This paper has been accepted for publication in the
IEEE Transactions on Robotics (T-RO), 2021. ©IEEE

Policy Search for Model Predictive Control with Application to Agile Drone Flight

Yunlong Song, Davide Scaramuzza

***Abstract*—Policy Search and Model Predictive Control (MPC) are two different paradigms for robot control: policy search has the strength of automatically learning complex policies using experienced data, while MPC can offer optimal control performance using models and trajectory optimization. An open research question is how to leverage and combine the advantages of both approaches. In this work, we provide an answer by using policy search for automatically choosing high-level decision variables for MPC, which leads to a novel *policy-search-for-model-predictive-control framework*. Specifically, we formulate the MPC as a parameterized controller, where the hard-to-optimize decision variables are represented as high-level policies. Such a formulation allows optimizing policies in a self-supervised fashion. We validate this framework by focusing on a challenging problem in agile drone flight: flying a quadrotor through fast-moving gates. Experiments show that our controller achieves robust and real-time control performance in both simulation and the real world. The proposed framework offers a new perspective for merging learning and control.**

Code: https://uzh-rpg.github.io/high_mpc
Video: https://youtu.be/Qei7oGiEIxY

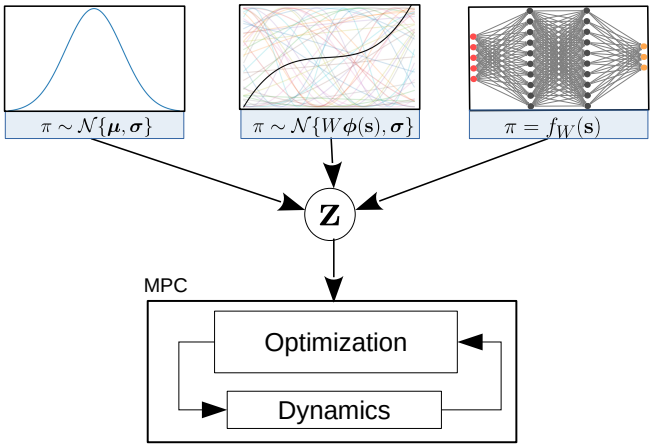

Fig. 1: An overview of the proposed *policy-search-for-model-predictive-control framework*.

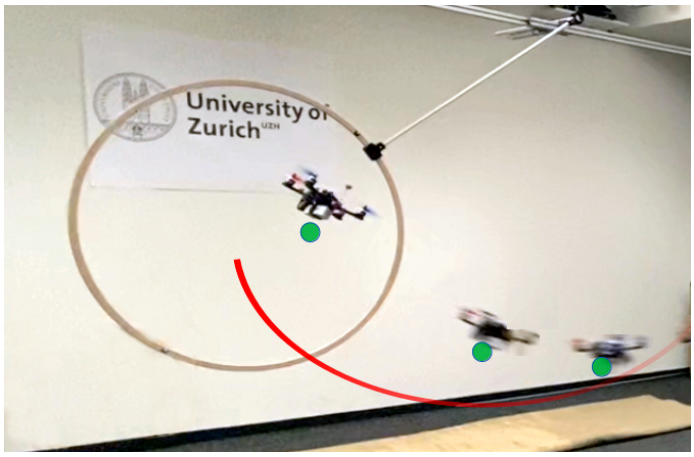

Fig. 2: An application of the proposed method for flying a quadrotor through a fast moving gate.

I. Introduction

Mobile robots operate in a dynamic world. Notably, quadrotors are agile robots that can navigate at high speeds in highly complex and dynamic environments otherwise inaccessible to humans. However, sudden environmental changes, like dynamic obstacles, can raise fundamental problems for the vehicle control since they require the vehicle to have fast reactions and replan its trajectory quickly.

An essential requirement for agile drone flight in dynamic environments is to adapt the vehicle trajectory rapidly depending on the environmental changes. State-of-the-art model-based approaches have shown to be effective for controlling the quadrotor in both static and dynamic environments [1]–[10]. For example, in the context of drone racing [11]–[13], the drone has to fly through a sequence of static gates (subjected to small disturbance) at extremely high speeds.

Model predictive control (MPC) [14] has been shown to be a powerful model-based approach for solving complex quadrotor control problems [3], [15]–[18]. For example, the perception-aware MPC [15] is a framework that unifies both planning and perception objectives. MPC is increasingly gaining popularity in many robotic domains, thanks to its capability of simultaneously dealing with complex nonlinear dynamic systems while satisfying different state and input constraints.

Despite the successes, many MPC applications still experience significant challenges, such as the requirement of an accurate mathematical model and the necessity of solving trajectory optimization problems online with the limited computational power of small-scale computers. In practice, the closed-loop performance of MPC for a specific task is sensitive to several design choices, including cost function formulation,

The authors are with the Robotics and Perception Group, Department of Informatics, University of Zurich, and Department of Neuroinformatics, University of Zurich and ETH Zurich, Switzerland (http://rpg.ifi.uzh.ch). This work was supported by the National Centre of Competence in Research (NCCR) Robotics through the Swiss National Science Foundation (SNSF) and the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 871479 (AERIAL-CORE) and the European Research Council (ERC) under grant agreement No. 864042 (AGILEFLIGHT).

hyperparameters, and the prediction horizon. As a result, a series of approximations, heuristics, and parameter tuning is employed, producing sub-optimal solutions.

On the other hand, reinforcement learning (RL) [19] methods, like policy search, allow solving continuous control problems with minimum prior knowledge about the task. The key idea of RL is to automatically train the policy via trial and error and maximize the task performance measured by the given reward function. While RL has achieved impressive results in solving a wide range of robot control tasks [20]–[23], the lack of interpretability of an end-to-end controller trained using RL is of significant concern by the control community [24].

Ideally, the control framework should be able to combine the advantages of both methods—the ability of model-based controllers, like MPC, to safely control a physical robot using the well-established knowledge in dynamic modeling and optimization and the power of RL to learn complex policies using experienced data automatically. Therefore, the resulting control framework can handle large-scale inputs, reduce human-in-the-loop design and tuning, and eventually achieve adaptive and optimal control performance. Despite these valuable features, designing such a system remains a significant challenge.

To this end, one line of research in the learning community has been focusing on developing data-efficient policy search methods using model priors. For instance, guided policy search (GPS) algorithms [25]–[27] opt for transforming RL into a supervised learning problem. The key idea in GPS is to use a trajectory optimization algorithm to collect training data for training neural networks via supervised learning. However, these methods still learn black-box control policies that suffer from poor generalizations. The second trend pertains to learning-based MPC [28]–[31], which can leverage real-world data to improve dynamics modeling and use model predictive path integral control (MPPI) [32] for optimization. Such algorithms generally have their roots in stochastic optimal control and require sampling a large amount of data in real-time for optimization, making those methods computationally expensive.

Contributions

In this work, we propose a new paradigm for merging learning and control: learning high-level policies for model predictive control using policy search. An overview of our approach is summarized in Fig. 1. Specifically, we consider the MPC as a parameterized controller and formulate the search of high-level decision variables for MPC as a probabilistic policy search problem. First, we use two general Gaussian policies for modeling the high-level decision variables and show that the policy updates have closed-form solutions. Second, we propose a self-supervised training method for learning neural network policies. Our key insight is that policy search is useful for making high-level decisions for MPC, allowing automatically learning and adapting hard-to-optimize parameters.

On the experiment side, we evaluate our approach by addressing a challenging problem towards autonomous agile drone flight in dynamic environments: controlling a quadrotor to fly through a sequence of fast-moving gates. The key advantage of our approach compared to the standard MPC formulation is that the desired traversal time, which is hard to optimize simultaneously with other state variables, can be learned offline and can be adaptively selected at runtime. The resulting controller, which consists of a trained neural network policy and an MPC, achieves real-time control performance of a physical drone. An illustration of the real-world experiment is shown in Fig. 2.

This work is an extension of our previous conference paper [33]. The earlier version of this work proposed learning Gaussian and neural network policies and demonstrated learning a single time variable for flying a quadrotor through a dynamic gate in simulation. In this paper, we additionally 1) introduce a new algorithm for learning a Gaussian linear policy, 2) demonstrate that our approach is a general framework that can learn multidimensional decision variables, not just a single variable, 3) demonstrate that our controller can control a drone to fly through multiple gates in simulation and outperforms a standard MPC and a trajectory sampling method, 4) deploy the algorithm on a physical drone and show that the trained neural network high-level policy can be transferred to the real world without fine-tuning.

II. Related Work

A. Policy Search for Robotics

Policy search [19] is a central area of reinforcement learning concerned with how to find an optimal parametric policy by maximizing the expected return of sampled trajectories. Depending on their exploration strategies for the stochastic trajectory generation, policy search methods can be categorized into step-based and episode-based methods [19], [34]. Most variations of policy search methods make use of step-based exploration strategies by adding different exploration noise in the *action space* at each control time step. Step-based policy search algorithms [35]–[38] are widely used for continuous control tasks, ranging from learning agile motor skills for legged robots [39] to controlling a simulated race car at its friction limits [22]. They learn end-to-end black-box control policies that can map observations directly to control commands.

By contrast, episode-based policy search methods [34], [40]–[42] add exploration noise in the *parameter space* of the policy only at the beginning of the episode. In particular, episode-based methods are widely used for learning movement primitives [43]–[45], which are compact parameterizations of the robot's control policy. For example, the task-parameterized dynamic motor primitives (DMPs) [43], [46] are popular compact policy representations in robotics. Adjusting their parameters allows robots to learn new skills quickly and solve many challenging robot control problems, such as playing *Baseball* [47], *Ball-in-the-cup* [48], and *Table Tennis* [49]. Episode-based policy search methods help learn compact skills representations that are not easy to model by human experts.

B. Data-driven Control with Model Predictive Control

1) MPC-guided Policy Search: Model-free policy search algorithms learn control policies via trial-and-error; however, they suffer from high sample complexities. Guided policy search [50] converts model-free policy search to supervised learning by iteratively collecting the training data using offline trajectory optimization [26], [50]–[52] or model predictive control [25], [27]. A key advantage of guided policy search is that it effectively trains deep neural network control policies, where the policy can handle complex and high-dimensional inputs from onboard sensors. For example, a deep sensorimotor policy, trained using MPC and imitation learning, enables an autonomous quadrotor to fly extreme acrobatic maneuvers with only onboard sensing and computation [27]. However, this line of work usually only uses the model during training and results in a policy specialized in a single task. Despite all of the successes achieved by guided policy search, the lack of generalization and robustness of the end-to-end policy remains a primary challenge.

2) Learning-based MPC: In the second paradigm, learning-based MPC [28]–[31], [53] can leverage real-world data to improve dynamic modeling or learn a cost function for MPC. It allows for a more robust and flexible MPC design. In particular, sampling-based MPC [53] algorithms are developed for handling complex cost criteria and general nonlinear dynamics. This is achieved by combining neural networks for the system dynamics approximation with the model predictive path integral (MPPI) control framework [53] for real-time control optimization. A crucial requirement for the sampling-based MPC is to generate a large number of samples in real-time, where the sampling procedure is generally performed in parallel by using graphics processing units (GPUs). Hence, it is computationally and memory expensive to run sampling-based MPC on embedded systems. These methods generally focus on learning dynamics for tasks where a dynamical model of the robots or its environment is challenging to derive analytically, such as aggressive autonomous driving around a dirt track [28].

Alternatively, differentiable MPC [54] treats the MPC as a differentiable policy class for reinforcement learning. Hence, by differentiating through the optimization problem using the Karush–Kuhn–Tucker (KKT) conditions of the convex approximation at a fixed point of the controller, it can also learn the costs and dynamics of an MPC controller via end-to-end learning. The analytical derivative relies on a fixed point of the controller, which, however, often does not exist when using neural networks to approximate the dynamics [54].

3) Learning Neural Network Policies for MPC: To combine the power of neural networks and the strength of standard MPC optimization, state-of-the-art systems [55]–[57] opt for using deep neural networks as standalone representation learning modules. Specifically, a neural network is trained to process high-dimensional data, such as images, and is used to predict low-dimensional state information for the MPC. For instance, [55] proposed to combine a Convolutional Neural Network (CNN) with an MPC controller to solve the problem of navigating a quadrotor to pass through multiple gates. The trained neural network predicts three-dimensional poses

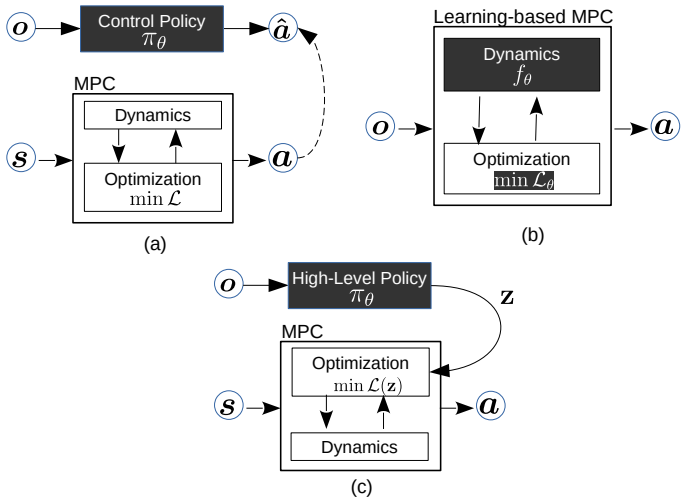

Fig. 3: Taxonomy of existing methods combining machine learning with model predictive control.

of the gate's center from image observations, and then, the MPC outputs control commands for the quadrotor to navigate through the predicted waypoints. Similarly, the method in [56] tackles an aggressive autonomous driving problem by using a CNN-based policy to predict a cost map of the track, which is then directly used for online trajectory optimization. A key advantage of this line of work is that it combines the benefits of both neural networks for high-dimensional data processing and MPC for robot control.

