Automatic Tuning for Data-driven Model Predictive Control

William Edwards[1], Gao Tang[1], Giorgos Mamakoukas[2], Todd Murphey[2], and Kris Hauser[1]

Abstract— **Model predictive control (MPC) is a powerful feedback technique that is often used in data-driven robotics. The performance of data-driven MPC depends on the accuracy of the model, which often requires careful tuning. Furthermore, specifying the task with an objective function and synthesizing a feedback policy are not straightforward and typically lead to suboptimal solutions driven by trial and error. To address these challenges, we present a method to jointly optimize the data-driven system identification, task specification, and control synthesis of unknown dynamical systems. We use our method to develop `AutoMPC`[3], a software package designed to automate and optimize data-driven MPC. Empirical evaluation on the pendulum swing-up, cart-pole swing-up, and half-cheetah running demonstrates that our method finds data-driven control policies that outperform offline reinforcement learning, without any hand-tuning.**

I. Introduction

Model predictive control (MPC) is a powerful framework for designing robot controllers. By leveraging knowledge of the dynamics, it can predict and optimize a robot's behavior over a multi-step time horizon and has been demonstrated to be effective on high-dimensional robots [32]. Moreover, MPC has been used as a component of model-based reinforcement learning (RL) solvers for continuous control problems to improve the sample complexity [9].

Successfully implementing MPC is challenging, as the control performance relies heavily on the accuracy of the model. For soft robots or robots with complex dynamics, i.e., aerodynamic or hydrodynamic interactions, developing a representation from first-principles can often be laborious or even intractable. Alternatively, researchers are increasingly relying on data-driven models using—among others— neural networks [25], Gaussian processes [24], or Koopman operators [1]. On the other hand, system identification (SysID) methods typically suffer from tedious hyperparameter tuning, scalability issues, or limited model capacity [26]. Further, when done manually, hyperparameter tuning is time-consuming and prone to errors.

Besides model accuracy, the control performance of MPC is also sensitive to factors such as the objective function, including regularization terms, the planning horizon, and state or control constraints. These hyperparameters create a large search space that is often left largely unexploited leading to suboptimal solutions. The optimizer must also be carefully chosen to exploit any nonlinearities in the dynamics. This is especially true for nonlinear objectives (e.g., "sparse rewards" in the RL community) and underactuated nonlinear systems.

This paper tackles the challenges of MPC by automating the process of joint SysID, task specification, and control synthesis as an end-to-end, data-driven hyperparameter optimization problem. For each hyperparameter setting, the system learns a dynamics model from a training set, instantiates an MPC controller, and evaluates closed-loop performance. In addition it is often impractical to test controllers on physical robots due to safety concerns and physical wear and tear. We propose a cross-validation-like performance measure that tests the controller on a simulation model learned through SysID on a holdout set. This approach measures the robustness of the controller to imperfect modeling and is shown to correlate well with true performance.

We implement these methods in a software package, `AutoMPC`, that aims to make MPC accessible to non-experts in the same way that AutoML libraries such as `auto-sklearn` [13] and `AutoKeras` [19] have done for supervised machine learning. The package implements a wide variety of SysID models (autoregression, Gaussian Processes (GP), Koopman operators, SINDy, neural networks) and optimizers (LQR, iLQR, direct transcription, and Model Predictive Path Integral (MPPI) control), and presents an open framework for contributors to add their own algorithms. Experiments on nonlinear control benchmarks demonstrate that `AutoMPC` can tune MPC pipelines to achieve superior performance to a state-of-the-art offline reinforcement learning algorithm. Although initial hyperparameter settings yield a learned dynamics model and optimizer that perform badly on the true dynamics, some or all MPC controllers generated by `AutoMPC` perform well within a couple of hours of tuning. This suggests that `AutoMPC` can be a useful tool for generating baseline controllers without requiring significant human intervention or expertise.

II. Related Work

Model Predictive Control: MPC is a widely used closed-loop control approach in robotics that incorporates state and control constraints [5], [25], [31]. For linear systems, the theory of closed-loop stability has been established and optimization is carried out efficiently with convex optimization solvers or avoided by offline precomputation known as explicit MPC [5]. However, for general nonlinear systems the closed-loop behavior is not yet well understood and successfully implementing MPC requires manual tuning.

Besides the high computational demand, MPC relies on accurately modeling the system dynamics. Although it has has been shown to achieve the desired performance when

*W. Edwards is partially supported by NIH Grant # R21-EY029877.

