Learning Novel Strategies for Model Predictive Control by Leveraging Experience

Jacob Isaac Sacks

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

Doctor of Philosophy

University of Washington

2023

Reading Committee:

Byron Boots, Chair

Dieter Fox

Matthew D. Golub

Karen Leung

Program Authorized to Offer Degree:
Computer Science and Engineering

©Copyright 2023
Jacob Isaac Sacks

University of Washington

Abstract

Learning Novel Strategies for
Model Predictive Control by Leveraging Experience

Jacob Isaac Sacks

Chair of the Supervisory Committee:
Byron Boots
Computer Science and Engineering

A major challenge in robotics is to design robust policies which enable complex and agile behaviors in the real world. On one end of the spectrum, we have model-free reinforcement learning (MFRL), which is incredibly flexible and general but often results in brittle policies. In contrast, model predictive control (MPC) continually re-plans at each time step to remain robust to perturbations and model inaccuracies. However, despite its real-world successes, MPC often under-performs the optimal strategy. This is due to model quality, myopic behavior from short planning horizons, and approximations due to computational constraints. And even with a perfect model and enough compute, MPC can get stuck in bad local optima, depending heavily on the quality of the optimization algorithm. Prior research on improving the performance of MPC has primarily focused on learning or fine-tuning a good dynamics model. However, these methods often train the models via system identification, which optimizes likelihood of the data under the model, rather than directly for the controller performance. Moreover, little work has attempted to improve the machinery of the optimization process.

In this thesis, we reinterpret MPC as a structured policy class which can be directly optimized to overcome these core problems facing model-based control and enable robust real-world decision making for robotics. First, we will present our work on developing an approach for learning the dynamics model and cost function of a gradient-based MPC

controller end-to-end by optimizing for task performance without unrolling the iterative solver. Next, we will reinterpret MPC from the perspective of online learning and propose a general family of MPC algorithms rooted in dynamic mirror descent, which include many established gradient- and sampling-based techniques as special cases. To overcome the sample inefficiency of popular sampling-based MPC methods, we then propose to learn a more efficient update rule for solving the online optimization problem via imitation learning. Following this work, we relax the Gaussian assumptions many sampling-based MPC algorithms make and show how to learn more expressive proposal distributions with generative models in order to more effectively search the space of plans. Finally, we show how to learn the update rule and warm-starting procedure of an MPC controller simultaneously via reinforcement learning and demonstrate its performance benefits over hand-designed MPC controllers and end-to-end policies trained via MFRL on a real quadrotor agile trajectory tracking task.

ACKNOWLEDGMENTS

This dissertation would not have been possible if it were not for the constant research and moral support I received throughout the process.

I would like to thank my advisor, Byron Boots, for his guidance and support during the Ph.D. program. I joined his lab at Georgia Tech with very little exposure to robotics and machine learning. Despite my inexperience, Byron gave me a chance and helped me grow into the researcher I am today. I especially appreciate the freedom he gave me in pursuing my interests and finding my own direction.

I would like to thank Matt Golub, Dieter Fox, and Karen Leung for serving on my thesis committee. In particular, I'd like to thank Matt for his willingness to spend time advising and mentoring me on research unrelated to this thesis. During a time when I was feeling especially lost, he gave me a chance to explore my interests in computational neuroscience despite having little background on the subject. This work helped rejuvenate my passion for research and reaffirmed my desire for an academic career. Additionally, he helped me significantly strengthen my defense presentation and dissertation through his detailed feedback.

I would like to thank the members of the Robot Learning Lab at both Georgia Tech and the University of Washington, who made such wonderful labmates throughout the years. I learned so much from everyone through our lab meetings, tutorial sessions, and random conversations in the lab. I have so many great memories from our lab socials, random hang outs, and fun adventures at conferences. I especially want to thank Mohak, Sandesh, and Sid for always listening to me vent when times were tough, giving me both technical and emotional support, and becoming some of my closest friends.

I would like to thank my family and friends for being such an incredible source of strength

and support throughout my Ph.D. My parents, Ruth and Michael, and siblings, Adam and Rachel, were always just a phone call away whenever I needed support or professional advice. When times got tough during the pandemic, I was able to move back home and work remotely from Austin. During this time, I went through one of the hardest times in my life, but they were there to help me through it. I would also like to thank my grandparents, Alan, Audree, Jerry, and Flora, aunts and uncles, Lynn, Milt, Liz, and Robert, and cousins, Aaron, Benji, Dan, and Josh, for all the love they've shown me throughout my life. I can't forget our family cat, Travis, who brightens my day with his unreasonable cuteness. Finally, I'd like to thank my friends in Seattle and Austin for giving me so many good memories and always being there for me. I especially want to thank Willie for being one of my closest friends since college, and Divya and Balaji for being like family away from home in both Atlanta and Seattle. I truly attribute much of where I am today to this constant love and support.

This dissertation represents the end of a long path of growth for me, and also the beginning of the next phase in my life. While I did not mention everyone here, I hold appreciation for all who helped me become the person I am today.

DEDICATION

To my loving family and dear friends.

TABLE OF CONTENTS

Page

List of Figures . . . iii

Chapter 1: Introduction . . . 1
- 1.1 Main Contributions . . . 3

Chapter 2: Sequential Decision Making under Uncertainty . . . 7
- 2.1 Reinforcement Learning . . . 7
- 2.2 Model Predictive Control . . . 9

Chapter 3: Differentiable MPC for End-to-End Planning and Control . . . 16
- 3.1 Differentiable Linear Quadratic Regulator . . . 17
- 3.2 Differentiable Model Predictive Control . . . 19
- 3.3 Experimental Results . . . 22
- 3.4 Discussion . . . 27

Chapter 4: An Online Learning Approach to Model Predictive Control . . . 28
- 4.1 An Online Learning Perspective on MPC . . . 28
- 4.2 A Family of MPC Algorithms Based on Dynamic Mirror Descent . . . 30
- 4.3 Experimental Results . . . 40
- 4.4 Discussion . . . 50

Chapter 5: Learning to Optimize in Model Predictive Control . . . 53
- 5.1 Learning to Optimize for Control . . . 54
- 5.2 Experiments . . . 59
- 5.3 Results . . . 62
- 5.4 Discussion . . . 65

Chapter 6: Learning Sampling Distributions for Model Predictive Control 67
6.1 Representation of the Learned Distribution 68
6.2 Formulating the Learning Problem 71
6.3 Parameterizing with Normalizing Flows 72
6.4 Training the Sampling Distributions 74
6.5 Experiments 78
6.6 Results 84
6.7 Discussion 100

Chapter 7: Deep Model Predictive Optimization 102
7.1 Problem Formulation 103
7.2 Algorithm Overview 105
7.3 Experiments 107
7.4 Discussion 115

Chapter 8: Conclusion 116

LIST OF FIGURES

Figure Number | Page

2.1 A depiction of the canonical structure of an MDP. 8

2.2 A simple example of the shift operator Φ. Here, the control distribution $\boldsymbol{\pi_\theta}$ consists of a sequence of $H = 5$ independent Gaussian distributions. The shift operator moves the parameters of the Gaussians one time step forward and replaces the parameters at $h = 4$ with some default parameters. 10

3.1 Runtime comparison of fixed point differentiation (FP) to unrolling the iLQR solver (Unroll), averaged over 10 trials. 23

3.2 Model and imitation losses for the LQR imitation learning experiments. . . 23

3.3 Learning results on the Pendulum and Cartpole environments. 25

3.4 Convergence results in the non-realizable pendulum task. 26

4.1 Diagram of the online learning perspective, where blue and red denote the learner and the environment, respectively. 29

4.2 Varying step size and number of samples for the Cartpole with (a) continuous and (b) discrete controls, with expected cost (EC, Equation (4.4)), probability of low cost (PLC, Equation (4.7)), and exponential utility (EU, Equation (4.9)). 41

4.3 Varying the loss parameters and step size with a fixed # of samples on Cartpole. 42

4.4 Rally car (right) and real-world AutoRally task (left). 43

4.5 Simulated AutoRally performance for different step sizes and # samples. . . 44

4.6 Simulated AutoRally task. 46

4.7 Car speeds when optimizing the exponential utility. The speeds and trajectories are very similar at step size 0.5, irrespective of the number of samples. At step size 1, though, 64 samples result in capricious maneuvers and low speeds, whereas 3840 samples result in smooth driving at high speeds. 47

4.8 Car speeds when optimizing the expected cost. All tested step sizes result in low speeds. At too low or too high of a step size, the car will drive along the wall or crash into it. 48

4.9 Car speeds with 1920 samples per gradient estimate and target of 9 m/s. . . 50

4.10 Car speeds with 64 samples per gradient estimate and target of 9 m/s. . . . 51

4.11 Car speeds with 64 samples per gradient estimate and target of 11 m/s. In Figure 4.11a, note the crash and U-turn at the top of the plot as well as the wider spread of the paths throughout the whole track. By contrast, in Figure 4.11b, the resulting paths are more consistent, and there are no failure points. 51

5.1 Example trajectories of the Franka Obstacles task, in which the Franka arm end effector (green) is tasked with reaching the goal (red) while avoiding obstacles. The top row is a novel environment with three obstacles, while the bottom row is a test environment. 63

5.2 Trajectories of the Franka arm end effector when controlled by **MPPI** with 512 samples (blue), **MPPI** with 16 samples (pink), and **L2O-MPC** with 16 samples (purple) to move from the starting position (red) to the goal (green) while avoiding obstacles (cyan). Plots in the same row are from the same environment but viewed from differing perspectives. 64

6.1 Computational graph of an episode which illustrates the interaction between the normalizing flow (NF), learned latent shift model (Shift), the MPPI update (MPPI) of the latent mean, the simulator (Sim), and the environment (Env). 77

6.2 Success rate and cost distribution on the PNGrid task across a different number of samples. 85

6.3 Visualization of trajectories in the PNGrid task across multiple random seeds for a fixed environmental layout. 86

6.4 Visualization of a trajectory and top samples from (top) `NFMPC`, (middle) `FlowMPPI`, and (bottom) `MPPI` on the PNGrid task. 87

6.5 Success rate and cost distribution on the PNRand environment across a different number of samples. 88

6.6 Visualization of a trajectory and top samples from (top) `NFMPC`, (middle) `FlowMPPI`, and (bottom) `MPPI` on the PNRand task. 89

6.7 Comparison of unconditional and conditional models on the PNRand environment across a different number of samples. 90

6.8 Success rate and cost distribution on the PNRandDyn environment across a different number of samples. 91

6.9 Visualization of a trajectory and top samples from (top) `NFMPC`, (middle) `FlowMPPI`, and (bottom) `MPPI` on the PNRandDyn task. 92

6.10 Comparison of unconditional and conditional models on the PNRandDyn environment across a different number of samples when trained in an environment with no obstacles (PNRand) and retrained with obstacles present. 93

6.11 Success rate and cost distribution on the FRANKA environment across a different number of samples. 94
6.12 Success rate and cost distribution on the FRANKAOBSTACLES environment across a different number of samples. 95
6.13 Visualization of a trajectory and top samples from (top) NFMPC, (middle) FlowMPPI, and (bottom) MPPI on the FRANKAOBSTACLES task. 96
6.14 Comparison of an unconditional model trained with and without obstacles to a conditional model in the FRANKAOBSTACLES environment across a different # of samples. 96
6.15 Breakdown of different cost terms for each controller on FRANKAOBSTACLES. 97
6.16 Success rate and cost distribution on the FRANKA task with (NFMPC) and without (NFMPC (No Shift) and **NFMPC (No Shift, Retrained**) the learned shift model. 98
7.1 The DMPO architecture consists of two learnable modules, the shift model and optimizer. The fixed rollout model module performs rollouts of the sampled control sequences. 105
7.2 Position tracking error of DMPO versus MPPI and E2E on tracking random infeasible zig-zag trajectories without any environmental disturbances. 110
7.3 Picture of the quadrotor with the attached plate and additional wind field. . 111
7.4 Position tracking error of DMPO versus MPPI and E2E on tracking random infeasible zig-zag trajectories with an attached plate and wind. 112
7.5 Example zig-zag trajectory with a 180° yaw flip performed by MPPI (8192 samples) and DMPO (512 samples). 112
7.6 Total cost (top), position error in meters (middle), and orientation error (bottom) of DMPO versus MPPI on tracking random yaw flip trajectories without any environmental disturbances (left) and with an attached plate and wind (right). 113

Chapter 1
INTRODUCTION

Generating complex and agile behaviors on real world robotic systems is fundamentally a problem of sequential decision making under uncertainty. Model-free reinforcement learning (MFRL) is a general approach to such problems that makes minimal assumptions and has been successfully deployed in the real world [1–3]. However, control policies trained with these methods are often brittle and do not generalize to out-of-distribution disturbances. For instance, consider an uninhabited aerial vehicle (UAV) following an aggressive trajectory in uncertain environments [4, 5]. If the UAV encounters unknown wind gusts that were not experienced in training, the policy will likely not be able to account for the change in dynamics and lead to a crash. Furthermore, due to the sample inefficiency of MFRL methods, they often train policies in simulation. While high-fidelity simulators are becoming more accessible, the true parameters of the system may be unknown. And there are often hard-to-model components (e.g. motor characteristics, contact, granular media) or disturbances (e.g. wind) which are simplified. This can create a sim-to-real gap due to the mismatch between the simulator and the true system. Even if the policies succeed in simulation, they often fail in the real world. We can partially remedy this issue using domain randomization (DR) [6–8], which samples simulator parameters and disturbances from a pre-specified distribution during training. While DR can improve the robustness of the learned policy, its efficacy is dependent on the distributions we select, as well as the nature of the problem we wish to solve.

Alternatively, model predictive control (MPC) [9–11] has emerged as a powerful paradigm for sequential decision making on real-world robotic systems. MPC has been successfully applied to helicopter aerobatics [12], robot manipulation [13–15], humanoid robot locomotion [16], robot-assisted dressing [17], and aggressive off-road driving [10, 18, 19]. The underlying

principle in MPC is that designing a complicated control algorithm that is globally optimal at every state is difficult and the system may be hard to model well. Instead, it iteratively finds a simple controller online that is good enough locally at each time step. Starting from an initial open-loop sequence over a short planning horizon, MPC leverages an approximate system model to refine the plan by solving an optimization problem online. At each time step, it takes the first control in the optimized plan and applies it to the system. It repeats the process from the resulting next state, creating an effective feedback controller. Although the planned sequence is often open-loop, because we are updating it using the current state, MPC effectively yields a state-feedback policy. Therefore, rather than design a single, globally-optimal policy, MPC opts for finding simple locally optimal policies at each state. By re-solving this optimization problem online, we can improve the robustness of MPC to perturbations and model inaccuracies. However, this also leads to increased computational demands compared to MFRL. In practice, we approximate the solution at each time step to run in real time. This involves warm-starting with the solution from the previous time step, which works well when the two problems are similar [19]. But if our system encounters a large perturbation, this warm-starting procedure can bias us towards a poor solution [20]. The performance of MPC also depends on the model quality and prediction horizon length [11, 21–23]. And even with a perfect model and enough computation, the optimization algorithm may get stuck in bad local optima due to the non-convexity of the cost landscape [24]. Altogether, these issues often lead to MPC under-performing the optimal policy.

To circumvent these limitations, there has been a long history of combining machine learning with MPC. This has primarily focused on learning or fine-tuning a good dynamics model [10, 17, 25–29], potentially from high-dimensional observations [14, 30–35]. However, these methods which learn approximate models from data often optimize the likelihood of the data under the model. This is in contrast to the objective we ultimately care about, which is the controller performance when leveraging the model for planning. This is known as the objective mismatch problem [36], and it can lead to situations where the controller exploits inaccuracies in the model. Other work targets the cost function, learning terminal

value functions [37–41] which can help mitigate myopic behavior due to a short planning horizon. In a similar vein, we can use cost-shaping terms [22] to improve performance with reduced prediction horizons. While these methods improve performance by modifying the objective function, they do not address limitations due to the optimization process. Learning terminal value functions can enable us to reduce the planning horizon and contend with short-sighted decisions. But we may still get stuck in local optima or require many policy updates to achieve good performance, increasing computational demands.

These prior works view the MPC optimization process as a fixed component of the pipeline, focusing on the model or cost function. Alternatively, we can treat the optimizer as another component of the MPC controller which can be improved via machine learning. And when learning any parameterized component of the MPC controller, we should optimize directly for performance on the desired task. To this end, the overarching research statement of this thesis is the following:

MPC defines a structured policy class which can be directly optimized for performance to overcome its limitations and enable robust real-world decision making for robotics.

Specifically, we will propose a method for learning the dynamics model and cost function jointly by optimizing the same objective to circumvent the objective mismatch problem. Then we will derive and learn more efficient optimization strategies for MPC that can help avoid poor local minima and improve controller performance with fewer computational resources.

1.1 Main Contributions

The primary research contributions of this thesis are:

1. **Differentiable MPC for End-to-End Planning and Control:** We present the foundations for using MPC as a differentiable policy class for reinforcement learning (RL) in continuous state and action spaces. Specifically, we differentiate through the KKT conditions of the convex approximation at a fixed point of the controller. Using this strategy, we are able to learn the cost and dynamics of a controller via end-to-end

learning, directly optimizing for controller performance. Our experiments focus on imitation learning in simulation, where we learn the cost and dynamics terms of an MPC policy class. We show that our MPC policies are significantly more data-efficient than a generic neural network. Moreover, our method is superior to traditional system identification, especially in a setting where the expert is unrealizable. Because we learn the model in conjunction with the controller, this approach successfully circumvents the objective mismatch problem. And these results illustrate the benefits of learning components of an MPC controller directly via task performance.

2. **An Online Learning Approach to MPC:** We show that there exists a close connection between MPC and online learning, an abstract theoretical framework for analyzing online decision making in the optimization literature. This new perspective provides a foundation for leveraging powerful online learning algorithms to design MPC algorithms. Specifically, we propose a new algorithm based on dynamic mirror descent (DMD), an online learning algorithm that is designed for non-stationary setups. Our algorithm, Dynamic Mirror Descent Model Predictive Control (`DMD-MPC`), represents a general family of MPC algorithms that includes many existing techniques as special instances. `DMD-MPC` also provides a fresh perspective on previous heuristics used in MPC and suggests a principled way to design new MPC algorithms. We demonstrate the flexibility of `DMD-MPC` by presenting a set of new MPC algorithms on a simple simulated cartpole and a simulated and real-world aggressive driving task. Our results indicate the extra design flexibility offered by `DMD-MPC` does make a difference in practice; by properly selecting hyperparameters which are obscured in the previous approaches, we are able to improve the performance of existing algorithms.

3. **Learning to Optimize in MPC:** Rather than hand-design the optimization algorithm used in MPC, we propose to *learn* how to optimize more effectively. In other words, rather than treat the update rule in MPC as a fixed component, we aim to directly improve it through learning. We show that this can be particularly useful

in sampling-based MPC, where we often wish to minimize the number of samples for computational reasons. Unfortunately, the cost of computational efficiency is a reduction in performance; fewer samples results in noisier updates. We show that we can contend with this noise by learning how to update the control distribution more effectively and make better use of the few samples that we have. Our learned controllers are trained via imitation learning to mimic an expert which has access to substantially more samples. We test the efficacy of our approach on multiple simulated robotics tasks in sample-constrained regimes and demonstrate that our approach can outperform a MPC controller with the same number of samples. These results demonstrate that learning the update rule is a viable alternative to hand-designing the MPC algorithms, especially under computational constraints.

4. **Learning Sampling Distributions for MPC:** We build upon recent efforts to incorporate more complex sampling distributions for MPC by leveraging the power of deep generative models to learn the sampling distribution. Specifically, we learn expressive sampling distributions for MPC which have simple latent spaces and move all online parameter updates and warm-start operations into this latent action space. We accomplish this by framing the learning problem as bi-level optimization and derive an approximate gradient through the MPC update of latent distribution to train the network directly for controller performance. Additionally, we show how to parameterize the sampling distribution such that we can incorporate box constraints on the controls Finally, we empirically evaluate our approach on simulated navigation and manipulation tasks. We demonstrate its ability to improve performance over baselines by taking full advantage of the learned latent space. We find that the performance of the controllers with our learned sampling distributions scales more gracefully with a reduction in the number of samples. This demonstrates that learning the sampling distribution is another powerful strategy for improving the optimization process in MPC. Moreover, directly optimizing them for task performance results in improved sample efficiency.

5. **Deep Model Predictive Optimization:** We propose Deep Model Predictive Optimization (`DMPO`), which learns the inner-loop of an MPC optimization algorithm directly via experience, specifically tailored to the needs of the control problem. In contrast to prior work, which learned the optimizer via imitation learning, we show how to train all components of the MPC optimization pipeline via RL by viewing MPC as a structured recurrent policy class. On a real quadrotor platform tracking infeasible zig-zag trajectories, we show that `DMPO` can outperform an end-to-end policy trained with MFRL by 19%. Tracking zig-zag trajectories with alternating 180° flips in the desired yaw, `DMPO` can improve error over a baseline MPC algorithm by up to 27% with $16\times$ fewer samples, saving $4.3\times$ memory requirements. And by exposing the quadrotor to turbulent wind fields with an attached cardboard drag plate, we show that `DMPO` can adapt zero-shot, matching the performance of the MPC baseline and outperforming the end-to-end policy by 14%. These results indicate the `DMPO` can improve performance over the best hand-designed MPC controller while retaining its robustness to out-of-distribution disturbances.

Chapter 2
SEQUENTIAL DECISION MAKING UNDER UNCERTAINTY

Many problems in robotics can be modeled in the general framework of sequential decision making under uncertainty. In this chapter, we present preliminaries on Markov decision processes (MDPs) and reinforcement learning (RL). We then delve into details on both gradient-based and sampling-based formulations of MPC.

2.1 Reinforcement Learning

At the core of RL is the formalism of the Markov decision process (MDP). This framework consists of 1) an agent, which makes decisions according to a policy π, and 2) the environment, which is provides feedback to the agent, guiding the evolution of its policy via experience. We illustrate this canonical structure in Figure 2.1, where the environment defines a discrete-time stochastic dynamical system. The agent receives the current state of the environment, x_t, and chooses a control input u_t according to a policy π. The agent then applies the control, and the environment state updates accordingly. This updated state, x_{t+1}, is again sent to the agent. The agent also receives a reward, which measures the quality of having applied control u_t in state x_t and ending up in state x_{t+1}. These rewards can be used to update the policy and improve its performance on the task.

Formally, we consider an infinite-horizon discounted MDP, which is defined by the tuple $\mathcal{M} = (\mathcal{X}, \mathcal{U}, P, r, \rho_0, \gamma)$, where $\mathcal{X}$ is the state space, $\mathcal{U}$ is the control space, $x_{t+1} \sim P(\cdot|x_t, u_t)$ is the transition dynamics, $r(x, u)$ is the reward function, $\rho_0(x_0)$ is the initial state distribution, and $\gamma \in (0, 1)$ is the discount factor. Given a closed-loop policy, $u \sim \pi(\cdot|x)$, its value function is defined as

$$V^\pi(x) = \mathbb{E}_{P,\pi}\left[\sum_{t=0}^{\infty} \gamma^t r(x_t, u_t)\middle| x_0 = x\right]. \tag{2.1}$$

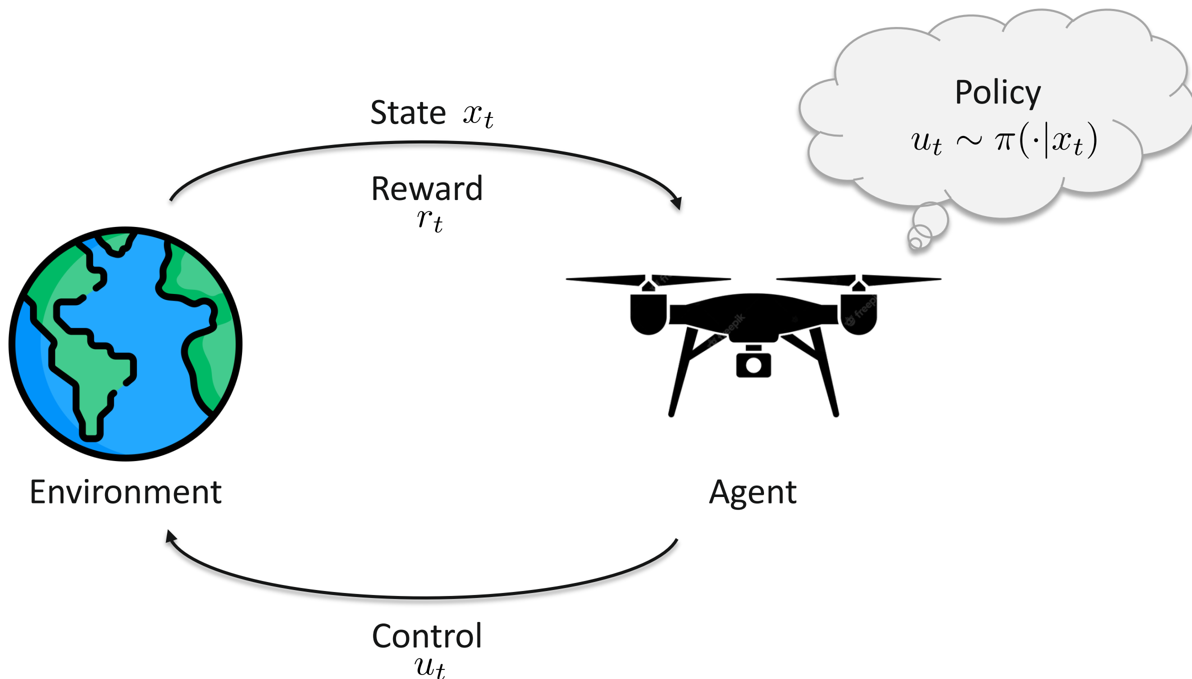


Figure 2.1: An MDP involves an agent interacting with the environment. The agent determines its input u_t according to its policy π given the current state of the environment, x_t. In environment updates its state and returns a scalar reward r_t based on state and control input.

The goal of RL is to find a policy π which maximizes the expected discounted reward, which is equivalent to maximizing the value function:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{x_0 \sim \rho_0} \Big[V^\pi(x_0) \Big]. \tag{2.2}$$

A common approach is to directly find π with policy gradient methods, which perform gradient descent with a zeroth-order approximation of the gradient using a finite number of samples. State-of-the-art approaches include actor-critic algorithms [42, 43], which in addition to learning π (actor), learn an estimate of V_π (critic) and use it as a baseline to reduce the gradient estimator variance. If we wish to train our policy over a range of tasks, we can additionally condition both π and V_π on task parameters, such as a goal state.

2.2 Model Predictive Control

Rather than find a single, globally optimal policy, MPC re-optimizes a local policy at each time step. It accomplishes this by predicting the system's behavior over a finite horizon H using an approximate model $\hat{P}$. Specifically, the goal of MPC is to find a distribution over an open-loop sequence of controls $\hat{\boldsymbol{u}}_t \sim \boldsymbol{\pi}_{\boldsymbol{\theta}_t}(\cdot)$, where $\hat{\boldsymbol{u}}_t \triangleq (\hat{u}_t, \hat{u}_{t+1}, \cdots, \hat{u}_{t+H-1})$ and $\boldsymbol{\theta}_t \in \Theta$, which is some set of feasible parameters. Applying $\hat{\boldsymbol{u}}_t$ to our approximate model starting at x_t, we have a predicted state sequence $\hat{\boldsymbol{x}}_t \triangleq (\hat{x}_t, \hat{x}_{t+1}, \cdots, \hat{x}_{t+H})$, with $\hat{x}_t = x_t$. Rather than rewards, we generally think about costs in MPC, which can simply be the reward with the sign flipped. That is, the want to minimize cost rather than maximize reward. The total cost incurred by this predicted trajectory is

$$C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) = \sum_{h=0}^{H-1} c(\hat{x}_{t+h}, \hat{u}_{t+h}) + c_{term}(\hat{x}_{t+H}), \tag{2.3}$$

where $c(\cdot, \cdot)$ is the instantaneous cost and $c_{term}(\cdot)$ is a terminal cost function. At each time step, we solve the following optimization problem:

$$\boldsymbol{\theta}_t \leftarrow \operatorname*{arg\,min}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} J(\boldsymbol{\pi}_{\boldsymbol{\theta}}; x_t), \tag{2.4}$$

where $J(\cdot)$ is a statistic defined on cost $C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$ such that its minimum occurs at the optimal $\boldsymbol{\theta}_t$. After optimizing our parameters, we sample the control sequence $\hat{\boldsymbol{u}}_t \sim \boldsymbol{\pi}_{\boldsymbol{\theta}_t}(\cdot)$, apply the first control to the real system (i.e. $u_t = \hat{u}_t$), and repeat the process. Because each parameter $\boldsymbol{\theta}_t$ depends on the current state, MPC effectively yields a state-feedback policy, even though the individual distributions give us an open-loop sequence.

This optimization problem can only be approximated in practice due to real-time constraints. One commonly applied heuristic is to bootstrap the previous approximate solution as an initialization for the current problem. This is effective because the optimization problems between two consecutive time steps share all control variables except the first and last. If our solution from the previous problem is $\boldsymbol{\theta}_{t-1}$, then our warm start for the current problem is

$$\tilde{\boldsymbol{\theta}}_t = \Phi(\boldsymbol{\theta}_{t-1}), \tag{2.5}$$

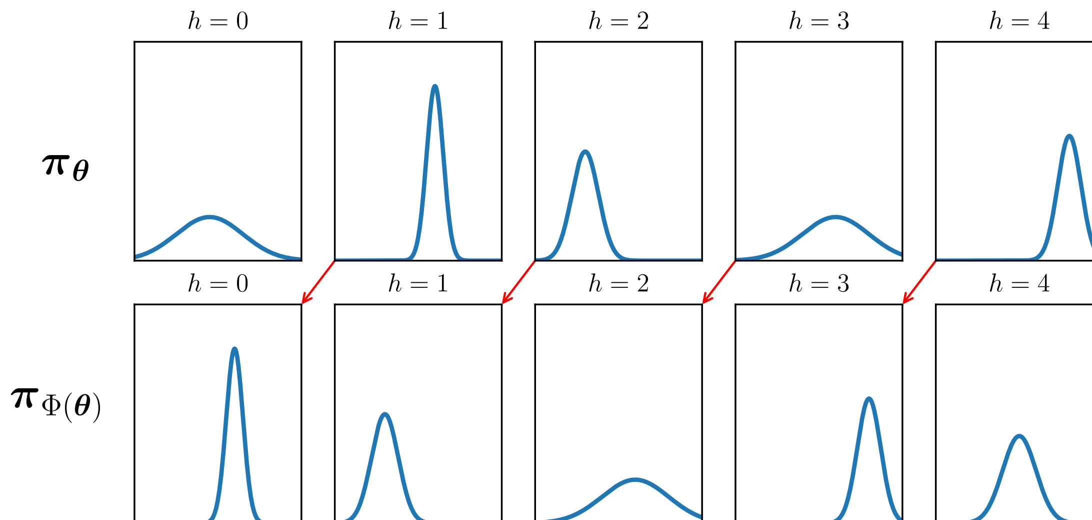


Figure 2.2: A simple example of the shift operator Φ. Here, the control distribution $\boldsymbol{\pi_\theta}$ consists of a sequence of $H = 5$ independent Gaussian distributions. The shift operator moves the parameters of the Gaussians one time step forward and replaces the parameters at $h = 4$ with some default parameters.

where $\Phi(\cdot)$ is called the shift operator. This shift model aims to predict the optimal decision at the next time step given the previous solution. In the context of MPC, it allows us to warm-start the optimization problem to speed up convergence. When there is a separate parameter for each time step in the horizon, the common choice is to simply shift the control sequence forward one time step and append some control at the end. A simple example of this operation is shown in Figure 6.16 Specifically, if $\boldsymbol{\theta}_{t-1} = (\theta_{t-1}, \theta_t, \ldots, \theta_{t+H-2})$, then we would set $\tilde{\boldsymbol{\theta}}_t = (\theta_t, \theta_{t+1}, \ldots, \theta_{t+H-2}, \bar{\theta})$, where $\bar{\theta}$ is often set to a constant, such as zero, randomly sampled, or repeats the last set of parameters along the horizon.

2.2.1 Gradient-Based Model Predictive Control

One option for solving the MPC problem is to apply an iterative, gradient-based optimization algorithm. This choice requires that our approximate dynamics model and the cost function

be differentiable with respect to our policy parameters. Indirect methods, such as shooting algorithms, solve the optimization problem by only searching over the space of controls. State trajectories are represented implicitly as the result of rolling out our dynamics (forward integration) given a control sequence. As such, the dynamics constraints are always satisfied, and the solution is strictly feasible. Alternatively, direct methods explicitly represent the state trajectory as free variables, and the dynamics are enforced as strict equality constraints on the problem. While both have their advantages, shooting methods tend to be faster and are more amenable to warm-starting solutions between iterations. As we mentioned in Section 2.2, warm-starting is critical to solving the problem under real-time constraints.

A well-studied gradient-based method is the linear-quadratic regulator (LQR), which assumes that our cost function is quadratic in the control trajectory and our dynamics model is linear. In the most general case, the dynamics are given by $\hat{x}_{t+1} = F_t \hat{\tau}_t + f_t$, where $\hat{\tau}_t = \{\hat{x}_t, \hat{u}_t\}$ concatenates the state and control, $F_t \in \mathbb{R}^{N \times (N+M)}$ is the dynamics matrix, and $f_t \in \mathbb{R}^N$ is a bias. The instantaneous and terminal costs are given by

$$c(\hat{x}_t, \hat{u}_t) = \frac{1}{2} \hat{\tau}_t^T C_t \hat{\tau}_t + c_t^T \hat{\tau}_t + \text{const.}, \qquad c_{term}(\hat{x}_{t+H}) = \frac{1}{2} \hat{x}_{t+H}^T C_{term} \hat{x}_{t+H}, \tag{2.6}$$

our statistic $\hat{J}(\boldsymbol{\theta}; x_t) = C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$ is simply the total trajectory cost, and our policy is $\hat{\boldsymbol{u}}_t \sim \boldsymbol{\pi}_{\boldsymbol{\theta}_t}(\cdot) = \delta(\hat{\boldsymbol{u}}_t - \theta)$. The optimal control is given by a feedback law of the form $\hat{u}_t^\star = K_t \hat{x}_t + k_t$, where the matrix K_t and vector k_t are computed via a backwards pass called the dynamic Ricatti recursion. The optimal plan is then $\hat{\boldsymbol{u}}_t^\star \triangleq (\hat{u}_t^\star, \hat{u}_{t+1}^\star, \cdots, \hat{u}_{t+H-1}^\star)$, and our policy is $\hat{\boldsymbol{u}}_t \sim \boldsymbol{\pi}_{\boldsymbol{\theta}_t}(\cdot) = \delta(\hat{\boldsymbol{u}}_t - \hat{\boldsymbol{u}}_t^\star)$. And in the infinite-horizon case with time-invariant dynamics and costs, we converge to a fixed point with a closed-form linear policy that can be found by solving a discrete algebraic Ricatti equation.

However, while LQR is a powerful tool, it relies on very restrictive assumptions about the nature of the dynamics and cost function. An extension of LQR which can handle nonlinear dynamics and non-quadratic cost functions is the iterative linear quadratic regulator (iLQR) [44]. Specifically, it iteratively forms a convex approximation of the original non-convex control problem with first- and second-order Taylor approximations of the dynamics and cost

functions, respectively. In the forward pass, we begin with an initial control sequence and rollout the original, nonlinear model with this sequence to get our state trajectory. Next, we linearize the dynamics and take a quadratic approximation of our cost around the resulting state-control trajectory. Using LQR, we use backward recursion to find the optimal controls of our approximated system, called the backward pass. We then recompute our approximation around this new control sequence and repeat until convergence to a local minima. If we take a second-order approximation of our dynamics, then we have the differential dynamic programming (DDP) algorithm [45], which admits quadratic convergence when the dynamics are smooth. However, this comes at the cost of computing Hessians of our dynamics, which can be computationally expensive.

Finally, it is important to note that in practice, we often have control limits, or inequality constraints on the control sequence. One common way to enforce them is to clamp the controls in the forward pass of iLQR or DDP. A downside of this approach is the proposed update to the control sequence may no longer be a valid descent direction. Alternatively, we can incorporate an element-wise squashing function $s(\hat{u}_t)$ in the nonlinear dynamics:

$$\hat{x}_{t+1} = f(\hat{x}_t, s(\hat{u}_t)), \qquad s(\hat{u}_t) = \frac{\bar{u} - \underline{u}}{2}(\tanh(\hat{u}_t) + 1), \tag{2.7}$$

where $\bar{u}$ and $\underline{u}$ are the upper and lower bounds, respectively, and f are the deterministic dynamics of the system. However, this can cause problems with gradient-based methods such as iLQR and DDP, because they ignore higher order terms introduced by this highly nonlinear function. Tassa et al.[45] proposed to solve a series of quadratic programs (QPs) with these box constraints explicitly enforced. This method can be combined with iLQR and introduces a negligible overhead compared to the forward pass of the main algorithm.

2.2.2 Sampling-Based Model Predictive Control

In contrast to gradient-based methods, sampling-based MPC places no restrictions on the differentiability of the dynamics or cost function and is straightforward to parallelize. This flexibility and simplicity has led them to become a cornerstone of contemporary approaches

to MPC. Like iLQR methods, sampling-based MPC rolls out an approximate dynamics model using a control sequence. However, rather than using a single sequence, they sample many possible control sequences from a policy distribution. The resulting state trajectories corresponding to each sample can then be computed in parallel. Finally, they use the resulting state-control trajectories to update the policy.

A popular, practical sampling-based MPC algorithm is Model Predictive Path Integral (MPPI) control [10, 18]. We assume that our policy is a factorized Gaussian of the form

$$\pi_{\boldsymbol{\theta}}(\hat{\boldsymbol{u}}) = \prod_{h=0}^{H-1} \pi_{\theta_h}(\hat{u}_{t+h}) = \prod_{h=0}^{H-1} \mathcal{N}(\hat{u}_{t+h}; \mu_{t+h}, \Sigma_{t+h}). \tag{2.8}$$

That is, each control in the sequence is assumed to be independent, and our policy parameters are the means and covariance matrices of each Gaussian. Additionally, we assume that we have a prior on our controls, $\bar{\pi}_{\tilde{\boldsymbol{\theta}}}(\hat{\boldsymbol{u}})$, which is also a Gaussian of the same form. When this prior is a factorized zero-mean Gaussian with the same covariance matrix as our policy, the system under this prior is uncontrolled and behaves according to what is called its passive dynamics. Another assumption that MPPI makes is that all the noise in the dynamics comes from the controls and that the actual dynamics is deterministic otherwise. With these assumptions, MPPI optimizes the system's free-energy:

$$\hat{J}(\boldsymbol{\theta}; x_t) = -\beta \log \mathbb{E}_{\bar{\boldsymbol{\pi}}_{\tilde{\boldsymbol{\theta}}}}\Big[\exp\Big(-\frac{1}{\beta}S(\hat{\boldsymbol{u}}_t)\Big)\Big], \tag{2.9}$$

where $\beta > 0$ is a scaling parameter, known as the temperature, the expectation is taken over the prior policy. Additionally, we have defined $S(\hat{\boldsymbol{u}}_t) = C(\hat{\boldsymbol{f}}(x_t, \hat{\boldsymbol{u}}_t), \hat{\boldsymbol{u}}_t)$, where $\hat{\boldsymbol{f}}(x_t, \hat{\boldsymbol{u}}_t)$ means iteratively apply our control sequence to the deterministic dynamics, f, starting at state x_t. This last equality states our assumption that the state trajectory is a deterministic function of the sampled controls. Next, we convert the expectation to be over the controlled dynamics with importance sampling:

$$\hat{J}(\boldsymbol{\theta}; x_t) = -\beta \log \mathbb{E}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}\Big[\frac{\bar{\pi}_{\tilde{\boldsymbol{\theta}}}(\hat{\boldsymbol{u}})}{\pi_{\boldsymbol{\theta}}(\hat{\boldsymbol{u}})}\exp\Big(-\frac{1}{\beta}S(\hat{\boldsymbol{u}}_t)\Big)\Big], \tag{2.10}$$

Williams et al. [10] showed that the optimal control distribution under this objective is

$$\boldsymbol{\pi}^*(\hat{\boldsymbol{u}}) = \frac{1}{\eta}\exp\Big(-\frac{1}{\beta}S(\hat{\boldsymbol{u}}_t)\Big)\bar{\boldsymbol{\pi}}_{\tilde{\boldsymbol{\theta}}}(\hat{\boldsymbol{u}}), \tag{2.11}$$

where η is a normalizing constant. However, we cannot directly sample from this distribution. As such, Williams et al. [10] set out to minimize the KL divergence between the optimal policy and one of the form of Equation (2.8). With these assumptions, the MPPI update rule for the policy mean:

$$\mu_{t+h} = \sum_{i=1}^{N} w_i \hat{u}_{t+h}^{(i)}, \quad w_i = \frac{e^{-\frac{1}{\beta}\bar{S}(\hat{\boldsymbol{u}}_t^{(i)})}}{\sum_{j=1}^{N} e^{-\frac{1}{\beta}\bar{S}(\hat{\boldsymbol{u}}_t^{(j)})}}, \quad \bar{S}(\hat{\boldsymbol{u}}_t^{(j)}) = S(\hat{\boldsymbol{u}}_t^{(j)}) - \beta \log\left(\frac{\bar{\boldsymbol{\pi}}_{\tilde{\boldsymbol{\theta}}}(\hat{\boldsymbol{u}}_t^{(j)})}{\boldsymbol{\pi}_{\boldsymbol{\theta}}(\hat{\boldsymbol{u}}_t^{(j)})}\right) \tag{2.12}$$

where the superscript (i) refers to the i-th sample. As such, MPPI update simply takes a weighted sum of the control sequence samples, where the weights depend on the costs of the resulting state-control trajectories. Note that this is actually an iterative procedure, and we can run it for multiple iterations to improve our estimate of the optimal distribution. Between time steps of MPC, we can use the shifted previous sequence of means as a warm-start by making it our prior. To enforce control limits, we can clamp or squash our control samples with a sigmoidal function.