III. Preliminary

We introduce mathematical formulations for both MPC and policy search. We discuss three kinds of policy representations that are widely used in RL and use them to model the high-level policies.

A. Model Predictive Control

We consider the problem of controlling a nonlinear deterministic dynamical system whose dynamics is defined by a differential equation $\dot{\mathbf{x}}_t = f(\mathbf{x}_t, \mathbf{u}_t)$, where $\mathbf{x}_t \in \mathbb{R}^n$ is the state vector, $\mathbf{u}_t \in \mathbb{R}^m$ is a vector of control commands, and $\dot{\mathbf{x}}_t \in \mathbb{R}^n$ is the derivative of the current state. In MPC, we approximate the actual continuous time differential equation using a set of discrete time integration $\mathbf{x}_{h+1} = \mathbf{x}_h + d_t \cdot \hat{f}(\mathbf{x}_h, \mathbf{u}_h)$, with d_t as the time interval between consecutive states and $\hat{f}$ as an approximated dynamical model.

Let $\mathbf{r} \in \mathbb{R}^k$ be a vector of reference states, e.g., a planned trajectory. We define a vector of high-level decision variables as $\mathbf{z} \in \mathbb{R}^N$. For example, for our application (Section V.), $\mathbf{z} = [z_1, \cdots, z_N]$ defines the time variables at which the robot should be passing the corresponding gate. At every control time step t, the system is in state $\mathbf{x}_t$. MPC takes the current state $\mathbf{x}_{\text{init}} = \mathbf{x}_t$, the reference states $\mathbf{r}$, and the high-level decision vector $\mathbf{z}$ as input.

Formally, MPC minimizes a cost function over a fixed finite time horizon H by solving an optimization problem:

$$\begin{aligned} \min_{\mathbf{u}_{1:H},\mathbf{x}_{1:H}} \quad & \mathcal{L} = \sum_{h=1}^{H} c(\mathbf{x}_h, \mathbf{u}_h; \mathbf{r}, \mathbf{z}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}, \mathbf{u}) = 0, \quad \mathbf{h}(\mathbf{x}, \mathbf{u}) \leq 0 \\ & \mathbf{x}_{h+1} = \mathbf{x}_h + d_t \cdot \hat{f}(\mathbf{x}_h, \mathbf{u}_h), \quad \mathbf{x}_1 = \mathbf{x}_{\text{init}} \end{aligned} \tag{1}$$

where $\mathbf{g}(\mathbf{x}, \mathbf{u})$ represents equality constraints and $\mathbf{h}(\mathbf{x}, \mathbf{u})$ represents inequality constraints.

Our goal is to find the optimal control command $\mathbf{u}^*$ for the current state such that we can execute it and move the robot to the next state. It is achieved by solving the trajectory optimization problem in real-time and by repeating this process at every control time step. Specifically, MPC minimizes the cost in the future states and outputs an optimal trajectory ($\boldsymbol{\tau}$) that consists of a sequence of control commands and states. Only the first command $\mathbf{u}^* = \mathbf{u}_1$ is executed on the robot.

In this work, we take the MPC as a parametric controller that is parameterized by the high-level decision variables $\mathbf{z}$. Therefore, modulating the variables $\mathbf{z}$ can result in different MPC outputs, denoted as $\boldsymbol{\tau} = \text{MPC}(\mathbf{z})$. For example, in the context of flying through a dynamic gate (Section V.), $\mathbf{z}$ can be the desired time at which the vehicle passes through the gate. This formulation allows us to incorporate reinforcement learning as well as different function representations into the MPC design.

B. Policy Search

We summarize policy search by following the derivation from [34], in particular, we focus our discussion on episode-based policy search (or episodic reinforcement learning). Unlike many step-based policy search algorithms, which explore the action space by adding exploration noise directly to the policy output, episode-based policy search adds perturbations in the policy parameter space. This kind of exploration is normally added at the beginning of an episode and a reward function $R(\boldsymbol{\tau})$ is used to evaluate the quality of trajectories $\boldsymbol{\tau}$ that are generated by sampled parameters $\boldsymbol{\theta}$. A comprehensive survey and tutorial about different policy search algorithms can be found here [34], [58].

Policy search algorithms try to update the policy parameters $\boldsymbol{\theta}$ by maximizing the expected return of sampled trajectories

$$J_{\boldsymbol{\theta}} = \mathbb{E}[R(\boldsymbol{\tau})|\boldsymbol{\theta}] \approx \int R(\boldsymbol{\tau}) p_{\boldsymbol{\theta}}(\boldsymbol{\tau}) d\boldsymbol{\tau}. \tag{2}$$

A list of episode-based policy search algorithms have been discussed in the literature, such as policy gradient methods [36], [59], [60], expectation-maximization (EM) methods [48], and information-theoretic methods [38], [61].

Policy gradient methods use gradient-ascent for maximizing the expected return and are simple to implement. However, it requires manual selection of learning rates and has an unstable learning process or slow convergence. Information-theoretic approaches rely on solving constrained optimization for maximizing the objective, and at the same time, they constrain the information loss by bounding the Kullback-Leibler (KL) divergence between the new policy and the old policy. The requirement of solving constrained optimization limits the usage of information-theoretic algorithms to solve high-dimensional problems, such as optimizing neural network policies, making it challenging to implement in reality.

On the other hand, the EM-based policy search algorithms provide closed-form solutions for many commonly used policy representations, and hence, do not require the user to specify the learning rate. In addition, they provide a good trade-off between computational efficiency and sample complexity. This is realized by formulating policy search as a probabilistic inference problem with latent variables, which leads to a weighted maximum likelihood estimate. Subsequently, we can use the Expectation-Maximization algorithm to update the policy parameters. We focus on a probabilistic model in which the search for high-level decision variables in the MPC optimization is treated as a probabilistic inference problem.

C. Policy Representations

We represent the high-level policy as $\pi_{\boldsymbol{\theta}}$, which is modeled as a probability distribution or a deterministic policy (e.g., a neural network), and use the policy to select high-level decision variables $\mathbf{z} \sim \pi_{\boldsymbol{\theta}}$. Here, $\boldsymbol{\theta}$ are the policy parameters that have to be trained.

1) Gaussian Policy, Constant Mean: First, we consider a simple scenario where the goal is to find a set of fixed decision variables $\mathbf{z}$. The variables are independent of the robot's state. We use a Gaussian distribution $\mathbf{z} \sim \pi_{\boldsymbol{\theta}} = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ to represent the policy, where $\boldsymbol{\mu}$ is a mean vector and $\boldsymbol{\Sigma}$ is a diagonal covariance matrix. The covariance matrix is needed in order to incorporate exploration. Therefore, the policy parameters are $\boldsymbol{\theta} = [\boldsymbol{\mu}, \boldsymbol{\Sigma}]$.

2) Gaussian Policy, Linear Mean: Second, we consider a more general problem in which we want to find a set of adaptive decision variables, denoted as $\mathbf{z} = f(\mathbf{s})$. The decisions variables are associated with the robot's context $\mathbf{s}$. We use a Linear Gaussian model $\mathbf{z} \sim \pi_{\boldsymbol{\theta}} = \mathcal{N}(\mathbf{W}\phi(\mathbf{s}), \boldsymbol{\Sigma})$ to denote the policy, in which the Gaussian mean $\boldsymbol{\mu} = \mathbf{W}\phi(\mathbf{s})$ is represented by a linear function approximator, linear with respect to the function parameters $\mathbf{W}$. Here, $\phi : \mathbf{s} \subset \mathbb{R}^N \rightarrow \mathbb{R}^M$ is a kernel featurizer that converts the states of dimension N into a vector of features of dimension M using basis functions, such as Radial Basis Functions (RBF) [19] or Random Fourier Features (RFF) [62]. Therefore, the policy parameters are $\boldsymbol{\theta} = [\mathbf{W}, \boldsymbol{\Sigma}]$.

3) Neural Network Policy: We use a neural network $\boldsymbol{z}_t = f_{\boldsymbol{\theta}}(\mathbf{o}_t)$ as a deterministic policy representation. Here, $f_{\boldsymbol{\theta}}$ represents the neural network and $\mathbf{o}_t$ is the robot's observation at different time step t. The solution for updating the parameters $\boldsymbol{\theta}$ of a neural network in policy search is difficult to derive analytically due to the highly nonlinear property of neural networks. Many deep reinforcement learning algorithms are based on policy gradients [63], which are known to have unstable learning processes or slow convergence. By contrast, we use a self-supervised learning algorithm for training the neural network policy (Algorithm 3).

IV. Probabilistic Policy Search for MPC

A. Problem Formulation

We treat MPC as a controller $\boldsymbol{\tau} = \text{MPC}(\mathbf{z})$ that is parameterized by the high-level decision variables $\boldsymbol{z}$. Here, $\boldsymbol{\tau} = [\mathbf{u}_h, \mathbf{x}_h]_{h\in 1,\cdots,H}$ is a trajectory generated by MPC given $\mathbf{z}$, where $\mathbf{u}_h$ are control commands and $\mathbf{x}_h$ are corresponding states of the robot. By perturbing $\mathbf{z}$, MPC can result in completely different trajectories $\boldsymbol{\tau}$. To find the optimal trajectory for a given task, the optimal $\mathbf{z}$ has to be defined in advance. First, we model $\mathbf{z}$ as a high-level policy represented by a probability distribution, specifically a parameterized Gaussian distribution. Then, we optimize the policy using probabilistic policy search (or probabilistic inference) algorithms. A visualization of the inference problem is given in Fig 4 (inspired by [34]).

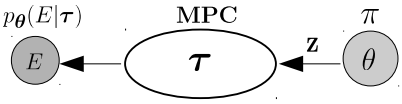

Fig. 4: A graphical model of probabilistic policy search for model predictive control.

To formulate the policy search as a latent variable inference problem, similar to [34], [64], [65], we introduce a binary "reward event" as an observed variable, denoted as $E = 1$. Maximizing the reward signal implies maximizing the probability of this "reward event". The probability of this reward event is given by $p(E|\boldsymbol{\tau}) \propto \exp\{R(\boldsymbol{\tau})\}$, where $R(\boldsymbol{\tau})$ is a reward function for evaluating the goodness of the MPC solution $\boldsymbol{\tau}$ with respect to a given evaluation metric of the task. This leads to the following maximum likelihood problem [34]:

$$\max_{\boldsymbol{\theta}} \quad \log p_{\boldsymbol{\theta}}(E=1) = \log \int_{\boldsymbol{\tau}} p(E|\boldsymbol{\tau}) p_{\boldsymbol{\theta}}(\boldsymbol{\tau}) d\boldsymbol{\tau}, \tag{3}$$

which is intractable to solve directly and can be approximated efficiently using Monte-Carlo Expectation-Maximization (MC-EM) [48], [66]. MC-EM algorithms find the maximum likelihood solution for the log marginal-likelihood (3) by introducing a variational distribution $q(\boldsymbol{\tau})$, and then, decompose the marginal log-likelihood into two terms:

$$\log p_{\boldsymbol{\theta}}(E=1) = \mathcal{L}_{\boldsymbol{\theta}}(q(\boldsymbol{\tau})) + D_{\text{KL}}(q(\boldsymbol{\tau}) || p_{\boldsymbol{\theta}}(\boldsymbol{\tau}|E)) \tag{4}$$

where the D_{KL} is the Kullback–Leibler (KL) divergence between $q(\boldsymbol{\tau})$ and the reward-weighted trajectory distribution $p_{\boldsymbol{\theta}}(\boldsymbol{\tau}|E)$. Here, $\mathcal{L}_{\boldsymbol{\theta}}(q(\boldsymbol{\tau}))$ is the lower bound of $\log p_{\boldsymbol{\theta}}(E=1)$ as $D_{\text{KL}} \geq 0$.

