ and the angle rate $\dot{\theta}$. The equation of motion is given as

$$I\ddot{\theta} = FL - b\dot{\theta} - T_{gravity},$$

where I is the moment of inertia of the pendulum, F is the force produced by the propellers, L is the distance from the pivot to the propellers, b is the friction coefficient, and $T_{gravity}$ is the torque from gravity.

The angle and angle rate of the pendulum are estimated using on-board inertial sensors (rate gyro and accelerometer). These estimates are relatively low-noise and so are treated as truth in the learning process. Policy roll-outs are performed by sending policy parameters to the on-board microcontroller which, in turn, implements a nonlinear deterministic GP policy using the received parameters. Observed state values and control inputs are sent back to the host computer upon completion of the roll-out as the policy learning is not performed on-board the robot. The control is executed at 20 Hz and each roll-out is 3 seconds long, giving 60 additional data points for each new policy update phase.

The prior comes from a very simple simulation of the physical pendulum with parameters such as mass, inertia, and commanded force roughly measured or estimated. Figure 4-5 shows a comparison of the proposed algorithm with both prior transition dynamics and initial policy parameters coming from the simulator. The simple simulator does not model physical effects such as aerodynamic interactions between the propellers or the change in generated torque as a function of angular rate. Thus, the initial policy parameters do not do much better than random control signals at stabilizing the system (episode 0 in Figure 4-5). However, with just 3 seconds of data from the initial policy roll-out, the proposed algorithm is able to consistently stabilize the pendulum. While the simulator does not model the physical system well enough to be used to generate an optimal policy, it is still very useful for increasing learning efficiency on the physical system. The zero-mean original PILCO algorithm, on the other hand, requires on average at least 3 times as much data from the real system before a stabilizing controller is learned.

Snapshots of the algorithm performance is shown in Figure 4-6. A video of the hardware using the two algorithms can be found at `http://youtu.be/kKClFx6l1HY`.

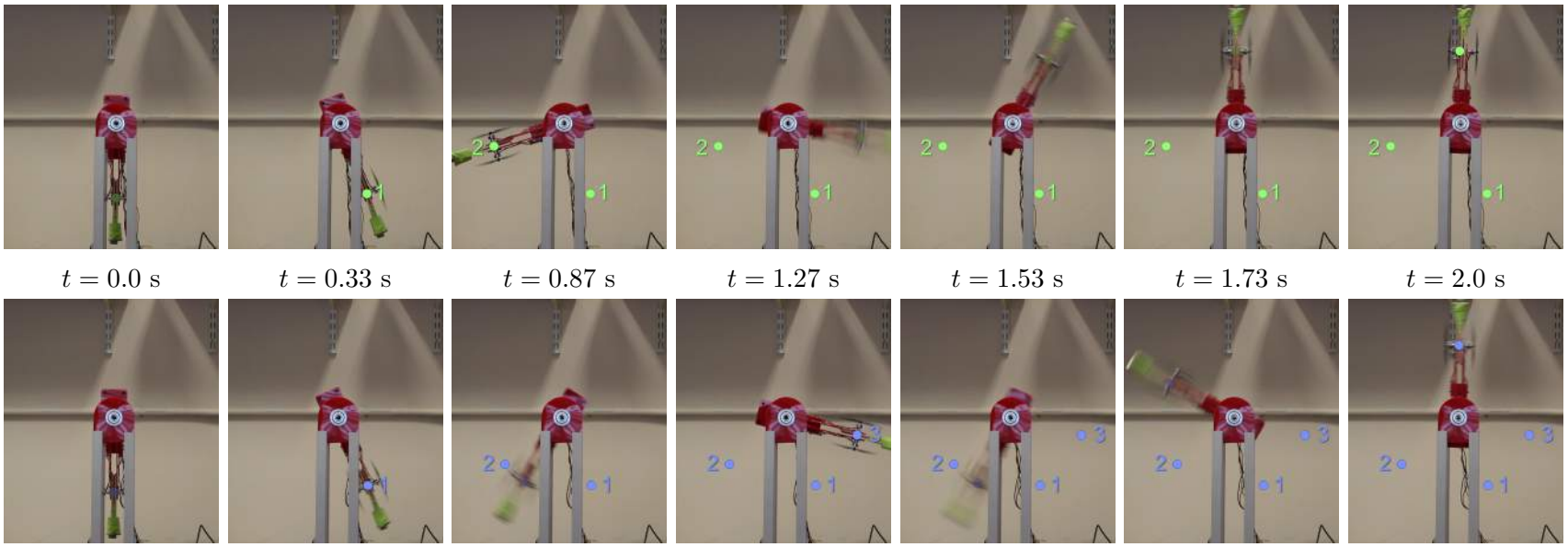


Figure 4-6: Snapshots of example converged policies using prior information (top row) and without prior information (bottom row). Note that, when using an informative prior, the hardware consistently converges to a policy requiring only two swing-backs, indicated by the green dots in the images. Conversely, when using a random policy initialization and no prior information, the hardware frequently converged to a suboptimal policy using three swing-backs, as in the bottom row. This behavior was also seen when transferring between simulators, as discussed in Section 4.2.1 The relative time of each of the snapshots is shown between the two rows. The video containing these results can be found at `http://youtu.be/kKClFx6l1HY`.

4.3 Discussion

The transfer framework developed in this chapter is centered around the PILCO algorithm. While other continuous state-action RL algorithms exist, PILCO is advantageous because it is model based, efficient, and tends to work well on real hardware. On the other hand, PILCO takes a long time to run during the off-line phase as the long-term GP predictions are expensive to compute.

PILCO does not have exploration strategy guarantees but rather relies an exploration-exploitation strategy that emerges from the saturating cost function used in the experiments. While this strategy appears to work well in general, as shown in the previous section, in my experience PILCO does not always converge to the global optimum. Even in simple problems, such as the inverted pendulum, the algorithm occasionally gets stuck in local minima. One advantage of the proposed modifications to the PILCO algorithm is the ability to apply any applicable learning strategy in the simulated domain and then use the data to form a prior for the modified PILCO algorithm. Also, since running the simulator is most likely much cheaper and easier than running the physical hardware, many learning instances could be run in the simulator before turning on the physical hardware.

The results in the previous section made sure to use data for the prior from a learning instance where the algorithm converged to near-optimal behavior. As such behavior will not, in general, be known beforehand in any giving learning setup, the prior could instead be chosen by repeatedly applying a learning algorithm in the simulator with varying initial conditions and then using the data from the best performance.

Note that using a simulated prior will not increase the algorithm performance in all cases. If the simulation is arbitrarily poor, the prior can bias the transition dynamics in the wrong directions. Given sufficient data from the target domain, the observed data will eventually overpower the prior. However, since PILCO is a policy search algorithm that follows local gradients, a poor prior can alter the policy search directions and lead to different policies than had no prior been used.

Note also that, as discussed in [7], applying policies learned in imperfect simulations can yield arbitrarily poor performance, possibly even causing harm to the robot if the policy is overly aggressive. Thus, transferring the initial policy should be done with care. If there is danger of harming the robot, the prior dynamics model can still be used, but with a less aggressive initial policy.

4.4 Summary

This chapter introduced a method for incorporating data from arbitrary simulators in the PILCO learning framework. The simulated data are used to learn a GP model of the simulated transition dynamics. The mean of these dynamics is then used as an informative prior for the real transition dynamics in the target domain. The appropriate equations were derived for predicting the mean and covariance of uncertain inputs using a GP with a mean modeled as an RBF network. The proposed extensions to the PILCO algorithm are demonstrated to result in faster and more robust convergence to good policies than the original PILCO algorithm. These results are shown in both simulated domains and on a physical inverted pendulum using a simple simulator.

Chapter 5

Reverse Transfer for Continuous States and Actions

This chapter introduces a framework, outlined in Figure 5-1, that combines simple and complex simulators with a real-world robot to efficiently find good policies, while minimizing the number of samples from the physical robot. Similar to MFRL from Chapter 3, the framework combines the strengths of various simulation levels. This chapter, though, focuses on domains that are modeled with continuous states and actions. The framework begins by first finding globally optimal policies in a simple model (Σ_s), and then using that solution to initialize a gradient-based learner in a more complex simulation (Σ_c). The policy and transition dynamics from the complex simulation are in turn used to guide the learning in the physical world (Σ_{rw}), as described in Chapter 4. In this chapter a method is developed for transferring information gathered in the physical world *back* to the learning agent in the complex simulation. The new information is used to re-evaluate whether the original simulated policy is still optimal given the updated knowledge from the real-world.

As shown in Chapter 3, this *reverse* transfer is critical to minimizing samples from the physical world. However, when using policy search algorithms, the reverse transfer is even more important as poor initial policies from the simple and complex simulations can lead the real-world learning agent to converge to a suboptimal policy. As with policy search methods in general, the framework presented here will not

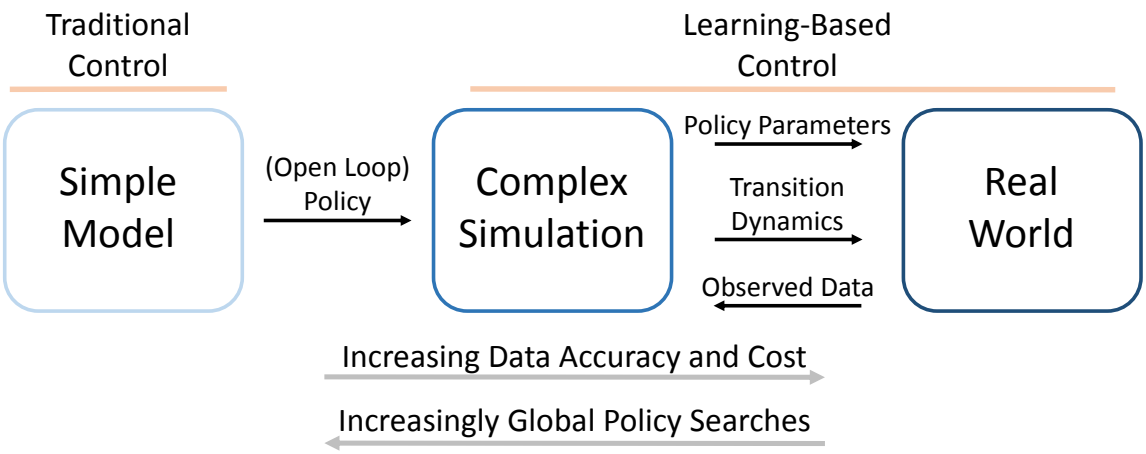


Figure 5-1: Many robotic systems utilize models of varying fidelity for design, validation, and control. Simple models are good for finding globally optimal policies using traditional control, while more complex simulations (including the real world) can use learning-based control methods to refine initial policies. By sending observed data from the real world back to the complex simulation, better policies are found in the real world with fewer samples.

guarantee convergence to an optimal policy, but, as shown empirically, will increase the probability of converging to a good policy while using few expensive real-world data samples.

5.1 Using Optimal Control to Initialize Policy Search

Because continuous state-action spaces are prohibitively large for performing global searches, many RL researchers use prior knowledge to initialize learning algorithms. This prior information often comes in the form of an initial policy from an expert demonstration [52]. Many examples of these initializations exist, such as learning helicopter aerobatics [1] and learning to play table tennis [74]. These expert initial policies give local search algorithms a good initial policy from which to start a policy search.

While initial policies from expert demonstrations can be extremely valuable, these demonstrations may not always be available. As an alternative to expert demonstrations, in this chapter, initial policies generated from simple models using traditional control techniques are used to initialize the learning in complex simulations. Control

methods, such as optimal control, provide globally optimal control policies and so are well suited for initializing local search algorithms. Gauss Pseudospectral Optimization Software (GPOPS) [80] is used to compute initial policies using simplified models of the environment. The resulting policies are then fed to a gradient-based learning agent in the complex simulation, as indicated in Figure 5-1.

Note that additional learning beyond the resulting policy from GPOPS is only needed when discrepancies exist between reality and the simple model. As any model (especially a simple one) will rarely, if ever, perfectly match the physical system, applying the open-loop optimal control policy from GPOPS to the real robot will, in general, result in sub-optimal or even dangerous behavior [7]. Therefore, these policies are instead treated as an initialization for the learning algorithm in the complex simulation.

The policies learned in the complex simulation are parameterized, with the parameter values found by the learning algorithm. As in Chapter 4, the primary focus in this chapter is on policies that are represented using a regularized radial basis function (RBF) network (equivalent to a deterministic Gaussian process), where the centers, the associated targets, and the length-scale values of the RBFs are the parameters to be learned. The RBF network provides a general function approximation scheme for representing various policies.

To transfer the open-loop optimal control policy to the complex simulator, the policy is approximated as a closed-loop controller using an RBF network. Given a fixed budget size on the number of RBFs allowed (typically based on the computational constraints of the learning algorithm in the complex simulation), the centers and targets of the RBF network are found by applying the k-means algorithm [41] to a discretrized representation of the optimal control policy and corresponding optimal states from the simple model. The length scale parameters are then found using evidence maximization, just as the hyperparmeters of a Gaussian process are typically computed. In practice, this is a quick and easy way to train the RBF network using the optimal control policy.

5.2 Reverse Transfer Using Gaussian Processes

The framework from Figure 5-1 is shown in Algorithm 6. The algorithm first uses optimal control methods to find a policy π_s^* that is globally optimal in the simple model (line 2). This policy is then approximated as an RBF network in line 3. Using these policy parameters, the PILCO algorithm is used to refine and update the initial policy using the more complex simulation in line 4. Once converged, the framework passes these new policy parameters and learned transition dynamics from the complex simulation to an instance of PILCO running in the real world (line 6), as described in Chapter 4.

After real data are observed, they can be passed back to the simulator and used to re-plan in much the same way that the MFRL algorithm re-plans using higher-fidelity data in Chapter 3 (line 7). Finally, in line 8, the algorithm exits when the re-planned policy is not sufficiently different from the policy previously found in the simulator, indicating that the new data provided by the real world will not cause new policy performance or new exploration in the simulator.

In practice, and where sufficient computational power exists, several instances of PILCO in the simulation environment should be run in parallel. One of the main advantages of using a simulator is the low cost of obtaining samples. Thus, to increase the probability of converging to the global optimum, or of converging to a new local optimum, the simulation can be run in parallel with various random seeds and initial conditions. Then, the best performing or most different policy can be tried on the real system.

To re-plan using real-world data, the algorithm must determine if a given state-action pair is sufficiently well known. When using discrete representations of the state-action space, the known-ness check is binary and depends on whether or not the current state-action pair has been observed a sufficient number of times, based on an assumption of how much noise is inherent in the system (see Section 3.3.1). With continuous states and actions, the number of times a specific state-action pair has been observed can not be counted as it is unlikely that the exact same data will

Algorithm 6 Continuous State-Action Reinforcement Learning using Multi-Direction Information Transfer

Input: Simple, deterministic simulation Σ_s, more complex, stochastic simulator Σ_c, and real world Σ_{rw}
Use optimal control methods to find policy π_s^* in Σ_s
Use k-means to approximate π_s^* as π_c^{init} with an RBF network
Run PILCO in Σ_c with $\pi_c = \pi_c^{init}$ as initial policy
while 1 **do**
    Run PILCO in Σ_{rw} using Algorithm 5
    Run PILCO in Σ_c to get π_c^{new}, combining GP predictions from Σ_{rw}
    if $\|\pi_c^{new} - \pi_c\| < \epsilon$ **then**
        break
    else
        $\pi_c = \pi_c^{new}$

be observed multiple times. Instead, the algorithm must determine if the current state-action pair is sufficiently "close" to states and actions about which there is little uncertainty. For this purpose, the covariance of the GP's representing the state-action space is used as a known-ness measure.

As described in Section 2.4, for a single output dimension the predictive covariance is

$$\Sigma_* = k(\boldsymbol{x}_*, \boldsymbol{x}_*) - k(\boldsymbol{x}_*, X)(K + \sigma_n^2 I)^{-1} k(X, \boldsymbol{x}_*). \tag{5.1}$$

This covariance is a measure of how uncertain the GP is about the true function at the test input $\boldsymbol{x}_*$, minus the learned noise variance $\sigma_n^2 I$. Thus, Σ_* approaches zero as the true function is correctly modeled (or at least thought to be correctly modeled). The actual predictive covariance of $\boldsymbol{y}^*$ is $\Sigma_* + \sigma_n^2 I$ and so does not approach zero unless the data observations are noise free. Therefore, the value of Σ_* relative to the underlying noise in the system, σ_n^2, can be used as a measure of how much uncertainty remains about the function at a given test point.

The covariance Σ_* will be used to determine the extent to which data should be trusted from the real world versus relying on simulated data. For the remainder of this section a subscript (rw) is used to indicate variables relating to data from the real world, and a subscript (sim) to denote variables relating to simulated data.

The general approach for using data from the real world when re-planning in the simulation will be to determine, for each sampled state-action pair when doing long-term predictions (when PILCO is sampling its model of the environment to see how well a given set of policy parameters will work), how much information is known about that state-action pair in the real world. As in Chapter 3, when the real-world data is well known, those transition dynamics will be used, otherwise, the algorithm will default to the simulated values. Unlike in MFRL, though, a continuous representation of the known-ness of a given state-action pair can now be used. Based on the ratio of Σ_* to $\sigma_n^2 I$, a scalar mixing value $p_{(rw)}$ is determined that dictates what percentage of the real-world data should be incorporated into the current prediction step. Thus, given $p_{(rw)}$, the predictive mean is a linear combination of the predicted mean from the simulated data and the real-world data as

$$\mu_* = \mu_{*_{(sim)}} + p_{(rw)}(\mu_{*_{(rw)}} - \mu_{*_{(sim)}}). \tag{5.2}$$

Similarly, using standard results of mixing normal distributions, the predictive covariance becomes

$$\begin{aligned}
\beta_{(sim)} &= (\mu_{*_{(sim)}} - \mu_*)(\mu_{*_{(sim)}} - \mu_*)^T + \Sigma_{*_{(sim)}} \\
\beta_{(rw)} &= (\mu_{*_{(rw)}} - \mu_*)(\mu_{*_{(rw)}} - \mu_*)^T + \Sigma_{*_{(rw)}} \\
\Sigma_* &= \beta_{(sim)} + p_{(rw)}(\beta_{(rw)} - \beta_{(sim)}),
\end{aligned} \tag{5.3}$$

where $\beta_{(sim)}$ and $\beta_{(rw)}$ are defined for notational convenience. The covariance between the input and output is similarly defined as

$$\Sigma_{\boldsymbol{x}_*, f_*} = \Sigma_{\boldsymbol{x}_*, f_{*(sim)}} + p_{(rw)}(\Sigma_{\boldsymbol{x}_*, f_{*(rw)}} - \Sigma_{\boldsymbol{x}_*, f_{*(sim)}}). \tag{5.4}$$

In PILCO, given a multidimensional problem, each output dimension is modeled using a separate GP. With deterministic inputs, these separate GP's would be independent, but in PILCO uncertain inputs are passed through the GP's to get long-term cost and state predictions. Given these uncertain inputs, the outputs of the GP's co-

Table 5.1: Parameter values used for the generalized logistic function that dictates the extent to which state-action pairs are known in the real world.