2.2.3 Machine Learning and MPC

There have been many recent attempts at combining machine learning with MPC, which mostly center around learning or fine-tuning a good dynamics model [10, 17, 25–29], potentially from high-dimensional observations [14, 30–35]. MPC from high-dimensional observations often involves learning a latent state space model which makes use of the true control actions. One exception is the work by Ha et al. [34], which learns a latent action space without considering a way to map them to the true action space which could be applied to a real system. Alternatively, some methods target the cost function, learning terminal value functions [37–41] or cost-shaping terms [22] to improve performance with reduced prediction horizons. Instead of learning dynamics and cost separately, other methods propose learning the entire optimal controller [22, 46–51] or planner [52–55] end-to-end. The main benefit of these approaches is that, instead of learning the dynamics on the surrogate objective of maximizing observation likelihoods, we can take a holistic approach that optimizes the entire setup for the main objective of minimizing cost or maximizing reward.

Using alternative sampling distributions is another viable strategy for improving the optimization process in MPC. In particular, there have been a few efforts which have explored learning the sampling distributions or alternative action spaces for MPC. Amos et al. [46] learn a latent action space for their proposed differentiable cross-entropy method (CEM) controller. However, they require all components of the pipeline to be differentiable and do not consider learning a shift model. Instead, they perform multiple iterations of CEM each time step, which can be expensive and impractical for real-time execution. Agarwal et al. [56] learn a normalizing flow in the latent space of a variational autoencoder (VAE) for the purposes of trajectory optimization. Yet, they also require differentiability, use expert demonstrations to learn the latent space of the VAE, and have no means of warm-starting between time steps. Wang et al. [57] propose to use a learned feedback policy to warm-start MPC, but still rely on a Gaussian perturbations to the proposed action sequence. The authors also explore performing online planning in the network's parameter space, which results in a massive action space that may be hard to scale. Power et al. [58, 59] also train a normalizing flow to use as the sampling distribution for MPC, but they do not learn a latent shift model and perform all operations in the control space. Moreover, they mix the latent samples with Gaussian perturbations to the current control-space mean, as they do not update the latent distribution directly. This prevents them from fully taking advantage of the learned distribution and throws away useful information which may improve performance.

Chapter 3

DIFFERENTIABLE MPC FOR END-TO-END PLANNING AND CONTROL

In Section 2.2, we assumed that our model and cost function were known and pre-specified. Traditionally, we either derive a model from the known physics of the system or learn one from data by optimizing data likelihood under the model. The latter option involves optimizing an objective different than controller performance, which leads to the well-known objective mismatch problem [36]. Drawing from our work in Amos et al. [49], in this chapter, we instead consider learning the components of MPC-based policies in an end-to-end fashion. Specifically, we consider the case where we have a parameterized cost function c_ϕ and dynamics model f_ϕ. By differentiating through the optimization problem, we can learn both directly by optimizing for controller performance. Beyond helping mitigate the objective mismatch problem [36], this approach enables us to treat MPC as a structured policy class. This is in contrast to black-box policies, such as neural networks, which can be very sample-inefficient to train.

Prior work simply unrolls the optimization procedure and differentiates through the computational graph [22, 46–48, 50–55]. This is expensive both in memory and computation, as the backward pass grows linearly with the number of solver iterations. Alternatively, Amos et al. [60] showed how to differentiate through generic optimization solvers without unrolling. However, their approach does not scale to the size of MPC problems, which are more efficiently solved with shooting-based methods. Instead, in this work, we show an efficient method for analytically differentiating through an iterative non-convex optimization procedure based upon a box-constrained iLQR solver [45]. Specifically, we show that once the solver has converged to a fixed point, the derivative of the loss with respect to the policy parameters can be computed using one additional backward pass of a modified LQR solver.

This makes the learning process more computationally tractable while still benefiting from structure.

3.1 Differentiable Linear Quadratic Regulator

We can write the LQR problem discussed in Section 2.2.1 as a constrained optimization problem as follows:

$$\hat{\boldsymbol{\tau}}^\star = \arg\min_{\hat{\boldsymbol{\tau}}} \sum_{t=1}^{T} \frac{1}{2}\hat{\tau}_t^T C_t \hat{\tau}_t + c_t^T \hat{\tau}_t, \quad \text{subject to} \quad \hat{x}_1 = x_{init},\ \hat{x}_{t+1} = F_t \hat{\tau}_t + f_t, \tag{3.1}$$

where $\hat{\boldsymbol{\tau}} \triangleq (\hat{\tau}_1, \hat{\tau}_2, \cdots, \hat{\tau}_T)$. Note that we are indexing from 1 and solving the problem for the entire episode length T rather than a finite horizon $H < T$ for notational simplicity. From a policy learning perspective, this can be interpreted as a module with unknown parameters $\phi = \{C_t, c_t, F_t, f_t\}_{t=1}^T$, which can be integrated into a larger end-to-end learning system. This involves computing the derivatives of our optimal solution, $\hat{\boldsymbol{\tau}}^\star$, with respect to the parameters ϕ. Instead of directly computing each gradient, we present an efficient way of computing derivatives of a loss function, ℓ, with resepct to the parameters:

$$\frac{\partial \ell}{\partial \phi} = \frac{\partial \ell}{\partial \hat{\boldsymbol{\tau}}^\star} \frac{\partial \hat{\boldsymbol{\tau}}^\star}{\partial \phi}. \tag{3.2}$$

By interpreting LQR from an optimization perspective, we associate dual variables, $\boldsymbol{\lambda} \triangleq (\lambda_0, \lambda_1, \cdots, \lambda_{T-1})$, with the state constraints. Specifically, λ_0 is associated with the initial state constraint $\hat{x}_1 = x_{init}$, and $\lambda_{1:T-1}$ are associated with the dynamics. The Lagrangian of the optimization problem is then:

$$\mathcal{L}(\hat{\boldsymbol{\tau}}, \boldsymbol{\lambda}) = \sum_{t=1}^{T} \left(\frac{1}{2}\hat{\tau}_t^T C_t \hat{\tau}_t + c_t^T \hat{\tau}_t \right) + \sum_{t=0}^{T-1} \lambda_t^T (F_t \hat{\tau}_t + f_t - \hat{x}_{t+1}), \tag{3.3}$$

where the initial constraint is represented by setting $F_0 = 0$ and $f_0 = x_{init}$. Differentiating Equation (3.3) with respect to $\hat{\boldsymbol{\tau}}^\star$ yields

$$\nabla_{\hat{\tau}_t} \mathcal{L}(\hat{\boldsymbol{\tau}}^\star, \boldsymbol{\lambda}^\star) = C_t \hat{\tau}_t^\star + c_t + F_t^T \lambda_t^\star - \begin{bmatrix} \lambda_{t-1}^\star \\ 0 \end{bmatrix} = 0. \tag{3.4}$$

Therefore, we can think of the normal approach to solving a finite-horizon LQR problem with the dynamic Riccati recursion as an efficient way of solving the following KKT system:

$$\overbrace{\begin{array}{c} \begin{array}{cccc} \hat{\tau}_t & \lambda_t & \hat{\tau}_{t+1} & \lambda_{t+1} \end{array} \\ \left[\begin{array}{c:cc:cc:c} \ddots & & & & & \\ \hdashline & C_t & F_t^\top & & & \\ & F_t & & [-I \quad 0] & & \\ \hdashline & & \begin{bmatrix} -I \\ 0 \end{bmatrix} & C_{t+1} & F_{t+1}^\top & \\ & & & F_{t+1} & & \\ \hdashline & & & & & \ddots \end{array}\right] \end{array}}^{K} \begin{bmatrix} \vdots \\ \hat{\tau}_t^\star \\ \lambda_t^\star \\ \hat{\tau}_{t+1}^\star \\ \lambda_{t+1}^\star \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ c_t \\ f_t \\ c_{t+1} \\ f_{t+1} \\ \vdots \end{bmatrix}. \tag{3.5}$$

Given an optimal nominal trajectory $\hat{\boldsymbol{\tau}}^\star$, Equation (3.4) gives us a way of computing the optimal dual variables $\boldsymbol{\lambda}^\star$ via a backward recursion:

$$\lambda_T^\star = C_{T,x} \hat{\tau}_T^\star + c_{T,x}, \quad \lambda_t^\star = F_{t,x}^T \lambda_{t+1}^\star + C_{t,x} \hat{\tau}_t^\star + c_{t,x}, \tag{3.6}$$

where $C_{t,x}$, $c_{t,x}$, and $F_{t,x}$ are the first block-rows of C_t, c_t, and F_t, respectively, which correspond to the state variable. With the optimal trajectory and dual variables, we can compute the gradients of the loss with respect to the parameters by implicitly differentiating through the KKT conditions. Applying the approach proposed by Amos et al. [60], the derivatives take the following form:

$$\begin{aligned} \nabla_{C_t} \ell &= \frac{1}{2} \left(d_{\hat{\tau}_t}^\star \otimes \hat{\tau}_t^\star + \hat{\tau}_t^\star \otimes d_{\hat{\tau}_t}^\star \right) & \nabla_{c_t} \ell &= d_{\hat{\tau}_t}^\star & \nabla_{x_{\text{init}}} \ell &= d_{\lambda_0}^\star \\ \nabla_{F_t} \ell &= d_{\lambda_{t+1}}^\star \otimes \hat{\tau}_t^\star + \lambda_{t+1}^\star \otimes d_{\hat{\tau}_t}^\star & \nabla_{f_t} \ell &= d_{\lambda_t}^\star & & \end{aligned} \tag{3.7}$$

where $\otimes$ is the outer product, and $d_{\hat{\tau}}^\star$ and $d_{\lambda}^\star$ are obtained by solving the linear system

$$K \begin{bmatrix} \vdots \\ d_{\hat{\tau}_t}^\star \\ d_{\lambda_t}^\star \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \nabla_{\hat{\tau}_t^\star} \ell \\ 0 \\ \vdots \end{bmatrix}. \tag{3.8}$$

Algorithm 1: Differentiable LQR

Input: Initial state x_{init}

Parameters: $\phi = \{C_t, c_t, F_t, f_t\}_{t=1}^T$

Forward Pass:

$\hat{\boldsymbol{\tau}}^\star = \text{LQR}_T(x_{\text{init}}; C_{1:T}, c_{1:T}, F_{1:T}, f_{1:T})$; ▷ Solve Equation (3.1)

Compute $\boldsymbol{\lambda}^\star$ with Equation (3.6)

Backward Pass:

$d^\star_{\hat{\boldsymbol{\tau}}} = \text{LQR}_T(0; C_{1:T}, \nabla_{\hat{\boldsymbol{\tau}}^\star}\ell, F_{1:T}, 0)$; ▷ Solve Equation (3.8)

Compute $d^\star_{\boldsymbol{\lambda}}$ with Equation (3.6)

Compute derivatives of ℓ with respect to ϕ and x_{init} using Equation (3.7)

Now, we can observe that Equation (3.8) is of the same form as Equation (3.5), and therefore, it can be solved efficiently with a backward recursion similar to that used to solve an LQR problem. The main difference is that we replace c_t with $\nabla_{\hat{\tau}_t^\star}\ell$ and f_t with 0. In fact, we can actually reuse much of the computation from our initial LQR solve in this backward pass. In Algorithm 1, we summarize the forward and backward passes for differentiable LQR.

3.2 Differentiable Model Predictive Control

Now that we can compute gradients through a LQR solution, extending our approach to iLQR with box constraints on the controls is relatively straightforward. Given general non-linear dynamics, non-quadratic cost functions, and control limits, the resulting constrained, non-convex MPC problem can be written in the general form as

$$\hat{\boldsymbol{\tau}}^\star = \arg\min_{\hat{\boldsymbol{\tau}}} \sum_{t=1}^T C_{\phi,t}(\hat{\tau}_t), \quad \text{subject to} \quad \hat{x}_1 = x_{init},\ \hat{x}_{t+1} = f_\phi(\hat{\tau}_t),\ \underline{u} \le \hat{u}_t \le \bar{u}, \tag{3.9}$$

where the cost function C_ϕ and dynamics f_ϕ are potentially parameterized by some ϕ and may be different at each time step. An iLQR controller solves Equation (3.9) by iteratively

forming and optimizing a convex approximation

$$\hat{\boldsymbol{\tau}}^{i+1} = \arg\min_{\hat{\boldsymbol{\tau}}} \sum_{t=1}^{T} \tilde{C}^i_{\phi,t}(\hat{\tau}_t), \quad \text{subject to} \;\; \hat{x}_1 = x_{init},\; \hat{x}_{t+1} = \tilde{f}^i_\phi(\hat{\tau}_t),\; \underline{u} \le \hat{u}_t \le \bar{u}, \tag{3.10}$$

where we have defined the second-order Taylor approximation of the cost around $\hat{\boldsymbol{\tau}}^i$ as

$$\tilde{C}^i_{\phi,t} = C_{\phi,t}(\hat{\tau}^i_t) + (p^i_t)^\top(\hat{\tau}_t - \hat{\tau}^i_t) + \frac{1}{2}(\hat{\tau}_t - \hat{\tau}^i_t)^\top H^i_t(\hat{\tau}_t - \hat{\tau}^i_t), \tag{3.11}$$

with $p^i_t = \nabla_{\hat{\tau}^i_t} C_{\phi,t}$ and $H^i_t = \nabla^2_{\hat{\tau}^i_t} C_{\phi,t}$. We also have a first-order Taylor approximation of the dynamics around the previous solution $\hat{\boldsymbol{\tau}}^i$ as

$$\tilde{f}^i_{\phi,t}(\hat{\tau}_t) = f_{\phi,t}(\hat{\tau}^i_t) + F^i_t(\hat{\tau}_t - \hat{\tau}^i_t), \tag{3.12}$$

with $F^i_t = \nabla_{\hat{\tau}^i_t} f_{\phi,t}$. Usually a fixed point of Equation (3.11) is reached, especially when the dynamics are smooth. Therefore, we can differentiate through the original problem in Equation (3.10) using the final convex approximation. Without box constraints, we could simply differentiate through this fixed point with LQR, as shown in Section 3.1. However, extending our approach to handle box constraints requires some additional machinery.

Differentiating Box-Constrained QPs. First, consider a generic box-constrained QP:

$$x^\star = \arg\min_x \;\; \frac{1}{2}x^\top Q x + p^\top x \;\; \text{subject to} \;\; Ax = b,\; \underline{x} \le x \le \overline{x}. \tag{3.13}$$

Given active inequality constraints at the solution in the form of $\tilde{G}x = \tilde{h}$, this turns into an equality-constrained optimization problem, with the solution given by the linear system

$$\begin{bmatrix} Q & A^\top & \tilde{G}^\top \\ A & 0 & 0 \\ \tilde{G} & 0 & 0 \end{bmatrix} \begin{bmatrix} x^\star \\ \lambda^\star \\ \tilde{\nu}^\star \end{bmatrix} = - \begin{bmatrix} p \\ b \\ \tilde{h} \end{bmatrix} \tag{3.14}$$

With some loss function ℓ that depends on $x^\star$, we can use the approach developed by Amos et al. [60] to obtain the derivatives with respect to our optimization problem parameters:

$$\nabla_Q \ell = \frac{1}{2}(d^\star_x \otimes x^\star + x^\star \otimes d^\star_x) \qquad \nabla_p \ell = d^\star_x \qquad \nabla_A \ell = d^\star_\lambda \otimes x^\star + \lambda^\star \otimes d^\star_x \qquad \nabla_b \ell = -d^\star_\lambda \tag{3.15}$$

where $d_x^\star$ and $d_\lambda^\star$ are obtained by solving the linear system

$$\begin{bmatrix} Q & A^\top & \tilde{G}^\top \\ A & 0 & 0 \\ \tilde{G} & 0 & 0 \end{bmatrix} \begin{bmatrix} d_x^\star \\ d_\lambda^\star \\ d_{\tilde{\nu}}^\star \end{bmatrix} = - \begin{bmatrix} \nabla_{x^\star} \ell \\ 0 \\ 0 \end{bmatrix} \tag{3.16}$$

The constraint $\tilde{G} d_x^\star = 0$ is equivalent to the constraint $d_{x_i}^\star = 0$ if $x_i^\star \in \{\underline{x}_i, \overline{x}_i\}$. Thus solving the system in Equation (3.16) is equivalent to solving the optimization problem

$$d_x^\star = \operatorname*{arg\,min}_{d_x} \; \frac{1}{2} d_x^\top Q d_x + (\nabla_{x^\star} \ell)^\top d_x \;\; \text{subject to} \;\; A d_x = 0, \; d_{x_i} = 0 \;\text{ if }\; x_i^\star \in \{\underline{x}_i, \overline{x}_i\} \tag{3.17}$$

Differentiating MPC with Box Constraints. At a fixed point, we can use Equation (3.15) to compute the derivatives of the MPC problem, where $d_{\hat{\tau}}^\star$ and $d_\lambda^\star$ are found by solving the linear system in Equation (3.8), with the additional constraint that $d_{\hat{u}_{t,i}} = 0$ if $\hat{u}_{t,i}^\star \in \{\underline{u}, \bar{u}\}$. Solving this system can be equivalently written as a zero-constrained LQR problem of the form

$$\begin{aligned} d_{\hat{\tau}}^\star = \operatorname*{arg\,min}_{d_{\hat{\tau}}} \; \sum_{t=1}^{T} \frac{1}{2} d_{\hat{\tau}_t}^\top H_t^n d_{\hat{\tau}_t} + (\nabla_{\hat{\tau}_t^\star} \ell)^\top d_{\hat{\tau}_t} \\ \text{subject to} \;\; d_{\hat{x}_1} = 0, \; d_{\hat{x}_{t+1}} = F_t^n d_{\hat{\tau}_t}, \; d_{\hat{u}_{t,i}} = 0 \;\text{ if }\; u_i^\star \in \{\underline{u}, \bar{u}\} \end{aligned} \tag{3.18}$$

where n is the iteration we reach a fixed point. In Algorithm 2, I summarize the forward and backward passes of the differentiable MPC module. To solve Equation (3.9), we use the box-DDP heuristic proposed by Tassa et al. [45]. Meanwhile, for the zero-constrained LQR problem in Equation (3.18), we use an LQR solver that zeros the appropriate controls.

Drawbacks of Our Approach. It may not always be computationally feasible to run the controller long enough to reach a fixed-point. Moreover, a fixed-point may not even exist when using neural networks to approximate the dynamics. In this case, using our approach will usually give the wrong gradients, and it may be better to simply unroll the iterative computation. However, unrolling scales linearly with the number of iLQR iterations used in the forward pass, while the fixed-point differentiation is constant time and only requires a single LQR solve.

Algorithm 2: Differentiable MPC

Input: Initial state x_{init} and control sequence u_{init}

Parameters: Parameters ϕ of the objective $C_\phi(\hat{\tau})$ and dynamics $f_\phi(\hat{\tau})$

Forward Pass:

$\hat{\boldsymbol{\tau}}^\star = \text{MPC}_{T,\underline{u},\bar{u}}(x_{\text{init}}, u_{\text{init}}; C_\phi, f_\phi)$; ▷ Solve Equation (3.9)

Once solver reaches a fixed point, obtain cost and dynamics approximations $\{H_t^n, F_t^n\}_{t=1}^T$

Compute $\boldsymbol{\lambda}^\star$ with Equation (3.6)

Backward Pass:

Set $\tilde{F}_t^n = F_t^n$ with the rows corresponding to the tight control constraints zeroed

$d_{\hat{\boldsymbol{\tau}}}^\star = \text{LQR}_T(0; H_{1:T}^n, \nabla_{\hat{\boldsymbol{\tau}}^\star}\ell, \tilde{F}_{1:T}^n, 0)$; ▷ Solve Equation (3.18)

Compute $d_{\boldsymbol{\lambda}}^\star$ with Equation (3.6)

Compute derivatives of ℓ with respect to H_t^n and F_t^n using Equation (3.7)

Differentiate these approximations with respect to ϕ and use the chain rule to get $\partial\ell/\partial\phi$

3.3 Experimental Results

We present several results that highlight the performance and capabilities of our differentiable MPC module in comparison to neural network polices and vanilla system identification (`SysID`). We demonstrate that differentiating through the fixed point yields superior runtime performance compared to an unrolled solver. Additionally, we demonstrate the ability of our method to recover the cost and dynamics of a controller via imitation learning (IL). Finally, we also show the benefit of directly optimizing the task loss over vanilla `SysID`, which may help address the objective mismatch problem. We have released our differentiable MPC solver as a standalone open source package that is available at `https://github.com/locuslab/mpc.pytorch`, and our experimental code for this paper is also openly available at `https://github.com/locuslab/differentiable-mpc`. We implemented all experiments in PyTorch [61].

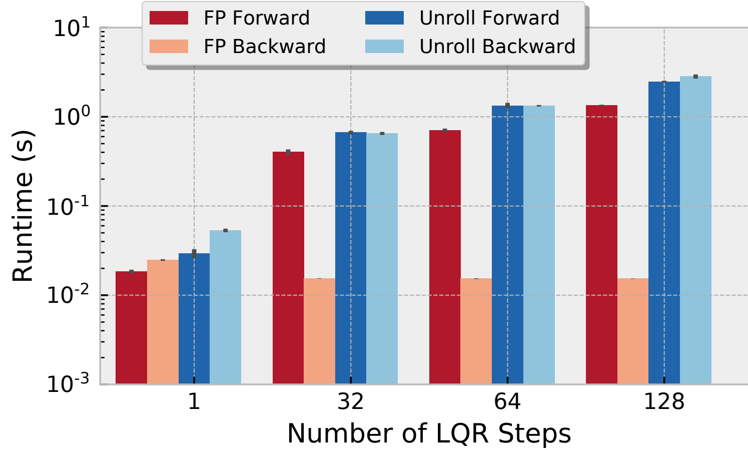


Figure 3.1: Runtime comparison of fixed point differentiation (FP) to unrolling the iLQR solver (Unroll), averaged over 10 trials.

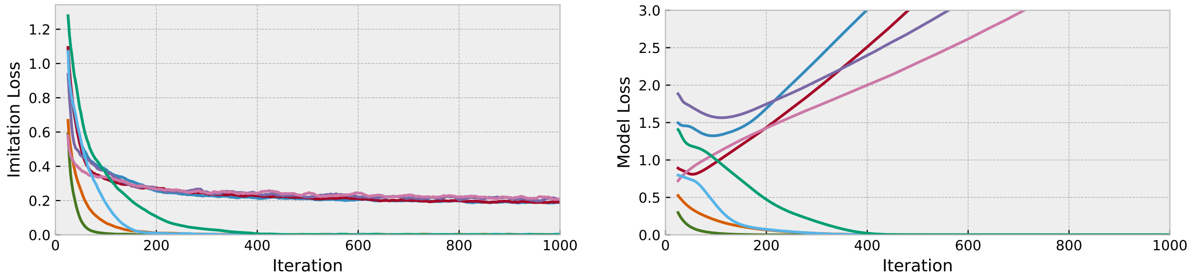


Figure 3.2: Model and imitation losses for the LQR imitation learning experiments.

MPC Solver Performance. Figure 3.1 highlights the performance of our differentiable MPC solver. We compare to the case where we instead unroll the computation and compute the gradients by differentiating through the entire unrolled chain. As expected, the unrolled operations incur a substantial additional cost. Specifically, our method is slightly more computationally and memory efficient in the forward pass, as it does not need to store intermediate computation for the backward pass. More importantly, it is significantly more efficient in the backward pass, especially when a large number of iLQR iterations are required.

The backward pass is essentially free in comparison, as it can reuse much of the computation from the forward pass and does not require multiple iterations.

Imitation Learning: The LQR Case. In our first set of experiments, we validate the MPC solver and gradient-based learning approach via IL in the LQR setting. That is, both the expert and learner are LQR controllers that share all information except the time-invariant linear system dynamics, $f(\hat{x}_t, \hat{u}_t) = A\hat{x}_t + B\hat{u}_t$. They share the same quadratic cost (the identity), control bounds ($\hat{u}_t \in [-1, 1]$), horizon (5 timesteps), and 3-dimensional state and control spaces. Given an initial state x, we obtain nominal actions from the controllers as $\hat{\boldsymbol{u}}(x;\phi)$, where $\phi = \{A, B\}$. We randomly initialize the learner's dynamics and minimize $\mathcal{L}(\phi) = \mathbb{E}_x\left[||\hat{\boldsymbol{\tau}}(x;\phi^\star) - \hat{\boldsymbol{\tau}}(x;\phi)||_2^2\right]$, which is the imitation loss, and $\phi^\star$ are the parameters of the expert. Using mini-batches of 32 examples, we differentiate $\mathcal{L}$ with respect to ϕ and take gradient steps with RMSprop [62]. In Figure 3.2, we show the model and imitation loss of eight randomly sampled initial dynamics, where the **model loss** is the mean squared error between the learner and expert parameters, or $\mathrm{MSE}(\phi, \phi^\star)$. For half of the trials, the model does converge to the true parameters and achieves a perfect imitation loss. Other trials get stuck in a local minimum and cause the approximate model to significantly diverge from the true model. This highlights that while LQR is convex, the learning problem is a (potentially difficult) non-convex optimization problem that does not typically have convergence guarantees.

Imitation Learning: Non-Convex Continuous Control. Next, we use IL to train the learner to solve the pendulum and cartpole benchmark tasks. The expert is an MPC controller that produces a nominal action sequence $\hat{\boldsymbol{u}}(x;\phi^\star)$. The goal is to optimize

$$\mathcal{L}(\phi) = \mathbb{E}_x\left[||\hat{\boldsymbol{u}}(x;\phi^\star) - \hat{\boldsymbol{u}}(x;\phi)||_2^2\right], \tag{3.19}$$

on *only* the observed controls and *no* state observations. We consider three variants of a MPC learner: 1) known cost and only learn the dynamics (*mpc.dx*), 2) known dynamics and

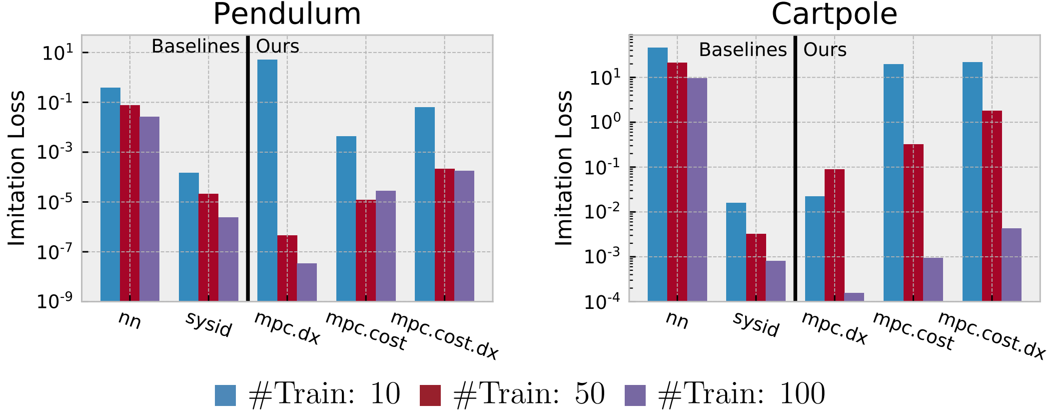


Figure 3.3: Learning results on the Pendulum and Cartpole environments.

only learn the cost (*mpc.cost*), and 3) approximate both the cost and dynamics (*mpc.cost.dx*). As a baseline, we compare to LSTMs (*nn*), which takes in the state x as input and predicts the nominal action sequence. We also consider the case where we know the cost function and simply approximate the dynamics by optimizing the next-state transitions (*sysid*).

In all settings that involve learning the dynamics (*sysid*, *mpc.dx*, and *mpc.cost.dx*), we use a parameterized version of the true dynamics. For the pendulum, the parameters consist of the mass, length, and gravity. Similarly, for the cartpole, the parameters are the cart's mass, pole's mass and length, and gravity. For cost learning, we parameterize the cost of the controller as the weighted distance to the goal state, $C(\hat{\tau}) = ||w_g \circ (\hat{\tau} - \tau_g)||_2^2$, where w_g are the weights and τ_g is the goal state. We found that simultaneously learning both terms is unstable, and instead, we alternate learning w_g and τ_g independently every 10 epochs. We collected a dataset of trajectories from our expert controller and varied the number of trajectories used to train the model. A single trial of our experiments takes 1-2 hours on a modern CPU. We optimize *nn* with Adam [63] using a learning rate of 10^{-4} and everything else with RMSprop [62] using a learning rate of 10^{-2} and decay of 0.5.

We show our results in Figure 3.3, which illustrates that in nearly every case, we are able to directly optimize imitation loss and outperform a generic neural network policy trained on the same information. In many cases, we are able to recover the true cost function and

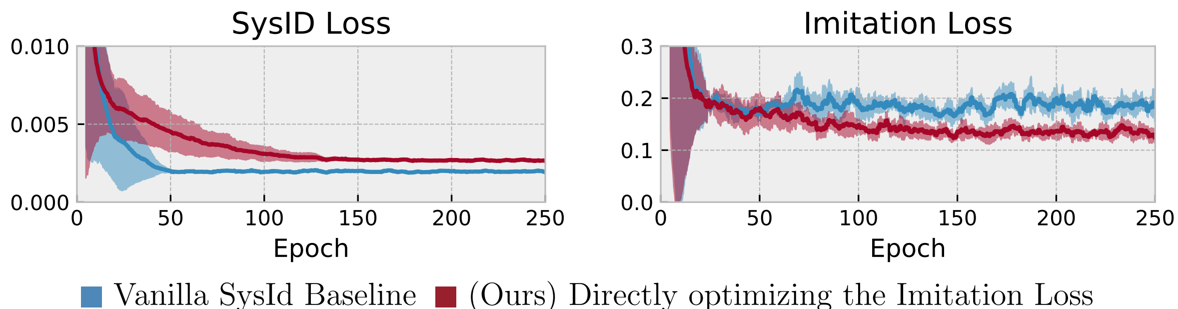


Figure 3.4: Convergence results in the non-realizable pendulum task.

dynamics of the expert. A notable result is that we are able to recover equivalent performance to vanilla system identification (`SysID`) using *only* the control information and *without* state information. While these are simple tasks, there are stark differences between the approaches. Due to the structure in the MPC policy, it is able to learn with much lower sample complexity than a typical network. But unlike pure `SysID`, the differentiable MPC policy can naturally be adapted to objectives besides state prediction, such as incorporating the additional cost learning portion.

Imitation Learning: SysID with a Non-Realizable Expert. The prior experiments are unrealistic in the sense that the expert's dynamics are in the model class being learned. As such, we next study a case where the expert's dynamics are *outside* of the model class being learned. Specifically, we explore adding an additional damping term to the Pendulum, as well as another force on the point-mass at the end (analogous to a "wind" force). However, the dynamics models we learn *do not* have these additional terms, and therefore, they *cannot* recover the expert's parameters. In Figure 3.4, we show that even though vanilla `SysID` is slightly better at optimizing the next-state transitions, it finds an inferior model for imitation compared to our approach. This highlights the affect of the objective mismatch problem and how our approach can potentially help mitigate it.

3.4 Discussion

This work lays the foundations for differentiating and learning MPC-based controllers within RL and IL. Our approach, in contrast to the more traditional strategy of "unrolling" a policy, has the benefit that it is much less computationally and memory intensive. The backward pass is essentially free given the number of iterations required for an iLQR optimizer to converge to a fixed point. We demonstrated that our approach, in the context of IL, has potential advantages over learning standard end-to-end policies and performing system identification. In particular, we argue that the goal of doing system identification is rarely in isolation and always serves the purpose of performing a more sophisticated task, such as imitation or policy learning. Typically, system identification is merely a surrogate for optimizing the task. We claim that the task's loss signal provides useful information to guide the dynamics learning. Our method provides one way of doing this, by allowing the task's loss function to be directly differentiated with respect to the dynamics function.

Chapter 4

AN ONLINE LEARNING APPROACH TO MODEL PREDICTIVE CONTROL

In Chapter 3, we focused on jointly learning the model and cost function of an MPC controller, viewing it as a differentiable policy class. However, we left the optimizer a fixed component, providing useful structure to the learned policy. But backbone of our method, iLQR, is a traditional technique for solving the MPC problem. In this chapter, we build on our work in Wagener et al. [19] and develop a unified framework for MPC which provides a means for hand-designing entirely new MPC algorithms. In particular, we reinterpret MPC from the perspective of *online learning* [64] and show that a wide variety of MPC algorithms actually follow the same general update rule. In view of this connection, we propose a generic framework, `DMD-MPC` (Dynamic Mirror Descent Model Predictive Control), for synthesizing MPC algorithms. `DMD-MPC` is based on a first-order online learning algorithm called dynamic mirror descent (DMD) [65]. We show that several existing MPC algorithms are specifical cases of `DMD-MPC`, given specific choices of step sizes, loss functions, and regaulrization. Furthermore, we demonstrate how new MPC algorithms can be derived systematically from `DMD-MPC` with only mild assumptions on the regularity of the cost function. This allows use to even work with discontinuous cost functions (like indicators) and discrete controls. Thus, `DMD-MPC` offers a spectrum from which practitioners can easily customize new algorithms for their algorithms.

4.1 An Online Learning Perspective on MPC

Online learning is an abstract theoretical framework which involves interactions between a learner and an environment over T rounds (Figure 4.1). At round t, the learner makes a decision $\tilde{\boldsymbol{\theta}}_t \in \boldsymbol{\Theta}$. The environment then chooses a loss function ℓ_t according to the learner's

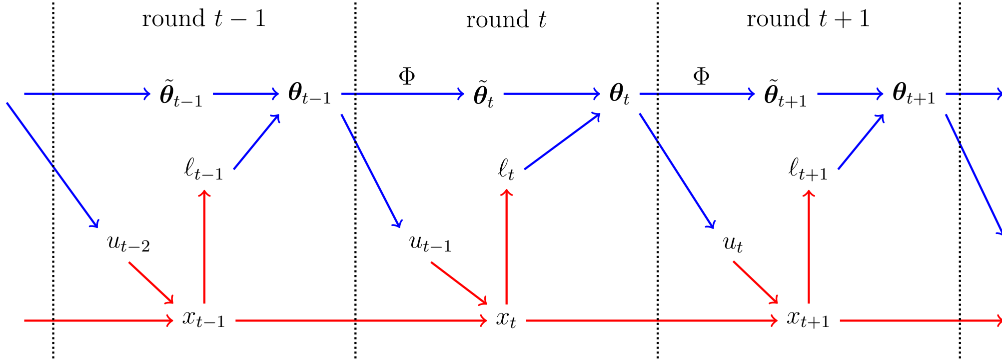


Figure 4.1: Diagram of the online learning perspective, where blue and red denote the learner and the environment, respectively.

decision, and the learner incurs a cost $\ell_t(\tilde{\boldsymbol{\theta}}_t)$. The learner chooses its next decision based on this cost and additional information, such as the gradient of the loss evaluated at $\tilde{\boldsymbol{\theta}}_t$. For non-stationary problems, the learner minimizes the accumulated costs $\sum_{t=1}^{T} \ell_t(\tilde{\boldsymbol{\theta}})$ by minimizing the dynamic regret [65], which is defined as

$$\text{D-Regret} = \sum_{t=1}^{N} \ell_t(\tilde{\boldsymbol{\theta}}_t) - \sum_{t=1}^{T} \ell_t(\boldsymbol{\theta}_t^\star), \tag{4.1}$$

where $\boldsymbol{\theta}_t^* \in \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \ell_t(\boldsymbol{\theta})$. Dynamic regret quantifies how sub-optimal the decisions $\tilde{\boldsymbol{\theta}}_1, \ldots, \tilde{\boldsymbol{\theta}}_T$ are given the corresponding loss functions. With respect to MPC, each round of online learning corresponds to a time step of our control system. The learner is the MPC algorithm, which in round t plays the decision $\tilde{\boldsymbol{\theta}}_t \in \boldsymbol{\Theta}$, the shifted policy parameter sequence, along with side information u_{t-1}, the control applied to the real system. The per-round loss is defined as $\ell_t(\cdot) = \hat{J}(\cdot; x_t)$, which is selected by the environment via the transition to state x_t after applying control u_{t-1}. Note that having small regret in this setup does not imply good absolute performance on the control system. Instead, it means that the plan produced by MPC after the shift operation remains close to optimal with respect to the new loss function ℓ_t. If the approximate dynamics model $\hat{f}$ is accurate, we can expect that bootstrapping via warm-starting would result in a small instantaneous gap $\ell_t(\tilde{\boldsymbol{\theta}}_t) - \ell_t(\boldsymbol{\theta}_t^\star)$, which is solely due to unpredictable future information (e.g. stochasticity).

Algorithm 3: Dynamic Mirror Descent MPC (`DMD-MPC`)

for *t=1, 2, . . . , T* **do**

Environment loss we receive is the MPC objective: $\ell_t(\cdot) = \hat{J}(\cdot; x_t)$

Apply DMD to get our new policy parameters:

$\boldsymbol{\theta}_t = \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \langle \gamma_t \nabla \ell_t(\tilde{\boldsymbol{\theta}}_t), \boldsymbol{\theta}\rangle + D_\psi(\boldsymbol{\theta}||\tilde{\boldsymbol{\theta}}_t)$

Sample $\hat{\boldsymbol{u}}_t \sim \boldsymbol{\pi}_{\boldsymbol{\theta}_t}$ and select the first control in the plan, $u_t = \hat{u}_t$

Apply the control to the system and sample $x_{t+1} \sim f(x_t, u_t)$

Shift our plan forward to warm-start the next time step: $\tilde{\boldsymbol{\theta}}_t = \Phi(\boldsymbol{\theta}_t)$

end

4.2 A Family of MPC Algorithms Based on Dynamic Mirror Descent

The online learning perspective on MPC suggests that good MPC algorithms can be designed from online learning algorithms known to achieve small dynamic regret. As such, we build upon a classical online learning algorithm known as dynamic mirror descent (DMD) [65], which in round t updates the policy parameters by the following update rule:

$$\boldsymbol{\theta}_t \leftarrow \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \langle \gamma_t g_t, \boldsymbol{\theta}\rangle + D_\psi(\boldsymbol{\theta}||\tilde{\boldsymbol{\theta}}_t), \quad \tilde{\boldsymbol{\theta}}_t = \Phi(\boldsymbol{\theta}_t), \tag{4.2}$$

where $g_t = \nabla \ell_t(\tilde{\boldsymbol{\theta}}_t)$, $\gamma_t > 0$ is the step size, Φ is called the shift model, and $D_\psi(\boldsymbol{\theta}||\boldsymbol{\theta}') = \psi(\boldsymbol{\theta}) - \psi(\boldsymbol{\theta}') - \langle \nabla\psi(\boldsymbol{\theta}'), \boldsymbol{\theta} - \boldsymbol{\theta}'\rangle$ is the Bregman divergence generated by a strictly convex function ψ on $\boldsymbol{\theta}$. This Bregman divergence acts as a regularizer to keep $\boldsymbol{\theta}$ close to $\tilde{\boldsymbol{\theta}}$.