The MC-EM algorithm is an iterative method that alternates between performing an Expectation (E) step and a Maximization (M) step. In the expectation step, we minimize the KL-divergence D_{KL}, which is equivalent to setting $q(\boldsymbol{\tau}) = p_{\boldsymbol{\theta}}(\boldsymbol{\tau}|E) \propto p(E|\boldsymbol{\tau}) p_{\boldsymbol{\theta}}(\boldsymbol{\tau})$. In the maximization step, we update the policy parameters by maximizing the expected complete data log-likelihood

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_i p(E|\boldsymbol{\tau}^{[i]}) \log p_{\boldsymbol{\theta}}(\boldsymbol{\tau}^{[i]}) \tag{5}$$

where each sample $\boldsymbol{\tau}^{[i]}$ is weighted by the probability of the "reward event", denoted as $p(E|\boldsymbol{\tau})$. The trajectory distribution $p_{\boldsymbol{\theta}}(\boldsymbol{\tau}^{[i]})$ can be replayed by the high-level policy $\pi_{\boldsymbol{\theta}}$. To transform the reward signal $R(\boldsymbol{\tau}^{[i]})$ of a sampled trajectory $\boldsymbol{\tau}^{[i]}$ into a probability distribution of the "reward event", we use the exponential transformation [34], [64], [65]:

$$d^{[i]} = p(E|\boldsymbol{\tau}) = \exp\left\{\beta R(\boldsymbol{\tau}^{[i]})\right\} \tag{6}$$

where the parameter $\beta \in \mathbb{R}_+$ denotes the inverse temperature of the soft-max distribution, higher value of β implies a more greedy policy update.

B. Learning Gaussian Policies

1) Gaussian Policy, Constant Mean: We first focus on solving a simple problem of learning a Gaussian policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ whose mean is a vector of unknown variables. We consider the robot at a fixed state $\mathbf{x}_0$, which does not change during learning. At the beginning of each training iteration, we randomly sample a list of parameters of length N from the current policy distribution $\pi_{\boldsymbol{\theta}}$ and evaluate the parameters via a predefined reward function $R(\boldsymbol{\tau})$, where $\boldsymbol{\tau}^{[i]}$ are the trajectories predicted by solving the MPC given the sampled variables $\boldsymbol{z}^{[i]}$.

In the Expectation step, we transform the computed reward signal $R(\boldsymbol{\tau})$ into a non-negative weight $d^{[i]}$ (improper probability distribution) via the exponential transformation (6). In the Maximization step, we update the policy parameters by optimizing the weighted maximum likelihood objective:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{ \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]}) \right\} \tag{7}$$

where the policy parameters, both the mean and the covariance, are updated using the following closed-form expressions:

$$\begin{aligned} \boldsymbol{\mu} &= \left(\sum_{i=1}^{N} d^{[i]} \mathbf{z}^{[i]} \right) / \left(\sum_{i=1}^{N} d^{[i]} \right) \\ \boldsymbol{\Sigma} &= \left(\sum_{i=1}^{N} d^{[i]} (\mathbf{z}^{[i]} - \boldsymbol{\mu})(\mathbf{z}^{[i]} - \boldsymbol{\mu})^T \right) / Y, \end{aligned} \tag{8}$$

where

$$Y = \left(\left(\sum_{i=1}^{N} d^{[i]} \right)^2 - \sum_{i=1}^{N} (d^{[i]})^2 \right) / \left(\sum_{i=1}^{N} d^{[i]} \right).$$

We repeat this process until the expectation of the sampled reward converges. After training (during policy evaluation), we simply take the mean vector of the Gaussian policy as the optimal decision variables for the MPC. Therefore, $\mathbf{z} = \boldsymbol{\mu}^*$ is the optimal MPC decision variables found by our policy search. A complete episode-based policy search for learning a high-level Gaussian policy for MPC is given in Algorithm 1. A detailed derivation of the above solution is available in the Appendix.

Algorithm 1: Learning Gaussian Policies for MPC

Input: $\pi_{\boldsymbol{\theta}}(\boldsymbol{\mu}, \boldsymbol{\Sigma}), N, \text{MPC}, \mathbf{x}_0, \mathbf{p}$
While not converged
  Sample variables: $\boldsymbol{z}^{[i]} \sim \pi_{\boldsymbol{\theta}}(\boldsymbol{\mu}, \boldsymbol{\Sigma})_{i=1\dots N}$
  Sample trajectories: $\boldsymbol{\tau}^{[i]} = \text{MPC.solve}(\mathbf{x}_0, \boldsymbol{z}^{[i]}, \mathbf{p})$
  Expectation:
    $d^{[i]} = \exp\left\{\beta R(\boldsymbol{\tau}^{[i]})\right\}$
  Maximization:
    $\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{\sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]})\right\}$
Output: Trained Policy $\pi_{\boldsymbol{\theta}^*}(\boldsymbol{\mu}^*, \boldsymbol{\Sigma}^*)$

2) Gaussian Policy, Linear Mean: Algorithm 1 can learn a Gaussian policy that only suits for a single experiment setting or a specific scenario. For generalizing the learned policy to different scenarios, we extend the algorithm by learning a Gaussian linear policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{s}) \sim \mathcal{N}(\mathbf{W}\boldsymbol{\phi}(\mathbf{s}), \boldsymbol{\Sigma})$ whose mean is a linear function approximator $\boldsymbol{\mu} = \mathbf{W}\boldsymbol{\phi}(\mathbf{s})$. We characterize a scenario by the robot's context, denoted by a vector $\mathbf{s}$. The problem of learning $\pi_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{s})$ is called contextual policy search and can be defined by maximizing the expected returns over all different contexts:

$$\max_{\boldsymbol{\theta}} \int_{\boldsymbol{s}} \rho(\boldsymbol{s}) \int_{\boldsymbol{z}} \pi_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{s}) \int_{\boldsymbol{\tau}} p(\boldsymbol{\tau}|\boldsymbol{z}, \boldsymbol{o}) R(\boldsymbol{\tau}, \boldsymbol{s}) d\boldsymbol{\tau} d\boldsymbol{z} d\boldsymbol{s} \quad (9)$$

where $\rho(\boldsymbol{s})$ is the distribution over $\boldsymbol{s}$. The objective (9) can be optimized using the standard MC-EM algorithm, and it results in a different weighted maximum likelihood objective:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{ \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]}|\boldsymbol{s}^{[i]}) \right\}. \quad (10)$$

Maximizing Eq. (10) results in closed-form solutions for the policy parameters:

$$\begin{aligned} \boldsymbol{W} &= \left(\boldsymbol{\Phi}^T \boldsymbol{D} \boldsymbol{\Phi} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{\Phi}^T \boldsymbol{D} \boldsymbol{\Theta} \\ \boldsymbol{\Sigma} &= \frac{\sum_{i=1}^N d^{[i]} \left(\boldsymbol{u}^{[i]} - \boldsymbol{W}^T \boldsymbol{\phi}\left(\boldsymbol{s}^{[i]}\right)\right) \left(\boldsymbol{u}^{[i]} - \boldsymbol{W}^T \boldsymbol{\phi}\left(\boldsymbol{s}^{[i]}\right)\right)^T}{Y}, \end{aligned} \quad (11)$$

where $\boldsymbol{\Phi} = [\phi(\boldsymbol{s})^{[i]}, \cdots, \phi(\boldsymbol{s})^{[N]}]$ is a matrix that contains converted feature vectors for all sampled contexts $\boldsymbol{s}$ and $\boldsymbol{D}$ is the diagonal weighting matrix containing the weights $d^{[i]}$. In the covariance matrix update, Y is the same as in Eq. (8). Here, λ is a small positive variable and $\boldsymbol{I}$ is an identity matrix. The introduce of $\lambda \boldsymbol{I}$ is for numerical stability when calculating the matrix inverse. A complete policy search for learning a Gaussian linear policy for MPC is given in Algorithm 2. A detailed derivation of the above solution is available in the Appendix.

We use the Random Fourier Features (RFF) [62] as the featurizer

$$\boldsymbol{\phi}(\boldsymbol{s})^{[i]} = \sin\left(\frac{\sum_i P_{ij} s^j}{v} + p^{[i]}\right) \quad (12)$$

where each element P_{ij} is randomly sampled from $\mathcal{N}(0, 1)$, v is a bandwidth parameter, and p is a random phase shift drawn from $U[-\pi, \pi)$. The bandwidth v is the only parameter that has to be tuned. The RFF-based linear policy has been used to solve many benchmark continuous control tasks, including the OpenAI gym benchmarks [67].

Algorithm 2: Learning Gaussian Linear Policies for MPC

Input: $\pi_{\boldsymbol{\theta}}(\mathbf{W}\phi(\mathbf{s}), \boldsymbol{\Sigma}), N, \text{MPC}, \mathbf{x}_0, \mathbf{p}$
While not converged
  Sample observations: $\mathbf{s}^{[i]} \sim \rho(\mathbf{s})$
  Sample variables: $\boldsymbol{z}^{[i]} \sim \pi_{\boldsymbol{\theta}}\left(\mathbf{W}\phi(\mathbf{s}^{[i]}), \boldsymbol{\Sigma}\right)_{i=1\dots N}$
  Sample trajectories: $\boldsymbol{\tau}^{[i]} = \text{MPC.solve}(\mathbf{x}_0, \boldsymbol{z}^{[i]}, \mathbf{p})$
  Expectation:
    $d^{[i]} = \exp\left\{\beta R(\boldsymbol{\tau}^{[i]})\right\}$
  Maximization:
    $\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{\sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]}|\mathbf{s}^{[i]})\right\}$
Output: Trained Policy $\pi_{\boldsymbol{\theta}^*}(\mathbf{W}^*\phi(\mathbf{s}), \boldsymbol{\Sigma}^*)$

C. Learning Neural Network Policies

Algorithm 2 can optimize a linear policy using an episodic policy search method. Such episodic policy search by design is used for learning policies in multi-task settings, where distributions over different tasks are well-defined. When controlling a robot in a highly dynamic environment, where the observations differ significantly from state to state, we use a step-based policy search algorithm. Also, we aim to learn a complex neural network policy for selecting adaptive decision variables and for processing relatively high dimensional observations. Such properties are potentially useful for the robot to adapt its behavior online in a highly dynamic environment.

We train the neural network policy by combining Algorithm 1 with supervised learning. A complete algorithm of learning neural network policies for MPC is given in Algorithm 3. We divide the learning process into two stages: 1) data collection, 2) policy learning. In the data collection stage, we randomly initialize the robot in a state $\mathbf{x}_t$ and find the optimal decision variables $\mathbf{z}_t^*$ via Algorithm 1. We aggregate the dataset by $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathbf{o}_t, \mathbf{z}_t^*)$, where $\mathbf{o}_t$ is the current observation of the robot. A sequence of optimal control actions $\mathbf{u}_t^*$ are computed by solving the MPC optimization, given the current state $\mathbf{x}_t$ of the robot and the learned variable $\mathbf{z}_t^*$. Only the first control command is applied to the robot; subsequently, the robot moves to the next state. Incrementally, we collect a set of data that has diverse training pairs $(\mathbf{o}_t, \mathbf{z}_t^*)$ consisting of an observation $\mathbf{o}_t$ as the neural network input and a ground-truth value $\mathbf{z}_t^*$ as the output.

It is important to note that at each simulation time step t, we run Algorithm 1 to solve multiple MPC optimizations in order to find the optimal decision variable for the current state. This step can be viewed as an online learning process in the simulator and is difficult to be run in real-time. During policy learning, we train the neural network by minimizing the mean-squared-error between the labels $\mathbf{z}_t^*$ and the output of the network $f_{\boldsymbol{\Phi}}(\mathbf{o}_t)$, using the standard stochastic gradient descent. After training, the neural network policy is deployed together with the MPC to control the vehicle. Since the resulting controller contains a high-level policy and an MPC, we name this controller High-MPC.

Algorithm 3: Learning Neural Network Policies for MPC

Input: $f_{\boldsymbol{\theta}}, \mathcal{D} = \{\}$
Data collection (repeat)
  Randomly reset the system: $\mathbf{x}_t, \mathbf{o}_t, \mathbf{p}_t, t = 0$
  While not done:
    $(\mathbf{z}_t = \boldsymbol{\mu}^*) \leftarrow$ **Algorithm 1** $(\mathbf{x}_0 = \mathbf{x}_t, \mathbf{p}_t)$
    Data collection: $\mathcal{D} \leftarrow \mathcal{D} \cup \{\mathbf{o}_t, \mathbf{z}_t\}$
    MPC optimization: $\mathbf{u}_t^* = \text{MPC.solve}(\mathbf{x}_t, \mathbf{z}_t, \mathbf{p}_t)$
    System transition: $\mathbf{x}_{t+1} \leftarrow f(\mathbf{x}_t, \mathbf{u}_t^*)$
Policy learning
  $\boldsymbol{\theta}_{\text{new}} = \arg\min_{\boldsymbol{\theta}} \|f_{\boldsymbol{\theta}}(\mathbf{o}_t) - \mathbf{z}_t\|^2$
Output: Learned deep high-level policy $f_{\boldsymbol{\theta}^*}$

V. Flying A Quadrotor through Dynamic Gates

We apply the proposed *policy-search-for-MPC* framework to address a challenging problem towards agile drone flight in dynamic environments, which is learning to fly through dynamic gates. The ability to fly through fast-moving gates enables the drone to traverse inside a dynamic environment, where the free space is changing rapidly. It is a difficult task since it requires simultaneously planning an accurate trajectory that passes through the center of moving gates and controlling the quadrotor to precisely follow the trajectory.