[1]W. Edwards, G. Tang, and K. Hauser are with the Department of Computer Science, University of Illinois at Urbana-Champaign. {wre2,gaotang2,kkhauser}@illinois.edu

[2]G. Mamakoukas and T. Murphey are with the Department of Mechanical Engineering, Northwestern University.{giorgosmamakoukas@u.northwestern.edu, t-murphey@northwestern.edu}

[3]The code and documentation for `AutoMPC` are available at http://github.com/williamedwards/autompc

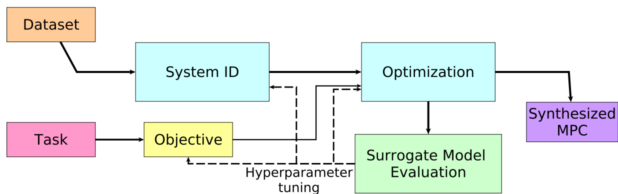

Fig. 1: The structure of the `AutoMPC` pipeline.

using models learned from data [9], inaccurate data-driven representations can cause suboptimal performance and even unstable control. Although robust MPC can address model inaccuracy [3] for linear systems, robust nonlinear control is harder to analyze and achieve. Our method addresses this issue by automatically tuning the hyperparameters with Bayesian optimization where the performance of the SysID, task specification, and control synthesis is tuned end-to-end on a simulated surrogate model.

System Identification and Data-driven Control: System identification has been thoroughly explored [26] and theory of data-driven control for linear systems is relatively well developed, with regret bounds derived for policy gradient methods [12]. However, nonlinear SysID and data-driven control remains an open research question. Recently, there has been significant interest in using deep neural networks [20], [25], [34], GP based methods [24] and Koopman operator theory [1], [27]. Both classical system identification techniques and deep learning-based techniques require careful hyperparameter tuning to obtain useful models.

Our approach is also related to model-based RL approaches [11], [30], [35], which explicitly learn a dynamics model and rewards, and run an optimizer to generate the agent's policy. However, to generate experience model-based RL accesses the true system, which is often infeasible or even unsafe. Our setting is also related to offline RL, which does not use online interaction with the robot [15], [21], [22]. `AutoMPC` is both offline and model-based, and also autotuned.

AutoML and Controller Tuning: Influential AutoML packages include `auto-sklearn` [13] and `AutoKeras`, but these apply to the supervised learning setting. In the control setting, a few authors have studied tuning MPC. Marco et al. [28] tune a cost function to compensate for errors in the dynamics model. Bayesian optimization has been used by Bansal et al. [2] to tune a parametric linear model to improve closed-loop performance, and by Piga et al. [29] to optimize the control horizon. Forgione et al [14] consider end-to-end MPC tuning, but they a) tune model parameters of the learned model rather than hyperparameters, which is susceptible to overfitting, and b) need access to the true dynamics of the system. `AutoMPC`, on the other hand, optimizes hyperparameters and does not require knowledge of the system, but instead evaluates performance via a cross-validation-like technique using surrogate dynamics.

III. Methods

We automate MPC using the framework illustrated in Fig. 1. As input, the user provides a dataset $\mathcal{D}$ of state/control trajectories, a performance metric J, a set of initial states $\mathbf{I}$, and optional state and control constraints ϕ. A complete MPC controller consists of a SysID model, learned from a training subset, and an optimizer. Each component has number of tunable *hyperparameters* (Sec. III-B), with a hyperparameter setting denoted a *configuration*. The controller defined by a configuration is optimized end-to-end using Bayesian optimization (Sec. III-C).

At the start of tuning, $\mathcal{D}$ is randomly partitioned a SysID training set $\mathcal{D}_I \subset \mathcal{D}$ and a holdout dataset $\mathcal{D}_H = \mathcal{D} \setminus \mathcal{D}_I$, which is used to train a *surrogate dynamics model*. We then evaluate configurations in an order selected by Bayesian optimization. To evaluate a configuration, we synthesize the corresponding MPC by learning a dynamics model from $\mathcal{D}_I$, initializing the optimizer, and selecting the objective function based on the configuration hyperparameters. Control actions are then selected by the optimizer using the dynamics model and objective function. We then evaluate the closed-loop performance of the MPC from the initial states $\mathbf{I}$ using the surrogate dynamics model. After a fixed number of iterations, the best performing controller is provided as output. Details are given below.