The first step of DMD in Algorithm 3 is reminiscent of the proximal update in the usual mirror descent algorithm. It can be thought of as an optimization step where the Bregman divergence acts as a regularization to keep $\boldsymbol{\theta}$ close to $\tilde{\boldsymbol{\theta}}_t$ Although $D_\psi(\boldsymbol{\theta}||\boldsymbol{\theta}')$ is not necessarily a metric (since it may not be symmetric), it is still useful to view it as a distance between $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$. Indeed, familiar examples of the Bregman divergence include the squared Euclidean distance and the KL divergence [66]. The second step of DMD in Algorithm 3 uses the shift model Φ to anticipate the optimal decision for the next round. In the context of MPC, a natural choice for the shift model is the shift-forward operation discussed previous in Section 2.2. This is because the per-round losses in two consecutive

rounds here concern problems with shifted time indices. Hall and Willett [65] show that the dynamic regret of DMD scales with how much the optimal decision sequence $\{\boldsymbol{\theta}_t^*\}$ deviates from Φ (i.e., $\sum_t ||\boldsymbol{\theta}_{t+1}^* - \Phi(\boldsymbol{\theta}_t^*)||$), which is proportional to the unpredictable elements of the problem. Applying DMD in Equation (4.2) to MPC results in our `DMD-MPC` algorithm, shown in Algorithm 3. It represents a family of MPC algorithms in which a specific instance is defined by a choice of: 1) objective $\hat{J}$, 2) control distribution $\boldsymbol{\pi_\theta}$, and 3) Bregman divergence D_ψ. Thus, we can use `DMD-MPC` as a generic strategy for synthesizing MPC algorithms. We use this recipe to recreate several existing MPC algorithms and demonstrate new ones that naturally arise from this framework.

4.2.1 Loss Functions

We discuss different choices of statistic $\hat{J}(\boldsymbol{\theta}, x_t)$, which serves as the objective optimized by MPC. As we do not assume differentiability in many cases, we provide expressions for gradients in terms of the likelihood-ratio derivative [67]. That is, for some function $L_t(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$, the gradient is

$$\nabla \ell_t(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\pi_\theta}, \hat{\boldsymbol{f}}}[L_t(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \nabla_{\boldsymbol{\theta}} \log \boldsymbol{\pi_\theta}(\hat{\boldsymbol{u}}_t)]. \tag{4.3}$$

However, if the dynamics and cost function are differentiable, we can directly compute the gradients. And depending on the form of the policy and dynamics distributions, we can use the reparameterization trick to obtain lower-variance gradient estimates.

Expected Cost. The most commonly used MPC objective is the H-step expected accumulated cost function under the model dynamics:

$$\ell_t(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\pi_\theta}, \hat{\boldsymbol{f}}}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)], \qquad \nabla \ell_t(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\pi_\theta}, \hat{\boldsymbol{f}}}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \nabla_{\boldsymbol{\theta}} \log \boldsymbol{\pi_\theta}(\hat{\boldsymbol{u}}_t)], \tag{4.4}$$

as it directly estimates the expected long-term behavior when $\hat{f}$ is accurate and H is sufficiently large enough.

Expected Utility. Instead of optimizing the average cost, we may care to optimize for some preference related to the trajectory cost. This may include having the cost be below some

pre-specified threshold. This idea can be formulated as a *utility* that returns a normalized score related to the preference for a given trajectory cost. Let us define a utility function at round t to be some $U_t : \mathbb{R}_+ \to [0,1]$ with the following properties: $U_t(0) = 1$, U_t is monotonically decreasing, and $\lim_{z\to+\infty} U_t(z) = 0$. Then, assuming that C is lower bounded by zero [1] , $U_t(C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t))$ attains maximum utility when we have zero cost, it never increases with the cost, and it approaches zero as the cost increases without bound. The per-round loss is defined as

$$\ell_t(\boldsymbol{\theta}) = -\log \mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[U_t(C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t))], \quad \nabla \ell_t(\boldsymbol{\theta}) = -\frac{\mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[U_t(C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t))\nabla_{\boldsymbol{\theta}} \log \boldsymbol{\pi_\theta}(\hat{\boldsymbol{u}}_t)]}{\mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[U_t(C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t))]}. \tag{4.5}$$

When we approximate this gradient with samples, we have

$$\nabla \ell_t(\boldsymbol{\theta}) \approx -\sum_{i=1}^{N} w_i \log \boldsymbol{\pi_\theta}(\hat{\boldsymbol{u}}_t^{(i)}), \qquad w_i = \frac{U_t(C(\hat{\boldsymbol{x}}_t^i, \hat{\boldsymbol{u}}_t^i))}{\sum_{j=1}^{N} U_t(C(\hat{\boldsymbol{x}}_t^j, \hat{\boldsymbol{u}}_t^j))}, \tag{4.6}$$

where the weights are computed as a softmax. This enables them to consider the relative utility of its corresponding trajectory. A cost $C_i = C(\hat{\boldsymbol{x}}_t^i, \hat{\boldsymbol{u}}_t^i)$ with high relative utility will push its corresponding weight w_i closer to one. Whereas, a low relative utility will cause w_i to be close to zero, effectively rejecting the corresponding sample.

One example utility function follows from a desire to make sure our system is below some cost threshold as often as possible. To encode this preference, we use the threshold utility $U_t(C) \triangleq \mathbb{I}[C \le C_{t,\max}]$, where $\mathbb{I}[\cdot]$ is the indicator function and $C_{t,\max}$ is a threshold parameter. Under this choice, the loss and its gradient become

$$\ell_t(\boldsymbol{\theta}) = -\log \mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[\mathbb{I}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \le C_{t,\max}]] = -\log \mathbb{P}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \le C_{t,\max}], \tag{4.7}$$

$$\nabla \ell_t(\boldsymbol{\theta}) = -\frac{\mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[\mathbb{I}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \le C_{t,\max}]\nabla_{\boldsymbol{\theta}} \log \boldsymbol{\pi_\theta}(\hat{\boldsymbol{u}}_t)]}{\mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[\mathbb{I}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \le C_{t,\max}]]}. \tag{4.8}$$

We can see that this loss function also gives us the probability of achieving cost below some threshold. This can potentially make optimization easier, as we are trying to make good

[1]If this is not the case, let $c_{min} \triangleq \inf_{\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t} C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$, which we assume is finite. We can then replace C with $\tilde{C}(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \triangleq C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) - c_{min}$.

trajectories as likely as possible rather than finding the best trajectory. However, if the threshold is set too low, the gradient estimate may have high variance due to a large number of rejected samples. As such, we often set this threshold adaptively, such as the largest cost of the top *elite fraction* of sampled trajectories with smallest costs [68]. This allows the controller to make the best sampled trajectories more likely.

We can also opt for a continuous surrogate of the indicator function, such as the exponential utility $U_t(C) \triangleq \exp(-\frac{1}{\beta}C)$, where $\beta > 0$ is a scaling parameter. Unlike the indicator function, the exponential utility provides non-zero feedback for any given cost and allows us to discriminate between costs. That is, if $C_1 > C_2$, then $U_t(C_1) < U_t(C_2)$. Furthermore, β acts as a continuous alternative to $C_{t,\max}$, dictating how quickly or slowly U_t decays to zero. This determines the cutoff point for rejecting given costs in a "soft" way. Under this choice, the loss and its gradient become

$$\ell_t(\boldsymbol{\theta}) = -\log \mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}\Big[\exp\Big(-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)\Big)\Big], \quad \nabla\ell_t(\boldsymbol{\theta}) = -\frac{\mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\nabla_\theta \log \pi_\theta(\hat{\boldsymbol{u}}_t)\Big]}{\mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\Big]}. \tag{4.9}$$

In optimal control, this loss function is known as the risk-seeking objective, due to an interpretation of its Taylor expansion [69], showing:

$$\lambda\ell_t(\boldsymbol{\theta}) \approx \mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)] - \frac{1}{\lambda}\mathbb{V}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{f}}}[C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)] \tag{4.10}$$

However, we derive it from a different perspective, using an exponential transformation to an approximate indicator, which is a common machine-learning trick (e.g. Chernoff bound [70]).

4.2.2 Algorithms

Based on different choices of Bregman divergence, loss function, and control distribution, we arrive at well-known MPC algorithms. We assume $\boldsymbol{\pi_\theta}$ factorizes as $\boldsymbol{\pi_\theta}(\hat{\boldsymbol{u}}_t) = \prod_{h=0}^{H-1}\pi_{\theta_h}(\hat{u}_{t+h})$, and $\boldsymbol{\theta} \triangleq (\theta_0, \theta_1, \ldots, \theta_{H-1})$ for some basic control distribution π_θ. With control distributions of this form, the shift operator can simply shift the control sequence forward one time step and append some control at the end, as discussed in Section 2.2.

Quadratic Divergence. The most common Bregman divergence is the quadratic divergence, which assumes a quadratic form: $D_\psi(\boldsymbol{\theta}||\boldsymbol{\theta}') = \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}')^T A(\boldsymbol{\theta} - \boldsymbol{\theta}')$ for some positive-definite matrix A. When A is the identity matrix, we get projected gradient descent, which has the update $\boldsymbol{\theta}_t = \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} ||\boldsymbol{\theta}_t - (\tilde{\boldsymbol{\theta}}_t - \gamma_t g_t)||^2$. When we define $A = \mathcal{F}(\tilde{\theta}_t)$ to be the Fisher information matrix of $\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}}$, we recover natural gradient descent [71]. While these are general update rules, we can further specialize the Bregman divergence to achieve faster learning when the per-round loss function is also quadratic. For instance, in the case of LQR or a variant known as LEQR (when the quadratic cost function is exponentiated), we can write that $\ell_t(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^T R_t \boldsymbol{\theta} + r_t^T \boldsymbol{\theta} + \text{const.}$, for some constant vector r_t and positive definite matrix R_t. We can then set $A = R_t$ and $\gamma_t = 1$, making $\boldsymbol{\theta}_t$ given by the mirror descent step in Equation (4.2) to be the optimal solution of LQR/LEQR.

More specifically, the dynamics of the system are given by

$$x_{t+1} = Ax_t + Bu_t + w_t \tag{4.11}$$

for some matrices $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$ and $w_t \sim \mathcal{N}(0, W)$, where $W \in \mathbb{S}^n_{++}$. For a control sequence $\hat{\boldsymbol{u}}_t$, noise sequence $\hat{\boldsymbol{w}}_t$, and initial state x_t, the resulting state sequence $\hat{\boldsymbol{x}}_t$ is found through convolution:

$$\begin{bmatrix} \hat{x}_t \\ \hat{x}_{t+1} \\ \hat{x}_{t+2} \\ \vdots \\ \hat{x}_{t+H} \end{bmatrix} = \begin{bmatrix} I \\ A \\ A^2 \\ \vdots \\ A^H \end{bmatrix} x_t + \begin{bmatrix} 0 & 0 & \cdots & 0 \\ B & 0 & \cdots & 0 \\ AB & B & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A^{H-1}B & A^{H-2}B & \cdots & B \end{bmatrix} \begin{bmatrix} \hat{u}_t \\ \hat{u}_{t+1} \\ \vdots \\ \hat{u}_{t+H-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 & \cdots & 0 \\ I & 0 & \cdots & 0 \\ A & I & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A^{H-1} & A^{H-2} & \cdots & I \end{bmatrix} \begin{bmatrix} \hat{w}_t \\ \hat{w}_{t+1} \\ \vdots \\ \hat{w}_{t+H-1} \end{bmatrix}, \tag{4.12}$$

or, in matrix form:

$$\hat{\boldsymbol{x}}_t = \boldsymbol{F}x_t + \boldsymbol{G}\hat{\boldsymbol{u}}_t + \boldsymbol{L}\hat{\boldsymbol{w}}_t, \tag{4.13}$$

where $\boldsymbol{F}$, $\boldsymbol{G}$, and $\boldsymbol{L}$ are defined naturally from the convolution equation above. Note that $\hat{\boldsymbol{w}}_t \sim \mathcal{N}(0, \boldsymbol{W})$, where $\boldsymbol{W} = \text{diag}(W, \ldots, W)$. Thus, we also have that

$$\hat{\boldsymbol{x}}_t \sim \mathcal{N}(\boldsymbol{F}x_t + \boldsymbol{G}\hat{\boldsymbol{u}}_t, \boldsymbol{L}\boldsymbol{W}\boldsymbol{L}^T). \tag{4.14}$$

We define the instantaneous and terminal costs as

$$c(x,u) = \frac{1}{2}x^T Q x + \frac{1}{2}u^T R u, \tag{4.15}$$

$$c_{term}(x) = \frac{1}{2}x^T Q_{term} x, \tag{4.16}$$

where $Q, Q_{term} \in \mathbb{S}^n_+$ and $R \in \mathbb{S}^m_{++}$. Thus, the statistic $C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$ is

$$C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) = \frac{1}{2}\hat{\boldsymbol{x}}_t^T \boldsymbol{Q} \hat{\boldsymbol{x}}_t + \frac{1}{2}\hat{\boldsymbol{u}}_t^T \boldsymbol{R} \hat{\boldsymbol{u}}_t, \tag{4.17}$$

where $\boldsymbol{Q} = \operatorname{diag}(Q, \ldots, Q, Q_{term})$ and $\boldsymbol{R} = \operatorname{diag}(R, \ldots, R)$. Our control distribution is a Dirac delta distribution located at the given parameter:

$$\boldsymbol{\pi}_{\boldsymbol{\theta}}(\hat{\boldsymbol{u}}_t) = \delta(\hat{\boldsymbol{u}}_t - \boldsymbol{\theta}). \tag{4.18}$$

For LQR, the loss is defined as

$$\begin{aligned} \ell_t(\boldsymbol{\theta}) &= \mathbb{E}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}_t}\left[\frac{1}{2}\hat{\boldsymbol{x}}_t^T \boldsymbol{Q} \hat{\boldsymbol{x}}_t + \frac{1}{2}\hat{\boldsymbol{u}}_t^T \boldsymbol{R} \hat{\boldsymbol{u}}_t\right] \\ &= \frac{1}{2}\boldsymbol{\theta}^T(\boldsymbol{G}^T\boldsymbol{Q}\boldsymbol{G} + \boldsymbol{R})\boldsymbol{\theta} + x_t^T \boldsymbol{F}^T \boldsymbol{Q}\boldsymbol{G}\boldsymbol{\theta} + \frac{1}{2}x_t^T \boldsymbol{F}^T \boldsymbol{Q}\boldsymbol{F} x_t + \frac{1}{2}\mathbb{E}\left[\hat{\boldsymbol{w}}_t^T \boldsymbol{L}^T \boldsymbol{Q}\boldsymbol{L}\hat{\boldsymbol{w}}_t\right] \\ &= \frac{1}{2}\boldsymbol{\theta}^T(\boldsymbol{G}^T\boldsymbol{Q}\boldsymbol{G} + \boldsymbol{R})\boldsymbol{\theta} + x_t^T \boldsymbol{F}^T \boldsymbol{Q}\boldsymbol{G}\boldsymbol{\theta} + \frac{1}{2}x_t^T \boldsymbol{F}^T \boldsymbol{Q}\boldsymbol{F} x_t + \frac{1}{2}\operatorname{tr}\big(\boldsymbol{Q}\boldsymbol{L}\boldsymbol{W}\boldsymbol{L}^T\big). \end{aligned} \tag{4.19}$$

We see this is a quadratic problem in $\boldsymbol{\theta}$ by defining

$$\boldsymbol{R}_t = \boldsymbol{G}^T\boldsymbol{Q}\boldsymbol{G} + \boldsymbol{R}, \quad \boldsymbol{r}_t = \boldsymbol{G}^T\boldsymbol{Q}\boldsymbol{F}x_t. \tag{4.20}$$

Similarly, for LEQR, the loss is defined as

$$\ell_t(\boldsymbol{\theta}) = -\log \mathbb{E}_{\boldsymbol{\pi}_{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}_t}\left[\exp\left(-\frac{1}{\lambda}\left(\frac{1}{2}\hat{\boldsymbol{x}}_t^T \boldsymbol{Q} \hat{\boldsymbol{x}}_t + \frac{1}{2}\hat{\boldsymbol{u}}_t^T \boldsymbol{R} \hat{\boldsymbol{u}}_t\right)\right)\right], \tag{4.21}$$

for some parameter $\lambda > 0$. For compactness, we define $\boldsymbol{Q'} = \frac{1}{\lambda}\boldsymbol{Q}$ and $\boldsymbol{R'} = \frac{1}{\lambda}\boldsymbol{R}$ so that the exponent contains $-\frac{1}{2}\hat{\boldsymbol{x}}_t^T \boldsymbol{Q'} \hat{\boldsymbol{x}}_t - \frac{1}{2}\hat{\boldsymbol{u}}_t^T \boldsymbol{R'} \hat{\boldsymbol{u}}_t$. In expanding the loss, we use the following fact:

Fact 1. For $x \sim \mathcal{N}(\mu, \Sigma)$, where $\Sigma \in \mathcal{S}^n_{++}$, and constants $A \in \mathcal{S}^n_+$ and $b \in \mathbb{R}^n$:

$$\mathbb{E}_x\left[\exp\left(-\frac{1}{2}x^\mathsf{T} A x - b^\mathsf{T} x\right)\right] = \frac{1}{\sqrt{|A\Sigma + I|}}\exp\left(-\frac{1}{2}\left(\mu^T\Sigma^{-1}\mu - (\Sigma^{-1}\mu - b)^T(A + \Sigma^{-1})^{-1}(\Sigma^{-1}\mu - b)\right)\right).$$

Proof. We expand the expectation and complete the square:

$$\begin{aligned}
\mathbb{E}_x\left[\exp\left(-\frac{1}{2}x^\top Ax - b^\top x\right)\right] &= \frac{1}{\sqrt{(2\pi)^n|\Sigma|}}\int \exp\left(-\frac{1}{2}(x-\mu)^\top\Sigma^{-1}(x-\mu)\right)\exp\left(-\frac{1}{2}x^\top Ax - b^\top x\right)\mathrm{d}x \\
&= \frac{1}{\sqrt{(2\pi)^n|\Sigma|}}\int \exp\left(-\frac{1}{2}\left[x^\top(A+\Sigma^{-1})x + 2(b-\Sigma^{-1}\mu)^\top x + \mu^\top\Sigma^{-1}\mu\right]\right)\mathrm{d}x \\
&= \frac{1}{\sqrt{(2\pi)^n|\Sigma|}}\exp(c)\int \exp\left(-\frac{1}{2}(x-\tilde{\mu})^\top\tilde{\Sigma}^{-1}(x-\tilde{\mu})\right)\mathrm{d}x \\
&= \frac{\sqrt{(2\pi)^n|\tilde{\Sigma}|}}{\sqrt{(2\pi)^n|\Sigma|}}\exp(c) \\
&= \frac{1}{\sqrt{|A+\Sigma^{-1}||\Sigma|}}\exp(c) \\
&= \frac{1}{\sqrt{|A\Sigma+I|}}\exp\left(-\frac{1}{2}\left(\mu^\top\Sigma^{-1}\mu - (\Sigma^{-1}\mu - b)^\top(A+\Sigma^{-1})^{-1}(\Sigma^{-1}\mu - b)\right)\right),
\end{aligned}$$

where

$$\tilde{\mu} = (A+\Sigma^{-1})^{-1}(\Sigma^{-1}\mu - b), \quad \tilde{\Sigma} = (A+\Sigma^{-1})^{-1},$$
$$c = -\frac{1}{2}\left(\mu^\top\Sigma^{-1}\mu - (\Sigma^{-1}\mu - b)^\top(A+\Sigma^{-1})^{-1}(\Sigma^{-1}\mu - b)\right)$$

□

We now expand the loss:

$$\begin{aligned}
\ell_t(\boldsymbol{\theta}) &= -\log\mathbb{E}_{\boldsymbol{\pi_\theta},\hat{\boldsymbol{x}}_t}\left[\exp\left(-\frac{1}{2}\hat{\boldsymbol{x}}_t^\top\boldsymbol{Q}'\hat{\boldsymbol{x}}_t - \frac{1}{2}\hat{\boldsymbol{u}}_t^\top\boldsymbol{R}'\hat{\boldsymbol{u}}_t\right)\right] \\
&= -\log\mathbb{E}_{\hat{\boldsymbol{x}}_t}\left[\exp\left(-\frac{1}{2}\hat{\boldsymbol{x}}_t^\top\boldsymbol{Q}'\hat{\boldsymbol{x}}_t - \frac{1}{2}\boldsymbol{\theta}^\top\boldsymbol{R}'\boldsymbol{\theta}\right)\right] \\
&= -\log\left\{\frac{1}{\sqrt{|\boldsymbol{Q}'\boldsymbol{LWL}^\top + I|}}\exp\left(-\frac{1}{2}[(\boldsymbol{F}x_t+\boldsymbol{G\theta})^\top(\boldsymbol{LWL}^\top)^{-1}(\boldsymbol{F}x_t+\boldsymbol{G\theta})\right.\right. \\
&\qquad - (\boldsymbol{F}x_t+\boldsymbol{G\theta})^\top(\boldsymbol{LWL}^\top\boldsymbol{Q}'\boldsymbol{LWL}^\top + \boldsymbol{LWL}^\top)^{-1}(\boldsymbol{F}x_t+\boldsymbol{G\theta}) \\
&\qquad \left.\left. + \boldsymbol{\theta}^\top\boldsymbol{R}'\boldsymbol{\theta}]\right)\right\} \\
&= \frac{1}{2}\left[(\boldsymbol{F}x_t+\boldsymbol{G\theta})^\top[(\boldsymbol{LWL}^\top)^{-1} + (\boldsymbol{LWL}^\top\boldsymbol{Q}'\boldsymbol{LWL}^\top + \boldsymbol{LWL}^\top)^{-1}](\boldsymbol{F}x_t+\boldsymbol{G\theta}) + \boldsymbol{\theta}^\top\boldsymbol{R}'\boldsymbol{\theta}\right] + \frac{1}{2}\log|\boldsymbol{Q}'\boldsymbol{LWL}^\top + I|.
\end{aligned}$$

We see this is a quadratic problem in $\boldsymbol{\theta}$ by defining

$$\begin{aligned}
\boldsymbol{R}_t &= \boldsymbol{G}^\top\left[(\boldsymbol{LWL}^\top)^{-1} + \left(\frac{1}{\lambda}\boldsymbol{LWL}^\top\boldsymbol{QLWL}^\top + \boldsymbol{LWL}^\top\right)^{-1}\right]\boldsymbol{G} + \frac{1}{\lambda}\boldsymbol{R} \\
\boldsymbol{r}_t &= \boldsymbol{G}^\top\left[(\boldsymbol{LWL}^\top)^{-1} + \left(\frac{1}{\lambda}\boldsymbol{LWL}^\top\boldsymbol{QLWL}^\top + \boldsymbol{LWL}^\top\right)^{-1}\right]\boldsymbol{F}x_t.
\end{aligned}$$

KL Divergence and the Exponential Family. When the control distribution is in the exponential family [72], a natural choice of Bregman divergence is the KL divergence. A distribution p_η, with natural parameter η, of a random variable u belongs to the exponential family if its probability density or mass function satisfies $p_\eta = \rho(u)\exp(\langle \eta, \phi(u)\rangle - A(\eta))$, where $\phi(u)$ is the sufficient statistics, $\rho(u)$ is called the carrier measure, and $A(\eta)$ is the log-partition function. We can also describe the distribution by its expectation parameter $\mu = \mathbb{E}_{p_\eta}[\phi(u)]$. There is a duality between these two parameterizations: $\mu = \nabla A(\eta)$ and $\eta = \nabla A^*(\mu)$, where $A^*(\mu) = \sup_{\eta\in\mathcal{H}}\langle \eta, \mu\rangle - A(\eta)$ is the Legendre transformation of A and $\mathcal{H} = \{\eta : A(\eta) < \infty\}$. The duality results in the property $KL(p_\eta || p_{\eta'}) = D_A(\eta' || \eta) = D_{A^*}(\mu || \mu')$. We can use this property to define the Bregman divergence in Equation (4.2) to optimize a control distribution $\boldsymbol{\pi}_{\boldsymbol{\theta}}$ in the exponential family.

First, we discuss the case where $\boldsymbol{\theta}$ is an expectation parameter, and $D_\Psi(\boldsymbol{\theta} || \tilde{\boldsymbol{\theta}}_t) = KL(\boldsymbol{\pi}_{\boldsymbol{\theta}} || \boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t})$. One example is when we have a discrete control space $\{1, 2, \ldots, m\}$ and the categorical distribution as the basic control distribution. That is, we set $\pi_{\theta_{t+h}} = \mathrm{Cat}(\theta_{t+h})$, where $\theta_{t+h} \in \Delta^m$ is the probability of choosing each discrete control option at the hth predicted time step, and Δ^m denotes the probability simplex in $\mathbb{R}^m$. This parameterization choice makes $\boldsymbol{\theta}$ an expectation parameter of $\boldsymbol{\pi}_{\boldsymbol{\theta}}$, and we choose our Bregman divergence to be $D_{KL}(\boldsymbol{\pi}_{\boldsymbol{\theta}} || \boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t})$. Using the likelihood-ratio gradient defined in Equation (4.3), the update direction is given by the following:

$$g_{t+h} = \mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}[L_t(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) e_{\hat{u}_{t+h}} \oslash \tilde{\theta}_{t+h}], \quad (h = 0, 1, \ldots, H-1), \tag{4.22}$$

where $e_{\hat{u}_{t+h}} \in \mathbb{R}^m$ has 0 for each element except at the index $\hat{u}_{t+h}$, where it is a 1, and $\oslash$ denotes element-wise division. The DMD update then becomes the exponentiated gradient algorithm [64]:

$$\theta_{t+h} = \frac{1}{Z_{t+h}} \tilde{\theta}_{t+h} \odot \exp(-\gamma_t g_{t+h}), \quad (h = 0, 1, \ldots, H-1), \tag{4.23}$$

where Z_{t+h} is the normalizer for θ_{t+h} and $\odot$ denotes element-wise multiplication. That is, instead of applying an additive gradient step to the parameters, the update exponentiates

the gradient and performs element-wise multiplication. This does a better job accounting for the geometry of the problem and makes projection a simple normalization operation.

Alternatively, we can set $\boldsymbol{\theta}$ as a natural parameter and use the Bregman divergence $D_{KL}(\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}}||\boldsymbol{\pi}_{\boldsymbol{\theta}_t})$. When the gradient is computed through the likelihood ratio in Equation (4.3), we can write it as the following expression:

$$g_t = \mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}[L_t(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)(\phi(\hat{\boldsymbol{u}}_t) - \tilde{\boldsymbol{\mu}}_t)], \tag{4.24}$$

where $\tilde{\boldsymbol{\mu}}_t$ is the expectation parameter of $\tilde{\boldsymbol{\theta}}_t$ and ϕ are the sufficient statistics of the control distribution. Next, we use the following property of the proximal update.

Proposition 4.2.1. *Let g_t be an update direction. Let $\mathcal{M}$ be the image of $\mathcal{H}$ under ∇A. If $\mu_t - \gamma_t g_t \in \mathcal{M}$ and $\eta_{t+1} = \arg\min_{\eta \in \mathcal{H}} \langle \gamma_t g_t, \eta \rangle + D_A(\eta||\eta_t)$, then $\mu_{t+1} = \mu_t - \gamma_t g_t$.*

Proof. We prove the first statement, with the second one following directly from the duality relationship. We can write

$$\begin{aligned}
\eta_{t+1} &= \arg\min_{\eta \in \mathcal{H}} \langle \gamma_t g_t, \eta \rangle + D_A(\eta||\eta_t) \\
&= \arg\min_{\eta \in \mathcal{H}} \langle \gamma_t g_t, \eta \rangle + A(\eta) - \langle \nabla A(\eta_t), \eta \rangle \\
&= \arg\min_{\eta \in \mathcal{H}} \langle \gamma_t g_t - \mu_t, \eta \rangle + A(\eta) \\
&= \arg\max_{\eta \in \mathcal{H}} \langle \mu_t - \gamma_t g_t, \eta \rangle - A(\eta) \\
&= \nabla A^*(\mu_t - \gamma_t g_t)
\end{aligned}$$

where the last equality is due to the assumption that $\mu_t - \gamma_t g_t \in \mathcal{M}$. Then applying ∇A on both sides and using the relationship that $\nabla A = (\nabla A^*)^{-1}$, we have $\mu_{t+1} = \nabla A(\eta_{t+1}) = \mu_t - \gamma_t g_t$. □

Under the assumption [2] in Proposition 4.2.1, the update rule becomes

$$\boldsymbol{\mu}_{t+1} = (1 - \gamma_t)\tilde{\boldsymbol{\mu}}_t + \gamma_t \mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}[L_t(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)\phi(\hat{\boldsymbol{u}}_t)]. \tag{4.25}$$

[2] If $\mu_t - \gamma_t g_t$ is not in $\mathcal{M}$, the update needs to perform a projection, the form of which is algorithm dependent.

That is, when $\gamma_t \in [0,1]$, the update to the expectation parameter μ_t is simply a convex combination of the sufficient statistics and the previous expectation parameter $\tilde{\boldsymbol{\mu}}_t$.

There are two prominent MPC algorithms that follow an update of this form. Consider a continuous control space and use the Gaussian distribution as our basic control distribution. That is, we set $\pi_{\theta_{t+h}}(\hat{u}_{t+h}) = \mathcal{N}(\hat{u}_{t+h}; m_{t+h}, \Sigma_{t+h})$ for some mean vector m_{t+h} and covariance matrix Σ_{t+h}. For $\pi_{\theta_{t+h}}$, we can choose sufficient statistics $\phi(\hat{u}_{t+h}) = (\hat{u}_{t+h}, \hat{u}_{t+h}\hat{u}_{t+h}^T)$, which results in the expectation parameter $\mu_{t+h} = (m_{t+h}, S_{t+h})$, where $S_{t+h} = \Sigma_{t+h} + m_{t+h}m_{t+h}^T$ is the second moment. Let us set θ_{t+h} to be the corresponding natural parameter, which is $\eta_h = (\Sigma_{t+h}^{-1}m_{t+h}, -\frac{1}{2}\Sigma_{t+h}^{-1})$. Then we have the update rule for $h = 0, \ldots, H-1$:

$$\begin{aligned} m_{t+h} &= (1-\gamma_t)\tilde{m}_{t+h} + \gamma_t \mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}[L_t(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)\hat{u}_{t+h}], \\ S_{t+h} &= (1-\gamma_t)\tilde{S}_{t+h} + \gamma_t \mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}[L_t(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)\hat{u}_{t+h}\hat{u}_{t+h}^T]. \end{aligned} \tag{4.26}$$

If we set ℓ_t to Equation (4.7) and $\gamma_t = 1$, we arrive at the cross-entropy method (CEM) [68]:

$$\begin{aligned} m_{t+h} &= (1-\gamma_t)\tilde{m}_{t+h} + \gamma_t \mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}\Big[\mathbb{I}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \le C_{t,\max}]\hat{u}_{t+h}\Big], \\ S_{t+h} &= (1-\gamma_t)\tilde{S}_{t+h} + \gamma_t \mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}\Big[\mathbb{I}[C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \le C_{t,\max}]\hat{u}_{t+h}\hat{u}_{t+h}^T\Big]. \end{aligned} \tag{4.27}$$

Although CEM is typically presented as updating the mean and covariance of a Gaussian distribution, the update rule is derived by matching the first and second moments between the Gaussian distribution and a uniform distribution over trajectories, whose costs are most $C_{t,max}$, which is identical to Equation (4.27). Alternatively, if we choose ℓ_t to be Equation (4.9) and do not update the covariance, we have

$$m_{t+h} = (1-\gamma_t)\tilde{m}_{t+h} + \gamma_t \frac{\mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)}\hat{u}_{t+h}\right]}{\mathbb{E}_{\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}, \hat{\boldsymbol{f}}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)}\right]}, \tag{4.28}$$

which is MPPI if we set $\gamma_t = 1$. This connection is also noted in [50], but their derivation is limited to the KL divergence and Gaussian case. While this shows that MPPI lies under the `DMD-MPC` framework, it also opens the door for modifying the algorithm. For instance, we can re-introduce the step size parameter rather than fixing it. Moreover, this perspective provides a way of performing covariance adaptation, which may help improve performance in practice.

4.2.3 Extensions

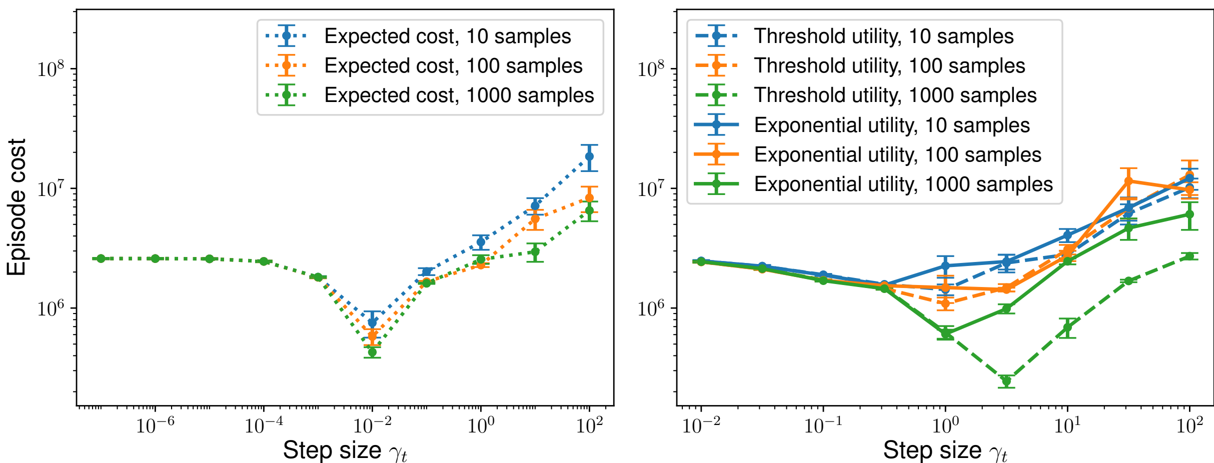
The control distributions in `DMD-MPC` can be fairly general (in addition to the categorical and Gaussian distributions that we discussed) and control constraints on the problem (e.g. control limits) can be directly incorporated through proper choices of control distributions. For instance, we could use the beta distribution, or through mapping the unconstrained control through some squashing function (e.g. tanh or clamp). Though our framework cannot directly handle state constraints, as in constrained optimization approaches, a constraint can be relaxed to an indicator function which activates if the constraint is violated. The indicator function can then be added to the cost function with some weight that encodes how strictly the constraint should be enforced. Moreover, different integration techniques, such as Gaussian quadrature [73], can be adopted to replace the likelihood-ratio derivative for computing the required gradient direction. We also note that the independence assumption on the control distribution is not necessary in our framework; time-correlated control distributions and feedback policies are straightforward to consider in `DMD-MPC`.

4.3 Experimental Results

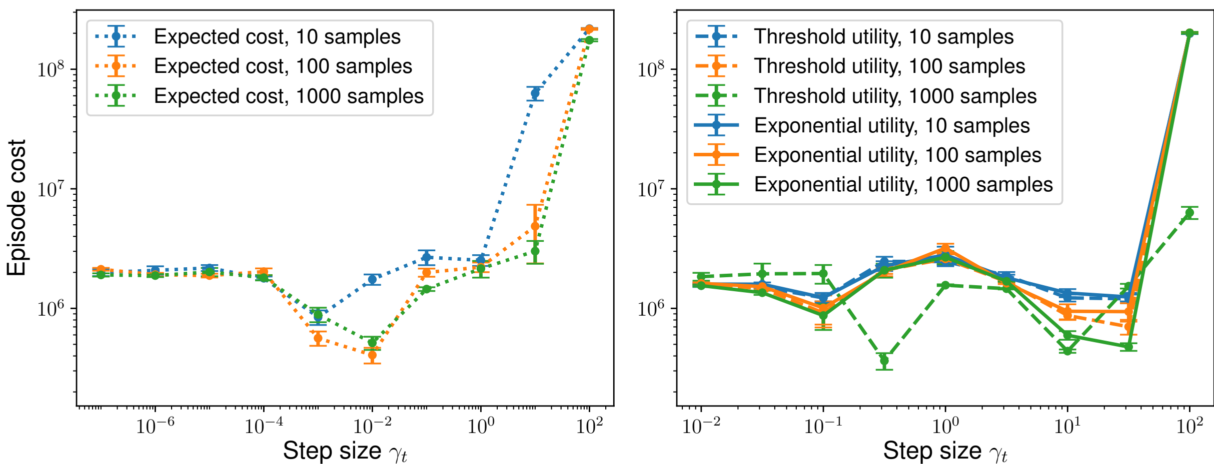
4.3.1 Cartpole

We first consider the classic Cartpole problem with both continuous and discrete control variants, in which we use a Gaussian with fixed covariance or categorical distribution, respectively. For both cases, MPC only has access to a biased stochastic model which uses a different pole length compared to the real cart. The state is $x_t = (p_t, \varphi_t, v_t, \dot{\varphi}_t)$, where p_t is the cart position, φ_t is the pole's angle, v_t and $\dot{\varphi}_t$ are the corresponding velocities, and the control u_t is the force applied to the cart. We define the instantaneous cost and terminal cost of the MPC problem as

$$
\begin{aligned}
c(x_t, u_t) &= 10p_t^2 + 500(\varphi_t - \pi)^2 + v_t^2 + 15\dot{\varphi}_t^2 + 1000 \cdot \mathbf{1}\left\{|\varphi_t - \pi| \geq \Delta\right\} \\
c_{\text{term}}(x_t) &= c(x_t, 0)
\end{aligned}
$$



(a) Continuous Controls

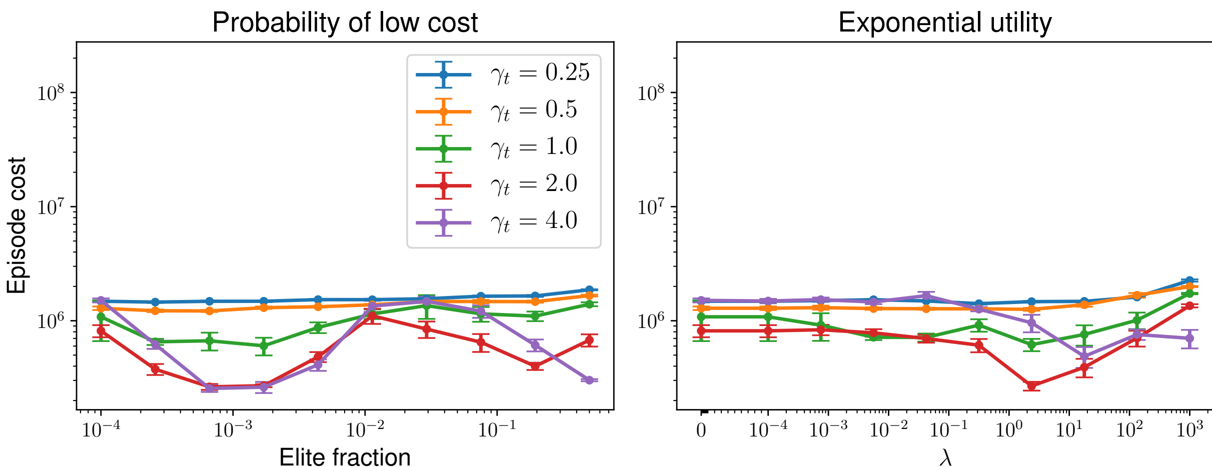


(b) Discrete Controls

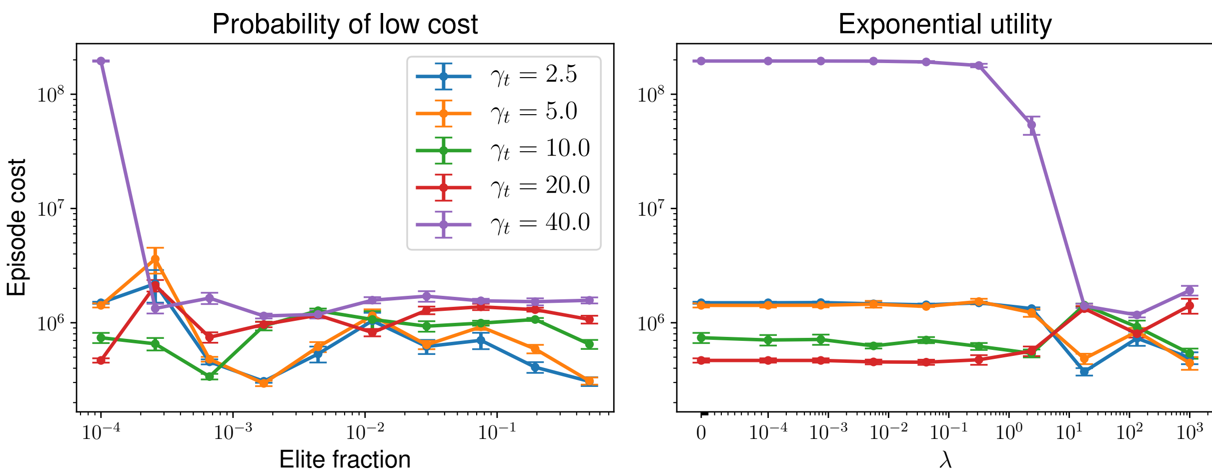
Figure 4.2: Varying step size and number of samples for the Cartpole with (a) continuous and (b) discrete controls, with expected cost (EC, Equation (4.4)), probability of low cost (PLC, Equation (4.7)), and exponential utility (EU, Equation (4.9)).

where Δ is some threshold. For our experiments, we set $\Delta = 12° = 0.21$ radians.