	Q	B	x_0
Value	1.5	400	0.02

vary and are no longer independent. Thus, for a given state-action pair, an individual $p_{(rw)}$ can not be determined for each output dimension, but a single scalar value is found based on the entire output covariance matrix.

For smoothness properties (derivatives of $p_{(rw)}$ will be needed later) a generalized logistic function $f(x)$ defined as

$$\mathrm{f}(x) = \frac{1}{(1 + Q\,\mathrm{e}^{B(x-x_0)})^{\frac{1}{Q}}} \tag{5.5}$$

is chosen to determine $p_{(rw)}$ based on the norm of the current covariance matrix and the norm of the learned noise variance parameters representing the noise in the observed data. The mixing probability $p_{(rw)}$ is therefore defined as

$$p_{(rw)} = \mathrm{f}\left(\frac{\|\Sigma_{*_{(rw)}}\|_F}{\|\left[\sigma^2_{n_1}, \ldots, \sigma^2_{n_E}\right]\|}\right) \tag{5.6}$$

where $\|.\|_F$ is the Frobenius norm. A plot of $p_{(rw)}$ versus the covariance ratio is shown in Figure 5-2, where the parameters of the logistic function are given in Table 5.1. With these parameters the real-world data are only used if there is very little uncertainty about the true function. Otherwise, the simulated data are used. The shape of the logistic function plays the same role that the known-ness parameter m_{known} played in MFRL, and is set based on how noisy the real world is believed to be. In all the experiments tried in this thesis, the parameters in Table 5.1 worked well, without needing to tune them for different domains.

Using equations 5.2-5.4, the transfer of information from the real world back to the complex simulation is achieved by computing $p_{(rw)}$ at each state-action pair during long-term state predictions and using the combined output of the mean, covariance,

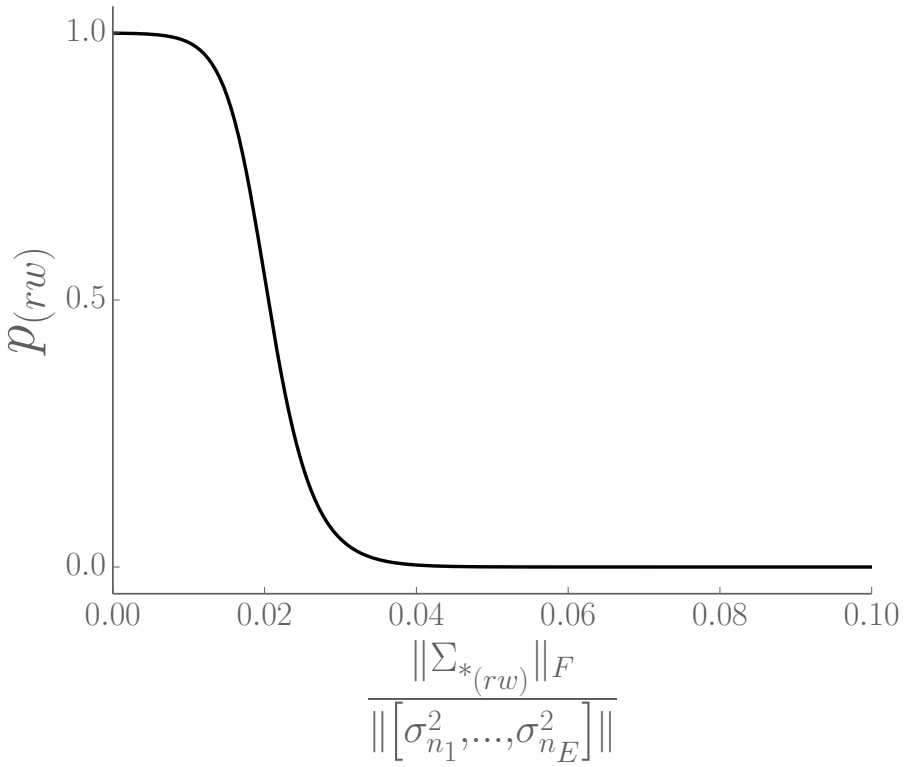


Figure 5-2: Generalized logistic function for determining to what extent data should be used from the real world.

and input-output covariance. As in Chapter 4, to use these equations in PILCO the partial derivatives of these expressions with respect to the input mean and covariance are also needed. These partial derivatives are given in Appendix D.

5.3 Results

Algorithm 6 is implemented in two simulated domains and one real domain to show the applicability of using real-world data to re-plan in a continuous state-action simulator. In the two simulated domains, the "real world" is a simulator that differs from the other simulation, demonstrating the performance of the algorithm in controlled environments.

Table 5.2: Parameter values for the simple 1-D simulation domain.

	σ	a	b
Σ_c	0.25	3	5
Σ_{rw}	0.25	3	1.25

5.3.1 Simple 1-D Problem

The first test domain is a single state, single action domain. This toy domain is used to illustrate how the reverse transfer in line 7 of Algorithm 6 is accomplished. The domain consists of a state x and action u. A saturating cost function $c(x)$ is defined as

$$c(x) = 1 - \mathrm{e}^{-\frac{x^2}{2\sigma^2}}, \tag{5.7}$$

with the dynamics being

$$\dot{x} = \begin{cases} -au^2, & u \geq 0 \\ -bu^2, & u < 0, \end{cases} \tag{5.8}$$

where $|u| \leq 1$. The parameters for Σ_c and Σ_{rw} are given in Table 5.2. Figure 5-3 shows the mean of the dynamics for the simulator and the real world, with a sampling rate of 20 Hz. Each episode starts with $x_0 = 3$ and, according to the cost function, tries to drive x to zero. Both positive and negative values of u result in negative velocities, but with varying magnitudes. In Σ_c, the optimal solution is to start with negative u, while in Σ_{rw} the optimal policy starts with positive u. With this simple domain, the optimal control portion of Algorithm 6 is omitted and π_c^{init} is instead randomly initialized. The parameterized policy in this domain is linear in the state as

$$u = Ax + b, \tag{5.9}$$

rather than using an RBF network as in the other domains.

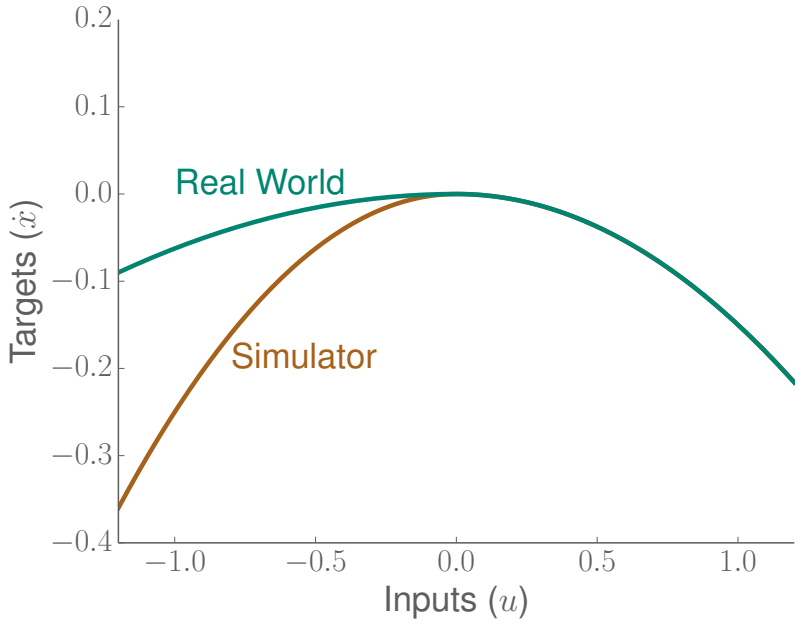


Figure 5-3: Dynamics for the 1-D domain. In the simulator, the optimal solution involves negative u while in the real world the optimal solution uses postive u.

Figure 5-4 shows the performance of Algorithm 6 on this simple problem. Each subplot shows the current representation of the GP modeling the transition dynamics (Equation 5.8). In Figure 5-4(a), the initial random policy explores the state space. After this initial exploration, the first iteration of the policy update phase (lines 7-9 in Algorithm 5) finds the optimal policy as seen by the low cost in Figure 5-4(b) and the new data all on the negative side of u.

The policy and transition dynamics from Σ_c are passed to Σ_{rw} as explained in Algorithm 5. Despite the dynamics being different from the simulator, the gradient-based policy update scheme does not explore policy parameters that would involve positive u values since the initial policy is in a local minimum. As seen in Figures 5-4(c) and 5-4(d), the policy update phase improves the initial policy, but stays on the negative side of u.

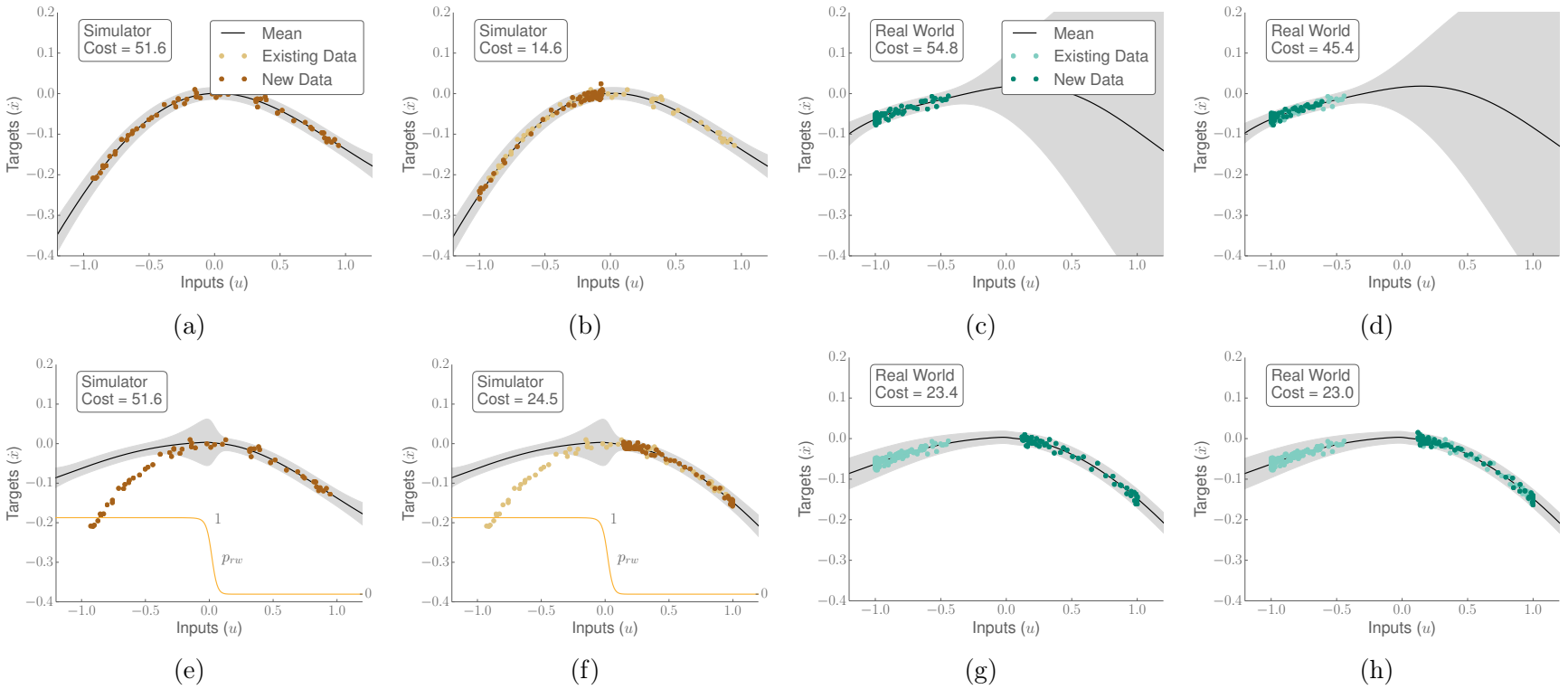


Figure 5-4: Algorithm 6 running on a simple 1-D domain. Each subplot shows the current representation of the GP modeling the transition dynamics (Equation 5.8). In Figures (a) and (b), an initial random policy explores the state space and then quickly converges to the optimal policy, as seen by the low cost in (b) and the new data all in the negative u range. Policy and transition dynamics from Σ_c are passed to Σ_{rw} and used in (c) and (d). The real-world policy is improved, but is stuck in a local minimum based on the initialization. In (e) and (f), model data is passed back to Σ_c which uses data from Σ_{rw} where possible to find a policy involving positive u. This optimal policy (with respect to Σ_{rw}) is passed to Σ_{rw} in (g) and (h) and the algorithm terminates.

Table 5.3: Parameter values for the two hill mountain car domain.

	a	b	m	D	σ
Σ_c	0	1	0.2	0.2	1
Σ_{rw}	2	1	0.2	0.2	1

Following Algorithm 6, the data observed from Σ_{rw} are passed back to Σ_c. The GP in Figures 5-4(e) and 5-4(f) is a combination of the data from Σ_{rw} where the variance was low (negative u values) and Σ_c elsewhere, shown by the embedded plots of $p_{(rw)}$. With this updated hybrid model of the dynamics, the algorithm converges to a policy that favors positive u first. While this new policy is not optimal in Σ_c, is it optimal *given* the data observed from the real world. This new policy is then passed back to Σ_{rw} whereupon the policy improvement phase largely keeps the same policy, converging to the true optimum in Σ_{rw}.

This simple example is clearly contrived to show the performance of Algorithm 6; however, it illustrates an important point of any gradient-based policy improvement method: convergence to local solutions is sensitive to the initial policy. Algorithm 6 attempts to formally provide a method to account for initial policies that come from inaccurate simulators. Note that the policy found in Figure 5-4(d) is not necessarily bad—it solves the problem of driving x to zero. The reverse transfer framework provides a way, though, to revisit the simulator and verify, given the updated information from the real world, whether or not a better policy exists for the real world.

5.3.2 Mountain Car with Two Hills

A slightly more complex simulated domain is now provided that shows Algorithm 6 operating in a multi-dimensional problem. This domain is loosely related to the standard RL mountain car domain [97], but where the car is fully powered and two hills exist. The car is represented by a position x and a velocity $\dot{x}$, with an input $|u| \leq 1$ related to the acceleration of the car. The car starts at $x = 0$ with the goal of driving to the top of either of the two hills in minimum time. The dynamics and

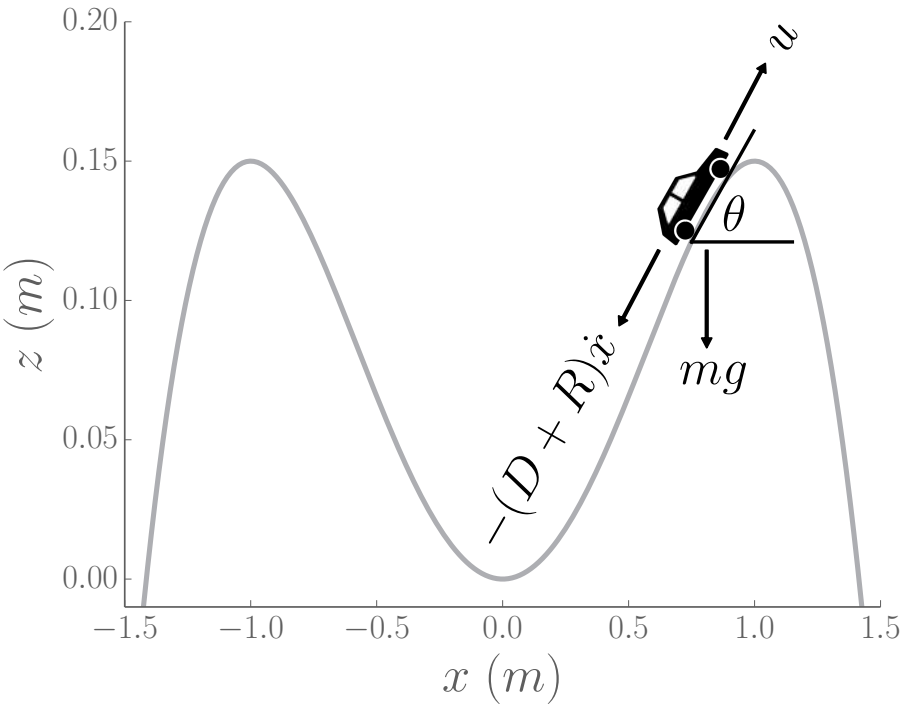


Figure 5-5: The two hill mountain car domain. The cost function encourages driving up one of the two hills while gravity, drag, and a rock penalty term pull the car down.

cost functions are given by

$$m\ddot{x} = u + mg\sin\theta(x) - D\dot{x} - R(a,b,x)\dot{x} \tag{5.10}$$

$$c(x) = 1 - \mathrm{e}^{-\frac{(x-1)^2}{2\sigma^2}} - \mathrm{e}^{-\frac{(x+1)^2}{2\sigma^2}} \tag{5.11}$$

where m is the mass of the car, $\theta(x)$ is the current slope of the hill, g is gravity, D is a drag coefficient, and R is a "rock" penalty term relating to the roughness of the ground. The values of these parameters in the two domains is given in Table 5.3. Figure 5-5 shows the domain with the forces acting on the car, while Figure 5-6 shows the R penalty term as a function of the a and b parameters and x.