A. Problem Definition

We assume a fully-observable discrete-time dynamical system with state $x_t \in \mathbb{R}^n$ and control $u_t \in \mathbb{R}^m$. We use the notation $\mathbf{x}_{t:r}$ and $\mathbf{u}_{p:q}$ to denote the sequences $(x_t, x_{t+1}, \ldots x_r)$ and $(u_p, u_{p+1}, \ldots, u_q)$ respectively. The dynamics of the system are written as

$$x_{t+1} = f(x_t, u_t).$$

At each time step, MPC optimizes a trajectory over a fixed horizon H with respect to some objective function L and constraints ϕ. This optimization has the form

$$\min_{\mathbf{x}_{t:t+H}, \mathbf{u}_{t:t+H-1}} L(\mathbf{x}_{t:t+H}, \mathbf{u}_{t:t+H-1})$$
$$\text{s.t. } x_{i+1} = f(\mathbf{x}_{0:i}, \mathbf{u}_{0:i}) \text{ and } \phi(\mathbf{x}_{t:t+H}, \mathbf{u}_{t:t+H-1}) \leq 0.$$

We do not have access to the true system dynamics, so we must estimate an approximate model $\hat{f}$ identified from $\mathcal{D}$.

The user designates a performance metric $J(\mathbf{x}_{1:T}, \mathbf{u}_{1:T})$ which scores rolled-out trajectories. We allow J to have arbitrary form. The terminal time T may be constant or dynamically determined by some goal condition, also specified.

Note that we distinguish the objective L used in the optimizer from the performance metric J, for several reasons. First, certain optimization methods require L to have a particular structure. For example, LQR requires L to be quadratic while iLQR requires L to be twice differentiable. Second, J is evaluated over the entire trajectory while L is only optimized over a fixed horizon H, and optimizing repeatedly over a fixed horizon may not lead to good overall performance. This is particularly true for metrics that include terminal cost, since terminal cost does not provide useful guidance to the MPC until near the end of the trajectory. Third, certain optimizers cannot accept state or control constraints, so the objective function may encode constraints as barriers in the objective function. Finally, L may include

regularization terms that penalize deviation from the training data.

Note that learning $\hat{f}$ from $\mathcal{D}$ can be viewed as supervised learning, but the MPC context adds additional considerations. For example, a simpler, less accurate model may be preferred to a more complex, more accurate one if the former is cheaper to evaluate, suffers from fewer local minima, or is less prone to overfitting. Moreover, the generalization performance of the model outside of the data distribution is critical, as the MPC may guide the system away from the distribution in order to "exploit" the model inaccuracies and produce unrealistic trajectories. This is known as *distribution shift* and can be partially alleviated by tuning L to penalize trajectories which deviate from the data distribution.

B. Hyperparameters

Hyperparameters can take on a mixture of continuous, integer, and categorical values. We also allow for conditional relationships between hyperparameters. For example, a continuous hyperparameter specifying the weight of a particular term in the objective function can be conditioned on a boolean hyperparameter which turns the term on or off.

First, we let $\mathcal{H}_S$ denote the set of hyperparameters for the SysID method. For example, the linear autoregression model uses an integer k to control the size of the state history. Next, we let $\mathcal{H}_L$ denote the set of objective function hyperparameters. For example, a quadratic cost function might rescale the diagonal values of the cost matrices. Finally, we let $\mathcal{H}_O$ denote the set of optimizer hyperparameters. This includes the planning horizon H as well as any other optimizer-specific settings, e.g., MPPI sets the number of trajectories that are sampled in each iteration. Considering these components together, we have a joint hyperparameter space $\mathcal{H} = \mathcal{H}_S \times \mathcal{H}_L \times \mathcal{H}_O$ for the end-to-end system.

C. Tuning with Surrogate Functions

`AutoMPC` implements tuning in three modes: 1) End-to-end tuning, which tunes the entire pipeline for closed-loop performance, 2) SysID tuning, which only tunes the dynamics model for accuracy, 3) Decoupled tuning, which first performs SysID tuning and then tunes the optimizer for closed-loop performance.

To search the hyperparameter space $\mathcal{H}$, we use the Bayesian optimization algorithm Sequential Model-based Algorithm Configuration (SMAC) [18], as implemented in the Python package `smac3`. SMAC builds a model using a random forest to predict the performance of a configuration h before it is evaluated. This model is used to select the next configuration to evaluate. The use of a random forest, in contrast to the Gaussian Process models used by many other Bayesian optimization algorithms, allows SMAC to handle structured hyperparameter spaces that contain a mixture of discrete and continuous hyperparameters, and conditional hyperparameter relationships. The details of the three tuning modes are as follows:

a) End-to-End tuning: Given a configuration $h \in \mathcal{H}$, we learn $\hat{f}$ from $\mathcal{D}_I$ and derive an MPC controller $\pi_{h,\hat{f}}$. We define the controller's *true performance* as

$$\mathbb{J}[h] = \sum_{s^i \in \mathbf{I}} J(\mathbf{x}^i_{1:T}, \mathbf{u}^i_{1:T-1}) \tag{1}$$

where $\mathbf{I}$ is the set of initial states, $x^i_1 = s_i$, and each trajectory is generated by a closed-loop rollout of $\pi_{h,\hat{f}}$ to the true dynamics f.