In our experiments, the pole is massless except for some weight at the end of the pole. The mass of the cart and pole weight are 0.711 kg and 0.209 kg, respectively. The true length of the



(a) Continuous Controls



(b) Discrete Controls

Figure 4.3: Varying the loss parameters and step size with a fixed # of samples on Cartpole.

pole is 0.326 m, whereas the length used in the model is 0.346 m. Each time step is modeled using an Euler discretization of 0.02 seconds. Each episode of the problem lasts 500 time steps (i.e, 10 seconds) and has episode cost equal to the sum of encountered instantaneous costs. Both the true system and the model apply Gaussian additive noise to the commanded control with zero mean and a standard deviation of 5 N. For the continuous system, the commanded

Figure 4.4: Rally car (right) and real-world AutoRally task (left).

control is clamped to ±25 N. For the discrete system, the controller can either command 10 N to the left, 10 N to the right, or 0 N. Both the discrete and continuous controller use a planning horizon of 50 time steps (i.e., 1 second). For the continuous controller, we keep the standard deviation of the Gaussian distribution fixed at 2 N for each time step in the planning horizon. When applying a control u_t on the real cartpole, we choose the mode of π_{θ_t} rather than sample from the distribution.

In Figure 4.2, we consider the interaction between the choice of loss and number of samples. We can achieve low cost when optimizing the expected cost in Equation (4.4) with a proper step size while being fairly robust to the number of samples. When using either of the utilities, the number of samples is more crucial in the continuous domain, with more samples allowing for larger step sizes. In the discrete domain (Figure 4.2b), the performance is largely unaffected by the number of samples when the step size is below 10, excluding the threshold utility with 1000 samples. In Figure 4.3, we explore varying the step size and loss parameters given 1000 samples. For the discrete case (Figure 4.3b), there's a more complicated interaction between the utility parameter and step size. That is, huge changes in cost occur when altering the utility and fixing the step size.

4.3.2 AutoRally

For a more realistic setting, we use the autonomous AutoRally platform [74] to run a high-speed driving task on a dirt track. The goal is to achieve as low a lap time as possible. The robot (Figure 4.4 is a 1:5 scale RC chassis capable of driving over 20 m/s (45 mph) and has a

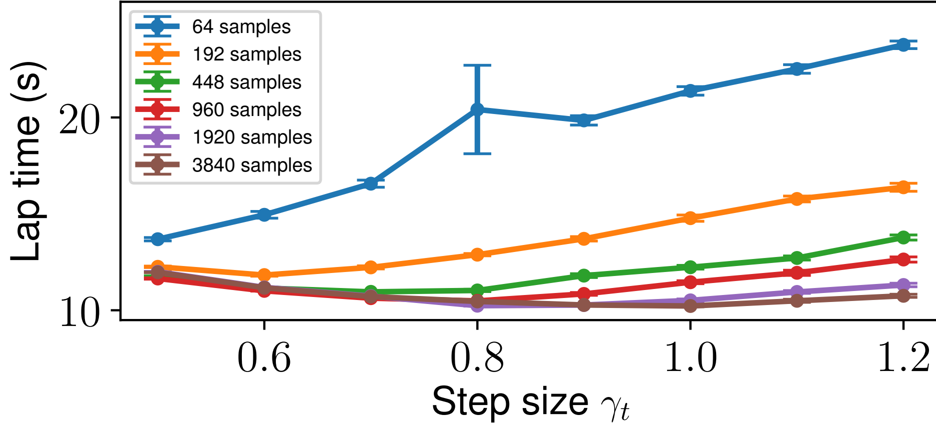


Figure 4.5: Simulated AutoRally performance for different step sizes and # samples.

desktop-class Intel Core i7 CPU and Nvidia GTX 1050 Ti GPU. In both simulated and real-world experiments, the dynamics model is a neural network which has been fit to data collected from human demonstrations. It is deterministic, so we don't need to estimate any expectations with respect to the dynamics. The state of the vehicle is $x_t = (p_{x,t}, p_{y,t}, \varphi_t, r_t, v_{x,t}, v_{y,t}, \dot{\varphi}_t)$, where $(p_{x,t}, p_{y,t})$ is the position of the car in the global frame, φ_t and r_t are the yaw and roll angles, $v_{x,t}$ and $v_{y,t}$ are the longitudinal and lateral velocities in the car frame, and $\dot{\varphi}_t$ is the yaw rate. The control u_t we apply is the throttle and steering angle. For some weights $w_1, \ldots, w_4$, the cost function is

$$c(x_t, u_t) = w_1|s_t - s_{\text{tgt}}|^k + w_2 M(p_{x,t}, p_{y,t}) + w_3 S_c(x_t)$$
$$c_{\text{end}}(\boldsymbol{x}_t) = w_4 C(\boldsymbol{x}_t).$$

Here, s_t and s_{tgt} are the current and target speed of the car, respectively. Note the speed is calculated as $s_t = \sqrt{v_{x,t}^2 + v_{y,t}^2}$. $M(p_{x,t}, p_{y,t})$ is the positional cost of the car (low cost in center of track, high cost at edge of track), $S_c(x_t)$ is an indicator variable which activates if the slip angle[3] exceeds a certain threshold, and $C(\boldsymbol{x}_t)$ is an indicator function which activates

[3]The slip angle is defined as $-\arctan \frac{v_{y,t}}{|v_{x,t}|}$, which gives the angle between the direction the car is pointing and the direction in which it is actually traveling.

Table 4.1: Cost function settings for AutoRally experiments.

	s_{tgt} (m/s)	k	w_1	w_2	w_3	w_4	Slip angle threshold (rad)
Gazebo simulator	11	1	30	250	10	10000	0.275
Real world	9 or 11	2	4.25	200	100	10000	0.9

if the car leaves the track at all in the trajectory. Note that the terminal cost depends on the trajectory instead of the terminal state. Each time step represents 0.02 seconds for every experiment except the real-world experiment with a target of 11 m/s where each time step represents 0.025 seconds. The length of the planning trajectory is 100 time steps (i.e., either 2 seconds or 2.5 seconds depending on the length of the time step). The values for the cost function parameters are given in Table 4.1.

The control space for each of the throttle and steering angle is normalized to the range $[-1, 1]$. For our experiments, we clamp the throttle to $[-1, 0.65]$. In simulated experiments, the standard deviations of the throttle and steering angle distributions were 0.3 and 0.275, respectively. In the real world experiments, they were both set to 0.3. When applying a control u_t on the car, we chose the mean of π_{θ_t} rather than sampling from the distribution. In simulation, the environment (Figure 4.6) is an elliptical track approximately 3 meters wide and 30 meters across at its furthest point. The real-world dirt track is about 5 meters wide and and has a track length of 170 meters. All reported results for simulated experiments were gathered using 30 consecutive laps in the counter-clockwise direction for each parameter setting. For real-world experiments, results were gathered using ten laps for each parameter setting when the target speed is 9 m/s and five laps for 11 m/s.

We first use the Gazebo simulator from the AutoRally repository to perform a sweep of algorithm parameters. In particular, we vary the step size and number of samples to evaluate how changing these parameters can affect the performance of `DMD-MPC`. For all experiments, the control distribution is a Gaussian with fixed covariance. In Figure 4.5, we see that although many samples coupled with large step sizes yield the smallest lap times, there are

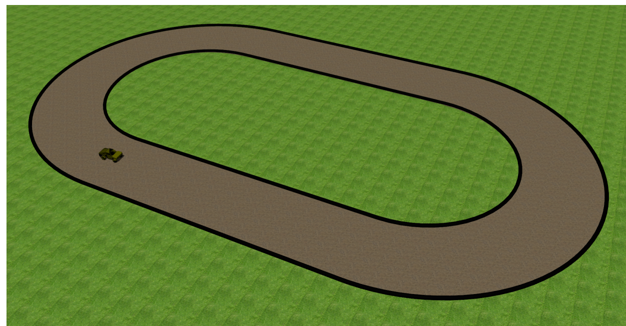
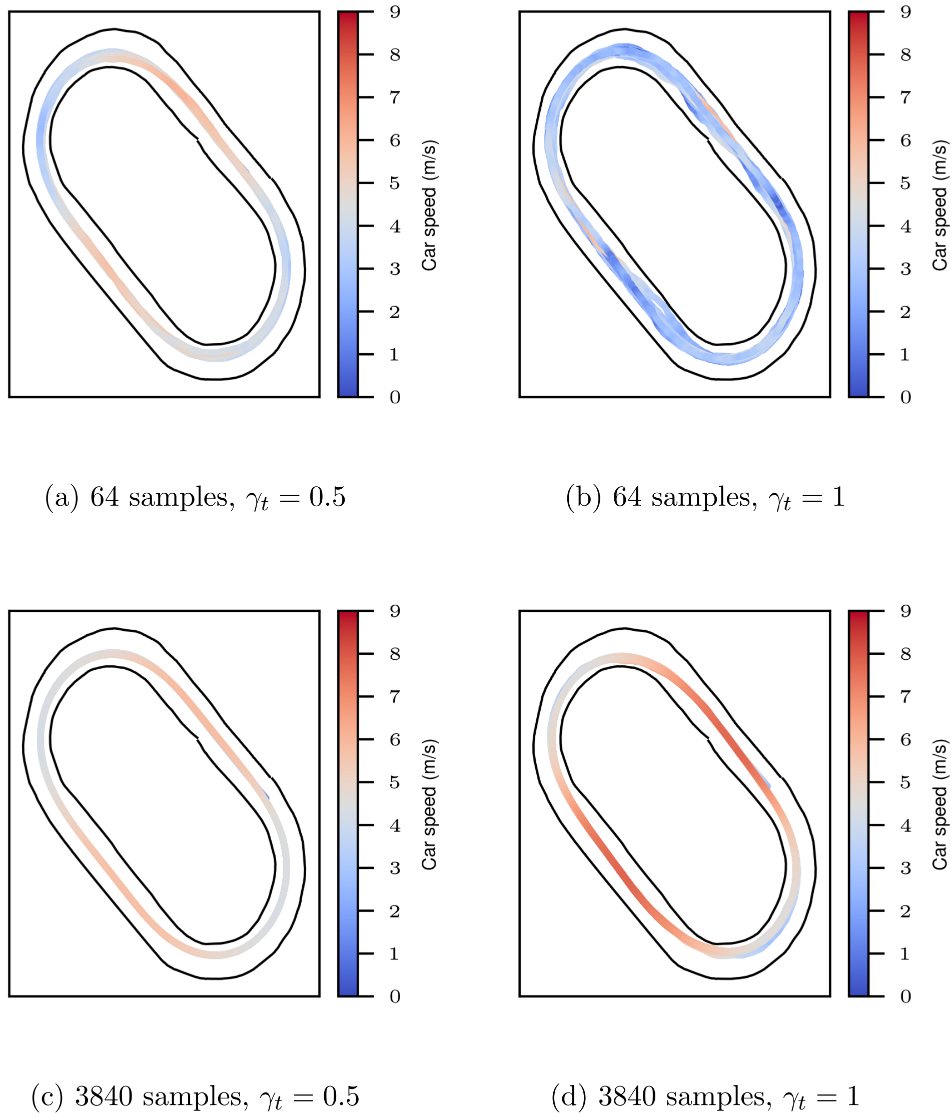


Figure 4.6: Simulated AutoRally task.

diminishing returns past 1920 samples per gradient. Indeed, with a proper step size, even as few as 192 samples can yield lap times within a couple seconds of 3840 samples using a step size of 1. We also observe that the curves converge as the step size decreases further, implying that only a certain number of samples are needed for a given step size. This is a particularly important advantage of `DMD-MPC` over methods like `MPPI`: by changing the step size, `DMD-MPC` can perform much more with fewer samples. This makes it a good choice for embedded systems with computational constraints.

Additionally, we qualitatively evaluate two particular extremes: few vs. many samples (64 vs. 3840) and small vs. large step size (0.5 vs. 1) by looking at the path and speed of the car during the episode (Figure 4.7). At small step sizes (Figures 4.7a and 4.7c), the path and speed profiles are rather similar, while with few samples and a large step size (Figure 4.7b), the car drives much more slowly and erratically, sometimes even stopping. In the ideal scenario with many samples and a large step size, the car can achieve consistently high speed while driving smoothly (Figure 4.7d). We also experimented with instead optimizing the expected cost and found performance was dramatically worse (Figure 4.8), even when using 3840 samples per gradient. At best, the car would drive in the center of the track at speeds below 4 m/s (Figure 4.8c), and at worst, the car would either slowly drive along the track walls (Figure 4.8a) or the controller would eventually produce NaN controls that would prematurely end the experiment (Figure 4.8d). This poor performance is likely due to most samples in the estimate of the gradient of the expected cost having very high cost (e.g., due



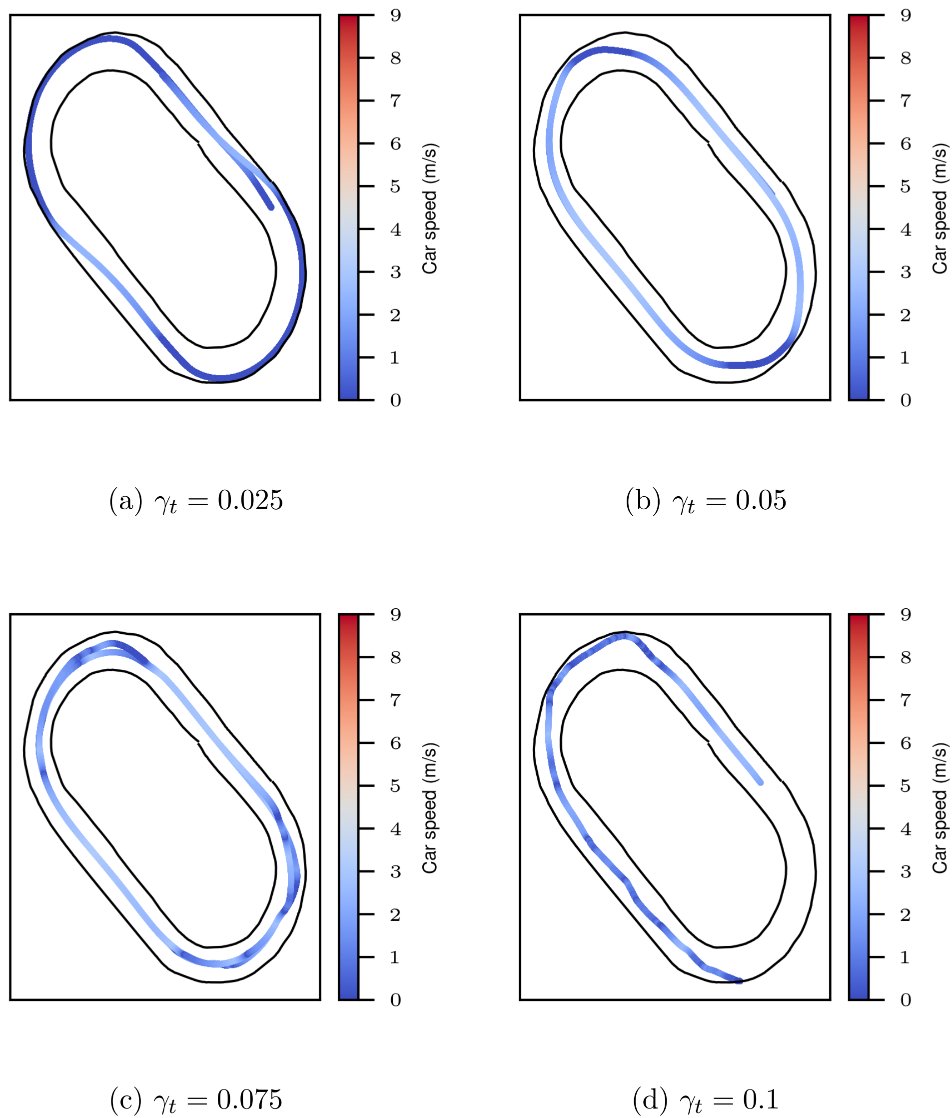
(a) 64 samples, $\gamma_t = 0.5$

(b) 64 samples, $\gamma_t = 1$

(c) 3840 samples, $\gamma_t = 0.5$

(d) 3840 samples, $\gamma_t = 1$

Figure 4.7: Car speeds when optimizing the exponential utility. The speeds and trajectories are very similar at step size 0.5, irrespective of the number of samples. At step size 1, though, 64 samples result in capricious maneuvers and low speeds, whereas 3840 samples result in smooth driving at high speeds.



(a) $\gamma_t = 0.025$

(b) $\gamma_t = 0.05$

(c) $\gamma_t = 0.075$

(d) $\gamma_t = 0.1$

Figure 4.8: Car speeds when optimizing the expected cost. All tested step sizes result in low speeds. At too low or too high of a step size, the car will drive along the wall or crash into it.

Table 4.2: Statistics for real-world experiments at target of 9 m/s.

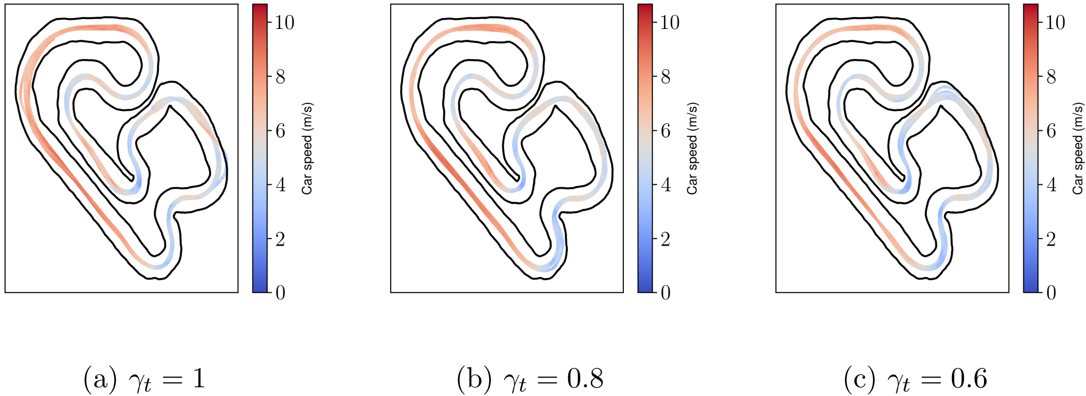
Samples	Step size γ_t	Lap time (s)	Avg. speed (m/s)	Max speed (m/s)
1920	1	31.76 ± 0.55	5.70 ± 0.16	9.21 ± 0.30
	0.8	31.81 ± 0.21	5.75 ± 0.03	9.03 ± 0.19
	0.6	32.83 ± 0.31	5.60 ± 0.05	8.62 ± 0.12
64	1	33.74 ± 0.78	5.45 ± 0.16	9.50 ± 0.22
	0.8	33.84 ± 0.80	5.46 ± 0.11	9.12 ± 0.26
	0.6	33.61 ± 0.74	5.50 ± 0.13	9.14 ± 0.42

to leaving the track) and contributing significantly to the gradient estimate. On the other hand, when estimating the gradient of the exponential objective, these high cost trajectories are assigned very low weights so that only low cost trajectories contribute to the gradient.

In the real-world setting, we ran two sets of experiments, each with a different target speed: one at 9 m/s and the other at 11 m/s. For the first set of experiments, Table 4.2 shows that there's a mild degradation in performance when decreasing the step size at 1920 samples, due to the car taking a longer path on the track. We see that there's a mild degradation in performance when decreasing the step size at 1920 samples, due to the car taking a longer path on the track. However, with 64 samples, the results seem unaffected by the step size. This could be because, despite the noise in the `DMD-MPC` update, the set-point controller in the car's steering serve acts as a filter, smoothing out the control signal. Videos of this experiment can be found at https://youtu.be/vZST3v0_S9w. For the second set of experiments, Table 4.3 shows that the statistics slightly improve with a decreased step size. However, qualitatively there is a larger difference between step sizes. With a step size of 1, the car often wobbles while driving, turns around at one point, and crashes in one of the trials. On the other hand, with a step size of 0.6, the car drives much more smoothly and achieves the aggressive driving task with no issues. Despite the smoothing effect of the low-level controllers in the car, the more stringent costs associated with the larger target speed cause

Table 4.3: Statistics for real-world experiments at target of 11 m/s.

Samples	Step size γ_t	Lap time (s)	Avg. speed (m/s)	Max speed (m/s)
64	1	31.05 ± 0.67	5.80 ± 0.26	10.17 ± 0.30
	0.6	30.30 ± 0.56	5.98 ± 0.15	10.30 ± 0.05



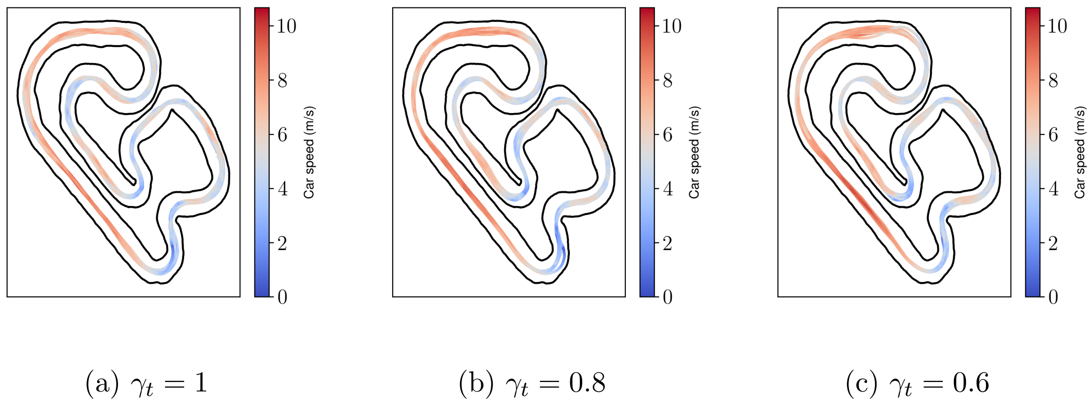
(a) $\gamma_t = 1$ (b) $\gamma_t = 0.8$ (c) $\gamma_t = 0.6$

Figure 4.9: Car speeds with 1920 samples per gradient estimate and target of 9 m/s.

the noise in the `DMD-MPC` update to manifest in the car's performance when using a step size of 1, which can be mitigated with a smaller step size. Videos of this experiment can be found at https://youtu.be/MhuqiHo2t98. Additionally, we provide qualitative evaluations of 1920 samples (Figure 4.9) vs. 64 samples (Figure 4.10) at a target speed of 9 m/s and 64 samples at a target speed of 11 m/s in Figure 4.11.

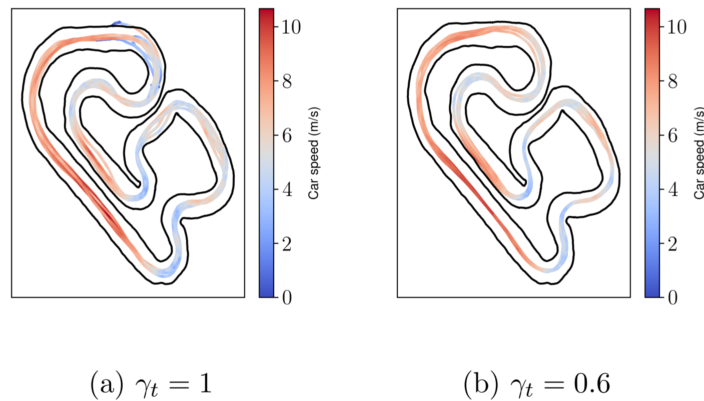
4.4 Discussion

This work presents a connection between MPC and online learning. From this connection, we proposed an algorithm based on dynamic mirror descent that can work for wide variety of settings and cost functions. We also discussed the choice of loss function within this online learning framework and the sort of preference each loss function imposes. From this general



(a) $\gamma_t = 1$ (b) $\gamma_t = 0.8$ (c) $\gamma_t = 0.6$

Figure 4.10: Car speeds with 64 samples per gradient estimate and target of 9 m/s.



(a) $\gamma_t = 1$ (b) $\gamma_t = 0.6$

Figure 4.11: Car speeds with 64 samples per gradient estimate and target of 11 m/s. In Figure 4.11a, note the crash and U-turn at the top of the plot as well as the wider spread of the paths throughout the whole track. By contrast, in Figure 4.11b, the resulting paths are more consistent, and there are no failure points.

algorithm and assortment of loss functions, we show several well known algorithms are special cases and presented a general update for members of the exponential family. We empirically validated our algorithm on continuous and discrete simulated problems and on a real-world

aggressive driving task. In this process, we also studied the parameter choices within the framework. We find, for example, that in our framework, a smaller number of rollout samples can be compensated for by varying other parameters, like the step size. The online learning and stochastic optimization viewpoints of MPC presented here open up new possibilities for using tools from these domains. This includes alternative efficient sampling techniques [73] and accelerated optimization methods [50] to derive new MPC algorithms.

Chapter 5

LEARNING TO OPTIMIZE IN MODEL PREDICTIVE CONTROL

The `DMD-MPC` perspective presented in Chapter 4 grounds sampling-based approaches to MPC in an optimization framework, opening the door for improving performance of existing algorithms by drawing on powerful optimization techniques. Most modern approaches to optimization use fixed update rules tailored to specific classes of problems. For first-order methods, this includes techniques such as momentum [75] or scaling updates based on the gradient history [63, 76–78]. Recently, research has explored *learning* to optimize [79], where the update rule is specified by a function approximator, such as a neural network, that can improve optimization performance with experience. In this chapter, drawing from our work in Sacks et al. [80], we leverage the optimization perspective of sampling-based MPC and adopt the learning-to-optimize framework in order to improve the update rule. Unlike in Chapter 3, we fix the dynamics and cost and instead focus on improving the optimization process.

For many practical sampling-based MPC algorithms, the primary challenge is finding a good trade-off between speed and accuracy. Using complex dynamics and cost functions can make each rollout prohibitively computationally expensive. To contend with this problem, one could use fewer samples to decrease computation, but this can increase the noise in the sample-based gradient, leading to poor performance. Therefore, our objective is to *learn* how to more effectively update the control distribution with a small number of samples. To this end, we employ imitation learning to train fast, low-sample controllers to imitate an expert which makes use of additional samples. The learned optimizer is better able to integrate information in the sample-constrained regime. Our experiments show that the learned controller is indeed able to make better use of fewer samples while remaining

competitive or outperforming the expert with the same number of samples. This indicates the potential to improve the utility of sampling-based MPC on embedded platforms.

5.1 Learning to Optimize for Control

Rather than hand design an update rule tailored to a specific subclass of problems, the learning-to-optimize approach aims to learn a update rule from experience. For a set of optimizee parameters $\theta \in \Theta$ and objective function $\ell(\theta)$, we find the minimizer $\theta^* = \arg\min_{\theta\in\Theta} \ell(\theta)$ with the update $\theta_{t+1} = m_\phi(\theta_t, t)$, where m is the learned optimizer, which can be of any parameterized function class with parameters ϕ. The majority of approaches to learning-to-optimize differentiate through the optimization process using gradient descent [81–85] or use reinforcement learning [86–89] with the goal of improving the training process of neural networks. However, we do not assume that the optimization process is end-to-end differentiable and instead use imitation learning to train the optimizer. Chen et al. [85] also make use of imitation learning, in which the experts are common hand-designed optimizers. In contrast, our learned optimizers only have access to noisier gradients than the expert. Additionally, most literature in this area targets optimizing deep neural networks, and thus must contend with the large parameter space of these models. Andrychowicz et al. [81] proposed to use a coordinate-wise optimizer, in which the parameters of the optimizer are shared across updates for all optimizee parameters. When the optimizer is implemented with a recurrent network, differences in the hidden state result in varying behaviors for each coordinate. A downside to this approach is that it throws away potentially useful information for improving the learned update. With a moderate number of parameters, we can jointly optimize them to capture more complex relationships.

5.1.1 Design of the Learnable Optimizer

As shown in the previous chapter, the MPPI update rule corresponds to performing mirror descent with an approximate gradient computed from N samples. Fewer samples results in a worse approximation and, therefore, a noisier update. One possible avenue for improving

performance would be to employ more advanced first-order methods [63, 75? –78]. However, by adopting the learning-to-optimize framework and replacing the update rule with a learned optimizer, we can potentially do better than a manually specified update. A naive approach for applying the learning-to-optimize framework to MPC would be to use the likelihood ratio gradient in Equation (4.3) approximated with samples as input to the learned optimizer to produce the updated parameters. This may be sufficient if our goal was to improve convergence speed to a local optima. However, our objective is instead to learn how to mitigate the effect of a low number of samples on gradient noise. The computation of the noisy gradient itself potentially throws away information that may be useful for improving the update. For instance, consider the `DMD-MPC` version of the `MPPI` update when approximated with Monte-Carlo samples:

$$\mu_{t+h} = (1-\gamma_t^{\mu})\tilde{\mu}_{t+h} + \gamma_t^{\mu}\sum_{i=1}^{N} w_i \hat{u}_{t+h}^{(i)}, \quad \Sigma_{t+h} = (1-\gamma_t^{\sigma})\tilde{\Sigma}_{t+h} + \gamma_t^{\sigma}\sum_{i=1}^{N} w_i m_{t+h}^{(i)} {m_{t+h}^{(i)}}^{T} \tag{5.1}$$

where we have used μ to indicate the mean rather than all expectation parameters, weights w_i are computed according to Equation (2.12), $m_{t+h} = \hat{u}_{t+h} - \mu_{t+h}$, and γ_t^{μ} and γ_t^{σ} are separate step sizes for the mean and covariance, respectively. Looking at Equation (5.1), we are simply computing a weighted sum of the samples. This collapses the information in each trajectory sample and its corresponding cost into a single vector. Therefore, instead of using the noisy gradient, we propose to use the individual components which form the gradient directly.

From Equation (5.1), we can see that the update is a function of the current mean $\tilde{\mu}_{t+h}$ and covariance $\tilde{\Sigma}_{t+h}$, sample weights $w_t^{(1:N)}$, and control samples $\hat{u}_{t+h}^{(1:N)}$. The sample weights themselves are actually a function of the total trajectory costs $C_t^{(1:N)}$, where $C_t = C(X_t, U_t)$. One potential choice would be to make each of these terms an input to the learned update:

$$\mu_{t+h}, \Sigma_{t+h} = m_{\phi}(\tilde{\mu}_{t+h}, \tilde{\Sigma}_{t+h}, C_t^{(1:N)}, \hat{u}_{t+h}^{(1:N)}). \tag{5.2}$$

A limitation of this choice of parameterization is that it assumes independence of the updates between time steps in the rollouts. While this is the case for vanilla `MPPI`, we could potentially learn a better update by incorporating information across time steps. However, if we

parameterize the optimizer with a fully-connected or recurrent neural network, this would result in a large number of parameters to learn, making optimization difficult. Instead, we alter the way in which we sample from the Gaussian policies to remedy this explosion in the dimensionality. As proposed by Bhardwaj et al. [15], we make use of low-discrepancy Halton sequences [90] to generate samples from the Gaussian policies. Normal pseudo-random sequences often result in clusters of sampled points, leaving many regions of the parameter space untouched. Low-discrepancy sequences are a deterministic alternative that alleviate this problem by correlating each point. Moreover, they have a faster rate of convergence for estimating moments of distributions [91]. A D-dimensional Halton sequence $x_1, x_2, \dots, x_N$, in which $x_i \in \mathbb{R}^D$ is generated by

$$x_i = (\phi_{p_1}(i), \dots, \phi_{p_D}(i)), \ \phi_{p_b}(i) = \sum_{j=1}^{\infty} a_j(p_b) p_b^{-j}, \tag{5.3}$$

where $p_1, \dots, p_D$ are consecutive prime numbers and $a_j(p_b) \in \{0, 1, \dots, p_b - 1\}$ such that the condition $i = \sum_{j=1}^{\infty} a_j(p_b) p_b^{j-1}$ holds. The Halton sequence is sampled once at the beginning of the rollout and then transformed using the mean and covariance of the Gaussian policy. Therefore, all sampled control sequences are a deterministic function of the current mean and covariance and can be excluded from the learned update without loss of information.

Rather than optimizing each time step independently, we leverage the structured nature of these samples to learn an update that optimizes the entire trajectory jointly using only cost information. The resulting update can then be written as

$$\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t = m_\phi(\tilde{\boldsymbol{\mu}}_t, \tilde{\boldsymbol{\Sigma}}_t, C_t^{(1:N)}), \tag{5.4}$$

where $\boldsymbol{\mu}_t \triangleq (\mu_t, \mu_{t+1}, \dots, \mu_{t+H-1})$, $\boldsymbol{\Sigma}_t \triangleq (\Sigma_t, \Sigma_{t+1}, \dots, \Sigma_{t+H-1})$, and $\tilde{\boldsymbol{\mu}}_t$ and $\tilde{\boldsymbol{\Sigma}}_t$ are defined similarly. We can think about the Halton sequence as giving us a sense of what the environment and cost landscape is like around the current state. And since the learned optimizer can potentially make better use of its inputs than the expert, we may be able to more effectively use fewer samples while maintaining similar performance. Finally, we note that Equation (5.1) is a convex combination of the previous control parameters and the weighted samples. The

Algorithm 4: DAGGER Training Loop

Input: Initial bootstrapped dataset $\mathcal{D}$, initial policy $\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_1}$, initial state distribution ρ

Parameters: Iterations K, probabilities $\{\beta_k\}_{k=1}^{K}$, rollouts per iteration R

for $k = 1, 2, \ldots, K$ **do**

  Initialize dataset $\mathcal{D}_i \leftarrow \emptyset$

  for r = *1, 2, ...,* R **do**

    Sample initial state $x_1 \sim \rho$

    $\tilde{\boldsymbol{\theta}}_{1:T}$, $C_{1:T}^{(1:M)}$, $\boldsymbol{\theta}_{1:T}^{expert} \leftarrow \text{Rollout}(x_1, \boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_1}, \beta_k)$

    Append $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \{(\tilde{\boldsymbol{\theta}}_{1:T}, C_{1:T}^{(1:M)}, \boldsymbol{\theta}_{1:T}^{expert})\}$

  end

  Aggregate datasets $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$

  Train optimizer parameters ϕ on $\mathcal{D}$

end

hidden state update in a gated recurrent unit (GRU) [92] is of the same form, except the multiplicative factor is also learned. Inspired by this similarity and the fact that multiplicative gating has shown to be effective in non-recurrent architectures [93], we use the update:

$$\begin{aligned} g_t^{\mu}, g_t^{\sigma}, h_t^{\mu}, h_t^{\sigma} &= m_{\phi}(\tilde{\boldsymbol{\mu}}_t, \tilde{\boldsymbol{\Sigma}}_t, C_t^{(1:N)}) \\ \boldsymbol{\mu}_t &= (1 - g_t^{\mu}) \odot \tilde{\boldsymbol{\mu}}_t + g_t^{\mu} \odot h_t^{\mu} \\ \boldsymbol{\Sigma}_t &= (1 - g_t^{\sigma}) \odot \tilde{\boldsymbol{\Sigma}}_t + g_t^{\sigma} \odot h_t^{\sigma}, \end{aligned} \tag{5.5}$$

where $\odot$ is the Hadamard product and g_t^{μ}, g_t^{σ} are passed through a sigmoid to ensure they are between zero and one. In our experiments, we found that this choice of parameterization significantly outperforms a simple fully-connected network.

5.1.2 Imitation Learning for Training the Optimizer

Unlike prior work in learning-to-optimize, we cannot assume that the optimization process itself is differentiable as it occurs online via interactions with the environment. Even in the simulated case, we do not want to assume that everything has been implemented in a

Algorithm 5: DAGGER Rollout Function

Input: State x_1, policy $\boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_1}$, probability β_k

Parameters: Rollout length T, expert samples N, learner samples M

Output: Shifted parameters $\tilde{\boldsymbol{\theta}}_{1:T}$, sample costs $C_{1:T}^{(1:M)}$, expert decisions $\boldsymbol{\theta}_{1:T}^{expert}$

for $t = 1, 2, \ldots, T$ **do**

  Sample controls from policy $\{\hat{\boldsymbol{u}}_t^{(i)}\}_{i=1}^{N} \sim \boldsymbol{\pi}_{\tilde{\boldsymbol{\theta}}_t}$

  Sample $X_t^{(i)}$ from dynamics $\hat{f}$ using $\hat{\boldsymbol{u}}_t^{(i)}, x_t$

  Compute costs $C_t^{(i)} \leftarrow C(X_t^{(i)}, U_t^{(i)})$

  Compute sample weights with Equation (2.12)

  Update $\tilde{\boldsymbol{\theta}}_t$ to $\boldsymbol{\theta}_t^{expert}$ using Equation (5.1)

  Sample $b \sim U(0, 1)$

  if $b \leq \beta_k$ **then**

    Set $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t^{expert}$

  else

    Update $\tilde{\boldsymbol{\theta}}_t$ to $\boldsymbol{\theta}_t^{learn}$ using Equation (5.5)

    Set $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_t^{learn}$

  end

  Sample $u_t \sim \pi_{\theta_t}$ or use mean $u_t \leftarrow \mu_t$

  Apply control to system $x_{t+1} \sim f(x_t, u_t)$

  Shift parameters $\tilde{\boldsymbol{\theta}}_{t+1} = \Phi(\boldsymbol{\theta}_t)$

end

differentiable fashion. We could use reinforcement learning (RL), although it generally has high sample complexity and may be slow to learn. Since we have access to a tuned optimal controller, imitation learning is a promising direction for training the optimizer. As such, our expert is an MPPI controller with unrestricted access to samples. That is, we provide the controller with as many samples as needed to achieve good performance. The learner is also an MPPI controller, but it has access to fewer samples, and the standard update is replaced with the learned optimizer. In our preliminary experiments, we tried using standard

behavioral cloning, in which we collect a dataset of expert demonstrations and train a policy offline via regression. However, this did not work well due to covariate shift between the expert and learner distributions. We used DAgger [94] to perform imitation learning, which is an interactive algorithm that aims to combat issues of covariate shift. The algorithm queries an expert online for corrective labels on learner visited states. We outline the main loop Algorithm 4.

First, we begin by collecting a bootstrap dataset in which only the expert is run. Next, each iteration k of DAgger, we run R rollouts according to Algorithm 5 by sampling some initial state x_1 from a known initial state distribution ρ. During a rollout, at each time step, we apply the controls from the expert with probability β_k and the learner with probability $1 - \beta_k$. The expert is always run in order to provide a corrective target for training the policy at the next iteration. Both the expert and learner controllers use the same trajectory samples, although the learner only receives a subset of them. Only the form of the update applied to the policy distribution is different. Generally, the mixing probabilities β_k are set according to a schedule such that we run the learner more often in later iterations. In our experiments, we set $\beta_k = p^k$ for some $p \in (0, 1)$. After running the rollouts, we collect the warm-started control distribution parameters $\tilde{\boldsymbol{\theta}}_{1:T}$, the trajectory costs $C_{1:T}^{(1:M)}$, and the updated expert control distribution parameters $\boldsymbol{\theta}_{1:T}^{expert}$ into a dataset $\mathcal{D}_i$. While we compute N samples for the expert, we only collect the $M \leq N$ used by the learner. This data is then aggregated into our main dataset $\mathcal{D}$ to train the optimizer.

5.2 Experiments

Implementation Details. In all experiments, our `MPPI` implementation is a modified version of the one developed by Bhardwaj et al. [15]. This implementation uses Halton sequences for generating control sequence samples and smooths the sampled trajectories with B-splines of degree 3. We use a fixed diagonal covariance for the sampling distribution and do not perform covariance adaptation. All hyperparameters were tuned using a grid search, and the optimal number of samples is what the expert controller has access to during data

generation and training. Now, the optimal choice of hyperparameters may be different for a given number of samples. Therefore, for a fair comparison, We tune the `MPPI` hyperparameters separately for each sample count used in our evaluation. Both `MPPI` and the neural networks are implemented in PyTorch [61].

Task Details. We evaluate on simulated tasks:

1. Cartpole: The task is to slide a cart along a rail to swing up the pole attached via an unactuated joint using only actuation from the cart. Both the expert and learner are given access to the true analytical dynamics. The initial position of the cart and pole are randomized at every episode, which lasts 200 time steps. An episode is successful if the pole is swung up with a linear and angular velocity near zero.