Figure 5-7 shows the evolution of Algorithm 6 in the two hill mountain car domain. In each plot the cost function, the position, and the velocity data are shown. The black line shows the roll-out data used to build the GP's, while the dashed lines show 10 more roll-outs with the same policy, showing the repeatability of the policy. The grey area indicates two standard deviations of the predicted values based on PILCO's

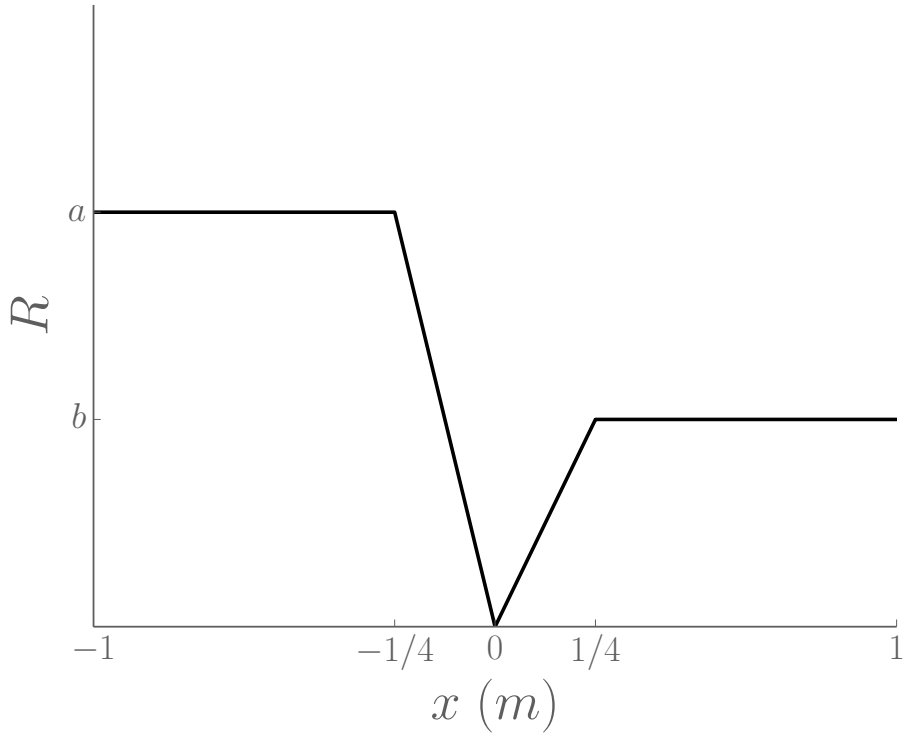


Figure 5-6: Definition of the rock penalty term in Equation 5.10.

understanding of the dynamics.

In this example, GPOPS is used to solve Σ_s, a deterministic version of Σ_c. Figure 5-7(a) shows π_c^{init} from Σ_s applied in Σ_c. The optimal control policy correctly identifies $x_{goal} = -1$ as the optimal strategy given the smaller rock force on that side, but the noise in Σ_c keeps this initial policy from maintaining balance at the top of the hill. Figure 5-7(b) shows the next episode of the PILCO algorithm, showing a policy that robustly solves the problem.

As in the previous section, policy and transition dynamics are passed to Σ_{rw}, where the rock force is much higher. Figures 5-7(c) and 5-7(d) show the first and third episodes in Σ_{rw}. The policy improvement algorithm improves the initial policy, but still drives the car to $x_{goal} = -1$.

The observed data are then passed back to Σ_c where the algorithm uses the information to converge to a new policy, driving instead to $x_{goal} = 1$. This new policy is shown in Figure 5-7(e). Finally, the updated policy and dynamics information are sent to the real world in Figure 5-7(f), where the algorithm keeps the optimal solution

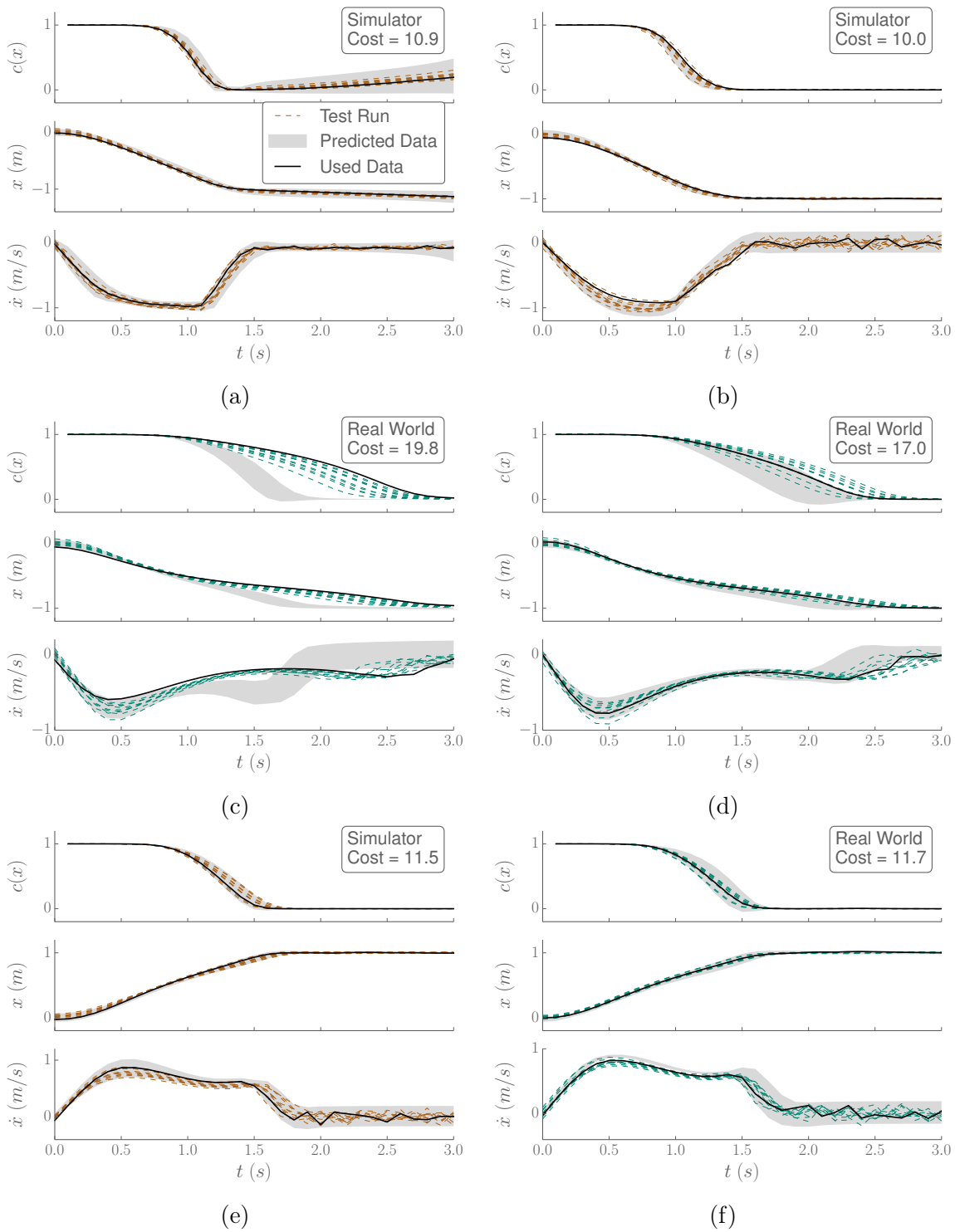


Figure 5-7: Algorithm 6 applied to a mountain car domain with two hills. Please refer to the text for details.

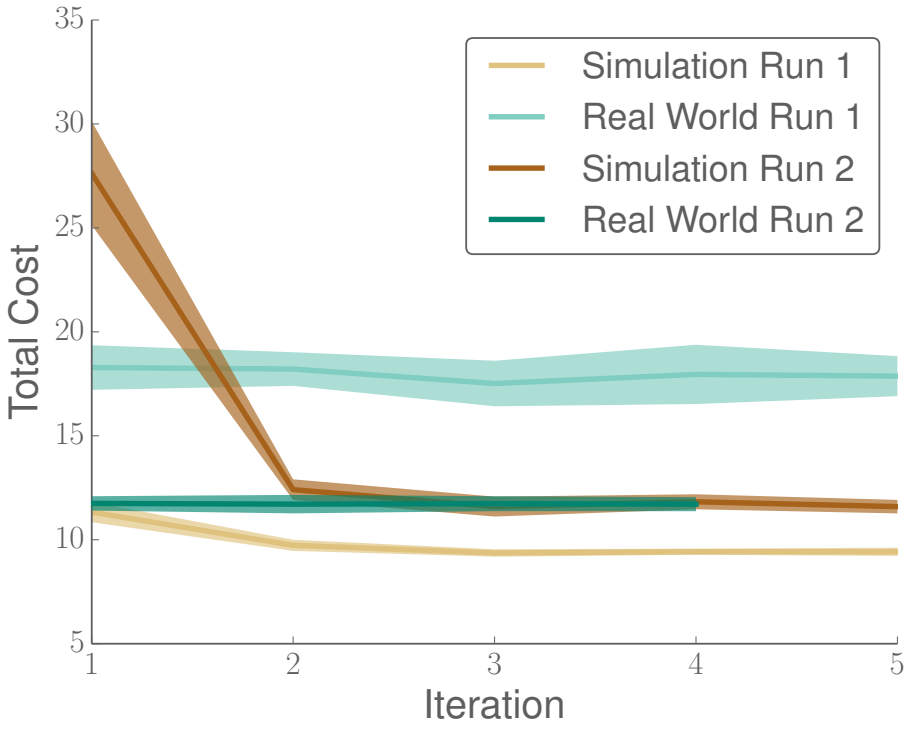


Figure 5-8: Total costs (with one standard deviation shown) of the policies in Σ_c and Σ_{rw} for the two hill mountain car domain. By returning to the simulation after the running the real world, the algorithm is able to improve the policy in the real world.

of driving to $x_{goal} = 1$.

A summary of the total costs of the policies in the two environments is shown in Figure 5-8. Using GPOPS, PILCO is able to quickly converge to the optimal solution in Σ_c during the first run. This policy, however, leads to a sub-optimal solution during the first real-world run, as the policy improvement step gets stuck in a local minimum. With a random re-initialization of the policy and the observed data from the real world, the algorithm converges to the policy that is optimal in the real world.

5.3.3 Drifting Car

The final example in this chapter shows Algorithm 6 applied to a small robotic car learning to drift, or drive sideways. The car is shown in Figure 5-9 and is the same car that was used in the experiments in Section 3.6, but with slick plastic tires instead of the stock rubber tires. With the slick tires on the hard lab floor, the car is quite

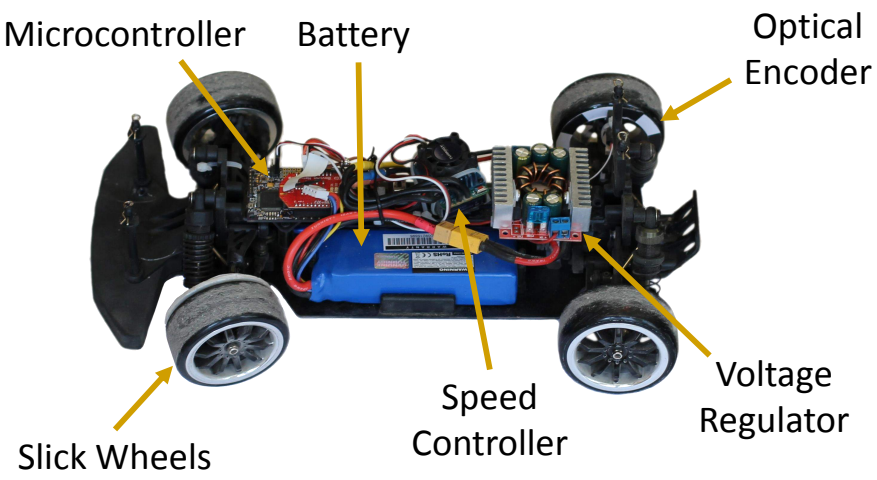


Figure 5-9: Car used for the drifting experiments. The wheels are made of hard plastic, resulting in very low friction on the lab linoleum floor. The car has been outfitted with an optical encoder for wheel speed measurements and a high current voltage regulator. An on-board microcontroller executes the policy roll-outs and transmits data back to a host computer for processing.

difficult to control, with most members of the lab being unable to drive it without running into the walls. More details on the car setup are provided in Appendix B.

The target task in this section is to control the car in a steady-state drift, where the car maintains a constant sideways velocity. Recent work demonstrated an LQR and backstepping control approach to stabilizing a car under steady-state drift conditions [111]. The controller calculates stead-state cornering conditions which rely on exact knowledge of tire and road forces and conditions. In reality, these values will not be known and inaccuracies in these parameters will yield steady-state errors in the control law. Also, the controller developed is only valid around an equilibrium and so open-loop commands based on expert driver data are needed to get the vehicle close to the desired state before the controller can be engaged.

Anecdotal evidence from comparing expert drivers on pavement (such as during stock car racing) to off-road racing (such as rally racing) suggests that when traction between the tires and road is high, minimum-time trajectories involve little, if any, vehicle side-slip; however, when traction is low, optimal trajectories involve sliding, or drifting, sideways around turns. These observations can also be shown

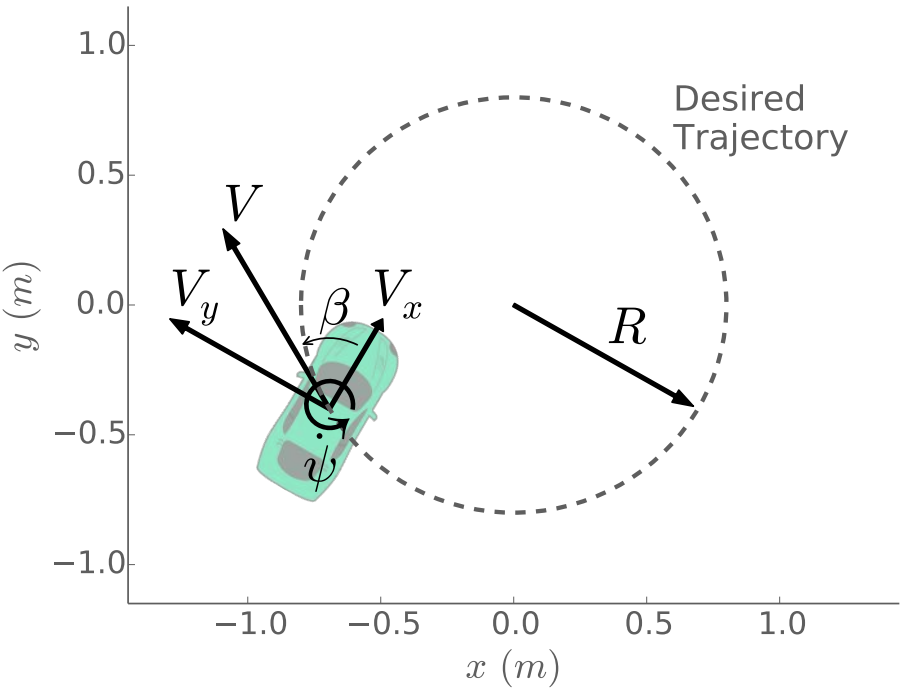
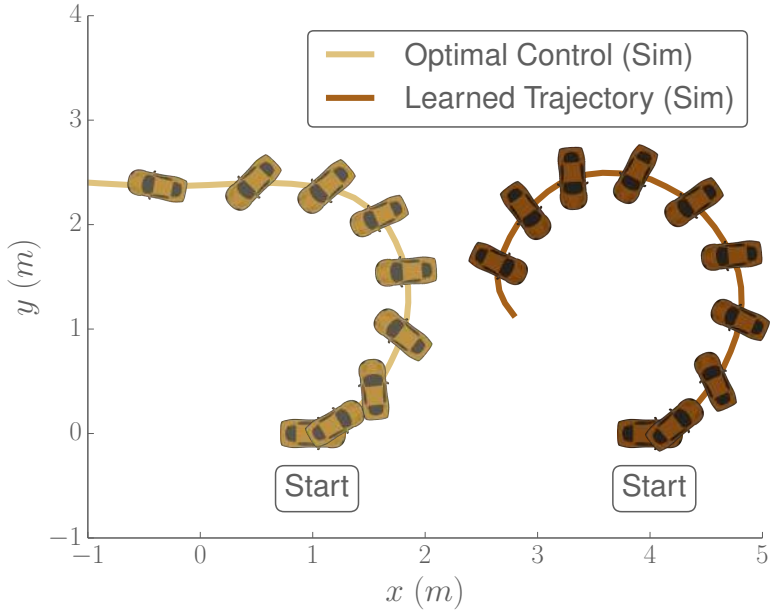


Figure 5-10: State variables used for the steady-state drifting domain. The body-frame forward and side velocities are V_x and V_y, respectively, with V being the total velocity and $\dot{\psi}$ the body turn rate. The slip (or drift) angle is β while R is the radius of curvature of the desired trajectory.

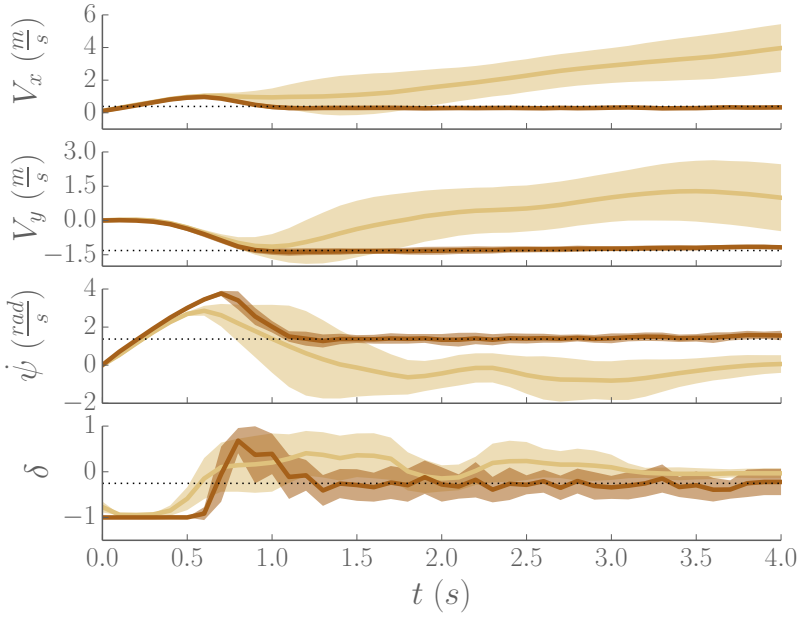
mathematically—as the tire traction decreases, minimum-time trajectories around turns involve more and more side-slip [103]. This example illustrates a way to fully automatically generate feedback controllers that can maintain steady-state drifting motion using very little data from the physical car.