1) Quadrotor Dynamics: We model the quadrotor as a rigid body controlled by four motors. The dynamics of the system can be written as:

$$\dot{\mathbf{p}}_{WB} = \mathbf{v}_{WB} \qquad \dot{\mathbf{q}}_{WB} = \frac{1}{2}\boldsymbol{\Lambda}(\boldsymbol{\omega}_B) \cdot \mathbf{q}_{WB} \tag{13}$$

$$\dot{\mathbf{v}}_{WB} = \mathbf{q}_{WB} \odot \mathbf{c} - \mathbf{g} \qquad \dot{\boldsymbol{\omega}}_{\boldsymbol{B}} = \mathbf{J}^{-1}(\boldsymbol{\eta} - \boldsymbol{\omega}_B \times \mathbf{J}\boldsymbol{\omega}_B) \tag{14}$$

where $\mathbf{p}_{WB}^q = [p_x^q, p_y^q, p_z^q]^T$ and $\mathbf{v}_{WB}^q = [v_x^q, v_y^q, v_z^q]^T$ are the position and the velocity vectors of the quadrotor in the world frame W. We use a unit quaternion $\mathbf{q}_{WB} = [q_w, q_x, q_y, q_z]^T$ to represent the orientation of the quadrotor and use $\boldsymbol{\omega}_B = [\omega_x, \omega_y, \omega_z]^T$ to denote the body rates (roll, pitch, and yaw respectively) in the body frame B. Here, $\mathbf{g} = [0, 0, -g_z]^T$ with $g_z = 9.81 m/s^2$ is the gravity vector, $\mathbf{J}$ is the inertia matrix, $\boldsymbol{\eta}$ is the three dimensional torque, and $\boldsymbol{\Lambda}(\boldsymbol{\omega}_B)$ is a skew-symmetric matrix. Finally, $\mathbf{c} = [0, 0, c]^T$ is the mass-normalized thrust vector. The full state of the quadrotor is defined as $\mathbf{x}^q = [\mathbf{p}^q, \mathbf{q}^q, \mathbf{v}^q]$ (we omit subscript for clarity).

2) Pendulum Dynamics: We model the dynamic gate as a nonlinear pendulum. We approximate the pendulum gate as a point mass that is suspended on a weightless and inextensible string of length L_{cm} from a fixed support whose position is $\boldsymbol{P}_{WP} = [x_f, y_f, z_f]$. The pendulum is subject to three forces: the gravity, the tension force results from the string pulling upon the bob of the pendulum, and a damping force due to friction and air resistance. We approximate the damping force by $f_d = -b * \dot{\theta}$, where $\dot{\theta}$ is the angular velocity and $b \in \mathbb{R}_+$ is a damping factor. We consider the pendulum's motion in the $y - z$ plane, meaning the pendulum rotates about the $x-$axis. Dynamics of the rotational motion is described by the following differential equations

$$\ddot{\theta}_x = -\left(m g_z L_{\text{cm}} \sin\theta_x / I + b\dot{\theta}_x\right), \ddot{\theta}_y = 0, \ddot{\theta}_z = 0 \tag{15}$$

where θ_x is the roll angle, and I is the moment of inertia. Dynamics of the translational motion are given by

$$\ddot{v}_x = 0, \quad \ddot{v}_y = l\cos(\theta_x)\ddot{\theta}_x, \quad \ddot{v}_z = l\sin(\theta_x)\ddot{\theta}_x \tag{16}$$

where l is the distance between the gate center and the fixed point. For the computational convenience, we transfer the Euler angles into a unit quaternion $\mathbf{q}_{WB}^g$ to represent the gate orientation. The full state of the gate center with respect to the inertial frame is represented using the state vector $\mathbf{x}^g = [\mathbf{p}^g, \mathbf{q}^g, \mathbf{v}^g]$.

A. Learning to Fly Through Dynamic Gates

Trajectory Optimization and Cost Function: We formulate the problem of learning to fly through dynamic gates. Our main goal is to find a trajectory that passes through the center of the moving gates. Such a trajectory optimization problem involves 1) decide a sequence of traversal times at which should the dynamic gates be passed, 2) given these traversal times, find a trajectory that passes these gates. Since the gates are moving quickly, the optimization faces a *chicken-and-egg* dilemma, namely, without obtaining the traversal times, it cannot determine the gates' state from which the vehicle should fly through, or without the gates' state, it cannot decide the traversal times.

We first make the assumption that a vector of desired traversal times $\mathbf{t} = [t_1, \cdots, t_i]$ for each gate i is given, where $0 < t_i < t_j$, if $i < j$ and $i, j \in [1, \cdots, n]$. Here, n is the total number of moving gates. Since we know the current states and the dynamic model of the moving gates, we can predict the future trajectory $\boldsymbol{\tau}_i = [\mathbf{x}_0^g, \cdots, \mathbf{x}_{t_i}^g]$ for each gate i, where $\mathbf{x}^g$ is a state vector that consists of position $\mathbf{p}^g$, linear velocity $\mathbf{v}^g$, and orientation $\mathbf{q}^g$. Therefore, we define a gate-pass cost $\mathcal{L}_{\text{gate-pass}}$ as the following quadratic cost

$$\mathcal{L}_{\text{gate-pass}} = \sum_{h=1}^{H-1} (\mathbf{x}_h^q - \mathbf{x}_{t_i}^g)^T \mathbf{Q}_p (\mathbf{x}_h^q - \mathbf{x}_{t_i}^g) \cdot p_h, \tag{17}$$

where $\mathbf{Q}_p$ is a diagonal cost matrix and p_h a Boolean variable defined as

$$p_h = \begin{cases} 1, & \text{iff } h = \lfloor t_i/d_t \rfloor, \\ 0, & \text{otherwise.} \end{cases} \tag{18}$$

Minimizing this loss function encourages the discretized states $\mathbf{x}_h^q$ to stay as closer as possible to the gate states $\mathbf{x}_{t_i}^g$, but only at the given desired traversal time t_i for the gate i. For other discretized states at $h \neq \lfloor t_i/d_t \rfloor$, the loss has no effects since $\mathcal{L}_{\text{gate-pass}} = 0$.

However, such a cost formulation requires very accurate dynamic modeling for both the quadrotor and the moving gates, since $\mathcal{L}_{\text{gate-pass}}$ only characterizes several sparse states that are close to time nodes of the given traversal time vector $\mathbf{t}$. In other words, the optimization treats the moving gates as a list of static waypoints and takes them as soft constraints, without considering their dynamic motions. To counteract

potential model errors and uncertainties, we define a gate-follow cost

$$\mathcal{L}_{\text{gate-follow}} = \sum_{h=1}^{H-1} (\mathbf{x}_h^q - \mathbf{x}_{t_i}^g)^T \mathbf{Q}_f (\mathbf{x}_h^q - \mathbf{x}_{t_i}^g) \cdot w_h \cdot (1 - p_h),$$
$$\text{where,} \quad w_h = \exp\left(-\alpha \cdot (h \cdot d_t - t_i)^2\right) \cdot \gamma_i.$$

Here, $\mathbf{Q}_f$ is a diagonal cost matrix, ω_h defines the exponential weights for following the gate's motion, $\alpha \in \mathbb{R}_+$ defines the temporal spread of the weight, and $\gamma_i \in \mathbb{R}_+$ specifies different weights for tracking different gates. The gate-follow cost provides an intuitive motivation: plan a trajectory that follows the gate if the time difference between the current time $h * d_t$ and the desired traversal time t_i is small; and does not follow the gate if the time difference is significant. In other words, it minimizes the difference between the quadrotor states and the gates' states gradually as the quadrotor approaches the gate.

In addition, we define a terminal cost $\mathcal{L}_{\text{terminal}}$ and an action regularization cost $\mathcal{L}_{\text{u}}$:

$$\mathcal{L}_{\text{terminal}} = (\mathbf{x}_H^q - \mathbf{x}^{\text{goal}})^T \mathbf{Q}_{\text{goal}} (\mathbf{x}_H^q - \mathbf{x}^{\text{goal}}) \tag{19}$$

$$\mathcal{L}_{\text{u}} = \sum_{h=1}^{H-1} (\mathbf{u}_h^q - \mathbf{u}_r)^T \mathbf{Q}_{\text{u}} (\mathbf{u}_h^q - \mathbf{u}_r) \tag{20}$$

where $\mathbf{x}^{\text{goal}}$ is a goal state for hovering and $\mathbf{u}_r = [g_z, 0, 0, 0]$. Here, we use body-rate control $\mathbf{u}_h = [c, \omega_x, \omega_y, \omega_z]$ as the inputs. The terminal cost encourages the quadrotor to fly toward a goal state $\mathbf{x}^{\text{goal}}$.

In summary, we have the following optimization problem

$$\min_{\boldsymbol{\tau}} \quad \mathcal{L}(\mathbf{x}_1, \mathbf{z}, \mathbf{r}) = \mathcal{L}_{\text{terminal}} + \mathcal{L}_{\text{gate-pass}} + \mathcal{L}_{\text{gate-follow}} + \mathcal{L}_{\text{u}}$$
$$\text{s.t.:} \quad \mathbf{u}_{\min} \leq \mathbf{u} \leq \mathbf{u}_{\max}$$
$$\mathbf{x}_{h+1} = \mathbf{x}_h + d_t \cdot \hat{f}(\mathbf{x}_h, \mathbf{u}_h), \quad \mathbf{x}_1 = \mathbf{x}_{\text{init}}$$

where the $\boldsymbol{\tau}$ represents the generated trajectory that consists of the state vector $\mathbf{x}^q = [\mathbf{p}^q, \mathbf{q}^q, \mathbf{v}^q]$ and the control inputs $\mathbf{u}$.

We define a vector of high-level decision variables $\mathbf{z} = [t_1, \gamma_1, \cdots, t_n, \gamma_n]$ that consists of the desired traversal time t_i and the desired weights γ_i for each moving gates. Here, n is the total number of gates. Given the decision variables $\mathbf{z}$, the current state of the quadrotor $\mathbf{x}_0$, the predicted future trajectories $\mathbf{r}$ of all moving gates, we can solve the trajectory optimization problem and find the optimal trajectory for the quadrotor to fly through all moving gates

$$\boldsymbol{\tau} = \text{MPC.solve}(\mathbf{z}, \mathbf{x}_0, \mathbf{p}). \tag{21}$$

Policy Search and Reward Function: The aforementioned optimization formulation relies on the assumption that the desired traversal time t_i for each individual gate is given. In addition, the gate-follow loss $\mathcal{L}_{\text{gate-follow}}$ requires additional variables γ_i to define the weights that are assigned to follow the gate motion in the $y-z$ plane. The high-level decision variables $\mathbf{z}$ are difficult to model or tune due to the dynamic properties of the task. Therefore, a key requirement for our MPC to solve the problem is to have both the desired traversal time and the desired weights in advance. A similar MPC formulation was discussed in [3], where the time variable at which a static waypoint should be reached by a quadrotor is also unknown. They solved the problem by manually selecting the variable via trial-and-error. In our case, the decision variables are much more difficult to select manually.

We solve the problem of finding optimal decision variables in MPC using policy search. We define a Euclidean distance reward function to evaluate the goodness of the trajectory generated by the optimization,

$$R(\boldsymbol{\tau}|\mathbf{z}) = -\sum_{i}^{n} \|\mathbf{p}_{h_i}^q - \mathbf{p}_{h_i}^g\|_2 - \lambda t_i \tag{22}$$

where $h_i = \lfloor t_i / d_t \rfloor$ is the time node at which the predicted quadrotor trajectory intersects with the gate state. This reward is a sparse signal that evaluates the "performance" of the sampled $\mathbf{z}$—high rewards indicate smaller traversal distance error and low rewards indicate large traversal distance error of the predicted trajectory. Hence, maximizing this reward signal leads to desired traversal time variables that allow the optimization to find a trajectory that passes through the center of the gate. Here, λt_i is a regularization term used for choosing smaller time variables.

VI. Experiments

We design our experiments to evaluate the proposed policy-search-for-MPC framework. Specifically, we aim to answer the following questions: 1) can we learn *state-independent* policies for the optimization (Section VI-A), 2) can we learn *state-dependent* linear policies for solving the optimization under different contexts (Section VI-B), 3) can we learn a neural network policy for adapting MPC on the fly (Section VI-C), 4) and finally, what is the performance of our system in the real world (Section VI-D).