2. Franka Reacher: A 7 degree-of-freedom (DOF) Franka Panda robot arm must reach a target goal from a fixed starting pose. The goal is randomly selected at the beginning of each episode. Both the expert and learner use the same kinematic model described in Bhardwaj et al. [15], which is different from the true dynamics of the simulator (Nvidia's Isaac Gym [95]). Each episode lasts for 500 time steps and is successful if the end effector reaches the target.

3. Franka Obstacles: This task is identical to Franka Reacher, except now there are two spherical obstacles placed in the environment which the arm must avoid. The obstacle and the goal positions are randomized at the beginning of each episode, which lasts for 600 time steps. An episode is successful if the end effector reaches the goal while avoiding collisions.

Evaluation. We evaluate the performance of the learned optimizer by varying the number of samples, up to the amount used by the expert. For each sample amount, we compare against a standard MPPI implementation with access to the same number of samples as the learned controller over 30 test rollouts. All test rollouts use a fixed set of start states, goals,

and obstacle locations. This is achieved by setting the random seed value to a pre-defined test seed. Our primary metric for comparison is success rate, which is defined as the percentage of times the task goal was achieved out of all trials. In the Franka Obstacles task, the placement of obstacles is randomized according to a pre-specified distribution. Therefore, this task allows us to evaluate the generalization capability of the learned optimizer to new environments which are drawn from a similar distribution. Additionally, we report statistics of the end effector test trajectories. Specifically, we compute a relative trajectory length and average jerk as the ratio between the statistics for the learned optimizer and baseline MPPI controller. The trajectory length is averaged over all test runs, while the jerk is averaged over only successful test runs.

Training Details. Prior work in learning-to-optimize made use of recurrent architectures which can account for the history of the gradients and optimization process. While we could potentially benefit from such an architecture, we found that a simple multi-layer perceptron (MLP) was sufficient to learn powerful optimizers. As such, in all experiments, the learned optimizer is represented with a two-layer MLP using ReLU activation functions. We use 1024, 2048, and 4096 hidden units per layer for the Cartpole, Franka Reacher, and Franka Obstacles tasks, respectively. To prevent overfitting, all networks are regularized with dropout [96] using a dropout probability of 0.1. We use the Adam optimizer [63] with a learning rate of 10^{-3} for Cartpole and Franka Reacher and 10^{-4} for Franka Obstacles. We normalize the total trajectory costs based on the mean and standard deviation of the training dataset. For all tasks, we bootstrapped the dataset with 1024 trajectories, in which only the expert's action was applied to the system. We ran DAgger for 20 iterations with 128 rollouts per iteration and a mixing probability schedule $\beta_k = 0.8^k$. For each iteration, we train the networks on the aggregated dataset for 1000 epochs with a batch size of 8. The dataset is divided into training and validation splits, where new trajectories collected with DAgger are appended only as training data and the validation set is held constant. After each epoch of training, the network is evaluated on held out validation data.

	# Samples	MPPI	L2O-MPC
Cartpole	8	100.0	100.0
	4	60.00	96.67
	2	10.00	90.00
Franka Reacher	64	100.0	100.0
	32	90.00	100.0
	16	63.33	100.0
	8	20.00	80.00
	4	10.00	63.33
	2	3.333	16.67
Franka Obstacles	512	80.00	80.00
	256	76.67	80.00
	128	73.33	76.67
	64	63.33	76.67
	32	33.33	66.67
	16	6.667	46.67

Table 5.1: Success rate across all tasks for a varying number of samples.

# Samples	Length	Avg. Jerk
512	1.091	1.008
256	1.030	1.023
128	0.975	1.049
64	0.916	1.077
32	0.858	1.104
16	0.854	1.527

Table 5.2: Trajectory statistics for the Franka Obstacles task.

5.3 Results

We report the success rate for all tasks in Table 5.1 and refer to standard MPPI by **MPPI** and MPC with the learned optimizer by **L2O-MPC**. Success rate is computed for each task based on the criteria discussed in Section 5.2. We can see that the performance of **MPPI** quickly drops off as the number of control sequence samples is reduced. For Cartpole, the performance of **L2O-MPC** remains fairly consistent even with a lower number of samples. While the performance drop is more pronounced in the Franka experiments, **L2O-MPC** still consistently matches or outperforms **MPPI** at each sample amount. In Franka Reacher,

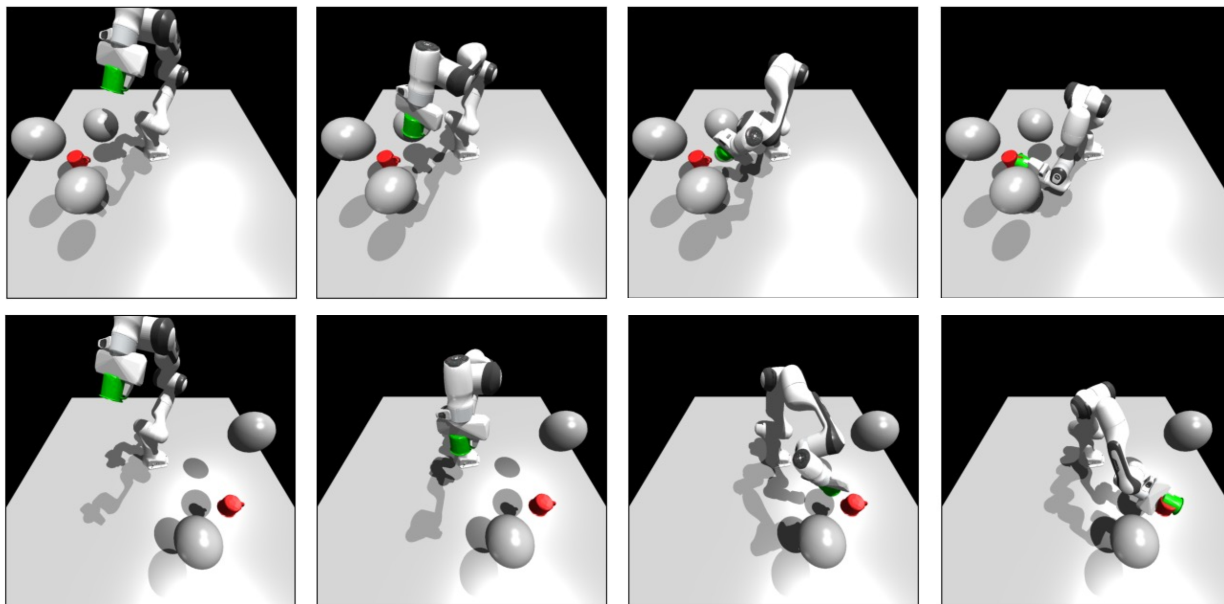


Figure 5.1: Example trajectories of the Franka Obstacles task, in which the Franka arm end effector (green) is tasked with reaching the goal (red) while avoiding obstacles. The top row is a novel environment with three obstacles, while the bottom row is a test environment.

L2O-MPC is able to withstand a 4× decrease in samples while still achieving an 100% success rate. Similarly, in Franka Obstacles, **L2O-MPC** only incurs a 4% performance decrease under an 8× decrease in samples.

Figure 5.1 illustrates two different example trajectories of the Franka arm avoiding different amounts of obstacles. These same environments are depicted in Figure 5.2, where we show end effector trajectories under **MPPI** with 512 samples, **MPPI** with 16 samples, and **L2O-MPC** with 16 samples. We can see that **MPPI** with 16 samples quickly diverges and is unable to reach the goal. On the other hand, **L2O-MPC** with 16 samples is able to better make use of the samples and still reach the goal while avoiding all obstacles. Moreover, the **L2O-MPC** controller was trained only on environments that contain two obstacles. Therefore, these results also indicate that the learned controller may generalize to novel environments on which it was not trained.

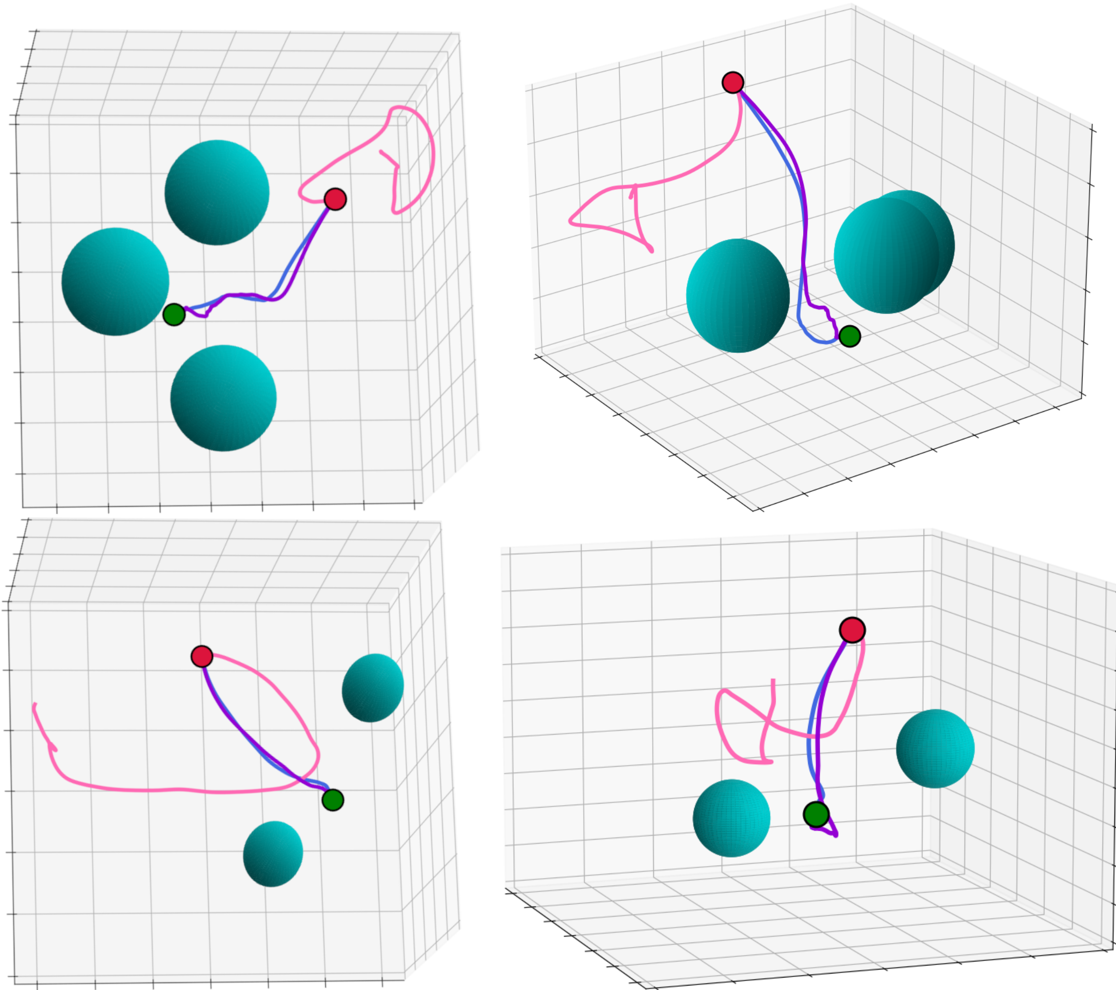


Figure 5.2: Trajectories of the Franka arm end effector when controlled by **MPPI** with 512 samples (blue), **MPPI** with 16 samples (pink), and **L2O-MPC** with 16 samples (purple) to move from the starting position (red) to the goal (green) while avoiding obstacles (cyan). Plots in the same row are from the same environment but viewed from differing perspectives.

Qualitatively, the **L2O-MPC** trajectories appear to be slightly more jittery than the **MPPI** expert. In Table 5.2, we provide the average relative jerk between **L2O-MPC** and **MPPI** for successful test runs at different sample counts. Indeed, we see that the **L2O-MPC** trajectories are less smooth than those of **MPPI**, and this effect is exacerbated at lower sample counts. Additionally, we provide the average relative trajectory length across all test environments. With more samples, **L2O-MPC** has slightly longer trajectories than **MPPI**, indicating that it is slower at reaching the goal. However, when given access to fewer samples, **L2O-MPC** consistently has shorter trajectories, as it more often reaches the goal. Therefore, while **L2O-MPC** is often jerkier and sometimes slower than the expert with full samples, it succeeds more often in achieving the desired objective in a timely fashion than **MPPI** given the same number of samples.

5.4 Discussion

We presented a method for improving upon standard sampling-based MPC algorithms by learning a better update rule. This provides a novel way to incorporate learning into model-based control algorithms, which is orthogonal to the standard approaches of learning or fine-tuning the dynamics model and/or cost function. We contend with noisy gradients by learning how to more effectively update the control distribution. By using structured sampling strategies, we are able to provide more information to the learned update and better utilize fewer samples. We show through empirical evaluations that our learned controllers remain competitive or outperform a baseline MPPI controller with access to the same number of samples. This demonstrates the viability of the learning-to-optimize framework in the context of control, opening the door for a variety of techniques to be applied to improving the performance of optimization-based controllers and planners. While we leveraged imitation learning to train the optimizers, this is just one possible option and an interesting direction for future work is to explore using reinforcement learning to see if it can outperform the expert and model-free methods. Since performance of sampling-based methods relies so heavily on thorough exploration of the sample space, another possible avenue is to learn how to generate

better samples in addition to better updates. Finally, the proposed technique is not limited to sampling-based MPC, and could be applied to other optimization-based controllers.

Chapter 6

LEARNING SAMPLING DISTRIBUTIONS FOR MODEL PREDICTIVE CONTROL

Another way of improving the optimization process in sampling-based MPC is to alter the sampling distribution. They are often kept simple, e.g. a Gaussian, such that we can efficiently sample and tractably update its parameters. However, this also has drawbacks: without much control over the distribution form, samples often lie in high-cost regions, hindering performance. This can be particularly problematic in complex environments with sparse costs or rewards, as a poorly parameterized distribution may hinder efficient exploration, leading the system into bad local optima. A side effect is that we often require many samples to accomplish the objective, increasing computational requirements. There have been extensions which target more complex distributions, such as Gaussian mixture models [97] and a particle method based on Stein variational gradient descent (SVGD) [98]. However, there is a large amount of structure in the environment that these methods fail to exploit. Instead, an alternative approach is to *learn* a sampling distribution which can leverage structure.

However, learning MPC sampling distributions generally requires differentiability of the dynamics and cost function [46, 56]. Power et al. [58, 59] circumvent this issue by leveraging normalizing flows (NFs) [99–101], which have a tractable log-likelihood. This property allows them to learn flexible distributions by directly optimizing the MPC cost without requiring differentiability via the likelihood-ratio gradient. However, a limitation of their approach is that all online updates to the distribution and warm-starting between time steps occur entirely in the control space, leaving the latent distribution fixed. This forces them to apply heuristics to generate samples by combining those from the learned distribution with Gaussian perturbations to the current control-space mean. These restrictions prevent us from fully

taking advantage of the learned distribution and potentially throws away useful information. Additionally, their approach does not allow for the incorporation of control constraints directly into the sampling distribution, which is important for real-world robots.

In this chapter, we discuss our work in Sacks et al. [102], in which we propose to alter the optimization machinery to operate entirely in the latent space. As the NF latent space follows a simple distribution, it remains feasible to perform MPC updates in this learned space and update the latent distribution online. Specifically, during an episode, the parameters of the latent distribution are updated with MPC while those of the NF remain fixed. Then during training, after each episode, the parameters of the NF are updated. We can frame this setup as a bi-level optimization problem [103] and derive a method for computing an approximate gradient through the latent MPC update to learn the flow. This involves treating MPC as a recurrent network, where the control distribution acts as a form of memory, and unrolling the computation to train with backpropagation-through-time (BPTT). However, it is no longer clear how to warm-start between time steps because there is no clear delineation of time in the latent space. Moreover, the usual method of warm-starting, which simply shifts the current plan forward in time, may be sub-optimal. Therefore, we additionally learn a shift model, which performs all warm-starting operations in the latent space. Finally, we show how to alter the NF architecture to incorporate box constraints on the sampled controls.

6.1 Representation of the Learned Distribution

Instead of using uninformed sampling distributions, learned distributions can potentially exploit structure in the environment to draw samples which are more likely to be collision-free and close to optimal. However, such learned distributions must be sufficiently expressive in order to better capture near-optimal, potentially multimodal, behavior. They must also be parameterized such that it is tractable to sample from and update online. If the distribution has a large number of parameters, the number of samples required to efficiently update them online may be computationally infeasible. And ideally, the form of our distribution would be such that we could find a closed-form update.

One path towards meeting these criteria is to maintain a simple latent distribution from which we can sample, and then learn a transformation of the samples which maps them to a more complex distribution. During training, we learn the parameters of this transformation, which can be conditioned on problem-specific information, such as the starting and goal configurations of the robot and obstacle placements. However, when executing the policy during an episode, the parameters of this learned transformation remain fixed, and instead, we update the parameters of the latent distribution. Concretely, we consider learning a distribution $\pi_{\theta,\lambda}$ defined implicitly:

$$\hat{\boldsymbol{z}}_t \sim p_\theta(\cdot), \quad \hat{\boldsymbol{u}}_t = h_\lambda(\hat{\boldsymbol{z}}_t; c) \tag{6.1}$$

where $\hat{\boldsymbol{z}}_t \triangleq (\hat{z}_t, \hat{z}_{t+1}, \cdots, \hat{z}_{t+H-1})$, c is a context variable describing the relevant information of the environment, p_θ is the latent distribution with parameters θ, and h_λ is the learned conditional transformation with parameters λ. Moving forward, we assume that both $\hat{\boldsymbol{z}}_t$ and $\hat{\boldsymbol{u}}_t$ are stacked as vectors in $\mathbb{R}^{MH}$. If p_θ is a Gaussian factorized as in Equation (2.8) and we assume that h_λ is invertible, we can prove that follow theorem.

Theorem 6.1.1. *Consider the optimization problem*

$$\theta_t = \arg\min_{\theta \in \Theta} \langle \gamma_t g_t, \theta \rangle + D_{KL}(\pi_{\theta,\lambda} || \pi_{\tilde{\theta}_t,\lambda}) \tag{6.2}$$

where we define

$$\pi_{\theta,\lambda}(\hat{\boldsymbol{u}}|c) = p_\theta(h_\lambda(\hat{\boldsymbol{u}}; c)) \left| det \, \frac{\partial h_\lambda}{\partial \hat{\boldsymbol{u}}} \right| \tag{6.3}$$

and h_λ is an invertible, deterministic transformation, p_θ a Gaussian factorized as in Equation (2.8), and θ is the natural parameters of the Gaussian. Then the corresponding update to the mean of the latent Gaussian is given by:

$$\boldsymbol{\mu}_t = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t + \gamma_t^\mu \frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)} h_\lambda(\hat{\boldsymbol{u}}_t; c)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]} = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t + \gamma_t^\mu \frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\hat{\boldsymbol{z}}_t\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]} \tag{6.4}$$

Proof. First, we note that

$$g_t = \nabla \hat{J}(\tilde{\theta}; x_t) = -\frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\nabla_{\tilde{\theta}} \log \pi_{\tilde{\theta},\lambda}(\hat{\boldsymbol{u}}_t)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]}, \tag{6.5}$$

and that

$$\log \pi_{\tilde{\theta},\lambda}(\hat{\boldsymbol{u}}|c) = \log p_{\tilde{\theta}}(h_\lambda(\hat{\boldsymbol{u}}; c)) + \log\left(\left|\det \frac{\partial h_\lambda}{\partial \hat{\boldsymbol{u}}}\right|\right). \tag{6.6}$$

Therefore, when computing the gradient of Equation (6.6) with respect to $\tilde{\theta}$, we can drop the log-determinant term as it does not depend on $\tilde{\theta}$. As such, we are left with the gradient of the latent Gaussian with respect to its natural parameters, or $\nabla_{\tilde{\theta}} \log \pi_{\tilde{\theta},\lambda}(\hat{\boldsymbol{u}}|c) = \nabla_{\tilde{\theta}} \log p_{\tilde{\theta}}(h_\lambda(\hat{\boldsymbol{u}}; c))$. Taking this gradient of the log-likelihood term with respect to the natural parameters, we have:

$$g_t = -\frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\left(\phi(h_\lambda(\hat{\boldsymbol{u}}_t; c)) - \tilde{\phi}_t\right)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]}, \tag{6.7}$$

where $\tilde{\phi}_t$ is the expectation parameter corresponding to natural parameter $\tilde{\theta}_t$ and $\phi(\cdot)$ is the sufficient statistics of the latent distribution. We rewrite the expectations in terms of $p_{\tilde{\theta}}(\cdot)$:

$$g_t = -\frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c)))}\left(\phi(\hat{\boldsymbol{z}}_t) - \tilde{\phi}_t\right)\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]}, \tag{6.8}$$

Next, looking at the KL divergence term, we can write:

$$\begin{aligned}
D_{KL}(\pi_{\theta,\lambda}||\pi_{\tilde{\theta}_t,\lambda}) &= \mathbb{E}_{\pi_{\theta,\lambda}}\left[\log \frac{\pi_{\theta,\lambda}(\hat{\boldsymbol{u}})}{\pi_{\tilde{\theta}_t,\lambda}(\hat{\boldsymbol{u}})}\right] \\
&= \mathbb{E}_{\pi_{\theta,\lambda}}\left[\log \frac{p_\theta(h_\lambda(\hat{\boldsymbol{u}}; c))}{p_{\tilde{\theta}}(h_\lambda(\hat{\boldsymbol{u}}; c))}\frac{\cancel{\left|\det \frac{\partial h_\lambda}{\partial \hat{\boldsymbol{u}}}\right|}}{\cancel{\left|\det \frac{\partial h_\lambda}{\partial \hat{\boldsymbol{u}}}\right|}}\right] \\
&= \mathbb{E}_{p_\theta}\left[\log \frac{p_\theta(\hat{\boldsymbol{z}})}{p_{\tilde{\theta}}(\hat{\boldsymbol{z}})}\right] \\
&= D_{KL}(p_\theta||p_{\tilde{\theta}_t}),
\end{aligned} \tag{6.9}$$

Then, using Proposition 1 from Wagener et al. [19], we can write the update rule as

$$\phi_t = (1-\gamma_t)\tilde{\phi}_t + \gamma_t \frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c)))}\phi(\hat{\boldsymbol{z}}_t)\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]} \tag{6.10}$$

And when the sufficient statistic is $\phi(\hat{\boldsymbol{z}}_t) = (\hat{\boldsymbol{z}}_t, \hat{\boldsymbol{z}}_t\hat{\boldsymbol{z}}_t^T)$, then we arrive at the usual MPPI update for the mean, but now defined in terms of the latent samples:

$$\boldsymbol{\mu}_t = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t + \gamma_t^\mu \frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\hat{\boldsymbol{z}}_t\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]} \tag{6.11}$$

which we can equivalently rewrite in terms of $\pi_{\tilde{\theta},\lambda}$ as:

$$\boldsymbol{\mu}_t = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t + \gamma_t^\mu \frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)}h_\lambda(\hat{\boldsymbol{u}}_t;c)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)}\right]} \tag{6.12}$$

□

Theorem 6.1.1 says that the corresponding DMD update to the latent mean is simply Equation (2.12), except that we replace the controls samples with the latents.

6.2 Formulating the Learning Problem

Learning the distribution $\pi_{\theta,\lambda}$ amounts to solving a bi-level optimization problem [103], in which one optimization problem is nested in another. The lower-level optimization problem involves updating the latent distribution parameters at each time step, θ_t, by minimizing the expected cost with DMD. The upper-level optimization problem consists of learning λ such that MPC performs well across a number of different environments. To formalize this, first consider that we have some distribution of environments $c \sim \mathcal{C}(\cdot)$ over which we wish MPC to perform well. For each environment, our system has some conditional initial state distribution $x_0 \sim \rho(\cdot|c)$. The objective we wish to minimize is then

$$\ell(\boldsymbol{\theta}, \lambda; c) = \mathbb{E}_{\boldsymbol{\pi_{\theta,\lambda}},\rho,f}\left[\sum_{t=0}^{T-1}\hat{J}(\theta_t, \lambda; x_t, c)\right] \tag{6.13}$$

where $\boldsymbol{\theta} = (\theta_0, \theta_1, \cdots, \theta_{T-1})$ and our cost statistic, $\hat{J}$, now depends on λ and c as well. This objective measures the expected performance of the intermediate plans produced by MPC along the T steps of the episode. Our desired bi-level optimization problem is:

$$\min_\lambda \; \mathbb{E}_{\mathcal{C}}\Big[\ell(\boldsymbol{\theta}(\lambda), \lambda; c)\Big] \quad \text{s.t. } \boldsymbol{\theta}(\lambda) \approx_\lambda \arg\min_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}, \lambda; c) \tag{6.14}$$

where $\approx_\lambda$ indicates that we approximate the solution of the optimization problem with an iterative algorithm that may also be parameterized by λ, as the exact minimizer is not available in closed form. Moreover, the notation $\boldsymbol{\theta}(\lambda)$ indicates the dependence of the lower-level solution on the upper-level parameters. Backpropagating through the solution of an iterative solver has proven to be a successful technique for bi-level optimization on a variety of hyperparameter optimization and meta-learning problems [104]. In our case, we solve the lower-level problem with DMD, where we also parameterize the shift model, $\Phi_\lambda(\cdot; c)$, making it a learnable component and conditioned on c.

The normal shift model in MPC simply shifts the control sequence forward one time step and appends a zero or random control at the end. However, because we are performing this update in the latent space, there is no clear delineation between time steps of the latent controls, as they are coupled according to the learned transformation. Therefore, there is no way to easily perform the equivalent shift operation in the latent space. As such, we instead learn this shift model along with the transformation. Besides, the standard approach described above may not be optimal. By learning it, we may be able to further improve performance. Performance hinges greatly on the quality of the shift model since we only run one iteration of the DMD update.

6.3 Parameterizing with Normalizing Flows

In order to optimize the upper-level objective in Equation (6.14) with respect to λ, we need to be able to compute the density $\pi_{\theta,\lambda}$ directly. Therefore, we choose to represent h_λ with a normalizing flow (NF) [99–101], which explicitly learns the density by defining an invertible transformation that maps latent variables to observed data. Generally, we compose a series of component flows together, i.e. $h_\lambda = h_{\lambda_K} \circ h_{\lambda_{K-1}} \circ \cdots \circ h_{\lambda_1}$, which define a series of intermediate variables $\hat{\boldsymbol{y}}_0, \ldots, \hat{\boldsymbol{y}}_{K-1}, \hat{\boldsymbol{y}}_K$, with $\hat{\boldsymbol{y}}_0 = \hat{\boldsymbol{z}}$ and $\hat{\boldsymbol{y}}_K = \hat{\boldsymbol{u}}$. The log-likelihood of the composed flow is given by:

$$\log \pi_{\theta,\lambda}(\hat{\boldsymbol{u}}|c) = \log p_\theta(\hat{\boldsymbol{z}}) - \sum_{i=1}^{K} \log \left| \det \frac{\partial \hat{\boldsymbol{y}}_i}{\partial \hat{\boldsymbol{y}}_{i-1}} \right|. \tag{6.15}$$

In this work, we make use of the affine coupling layer proposed by Dinh et al. [101] as part of the real non-volume-preserving (RealNVP) flow. The core idea is to split the input $\hat{\boldsymbol{u}}$ into two partitions $\hat{\boldsymbol{u}} = (\hat{\boldsymbol{u}}_{I_1}, \hat{\boldsymbol{u}}_{I_2})$, where I_1 and I_2 are a partition of $[1, MH]$, and apply

$$\hat{\boldsymbol{y}}_{I_1} = \hat{\boldsymbol{u}}_{I_1}, \quad \hat{\boldsymbol{y}}_{I_2} = \hat{\boldsymbol{u}}_{I_2} \odot \exp s_\lambda(\hat{\boldsymbol{u}}_{I_1}, c) + t_\lambda(\hat{\boldsymbol{u}}_{I_1}, c), \tag{6.16}$$

where s_λ and t_λ are the scale and translation terms, which are represented with arbitrary neural networks, and $\odot$ is the Hadamard product. This makes computing the log-determinant term in Equation (6.15) and inverting the flow fast and efficient.

Now, in robotics, we often have lower and upper limits on the controls. These are usually enforced in sampling-based MPC by either clamping the control samples or passing them through a scaled sigmoid. However, instead of enforcing the constraints heuristically after sampling, we learn a constrained sampling distribution directly. Since the sigmoid function is invertible and has a tractable log-determinant, we can simply append one after h_{λ_K} in the NF and scale it by the control limits. That is, we wish to use a sigmoid layer in our normalizing flow to constrain our control sample $\hat{\boldsymbol{u}}$ such that each control along the horizon is between $\underline{u}$ and $\bar{u}$. Since the sigmoid function is invertible, if we append a sigmoid layer at the end of our flow, we have that

$$\hat{\boldsymbol{u}} = w\sigma(\hat{\boldsymbol{y}}_{K-1}) + b \quad \Longleftrightarrow \quad \hat{\boldsymbol{y}}_{K-1} = \sigma^{-1}\Big(\frac{\hat{\boldsymbol{u}} - b}{w}\Big) = \log\Big(\frac{\hat{\boldsymbol{u}} - b}{w - \hat{\boldsymbol{u}} + b}\Big), \tag{6.17}$$

where $w = \bar{u} - \underline{u}$ and $b = \underline{u}$, the sigmoid and logit functions are applied element-wise, and the scaling and translation are broadcasted to each element of the vector $\hat{\boldsymbol{u}}$. The derivative of the forward transformation is given by:

$$\begin{aligned}\frac{\partial}{\partial x}\Big(w\sigma(x) + b\Big) &= w\sigma(x)(1 - \sigma(x)) \\ &= \frac{w}{1 + e^{-x}}\Big(1 - \frac{1}{1 + e^{-x}}\Big) \\ &= \frac{w}{(1 + e^{-x})(1 + e^{x})}.\end{aligned} \tag{6.18}$$

Since the sigmoid is applied element-wise, it has a diagonal Jacobian, the log-determinant of

which is simply the sum of the log of its diagonal terms:

$$\log \left| \det \frac{\partial \hat{\boldsymbol{u}}}{\partial \hat{\boldsymbol{y}}_{K-1}} \right| = \sum_{i=1}^{MH} \log(w) - \log(1 + e^{-\hat{u}_i}) - \log(1 + e^{\hat{u}_i}), \tag{6.19}$$

where we can implement the last two terms with the Softplus activation function. In the reverse direction, we have that:

$$\frac{\partial}{\partial x}\left(\sigma^{-1}\left(\frac{x-b}{w}\right)\right) = \frac{1}{x-b} + \frac{1}{w-x-b} \tag{6.20}$$

Therefore, the log-determinant is given by:

$$\log \left| \det \frac{\partial \hat{\boldsymbol{y}}_{K-1}}{\partial \hat{\boldsymbol{u}}} \right| = -\sum_{i=1}^{MH} \Big(\log(\hat{u}_i - b) + \log(w - \hat{u}_i - b) \Big), \tag{6.21}$$

As such, computing the inverse and the log-determinant terms of the Jacobian is efficient and fast in both directions, adding minimal overhead to the flow. This ensures that control constraints are satisfied by design and taken into account while learning the distribution.

6.4 Training the Sampling Distributions

Computing gradients through the upper-level objective is not straightforward, as both the expectation and the inner terms of Equation (6.13) depend on λ. Therefore, the state distribution depends on the NF and latent shift model. One way around this issue is to consider a modified objective at each batch d:

$$\ell_d(\boldsymbol{\theta}, \lambda; c) = \mathbb{E}_{\boldsymbol{\pi}_{\boldsymbol{\theta},\lambda_d},\rho,f}\left[\sum_{t=0}^{T-1} \hat{J}(\theta_t, \lambda; x_t, c)\right], \tag{6.22}$$

which fixes the outer expectation to be with respect to the current policy. Intuitively, this choice trains the NF to optimize the MPC cost function under the state distribution resulting from the current policy $\boldsymbol{\pi}_{\boldsymbol{\theta},\lambda_d}$. This effectively ignores the dependency of the action chosen to interact with the true environment and the learned parameters. We then update the outer expectation distribution at each episode, overcoming the covariate shift problem that would otherwise arise.

Now, we only have to focus on computing the gradient $\nabla_\lambda \hat{J}(\theta_t(\lambda), \lambda; x_t, c)|_{\lambda=\lambda_d}$ for each time step, which can be computed similar to Equation (4.9) and approximated with Monte Carlo sampling:

$$\nabla \hat{J}(\theta_t(\lambda), \lambda; x_t, c) \approx -\sum_{i=1}^{N} w_i \nabla_\lambda \log \pi_{\theta_t(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c), \tag{6.23}$$

where the weights w_i are defined according to Equation (2.12). This can be implemented in autodifferentiation software by differentiating the objective function

$$\hat{J}(\theta_t(\lambda), \lambda; x_t, c) = -\sum_{i=1}^{N} w_i \log \pi_{\theta_t(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c), \tag{6.24}$$

where we do not compute gradients through the computation of the weights w_i. The log-likelihood is given by Equation (6.15), the gradient of which involves computing the backwards pass through the network h_λ. However, we also have to consider the dependence of the latent distribution parameters $\boldsymbol{\theta}(\lambda)$ on λ. Therefore, we must backpropagate through the MPC update, which has the form:

$$\boldsymbol{\mu}_t(\lambda) = (1-\gamma_t)\tilde{\boldsymbol{\mu}}_t(\lambda) + \gamma_t \Delta\boldsymbol{\mu}_t, \quad \Delta\boldsymbol{\mu}_t = \frac{\mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)} h_\lambda(\hat{\boldsymbol{u}}_t; c)\right]}{\mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]} \tag{6.25}$$

where the previous shifted mean $\tilde{\boldsymbol{\mu}}_t(\lambda)$ is given by the learned latent shift model. Note that we have rewritten the expectations in terms of the control distribution, rather than the latent distribution. This is necessary in order to derive the following approximate gradient without requiring differentiability. To compute the gradient of Equation (6.25), we must approximate the gradient of $\Delta\boldsymbol{\mu}_t$ with respect to λ, which we show below.

Theorem 6.4.1. *Let the MPPI update of the latent mean be given by*

$$\boldsymbol{\mu}_t(\lambda) = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t(\lambda) + \gamma_t^\mu \Delta\boldsymbol{\mu}_t, \qquad \Delta\boldsymbol{\mu}_t = \frac{\mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)} h_\lambda(\hat{\boldsymbol{u}}_t; c)\right]}{\mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]}. \tag{6.26}$$

Then the gradient of $\Delta\boldsymbol{\mu}_t$ with respect to λ can be approximated as

$$\frac{\partial \Delta\boldsymbol{\mu}_t}{\partial\lambda} \approx M_1 - M_2 M_3 \tag{6.27}$$

where

$$M_1 = \sum_{i=1}^{N} w_i \Big[\nabla_\lambda h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) + h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c)\Big],$$
$$M_2 = \sum_{i=1}^{N} w_i h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c), \quad M_3 = \sum_{i=1}^{N} w_i \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c). \tag{6.28}$$

Proof. First, we rewrite $\Delta\boldsymbol{\mu}_t = \frac{N(\lambda)}{D(\lambda)}$, where

$$N(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda}, \hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)} h_\lambda(\hat{\boldsymbol{u}}_t; c)\Big] \qquad D(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda}, \hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)}\Big]. \tag{6.29}$$

Then by the quotient rule of calculus, we have

$$\frac{\partial \Delta\boldsymbol{\mu}_t}{\partial \lambda} = \frac{\nabla_\lambda N(\lambda)}{D(\lambda)} - \frac{N(\lambda)}{D(\lambda)} \frac{\nabla_\lambda D(\lambda)}{D(\lambda)}. \tag{6.30}$$

We can compute each of these individual terms using the likelihood ratio gradients

$$\nabla_\lambda N(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda}, \hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)} \Big(\nabla_\lambda h_\lambda(\hat{\boldsymbol{u}}_t; c) + h_\lambda(\hat{\boldsymbol{u}}_t; c) \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c)\Big)\Big] \tag{6.31}$$

$$\nabla_\lambda D(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda}, \hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)} \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c)\Big] \tag{6.32}$$

Since each of the terms that make up our gradient in Equation (6.30) are divided by $D(\lambda)$, when we approximate them with Monte Carlo sampling, we can actually write them in terms of the same weights used by MPPI in the forward pass:

$$\frac{\nabla_\lambda N(\lambda)}{D(\lambda)} = \sum_{i=1}^{N} w_i \Big[\nabla_\lambda h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) + h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c)\Big] = M_1 \tag{6.33a}$$

$$\frac{N(\lambda)}{D(\lambda)} = \sum_{i=1}^{N} w_i h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) = M_2 \tag{6.33b}$$

$$\frac{\nabla_\lambda D(\lambda)}{D(\lambda)} = \sum_{i=1}^{N} w_i \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c) = M_3 \tag{6.33c}$$

□

we can compute as $\frac{\partial \Delta\boldsymbol{\mu}_t}{\partial \lambda} \approx M_1 - M_2 M_3$, where we define:

$$M_1 = \sum_{i=1}^{N} w_i \Big[\nabla_\lambda h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) + h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c)\Big],$$
$$M_2 = \sum_{i=1}^{N} w_i h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c), \quad M_3 = \sum_{i=1}^{N} w_i \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)} | c). \tag{6.34}$$

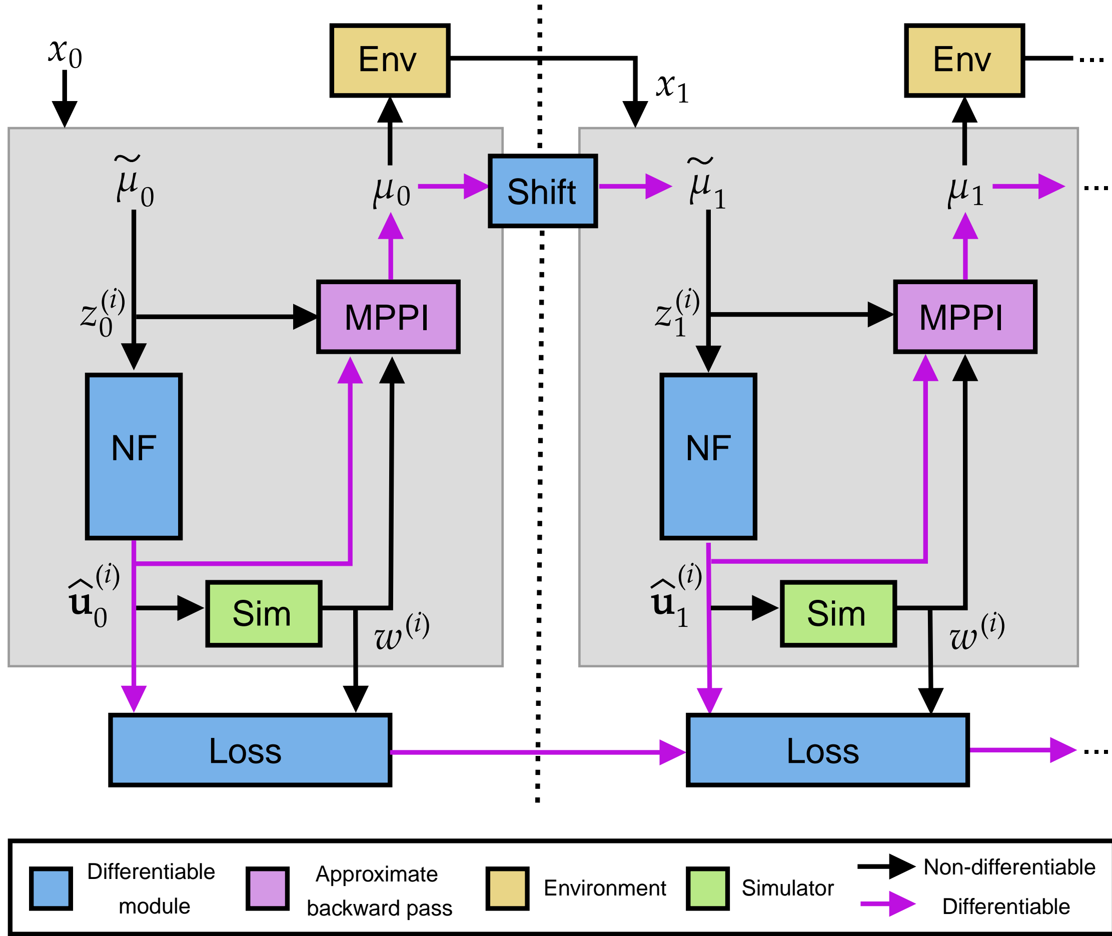


Figure 6.1: Computational graph of an episode which illustrates the interaction between the normalizing flow (NF), learned latent shift model (Shift), the MPPI update (MPPI) of the latent mean, the simulator (Sim), and the environment (Env).