The equations of motion of the complex simulator Σ_c for the drifting car domain are described in Appendix B. The goal of the learning algorithm is to control the vehicle to constant forward, side, and turn rate velocities, resulting in a steady-state drifting motion as indicated by Figure 5-10. The vehicle state consists of body-frame component velocities V_x and V_y, a turn rate $\dot{\psi}$, and the current wheel speed ω. The action δ is the command sent to the on-board servo actuating the wheel steering. The cost function is

$$c(x) = 1 - \mathrm{e}^{\left[-\left((V_x - V_{x_{ss}})^2 + (V_y - V_{y_{ss}})^2 + (\dot{\psi} - \dot{\psi}_{ss})^2\right)/\left(2\sigma^2\right)\right]}, \tag{5.12}$$



(a)



(b)

Figure 5-11: Policy roll-outs of the simulated car learning to drift. The light colored lines show the performance of the optimal policy from Σ_s when first applied to Σ_c. The dark lines show the improvement the policy search algorithm is able to make over the initial policy. A single representative roll-out of each type is shown in (a), while (b) shows the mean and one standard deviation of five learning runs, where each policy was executed five times. Note that only the velocity states are controlled here, and so the position starting locations in (a) are irrelevant.

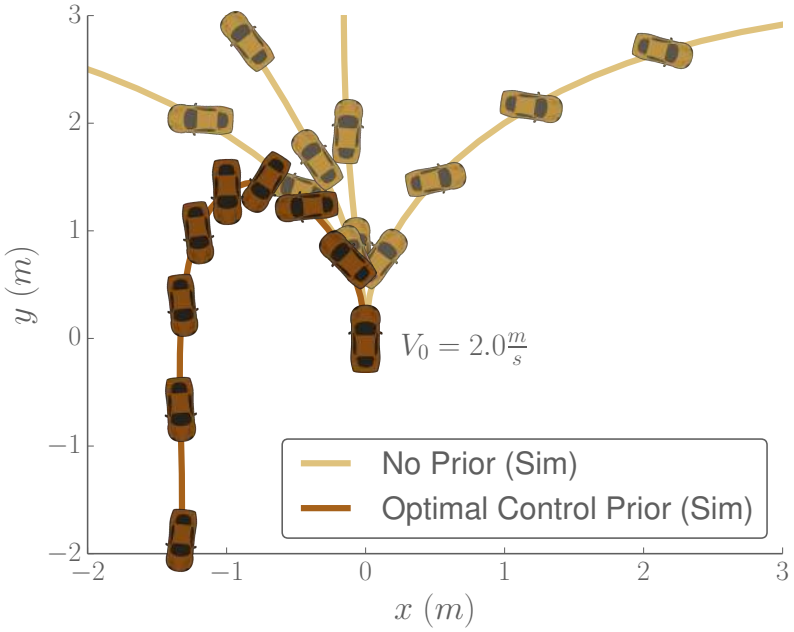
where the subscript ss signifies the desired steady-state values for that variable. The magnitude of these values and how they are computed is described in Appendix B. This cost function favors steady state drifting where the state variables are kept at a constant value, with a non-zero V_y component.

Importantly, the action for the car is now a continuous signal rather than a set of discrete, higher-level path commands like in Section 3.6. The throttle setting was kept constant as a car can maintain a steady-state drift with just modulating the steering command. The car used a high-current switching voltage regulator to keep the battery voltage from affecting the performance of the learning algorithm.

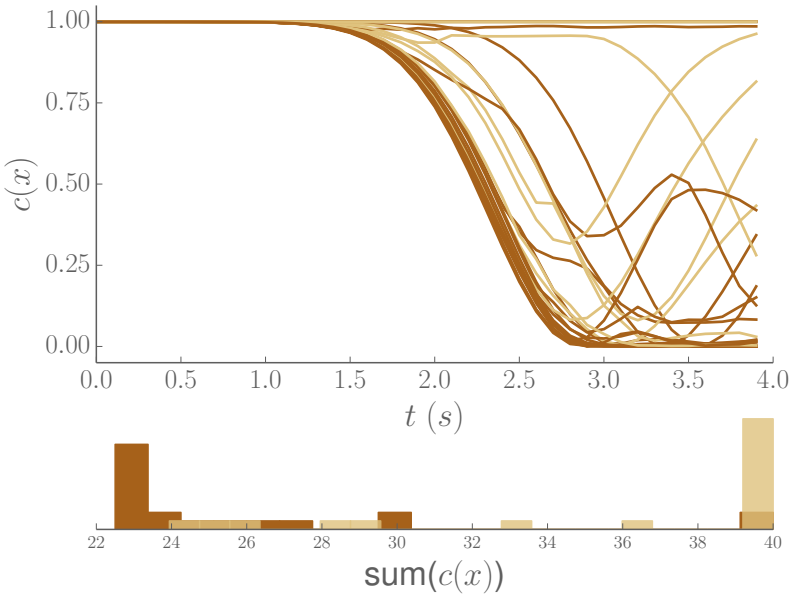
As in the previous domain, Σ_s is a deterministic version of Σ_c. GPOPS is used to solve for π_s^*, the optimal policy with respect to the deterministic simulation. Figure 5-11 shows the performance of the initial policy from GPOPS in the noisy simulator compared to the policy from Σ_c after several policy improvement steps are made. The car is able to quickly initiate a drifting motion by turning into the drift, briefly counter steering, and then settling on a near constant steering angle.

Figure 5-12 shows the performance of the PILCO algorithm in Σ_c when the deterministic simulator is not used. In this example, the car starts with a forward velocity of 2 m/s and then tries to turn around as quickly as possible, maintaining a velocity of 3 m/s in the opposite direction. The cost function penalizes sideways motion, encouraging solutions that reverse direction in minimum time. Here, both the steering and throttle commands are learned. In this domain, the problem is sufficiently complex with enough local minima that PILCO is unable to solve the problem without prior information. The problem was run several times with varying initial random policy parameters, but in almost every case PILCO was unable to find a policy that quickly reversed the car's direction. Figure 5-12(a) shows some representative samples of the performance of PILCO in Σ_c using an optimal control prior versus a randomly initialized policy. In Figure 5-12(b), immediate and cumulative costs of 20 learning runs for each case are displayed. As shown by the histogram, without prior information, PILCO was rarely able to solve the problem.

This behavior is perhaps not unexpected. As noted in [52], many, if not most,

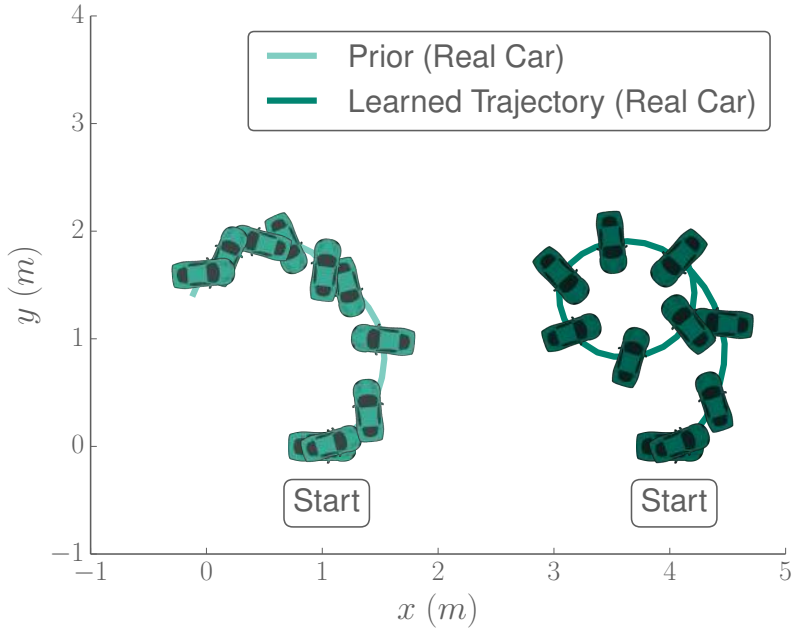


(a)

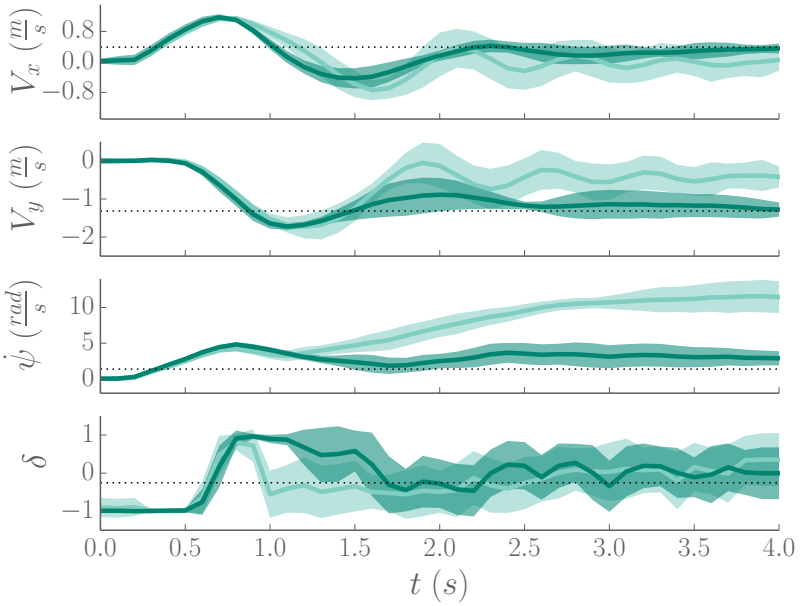


(b)

Figure 5-12: The effect of a good prior on the performance of PILCO. The light colored lines show the results of applying PILCO to the car turning domain with randomly initialized policy parameters (showing converged policy). In almost every case, the algorithm is unable to find the correct control inputs to turn the car around, instead getting stuck in local optima. Figure (a) shows some representative samples of the two cases while (b) shows the instantaneous and cumulative costs for 20 learning runs for each case. This figure highlights the need for the optimal control prior for complicated domains.



(a)



(b)

Figure 5-13: Policy roll-outs of the real car learning to drift. The light colored lines show the performance of the optimal policy from Σ_c when first applied to Σ_{rw}. The dark lines show the improvement the policy search algorithm is able to make over the initial policy. A single representative roll-out of each type is shown in (a), while (b) shows the mean and one standard deviation of three learning runs, where each policy was executed five times. The real car requires a more counter steering than the simulated car to slow the initial vehicle rotation. Note that only the velocity states are controlled here, and so the position starting locations in (a) are irrelevant.

Figure 5-14: Results of Algorithm 6 applied to the real drifting car. Snapshots of the car every 0.25 seconds are shown. The vehicle starts from rest and quickly enters a controlled drifting motion with a 0.8 m radius.

researchers doing RL in real robotics domains initialize the problem with problem-specific data, often through example demonstrations or hand-crafted policies. In this domain, no one in the lab was expert enough to drive the car in a steady-state drift. It was also not immediately clear how to hand-craft a policy that would lead to this behavior. Thus, the proposed algorithm uses principles of optimal control to take the place of these demonstrations and specific policies.

Figure 5-13 shows the performance of Algorithm 6 applied to the real car. Figure 5-14 shows snapshots of the car as it begins a steady-state drift from rest. The car is able to initiate and maintain a steady-state drifting motion with a trajectory radius of approximately 0.8 m. Drifting with other trajectory radii was also successfully accomplished. The learned real-world policy looks very similar to the optimal policy from Σ_c, but with a longer period of counter-steering (see δ in Figure 5-13 from 0.7–1.5 seconds). This highlights the utility of incorporating simulated data into the learning framework. Rather than having to learn the entire policy from scratch, the

learning agent just needed to learn those parts of the policy that were different in the real world when compared to the simulator.

Due to limitations in the size of the testing area, the drifting domain was unable to be solved without prior information from the simulators. Executing random actions (the typical initialization for PILCO) always resulted in the car quickly running into a wall. In this case, the simulated data are necessary to solve the problem.

A video showing the performance of the drifting car can be seen at `https://youtu.be/opsmd5yuBF0`.

5.4 Summary

This chapter developed a framework for incorporating optimal control solutions into the learning process and for transferring data from real-world experiments back to simulators, allowing the simulator to re-plan using the updated information. The re-planning both validates policies from the real world and possibly leads to better real-world policies by exploring more of the state-action space in the simulator. The reverse transfer was combined together with the forward propagation from Chapter 4, resulting in a continuous state-action version of MFRL. The framework was applied to two simulated domains, a simple 1-D example and a two-hill mountain car, both showing that the reverse transfer is necessary to find the optimal solution in the real world. Finally, the algorithm was applied to a robotic car learning to drift. The task was sufficiently complex that PILCO was unable to solve the problem without the use of prior information.

Chapter 6

Conclusions and Future Work

This thesis focused on the core idea of using simulators to increase the efficiency of reinforcement learning in the real world. RL for robots is particularly difficult as getting samples from real robots can be tedious. The thesis explored various ways of using simulators to transfer sampling burden from the real robot to the simulators, resulting in two frameworks that increase the efficiency and performance of robotic RL.

Chapter 3 introduced MFRL, which extends lessons from the multi-fidelity optimization community to sequential decision making problems. MFRL uses a sequence of optimistic simulators to transfer heuristics from lower- to higher-fidelity simulators. The framework also allows agents in lower-fidelity simulators to plan using higher-fidelity learned model parameters, a tactic shown to be crucial for minimizing sub-optimal steps in the real world. MFRL can leverage existing robotic simulators to decrease the dependence of RL algorithms the physical hardware.

In MFRL, the learning agents maintain theoretical sample efficiency guarantees over the learning process because of the integration of the KWIK-Rmax framework. The empirical results show that MFRL is also efficient in practice. Experiments with a robotic car show that, not only is the framework theoretically sound, but it is also a practical technique for scaling RL algorithms to real-world decision making problems.

While MFRL primarily focused on domains where the state-action space is represented discretely, Chapters 4 and 5 move towards domains with continuous represen-

tations of the state-action space. Chapter 4 introduced a method for incorporating data from arbitrary simulators in the PILCO policy search learning framework. The simulated data were used to learn a GP model of the simulated transition dynamics. The mean of these dynamics was then used as an informative prior for the real transition dynamics in the target domain. The appropriate equations for predicting the mean and covariance of uncertain inputs using a GP with a mean modeled as an RBF network were derived. The proposed extensions to the PILCO algorithm were demonstrated to result in faster and more robust convergence to good policies than the original PILCO algorithm. These results were shown in both simulated domains and on a physical inverted pendulum using a simple simulator.

Finally, Chapter 5 developed a framework that incorporated the simulated prior from Chapter 4 with a method for sending data back from the real world to the simulator. This reverse transfer was shown to be crucial when operating in domains that have many local minima, as policy search algorithms are particularly sensitive to the initial parameters. The reverse transfer algorithm allowed the learning agent to return to the simulator to verify if the current policy should be modified given the updated information from the real world, without excessively running the physical hardware. The framework also provides a method for using optimal control to initialize policy parameters, significantly increasing convergence time and accuracy, and, in some cases, playing a critical role in convergence to non-trivial solutions. The framework developed in Chapters 4 and 5 was tested on a robotic car learning to drift. Hardware results showed the benefit of using simulated data in the learning process.

6.1 Future Work

There are several possible avenues to extend this current work. The MFRL algorithm developed in Chapter 3 relies heavily on several assumptions that would be interesting to revisit and possibly relax. Perhaps the biggest of these assumptions is the optimistic simulator chain. While the chain of optimistic simulators makes some

intuitive sense, and I believe it to be mostly valid in many domains, verifying that a chain of simulators is optimistic based on the definition in Equation 3.1 is just as difficult as solving the original RL problem in all the levels, as it requires a knowledge of the optimal Q-values. An interesting future work would be to develop methods that can efficiently (and perhaps approximately) check if a chain of simulators are optimistic either before learning begins or as part of the learning process. As part of this, the relaxation parameter β_i could be modified to be a function of the state and action space. That way, if only certain states and actions are not optimistic in a given level, the β parameter can efficiently account for that non-optimism.

Another future area of research could be investigating the mapping function ρ_i, that maps states between different simulator levels. In this thesis, the mapping function was typically assumed to be identity, with only some very preliminary work on non-identity mappings. Using ideas from transfer learning, more intelligent mapping functions could be developed. In realistic learning scenarios, it is very likely that simple simulators will have different state spaces from more complex simulators, and a better understanding of this mapping relationship will be needed to use MFRL in these environments. Along the same lines, it is possible that various levels of simulators might have different action spaces as well. A similar mapping function for action spaces could be added to MFRL.

Finally, there are many interesting research options relating to combining simulators and real environments using model-based policy search methods. This thesis only thoroughly considered one such algorithm, PILCO. Exploring other model-based policy search methods and new ways of transferring information between simulator levels might yield insightful results.

Appendix A

Index to Multimedia Extensions

Extension	Link	Description
Video 1	`https://youtu.be/-UYu0cGER-s`	Example propagation of the MFRL algorithm in the puddle world domain
Video 2	`https://youtu.be/c_d0Is3bxXA`	Demonstration of the MFRL algorithm on a robotic car
Video 3	`https://youtu.be/kKClFx6l1HY`	PILCO using an informative simulated prior applied to an inverted pendulum
Video 4	`https://youtu.be/opsmd5yuBF0`	Continuous state-action MFRL applied to a drifting robotic car

Appendix B

Car Dynamics Simulator

B.1 Race Track Car

The dynamics simulator used in the robotic car results uses slightly simplified versions of equations outlined in [110]. For completeness, the simulation equations used are detailed below. The equations and parameters in this section are used for the race track MFRL results in Chapter 3. The following section gives the equations and parameters used for the drifting results from Chapter 5.