A. Learning State-Independent Time Variables for Trajectory Optimization

The first problem is a single trajectory optimization problem, where the goal is to find a safe trajectory that passes through several dynamic gates for a given initial state. We define three dynamic gates that are modeled using the same pendulum dynamics and are initialized at different positions. We use a prediction time horizon of $t_H = 3.0\,\text{s}$ and a discretize time step of $d_t = 0.05\,\text{s}$ for trajectory optimization, it results in a total discretization of $H = 40$ nodes. We use CasADi [68] with IPOP [69] as the solver for the numerical optimization. We learn a vector of decision variables $\mathbf{z}$ using Algorithm 1, where $\mathbf{z}$ is modeled as a high-level policy and is represented using a Gaussian distribution $\mathbf{z} \sim \pi_{\boldsymbol{\theta}}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

Fig. 5 shows the predicted trajectory with the learned parameters. The optimization successfully plans a trajectory that passes through the center of all moving gates at the given learned traversal time. Besides, we achieve small traversal distance errors for all gates. The distance errors for gates $(1, 2, 3)$ are $(0.13\,\text{m}, 0.15\,\text{m}, 0.30\,\text{m})$ respectively. It is important to highlight that the predicted quadrotor position gradually follows the predicted gate's center only when the quadrotor is close to the gate. Such a feature has crucial effects on real-world deployment since the dynamic modeling of the system

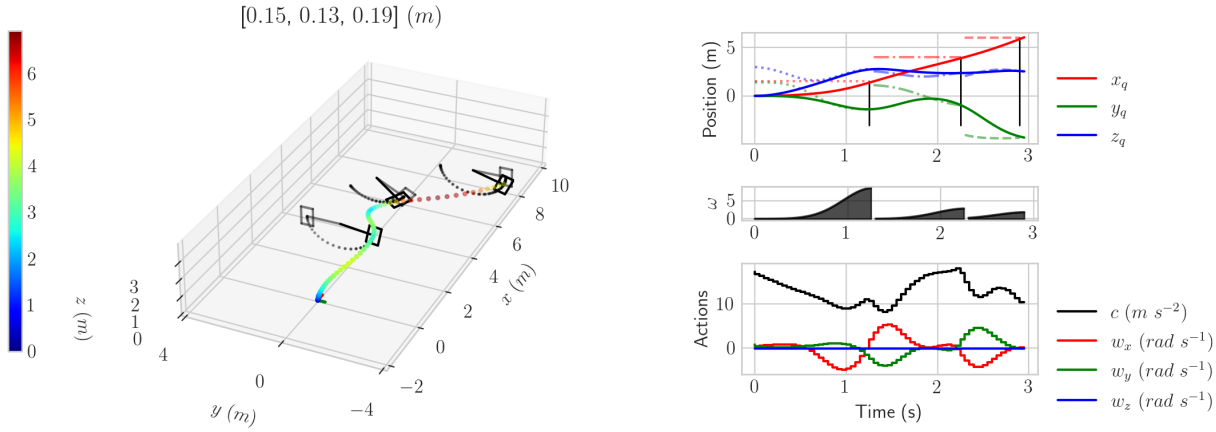

Fig. 5: **Left:** A planned trajectory for flying through 3 dynamic gates. The initial states of the moving gates are indicated by grey color. The quadrotor velocity ($\mathrm{m\,s^{-1}}$) is indicated by the color bar. **Right:** *Top:* The predicted positions of three moving gates (dashed line) and of the quadrotor (solid line). The learned traversal times for three gates (vertical line). *Middle:* Learned weights (ω_h) for the gate-following loss. *Bottom:* Control commands of the quadrotor.

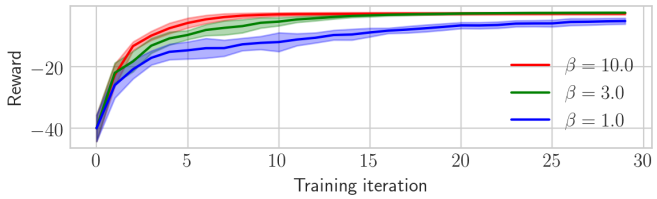

Fig. 6: Learning curves of the Gaussian policy. Each curve is obtained using different temperature parameters β for the policy training. We train six different randomly initialized policies for each temperature parameter. We compute the mean (solid line) and standard deviation (shadow region).

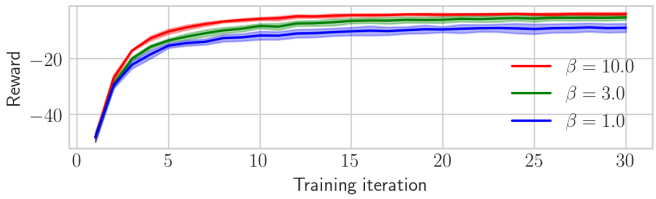

Fig. 7: Learning curves of the Gaussian linear policy. Each curve is obtained using different temperature parameters β for the policy training. We train six different randomly initialized policies for each temperature parameter. We compute the mean (solid line) and standard deviation (shadow region).

is prone to error and the quadrotor has to follow the gate center when it is approaching the gate. Moreover, for the time stages that are far away from the desired traversal time, the pendulum motion has less influence on the quadrotor, leaving more extra control authority to counteract disturbance.

Fig. 6 shows the learning progress of the Gaussian policy, which has randomly initialized weights (both mean and variance). The learning is data-efficient and stable as the policy converges after only a few training iterations, e.g., 10 iterations for $\beta = 10.0$. We train the policy for 30 iterations to make sure that the policy is fully converged. The policy update at each training step requires 30 samples, resulting in 900 MPC optimizations in total. Besides, a high temperature $\beta = 10$ results in a fast (greedy) policy update while a low temperature can have slow convergence.

B. Learning State-Dependent Time Variables for Adaptive Trajectory Optimization Using A Linear Policy

For generalizing the learned high-level policies to different settings or contexts (denoted as a vector $\mathbf{s}$), we train a linear function approximator $\mathbf{z} = f_{\mathbf{W}}(\mathbf{s})$. Specifically, we want to predict the decision variables conditioned on different initialization of the moving gates. We characterize different settings using $\mathbf{s} = [\theta_x^1, \theta_x^2, \theta_x^3]$, where $\theta_x^i, i = 1, 2, 3$ are the initial angles of the gates about the x-axis and are randomly initialized. We represent the policy using a Gaussian linear model $\mathbf{z} \sim \pi_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{W}\phi(\mathbf{s}), \boldsymbol{\Sigma})$ (Section III-C2), where the policy parameters $\boldsymbol{\theta} = [\mathbf{W}, \boldsymbol{\Sigma}]$ are updated using Algorithm 2. We use the RFF featurizer for $\phi(\mathbf{s})$, in which the feature bandwidth is specified as 0.1 and the feature dimension is 40.

Fig. 6 shows the learning progress of the Gaussian linear policy. Similar to training a Gaussian policy, the learning of a Gaussian linear policy is also very data-efficient and stable, thanks to the closed-form solution (Eq. (11)) for updating the policy parameters. Here, the policy update at each training step requires 300 samples, resulting in 9000 MPC optimizations for learning a Gaussian linear policy.

TABLE I shows the evaluation results. The goal is to plan a trajectory for passing through 3 individual moving gates, where the initial states of the gates are randomly selected. Given a planned trajectory, we compute the traversal distance from the quadrotor center to the gate center, at the predicted time instances. We run the experiment repeatedly 100 times and compute the mean and the standard deviation of the

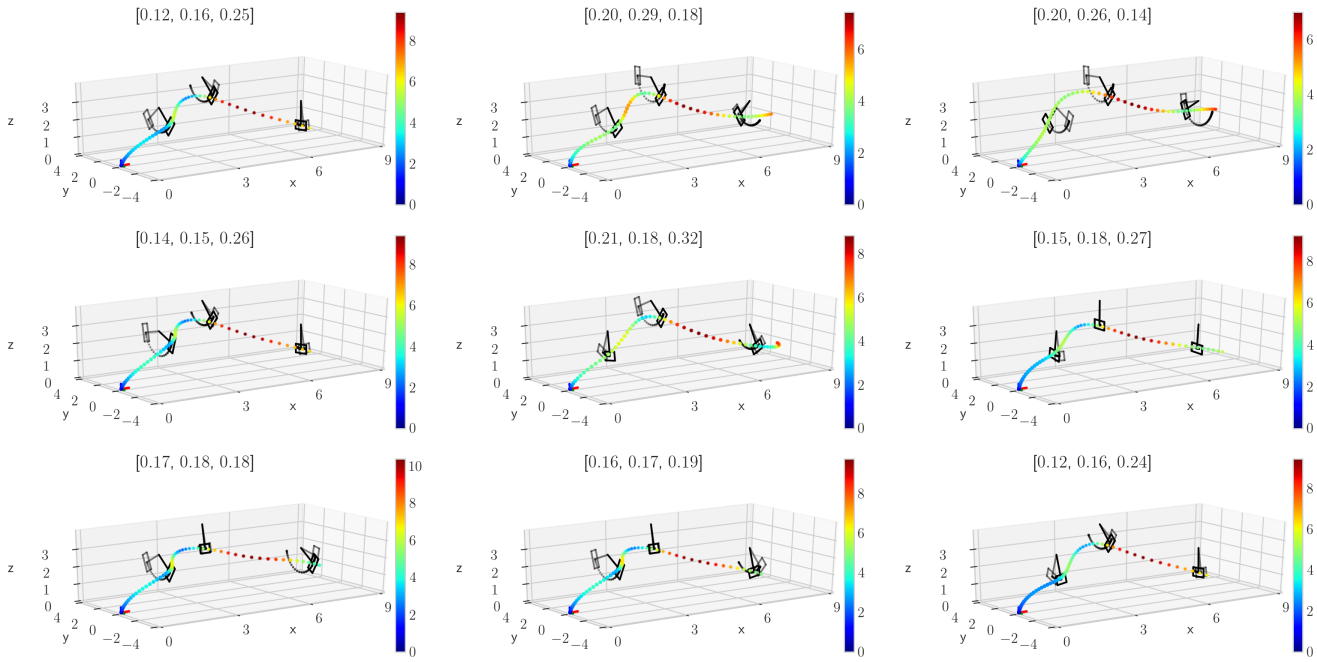

Fig. 8: Evaluation of the trained linear high-level policy with randomly initialized pendulum states. The generated quadrotor trajectories are colored by their speeds that are indicated using the color bars. The traversal distance errors between the quadrotor center and the gate center at the desired traversal times are indicated in the figure title.

TABLE I: Evaluation of the trained Linear High-Level Policy.

Approach	Number of Gates	Success Rates (%)			Mean (m)			Standard Deviation (m)		
		G1	G2	G3	G1	G2	G3	G1	G2	G3
Random	3	0	0	0	2.75	3.46	3.45	0.45	0.25	0.99
Heuristic	3	100	0	0	0.14	1.98	1.14	0.02	0.50	0.29
Ours	**3**	**100**	**100**	**94**	**0.17**	**0.21**	**0.30**	**0.04**	**0.06**	**0.15**

traversal distance errors. In addition, we define a success planning if the traversal distance errors for all gates are lower than $0.5\,\mathrm{m}$. We report the success rates for the 100 trials in TABLE I. Note that G1, G2, and G3 represent each gate separately. Our approach outperforms two simple baselines: one uses randomly sampled time variables, and one uses a heuristic-based selection of the time variables. Fig. 8 shows a visualization of 9 randomly sampled examples.

C. Learning Adaptive Variables for Dynamic Gates via Neural Network Policies

For learning a policy that is useful for the online parameters adaption or compatible with high-dimensional sensory observations, we train a neural network policy. The trained policy can hence be used for adaptively making high-level decisions for the MPC at each control time step, resulting in a closed-loop controller. We use a Multilayer Perceptron (MLP) as the policy representation. In reality, since the environment is only partially observable, we can only observe one gate ahead. We define the observation at the time step t as $\mathbf{o}_t = \mathbf{x}_t^q - \mathbf{x}_t^g$, which represents the difference between the quadrotor's current state $\mathbf{x}_t^q$ and the next gate's state $\mathbf{x}_t^g$. The output of the neural network is the desired traversal time $\mathbf{z} = [t_1]$ for flying through the next gate.

We use Algorithm 3 to train the neural network policy in simulation, in which it combines Algorithm 1 with self-supervised learning. First, we reset the system in random states, meaning we use a randomly initialized position, velocity, and orientation for the quadrotor and a random initial angle for the pendulum gate. We find the optimal traversal time $\mathbf{z}_t^*$ in this state using the policy search algorithm (Algorithm 1). We create a training pair $(\mathbf{o}_t, \mathbf{z}_t^*)$ by associating the current observation with the current optimal variables. Then, we simulate the quadrotor to the next state by solving the MPC optimization using the optimal traversal time $\mathbf{z}_t^*$ and apply the optimal control command to the simulated quadrotor. We also integrate the pendulum dynamics to simulate the gate motion.