Note that computing the gradients $\nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c)$ as in Theorem 6.4.1 will also require us to backpropagate through the shift model due to the dependence of $\tilde{\theta}$ on λ. Therefore, even when the step size is set to one, i.e. $\gamma_t = 1$, we introduce a form of recurrence.

We visualize the computational graph of a single episode in Figure 6.1. The normalizing flow (NF), shift operator (Shift), and loss computation (Loss) are colored blue, indicating that they are differentiable modules. The MPPI update (MPPI) is colored in purple, to indicate that we are approximating the gradient as above, which can be implemented as a custom

backwards pass in autodifferentiation software. All of the paths of the graph through which the gradients flow during the backwards pass are colored in purple. The simulator (Sim), colored in green, is solely used to compute the weights for the MPPI update, which are reused in the backwards pass. At each time step, the updated mean $\boldsymbol{\mu}_t$ defines a latent Gaussian, which is used to generate controls applied to the actual environment (Env), denoted by the yellow box. As previously stated, we do not differentiate through this interaction, effectively ignoring the dependency of the actions on the parameters λ. Note that both the latent variables and controls are being passed to the MPPI module. This is because during the forward pass, we can simply take the weighted sum of latent variables. However, during the backward pass, we have to re-run the network with the controls to compute the approximate gradients, as discussed in Section 6.4.

6.5 Experiments

In all experiments, we denote our proposed approach as `NFMPC`, the baseline MPPI implementation as `MPPI`, and the method by Power et al. [58, 59] as `FlowMPPI`. We evaluate on a fixed set of environments, which includes start states, goal locations, and obstacle placements, and run 32 rollouts for each sample amount. Our primary metrics for comparison are the success rate, defined as the percentage of times the task goal was achieved, and the average cost of trajectories which successfully completed the task.

6.5.1 Controller Details

We use a modified version of the MPPI implementation by Bhardwaj et al. [15], which is implemented in PyTorch [61]. For the MPPI baseline, we perform covariance adaptation of the full covariance matrix across the horizon and control dimensions. The initial covariance matrix is always an identity matrix scaled by an initial covariance scalar hyperparameter σ^2. Additionally, this implementation uses Halton sequences [90] for generating control sequence samples and smooths the sampled trajectories with n-degree B-splines in some tasks. When B-splines are used, we sample the Halton sequence once at the beginning of a rollout and

Table 6.1: Controller Hyperparameters

	Parameter	**Environment** PNGrid	PNRand	Franka
	Horizon (H)	32	64	32
	Temperature (β)	10^{-32}	10^{-32}	10^{-4}
MPPI	Init. cov. (σ^2)	10	100	0.1
	Step size (γ)	0.7	1	1
	Spline knots (n)	None	None	4
FlowMPPI	Init. cov. (σ^2)	10	10	0.1
	Latent cov.	1	1	0.1
	Latent mean penalty (λ)	10^{-4}	10^{-3}	1
NFMPC	Latent cov.	1	1	0.1

then transform it using the mean and covariance of the Gaussian distribution. Additionally, we ensure that the mean of the Gaussian is always in the set of samples. All hyperparameters of the controllers were chosen via a grid search and the final choices for each task are listed in Table 6.1. When it improves performance, we warm-start the controllers prior to the first time step by running the MPC update for 100 iterations to ensure convergence. Additionally, we normalize the total trajectory costs prior to computing the softmax weights.

In general, we use the same settings for `NFMPC` and `FlowMPPI` that were found in this grid search. However, we do not performance covariance adaptation on the latent Gaussian and assume the flow learns how to adjust sample spread as needed. Additionally, we take the previous mean in the control space, shift it forward, and add it to the set of control samples at the next time steps. While the learned latent shift model handles this well in most cases, we found adding this sample sped up training and slightly improved performance. We always use Halton sequences to sample from the latent Gaussian, but never use B-splines. For

`FlowMPPI`, we always use half the samples for sampling from the NF and half for the Gaussian perturbations on the current control-space mean. We do perform covariance adaptation on the perturbation Gaussian and re-tune its initial covariance. Finally, we have the additional λ parameter, which penalizes the latent Gaussian samples from deviating too much from the projection of the current control-space mean into the latent space.

6.5.2 Planar Robot Navigation

We first apply `NFMPC` to a planar navigation environment, with a state space of $x_t \in \mathbb{R}^4$, which consists of the robot's 2D position, (p_x, p_y), and velocity, (v_x, v_y), and a control space of $u_t \in \mathbb{R}^2$, which are the robot's 2D acceleration commands. The robot has double-integrator dynamics with additive Gaussian noise on the controls, as described by the following equations:

$$\begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}_{t+1} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}_{t} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} (u_t + w_t), \quad w_t \sim \mathcal{N}(0, \sigma \mathrm{I}), \tag{6.35}$$

where we set $\Delta t = 0.1$ and $\sigma = 1$. Additionally, we added acceleration limits of $\underline{u} = -10$ and $\bar{u} = 10$ for both directions. The cost function consists of the Euclidean distance to the goal, a signed-distance field representation of the obstacles, and a term which encourages the robot to stay within the bounds of the map, and a quadratic control penalty:

$$c(x, u) = w_{goal}||x - x_g||_2^2 + w_{bound} c_{bound}(x_t) + w_{coll} \mathrm{SDF}(p_x, p_y) + w_{ctrl}||u||_2^2, \tag{6.36}$$

where we define the map bound cost as:

$$c_{bound}(x) = \sum_{i \in (x,y)} \mathbb{I}[(p_i > \bar{p}_i) \ || \ (p_i < \underline{p}_i)] \min\left((p_i - \bar{p}_i)^2, (p_i - \underline{p}_i)^2\right). \tag{6.37}$$

In the above equations, x_g is the goal state, $\bar{p}_i$ and $\underline{p}_i$ are the upper and lower bounds of the map for each coordinate, and $\mathrm{SDF}(\cdot, \cdot)$ indexes an image which represents the signed-distance field. We mainly focus on the PNRandDyn task, which involves steering the robot towards

a goal position while avoiding eight dynamic obstacles. While the obstacles drift randomly in the environment with Gaussian steps, their positions are clipped to be within map bounds. We do not update their position if the perturbation would bring it too close to the robot or goal location to prevent collisions which the robot cannot react to in time to avoid. An episode lasts for 200 time steps and is considered successful if the agent reaches the goal without colliding into any obstacles. We also consider a static version of this environment (PNRand), which simply places the eight obstacles randomly in the environment, and a version which arranges the obstacles in a fixed grid (PNGrid).

6.5.3 Franka Panda Arm

Next, we apply `NFMPC` to a Franka Panda arm environment, which defines the robot state in joint space with $x_t \in \mathbb{R}^{21}$, consisting of each joint's angle θ_i, angular velocity $\dot{\theta}_i$, and angular acceleration $\ddot{\theta}_i$. Its control space is $u_t \in \mathbb{R}^7$, which are the angular acceleration commands for each joint. The dynamics are deterministic and implemented by the Nvidia Isaac Gym simulator [95]. However, the MPC controllers use a simpler kinematic model defined by Bhardwaj et al. [15], which is implemented in a batch fashion by leveraging its linearity:

$$\ddot{\boldsymbol{\Theta}} = \boldsymbol{u}, \quad \dot{\boldsymbol{\Theta}} = \dot{\Theta}_t + S_l(1)\mathrm{diag}(\Delta t)\ddot{\boldsymbol{\Theta}}, \quad \boldsymbol{\Theta} = \Theta_t + S_l(1)\mathrm{diag}(\Delta t)\dot{\boldsymbol{\Theta}}, \tag{6.38}$$

where the bold symbols indicate that they consist of values along the entire horizon, $\boldsymbol{\Theta}$ includes the angles for all joints, $S_l(1)$ is a lower triangular matrix filled with 1, and Δt is a vector of time steps across the horizon. We use smaller time steps earlier along the horizon and larger ones for later time steps. By implementing the dynamics in batch, we avoid iteratively unrolling the dynamics, speeding up controller computation significantly. For computing cost, we also require the Cartesian poses X, velocities $\dot{X}$, and accelerations $\ddot{X}$ of the end-effector. These are obtained via:

$$X = \mathrm{FK}(\Theta), \quad \dot{X} = J(\Theta)\dot{\Theta}, \quad \ddot{X} = \dot{J}(\Theta)\dot{\Theta} + J(\Theta)\ddot{\Theta} \tag{6.39}$$

where $\mathrm{FK}(\Theta)$ are the forward kinematics and $J(\Theta)$ is the kinematic Jacobian. The cost function is a weighted sum of a number of terms, as defined by Bhardwaj et al. [15], which

includes: distance of end-effector to the goal pose c_{pose}, a time varying velocity limit cost c_{stop} that enables stopping within the specified horizon, a joint limit cost c_{joint}, a manipulability cost c_{manip} which encourages the arm to avoid singular configurations, a self-collision cost $c_{self-coll}$, and a obstacle collision cost c_{coll}. The overall final cost function is then:

$$c(x,u) = w_p c_{pose}(x) + w_s c_{stop}(x) + w_j c_{joint}(x) + w_m c_{manip}(x) + w_c(c_{self-coll}(x) + c_{coll}(x)). \quad (6.40)$$

The self-collision cost is implemented with a neural network that predicts the closest distance between the links of the robot given a configuration. The collision cost is a binary cost which uses a learned collision checking function that operates directly on raw point cloud data and classifies if a robot link is in collision. See Bhardwaj et al. [15] for further details about each of the individual cost terms. We consider the FrankaObstacles task, where we control the 7 degree-of-freedom (DOF) Franka Panda robot arm and steer it towards a target goal from a fixed starting pose while avoiding a single pole obstacle. The obstacle and goal positions are randomized at the beginning of each episode, which lasts for 600 time steps. An episode is considered successful if the end effector reaches the target position under the time constraints while avoiding the obstacle. We also consider a simplified version which has no obstacles, which we call Franka.

6.5.4 Architectural and Training Details

All NFs for both `NFMPC` and `FlowMPPI` were implemented in PyTorch and contain affine coupling layers which use multilayer perceptrons (MLPs) for both the scale and translation terms. The scale and translation networks use Tanh and ReLU activations, respectively. We also employ layer normalization [105] in these networks to help prevent overfitting. Interestingly, we found that adding batch normalization between each layer, as proposed by Dinh et al. [101], actually hurt performance and was therefore excluded. In all environments, we use 5 RealNVP blocks for the NF, and each MLP has a hidden dimensionality of 128 neurons. For the PNGrid, Franka, and FrankaObstacles tasks, the shift model is also an MLP with a single hidden layer of 128 neurons and a ReLU activation function. In

PNRand and PNRandDyn the shift model is implemented as an LSTM with a hidden dimensionality of 128 neurons. For a fair comparison, the same architecture was used for both `FlowMPPI` and `NFMPC`. We trained all networks with the Adam optimizer [63] using a learning rate of 10^{-4}.

To train the `NFMPC` variants, we following the training procedure in Algorithm 6, which is carried out training over D episodes. In each episode d, we sample an environment from $\mathcal{C}$ and initial state from ρ. We then perform our rollouts by sampling latent controls $\hat{\boldsymbol{z}}_t^{(1:N)}$, passing them through the normalizing flow to get controls $\hat{\boldsymbol{u}}_t^{(1:N)}$, and then applying them to our approximate dynamics model and cost function to get weights $w_t^{(1:N)}$. These variables are used to update the latent distribution of the policy to θ_t. Next, we can either sample a control from the policy or use a control corresponding to the latent mean. We apply this control to the true system and shift the latent distribution parameters forward with the shift model. Finally, we compute the loss for the current time step, accumulate the loss to our running sum, and repeat for T time steps. Once the episode is complete, we update λ_d using the gradient of the loss and carry on to the next episode. Every 100 environments, we test the controller on 10 held out environments and save the current model if it outperforms the previous best on this validation set. While this introduces a variable number of episodes used to train the models, we generally find convergence between 2000 and 7000 episodes.

In contrast, to train `FlowMPPI`, we do not actually run an episode. Instead, we generate the environment and take a gradient step on the initial distribution of the flow conditioned on the environmental information. We train `FlowMPPI` over 10000 randomly generated environments for each task. We achieved the best performance by initializing the flow with the `FlowMPPI` solution, and then refining the flow and learning the shift model jointly as above. When we condition the NF on additional information, this is simply appended to the current input of each shift and translation network directly. This conditional information includes start and goal locations for PNGrid and Franka and start, goal, and obstacle locations for PNRand and FrankaObstacles. For PNRandDyn we use the current state instead of the initial location, which was found to improve performance for all controllers. For the

Algorithm 6: Training Loop

Input: Environment dist. $\mathcal{C}$, initial state dist. ρ, initial param. $\tilde{\theta}_0$, λ_0

Parameters: # episodes D, episode length T, # samples N

for $d = 1, 2, \ldots, D$ **do**
- Sample environment $c \sim \mathcal{C}(\cdot)$
- Sample initial state $x_0 \sim \rho(\cdot|c)$
- Initialize episode loss $l_d \leftarrow 0$
- **for** $t = 0, 1, \ldots, T_k - 1$ **do**
  - $(\hat{\boldsymbol{u}}_t, \hat{\boldsymbol{z}}_t, w_t)^{(1:N)} \leftarrow \text{Rollout}(x_t,\, c,\, \tilde{\theta}_t,\, \lambda_d)$
  - Update $\tilde{\theta}_t$ to θ_t using latent MPPI update
  - Sample $u_t \sim \pi_{\theta_t, \lambda_d}$ or $u_t \leftarrow h^{-1}_{\lambda_d}(\mu_t; c)$
  - Apply control to system $x_{t+1} \sim f(x_t, u_t)$
  - Shift parameters $\tilde{\theta}_{t+1} \leftarrow \Phi_{\lambda_d}(\theta_t, c)$
  - Compute loss $\hat{J}_t(\theta_t, \lambda_d; x_t, c)$
  - Accumulate loss $\ell_d \leftarrow \ell_d + \hat{J}_t$
- **end**
- Update λ_d to λ_{d+1} with $\nabla_\lambda \ell_d$ using SGD

end

obstacles, the conditional information is the Cartesian coordinates of each object of interest stacked together in a vector.

6.6 Results

6.6.1 Planar Robot Navigation

PNGrid. We first consider a variant of the planar navigation task that involves eight static obstacles arranged in a grid (PNGrid). The NF for `FlowMPPI` is conditioned on the obstacle locations, current state, and goal position, while for `NFMPC`, it is unconditional. Note that we do not condition the shift model, as this consistently hurt performance. We quantitatively

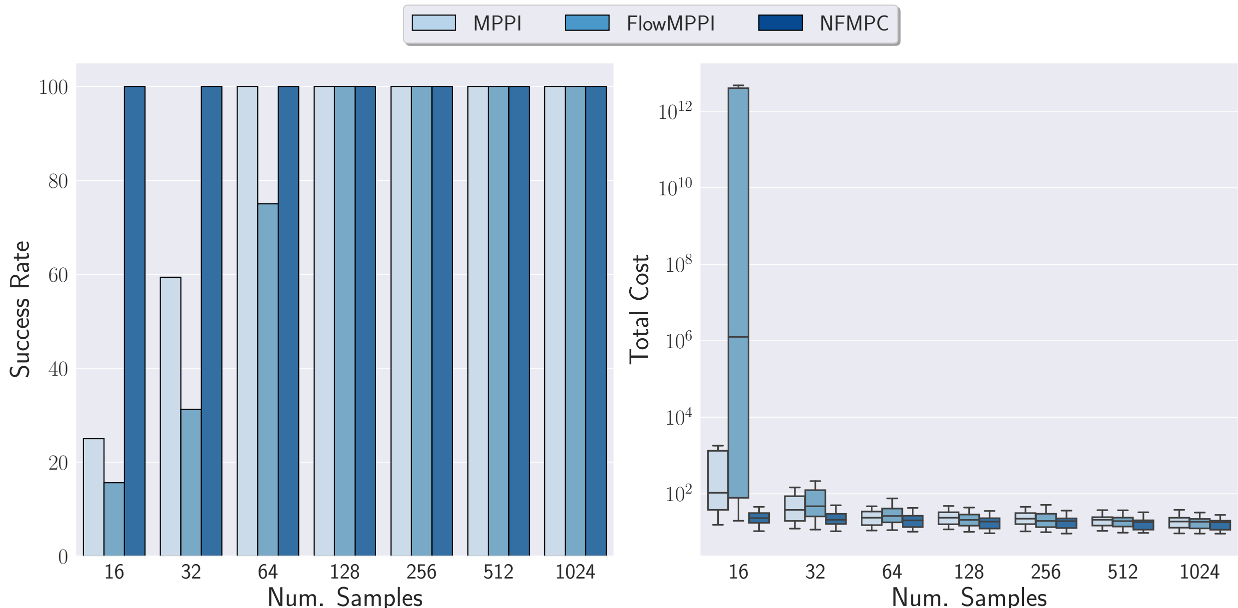


Figure 6.2: Success rate and cost distribution on the PNGrid task across a different number of samples.

compare all controllers in Figure 6.2 and find that `NFMPC` consistently matches or outperforms both `MPPI` and `FlowMPPI` at each sample quantity in terms of both success rate and average trajectory cost of successful trajectories. At 1024 samples, `NFMPC` achieves a 29% and 17% reduction in cost over `MPPI` and `FlowMPPI`, respectively. We also find that `NFMPC` scales more gracefully than `MPPI` as the number of samples is reduced. For instance, `NFMPC` is able to withstand a 64× decrease in the number of samples (1024 to 16) while still achieving a 100% success rate, although the average trajectory cost increases by 91%. Meanwhile, `MPPI` at 16 samples has a 97% success rate and a 26% increase in average trajectory cost over `NFMPC` with the same number of samples. In fact, we found that while `FlowMPPI` improves over `MPPI` at higher sample counts, it actually performs significantly worse with fewer samples. At 16 samples, `FlowMPPI` achieves only a 6% success rate and has over a 2× worse average trajectory cost than `NFMPC`.

This is potentially because in the standard implementation of `FlowMPPI`, half of the

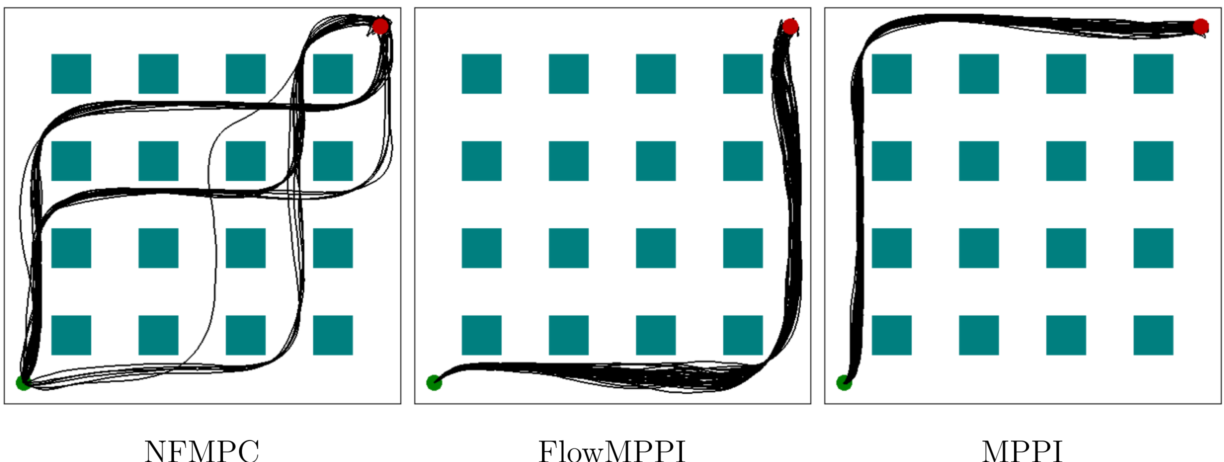


Figure 6.3: Visualization of trajectories in the PNGrid task across multiple random seeds for a fixed environmental layout.

samples come from the NF and the other half are Gaussian perturbations of the current control-space mean. Initially, the samples coming from the NF provide a good initialization for the control distribution mean. However, as the latent Gaussian distribution used by the NF is never updated, half of our samples are always coming from this same distribution. When the environment or task is more complex, the conditioning information provided to the NF is enough to transform the samples in a useful way. However, in this simple environment, our hypothesis is that as the robot moves in the environment, these samples may cease to be as useful or informative. We then have to rely on the other half of samples coming from Gaussian perturbations of the control-space mean to do most of the work. As such, we effectively have half the budget of samples to work with than `MPPI` would, as the samples from the NF potentially do not provide much useful information. Meanwhile, because `NFMPC` is trained recurrently and updates are performed in the latent space, it can better exploit structure in the environment to transform samples.

To better understand what `NFMPC` is doing differently, we superimpose 32 different trajectories with fixed start and goal positions using each controller in Figure 6.3. We find that both `MPPI` and `FlowMPPI` always select the same path through the environment. Meanwhile,

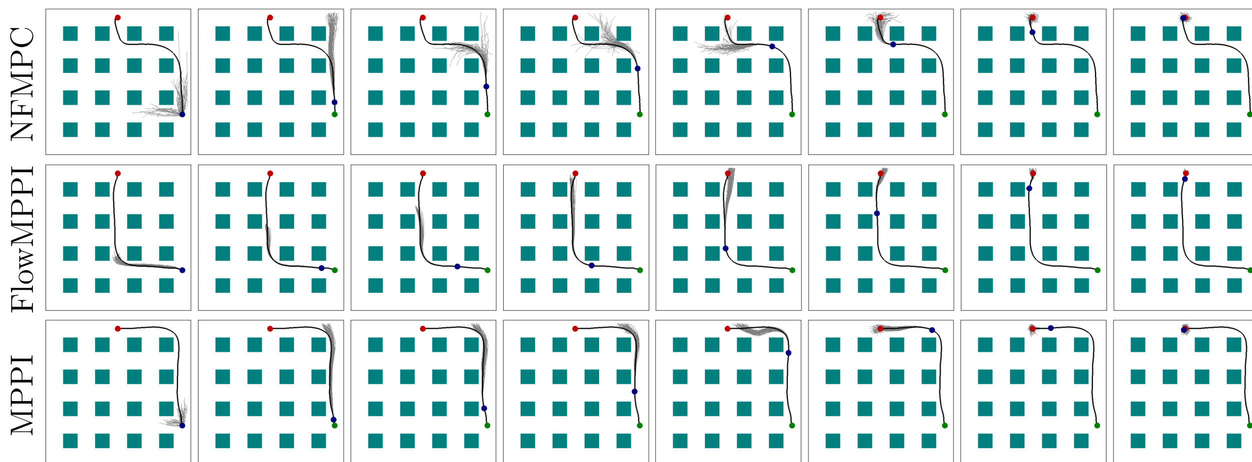


Figure 6.4: Visualization of a trajectory and top samples from (top) NFMPC, (middle) FlowMPPI, and (bottom) MPPI on the PNGRID task.

NFMPC is able to discover different paths through the environment, allowing it to better react to the stochastic perturbations that knock it off the current plan and improve performance. Additionally, we visualize the resulting trajectories and top samples drawn from the distributions for all controllers on one of the validation environments in Figure 6.4. As we would expect, the initial trajectory from FlowMPPI is better than those of the other two controllers, which basically lay in straight lines in front of the robot. However, NFMPC is able to discover a slightly faster route to the goal as it proceeds in the environment. The baseline MPPI controller takes a similar, but slightly longer, route to the goal. Additionally, it takes a longer time to ramp up its velocity compared to the other two controllers.

PNRand. Next, we consider a variant in which the eight obstacles are randomly placed (PNRAND). We consider the case where we condition the NF on obstacle locations, initial state, and goal position. However, it is important to note that we do not condition the shift model, as we found this consistently hurt performance. In Figure 6.5, we display the quantitative results and find that NFMPC performed about on par with FlowMPPI, which outperformed MPPI. Unlike on the PNGRID task, both NFMPC and FlowMPPI scale similarly

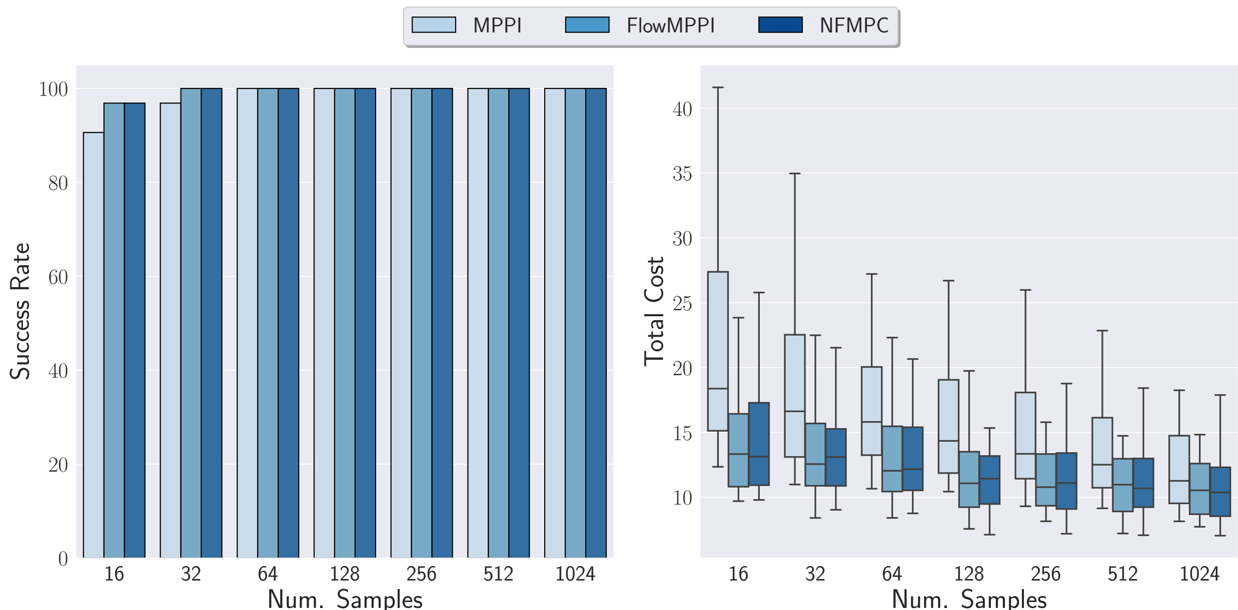


Figure 6.5: Success rate and cost distribution on the PNRand environment across a different number of samples.

with a reduction of samples and better than MPPI. This can be partly attributed to the fact that the obstacles are more spaced out and there are more "holes" in the environment than in the grid. Therefore, it is easier to avoid collisions, possibly contributing to the higher success rates when the controller has access to fewer samples. Moreover, conditioning on the obstacle locations provides the NF more information, which can be exploited without requiring us to update the latent distribution.

We again visualize the trajectories and top samples for all controllers on a validation environment in Figure 6.6. First, we note that the samples in FlowMPPI appear to be better spread around in the environment to search for good paths towards the goal. Meanwhile, NFMPC seems to have all top samples directed in one direction. All models seem to find the same path, with MPPI oscillating more near the goal and reaching the goal more slowly than NFMPC and FlowMPPI. The similar performance of NFMPC and FlowMPPI can be partially attributed to the fact that all we may really need to succeed in this environment is a good

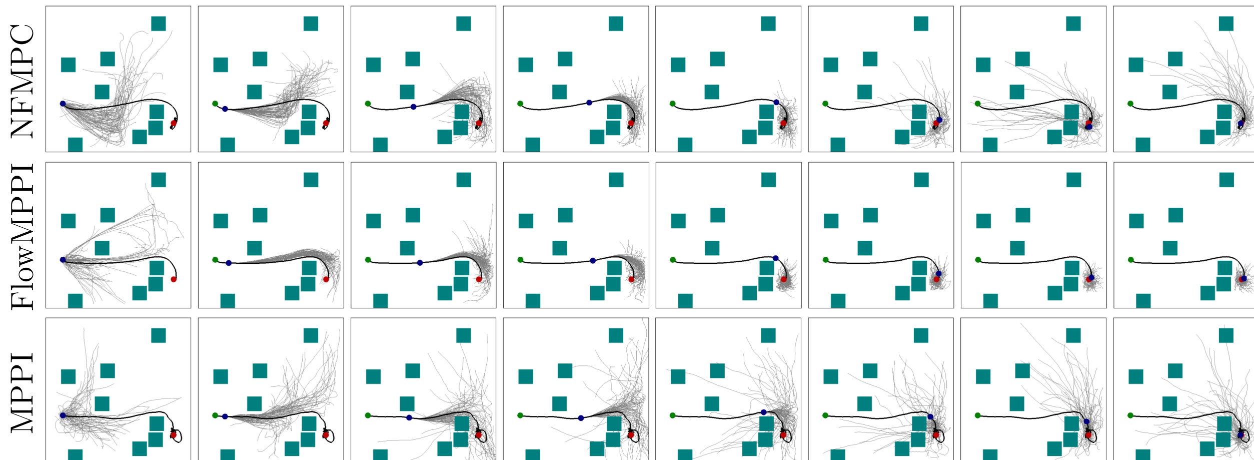


Figure 6.6: Visualization of a trajectory and top samples from (top) `NFMPC`, (middle) `FlowMPPI`, and (bottom) `MPPI` on the PNRand task.

initial trajectory which steers us towards the goal. Whether we refine this trajectory with a learned latent shift model or with Gaussian perturbations in the control space does not appear to make much difference in performance. However, in PNRandDyn, we demonstrated that there is a significant advantage to our approach, indicating that dynamic environments may be a good application for `NFMPC`.

Finally, in order to evaluate the benefit of conditioning the NF, we compare the performance of `NFMPC` with and without conditioning the flow on PNRand in Figure 6.7. We find that the conditional model consistently outperforms the unconditional model in terms of median cost, with the gap growing at reduced sample counts. This may be because the dynamics in PNRand are rather simple, and there may not be much general structure for the unconditional model to exploit since the obstacle locations are entirely random. Therefore, while the unconditional model does fairly well, conditioning the flow, and not the shift model, seems to enable further gains, even in this simple task.

PNRandDyn. Now we consider the case where each obstacle's current position is perturbed with Gaussian noise and clipped to be within map bounds, and the starting and goal locations

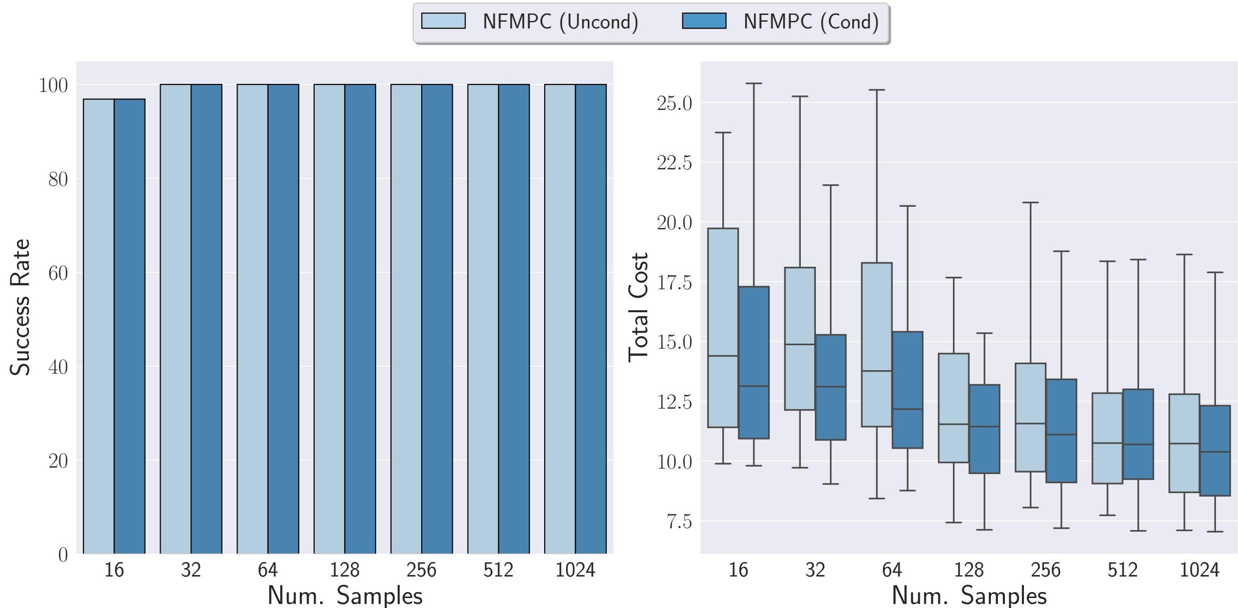


Figure 6.7: Comparison of unconditional and conditional models on the PNRand environment across a different number of samples.

of the robot are randomized in each episode. The NF for both `NFMPC` and `FlowMPPI` is conditioned on the obstacle locations, current state, and goal position. Note that we do not condition the shift model, as this consistently hurt performance.

We quantitatively compare all controllers in Figure 6.8. The trajectory cost box plots represent the median and quartiles of the distribution. We find that `NFMPC` consistently outperforms the both `MPPI` and `FlowMPPI`. While all controllers reach a 100% success rate at 1024 samples, `NFMPC` achieves a 20% and 8% lower median cost over `MPPI` and `FlowMPPI`, respectively. We also find that `NFMPC` scales more gracefully overall as the number of samples is reduced. For instance, it is able to withstand a 32$\times$ decrease in the number of samples (1024 to 32) while reducing success rate by only 3% and increasing median cost by 8%, nearly matching `FlowMPPI` at 1024 samples. In comparison, `MPPI` reduces success rate by 6% and increases median cost by 43%. Similarly, `NFMPC` outperforms `FlowMPPI` at all sample amounts in terms of median cost, although at 16 samples `FlowMPPI` achieves a slightly higher success

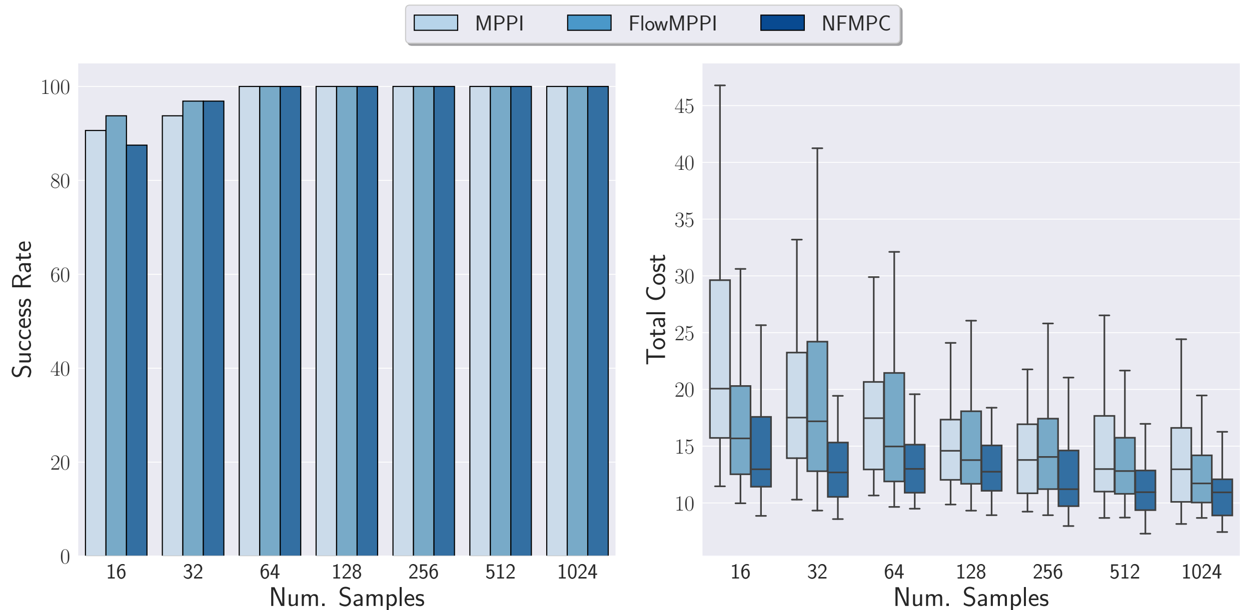


Figure 6.8: Success rate and cost distribution on the PNRandDyn environment across a different number of samples.

rate than `NFMPC`. We visualize trajectories and top samples in Figure 6.9. The green and red dots are the starting and goal locations, respectively, and the blue dot is the current position of the robot at the given time step. The thick black line is the resulting path taken by the controller, while the gray lines are the top samples generated at the current state. In this example, `FlowMPPI` nearly collides with an obstacle, while `NFMPC` and `MPPI` are able to safely reach the goal. `FlowMPPI` over commits to a narrow corridor and is unable to reroute in time to account for the new obstacle location. `NFMPC` takes a similar trajectory to `FlowMPPI`, however, it pauses until the obstacle moves out of the way to proceed towards the goal.

Additionally, we also performed additional ablation studies where we considered training an unconditional model, as we did before in the PNRand task. Moreover, we explored transferring controllers trained in the PNRand task to this dynamic version of the environment. We plot the quantitative results from these ablation studies in Figure 6.10. Again, we find that conditional models consistently outperform unconditional ones, with the gap in

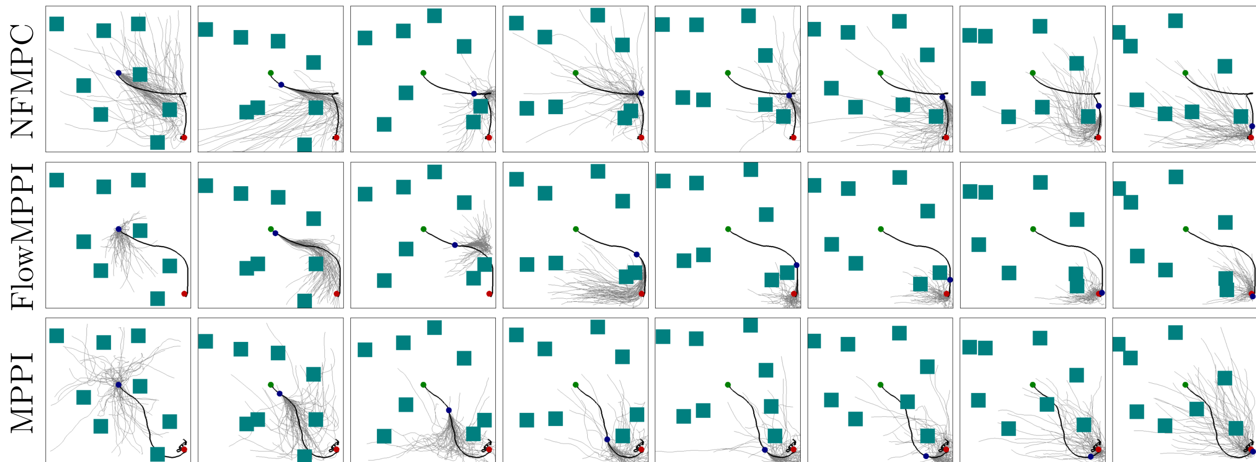


Figure 6.9: Visualization of a trajectory and top samples from (top) NFMPC, (middle) FlowMPPI, and (bottom) MPPI on the PNRandDyn task.

performance more pronounced. Not only is there a reduction in median cost for unconditional controllers, but it sometimes fails in environments in which its conditional counterpart succeeds. Transferring the models trained in PNRand works surprisingly well at 1024 samples. However, at lower sample counts, there is a more pronounced difference in the transferred controllers to those specifically trained in the dynamic environment. Therefore, it appears that controllers are more efficient at exploring the environments they are trained on, and unsurprisingly, using more samples can partially overcome this gap.

6.6.2 Franka Panda Arm

Franka. Next, we apply NFMPC to the Franka task, first without obstacles and just a target goal. We plot our quantitative results in Figure 6.11 and find that NFMPC again consistently matches or outperforms MPPI and FlowMPPI at each sample amount. In fact, NFMPC always achieves a 100% success rate at all sample counts, while both MPPI and FlowMPPI significantly drop in performance at lower amounts of samples. Moreover, FlowMPPI sometimes actually performed worse than MPPI, indicating that conditioning on just goal location does not help as much in this scenario.