The full state of the vehicle is $s = [x, y, V_x, V_y, \psi, \dot{\psi}, \omega]$, where x and y are the inertial-frame positions, V_x and V_y are the body-frame velocities, ψ is the heading, $\dot{\psi}$ is the rotational rate, and ω is the wheel velocity. Since the car is a 4-wheel drive vehicle where the front and rear wheels are mechanically connected, the simplifying assumption is made that all four wheels are rotating at the same speed.

At each iteration (time step) of the simulation, the derivative of the state s is calculated and then integrated over a fixed time step using a fourth-order Runge-Kutta method. To calculate $\dot{s}$, the total velocity V, the slip angle β, and the vehicle

Table B.1: Parameters used for Car Simulator (rubber tires)

Parameter	**Value**	**Description**
m (kg)	0.906	Mass
I_z (kg m^2)	3.78e-3	Mass moment of Inertia about z
l_F (m)	0.1	Forward body length
l_R (m)	0.107	Rear body length
h (m)	0.028	Height to center of gravity
r (m)	0.03	Wheel radius
B	9.5	Magic parameter
C	1.1	Magic parameter
D	0.62	Magic parameter
δ_{max} (deg)	18	Maximum turn command
ω_{CL^+}	4.45	Wheel control gain
ω_{CL^-}	1.5	Wheel control gain
a_{noise}	8	Acceleration noise variance
$\ddot{\psi}_{noise}$	12	Angular acceleration noise variance
$\dot{\omega}_{noise}$	100	Wheel acceleration noise variance

front and rear inertial frame velocities are first calculated.

$$\beta = \tan^{-1}\left(\frac{V_y}{V_x}\right)$$

$$V = \sqrt{V_x^2 + V_y^2}$$

$$V_{F_x} = V\cos(\beta - \delta) + \dot{\psi} l_F \sin\delta$$

$$V_{F_y} = V\sin(\beta - \delta) + \dot{\psi} l_F \cos\delta$$

$$V_{R_x} = V\cos\beta$$

$$V_{R_y} = V\sin\beta - \dot{\psi} l_R$$

Next, these velocities are used to calculate the theoretical slip quantities

$$\begin{aligned} s_{F_x} &= \frac{V_{F_x} - \omega r}{|\omega r|} & s_{F_y} &= \frac{V_{F_y}}{|\omega r|} \\ s_{R_x} &= \frac{V_{R_x} - \omega r}{|\omega r|} & s_{R_y} &= \frac{V_{R_y}}{|\omega r|} \\ s_F &= \sqrt{s_{F_x}^2 + s_{F_y}^2} & s_R &= \sqrt{s_{R_x}^2 + s_{R_y}^2}, \end{aligned}$$

from which the friction forces can be calculated as

$$\begin{aligned} \mu_{F_x} &= \frac{-s_{F_x}}{s_F \mathrm{MF}(s_F)} & \mu_{F_y} &= \frac{-s_{F_y}}{s_F \mathrm{MF}(s_F)} \\ \mu_{R_x} &= \frac{-s_{R_x}}{s_R \mathrm{MF}(s_R)} & \mu_{R_y} &= \frac{-s_{R_y}}{s_R \mathrm{MF}(s_R)}. \end{aligned}$$

The MF function refers to Pacejka's "magic formula" [78] and is defined as

$$\mathrm{MF(s)} = D \sin\left(C \tan^{-1}\left(Bs\right)\right)$$

with parameters B, C, and D. The front and rear wheel normal loads can then be calculated as

$$\begin{aligned} f_{F_z} &= \frac{l_R mg - hmg\mu_{R_x}}{l_R + l_F + h(\mu_{F_x}\cos\delta - \mu_{F_y}\sin\delta - \mu_{R_x})} \\ f_{R_z} &= mg - f_{F_z}. \end{aligned}$$

Using the above, $\dot{s}$ can now be calculated as

$$\dot{x} = V_x \cos\psi - V_y \sin\psi \tag{B.1}$$

$$\dot{y} = V_x \sin\psi + V_y \cos\psi \tag{B.2}$$

$$\dot{V_x} = \frac{1}{m}\left(mV_y\dot{\psi} + \mu_{F_x} f_{F_z} \cos\delta - \mu_{F_y} f_{F_z} \sin\delta + \mu_{R_x} f_{R_z}\right) + a_{noise} \tag{B.3}$$

$$\dot{V_y} = \frac{1}{m}\left(-mV_y\dot{\psi} + \mu_{F_x} f_{F_z} \sin\delta + \mu_{F_y} f_{F_z} \cos\delta + \mu_{R_y} f_{R_z}\right) + a_{noise} \tag{B.4}$$

$$\ddot{\psi} = \frac{1}{I_z}\left(\left(\mu_{F_y} f_{F_z} \cos\delta + \mu_{F_x} f_{F_z} \sin\delta\right) l_F - \mu_{R_y} f_{F_z} l_R\right) + \ddot{\psi}_{noise} \tag{B.5}$$

$$\dot{\omega} = \omega_{CL^{+/-}}\left(\omega_{des} - \omega\right) + \dot{\omega}_{noise} \tag{B.6}$$

The parameters and their values and descriptions used in the state equations are shown in Table B.1. The parameter values were found by either direct measurement or by fitting collected data to the equations. The input to the simulator consists of a desired wheel speed ω_{des} and a wheel turn angle δ. On the actual car, Equation B.6 is implemented as a PI controller. In the simulation, this controller is modeled as a simple first order system where the gain for increasing wheel speed ω_{CL^+} is larger than the gain for decreasing wheel speed, ω_{CL^-}. The gains were found by matching empirical data to the structure of Equation B.6.

B.2 Drifting Car

The previous section gave the equations of motion and parameters for the car used in the race track hardware experiments from Chapter 3 and pictured in Figure 3-14. In the drifting results from Chapter 5, the same car was used but with new tires, battery, and power electronics. The drifting results required (relatively) high rate feedback controllers, with the output of the learned controller being commands to the on-board actuators. Thus, in this configuration, the on-board wheel speed controller is not needed as the learning algorithm dictates the throttle command to the motor speed controller. To keep battery voltage from affecting the performance of the learning algorithm, the car was outfitted with a 3 cell LiPo battery and a high-

Table B.2: Parameters used for Drifting Car Simulator (slick tires)

Parameter	**Value**	**Description**
m (kg)	1.143	Mass
I_z (kg m^2)	4.8e-3	Mass moment of inertia about z
B	3.0	Magic parameter
C	1.5	Magic parameter
D	0.2	Magic parameter
I_ω (kg m^2)	2.56e-5	Wheel mass moment of inertia

current switching voltage regulator to provide a constant 8 V to the speed controller. Also, to make the drifting problem more interesting, the original rubber tires were replaced with slick, hard plastic tires, resulting in very little friction between the wheels and the laboratory linoleum floor. Finally, the car's front and rear differentials were locked so that all the wheels would spin at the same rate. Table B.2 shows the parameters of the car with the updated battery and new wheels. The parameters not listed are the same as those in Table B.1.

As drifting depends only on the body-frame velocities of the vehicle, the equations of motion are given by Equations B.3-B.5, with the dynamics of the wheel being

$$\dot{\omega} = \frac{1}{I_\omega}\left(T - \frac{f_{F_x} + f_{R_x}}{2} r\right), \tag{B.7}$$

where T is the throttle command.

A steady-state drift is defined as an operating regime where the body-frame vehicle velocities and the wheel speed all maintain constant values [110]. The constraints that $\dot{V}_x = \dot{V}_y = \ddot{\psi} = \dot{\omega} = 0$ give four equations with six unknowns (four state variables and two control variables). Therefore, to find a feasible steady-state drifting condition, two of the six unknowns are fixed and the remaining non-linear system of equations is solved using standard numerical methods. Note that there are many feasible drifting conditions. A manifold of these conditions is created by varying the chosen fixed variables [111]. The steady-state values from the non-linear equation solver become inputs to the cost function given in Equation 5.12.

For the results in Section 5.3.3, the throttle was fixed at 50% and the radius of the steady-state drift was fixed at

$$R = \frac{V}{\dot{\psi}} = 0.8 \text{ m}, \tag{B.8}$$

although experiments where also successfully implemented at other radii. These values result in the following steady-state control variables:

$$\begin{aligned} V_{x_{ss}} &= 0.32 \text{ m/s} \\ V_{y_{ss}} &= -1.18 \text{ m/s} \\ \dot{\psi}_{ss} &= 1.53 \text{ rad/s} \\ \omega_{ss} &= 184.5 \text{ rad/s} \\ \delta_{ss} &= 4.86 \text{ deg}, \end{aligned}$$

which were used in the cost function for both simulation and hardware results.

Appendix C

Prior for PILCO

This appendix derives the main equations and derivatives required to implement the informative prior from a simulator in PILCO [25] (see Chapter 4). First, the mean, covariance, and input-output covariance of the predictive mean of a Gaussian process (GP) are derived when the prior mean is a radial basis function (RBF) network. Then, the partial derivatives of the predictive distribution with respect to the input distribution are detailed.

C.1 Predictive Distribution

Here Eq. (4.1)-(4.3) from Section 4.1 are derived. Following the outline of the derivations in [25] and [15] the predictive mean of uncertain input $\boldsymbol{x}_* \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ is given by

$$\mu_* = \mathrm{E}_{\boldsymbol{x}_*, f} f[f(\boldsymbol{x}_*)] = \mathrm{E}_{\boldsymbol{x}_*}[\mathrm{E}_f[f(\boldsymbol{x}_*)]] = \mathrm{E}_{\boldsymbol{x}_*}[k(\boldsymbol{x}_*, X)\boldsymbol{\beta} + m(\boldsymbol{x}_*)]. \tag{C.1}$$

The prior mean function $m(\boldsymbol{x}_*)$ is assumed to be the mean of a GP that is trained using data from a simulator. Thus,

$$m(\boldsymbol{x}_*) = k_p(\boldsymbol{x}_*, X_p)\boldsymbol{\beta}_p$$

where $\{X_p, \boldsymbol{y_p}\}$ are the simulated data, $\boldsymbol{\beta}_p = (K_p + \sigma_{n_p}^2 I)^{-1}(\boldsymbol{y_p} - m(X_p))$, $K_p = k_p(X_p, X_p)$, and $\sigma_{n_p}^2$ is the noise variance parameter of the simulated data. Note that the prior mean is trained using a zero-prior GP. Substituting the form of the mean function into Eq. (C.1) yields

$$\mu_* = \boldsymbol{\beta}^T \boldsymbol{q} + \boldsymbol{\beta}_p^T \boldsymbol{q}_p, \tag{C.2}$$

where $q_i = \alpha^2 |\Sigma\Lambda^{-1} + I|^{-1/2} \exp(-\frac{1}{2}\boldsymbol{\nu}_i^T(\Sigma + \Lambda)^{-1}\boldsymbol{\nu}_i)$ with $\boldsymbol{\nu}_i = \boldsymbol{x}_i - \boldsymbol{\mu}$. The corresponding prior terms are similar with $q_{p_i} = \alpha_p^2 |\Sigma\Lambda_p^{-1} + I|^{-1/2} \exp(-\frac{1}{2}\boldsymbol{\nu}_{p_i}^T(\Sigma + \Lambda_p)^{-1}\boldsymbol{\nu}_{p_i})$ and $\boldsymbol{\nu}_{p_i} = \boldsymbol{x}_{p_i} - \boldsymbol{\mu}$.

Multi-output regression problems can be solved by training a separate GP for each output dimension. When the inputs are uncertain, these output dimensions covary. The covariance for different output dimensions a and b is now computed as

$$\begin{aligned}\mathrm{Cov}_{\boldsymbol{x}_*, f}[f_a(\boldsymbol{x}_*), f_b(\boldsymbol{x}_*)] =& \mathrm{E}_{\boldsymbol{x}_*}[\mathrm{Cov}_f[f_a(\boldsymbol{x}_*), f_b(\boldsymbol{x}_*)]] + \mathrm{E}_{\boldsymbol{x}_*}[\mathrm{E}_f[f_a(\boldsymbol{x}_*)]\mathrm{E}_f[f_b(\boldsymbol{x}_*)]] \\ &- \mathrm{E}_{\boldsymbol{x}_*}[\mathrm{E}_f[f_a(\boldsymbol{x}_*)]]\mathrm{E}_{\boldsymbol{x}_*}[\mathrm{E}_f[f_b(\boldsymbol{x}_*)]].\end{aligned} \tag{C.3}$$

As noted in [15], due to the independence assumptions of the GPs, the first term in Eq. (C.3) is zero when $a \neq b$. Also, for a given output dimension, $\mathrm{Cov}_f[f_a(\boldsymbol{x}_*), f_b(\boldsymbol{x}_*)]$ does not depend on the prior mean function. Therefore, using the results of [25], the first term in Eq. (C.3) becomes

$$\mathrm{E}_{\boldsymbol{x}_*}[\mathrm{Cov}_f[f_a(\boldsymbol{x}_*), f_b(\boldsymbol{x}_*)]] = \delta_{ab}(\alpha_a^2 - \mathrm{tr}((K_a + \sigma_{\epsilon_a}^2 I)^{-1} Q)), \tag{C.4}$$

where δ_{ab} is 1 when $a = b$ and 0 otherwise, and

$$\begin{aligned} Q &= \int k_a(\boldsymbol{x}_*, X)^T k_b(\boldsymbol{x}_*, X) p(\boldsymbol{x}_*) d\boldsymbol{x}_* \\ Q_{ij} &= |R|^{-1/2} k_a(\boldsymbol{x}_i, \boldsymbol{\mu}) k_b(\boldsymbol{x}_j, \boldsymbol{\mu}) \exp(\tfrac{1}{2} \boldsymbol{z}_{ij}^T T^{-1} \boldsymbol{z}_{ij}) \\ R &= \Sigma(\Lambda_a^{-1} + \Lambda_b^{-1}) + I \\ T &= \Lambda_a^{-1} + \Lambda_b^{-1} + \Sigma^{-1} \\ \boldsymbol{z}_{ij} &= \Lambda_a^{-1} \boldsymbol{\nu}_i + \Lambda_b^{-1} \boldsymbol{\nu}_j. \end{aligned} \tag{C.5}$$

The third term in Eq. (C.3) is computed using Eq. (C.2) as

$$\mathrm{E}_{\boldsymbol{x}_*}[\mathrm{E}_f[f_a(\boldsymbol{x}_*)]]\mathrm{E}_{\boldsymbol{x}_*}[\mathrm{E}_f[f_b(\boldsymbol{x}_*)]] = \left(\boldsymbol{\beta}_a^T \boldsymbol{q}_a + \boldsymbol{\beta}_{p_a}^T \boldsymbol{q}_{p_a}\right) \left(\boldsymbol{\beta}_b^T \boldsymbol{q}_b + \boldsymbol{\beta}_{p_b}^T \boldsymbol{q}_{p_b}\right). \tag{C.6}$$

Finally, the second term in Eq. (C.3) is computed as

$$\begin{aligned} \mathrm{E}_{\boldsymbol{x}_*}[\mathrm{E}_f[f_a(\boldsymbol{x}_*)]\mathrm{E}_f[f_b(\boldsymbol{x}_*)]] =& \mathrm{E}_{\boldsymbol{x}_*}[k(\boldsymbol{x}_*, X)\boldsymbol{\beta}_a k(\boldsymbol{x}_*, X)\boldsymbol{\beta}_b + m_a(\boldsymbol{x}_*)m_b(\boldsymbol{x}_*) + \\ & m_a(\boldsymbol{x}_*)k(\boldsymbol{x}_*, X)\boldsymbol{\beta}_b + k(\boldsymbol{x}_*, X)\boldsymbol{\beta}_a m_b(\boldsymbol{x}_*)]. \end{aligned} \tag{C.7}$$

As above, each term will be computed separately. Using Eq. (C.5), the first term in Eq. (C.7) becomes

$$\mathrm{E}_{\boldsymbol{x}_*}[k(\boldsymbol{x}_*, X)\boldsymbol{\beta}_a k(\boldsymbol{x}_*, X)\boldsymbol{\beta}_b] = \boldsymbol{\beta}_a^T Q \boldsymbol{\beta}_b. \tag{C.8}$$

Similarly, the second term in Eq. (C.7) is

$$\mathrm{E}_{\boldsymbol{x}_*}[m_a(\boldsymbol{x}_*)m_b(\boldsymbol{x}_*)] = \mathrm{E}_{\boldsymbol{x}_*}[k_p(\boldsymbol{x}_*, X_p)\boldsymbol{\beta}_{p_a} k_p(\boldsymbol{x}_*, X_p)\boldsymbol{\beta}_{p_b}] = \boldsymbol{\beta}_{p_a}^T Q_p \boldsymbol{\beta}_{p_b}, \tag{C.9}$$

where Q_p is defined analogously to Eq. (C.5) but using the prior rather than the current data. The third term in Eq. (C.7) is

$$\mathrm{E}_{\boldsymbol{x}_*}[m_a(\boldsymbol{x}_*)k(\boldsymbol{x}_*, X)\boldsymbol{\beta}_b] = \boldsymbol{\beta}_{p_a}^T \mathrm{E}_{\boldsymbol{x}_*}[k_p(X_p, \boldsymbol{x}_*)k(\boldsymbol{x}_*, X)]\boldsymbol{\beta}_b = \boldsymbol{\beta}_{p_a}^T ({}^p\hat{Q})\boldsymbol{\beta}_b, \tag{C.10}$$

where ${}^{p}\hat{Q}$ is defined as

$$
\begin{aligned}
{}^{p}\hat{Q} &= \int k_{p_a}(\boldsymbol{x}_*, X_p)^T k_b(\boldsymbol{x}_*, X) p(\boldsymbol{x}_*) d\boldsymbol{x}_* \\
{}^{p}\hat{Q}_{ij} &= |{}^{p}\hat{R}|^{-1/2} k_{p_a}(\boldsymbol{x}_{p_i}, \boldsymbol{\mu}) k_b(\boldsymbol{x}_j, \boldsymbol{\mu}) \exp(\tfrac{1}{2}({}^{p}\hat{\boldsymbol{z}}_{ij})^T ({}^{p}\hat{T})^{-1} ({}^{p}\hat{\boldsymbol{z}}_{ij}) \\
{}^{p}\hat{R} &= \Sigma(\Lambda_{p_a}^{-1} + \Lambda_b^{-1}) + I \\
{}^{p}\hat{T} &= \Lambda_{p_a}^{-1} + \Lambda_b^{-1} + \Sigma^{-1} \\
{}^{p}\hat{\boldsymbol{z}}_{ij} &= \Lambda_{p_a}^{-1} \boldsymbol{\nu}_{p_i} + \Lambda_b^{-1} \boldsymbol{\nu}_j.
\end{aligned}
\tag{C.11}
$$