We repeat this process at each simulation time step until the quadrotor flies through the gate or it reaches the maximum simulation steps. In total, we collect 40,000 samples, which takes a multi-core CPU several hours to collect the training data. However, the total sampling time can be significantly reduced using parallel processing or multithreading. We implement a fully-connected MLP with two hidden layers of 32 units each and ReLU as the activation function. The training of network weights takes less than 10 minutes on a standard notebook with an Nvidia Quadro P1000 graphics card.

We use an open-source quadrotor simulator called Flightmare [70] to simulate the race track and the quadrotor dynamics. Fig. 9 shows a visualization of the race track in Flightmare. We design a dynamic race track that contains 5 moving gates. All moving gates are modeled using the same

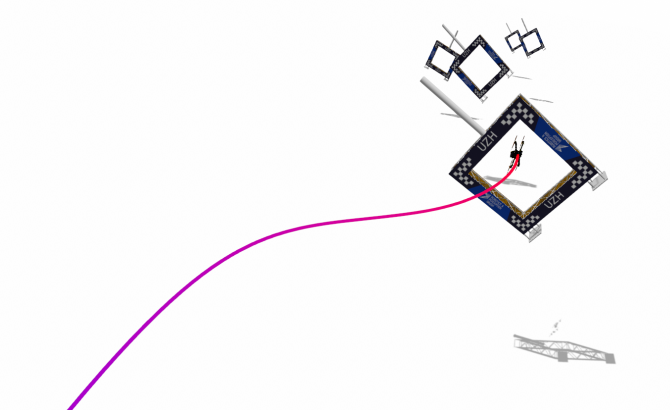

Fig. 9: A visualization of the dynamic racing environment in the Flightmare simulator [70].

pendulum dynamics and rotating around the x-axis.

The gates are attached to different fixed points and are initialized randomly by sampling its initial rotational angle θ_x from a uniform distribution $\theta_x \sim U(-\pi/2, \pi/2)$. Specifically, the gates are separated in the x-axis with a fixed position offset of δ_x, where $\delta_x = p_x^{g,i} - p_x^{g,j} \in [0.5, 1.0, 2.0, 3.0, 4.0, \cdots, 9.0]$ (m) is the difference between two consecutive pendulum gates i and j. Besides, we initialize the quadrotor with different initial velocities of $v_x^q \in [0.0, 1.0, \cdots, 9.0]$ $(\mathrm{m\,s^{-1}})$ in the forward direction.

We run 20 trials for each combination of the position offset and the initial quadrotor velocity (δ_x, v_x^q) and compute the success rate and the averaged traversal distance error. The averaged traversal distance error is computed using the Euclidean distance from the quadrotor center the gate. We compute the success rate by defining a task as a success if all 5 traversal distance errors are less than $0.5\,\mathrm{m}$.

We compare our approach against two baselines: a standard MPC formulation, which does not have access to the desired traversal time, and a fast motion primitive generator [4]. Since the desired time for flying through the next gate is not known beforehand, the desired traversal position is also a prior unknown. We formulate a simple MPC optimization problem as the following

$$\begin{aligned} \min_{\mathbf{u},\mathbf{x}} \mathcal{L} &= \mathcal{L}_{\text{terminal}} + \mathcal{L}_{\text{u}} + \sum_{h=1}^{H-1} (\mathbf{x}_h^q - \mathbf{x}_h^g)^T \mathbf{Q} (\mathbf{x}_h^q - \mathbf{x}_h^g) \\ \text{s.t.:} \quad & \mathbf{u}_{\text{min}} \leq \mathbf{u} \leq \mathbf{u}_{\text{max}} \\ & \mathbf{x}_{h+1}^q = \mathbf{x}_h^q + d_t \cdot \hat{f}(\mathbf{x}_h^q, \mathbf{u}_h), \quad \mathbf{x}_1^q = \mathbf{x}_{\text{init}} \end{aligned}$$

where $\mathbf{x}_h^g$ is the predicted future state of the moving gate, $\mathcal{L}_{\text{terminal}}$ (Eq. 19) is the terminal cost, and $\mathcal{L}_{\text{u}}$ (Eq. 20) is the action cost. Here, $\mathbf{Q} = \text{diag}(0, 100, 100, 10, 10, 10, 10, 0, 10, 10)$ is a time-invariant diagonal cost matrix, which is specified for tracking the gate motion in the y-axis and the z-axis. Solving the above optimization problem results in a trajectory that follows the gate motion in the $y - z$ plane (defined by the stage cost) and reaches an end position located behind the gate (defined by the terminal cost).

As a second baseline, we use a minimum jerk trajectory [4] together with a high-level trajectory sampling scheme. We sample possible traversal trajectories by searching for the desired traversal time. We define a time interval of $T = [0, 3]$ s with a discretization time step of $dt = 0.1\,\mathrm{s}$ for sampling. At each control time step, a total number of 30 trajectories are sampled. The start state of the trajectory is defined by the current quadrotor state. The end state is partially defined, meaning some components are left free. The waypoint (the end position of the traversal trajectory) is calculated using a pendulum model of the moving gate and the sampled time. The end velocities and accelerations are defined as $\mathbf{v} = [\text{None}, 0, 0]$ and $\mathbf{a} = [\text{None}, \text{None}, \text{None}]$, which means that the end velocity v_x in the forward direction (x-axis) and the accelerations are left free. We use zero velocities for the end state at the y-axis and x-axis to reduce the risk of having over-aggressive motion primitives. We reject trajectories that do not satisfy the input constraints and select an optimal trajectory that has less traversal time.

Fig. 10 shows the evaluation results. In summary, our approach achieves the highest success rates and smallest traversal distance errors, particularly when the quadrotor has a small initial velocity and the separation distances among gates are large enough. When the quadrotor is initialized with a high forward velocity and the distance among gates are small, Our approach is prone to failure of the task due to the physical constraint of the platform. For example, the quadrotor does not have enough control authority to break immediately.

The standard MPC has achieved lower success rates and larger traversal distance errors. This is due to the cost function formulation in the MPC optimization, in which the optimization does not have access to the desired traversal time, and thus, has to minimize the distance between the quadrotor and the gate center in all optimization states. Such an optimization scheme results in very aggressive trajectories and large control inputs [33]. Therefore, it is very important to obtain the traversal time prior to the optimization and update the time at each control time adaptively for the closed-loop control. Optimizing the time jointly with other optimization variables is also possible, but might result in a complicated optimization problem that is difficult to solve in real-time.

The sampling-based motion primitives method shows a competitive performance to our method. There are multiple challenges when applying the sampling-based method to our task. First, the motion primitive is defined for state-to-state transition, in which the end velocities are difficult to be specified in advance. For example, a small velocity in the flight direction can result in a slow forward motion while a high velocity can lead to aggressive flight and gate overshooting. Second, the final performance of a task highly depends on the high-level planner, which is used for selecting the optimal motion primitive. Designing such a high-level planner is a challenging task, in which prior research generally uses heuristic-based search [4], [71], [72], which could produce suboptimal solutions.

Fig. 10: Baseline comparison of flying through 5 moving gates using our High-MPC (left), a standard MPC (middle), and a high-frequency motion primitive sampling method [4] (right). Overall, our High-MPC outperforms both baselines in terms of gate traversal distance errors and success rates.

D. Real-world Deployment

To test the robustness and the real-time performance of our system, we deploy our approach in the real world with a physical racing drone. The drone is built from off-the-shelf components used for first-person-view racing (Fig. 11). The drone features a carbon frame with stiff 5-inch propellers, a Lumenier flight controller, an Odroid XU4 single board computer, and a Laird RM024 radio module for receiving control commands. The platform weights $0.775\,\mathrm{kg}$. We design a pendulum gate that comprises a straight steel stick and a wooden loop. The stick has a weight of $0.3\,\mathrm{kg}$ and a length of $1.0\,\mathrm{m}$. The wooden loop has a weight of $0.46\,\mathrm{kg}$ and a radius of $0.45\,\mathrm{m}$. We attach the pendulum gate to a fixed stand. The task goal is to control the quadrotor to fly through the pendulum gate and hover it at a goal position located behind the gate.

All the presented flight experiments were conducted with an OptiTrack motion capture system. We use Extended Kalman Filters (EKF) for estimating both the quadrotor state and the pendulum state using observations from the OptiTrack. The state estimation runs at $200\,\mathrm{Hz}$. We use ACADO [73] for the MPC optimization and qpOASES as a solver in order to achieve real-time control performance of the quadrotor. As discretization step, we chose $d_t = 0.1\,\mathrm{s}$ with a prediction time horizon of $t_T = 2.0\,\mathrm{s}$. The control command solved by the MPC with a desired traversal time variable obtained from the neural network high-level policy are updated at $50\,\mathrm{Hz}$. Our approach can achieve real-time control and is computationally efficient. The computational time for solving an MPC optimization at each control time step is on average less than $5\,\mathrm{ms}$ and for the neural network inference is on average less than $2\,\mathrm{ms}$ (without TensorRT optimization).

We set up the experiment by putting the pendulum gate at a random initial angle and then dropping it. The pendulum gate swings back and forth freely with a periodic motion. The period of the pendulum decreases over time due to friction and air drag, which are difficult to model precisely. We approximate the pendulum dynamics using Eq. (15) & Eq. (16) with a roughly estimated damping factor $b = 0.2$. We use a simple dynamic model for predicting the future trajectory of the gate. We place our drone at a random position in front of the gate. Fig. 12 shows four different trials of the real world experiment conducted in a confined environment. As a result, our approach successfully controls the quadrotor to fly through the fast-moving gate.

Fig. 13 shows a comparison of the executed trajectories between the quadrotor and the moving gate. The vertical black dashed line indicates the traversal moment, at which the quadrotor flies through the gate. The quadrotor follows the gate's motion in both the y-axis (blue) and the z-axis when it approaches to the gate. Besides, the desired traversal time t_{tra} predicted by the neural network together with the corresponding gate-following weighting variable ω is plotted on the right-hand side. Both t_{tra} and ω are used by the MPC for simultaneous planning a trajectory and controlling the vehicle. When t_{tra} is close to zero, the weighting variable increases to a maximum value $\omega \approx 1$, which indicates that the desired traversal time is now and the quadrotor should try to follow the periodic motion in the $y-z$ plane. By contrast, when t_{tra} is very large (e.g., $t_{\text{tra}} = 4.0$), the weighting variable decreases

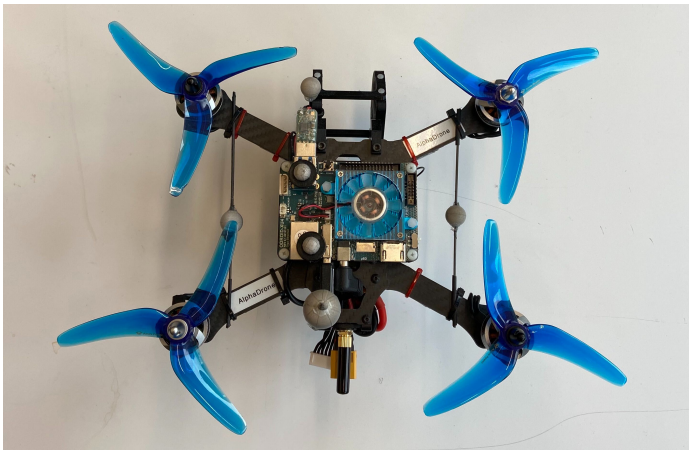

Fig. 11: A racing drone used for the real-world experiment.

to a minimum value $\omega \approx 0$, meaning that there is no need to follow the gate since the gate is either far away or has been passed already. Note that t_{tra} is not the prediction horizon in the MPC optimization.

E. Connections with Prior Trajectory Planning Methods

In general, existing works on quadrotor trajectory planning (or kinodynamic motion planning) in dynamic environments can be divided into sampling-based and optimization-based approaches. Sampling-based algorithms [74], such as RRT* [75], [76], are provably optimal in the limit of infinite samples. To achieve real-time replanning, sampling-based algorithms leverage efficient motion primitive generators to provide a closed-form solution for state-to-state trajectories, e.g., the minimum jerk [4] or minimum snap [2] trajectories. This line of work manifests a significant computational advantage and real-time planning performance, however, relies on simplified dynamics or differentially flat dynamics of a quadrotor. Moreover, they need to relax the single actuator constraints to limit the per-axis acceleration, which might render planned trajectories conservative.

Optimization-based approaches overcome these limitations by enforcing the system dynamics and thrust limits as constraints. They use discrete-time state-space representations for the trajectory, and can handle nonlinear dynamics and different constraints. For passing through dynamic gates, the allocation of the traversal times for the moving waypoints is a priori unknown, rendering the problem formulation complicated. State-of-the-art approaches address similar problems for passing through static waypoints using either heuristic search [3] or formulating a complicated optimization problem using complementary progress constraints [18].