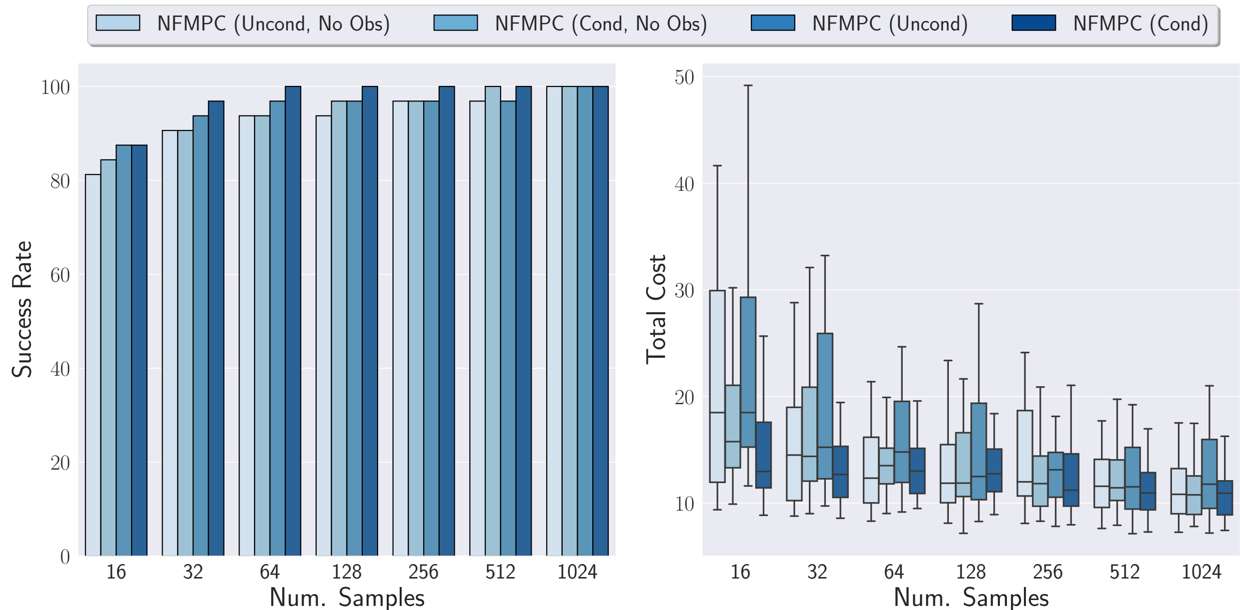


Figure 6.10: Comparison of unconditional and conditional models on the PNRandDyn environment across a different number of samples when trained in an environment with no obstacles (PNRand) and retrained with obstacles present.

FrankaObstacles. Now we consider the case where there is a single pole obstacles randomly placed in the environment. The NF for both `NFMPC` and `FlowMPPI` is conditioned on the obstacle locations, initial state, and goal position. It is also important to note that no controller achieves a 100% success rate, as not every randomly generated environment is feasible. As shown in Figure 6.12, at 1024 samples, `NFMPC` achieves a success rate of 84%, while `FlowMPPI` and `MPPI` only succeed 81% and 78% of the time, respectively. Again, `NFMPC` scales better with a reduced number of samples. With a 64× decrease in the number of samples (1024 to 16), `NFMPC` only drops in success rate by 29%. Meanwhile, `FlowMPPI` decreases by nearly 70% to a success rate of 25%. However, both fair better than `MPPI`, which drops by 80% to a success rate of 16%. These results support the hypothesis that learning to perform MPC updates in the latent space of the NF and training the controller as a recurrent network improves performance.

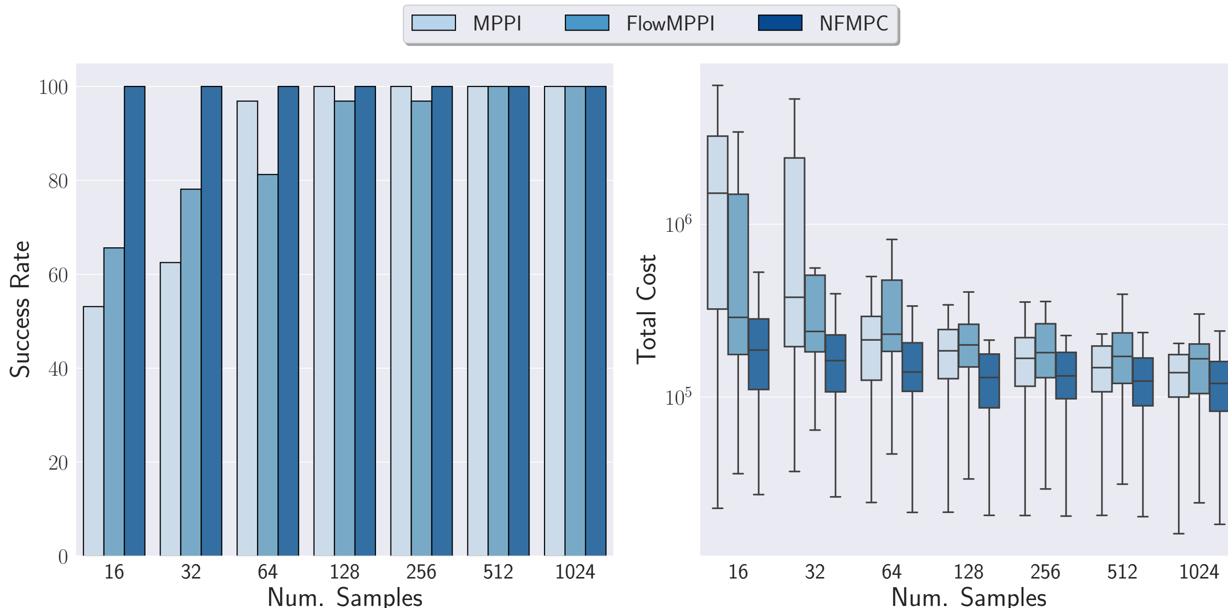


Figure 6.11: Success rate and cost distribution on the Franka environment across a different number of samples.

To more clearly understand what `NFMPC` is doing differently, we visualize the performance of all three controllers on a held-out validation environment in Figure 6.13. The green and red markers indicate the end-effector and goal positions, respectively, while the single pole is the obstacle which must be avoided. Additionally, the green trajectories represent the top samples from the controller at each time step. While `MPPI` collides with the obstacle, both `NFMPC` and `FlowMPPI` learn to take a different path which is collision-free. `FlowMPPI` generates better initial trajectories than `NFMPC`, which are more pointed towards the goal location and achieve a greater velocity. However, `NFMPC` is able to better adapt the distribution throughout the episode and reach the goal more quickly.

Additionally, we perform an additional ablation study in which we again compare an unconditional and conditional model on the FrankaObstacles environment in Figure 6.14. We also show the performance of transferring the unconditional model trained on the Franka environment without obstacles to an environment which contains the single pole obstacle. At

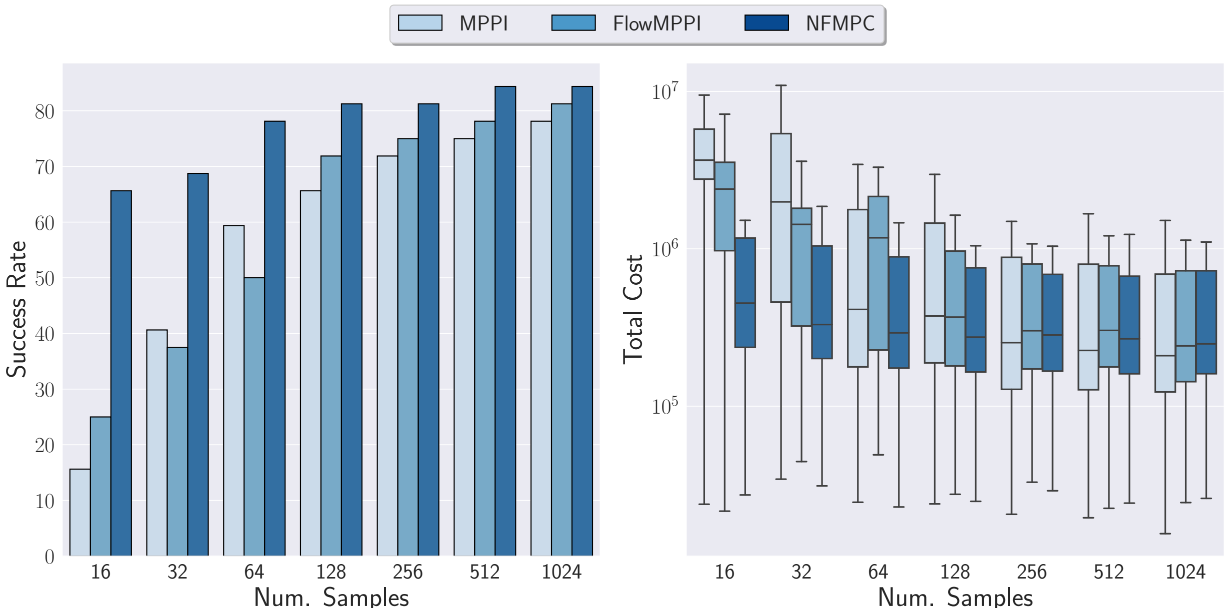


Figure 6.12: Success rate and cost distribution on the FrankaObstacles environment across a different number of samples.

1024 samples, the conditional model performs the best in terms of success rate. Surprisingly, the unconditional model trained without obstacles performs best in terms of median cost and scales better to fewer samples. Therefore, while the conditional model more often finds a feasible path to the goal, the unconditional model is better able to exploit structure across environments to find lower cost trajectories. One possible explanation is that because the unconditional model is trained without knowing the specific obstacle locations, it has to be more robust to variation in the environment. This also shows that transferring the learned distribution to novel environments is possible. However, since we only have a single static obstacle, it is not clear if these findings would generalize to more complex environments.

6.6.3 Breakdown of Trajectory Cost.

We would like to better understand how the learned controllers improve upon the baseline on the FrankaObstacles task. As discussed in Section 6.5, the cost function for the Franka

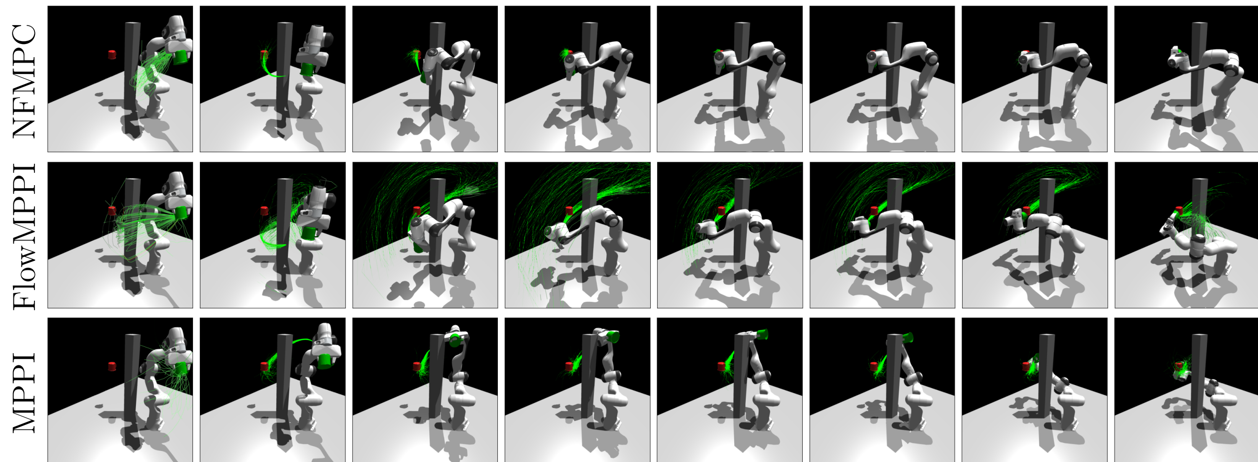


Figure 6.13: Visualization of a trajectory and top samples from (top) `NFMPC`, (middle) `FlowMPPI`, and (bottom) `MPPI` on the FrankaObstacles task.

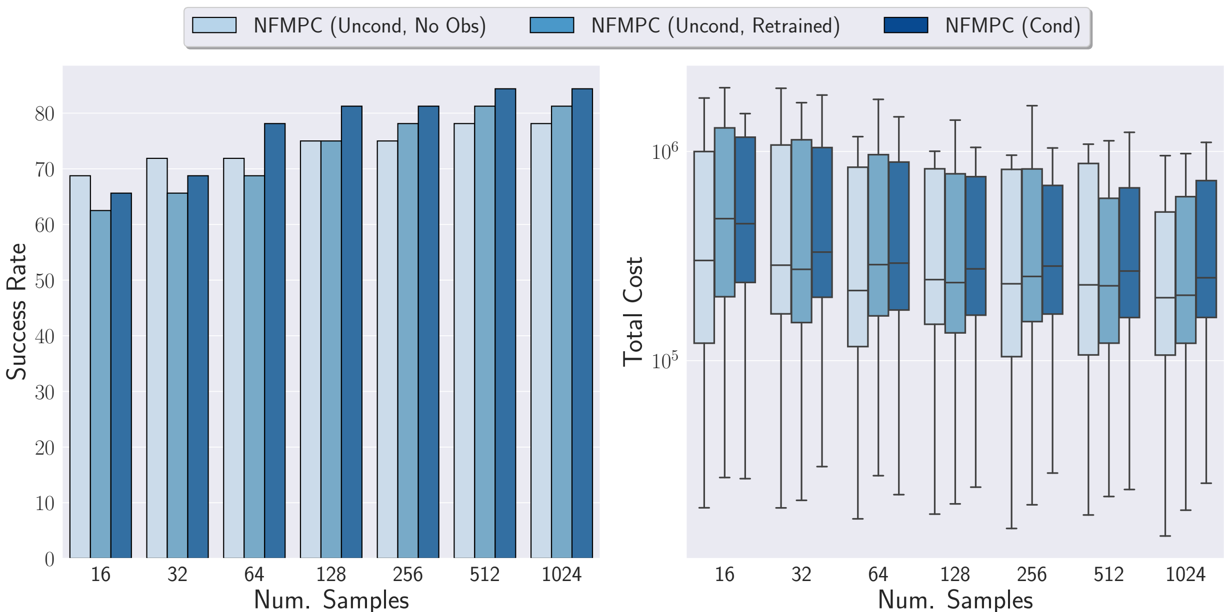


Figure 6.14: Comparison of an unconditional model trained with and without obstacles to a conditional model in the FrankaObstacles environment across a different # of samples.

tasks is composed of multiple terms. By inspecting the averages for each term, we hope to gain insight into the learned sampling distributions. Briefly, we consider a manipulability cost

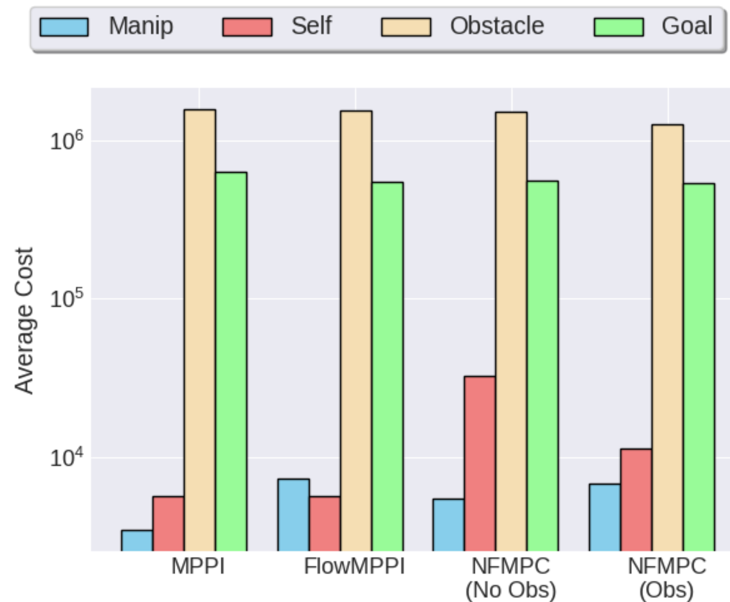


Figure 6.15: Breakdown of different cost terms for each controller on FrankaObstacles.

(Manip), which encourages the arm to avoid singular configurations, a self-collision cost (Self), an obstacle collision cost (Obstacle), and a distance-to-goal cost (Goal). The cost breakdown for these terms in shown in Figure 6.15. The baseline `MPPI` controller achieves the lowest manipulability cost. Since this term is not weighted as highly, it makes sense that training the normalizing flow would focus on minimizing terms which were more heavily weighted. Meanwhile, `FlowMPPI` achieves the lowest self-collision cost, which may be due to the fact that it starts off with a better initial plan. However, `NFMPC` trained with obstacles (`NFMPC (Obs)`) achieves the lowest obstacle collision and distance-to-goal cost. One possible explanation for this improvement is that, because we train `NFMPC` with BPTT, we are potentially able to account for errors which arise due to the inaccurate model. Additionally, learning the shift model may be an important component to this improvement, which we explore below. Finally, we note that `NFMPC (Obs)` is better than `NFMPC` trained without obstacles (`NFMPC (No Obs)`) in all cost terms except the manipulability cost, as it is in distribution for the task. As we would expect, controllers trained in environments with obstacles perform better.

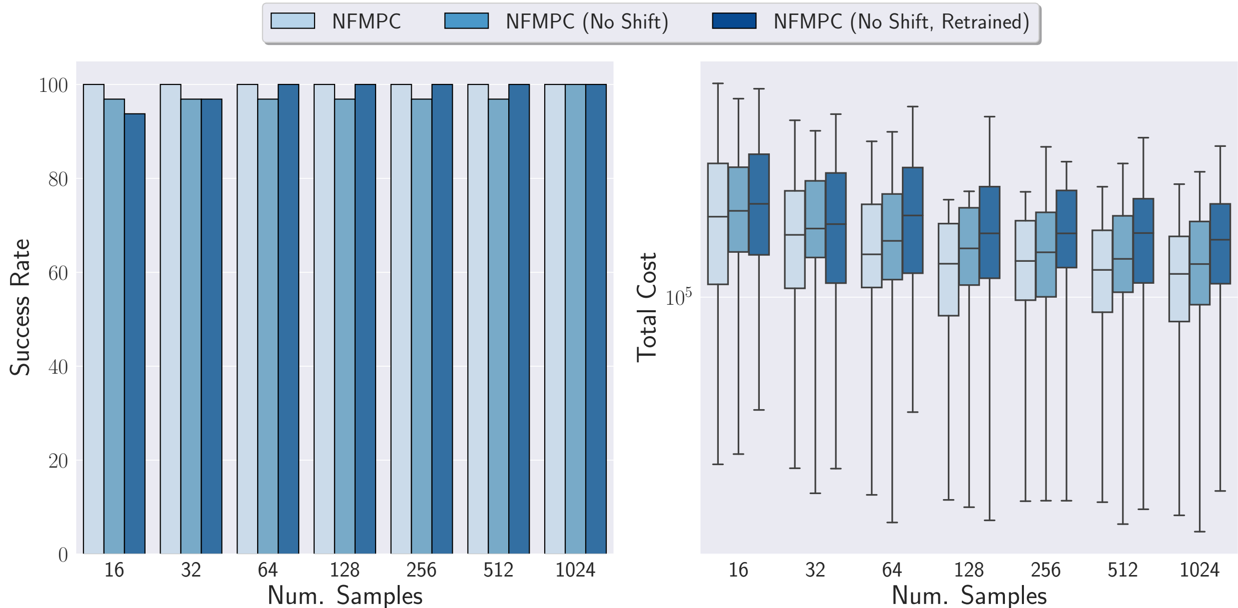


Figure 6.16: Success rate and cost distribution on the Franka task with (`NFMPC`) and without (`NFMPC (No Shift)` and **NFMPC (No Shift, Retrained)**) the learned shift model.

6.6.4 Shift Model Ablation.

One of our main contributions which sets us apart from prior work is that we learn a latent shift model and train the controller as a recurrent network. To determine the impact of this choice, we consider an alternate scenario where the control sequence is instead shifted in control space. That is, we shift the mean in control space, as in the baseline `MPPI` controller, and then run the flow backwards to infer the corresponding latent mean, which is used to bootstrap the next iteration. We consider two cases: 1) taking a pre-trained controller and removing the shift model (`NFMPC (No Shift)`) and 2) retraining the controller entirely from scratch without the shift model (**NFMPC (No Shift, Retrained)**). In Figure 6.16, we show the results of this ablation study on the Franka task. Removing the shift model strictly hurts performance, even when we retrain the normalizing flow from scratch. Moreover, retraining the flow actually results in worse performance than simply removing the shift model from the pre-trained controller. This implies that training the entire controller with

BPTT allows it to discover lower-cost trajectories to the goal.

6.6.5 Performance Overhead of Normalizing Flow

We measured the average change in wall clock time across different amounts of samples for `NFMPC` and `FlowMPPI` compared to the baseline `MPPI` implementation on our NVIDIA Titan V GPU. For the Planar Robot Navigation and Franka Panda Arm experiments, the average change is 1.61× and 1.91×, respectively. Moreover, compared to `MPPI` with 1024 samples, the change in wall clock time for `NFMPC` and `FlowMPPI` with 16 samples is approximately 1.01× and 1.14×, respectively. Therefore, the introduction of the normalizing flow (NF) has a notable impact on wall clock time for both methods. However, this overhead is expected and does not prohibit either method's utility in the real world. And there are still clear performance advantages of `NFMPC` over the baselines in terms of success rate and average trajectory cost for a given sample amount.

Furthermore, the performance of `MPPI` reported in the paper and the above timing comparisons are when the optimization is run for a single iteration per time step. However, the performance of `MPPI` generally improves with an increased number of iterations at the cost of an increased runtime. For instance, we compare the performance of `MPPI` run for 3 iterations per time step with `NFMPC` run for a single iteration, both using 1024 samples. In the Franka task, we can reduce the average trajectory cost of MPPI to be only 12% worse than `NFMPC`, rather than the previous 19%. When we introduce obstacles for the FrankaObstaclestask, `MPPI` with 3 iterations can actually match the success rate of `NFMPC`, albeit with a worse average trajectory cost. However, in this case, `NFMPC` actually results in an average reduction of wall clock time by 24.8%. Similarly, for the PNGrid task, `MPPI` with 3 iterations reduces the average trajectory cost to be only 10% worse than `NFMPC`, rather than the previous 40%. However, this again comes at the cost of increased runtime, as `NFMPC` reduces the average wall clock time by 26.8%. Therefore, `NFMPC` allows us to surpass the performance of `MPPI` run with more iterations while reducing the required runtime.

Additionally, `NFMPC` has substantial runtime benefits over `FlowMPPI` due to its improved

scaling. For the PNGridtask, NFMPC with 128 samples outperforms FlowMPPI with 1024 samples while reducing average runtime by 22%. Similarly, for the Frankatask, NFMPC with 64 samples outperforms FlowMPPI with 1024 samples while reducing average runtime by 43%. And for both Franka tasks (Franka and FrankaObstacles), NFMPC with 16 samples outperforms FlowMPPI with 128 samples while reducing average runtime by 8%. As such, there are clear runtime benefits for NFMPC over both FlowMPPI and MPPI run for additional iterations. Finally, it is also important to note that we did not perform a hyperparameter sweep on the NF, and it may be possible to significantly reduce the size of the network while retaining performance benefits.

6.7 Discussion

We presented a method for learning MPC sampling distributions with normalizing flows (NFs) which moves all online parameter updates and warm-starting operations into the latent space. We show how to frame the problem as bi-level optimization and derive an approximate gradient through the MPC update to train the distributions. Additionally, we illustrate how to incorporate control box constraints directly into the NF architecture. Through our empirical evaluations in both simulated navigation and manipulation problems, we demonstrate that our approach is able to surpass the performance of all baselines. Moreover, we find that controllers which move all operations into the latent space are often able to scale more gracefully with a reduction in the number of samples. These results indicate the importance of leveraging the latent space in learned sampling distributions for MPC and a potential avenue for reducing the computational requirements of sampling-based MPC for resource-constrained systems. Finally, because we learn all components of the controller through episodic interactions with the environment, they can potentially be trained to account for the modeling errors.

However, a major limitation our method is that the learned distribution and shift model are only valid for a fixed horizon and control dimensionality. Therefore, these components cannot be directly transferred to new robots or for alternate horizons without being retrained. However, this could potentially be remedied by novel architectural innovations and training

distributions across both environments and robots. Moreover, the learned distribution is specific to the environmental distributions on which it was trained. Therefore, it does not always perform as well when transferred to out-of-distribution environments. However, this is always going to be a challenge for any learning-based method and addressing it is an open problem for future research. For instance, our approach could be combined with the method developed by Power et al. [58, 59], which learned a generative model of environments and used this learned distribution to perform a projection step on the conditional NF. However, this approach would only be valid for conditional models, and addressing transfer of unconditional models is an unresolved question. Finally, both our approach and `FlowMPPI` introduce an additional overhead to due to running the NF which cannot be ignored. That being said, the size of the networks we consider in this work are fairly small and running the NF can be easily parallelized on a GPU. This increase in wall clock time may be worth the additional performance gains. Additionally, we scale better with a reduction in samples compared to all baselines. Therefore, we can potentially alleviate some of the introduced overhead by reducing the number of samples while still meeting the application demands.

Chapter 7

DEEP MODEL PREDICTIVE OPTIMIZATION

In Chapter 5, we proposed to replace the hand-designed optimization algorithm in MPC with a learned update rule. However, we relied on imitation learning using a MPC expert with access to substantially more samples than the learner. This proved to be a viable strategy for improving performance in sample-constrained regimes. But optimizers trained this way are ultimately limited by the quality of the expert. Rather than mimic controllers with access to more computational resources, ideally we could learn an update rule which enables the controller to surpass the performance of a hand-designed controller. In Chapter 3 and Chapter 6, we discussed learning components of the MPC pipeline by directly optimizing for performance rather than a surrogate loss. To improve the performance of MPC while retaining its robustness, drawing from our work in Sacks et al. [106], we propose Deep Model Predictive Optimization (`DMPO`), which *learns* an optimizer and warm-starting procedure directly via experience. That is, `DMPO` learns how to perform model-based planning more effectively while considering the computational demands for real-time deployment. We can also interpret `DMPO` as a recurrent policy class which incorporates the structure of MPC as an inductive bias. Our key contributions are:

1. We develop `DMPO`, a general approach for learning the inner-loop of the MPC optimizer and warm-starting procedure via reinforcement learning by viewing MPC as a structured recurrent policy class;

2. On a real quadrotor platform (Crazyflie 2.1 with upgraded motors) tracking infeasible zig-zag trajectories, we show that `DMPO` can outperform an end-to-end (`E2E`) policy trained with MFRL by 19%;

3. Tracking zig-zag trajectories with alternating 180° flips in the desired yaw, `DMPO` can improve error over a baseline MPC algorithm by up to 27% with $16\times$ fewer samples, saving $4.3\times$ memory requirements;

4. By exposing the quadrotor to turbulent wind fields with an attached cardboard drag plate, we show that `DMPO` can adapt zero-shot, matching the performance of the MPC baseline and outperforming `E2E` by 14%.

These experimental evaluations illustrate that `DMPO` is an effective strategy for improving the performance of model-based control under computational constraints. It also demonstrates the potential for `DMPO` to close the gap with MFRL while remaining robust to unmodeled disturbances.

7.1 Problem Formulation

In `DMPO`, we learn an update rule of the form in Equation (5.4) with RL, treating MPC as a structured policy class. However, the common shift-forward operation used to warm-start the optimization only works well when problems at adjacent time steps are similar, which may not be true if there are substantial perturbations. If there are substantial perturbations to the system, the optimal solution may lie far away from the current one. This choice of warm-start is also independent of the value of each decision variable. Therefore, `DMPO` additionally learns a shift model of the form

$$\tilde{\boldsymbol{\theta}}_{t+1} = \Phi_\phi(\boldsymbol{\theta}_t). \tag{7.1}$$

Such a shift model is joint across time steps and control dimensions. It also provides us with a parameter-dependent shift operation, which can take advantage of structure and the types of perturbations known to effect the system.

To learn these components with RL, we need to actually consider two different policies. There is the local policy output by MPC at each time step, $\hat{\boldsymbol{u}}_t \sim \boldsymbol{\pi}^{MPC}_{\boldsymbol{\theta}_t}(\cdot)$, which we call the optimizee. Additionally, the optimizer defines a policy over the parameters of the optimizee,

$\boldsymbol{\theta}_t \sim \boldsymbol{\pi}_\phi(\tilde{\boldsymbol{\theta}}_t, C_t^{(1:N)})$. Combined with the shift model, we have

$$\boldsymbol{\theta}_t \sim \boldsymbol{\pi}_\phi(\Phi_\phi(\boldsymbol{\theta}_{t-1}), C_t^{(1:N)}). \tag{7.2}$$

From the perspective of RL, $\boldsymbol{\pi}_\phi$ is the policy we wish to learn, while $\boldsymbol{\pi}_{\boldsymbol{\theta}_t}^{MPC}$ is part of the environment dynamics. This means we have to consider both the state of the system and optimizer, forming an auxiliary MDP $\hat{\mathcal{M}} = (\mathcal{H}, \Theta, \hat{P}, r, \hat{\rho}_0, \gamma)$. We have auxiliary states $h \in \mathcal{H}$, where $h_t = (x_t, \boldsymbol{\theta}_{t-1})$, actions $\boldsymbol{\theta}_t \in \Theta$, which are the optimizer outputs, and dynamics

$$h_{t+1} = \begin{bmatrix} x_{t+1} \\ \boldsymbol{\theta}_t \end{bmatrix} \sim \hat{P}(\cdot | h_t, \boldsymbol{\theta}_t) = \begin{bmatrix} P(\cdot | x_t, u_t) \\ \delta(\boldsymbol{\theta}_t) \end{bmatrix}, \tag{7.3}$$

where the control of the original system, u_t, is a function of the auxiliary action, $\boldsymbol{\theta}_t$, via

$$u_t = \hat{u}_t, \quad \hat{\boldsymbol{u}}_t \sim \boldsymbol{\pi}_{\boldsymbol{\theta}_t}(\cdot). \tag{7.4}$$

In practice, we usually set the control to be the mean at the current time step, $u_t = \hat{\mu}_t$, rather than sample from the optimizee policy. From the perspective of the original MDP, the current parameters of the MPC policy actually form a recurrent state. Therefore, even if `DMPO` is parameterized with feedforward networks, we are still effectively forming a recurrent policy. The initial state distribution $\hat{\rho}_0$ now samples both an initial system state and set of optimizer parameters. And the reward function is still defined on the original state-control space, $r(x_t, u_t)$, but now u_t is a function of the optimizer action $\boldsymbol{\theta}_t$ due to Equation (7.4).

Finally, in actor-critic algorithms, we also simultaneously learn the value function of the policy, which in this case is the optimizer, $\boldsymbol{\pi}_\phi$. It is common to use learned value functions as terminal costs for MPC, where this critic would be for the optimizee policy, $\boldsymbol{\pi}_{\boldsymbol{\theta}_t}^{MPC}$. A common method for improving the performance of MPC is to learn a value function, which is used as a terminal cost in the rollouts. This is equivalent to learning a critic for the optimizee policy, $\boldsymbol{\pi}_{\boldsymbol{\theta}_t}^{MPC}$. Instead, we are interested in learning a critic for the optimizer policy, $\boldsymbol{\pi}_\phi$. This means the critic should be defined on the auxiliary state, $V^\pi(h_t)$, making it a function of both the state of the system and optimizer. Note that the optimizer never directly receives

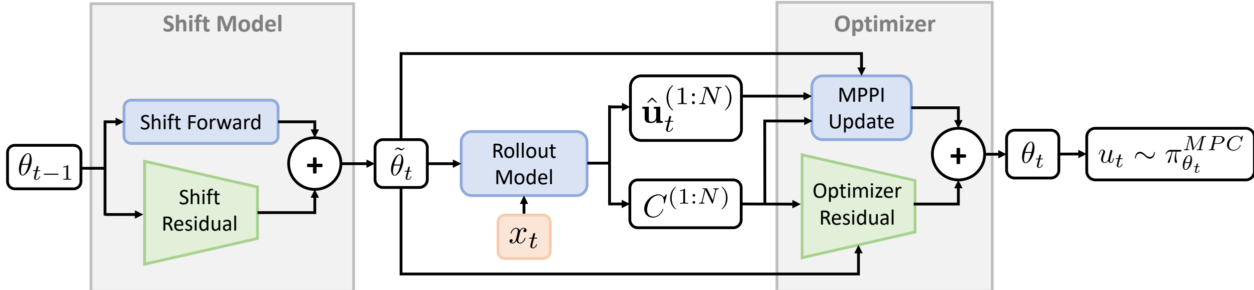


Figure 7.1: The DMPO architecture consists of two learnable modules, the shift model and optimizer. The fixed rollout model module performs rollouts of the sampled control sequences.

the system state x_t, instead being a function of trajectory costs, which can be thought of as partial observations of state. While we could condition on state as well, we found that this actually hurts generalization performance, especially for sim-to-real transfer. Intuitively, there may be multiple states for which we have the same cost distribution. This means that the optimizer may encounter out-of-distribution states during testing which have in-distribution trajectory costs. Therefore, operating in cost space seems to be a useful state representation, at least in our context.

7.2 Algorithm Overview

A valid instantiation of DMPO is to learn both the optimizer and shift model from scratch. Instead, we learn residuals on the MPPI update and shift-forward operation. While the closed-form MPPI update may be sub-optimal, it still can be fairly robust and generalize well to different tasks. By learning each component as a residual operation, it can alter the reward landscape in a way that can simplify exploration. If the MPPI controller is already good, it allows us to potentially inherit its robustness and generalization capabilities, with the residual providing small corrections. But even if the proposed MPPI update is far from optimal, such as in the case when the controller has access to few samples, it can still provide a hint about a good direction to search. We illustrate each module of DMPO in Figure 7.1, which we describe below:

Shift Model. Let us define $\tilde{\boldsymbol{\theta}}_t = (\tilde{\boldsymbol{\mu}}_t, \tilde{\boldsymbol{\Sigma}}_t)$ as the shifted parameters of our optimizee policy. Then we implement the shift model, Φ_ϕ, as a residual update:

$$\begin{gathered} \hat{\boldsymbol{\mu}}_t, \hat{\boldsymbol{\Sigma}}_t = \Phi_\phi(\boldsymbol{\theta}_t) \\ \tilde{\boldsymbol{\mu}}_t = \boldsymbol{\mu}_t^{SHIFT} + \hat{\boldsymbol{\mu}}_t, \quad \tilde{\boldsymbol{\Sigma}}_t = \boldsymbol{\Sigma}_t^{SHIFT} \odot \hat{\boldsymbol{\Sigma}}_t, \end{gathered} \tag{7.5}$$

where $\boldsymbol{\mu}_t^{SHIFT}$ and $\boldsymbol{\Sigma}_t^{SHIFT}$ are the mean and covariance following the normal shift forward operation and $\odot$ is the Hadamard product. Here, $\hat{\boldsymbol{\mu}}_t$ is the residual shifted mean which corrects the shift-forward operation output. And $\hat{\boldsymbol{\Sigma}}_t$ is a scale factor on the current covariance matrix, which we found worked better than an additive update. In our implementation, the network actually outputs the logarithm of the covariance, which is exponentiated and clipped to remain within a valid range. While both updates could be additive, we found that the multiplicative update for the covariance worked better in practice.

Rollout Model. We use fixed samples from a standard Gaussian, scaling and shifting them by $\tilde{\boldsymbol{\Sigma}}_t$ and $\tilde{\boldsymbol{\mu}}_t$, respectively, to get $\hat{\boldsymbol{u}}_t^{(1:N)}$. Additionally, we always include the current mean as one of the samples. After this reparameterization, we roll out the open-loop control sequences with our dynamics model to get a set of N scalar total trajectory costs concatenated into a vector $C^{(1:N)}$. Since we also run the `MPPI` update, we compute weights using Equation (5.1). Importantly, this is the only module which makes use of the current system state x_t. Theoretically, the shift model and optimizer could also be conditioned on state, but we found this actually hurt performance during sim-to-real transfer.

Optimizer. We compute the normal `MPPI` update using Equation (5.1) to get $\boldsymbol{\theta}_t^{MPC} = (\boldsymbol{\mu}_t^{MPC}, \boldsymbol{\Sigma}_t^{MPC})$. The `DMPO` update for the mean is then

$$\begin{gathered} \hat{\boldsymbol{\mu}}_t, \boldsymbol{g}_t, \boldsymbol{\sigma}_t^{\mu} = m_\phi^{\mu}(\tilde{\boldsymbol{\theta}}_t, C_t^{1:N}) \\ \boldsymbol{\mu}_t = (1 - \boldsymbol{g}_t) \odot \boldsymbol{\mu}_t^{MPC} + \boldsymbol{g}_t \odot \hat{\boldsymbol{\mu}}_t, \end{gathered} \tag{7.6}$$

where $\boldsymbol{g}_t$ is a gating term, which has the same dimension as the mean and is passed through a sigmoid to ensure it is between zero and one. It allows the network to modulate how much it relies on MPC versus the network output. Since we are using PPO [42] to train the components of `DMPO`, we actually need a distribution over proposed mean and covariance

updates. Therefore, the network also outputs a standard deviation $\boldsymbol{\sigma}_t^{\mu}$, which then defines our optimizer policy for the mean, $\boldsymbol{\pi}_\phi^{\mu} = \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t^{\mu})$. During training, we sample the mean update from this policy, but simply use the mean at test time. Similarly, for the covariance:

$$\hat{\boldsymbol{\Sigma}}_t, \boldsymbol{\sigma}_t^{\Sigma} = m_\phi^{\Sigma}(\tilde{\boldsymbol{\theta}}_t, C_t^{1:N}), \quad \boldsymbol{\Sigma}_t = \boldsymbol{\Sigma}_t^{MPC} \odot \hat{\boldsymbol{\Sigma}}_t, \tag{7.7}$$

where the optimizer scales the covariance matrix proposed by MPC. Our optimizer policy for the covariance is then $\boldsymbol{\pi}_\phi^{\Sigma} = \mathcal{N}(\boldsymbol{\Sigma}_t, \boldsymbol{\sigma}_t^{\Sigma})$. It is also important to note that these mean and covariance updates jointly depend on all the current parameters values, $\tilde{\boldsymbol{\theta}}_t$. This means that we consider the current covariance while updating the mean, and vice versa. Finally, the sampled mean and covariance define the optimizee policy $\boldsymbol{\pi}_{\boldsymbol{\theta}_t}^{MPC}$.

7.3 Experiments

7.3.1 Task and Implementation Details.

Quadrotor Trajectory Tracking. We perform all evaluations on a quadrotor trajectory tracking problem in which the desired trajectories are infeasible zig-zags with and without yaw flips. These zig-zags linearly connect a series of random waypoints, while the yaw flips are a 180° change to the desired yaw at each waypoint. For the real hardware experiments, we use the Bitcraze Crazyflie 2.1 equipped with the longer 20 mm motors from the thrust upgrade bundle. State estimation for position and velocity is provided by an OptiTrack motion capture system, while the Crazyflie provides orientation estimates via a 2.4 GHz radio. An offboard computer receives the state estimates and runs all controllers at a rate of 50 Hz. All controllers operate on desired body thrust f_{des} and angular velocity $\boldsymbol{\omega}_{des}$ commands. We convert angular velocity commands to torques using a custom low-level PI attitude rate controller. Using these torques and the desired thrust, we convert the commands to motor thrusts using an invertible actuation matrix.

For training and the MPC dynamics model, we use the following dynamics with a dt = 0.02:

$$\dot{\boldsymbol{p}} = \boldsymbol{v}, \quad m\dot{\boldsymbol{v}} = m\boldsymbol{g} + \boldsymbol{R}\boldsymbol{e}_3 f, \quad \dot{\boldsymbol{R}} = \boldsymbol{R}S(\omega), \tag{7.8}$$

where $\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{g} \in \mathbb{R}^3$ are the position, velocity, and gravity vectors in the world frame, $\boldsymbol{R} \in \mathrm{SO}(3)$ is the attitude rotation matrix, $\boldsymbol{\omega} \in \mathbb{R}^3$ is the angular velocity in the body frame, $S(\cdot) : \mathbb{R}^3 \to \mathrm{so}(3)$ maps a vector to its skew-symmetric matrix form, $\boldsymbol{e}_3$ is a unit vector in the Z direction, and m is the mass. The state of the system is then $\boldsymbol{x} = (\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{q}, \boldsymbol{\omega})$, where $\boldsymbol{q}$ is the quaternion representation of $\boldsymbol{R}$, and $\boldsymbol{u} = (f_{des}, \boldsymbol{\omega}_{des})$. Rather than explicitly model the angular velocity dynamics and low-level controller, we convert $\boldsymbol{u}$ to the thrust f and angular velocity $\boldsymbol{\omega}$ using a first-order time delay model:

$$\begin{aligned} \boldsymbol{\omega}_t &= \boldsymbol{\omega}_{t-1} + k(\boldsymbol{\omega}_{des} - \boldsymbol{\omega}_{t-1}) \\ f_t &= f_{t-1} + k(f_{des} - f_{t-1}). \end{aligned} \tag{7.9}$$

We trained the models across a series of zig-zags with random waypoints, with a yaw flip at each change in waypoint. The cost function includes terms defined on position and orientation tracking performance. Additionally, a control penalty was necessary for sim-to-real transfer. Without it, `DMPO` would exploit the simulator and learn aggressive commands which are difficult to perform on the real system.