The forth term in Eq. (C.7) is analogously defined as $\boldsymbol{\beta}_a^T \hat{Q}^p \boldsymbol{\beta}_{p_b}$, where

$$
\begin{aligned}
\hat{Q}^p &= \int k_a(\boldsymbol{x}_*, X)^T k_{p_b}(\boldsymbol{x}_*, X_p) p(\boldsymbol{x}_*) d\boldsymbol{x}_* \\
\hat{Q}^p_{ij} &= |\hat{R}^p|^{-1/2} k_a(\boldsymbol{x}_i, \boldsymbol{\mu}) k_{p_b}(\boldsymbol{x}_{p_j}, \boldsymbol{\mu}) \exp(\tfrac{1}{2}(\hat{\boldsymbol{z}}^p_{ij})^T (\hat{T}^p)^{-1} \hat{\boldsymbol{z}}^p_{ij}) \\
\hat{R}^p &= \Sigma(\Lambda_a^{-1} + \Lambda_{p_b}^{-1}) + I \\
\hat{T}^p &= \Lambda_a^{-1} + \Lambda_{p_b}^{-1} + \Sigma^{-1} \\
\hat{\boldsymbol{z}}^p_{ij} &= \Lambda_a^{-1} \boldsymbol{\nu}_i + \Lambda_{p_b}^{-1} \boldsymbol{\nu}_{p_j}.
\end{aligned}
\tag{C.12}
$$

Combining Eq. (C.4)-(C.12) the covariance for an uncertain input with multiple outputs is obtained. Writing this covariance element-wise results in

$$
\begin{aligned}
\sigma^2_{ab} =& \delta_{ab}(\alpha_a^2 - \operatorname{tr}((K_a + \sigma^2_{\epsilon_a} I)^{-1} Q)) + \boldsymbol{\beta}_a^T Q \boldsymbol{\beta}_b + \boldsymbol{\beta}_{p_a}^T Q_p \boldsymbol{\beta}_{p_b} + \boldsymbol{\beta}_{p_a}^T {}^{p}\hat{Q} \boldsymbol{\beta}_b + \boldsymbol{\beta}_a^T \hat{Q}^p \boldsymbol{\beta}_{p_b} - \\
& \left(\boldsymbol{\beta}_a^T \boldsymbol{q}_a + \boldsymbol{\beta}_{p_a}^T \boldsymbol{q}_{p_a}\right) \left(\boldsymbol{\beta}_b^T \boldsymbol{q}_b + \boldsymbol{\beta}_{p_b}^T \boldsymbol{q}_{p_b}\right).
\end{aligned}
\tag{C.13}
$$

The final derivation needed for propagating uncertain inputs through the GP transition model in the PILCO algorithm is the covariance between the uncertain test input $\boldsymbol{x}_* \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ and the predicted output $f(\boldsymbol{x}_*) \sim \mathcal{N}(\mu_*, \Sigma_*)$. This covariance is

calculated as

$$\begin{aligned}\Sigma_{\boldsymbol{x}_*,f_*} =& \mathrm{E}_{\boldsymbol{x}_*,f}[\boldsymbol{x}_* f(\boldsymbol{x}_*)^T] - \mathrm{E}_{\boldsymbol{x}_*}[\boldsymbol{x}_*]\mathrm{E}_{\boldsymbol{x}_*,f}[f(\boldsymbol{x}_*)]^T \\ =& \mathrm{E}_{\boldsymbol{x}_*,f}[\boldsymbol{x}_* k(\boldsymbol{x}_*, X)\boldsymbol{\beta}] - \mathrm{E}_{\boldsymbol{x}_*}[\boldsymbol{x}_*]\mathrm{E}_{\boldsymbol{x}_*}[k(\boldsymbol{x}_*, X)\boldsymbol{\beta}]^T + \\ & \mathrm{E}_{\boldsymbol{x}_*,f}[\boldsymbol{x}_* k_p(\boldsymbol{x}_*, X_p)\boldsymbol{\beta}_p] - \mathrm{E}_{\boldsymbol{x}_*}[\boldsymbol{x}_*]\mathrm{E}_{\boldsymbol{x}_*}[k_p(\boldsymbol{x}_*, X_p)\boldsymbol{\beta}_p]^T.\end{aligned}$$

Here, the input-output covariance has been separated into a part that comes from the current data and a part that comes from the prior data. Therefore, the results from [25] can be directly applied to obtain

$$\Sigma_{\boldsymbol{x}_*,f_*} = \Sigma(\Sigma+\Lambda)^{-1}\sum_{i=1}^{n}\beta_i q_i(\boldsymbol{x}_i - \boldsymbol{\mu}) + \Sigma(\Sigma+\Lambda_p)^{-1}\sum_{i=1}^{n_p}\beta_{p_i} q_{p_i}(\boldsymbol{x}_{p_i} - \boldsymbol{\mu}). \tag{C.14}$$

Note that the derivation above does not assume that there are the same number of data points in the prior GP and the current GP. Thus, the matrices ${}^p\hat{Q}$ and $\hat{Q}^p$ need not be square.

C.2 Partial Derivatives

Given the predictive distribution $\mathcal{N}(\mu_*, \Sigma_*)$ from Section C.1, the partial derivative of the predictive mean μ_* with respect to the input mean $\boldsymbol{\mu}$ is first computed. Using the mean derived in Eq. (C.2) results in

$$\begin{aligned}\frac{\partial \mu_*}{\partial \boldsymbol{\mu}} =& \sum_{i=1}^{n}\beta_i\frac{\partial q_i}{\partial \boldsymbol{\mu}} + \sum_{i=1}^{n_p}\beta_{p_i}\frac{\partial q_{p_i}}{\partial \boldsymbol{\mu}} \\ =& \sum_{i=1}^{n}\beta_i q_i(\boldsymbol{x}_i - \boldsymbol{\mu})^T(\Sigma+\Lambda)^{-1} + \sum_{i=1}^{n_p}\beta_{p_i} q_{p_i}(\boldsymbol{x}_{p_i} - \boldsymbol{\mu})^T(\Sigma+\Lambda_p)^{-1}.\end{aligned} \tag{C.15}$$

The derivative of the predictive mean with respect to the input covariance is written as

$$\frac{\partial \mu_*}{\partial \Sigma} = \sum_{i=1}^{n}\beta_i\frac{\partial q_i}{\partial \Sigma} + \sum_{i=1}^{n_p}\beta_{p_i}\frac{\partial q_{p_i}}{\partial \Sigma}, \tag{C.16}$$

where, as in Eq. (C.15), the derivative consists of two distinct parts, one from the current data and one from the prior data. Using results from [25], Eq. C.16 becomes

$$\begin{aligned}\frac{\partial \mu_*}{\partial \Sigma} =& \sum_{i=1}^{n} \beta_i q_i \left(-\tfrac{1}{2}((\Lambda^{-1}\Sigma + I)^{-1}\Lambda^{-1})^T - \tfrac{1}{2}(\boldsymbol{x}_i - \boldsymbol{\mu})^T \frac{\partial (\Lambda + \Sigma)^{-1}}{\partial \Sigma}(\boldsymbol{x}_i - \boldsymbol{\mu}) \right) + \\ & \sum_{i=1}^{n_p} \beta_{p_i} q_{p_i} \left(-\tfrac{1}{2}((\Lambda_p^{-1}\Sigma + I)^{-1}\Lambda_p^{-1})^T - \tfrac{1}{2}(\boldsymbol{x}_{p_i} - \boldsymbol{\mu})^T \frac{\partial (\Lambda_p + \Sigma)^{-1}}{\partial \Sigma}(\boldsymbol{x}_{p_i} - \boldsymbol{\mu}) \right),\end{aligned} \tag{C.17}$$

where, for D input dimensions and E output dimensions and $u, v = 1, \ldots, D + E$

$$\frac{\partial (\Lambda + \Sigma)^{-1}}{\partial \Sigma_{(uv)}} = -\tfrac{1}{2}\left((\Lambda + \Sigma)^{-1}_{(:,u)} (\Lambda + \Sigma)^{-1}_{(v,:)} + (\Lambda + \Sigma)^{-1}_{(:,v)} (\Lambda + \Sigma)^{-1}_{(u,:)} \right), \tag{C.18}$$

and the corresponding prior term

$$\frac{\partial (\Lambda_p + \Sigma)^{-1}}{\partial \Sigma_{(uv)}} = -\tfrac{1}{2}\left((\Lambda_p + \Sigma)^{-1}_{(:,u)} (\Lambda_p + \Sigma)^{-1}_{(v,:)} + (\Lambda_p + \Sigma)^{-1}_{(:,v)} (\Lambda_p + \Sigma)^{-1}_{(u,:)} \right). \tag{C.19}$$

Next, the partial derivatives of the predictive covariance Σ_* with respect to the input mean and covariance are derived. These derivatives are taken element-wise for output dimensions a and b using Eq. (C.13). The derivative with respect to the input mean becomes

$$\begin{aligned}\frac{\partial \sigma^2_{ab}}{\partial \boldsymbol{\mu}} =& \delta_{ab} \left(-(K_a + \sigma^2_{\epsilon_a} I)^{-1} \frac{\partial Q}{\partial \boldsymbol{\mu}} \right) + \\ & \boldsymbol{\beta}_a^T \left(\frac{\partial Q}{\partial \boldsymbol{\mu}} - \frac{\partial \boldsymbol{q}_a}{\partial \boldsymbol{\mu}} \boldsymbol{q}_b^T - \boldsymbol{q}_a \frac{\partial \boldsymbol{q}_b^T}{\partial \boldsymbol{\mu}} \right) \boldsymbol{\beta}_b + \boldsymbol{\beta}_{p_a}^T \left(\frac{\partial Q_p}{\partial \boldsymbol{\mu}} - \frac{\partial \boldsymbol{q}_{p_a}}{\partial \boldsymbol{\mu}} \boldsymbol{q}_{p_b}^T - \boldsymbol{q}_{p_a} \frac{\partial \boldsymbol{q}_{p_b}^T}{\partial \boldsymbol{\mu}} \right) \boldsymbol{\beta}_{p_b} + \\ & \boldsymbol{\beta}_a^T \left(\frac{\partial \hat{Q}^p}{\partial \boldsymbol{\mu}} - \frac{\partial \boldsymbol{q}_a}{\partial \boldsymbol{\mu}} \boldsymbol{q}_{p_b}^T - \boldsymbol{q}_a \frac{\partial \boldsymbol{q}_{p_b}^T}{\partial \boldsymbol{\mu}} \right) \boldsymbol{\beta}_{p_b} + \boldsymbol{\beta}_{p_a}^T \left(\frac{\partial ({}^p\hat{Q})}{\partial \boldsymbol{\mu}} - \frac{\partial \boldsymbol{q}_{p_a}}{\partial \boldsymbol{\mu}} \boldsymbol{q}_b^T - \boldsymbol{q}_{p_a} \frac{\partial \boldsymbol{q}_b^T}{\partial \boldsymbol{\mu}} \right) \boldsymbol{\beta}_b,\end{aligned} \tag{C.20}$$

where, from [25],

$$\frac{\partial Q_{ij}}{\partial \boldsymbol{\mu}} = Q_{ij}((\Lambda_a + \Lambda_b)^{-1}(\Lambda_b \boldsymbol{x}_i + \Lambda_a \boldsymbol{x}_j) - \boldsymbol{\mu})((\Lambda_a + \Lambda_b)^{-1} + \Sigma)^{-1} \tag{C.21}$$

and similarly

$$\frac{\partial Q_{p_{ij}}}{\partial \boldsymbol{\mu}} = Q_{p_{ij}}((\Lambda_{p_a} + \Lambda_{p_b})^{-1}(\Lambda_{p_b}\boldsymbol{x}_{p_i} + \Lambda_{p_a}\boldsymbol{x}_{p_j}) - \boldsymbol{\mu})((\Lambda_{p_a} + \Lambda_{p_b})^{-1} + \Sigma)^{-1} \quad \text{(C.22)}$$

$$\frac{\partial({}^{p}\hat{Q}_{ij})}{\partial \boldsymbol{\mu}} = {}^{p}\hat{Q}_{ij}((\Lambda_{p_a} + \Lambda_b)^{-1}(\Lambda_b\boldsymbol{x}_{p_i} + \Lambda_{p_a}\boldsymbol{x}_j) - \boldsymbol{\mu})((\Lambda_{p_a} + \Lambda_b)^{-1} + \Sigma)^{-1} \quad \text{(C.23)}$$

$$\frac{\partial \hat{Q}^p_{ij}}{\partial \boldsymbol{\mu}} = \hat{Q}^p_{ij}((\Lambda_a + \Lambda_{p_b})^{-1}(\Lambda_{p_b}\boldsymbol{x}_i + \Lambda_a\boldsymbol{x}_{p_j}) - \boldsymbol{\mu})((\Lambda_a + \Lambda_{p_b})^{-1} + \Sigma)^{-1}. \quad \text{(C.24)}$$

Note that $\frac{\partial \boldsymbol{q}}{\partial \boldsymbol{\mu}}$ and $\frac{\partial \boldsymbol{q}_p}{\partial \boldsymbol{\mu}}$ are given in Eq. (C.15).

The derivative of the predictive covariance with respect to the input covariance is

$$\begin{aligned}\frac{\partial \sigma^2_{ab}}{\partial \Sigma} =& \delta_{ab}\left(-(K_a + \sigma^2_{\epsilon_a} I)^{-1}\frac{\partial Q}{\partial \Sigma}\right) + \\ & \boldsymbol{\beta}_a^T\left(\frac{\partial Q}{\partial \Sigma} - \frac{\partial \boldsymbol{q}_a}{\partial \Sigma}\boldsymbol{q}_b^T - \boldsymbol{q}_a\frac{\partial \boldsymbol{q}_b^T}{\partial \Sigma}\right)\boldsymbol{\beta}_b + \boldsymbol{\beta}_{p_a}^T\left(\frac{\partial Q_p}{\partial \Sigma} - \frac{\partial \boldsymbol{q}_{p_a}}{\partial \Sigma}\boldsymbol{q}_{p_b}^T - \boldsymbol{q}_{p_a}\frac{\partial \boldsymbol{q}_{p_b}^T}{\partial \Sigma}\right)\boldsymbol{\beta}_{p_b} + \\ & \boldsymbol{\beta}_a^T\left(\frac{\partial \hat{Q}^p}{\partial \Sigma} - \frac{\partial \boldsymbol{q}_a}{\partial \Sigma}\boldsymbol{q}_{p_b}^T - \boldsymbol{q}_a\frac{\partial \boldsymbol{q}_{p_b}^T}{\partial \Sigma}\right)\boldsymbol{\beta}_{p_b} + \boldsymbol{\beta}_{p_a}^T\left(\frac{\partial({}^{p}\hat{Q})}{\partial \Sigma} - \frac{\partial \boldsymbol{q}_{p_a}}{\partial \Sigma}\boldsymbol{q}_b^T - \boldsymbol{q}_{p_a}\frac{\partial \boldsymbol{q}_b^T}{\partial \Sigma}\right)\boldsymbol{\beta}_b,\end{aligned} \quad \text{(C.25)}$$

where, from [25],

$$\begin{aligned}\frac{\partial Q_{ij}}{\partial \Sigma} =& -\tfrac{1}{2}Q_{ij}\left[(\Lambda_a^{-1} + \Lambda_b^{-1})R^{-1} - \boldsymbol{y}_{ij}^T\Xi\boldsymbol{y}_{ij}\right] \quad \text{(C.26)} \\ \boldsymbol{y}_{ij} =& \Lambda_b(\Lambda_a + \Lambda_b)^{-1}\boldsymbol{x}_i + \Lambda_a(\Lambda_a + \Lambda_b)^{-1}\boldsymbol{x}_j - \boldsymbol{\mu} \\ \Xi_{(uv)} =& \tfrac{1}{2}(\Phi_{(uv)} + \Phi_{(vu)}) \\ \Phi_{(uv)} =& \left(((\Lambda_a^{-1} + \Lambda_b^{-1})^{-1} + \Sigma)^{-1}_{(:,u)} \times ((\Lambda_a^{-1} + \Lambda_b^{-1})^{-1} + \Sigma)^{-1}_{(v,:)}\right).\end{aligned}$$