VII. Discussion

A. Choice of Policy Representations

In Section IV, we have presented algorithms for training three different policies, including Gaussian policy, Gaussian linear policy, and neural network policy. Our formulation is general and is designed to be suitable for a wide variety of robotic tasks.

The Gaussian policy finds state-independent decision variables for a single trajectory optimization, namely, the learned decision variables are fixed parameters. In practice, many MPC applications require manually tuning of the hyperparameters, such as the prediction horizon. The Gaussian policy is useful for automatically selecting those hyperparameters.

The Gaussian linear policy learns state-dependent policies via episodic policy search and function approximations. Such policy representations are useful for generalizing robot skills to multiple contexts: a lower-level MPC, parameterized via the parameters Z, controls the robot for a given context s and a high-level policy $\pi(Z|s)$ generalizes among different contexts s. In the context of reinforcement learning, it is generally referred to as contextual policy search [34], [77]. Hence, the Gaussian linear policy has the advantage of efficiently learning linear function representations (with nonlinear kernel features) using a small number of samples.

The neural network policy learns state-dependent policies in a step-based setting, where the policy $\pi(Z_t|o_t)$ needs to adapt its decision variables Z_t based on the observation o_t at each control time step t. The above-mentioned episodic policy search cannot be used directly because the step-based policy search uses different exploration strategies and relies on different policy evaluation methods. Instead, we propose to combine Algorithm 1 with a self-supervised learning scheme for the step-based setting. We show the resulting algorithm can be used for learning a complex neural network policy. The neural network policy is more helpful for adapting decision variables online or potentially for processing high-dimensional observations.

B. Design of the Loss Function and the Reward Function

The loss function design in the MPC optimization involves generating desired control commands for the robot such that the predicted future states match predefined high-level goals. Our approach allows more flexible and automatic loss function design: by taking the MPC as a parameterized controller and using policy search for automatically selecting the desired parameters. The reward function design for policy search has very flexible formulations, such as quadratic, sparse, or exponential.

C. Limitations

First, learning neural networks is computationally expensive and data-hungry. During data collection, the proposed self-supervised policy search (see Algorithm 3) requires running Algorithm 1 in the loop, which is an expensive process since it needs to solve multiple MPC optimizations for each state. Second, Algorithm 3 was designed for learning neural networks for both online decision making and handling high-dimensional observations. Our experiments did not exploit the full potential of using deep neural networks for processing high-dimensional observations, such as images. Finally, our experiment results are based on accurate state estimation with low latency, which is general not the case when using onboard sensing and computing. It is also crucial to consider many factors for real-world scenarios, such as noisy state estimation and system delays.

Fig. 12: Controlling a quadrotor to fly through a dynamic gate using High-MPC under random initializations.

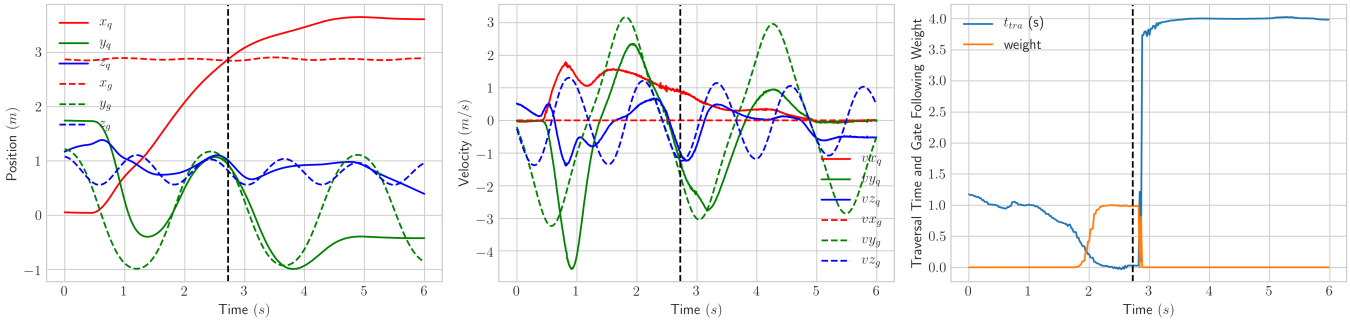

Fig. 13: Trajectories of the quadrotor (solid line) and the pendulum gate (dashed line). The plot on the right-hand side shows the predicted traversal time variable and the weight for minimizing the gate-following cost. The vertical dashed line indicates the traversal time.

VIII. Conclusion

This paper proposed a novel framework that unifies the advantages of both probabilistic policy search and MPC. Our approach improves over the standard MPC formulation by augmenting the MPC controller with learned high-level policies that can automatically choose hard-to-optimize decision variables. Our framework allows a versatile design of different policy representations, ranging from state-independent Gaussian policies to complex neural networks.

As a second contribution, we addressed a challenging problem in agile drone flight: controlling a quadrotor to fly through fast-moving gates. We successfully demonstrated the effectiveness of our approach in both simulation and the real world. To the best of our knowledge, our approach is the first to attempt to address this problem and, hence, can serve as an important baseline for future work.

We release the source code of learning high-level policies for MPC and provide additional theoretical derivations in the Appendix. Future work concerns improving the sample efficiency of the high-level policy training using more advanced policy search methods, such as relative entropy policy search [78]. On the application side, applying the proposed framework to other robotic tasks, such as addressing more complex trajectory optimization problems or using a high-level policy for dealing with model errors, are promising avenues for future research.

References

[1] D. Falanga, K. Kleber, and D. Scaramuzza, "Dynamic obstacle avoidance for quadrotors with event cameras," *Science Robotics*, vol. 5, no. 40, 2020.

[2] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 2520–2525.

[3] M. Neunert, C. De Crousaz, F. Furrer, M. Kamel, F. Farshidian, R. Siegwart, and J. Buchli, "Fast nonlinear model predictive control for unified trajectory optimization and tracking," in *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2016, pp. 1398–1404.

[4] M. W. Mueller, M. Hehn, and R. D'Andrea, "A computationally efficient motion primitive for quadrocopter trajectory generation," *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1294–1310, 2015.

[5] B. Zhou, J. Pan, F. Gao, and S. Shen, "RAPTOR: robust and perception-aware trajectory replanning for quadrotor fast flight," *CoRR*, 2020.

[6] S. Karaman and E. Frazzoli, "High-speed flight in an ergodic forest," in *2012 IEEE International Conference on Robotics and Automation*. IEEE, 2012, pp. 2899–2906.

[7] G. Loianno, C. Brunner, G. McGrath, and V. Kumar, "Estimation, control, and planning for aggressive flight with a small quadrotor with a single camera and imu," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 404–411, 2016.

[8] R. E. Allen and M. Pavone, "A real-time framework for kinodynamic planning in dynamic environments with application to quadrotor obstacle avoidance," *Robotics and Autonomous Systems*, vol. 115, pp. 174–193, 2019.

[9] C. Richter, A. Bry, and N. Roy, "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments," in *Robotics research*. Springer, 2016, pp. 649–666.

[10] B. Landry, R. Deits, P. R. Florence, and R. Tedrake, "Aggressive quadrotor flight through cluttered environments using mixed integer programming," in *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2016, pp. 1469–1475.

[11] P. Foehn, D. Brescianini, E. Kaufmann, T. Cieslewski, M. Gehrig, M. Muglikar, and D. Scaramuzza, "Alphapilot: Autonomous drone racing," *RSS: Robotics, Science, and Systems*, 2020.

[12] A. Loquercio, E. Kaufmann, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Deep drone racing: From simulation to reality with domain randomization," *IEEE Transactions on Robotics*, vol. 36, no. 1, pp. 1–14, 2019.

[13] H. Moon, J. Martinez-Carranza, T. Cieslewski, M. Faessler, D. Falanga, A. Simovic, D. Scaramuzza, S. Li, M. Ozo, C. De Wagter, *et al.*, "Challenges and implemented technologies used in autonomous drone racing," *Intelligent Service Robotics*, vol. 12, no. 2, pp. 137–148, 2019.

[14] J. Rawlings and D. Mayne, *Model Predictive Control: Theory and Design*. Nob Hill Pub., 2009. [Online]. Available: https://books.google.ch/books?id=3_rfQQAACAAJ

[15] D. Falanga, P. Foehn, P. Lu, and D. Scaramuzza, "PAMPC: Perception-aware model predictive control for quadrotors," in *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, 2018.

[16] M. Kamel, T. Stastny, K. Alexis, and R. Siegwart, "Model predictive control for trajectory tracking of unmanned aerial vehicles using robot operating system," in *Robot operating system (ROS)*. Springer, 2017, pp. 3–39.

[17] H. Nguyen, M. Kamel, K. Alexis, and R. Siegwart, "Model predictive control for micro aerial vehicles: A survey," *arXiv preprint arXiv:2011.11104*, 2020.

[18] P. Foehn, A. Romero, and D. Scaramuzza, "Time-optimal planning for quadrotor waypoint flight," *Science Robotics*, vol. 6, no. 56, 2021.

[19] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

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

[21] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning quadrupedal locomotion over challenging terrain," *Science robotics*, vol. 5, no. 47, 2020.

[22] Y. Song, H. Lin, E. Kaufmann, P. Duerr, and D. Scaramuzza, "Autonomous overtaking in gran turismo sport using curriculum reinforcement learning," in *2021 IEEE international conference on robotics and automation (ICRA)*, 2021.

[23] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing with deep reinforcement learning," in *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, 2021.

[24] P. L. Donti, M. Roderick, M. Fazlyab, and J. Z. Kolter, "Enforcing robust control guarantees within neural network policies," *arXiv preprint arXiv:2011.08105*, 2020.

[25] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search," in *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2016, pp. 528–535.

[26] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[27] E. Kaufmann, A. Loquercio, R. Ranftl, M. Müller, V. Koltun, and D. Scaramuzza, "Deep drone acrobatics," *RSS: Robotics, Science, and Systems*, 2020.

[28] G. Williams, N. Wagener, B. Goldfain, P. Drews, J. M. Rehg, B. Boots, and E. A. Theodorou, "Information theoretic mpc for model-based reinforcement learning," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 1714–1721.

[29] J. Kabzan, L. Hewing, A. Liniger, and M. N. Zeilinger, "Learning-based model predictive control for autonomous racing," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3363–3370, 2019.

[30] C. J. Ostafew, A. P. Schoellig, and T. D. Barfoot, "Robust constrained learning-based nmpc enabling reliable mobile robot path tracking," *The International Journal of Robotics Research*, vol. 35, no. 13, pp. 1547–1563, 2016.

[31] U. Rosolia and F. Borrelli, "Learning how to autonomously race a car: a predictive control approach," *IEEE Transactions on Control Systems Technology*, 2019.

[32] G. Williams, A. Aldrich, and E. A. Theodorou, "Model predictive path integral control: From theory to parallel computation," *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 2, pp. 344–357, 2017.

[33] Y. Song and D. Scaramuzza, "Learning high-level policies for model predictive control," in *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, 2020.

[34] M. P. Deisenroth, G. Neumann, and J. Peters, "A survey on policy search for robotics," *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[35] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[36] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[37] S. M. Kakade, "A natural policy gradient," *Advances in neural information processing systems*, vol. 14, pp. 1531–1538, 2001.

[38] J. Peters, K. Muelling, and Y. Altun, "Relative entropy policy search," in *Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence (AAAI), Physically Grounded AI Track*, 2010.

[39] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, 2019.

[40] F. Stulp and O. Sigaud, "Path integral policy improvement with covariance matrix adaptation," in *Proceedings of the 29th International Conference on Machine Learning (ICML)*, Edinbourg, United Kingdom, June 2012, pp. 0–0.

[41] Y. Sun, D. Wierstra, T. Schaul, and J. Schmidhuber, "Efficient natural evolution strategies," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 539–546.

[42] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber, "Policy gradients with parameter-based exploration for control," in *International Conference on Artificial Neural Networks*. Springer, 2008, pp. 387–396.

[43] S. Schaal, "Dynamic movement primitives–a framework for motor control in humans and humanoid robotics," in *Adaptive motion of animals and machines*. Springer, 2006, pp. 261–280.

[44] A. Paraschos, C. Daniel, J. R. Peters, and G. Neumann, "Probabilistic movement primitives," in *Advances in neural information processing systems*, 2013, pp. 2616–2624.

[45] B. Williams, M. Toussaint, and A. J. Storkey, "Modelling motion primitives and their timing in biologically executed movements," in *Advances in neural information processing systems*, 2008, pp. 1609–1616.

[46] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical movement primitives: Learning attractor models for motor behaviors," *Neural Computation*, vol. 25, no. 2, pp. 328–373, 2013.

[47] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural networks*, vol. 21, no. 4, pp. 682–697, 2008.

[48] J. Kober and J. R. Peters, "Policy search for motor primitives in robotics," in *Advances in neural information processing systems*, 2009, pp. 849–856.