Hyperparameters and Training. The optimizer, shift model, and value function were all parameterized with multi-layer perceptrons (MLPs) with a single hidden layer of 256 and ReLU activation functions implemented in PyTorch [61]. Since the value function operates on the full auxiliary state, we needed to make it aware of the reference trajectory. Therefore, we give it the desired trajectory for the next 32 time steps with a stride of 4, forming a 56-dimensional conditioning vector. We then make the value function dependent on this condition that provides a short window of the future desired states. However, we note that this is not necessary for the optimizer or shift model, as they do not operate on states. Instead, all information about the reference trajectory is implicitly represented in the total trajectory costs of the rollouts. For the network initialization, we set the last layer of each MLP to have a weight distribution of $\mathcal{N}(0, 0.001)$. This made each residual term effectively zero, allowing us to start with the `MPPI` update and shift-forward operation for warm-starting.

We trained the optimizers with PPO [42] on an NVIDIA RTX 3080 GPU with a $\gamma = 0.99$. To update DMPO, we used Adam [63] with a learning rate of 10^{-6} and 10^{-4} for the actor and critic, respectively. Learning the DMPO residual optimizers only took up to 1000 iterations of PPO to achieve good performance. We used Generalized Advantage Estimation (GAE) [107] to compute the advantages for policy updates, with a discount of $\gamma = 0.99$ and $\lambda = 0.95$. Each episode was a total of 100 steps (2 seconds), and we ran 8 environments in parallel. However, we intermittently would validation the current policy on a set of held-out trajectories, running the episodes for 200 steps (4 seconds). We used these validation episodes for model selection and only test the model from iteration which had the best validation performance. To improve performance, we used domain randomization (DR) [6–8] on the mass, randomly scaling it by a factor in $[0.7\times, 1.3\times]$, and the delay coefficient k, selecting it in $[0.2, 0.6]$. We also applied a constant force perturbation with a randomized direction and magnitude at the beginning of each episode in $[-3.5\ N, 3.5\ N]$.

Baselines. Our two baselines are MPPI and an end-to-end (E2E) 3-layer MLP policy operating on states, conditioned on desired trajectories. The desired trajectories consist of the 10 desired positions up to 0.6 seconds in the future, evenly spaced in time. E2E was also trained with PPO using DR, but with a learning rate of 3×10^{-4} and took about 10^7 iterations to converge. We used a custom implementation of MPPI in PyTorch [61], which used Halton sequences [90] for generating the fixed samples from a standard Gaussian. While normal pseudo-random sequences leave many regions of the parameter space untouched, Halton sequences correlate each point to alleviate this problem. Since we are using fixed samples, these type of low-discrepancy sequences improve our ability to explore the space of valid actions. Additionally, we scale each total trajectory cost by the minimum and maximum cost before computing the weights. This helps make the choice of temperature, β, invariant to the magnitude of the cost and makes it easier to tune. We tune all hyperparameters of the MPPI controller using a grid search in simulation on a fixed set of desired trajectories.

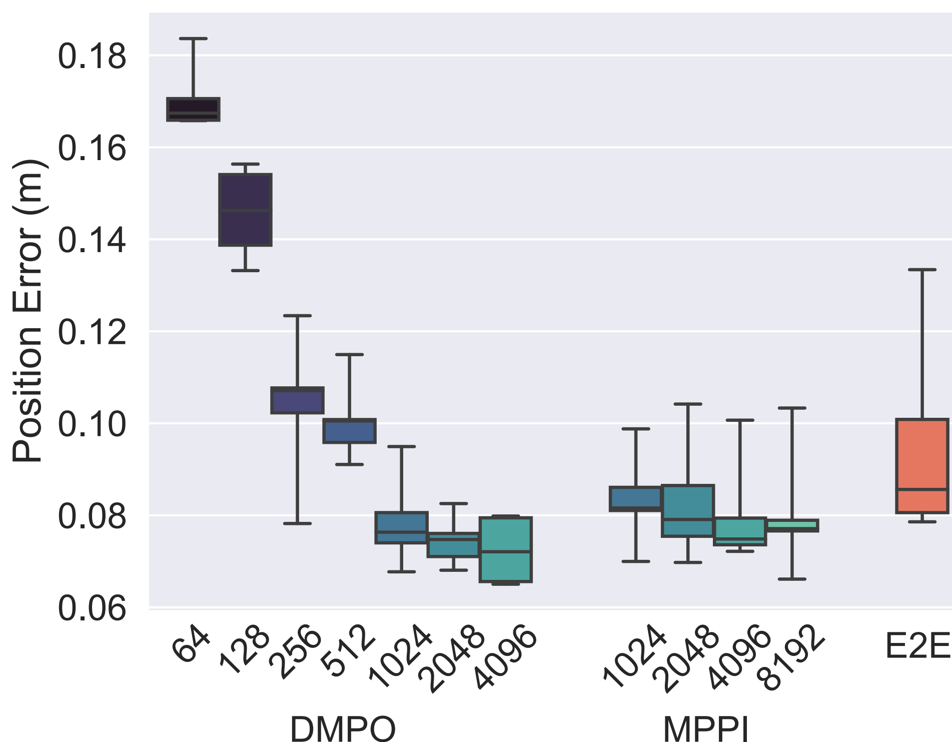


Figure 7.2: Position tracking error of DMPO versus MPPI and E2E on tracking random infeasible zig-zag trajectories without any environmental disturbances.

7.3.2 Tracking Performance on Infeasible Zig-Zags

We begin by evaluating the performance of DMPO compared to MPPI and E2E on random zig-zag trajectories without the presence of additional disturbances. For each controller, we evaluate the performance across 5 different fixed trajectory seeds. Figure 7.2 reports the position tracking error of each controller, varying the number of samples for DMPO and MPPI. The box plots represent the median and quartiles of the total episode costs. Given 512 or less samples, MPPI would consistently crash, while DMPO is able to successfully complete the task with as few as 64 samples. We found that increasing the number of samples for MPPI only helps to a point, after which performance can suffer. However, increasing the number of samples generally improves the median performance of DMPO. And DMPO with 1024 samples outperforms the best MPPI controller (with 4096 samples) by 7% and E2E by 19% in terms

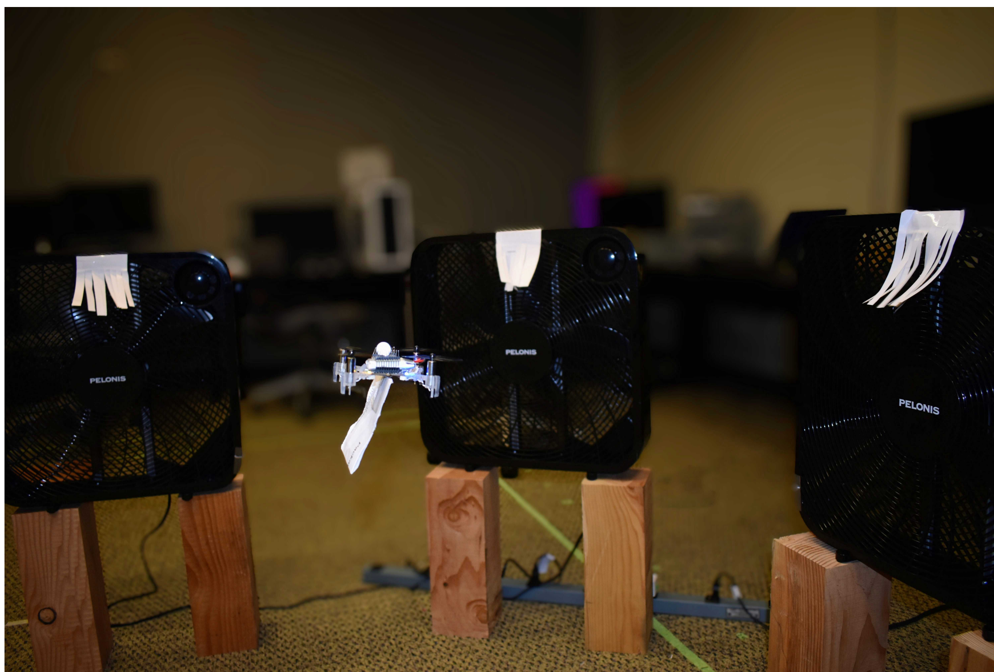


Figure 7.3: Picture of the quadrotor with the attached plate and additional wind field.

of the median error. Moreover, `E2E` has a substantially greater variance than both `DMPO` or `MPPI`, except with few samples.

In order to gauge whether `DMPO` retains the robustness of MPC, we test the Crazyflie in a scenario with an unknown wind field generated by three fans. Additionally, we attached a soft cardboard plate hanging below the chassis, which creates drag and adds additional mass (see Figure 7.3). The combination of the fans and cardboard plate creates highly dynamic and state-dependent disturbances which are not encountered during training. We report the results of these perturbations in Figure 7.4. The performance of all three controllers got worse, although `DMPO` can still remain in the air with as few as 64 samples. `DMPO` with 1024 samples nearly matches the performance of the best `MPPI` controller with 8192 samples. And it still surpasses the performance of `E2E` by 14%, with the improvement growing with samples. This illustrates how the structure of MPC can be useful for improved robustness.

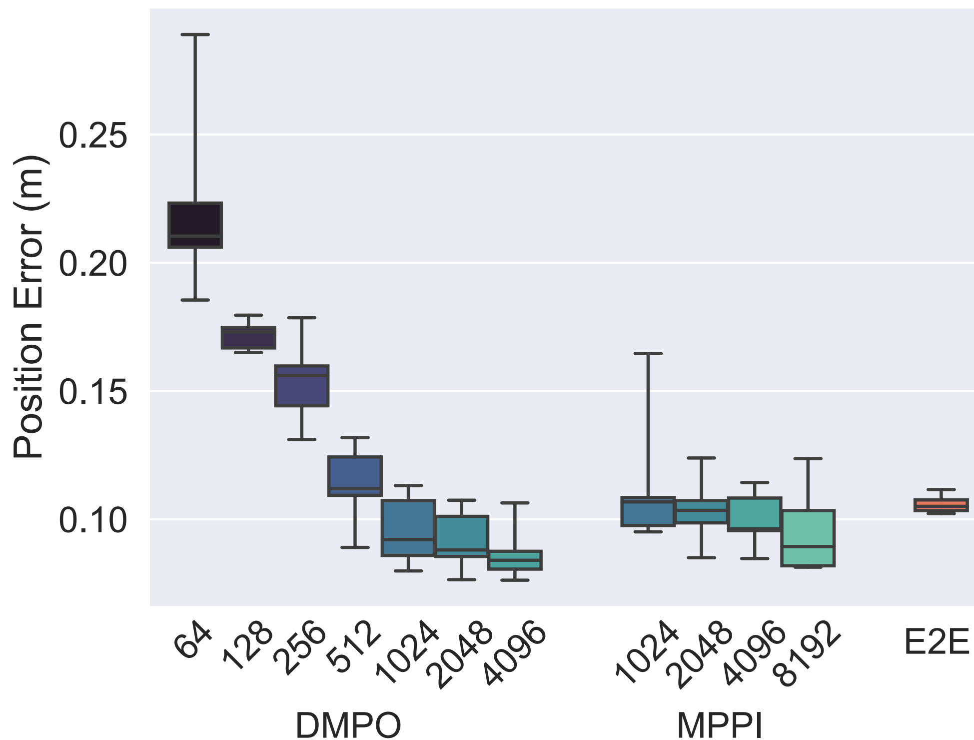


Figure 7.4: Position tracking error of `DMPO` versus `MPPI` and `E2E` on tracking random infeasible zig-zag trajectories with an attached plate and wind.

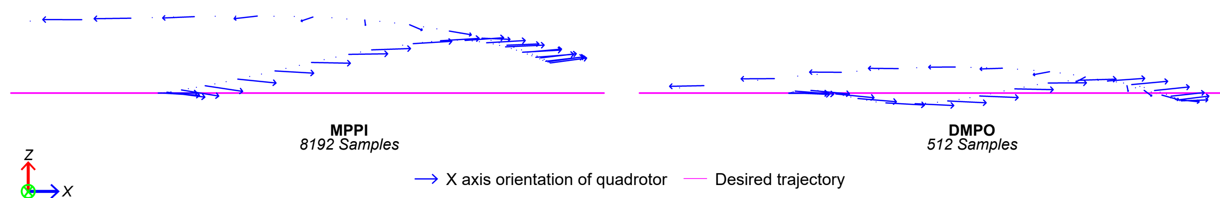


Figure 7.5: Example zig-zag trajectory with a 180° yaw flip performed by `MPPI` (8192 samples) and `DMPO` (512 samples).

7.3.3 Tracking Performance on Yaw Flips

Next, we evaluate performance on zig-zags with yaw flips. The `E2E` baseline could not successfully transfer to the real system on this task. The left of Figure 7.6 reports the

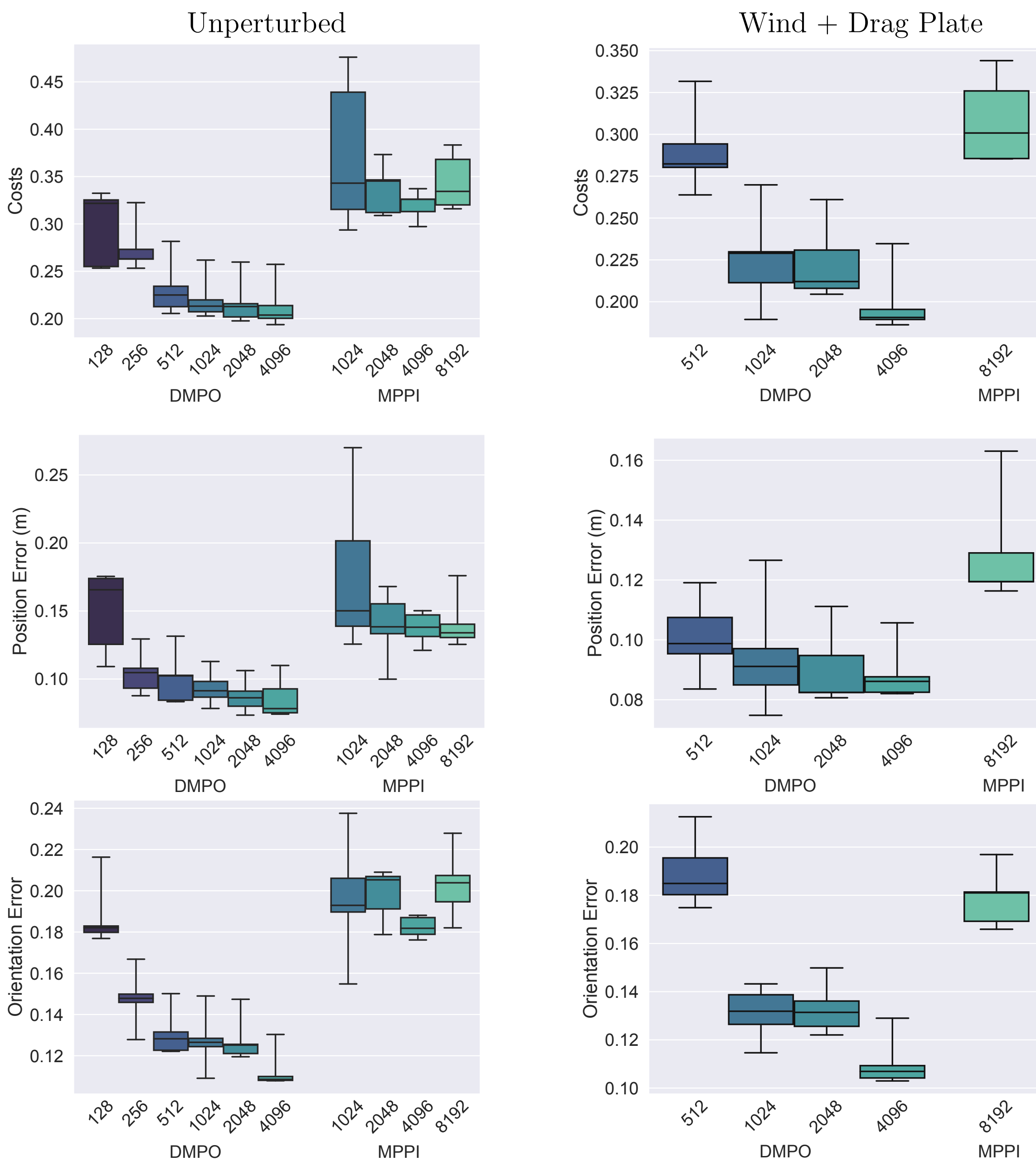


Figure 7.6: Total cost (top), position error in meters (middle), and orientation error (bottom) of `DMPO` versus `MPPI` on tracking random yaw flip trajectories without any environmental disturbances (left) and with an attached plate and wind (right).

Table 7.1: Relative memory usage between MPPI with 4096 samples and DMPO.

# DMPO Samples	256	512	1024	2048
Memory Reduction	4.3×	3.1×	1.9×	1.1×

performance of DMPO and MPPI without the presence of additional disturbances. Again, with 512 samples or fewer, MPPI would consistently crash, while DMPO with as few as 128 can successfully stay in the air. At 256 samples, DMPO outperforms the best MPPI controller with 4096 samples by over 27%. And in this much harder scenario, DMPO with 4096 samples is 64% better than MPPI. Breaking down the cost, we see that DMPO with 256 or more samples does a much better job in tracking both desired position and orientation. Figure 7.5 illustrates an example trajectory and how DMPO has much better position tracking (especially in the Z direction) and rotates more rapidly to improving orientation tracking.

We report the perturbation results on the right of Figure 7.6. In this scenario, MPPI needed 8192 samples to avoid crashing, while DMPO remained robust at 512 samples, outperforming MPPI by 7%. And given 4096 samples, DMPO is over 57% better than MPPI. We again see that position error is substantially lower for DMPO. In contrast, DMPO at 512 samples was slightly worse than MPPI for orientation error. Yet, DMPO quickly got better at tracking orientation with 1024 or more samples. Comparing DMPO in this case to the unperturbed scenario, we see that its performance is markedly similar. In fact, it only got worse by about 7% on average, while MPPI incurred about 11% more cost. The median position and orientation errors are only slightly larger with disturbances, except for the 512-sample DMPO, which had substantially worse orientation tracking. Together, these results indicate the zero-shot generalization capability of DMPO in the presence of unknown disturbances. Additional results can be found at https://tinyurl.com/mr2ywmnw.

7.3.4 Compute Requirements

DMPO with 256 samples is 1.2× faster than MPPI with 4096 samples on our offboard computer while outperforming it on the yaw flip task. And the savings may be even greater for on-board

compute which is more constrained. We report the memory usage of `DMPO` for various samples compared to `MPPI` with 4096 samples in Table 7.1. With 256 samples, `DMPO` requires $4.3\times$ less memory. Altogether, this means that we can achieve better performance while using less compute and memory compared to `MPPI`. However, we will always be slower than `E2E` due to the need for rollouts.

7.4 Discussion

We devised `DMPO`, a method for jointly learning the optimizer and warm-starting procedure for MPC. By framing the optimizer as a policy in an auxiliary MDP, we showed how MPC could be treated as a structured policy class and learned via MFRL. We evaluated `DMPO` on a real quadrotor platform tracking infeasible zig-zag trajectories and showed it can outperform `E2E` and `MPPI` controllers with far fewer samples. And `DMPO` is even more robust than `MPPI` to unseen disturbances, such as unknown wind fields and an attached cardboard drag plate. Moreover, since `DMPO` can accomplish this level of performance with fewer samples, it can save up to $4.3\times$ memory and reduce runtime by $1.2\times$ compared to `MPPI`. This indicates `DMPO` is a viable strategy to leverage the robustness of MPC while improving upon these hand-designed controllers and better match the optimal policy.

However, DMPO does have some limitations. Each optimizer is specific to a choice of horizon, number of samples, and system. Therefore, these components cannot be directly transferred to new robots or alternate choices of hyperparameters. Future work could remedy these limitations with novel architectural innovations and training the optimizers across a range of hyperparameters, tasks, and even potentially robots. Furthermore, we had to use domain randomization to train DMPO due to a mismatch between our simulator and the real Crazyflie. With better modeling, we could potentially improve the performance of our learned policies, as well as the baseline MPPI controller. Finally, as with any MFRL method, there is a fairly high variance in the performance of the generated controllers. And sim-to-real transfer can be sensitive to the hyperparameters of PPO, domain randomization distributions, and choice of reward.

Chapter 8
CONCLUSION

In this thesis, we explored multiple avenues for improving the performance of model-based control with learning. Specifically, we treated MPC as both a structured policy class which can be directly optimized for performance and a learning algorithm in its own right. Rather than treat the model and cost function as fixed components of the MPC pipeline, we proposed to jointly learn them as parameterized modules in an end-to-end fashion. We demonstrated in simulation that models learned this way outperform vanilla system identification, illustrating that our technique is a viable strategy for overcoming the objective mismatch problem. We then showed that the optimization process in MPC actually defines an online learning algorithm and proposed dynamic mirror descent as a unified framework for designing MPC strategies. Rather than hand-design the optimizer in MPC, we then proposed to learn the update rule via imitation learning to improve performance under computational constraints. We also explored learning the sampling distribution as an alternative strategy for improving the optimization process of an MPC controller. Specifically, we showed the advantages to performing all optimization in the latent space of a deep generative model which was directly optimized for controller performance. Finally, we extended our work on learned optimization to learn both update rules and warm-starting procedures with reinforcement learning. This enabled learned controllers to surpass both the best hand-designed controllers and end-to-end policies on a real-world quadrotor trajectory tracking task.

We have thus shown the importance of learning components of MPC by directly optimizing its performance on the desired task. When learning models this way, we found they were often wrong in terms of the ground truth system. However, they were useful, in that they enabled better performance when coupled with the controller. In the context of learned optimization,

using reinforcement learning to train the update rule enabled the controllers to surpass the best hand-designed optimization algorithm. This is in contrast to imitation learning, which trains the controllers with a surrogate loss that limits their ultimate performance to that of the expert demonstrations. When learning sampling distributions, directly optimizing for task performance in an end-to-end fashion enabled us to outperform a baseline method which learned a distribution in a way that was agnostic to how it would be used during run time. Additionally, these results illustrate the benefit of the structure in MPC for policy learning. When learning the model and cost function end-to-end, we were able to achieve good performance with substantially less data than a neural network policy. And our learned optimizers similarly outperformed end-to-end policies while requiring less interactions with the simulator during training. More importantly, our method retained the robustness of MPC, transferring zero-shot to difficult unknown perturbations. Therefore, the structure of MPC not only improves sample efficiency, but also aids in the robustness of the learned policy. These properties are critical for incorporating learning in real-world robotic systems.

Finally, there are some clear directions for future work that have emerged through these results. Right now, we tackle learning the model and cost function, the update rule, and the sampling distribution in isolation. One thrust would be to unify learning these different components under a common framework. It is also not clear when to use which method and which components are most critical for improved performance. An exploratory study that investigates each design choice on a real world robotic system would help elucidate when to use what approach. Another strategy would be to incorporate more structure in the learned optimizers to better determine 1) what they are doing to improve performance and 2) isolate which attributes are most important. Such insights could also potentially allow us to distill common learned strategies into a closed-form symbolic update. Additionally, the learned optimizers are currently specific to the number of samples we choose. And both the learned optimizers and sampling distributions are valid only for a fixed horizon. An interesting future direction would be to explore architectural design choices that enable flexibility in the number of samples and length of the horizon without requiring retraining. Finally, using reinforcement

learning to train the optimizer is largely feasible when we have access to a good simulator. One path around this would be to explore learned MPC controllers as a policy class in offline reinforcement learning. Therefore, we could collect data using an existing MPC algorithm on the real system and leverage this data to improve the optimizer offline without requiring additional interactions with the environment. Another strategy would be to formulate the learned optimizer in such a way that we can guarantee performance is lower bounded by the controller on which we are learning the residual. Both approaches would take advantage of the fact that we already have a functional MPC controller prior to learning, enabling us to avoid starting from scratch.

BIBLIOGRAPHY

[1] Marcin Andrychowicz OpenAI, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W Pachocki, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, et al. Learning Dexterous In-Hand Manipulation. *arXiv preprint arXiv:1808.00177*, 2018.

[2] Kevin Huang, Rwik Rana, Guanya Shi, Alexander Spitzer, and Byron Boots. DATT: Deep Adaptive Trajectory Tracking for Quadrotor Control. In *Conference on Robot Learning (CoRL)*, 2023.

[3] Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Champion-Level Drone Racing Using Deep Reinforcement Learning. *Nature*, 620(7976):982–987, 2023.

[4] Michael O'Connell, Guanya Shi, Xichen Shi, Kamyar Azizzadenesheli, Anima Anandkumar, Yisong Yue, and Soon-Jo Chung. Neural-fly enables rapid learning for agile flight in strong winds. *Science Robotics*, 7(66):eabm6597, 2022.

[5] Guanya Shi, Xichen Shi, Michael O'Connell, Rose Yu, Kamyar Azizzadenesheli, Animashree Anandkumar, Yisong Yue, and Soon-Jo Chung. Neural lander: Stable drone landing control using learned dynamics. In *2019 international conference on robotics and automation (icra)*, pages 9784–9790. IEEE, 2019.

[6] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017.

[7] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-Real Transfer of Robotic Control with Dynamics Randomization. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3803–3810. IEEE, 2018.

[8] Xiaoyu Chen, Jiachen Hu, Chi Jin, Lihong Li, and Liwei Wang. Understanding Domain Randomization for Sim-to-Real Transfer. *arXiv preprint arXiv:2110.03239*, 2021.

[9] Manfred Morari and Jay H Lee. Model predictive control: past, present and future. *Computers & chemical engineering*, 23(4-5):667–682, 1999.

[10] Grady Williams, Nolan Wagener, Brian Goldfain, Paul Drews, James M Rehg, Byron Boots, and Evangelos A Theodorou. Information theoretic MPC for model-based reinforcement learning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1714–1721. IEEE, 2017.

[11] Chenkai Yu, Guanya Shi, Soon-Jo Chung, Yisong Yue, and Adam Wierman. The power of predictions in online control. *Advances in Neural Information Processing Systems*, 33:1994–2004, 2020.

[12] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous Helicopter Aerobatics through Apprenticeship Learning. *The International Journal of Robotics Research (IJRR)*, 29(13):1608–1639, 2010.

[13] Vikash Kumar, Emanuel Todorov, and Sergey Levine. Optimal Control with Learned Local Models: Application to Dexterous Manipulation. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 378–383. IEEE, 2016.

[14] Chelsea Finn and Sergey Levine. Deep Visual Foresight for Planning Robot Motion. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2786–2793. IEEE, 2017.

[15] Mohak Bhardwaj, Balakumar Sundaralingam, Arsalan Mousavian, Nathan Ratliff, Dieter Fox, Fabio Ramos, and Byron Boots. STORM: An Integrated Framework for

Fast Joint-Space Model-Predictive Control for Reactive Manipulation. *arXiv preprint arXiv:2104.13542*, 2021.

[16] Tom Erez, Kendall Lowrey, Yuval Tassa, Vikash Kumar, Svetoslav Kolev, and Emanuel Todorov. An integrated system for real-time Model Predictive Control of humanoid robots. In *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 292–299. IEEE, 2013.

[17] Zackory Erickson, Henry M Clever, Greg Turk, C Karen Liu, and Charles C Kemp. Deep Haptic Model Predictive Control for Robot-Assisted Dressing. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4437–4444. IEEE, 2018.

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

[19] Nolan Wagener, Ching-An Cheng, Jacob Sacks, and Byron Boots. An Online Learning Approach to Model Predictive Control. *arXiv preprint arXiv:1902.08967*, 2019.

[20] Tom Erez, Yuval Tassa, and Emanuel Todorov. Infinite-Horizon Model Predictive Control for Periodic Tasks with Contacts. *Robotics: Science and systems VII*, page 73, 2012.

[21] Nan Jiang, Alex Kulesza, Satinder Singh, and Richard Lewis. The Dependence of Effective Planning Horizon on Model Accuracy. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1181–1189, 2015.

[22] Aviv Tamar, Garrett Thomas, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Learning from the Hindsight Plan — Episodic MPC Improvement. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 336–343. IEEE, 2017.

[23] Tongxin Li, Ruixiao Yang, Guannan Qu, Guanya Shi, Chenkai Yu, Adam Wierman, and Steven Low. Robustness and consistency in linear quadratic control with untrusted predictions. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–35, 2022.

[24] Avik Jain, Lawrence Chan, Daniel S Brown, and Anca D Dragan. Optimal Cost Design for Model Predictive Control. In *Learning for Dynamics and Control*, pages 1205–1217. PMLR, 2021.

[25] Ian Lenz, Ross A Knepper, and Ashutosh Saxena. DeepMPC: Learning Deep Latent Features for Model Predictive Control. In *Robotics: Science and Systems (R:SS)*. Rome, Italy, 2015.

[26] Juš Kocijan, Roderick Murray-Smith, Carl Edward Rasmussen, and Agathe Girard. Gaussian process model based predictive control. In *American Control Conference (ACC)*, volume 3, pages 2214–2219. IEEE, 2004.

[27] Justin Fu, Sergey Levine, and Pieter Abbeel. One-Shot Learning of Manipulation Skills with Online Dynamics Adaptation and Neural Network Priors. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4019–4026. IEEE, 2016.

[28] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models. *arXiv preprint arXiv:1805.12114*, 2018.

[29] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[30] Niklas Wahlström, Thomas B Schön, and Marc Peter Deisenroth. From Pixels to Torques: Policy Learning with Deep Dynamical Models. *arXiv preprint arXiv:1502.02251*, 2015.

[31] Manuel Watter, Jost Tobias Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images. *arXiv preprint arXiv:1506.07365*, 2015.

[32] Ershad Banijamali, Rui Shu, Hung Bui, and Ali Ghodsi. Robust Locally-Linear Controllable Embedding. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1751–1759. PMLR, 2018.

[33] Frederik Ebert, Chelsea Finn, Sudeep Dasari, Annie Xie, Alex Lee, and Sergey Levine. Visual Foresight: Model-Based Deep Reinforcement Learning for Vision-Based Robotic Control. *arXiv preprint arXiv:1812.00568*, 2018.

[34] Jung-Su Ha, Young-Jin Park, Hyeok-Joo Chae, Soon-Seo Park, and Han-Lim Choi. Adaptive Path-Integral Autoencoder: Representation Learning and Planning for Dynamical Systems. *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[35] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning Latent Dynamics for Planning from Pixels. In *International Conference on Machine Learning (ICML)*, pages 2555–2565. PMLR, 2019.

[36] Nathan Lambert, Brandon Amos, Omry Yadan, and Roberto Calandra. Objective Mismatch in Model-Based Reinforcement Learning. 2020.

[37] Mingyuan Zhong, Mikala Johnson, Yuval Tassa, Tom Erez, and Emanuel Todorov. Value Function Approximation and Model Predictive Control. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 100–107. IEEE, 2013.

[38] Ugo Rosolia and Francesco Borrelli. Learning Model Predictive Control for Iterative Tasks. A Data-Driven Control Framework. *IEEE Transactions on Automatic Control*, 63(7):1883–1896, 2017.

[39] Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. Plan Online, Learn Offline: Efficient Learning and Exploration via Model-Based Control. *arXiv preprint arXiv:1811.01848*, 2018.

[40] Mohak Bhardwaj, Sanjiban Choudhury, and Byron Boots. Blending MPC & Value Function Approximation for Efficient Reinforcement Learning. *arXiv preprint arXiv:2012.05909*, 2020.

[41] Mohak Bhardwaj, Ankur Handa, Dieter Fox, and Byron Boots. Information Theoretic Model Predictive Q-Learning. In *Learning for Dynamics & Control (L4DC)*, pages 840–850. PMLR, 2020.

[42] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[43] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In *International Conference on Machine Learning (ICML)*, pages 1889–1897. PMLR, 2015.

[44] Weiwei Li and Emanuel Todorov. Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems. In *International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, pages 222–229. Citeseer, 2004.

[45] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-Limited Differential Dynamic Programming. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175. IEEE, 2014.

[46] Brandon Amos and Denis Yarats. The Differentiable Cross-Entropy Method. In *International Conference on Machine Learning (ICML)*, pages 291–302. PMLR, 2020.

[47] Peter Karkus, David Hsu, and Wee Sun Lee. QMDP-Net: Deep Learning for Planning under Partial Observability. *arXiv preprint arXiv:1703.06692*, 2017.

[48] Masashi Okada, Luca Rigazio, and Takenobu Aoshima. Path Integral Networks: End-to-End Differentiable Optimal Control. *arXiv preprint arXiv:1706.09597*, 2017.

[49] Brandon Amos, Ivan Dario Jimenez Rodriguez, Jacob Sacks, Byron Boots, and J Zico Kolter. Differentiable MPC for End-to-end Planning and Control. *arXiv preprint arXiv:1810.13400*, 2018.

[50] Masashi Okada and Tadahiro Taniguchi. Acceleration of Gradient-based Path Integral Method for Efficient Optimal and Inverse Optimal Control. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3013–3020. IEEE, 2018.

[51] Marcus Pereira, David D Fan, Gabriel Nakajima An, and Evangelos Theodorou. MPC-Inspired Neural Network Policies for Sequential Decision Making. *arXiv preprint arXiv:1802.05803*, 2018.

[52] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value Iteration Networks. *arXiv preprint arXiv:1602.02867*, 2016.

[53] Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Universal Planning Networks: Learning Generalizable Representations for Visuomotor Control. In *International Conference on Machine Learning (ICML)*, pages 4732–4741. PMLR, 2018.

[54] Tianhe Yu, Gleb Shevchuk, Dorsa Sadigh, and Chelsea Finn. Unsupervised Visuomotor Control through Distributional Planning Networks. *arXiv preprint arXiv:1902.05542*, 2019.

[55] Mohak Bhardwaj, Byron Boots, and Mustafa Mukadam. Differentiable Gaussian Process Motion Planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 10598–10604. IEEE, 2020.

[56] Shubhankar Agarwal, Harshit Sikchi, Cole Gulino, and Eric Wilkinson. Imitative Planning using Conditional Normalizing Flow. *arXiv preprint arXiv:2007.16162*, 2020.

[57] Tingwu Wang and Jimmy Ba. Exploring Model-Based Planning with Policy Networks. *arXiv preprint arXiv:1906.08649*, 2019.

[58] Thomas Power and Dmitry Berenson. Variational Inference MPC for Robot Motion with Normalizing Flows. *Advances in Neural Information Processing Systems (NeurIPS) Workshop on Robot Learning: Self-Supervised and Lifelong Learning*, 2021.

[59] Thomas Power and Dmitry Berenson. Variational Inference MPC using Normalizing Flows and Out-of-Distribution Projection. *arXiv preprint arXiv:2205.04667*, 2022.

[60] Brandon Amos and J Zico Kolter. OptNet: Differentiable Optimization as a Layer in Neural Networks. In *International Conference on Machine Learning (ICML)*, pages 136–145. PMLR, 2017.

[61] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)*, 32:8026–8037, 2019.

[62] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of recent magnitude. *Coursera: Neural Networks for Machine Learning*, 2012.

[63] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[64] Elad Hazan. Introduction to Online Convex Optimization. *Foundations and Trends® in Optimization*, 2(3-4):157–325, 2016.

[65] Eric Hall and Rebecca Willett. Dynamical Models and Tracking Regret in Online Convex Programming. In *International Conference on Machine Learning (ICML)*, pages 579–587. PMLR, 2013.

[66] Arindam Banerjee, Srujana Merugu, Inderjit S Dhillon, Joydeep Ghosh, and John Lafferty. Clustering with Bregman divergences. *Journal of machine learning research*, 6(10), 2005.

[67] Peter W Glynn. Likelihood Ratio Gradient Estimation for Stochastic Systems. *Communications of the ACM*, 33(10):75–84, 1990.

[68] Zdravko I Botev, Dirk P Kroese, Reuven Y Rubinstein, and Pierre L'Ecuyer. The Cross-Entropy Method for Optimization. In *Handbook of Statistics*, volume 31, pages 35–59. Elsevier, 2013.

[69] Bart van den Broek, Wim Wiegerinck, and Hilbert Kappen. Risk Sensitive Path Integral Control. *arXiv preprint arXiv:1203.3523*, 2012.

[70] Herman Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952.

[71] Youhei Akimoto, Yuichi Nagata, Isao Ono, and Shigenobu Kobayashi. Theoretical Foundation for CMA-ES from Information Geometry Perspective. *Algorithmica*, 64:698–716, 2012.

[72] Frank Nielsen and Vincent Garcia. Statistical Exponential Families: A Digest With Flash Cards. *arXiv preprint arXiv:0911.4863*, 2009.

[73] Richard Bellman, John Casti, et al. Differential Quadrature and Long-Term Integration. *Journal of mathematical analysis and Applications*, 34(2):235–238, 1971.

[74] Brian Goldfain, Paul Drews, Changxi You, Matthew Barulic, Orlin Velev, Panagiotis Tsiotras, and James M Rehg. AutoRally: An Open Platform for Aggressive Autonomous Driving. *IEEE Control Systems Magazine*, 39(1):26–55, 2019.

[75] Ning Qian. On the Momentum Term in Gradient Descent Learning Algorithms. *Neural Networks*, 12(1):145–151, 1999.

[76] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12(7), 2011.

[77] Matthew D Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv preprint arXiv:1212.5701*, 2012.

[78] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the Convergence of Adam and Beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[79] Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to Optimize: A Primer and A Benchmark. *arXiv preprint arXiv:2103.12828*, 2021.

[80] Jacob Sacks and Byron Boots. Learning to Optimize in Model Predictive Control. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 10549–10556. IEEE, 2022.

[81] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3981–3989, 2016.

[82] Sachin Ravi and Hugo Larochelle. Optimization as a Model for Few-Shot Learning. *International Conference on Learning Representations (ICLR)*, 2016.

[83] Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning Gradient Descent: Better Generalization and Longer Horizons. In *International Conference on Machine Learning (ICML)*, pages 2247–2255. PMLR, 2017.

[84] Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando Freitas. Learning to Learn without Gradient Descent by Gradient Descent. In *International Conference on Machine Learning (ICML)*, pages 748–756. PMLR, 2017.

[85] Tianlong Chen, Weiyi Zhang, Zhou Jingyang, Shiyu Chang, Sijia Liu, Lisa Amini, and Zhangyang Wang. Training Stronger Baselines for Learning to Optimize. *Advances in Neural Information Processing Systems (NeurIPS)*, 33, 2020.

[86] Ke Li and Jitendra Malik. Learning to Optimize. *arXiv preprint arXiv:1606.01885*, 2016.

[87] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[88] Ke Li and Jitendra Malik. Learning to Optimize Neural Nets. *arXiv preprint arXiv:1703.00441*, 2017.

[89] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural Optimizer Search with Reinforcement Learning. In *International Conference on Machine Learning (ICML)*, pages 459–468. PMLR, 2017.

[90] John H Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, 1964.

[91] Harald Niederreiter. *Random Number Generation and quasi-Monte Carlo Methods*. SIAM, 1992.

[92] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078*, 2014.

[93] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional Image Generation with PixelCNN Decoders. *arXiv preprint arXiv:1606.05328*, 2016.

[94] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[95] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning. *arXiv preprint arXiv:2108.10470*, 2021.

[96] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[97] Masashi Okada and Tadahiro Taniguchi. Variational Inference MPC for Bayesian Model-based Reinforcement Learning. In *Conference on Robot Learning (CoRL)*, pages 258–272. PMLR, 2020.

[98] Alexander Lambert, Adam Fishman, Dieter Fox, Byron Boots, and Fabio Ramos. Stein Variational Model Predictive Control. *arXiv preprint arXiv:2011.07641*, 2020.

[99] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear Independent Components Estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[100] Danilo Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows. In *International Conference on Machine Learning (ICML)*, pages 1530–1538. PMLR, 2015.

[101] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. *arXiv preprint arXiv:1605.08803*, 2016.

[102] Jacob Sacks and Byron Boots. Learning Sampling Distributions for Model Predictive Control. In *Conference on Robot Learning (CoRL)*, pages 1733–1742. PMLR, 2023.

[103] Jonathan F Bard. *Practical Bilevel Optimization: Algorithms and Applications*, volume 30. Springer Science & Business Media, 2013.

[104] Amirreza Shaban, Ching-An Cheng, Nathan Hatch, and Byron Boots. Truncated Back-propagation for Bilevel Optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1723–1732. PMLR, 2019.

[105] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer Normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[106] Jacob Sacks, Rwik Rana, Kevin Huang, Alex Spitzer, Guanya Shi, and Byron Boots. Deep Model Predictive Optimization. *arXiv preprint arXiv:2310.04590*, 2023.

[107] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438*, 2015.