As before, the terms containing the prior data are similar as

$$\begin{aligned}
\frac{\partial Q_{p_{ij}}}{\partial \Sigma} =& -\tfrac{1}{2} Q_{p_{ij}} \left[(\Lambda_{p_a}^{-1} + \Lambda_{p_b}^{-1}) R_p^{-1} - \boldsymbol{y}_{p_{ij}}^T \Xi_p \boldsymbol{y}_{p_{ij}} \right] && \text{(C.27)} \\
\boldsymbol{y}_{p_{ij}} =& \Lambda_{p_b} (\Lambda_{p_a} + \Lambda_{p_b})^{-1} \boldsymbol{x}_{p_i} + \Lambda_{p_a} (\Lambda_{p_a} + \Lambda_{p_b})^{-1} \boldsymbol{x}_{p_j} - \boldsymbol{\mu} \\
\Xi_{p_{(uv)}} =& \tfrac{1}{2} (\Phi_{p_{(uv)}} + \Phi_{p_{(uv)}}) \\
\Phi_{p_{(uv)}} =& \left(((\Lambda_{p_a}^{-1} + \Lambda_{p_b}^{-1})^{-1} + \Sigma)^{-1}_{(:,u)} ((\Lambda_{p_a}^{-1} + \Lambda_{p_b}^{-1})^{-1} + \Sigma)^{-1}_{(v,:)} \right) \\
\frac{\partial ({}^p\hat{Q}_{ij})}{\partial \Sigma} =& -\tfrac{1}{2} ({}^p\hat{Q}_{ij}) \left[(\Lambda_{p_a}^{-1} + \Lambda_b^{-1}) ({}^pR)^{-1} - ({}^p\boldsymbol{y}_{ij})^T ({}^p\Xi) ({}^p\boldsymbol{y}_{ij}) \right] && \text{(C.28)} \\
{}^p\boldsymbol{y}_{ij} =& \Lambda_b (\Lambda_{p_a} + \Lambda_b)^{-1} \boldsymbol{x}_{p_i} + \Lambda_{p_a} (\Lambda_{p_a} + \Lambda_b)^{-1} \boldsymbol{x}_j - \boldsymbol{\mu} \\
{}^p\Xi_{(uv)} =& \tfrac{1}{2} ({}^p\Phi_{(uv)} + {}^p\Phi_{(uv)}) \\
{}^p\Phi_{(uv)} =& \left(((\Lambda_{p_a}^{-1} + \Lambda_b^{-1})^{-1} + \Sigma)^{-1}_{(:,u)} ((\Lambda_{p_a}^{-1} + \Lambda_b^{-1})^{-1} + \Sigma)^{-1}_{(v,:)} \right) \\
\frac{\partial \hat{Q}^p_{ij}}{\partial \Sigma} =& -\tfrac{1}{2} \hat{Q}^p_{ij} \left[(\Lambda_a^{-1} + \Lambda_{p_b}^{-1}) (R^p)^{-1} - (\boldsymbol{y}^p_{ij})^T \Xi^p \boldsymbol{y}^p_{ij} \right] && \text{(C.29)} \\
\boldsymbol{y}^p_{ij} =& \Lambda_{p_b} (\Lambda_a + \Lambda_{p_b})^{-1} \boldsymbol{x}_i + \Lambda_a (\Lambda_a + \Lambda_{p_b})^{-1} \boldsymbol{x}_{p_j} - \boldsymbol{\mu} \\
\Xi^p_{(uv)} =& \tfrac{1}{2} (\Phi^p_{(uv)} + \Phi^p_{(uv)}) \\
\Phi^p_{(uv)} =& \left(((\Lambda_a^{-1} + \Lambda_{p_b}^{-1})^{-1} + \Sigma)^{-1}_{(:,u)} ((\Lambda_a^{-1} + \Lambda_{p_b}^{-1})^{-1} + \Sigma)^{-1}_{(v,:)} \right).
\end{aligned}$$

Note that $\frac{\partial \boldsymbol{q}}{\partial \Sigma}$ and $\frac{\partial \boldsymbol{q}_p}{\partial \Sigma}$ are given in Eq. (C.17).

The final derivatives are the partial derivatives of the input-output covariance with respect to the input mean and covariance. Note that from Eq. (C.14), $\Sigma_{\boldsymbol{x}_*, f_*}$ consists of two distinct but similar parts, one from the current data and one from the

prior data. Thus, applying results from [25], these derivatives are

$$\begin{aligned}\frac{\partial \Sigma_{\boldsymbol{x}_*, f_*}}{\partial \boldsymbol{\mu}} =& \Sigma(\Sigma+\Lambda)^{-1} \sum_{i=1}^{n} \beta_i \left((\boldsymbol{x}_i - \boldsymbol{\mu}) \frac{\partial q_i}{\partial \boldsymbol{\mu}} - q_i I \right) + \\ & \Sigma(\Sigma+\Lambda_p)^{-1} \sum_{i=1}^{n_p} \beta_{p_i} \left((\boldsymbol{x}_{p_i} - \boldsymbol{\mu}) \frac{\partial q_{p_i}}{\partial \boldsymbol{\mu}} - q_{p_i} I \right) \end{aligned} \tag{C.30}$$

$$\begin{aligned}\frac{\partial \Sigma_{\boldsymbol{x}_*, f_*}}{\partial \Sigma} =& \left((\Sigma+\Lambda)^{-1} + \Sigma \frac{\partial (\Sigma+\Lambda)^{-1}}{\partial \Sigma} \right) \sum_{i=1}^{n} \beta_i q_i (\boldsymbol{x}_i - \boldsymbol{\mu}) + \\ & \Sigma(\Sigma+\Lambda)^{-1} \sum_{i=1}^{n} \beta_i (\boldsymbol{x}_i - \boldsymbol{\mu}) \frac{\partial q_i}{\partial \Sigma} + \\ & \left((\Sigma+\Lambda_p)^{-1} + \Sigma \frac{\partial (\Sigma+\Lambda_p)^{-1}}{\partial \Sigma} \right) \sum_{i=1}^{n_p} \beta_{p_i} q_{p_i} (\boldsymbol{x}_{p_i} - \boldsymbol{\mu}) + \\ & \Sigma(\Sigma+\Lambda_p)^{-1} \sum_{i=1}^{n_p} \beta_{p_i} (\boldsymbol{x}_{p_i} - \boldsymbol{\mu}) \frac{\partial q_{p_i}}{\partial \Sigma}, \end{aligned} \tag{C.31}$$

where $\partial(\Sigma+\Lambda)^{-1}/\partial\Sigma$ and $\partial(\Sigma+\Lambda_p)^{-1}/\partial\Sigma$ are defined in Eq. (C.18) and Eq. (C.19), respectively.

This concludes the derivation of the partial derivatives needed to implement PILCO with a prior mean function that is an RBF network.

Appendix D

Derivatives for Continuous State-Action Reverse Transfer

This appendix derives the derivatives needed to use the moment matching method from [25] for doing Gaussian process (GP) predictions when using uncertain inputs. In this case, the GP predictions are combinations of the predictions of two GP's, one from real-world data (rw) and one from simulated data (sim). In the equations below superscript letters encased in parentheses indicate indices of a vector or matrix.

First, since the proportion to which the real data is valued is a function of both the input mean and the covariance (see Equation 5.6 and note that $\Sigma_{*_{(rw)}}$ depends on $\boldsymbol{\mu}$ and Σ), the derivatives of $p_{(rw)}$ with respect to $\boldsymbol{\mu}$ and Σ are needed.

The derivative of the generalized logistic function from Equation 5.5 with respect to the input is

$$\frac{\mathrm{d}\,\mathrm{f}(x)}{\mathrm{d}\,x} = -B\,\mathrm{e}^{B(x-x_0)}\left(Q\,\mathrm{e}^{B(x-x_0)}+1\right)^{\left(-\frac{1}{Q}+1\right)}. \tag{D.1}$$

Defining a helper variable

$$\gamma = \frac{\|\Sigma_{*_{(rw)}}\|_F}{\|\left[\sigma_{n_1}^2,\ldots,\sigma_{n_E}^2\right]\|},$$

the chain rule of differentiation is applied to get

$$\begin{aligned}\frac{\partial p_{(rw)}}{\partial \boldsymbol{\mu}} &= \frac{\mathrm{d}\,\mathrm{f}\,(\gamma)}{\mathrm{d}\,\gamma} \times \frac{\mathrm{d}\,\gamma}{\mathrm{d}\|\Sigma_{*_{(rw)}}\|_F} \times \frac{\mathrm{d}\|\Sigma_{*_{(rw)}}\|_F}{\mathrm{d}\,\Sigma_{*_{(rw)}}} \times \frac{\partial \Sigma_{*_{(rw)}}}{\partial \boldsymbol{\mu}} \\ &= \underbrace{\frac{\mathrm{d}\,\mathrm{f}\,(\gamma)}{\mathrm{d}\,\gamma}}_{1\times 1} \times \underbrace{\frac{1}{\|\left[\sigma_{n_1}^2, \ldots, \sigma_{n_E}^2\right]\|}}_{1\times 1} \times \underbrace{\frac{\Sigma_{*_{(rw)}}}{\|\Sigma_{*_{(rw)}}\|_F}}_{E\times E} \times \underbrace{\frac{\partial \Sigma_{*_{(rw)}}}{\partial \boldsymbol{\mu}}}_{E\times E\times D}.\end{aligned}$$

Since $p_{(rw)}$ is a scalar, $\frac{\partial p_{(rw)}}{\partial \boldsymbol{\mu}}$ is a $1 \times D$ vector, which is computed element-wise as

$$\frac{\partial p_{(rw)}}{\partial \boldsymbol{\mu}^{(i)}} = \frac{\mathrm{d}\,\mathrm{f}\,(\gamma)}{\mathrm{d}\,\gamma} \times \frac{1}{\|\left[\sigma_{n_1}^2, \ldots, \sigma_{n_E}^2\right]\|} \times \frac{1}{\|\Sigma_{*_{(rw)}}\|_F} \times \sum_{k=1}^{E}\sum_{l=1}^{E} \Sigma_{*_{(rw)}}^{(k,l)} \frac{\partial \Sigma_{*_{(rw)}}^{(k,l)}}{\partial \boldsymbol{\mu}^{(i)}}. \tag{D.2}$$

Again using the chain rule, the partial derivative of $p_{(rw)}$ with respect to the input covariance is computed as

$$\frac{\partial p_{(rw)}}{\partial \Sigma} = \underbrace{\frac{\mathrm{d}\,\mathrm{f}\,(\gamma)}{\mathrm{d}\,\gamma}}_{1\times 1} \times \underbrace{\frac{1}{\|\left[\sigma_{n_1}^2, \ldots, \sigma_{n_E}^2\right]\|}}_{1\times 1} \times \underbrace{\frac{\Sigma_{*_{(rw)}}}{\|\Sigma_{*_{(rw)}}\|_F}}_{E\times E} \times \underbrace{\frac{\partial \Sigma_{*_{(rw)}}}{\partial \Sigma}}_{E\times E\times D\times D},$$

where the $D \times D$ output is calculated element-wise as

$$\frac{\partial p_{(rw)}}{\partial \Sigma^{(i,j)}} = \frac{\mathrm{d}\,\mathrm{f}\,(\gamma)}{\mathrm{d}\,\gamma} \times \frac{1}{\|\left[\sigma_{n_1}^2, \ldots, \sigma_{n_E}^2\right]\|} \times \frac{1}{\|\Sigma_{*_{(rw)}}\|_F} \times \sum_{k=1}^{E}\sum_{l=1}^{E} \Sigma_{*_{(rw)}}^{(k,l)} \frac{\partial \Sigma_{*_{(rw)}}^{(k,l)}}{\partial \Sigma^{(i,j)}}. \tag{D.3}$$

Given the derivatives of $p_{(rw)}$, the derivatives of the prediction mean with respect to the input mean can now be calculated as

$$\frac{\partial \mu_*}{\partial \boldsymbol{\mu}} = \frac{\partial \mu_{*_{(sim)}}}{\partial \boldsymbol{\mu}} + p_{(rw)}\left(\frac{\partial \mu_{*_{(rw)}}}{\partial \boldsymbol{\mu}} - \frac{\partial \mu_{*_{(sim)}}}{\partial \boldsymbol{\mu}}\right) + \left(\mu_{*_{(rw)}} - \mu_{*_{(sim)}}\right)\left(\frac{\partial p_{(rw)}}{\partial \boldsymbol{\mu}}\right)^T, \tag{D.4}$$

and the prediction covariance with respect to the input covariance as

$$\frac{\partial \mu_*^{(i)}}{\partial \Sigma} = \frac{\partial \mu_{*_{(sim)}}^{(i)}}{\partial \Sigma} + p_{(rw)}\left(\frac{\partial \mu_{*_{(rw)}}^{(i)}}{\partial \Sigma} - \frac{\partial \mu_{*_{(sim)}}^{(i)}}{\partial \Sigma}\right) + \left(\mu_{*_{(rw)}}^{(i)} - \mu_{*_{(sim)}}^{(i)}\right)\frac{\partial p_{(rw)}}{\partial \Sigma}. \tag{D.5}$$

Similarly, the derivative of the input-output covariance with respect to the input

mean is calculated element-wise as

$$\frac{\partial \Sigma_{\boldsymbol{x}_*, f_*}}{\partial \boldsymbol{\mu}^{(i)}} = \frac{\partial \Sigma_{\boldsymbol{x}_*, f_{*(sim)}}}{\partial \boldsymbol{\mu}^{(i)}} + p_{(rw)} \left(\frac{\partial \Sigma_{\boldsymbol{x}_*, f_{*(rw)}}}{\partial \boldsymbol{\mu}^{(i)}} - \frac{\partial \Sigma_{\boldsymbol{x}_*, f_{*(sim)}}}{\partial \boldsymbol{\mu}^{(i)}} \right) + \left(\Sigma_{\boldsymbol{x}_*, f_{*(rw)}} - \Sigma_{\boldsymbol{x}_*, f_{*(sim)}} \right) \frac{\partial p_{(rw)}}{\partial \boldsymbol{\mu}^{(i)}}, \tag{D.6}$$

and with respect to the input covariance as

$$\frac{\partial \Sigma^{(i,j)}_{\boldsymbol{x}_*, f_*}}{\partial \Sigma} = \frac{\partial \Sigma^{(i,j)}_{\boldsymbol{x}_*, f_{*(sim)}}}{\partial \Sigma} + p_{(rw)} \left(\frac{\partial \Sigma^{(i,j)}_{\boldsymbol{x}_*, f_{*(rw)}}}{\partial \Sigma} - \frac{\partial \Sigma^{(i,j)}_{\boldsymbol{x}_*, f_{*(sim)}}}{\partial \Sigma} \right) + \left(\Sigma^{(i,j)}_{\boldsymbol{x}_*, f_{*(rw)}} - \Sigma^{(i,j)}_{\boldsymbol{x}_*, f_{*(sim)}} \right) \frac{\partial p_{(rw)}}{\partial \Sigma}. \tag{D.7}$$

To calculate the partial derivatives of the predictive covariance, some helper variables are first defined as

$$\begin{aligned}
\Phi^{(i,j)}_{(sim)} &= \left(\mu^{(i)}_{*(sim)} - \mu^{(i)}_* \right) \left(\frac{\partial \mu^{(j)}_{*(sim)}}{\partial \boldsymbol{\mu}} - \frac{\partial \mu^{(j)}_*}{\partial \boldsymbol{\mu}} \right) + \\
&\quad \left(\mu^{(j)}_{*(sim)} - \mu^{(j)}_* \right) \left(\frac{\partial \mu^{(i)}_{*(sim)}}{\partial \boldsymbol{\mu}} - \frac{\partial \mu^{(i)}_*}{\partial \boldsymbol{\mu}} \right) + \frac{\partial \Sigma^{(i,j)}_{*(sim)}}{\partial \boldsymbol{\mu}} \\
\Phi^{(i,j)}_{(rw)} &= \left(\mu^{(i)}_{*(rw)} - \mu^{(i)}_* \right) \left(\frac{\partial \mu^{(j)}_{*(rw)}}{\partial \boldsymbol{\mu}} - \frac{\partial \mu^{(j)}_*}{\partial \boldsymbol{\mu}} \right) + \\
&\quad \left(\mu^{(j)}_{*(rw)} - \mu^{(j)}_* \right) \left(\frac{\partial \mu^{(i)}_{*(rw)}}{\partial \boldsymbol{\mu}} - \frac{\partial \mu^{(i)}_*}{\partial \boldsymbol{\mu}} \right) + \frac{\partial \Sigma^{(i,j)}_{*(rw)}}{\partial \boldsymbol{\mu}}.
\end{aligned}$$

Using $\Phi_{(sim)}$ and $\Phi_{(rw)}$, the partial derivative of the predictive covariance with respect to the input mean are calculated as

$$\frac{\partial \Sigma^{(i,j)}_*}{\partial \boldsymbol{\mu}} = \Phi^{(i,j)}_{(sim)} + p_{(rw)} \left(\Phi^{(i,j)}_{(rw)} - \Phi^{(i,j)}_{(sim)} \right) + \left(\beta^{(i,j)}_{(rw)} - \beta^{(i,j)}_{(sim)} \right) \left(\frac{\partial p_{(rw)}}{\partial \boldsymbol{\mu}} \right)^T. \tag{D.8}$$

Defining similar helper variables for the covariance

$$\Psi_{(sim)}^{(i,j)} = \left(\mu_{*(sim)}^{(i)} - \mu_{*}^{(i)}\right)\left(\frac{\partial \mu_{*(sim)}^{(j)}}{\partial \Sigma} - \frac{\partial \mu_{*}^{(j)}}{\partial \Sigma}\right) + \left(\mu_{*(sim)}^{(j)} - \mu_{*}^{(j)}\right)\left(\frac{\partial \mu_{*(sim)}^{(i)}}{\partial \Sigma} - \frac{\partial \mu_{*}^{(i)}}{\partial \Sigma}\right) + \frac{\partial \Sigma_{*(sim)}^{(i,j)}}{\partial \Sigma}$$

$$\Psi_{(rw)}^{(i,j)} = \left(\mu_{*(rw)}^{(i)} - \mu_{*}^{(i)}\right)\left(\frac{\partial \mu_{*(rw)}^{(j)}}{\partial \Sigma} - \frac{\partial \mu_{*}^{(j)}}{\partial \Sigma}\right) + \left(\mu_{*(rw)}^{(j)} - \mu_{*}^{(j)}\right)\left(\frac{\partial \mu_{*(rw)}^{(i)}}{\partial \Sigma} - \frac{\partial \mu_{*}^{(i)}}{\partial \Sigma}\right) + \frac{\partial \Sigma_{*(rw)}^{(i,j)}}{\partial \Sigma},$$

the partial derivative of the predicted covariance with respect to the input covariance are

$$\frac{\partial \Sigma_{*}^{(i,j)}}{\partial \Sigma} = \Psi_{(sim)}^{(i,j)} + p_{(rw)}\left(\Psi_{(rw)}^{(i,j)} - \Psi_{(sim)}^{(i,j)}\right) + \left(\beta_{(rw)}^{(i,j)} - \beta_{(sim)}^{(i,j)}\right)\frac{\partial p_{(rw)}}{\partial \Sigma}. \tag{D.9}$$

In summary, Equations D.1-D.9, together with the partial derivatives in [25] and Appendix C define the partial derivatives needed to implement the moment matching algorithm for approximating the output of the combined real-world and simulation GP from Chapter 5.