[49] J. Kober, E. Oztop, and J. Peters, "Reinforcement learning to adjust robot movements to new situations," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[50] S. Levine and V. Koltun, "Guided policy search," in *Proceedings of the 30th International Conference on Machine Learning*, 2013, pp. 1–9.

[51] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in *Advances in Neural Information Processing Systems*, vol. 27, 2014, pp. 1071–1079.

[52] S. Levine and V. Koltun, "Learning complex neural network policies with trajectory optimization," in *International Conference on Machine Learning*, 2014, pp. 829–837.

[53] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Information-theoretic model predictive control: Theory and applications to autonomous driving," *IEEE Transactions on Robotics*, vol. 34, no. 6, pp. 1603–1622, 2018.

[54] B. Amos, I. Jimenez, J. Sacks, B. Boots, and J. Z. Kolter, "Differentiable mpc for end-to-end planning and control," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 8289–8300.

[55] E. Kaufmann, M. Gehrig, P. Foehn, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Beauty and the beast: Optimal methods meet learning for drone racing," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 690–696.

[56] P. Drews, G. Williams, B. Goldfain, E. A. Theodorou, and J. M. Rehg, "Aggressive deep driving: Combining convolutional neural networks and model predictive control," in *Conference on Robot Learning*, 2017, pp. 133–142.

[57] P. Drews, G. Williams, B. Goldfain, E. A. Theodorou, and J. M. Rehg, "Vision-based high-speed driving with a deep dynamic observer," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1564–1571, 2019.

[58] K. Chatzilygeroudis, V. Vassiliades, F. Stulp, S. Calinon, and J.-B. Mouret, "A survey on policy search algorithms for learning robot controllers in a handful of trials," *IEEE Transactions on Robotics*, vol. 36, no. 2, pp. 328–347, 2019.

[59] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, vol. 3. IEEE, 2004, pp. 2619–2624.

[60] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, pp. 2219–2225.

[61] C. Daniel, G. Neumann, O. Kroemer, and J. Peters, "Hierarchical relative entropy policy search," *Journal of Machine Learning Research*, vol. 17, no. 93, pp. 1–50, 2016.

[62] A. Rajeswaran, K. Lowrey, E. V. Todorov, and S. M. Kakade, "Towards generalization and simplicity in continuous control," in *Advances in Neural Information Processing Systems*, 2017, pp. 6550–6561.

[63] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[64] G. Neumann *et al.*, "Variational inference for policy search in changing situations," in *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, 2011, pp. 817–824.

[65] M. Toussaint, "Robot trajectory optimization using approximate inference," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1049–1056.

[66] N. Vlassis and M. Toussaint, "Model-free reinforcement learning as mixture learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009, pp. 1081–1088.

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

[68] J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "Casadi: a software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019.

[69] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming*, vol. 106, no. 1, pp. 25–57, 2006.

[70] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Conference on Robot Learning*, 2020.

[71] B. Zhou, F. Gao, L. Wang, C. Liu, and S. Shen, "Robust and efficient quadrotor trajectory generation for fast autonomous flight," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3529–3536, 2019.

[72] S. Liu, K. Mohta, N. Atanasov, and V. Kumar, "Search-based motion planning for aggressive flight in se (3)," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2439–2446, 2018.

[73] B. Houska, H. Ferreau, and M. Diehl, "ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.

[74] M. Elbanhawi and M. Simic, "Sampling-based robot motion planning: A review," *Ieee access*, 2014.

[75] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[76] D. J. Webb and J. Van Den Berg, "Kinodynamic rrt*: Asymptotically optimal motion planning for robots with linear dynamics," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 5054–5061.

[77] A. G. Kupcsik, M. P. Deisenroth, J. Peters, and G. Neumann, "Data-efficient generalization of robot skills with contextual policy search," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[78] J. Peters, K. Mulling, and Y. Altun, "Relative entropy policy search," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

ACKNOWLEDGMENT

We thank Thomas Längle, Roberto Tazzari, Manuel Sutter, Elia Kaufmann, Antonio Loquercio, Philipp Foehn, Angel Romero, and Sihao Sun for their help or the valuable discussions.

APPENDIX

We provide detailed derivations for both updating the Gaussian policy and the Gaussian linear policy using weighted maximum likelihood.

A. Derivation of Algorithm 1

The objective of maximizing a Gaussian policy in Algorithm 1 is defined as

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{ \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]}) \right\}$$

where the log-likelihood of the Gaussian policy is given by

$$\begin{aligned}
&\log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{\theta}) \\
&\quad = \log \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \\
&\quad = \log \frac{\exp\left(-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{z}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} \\
&\quad = -\frac{k}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}(\boldsymbol{z}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{z}-\boldsymbol{\mu})
\end{aligned}$$

In order to find the $\boldsymbol{\theta}$ that maximizes the reward, we take the derivative to the policy parameters $\boldsymbol{\theta} = [\boldsymbol{\mu}, \boldsymbol{\Sigma}]$, separately, and set the gradients to zero. We first compute the solution for updating the mean $\boldsymbol{\mu}$

$$\begin{aligned}
&\nabla_{\boldsymbol{\mu}} \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]}) \\
&\quad = \sum_i d^{[i]} \nabla_{\boldsymbol{\mu}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]}) \\
&\quad = \sum_i d^{[i]} \nabla_{\boldsymbol{\mu}} \Big(-\frac{k}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}| \\
&\quad - \frac{1}{2}(\boldsymbol{z}^{[i]}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{z}^{[i]}-\boldsymbol{\mu}) \Big) \\
&\quad = \sum_i d^{[i]} \left(-\nabla_{\boldsymbol{\mu}} \frac{1}{2}(\boldsymbol{z}^{[i]}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{z}^{[i]}-\boldsymbol{\mu}) \right) \\
&\quad = + \sum_i d^{[i]} (\boldsymbol{z}^{[i]}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} = \mathbf{0}.
\end{aligned}$$

By solving for $\boldsymbol{\mu}$, we obtain

$$\boldsymbol{\mu}_{\text{new}} = \frac{\sum_{i=1}^{N} d^{[i]} \boldsymbol{z}^{[i]}}{\sum_{i=1}^{N} d^{[i]}}$$

Second, we compute the solution for the covariance matrix $\boldsymbol{\Sigma}$,

$$\begin{aligned}
&\nabla_{\boldsymbol{\Sigma}} \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]}) \\
&= \sum_i d^{[i]} \nabla_{\boldsymbol{\Sigma}} \Big(-\frac{k}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}| \\
&\quad - \frac{1}{2}(\boldsymbol{z}^{[i]} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{z}^{[i]} - \boldsymbol{\mu}) \Big) \\
&= \sum_i d^{[i]} (-\frac{1}{2}\boldsymbol{\Sigma}^{-1} + \frac{1}{2}\boldsymbol{\Sigma}^{-1}(\boldsymbol{z}^{[i]} - \boldsymbol{\mu})(\boldsymbol{z}^{[i]} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}) \\
&= -\frac{1}{2}\boldsymbol{\Sigma}^{-1} \sum_i d^{[i]} + \\
&\quad \frac{1}{2}\boldsymbol{\Sigma}^{-1} \left(\sum_i d^{[i]} (\boldsymbol{z}^{[i]} - \boldsymbol{\mu})(\boldsymbol{z}^{[i]} - \boldsymbol{\mu})^T \right) \boldsymbol{\Sigma}^{-1} \\
&= \mathbf{0}.
\end{aligned}$$

By solving for $\boldsymbol{\Sigma}$, we obtain

$$\boldsymbol{\Sigma}_{\text{new}} = \frac{\sum_{i=1}^{N} d^{[i]} (\mathbf{z}^{[i]} - \boldsymbol{\mu})(\mathbf{z}^{[i]} - \boldsymbol{\mu})^T}{\sum_i d^{[i]}}.$$

Note that in Eq. (8), we use

$$Y = \frac{\left(\sum_{i=1}^{N} d^{[i]}\right)^2 - \sum_{i=1}^{N} (d^{[i]})^2}{\sum_{i=1}^{N} d^{[i]}}$$

as the demonimator to obtain an unbiased estimate of the covariance.

B. Derivation of Algorithm 2

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \left\{ \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]} | \mathbf{s}^{[i]}) \right\}.$$

where the log-likelihood of the Gaussian policy is given by

$$\begin{aligned}
&\log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}|\mathbf{s};\boldsymbol{\theta}) \\
&= \log \mathcal{N}(\boldsymbol{z}|\mathbf{W}\boldsymbol{\phi}(\mathbf{s}), \boldsymbol{\Sigma}) \\
&= \log \frac{\exp\left(-\frac{1}{2}(\boldsymbol{z} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{z} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))\right)}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} \\
&= -\frac{k}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}| \\
&\quad - \frac{1}{2}(\boldsymbol{z} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{z} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))
\end{aligned}$$

Similar to the previous derivation, in order to find the $\boldsymbol{\theta}$ maximizing the reward, we take the derivative to the policy parameters $\boldsymbol{\theta} = [\boldsymbol{W}, \boldsymbol{\Sigma}]$, separately, and set the gradients to zero. We first compute the solution for the parameters $\boldsymbol{W}$ that are used for approximating the mean,

$$\begin{aligned}
&\nabla_{\mathbf{W}} \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]} | \mathbf{s}^{[i]}) \\
&= \frac{1}{2} \sum_i d^{[i]} \nabla_{\mathbf{W}} (\boldsymbol{z}^{[i]} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{z}^{[i]} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s})) \\
&= \boldsymbol{\Sigma}^{-1} \sum_i d^{[i]} (\boldsymbol{z}^{[i]} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))\boldsymbol{\phi}(\mathbf{s})^T = \mathbf{0} \\
\Rightarrow &\sum_i d^{[i]} \boldsymbol{z}^{[i]} \boldsymbol{\phi}(\mathbf{s})^T = \sum_i d^{[i]} \mathbf{W}\boldsymbol{\phi}(\mathbf{s})\boldsymbol{\phi}(\mathbf{s})^T.
\end{aligned}$$

We can rewrite above equation in a matrix form

$$\begin{aligned}
\mathbf{W}^T \boldsymbol{\Phi}^T \mathbf{D} \boldsymbol{\Phi} &= \mathbf{Z}^T \mathbf{D} \boldsymbol{\Phi} \\
\Rightarrow \boldsymbol{W} &= \left(\boldsymbol{\Phi}^T \boldsymbol{D} \boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^T \boldsymbol{D} \boldsymbol{Z}
\end{aligned}$$

where $\boldsymbol{\Phi} = [\phi(\boldsymbol{s}^{[i]}), \cdots, \phi(\boldsymbol{s}^{[N]})]$ is a matrix that contains converted feature vectors for all sampled observations $\boldsymbol{s}$ and $\boldsymbol{D}$ is the diagonal weighting matrix containing the weights $d^{[i]}$. Here, $\boldsymbol{Z}$ contains the sampled parameters $[\boldsymbol{z}^i, \cdots, \boldsymbol{z}^N]$. Note that, in Eq. (11), the introduce of $\lambda\mathbf{I}$ is for numerical stability.

Second, we compute the solution for the covariance matrix $\boldsymbol{\Sigma}$,

$$\begin{aligned}
&\nabla_{\boldsymbol{\Sigma}} \sum_i d^{[i]} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{z}^{[i]} | \mathbf{s}^{[i]}) \\
&= -\frac{1}{2} \sum_i d^{[i]} \nabla_{\boldsymbol{\Sigma}} (\log|\boldsymbol{\Sigma}| \\
&+ (\boldsymbol{z}^{[i]} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{z}^{[i]} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))) \\
&= -\frac{1}{2}\boldsymbol{\Sigma}^{-1} \sum_i d^{[i]} \\
&+ \frac{1}{2}\boldsymbol{\Sigma}^{-1} \left(\sum_i d^{[i]} (\boldsymbol{z}^{[i]} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))(\boldsymbol{z}^{[i]} - \mathbf{W}\boldsymbol{\phi}(\mathbf{s}))^T \right) \boldsymbol{\Sigma}^{-1} \\
&= \mathbf{0}.
\end{aligned}$$

By solving for $\boldsymbol{\Sigma}$, we obtain

$$\boldsymbol{\Sigma} = \frac{\sum_{i=1}^{N} d^{[i]} \left(\boldsymbol{u}^{[i]} - \boldsymbol{W}^T \boldsymbol{\phi}\left(\boldsymbol{s}^{[i]}\right)\right) \left(\boldsymbol{u}^{[i]} - \boldsymbol{W}^T \boldsymbol{\phi}\left(\boldsymbol{s}^{[i]}\right)\right)^T}{\sum_i d^{[i]}}$$

Similar to Algorithm 1, we use

$$Y = \frac{\left(\sum_{i=1}^{N} d^{[i]}\right)^2 - \sum_{i=1}^{N} (d^{[i]})^2}{\sum_{i=1}^{N} d^{[i]}}$$

as the demonimator to obtain an unbiased estimate of the covariance.