Bibliography

[1] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[2] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems (NIPS)*, page 2007. MIT Press, 2007.

[3] Pieter Abbeel, Morgan Quigley, and Andrew Y. Ng. Using inaccurate models in reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 1–8, 2006.

[4] Natalia M Alexandrov, Robert Michael Lewis, Clyde R Gumbert, Lawrence L Green, and Perry A Newman. Approximation and model management in aerodynamic optimization with variable-fidelity models. *Journal of Aircraft*, 38(6):1093–1101, 2001.

[5] Chae H An, Christopher G Atkeson, and John M Hollerbach. *Model-based control of a robot manipulator*, volume 214. MIT press Cambridge, MA, 1988.

[6] Suguru Arimoto, Sadao Kawamura, and Fumio Miyazaki. Bettering operation of robots by learning. *Journal of Robotic systems*, 1(2):123–140, 1984.

[7] Christopher G Atkeson and Stefan Schaal. Robot learning from demonstration. In *International Conference on Machine Learning (ICML)*, pages 12–20, 1997.

[8] M. Gheshlaghi Azar, E. Brunskill, and A. Lazaric. Sequential transfer in multi-armed bandits with finite set of models. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2220–2228, 2013.

[9] J. Andrew Bagnell and Jeff Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2001.

[10] Bram Bakker, Viktor Zhumatiy, Gabriel Gruener, and Jürgen Schmidhuber. A robot that reinforcement-learns to identify and memorize important previous observations. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, pages 430–435. IEEE, 2003.

[11] Bram Bakker, Viktor Zhumatiy, Gabriel Gruener, and Jürgen Schmidhuber. Quasi-online reinforcement learning for robots. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 2997–3002. IEEE, 2006.

[12] Stephen Balakirsky and Elena Messina. A simulation framework for evaluating mobile robots. In *Proceedings of the Performance Metrics for Intelligent Systems Workshop*, pages 1–8, 2002.

[13] A Barto and RH Crites. Improving elevator performance using reinforcement learning. *Advances in neural information processing systems*, 8:1017–1023, 1996.

[14] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[15] B Bischoff, D Nguyen-Tuong, H van Hoof, A McHutchon, CE Rasmussen, A Knoll, J Peters, and MP Deisenroth. Policy search for learning robot control using sparse data. In *IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, June 2014. IEEE.

[16] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

[17] Ronen Brafman and Moshe Tennenholtz. R-Max - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *Journal of Machine Learning Research (JMLR)*, 3:213–231, 2002.

[18] Douglas A Bristow, Marina Tharayil, and Andrew G Alleyne. A survey of iterative learning control. *Control Systems, IEEE*, 26(3):96–114, 2006.

[19] E. Brunskill, B. Leffler, L. Li, M. Littman, and N. Roy. Provably efficient learning with typed parametric models. *Journal of Machine Learning Research (JMLR)*, 10:1955–1988, 2009.

[20] E. Brunskill and L. Li. Sample complexity of multi-task reinforcement learning. In *International Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 122–131, 2013.

[21] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

[22] Mark Cutler, Thomas J. Walsh, and Jonathan P. How. Reinforcement learning with multi-fidelity simulators. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3888–3895, Hong Kong, 2014.

[23] Mark Cutler, Thomas J. Walsh, and Jonathan P. How. Real-world reinforcement learning via multifidelity simulators. *IEEE Transactions on Robotics*, 31(3):655–671, June 2015.

[24] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5:142–150, 1989.

[25] M Deisenroth, Dieter Fox, and C Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PP(99), 2014.

[26] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 465–472, 2011.

[27] Marc Peter Deisenroth. *Efficient reinforcement learning using gaussian processes*, volume 9. KIT Scientific Publishing, 2010.

[28] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1–2):1–142, 2013.

[29] Marc Peter Deisenroth, Carl Edward Rasmussen, and Dieter Fox. Learning to control a low-cost manipulator using data-efficient reinforcement learning. In *Robotics: Science and Systems (RSS)*, Los Angeles, California, June 2011.

[30] Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 240–247, 2008.

[31] Yong Duan, Qiang Liu, and XinHe Xu. Application of reinforcement learning in robot soccer. *Engineering Applications of Artificial Intelligence*, 20(7):936–950, 2007.

[32] Andrew H Fagg, David Lotspeich, Joel Hoff, and George A Bekey. Rapid reinforcement learning for reactive control policy design in autonomous robots. In *Proceedings of the 1994 World Congress on Neural Networks*, volume 2, pages 118–126, 1994.

[33] Fernando Fernández, Javier García, and Manuela Veloso. Probabilistic policy reuse for inter-task transfer learning. *Robotics and Autonomous Systems*, 58(7):866–871, 2010.

[34] Borja Fernández-Gauna, José Manuel López-Guede, and Manuel Graña. Transfer learning with partially constrained models: Application to reinforcement learning of linked multicomponent robot system control. *Robotics and Autonomous Systems*, 61(7):694–703, 2013.

[35] Alexander IJ Forrester, András Sóbester, and Andy J Keane. Multi-fidelity optimization via surrogate modelling. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 463, pages 3251–3269. The Royal Society, 2007.

[36] Christopher T Freeman, Paul L Lewin, Eric Rogers, and James D Ratcliffe. Iterative learning control applied to a gantry robot and conveyor system. *Transactions of the Institute of Measurement and Control*, 32(3):251–264, 2010.

[37] Evangelia Gazi, Warren D Seider, and Lyle H Ungar. A non-parametric monte carlo technique for controller verification. *Automatica*, 33(5):901–906, 1997.

[38] A. Girard, C. Rasmussen, J. Qui nonero Candela, and R. Murray-Smith. Gaussian process priors with uncertain inputs — application to multiple-step ahead time series forecasting. In *Advances in Neural Information Processing Systems 15 (NIPS)*, 2005.

[39] Robert C. Grande, Thomas J. Walsh, and Jonathan P. How. Sample efficient reinforcement learning with Gaussian processes. In *International Conference on Machine Learning (ICML)*, pages 1332–1340, June 2014.

[40] Sehoon Ha and Katsu Yamane. Reducing hardware experiments for model learning and policy optimization. In *IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, Washington, Mar 2015.

[41] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[42] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13 – 30, 1963.

[43] Gabriel M Hoffmann, Claire J Tomlin, Michael Montemerlo, and Sebastian Thrun. Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing. In *American Control Conference (ACC)*, pages 2296–2301, 2007.

[44] Philip J Holmes, John L Lumley, Gal Berkooz, Jonathan C Mattingly, and Ralf W Wittenberg. Low-dimensional models of coherent structures in turbulence. *Physics Reports*, 287(4):337–384, 1997.

[45] Winfried Ilg, Jan Albiez, H Jedele, Karsten Berns, and Rüdiger Dillmann. Adaptive periodic movement control for the four legged walking machine bisam. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 3, pages 2354–2359. IEEE, 1999.

[46] Ashok Mohan Jadhav and K Vadirajacharya. Performance verification of pid controller in an interconnected power system using particle swarm optimization. *Energy Procedia*, 14:2075–2080, 2012.

[47] Nick Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior*, 6(2):325–368, 1998.

[48] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[49] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[50] Michael J. Kearns and Satinder P. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232, 2002.

[51] J. Ko, D.J. Klein, D. Fox, and D. Haehnel. Gaussian processes and reinforcement learning for identification and control of an autonomous blimp. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 742–747. IEEE, 2007.

[52] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[53] Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203, 2011.

[54] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[55] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2619–2624, 2004.

[56] J. Zico Kolter and Andrew Y. Ng. Policy search via the signed derivative. In *Robotics: Science and Systems (RSS)*, Seattle, USA, June 2009.

[57] Sylvain Koos, J-B Mouret, and Stéphane Doncieux. The transferability approach: Crossing the reality gap in evolutionary robotics. *Evolutionary Computation, IEEE Transactions on*, 17(1):122–145, 2013.

[58] Andras Gabor Kupcsik, Marc Peter Deisenroth, Jan Peters, and Gerhard Neumann. Data-efficient generalization of robot skills with contextual policy search. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1401–1407, 2013.

[59] Tak Kit Lau and Yun-hui Liu. Stunt driving via policy search. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4699–4704, 2012.

[60] Alessandro Lazaric. *Knowledge Transfer in Reinforcement Learning*. PhD thesis, Politecnico di Milano, 2008.

[61] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Transfer of samples in batch reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 544–551. ACM, 2008.

[62] Sergey Levine and Vladlen Koltun. Variational policy search via trajectory optimization. In *Advances in Neural Information Processing Systems (NIPS)*, pages 207–215, 2013.

[63] Robert Michael Lewis and Stephen G Nash. A multigrid approach to the optimization of systems governed by differential equations. *AIAA paper*, 4890:2000, 2000.

[64] Lihong Li. *A Unifying Framework for Computational Reinforcement Learning Theory*. PhD thesis, Rutgers University, New Brunswick, NJ, 2009.

[65] Lihong Li, Michael L. Littman, Thomas J. Walsh, and Alexander L. Strehl. Knows what it knows: a framework for self-aware learning. *Machine Learning*, 82(3):399–443, 2011.

[66] Odalric-Ambrym Maillard, Timothy A. Mann, and Shie Mannor. "how hard is my mdp?" the distribution-norm to the rescue. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1835–1843, 2014.

[67] Timothy A. Mann and Yoonsuck Choe. Directed exploration in reinforcement learning with transferred knowledge. In *European Workshop on Reinforcement Learning*, pages 59–76, 2012.

[68] David Meger, Juan Camilo Gamboa Higuera, Anqi Xu, and Gregory Dudek. Learning legged swimming gaits from experience. In *IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, WA, 2015. IEEE.

[69] Rico Moeckel, Yura N Perov, Anh The Nguyen, Massimo Vespignani, Stephane Bonardi, Soha Pouya, Alexander Sproewitz, Jesse van den Kieboom, Frederic Wilhelm, and Auke Jan Ijspeert. Gait optimization for roombots modular robots—matching simulation and reality. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3265–3272, 2013.

[70] A. Molina-Cristobal, P. R. Palmer, B. A. Skinner, and G. T. Parks. Multi-fidelity simulation modelling in optimization of a submarine propulsion system. In *Vehicle Power and Propulsion Conference (VPPC)*, pages 1–6, 2010.

[71] Andrew W Moore. Fast, robust adaptive control by learning only forward models. *Advances in neural information processing systems*, pages 571–571, 1993.

[72] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, pages 103–130, 1993.

[73] Fabian L Mueller, Angela P Schoellig, and Raffaello D'Andrea. Iterative learning of feed-forward corrections for high-performance tracking. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3276–3281. IEEE, 2012.

[74] Katharina Mülling, Jens Kober, Oliver Kroemer, and Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.

[75] Bojan Nemec, Minija Tamosiunaite, F Worgotter, and Ales Ude. Task adaptation through exploration and action sequencing. In *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, pages 610–616. IEEE, 2009.

[76] Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Proceedings of International Symposium on Experimental Robotics*, Singapore, June 2004.

[77] Duy Nguyen-Tuong and Jan Peters. Using model knowledge for learning inverse dynamics. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2677–2682. IEEE, 2010.

[78] Hans B Pacejka, Egbert Bakker, and Lars Nyborg. Tyre modelling for use in vehicle dynamics studies. *SAE paper*, 870421, 1987.

[79] Sanghyuk Park, John Deyst, and Jonathan P How. Performance and lyapunov stability of a nonlinear path following guidance method. *Journal of Guidance, Control, and Dynamics*, 30(6):1718–1728, 2007.

[80] Michael A Patterson and Anil V Rao. Gpops-ii: A matlab software for solving multiple-phase optimal control problems using hp-adaptive gaussian quadrature collocation methods and sparse nonlinear programming. *ACM Transactions on Mathematical Software (TOMS)*, 41(1):1, 2014.

[81] Jan Peters, Katharina Mülling, and Yasemin Altun. Relative entropy policy search. In *AAAI*, 2010.

[82] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.

[83] Nestor V Queipo, Raphael T Haftka, Wei Shyy, Tushar Goel, Rajkumar Vaidyanathan, and P Kevin Tucker. Surrogate-based analysis and optimization. *Progress in aerospace sciences*, 41(1):1–28, 2005.

[84] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, 2006.

[85] Karl Johan Åström and Björn Wittenmark. *Adaptive Control*. Addison-Wesley Publishing Company, 1989.

[86] SS Ravindran. A reduced-order approach for optimal control of fluids using proper orthogonal decomposition. *International journal for numerical methods in fluids*, 34(5):425–448, 2000.

[87] Theodore J Rivlin. *An introduction to the approximation of functions*. Courier Dover Publications, 2003.

[88] T. D. Robinson, K. E. Willcox, M. S. Eldred, and R. Haimes. Multifidelity optimization for variable-complexity design. In *AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, pages 1–18, Portsmouth, VA, 2006.

[89] TD Robinson, MS Eldred, KE Willcox, and R Haimes. Surrogate-based optimization using multifidelity models with variable parameterization and corrected space mapping. *Aiaa Journal*, 46(11):2814–2822, 2008.

[90] Erik J. Schlicht, Ritchie Lee, David H. Wolpert, Mykel J. Kochenderfer, and Brendan Tracey. Predicting the behavior of interacting humans by fusing data from multiple sources. In *International Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 756–764, 2012.

[91] Jeff G Schneider. Exploiting model uncertainty estimates for safe dynamic control learning. *Advances in neural information processing systems*, pages 1047–1053, 1997.

[92] Angela P Schoellig, Fabian L Mueller, and Raffaello D'Andrea. Optimization-based iterative learning for precise quadrocopter trajectory tracking. *Autonomous Robots*, 33(1-2):103–127, 2012.

[93] O Selfridge and RS Sutton. Training and tracking in robotics. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 670–672, 1985.

[94] Aaron Staranowicz and Gian Luca Mariottini. A survey and comparison of commercial and open-source robotic simulator software. In *Proceedings of the 4th International Conference on Pervasive Technologies Related to Assistive Environments*, pages 56:1–56:8, 2011.

[95] Alexander L. Strehl, Lihong Li, and Michael L. Littman. Reinforcement learning in finite mdps: Pac analysis. *Journal of Machine Learning Research (JMLR)*, 10:2413–2444, 2009.

[96] Alexander L. Strehl and Michael L. Littman. Online linear regression and its application to model-based reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1417–1424, 2007.

[97] R. Sutton and A. Barto. *Reinforcement Learning, an Introduction*. MIT Press, Cambridge, MA, 1998.

[98] Richard Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning*, pages 216–224, 1990.

[99] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Neural Information Processing Systems (NIPS)*, pages 1038–1044, 1996.

[100] Mikhail M Svinin, Kazuaki Yamada, and Kanji Ueda. Emergent synthesis of motion patterns for locomotion robots. *Artificial Intelligence in Engineering*, 15(4):353–363, 2001.

[101] Istvan Szita and András Lörincz. The many faces of optimism: a unifying approach. In *ICML*, pages 1048–1055, 2008.

[102] Istvan Szita and Csaba Szepesvári. Model-based reinforcement learning with nearly tight exploration complexity bounds. In *International Conference on Machine Learning (ICML)*, pages 1031–1038, 2010.

[103] Davide Tavernini, Matteo Massaro, Efstathios Velenis, Diomidis I Katzourakis, and Roberto Lot. Minimum time cornering: the effect of road surface and car transmission layout. *Vehicle System Dynamics*, 51(10):1533–1547, 2013.

[104] Matthew E. Taylor. *Transfer in Reinforcement Learning Domains*, volume 216 of *Studies in Computational Intelligence*. Springer, 2009.

[105] Matthew E Taylor and Peter Stone. Representation transfer for reinforcement learning. In *AAAI 2007 Fall Symposium on Computational Approaches to Representation Change during Learning and Development*, 2007.

[106] Matthew E. Taylor, Peter Stone, and Yaxin Liu. Transfer learning via intertask mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(1):2125–2167, 2007.

[107] G. Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, 1995.

[108] Roy M Turner and Elise H Turner. Simulating an autonomous oceanographic sampling network: A multi-fidelity approach to simulating systems of systems. In *OCEANS 2000 MTS/IEEE Conference and Exhibition*, volume 2, pages 905–911. IEEE, 2000.

[109] Jur Van Den Berg, Stephen Miller, Daniel Duckworth, Humphrey Hu, Andrew Wan, Xiao-Yu Fu, Ken Goldberg, and Pieter Abbeel. Superhuman performance of surgical tasks by robots using iterative learning from human-guided demonstrations. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2074–2081. IEEE, 2010.

[110] Efstathios Velenis, Emilio Frazzoli, and Panagiotis Tsiotras. Steady-state cornering equilibria and stabilisation for a vehicle during extreme operating conditions. *International Journal of Vehicle Autonomous Systems*, 8(2):217–241, 2010.

[111] Efstathios Velenis, Diomidis Katzourakis, Emilio Frazzoli, Panagiotis Tsiotras, and Riender Happee. Steady-state drifting stabilization of RWD vehicles. *Control Engineering Practice*, 19(11):1363 – 1376, 2011.

[112] Gerhard Venter, Raphael T Haftka, and James H Starnes. Construction of response surface approximations for design optimization. *AIAA journal*, 36(12):2242–2249, 1998.

[113] Felipe A. Viana, Valder Steffen, Jr., Sergio Butkewitsch, and Marcus Freitas Leal. Optimization of aircraft structural components by using nature-inspired algorithms and multi-fidelity approximations. *Journal of Global Optimization*, 45(3):427–449, 2009.

[114] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[115] Elly Winner and Manuela M. Veloso. Multi-fidelity robotic behaviors: Acting with variable state information. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 872–877, 2000.

[116] Juan Cristóbal Zagal and Javier Ruiz-Del-Solar. Combining simulation and reality in evolutionary robotics. *Journal of Intelligent and Robotic Systems*, 50(1):19–39, 2007.