Without access to f, we define a *surrogate performance* $\hat{\mathbb{J}}$ that is identical to (1) except the rollout is performed with respect to a learned surrogate dynamics model $\hat{f}_{surr}$, which is learned on the holdout set $\mathcal{D}_H$. We avoiding sharing data between system ID and surrogate training sets to ensure that the surrogate is not identical to model $\hat{f}$ used for control. We use Bayesian optimization to minimize $\hat{\mathbb{J}}$ over $\mathcal{H}$. Although $\hat{\mathbb{J}}$ is an imperfect approximation of $\mathbb{J}$, our experiments show that it is still useful for tuning. Specifically, for two controllers π_1, π_2 with a non-negligible difference in $\mathbb{J}(\pi_1)$ and $\mathbb{J}(\pi_2)$, it almost always holds that $\hat{\mathbb{J}}(\pi_1) < \hat{\mathbb{J}}(\pi_2)$.

b) SysID Tuning: This optimizes the SysID hyperparameters $\mathcal{H}_S$ for accuracy, akin to classical model selection. `AutoMPC` allows the choice to tune for 1-step prediction accuracy or k-step prediction accuracy on the testing set $\mathcal{D}_H$. However, experiments do not show a major controller performance difference between these methods.

c) Decoupled tuning: This approach first performs SysID tuning to fix $\hat{f}$ and then tunes the configuration over $\mathcal{H}_L \times \mathcal{H}_O$ to optimize performance $\hat{\mathbb{J}}$.

IV. Implementation

We provide an implementation of `AutoMPC` as an open-source Python library. We designed the package so that: *1)* `AutoMPC` should be accessible enough that a non-expert should be able to achieve a performant MPC with minimal manual tuning; *2)* `AutoMPC` should be useful to experts, providing tools to analyze system performance, and allowing a combination of automatic and manual fine-tuning; *3)* `AutoMPC` should provide a uniform API for components, allowing users to implement their own methods to be tuned by `AutoMPC`. The components implemented in the current version of `AutoMPC` and their hyperparameters are summarized in Fig. I and described in more detail next.

A. System Identification

Each SysID technique estimates the dynamics function f from a dataset $\mathcal{D}_I$.

ARX: A linear autoregression predicting the state as a linear function of the state and control history for the previous k time steps. That is

$$x_{t+1} = [x_t, \ldots x_{t-k+1}, u_t, \ldots, u_{t-k+1}]\theta$$

with θ the model coefficients. Training is performed using least-squares regression on the prediction error. The hyperparameter for ARX is the size of the history window k.

Koopman Operators learn a linear operator over an augmented state $\bar{x} = [x, \phi_1(x), \ldots, \phi_s(x)]^T$, where $\phi_1, \ldots, \phi_s$ are referred to as basis functions [8]. We use hyperparameters

TABLE I: For each SysID method, objective function, and optimization method, we list the total number of hyperparameters, and the names of hyperparameters. The possible number of active hyperparameters, which varies depending on the choice of values, is listed in parentheses.

SysID	#hyper. (act.)	Hyperparameters
ARX	1 (1)	`history`
Koopman	4 (3-4)	`usepolybasis, polydegree, usetrigbasis, trigfreq`
SINDy	4 (2-4)	`usepolybasis, polydegree, usetrigbasis, trigfreq`
GP	1 (1)	`inducingcount`
MLP	7 (4-7)	`numhiddenlayers, hiddensize{1,2,3,4}, learnrate, activation`
Objective	#hyper (act.)	Hyperparameters
Simple Quadratic	$2n+m$ ($2n+m$)	`qdiagvals{1,..,n}, fdiagvals{1,..,n}, rdiagvals{1,..,m}`
Gaussian Reg. Term	2 (2)	`stateregweight, controlregweight`
Optimization	#hyper. (act.)	Hyperparameters
LQR	2 (1-2)	`ishorizonfinite, horizon`
Direct Transcription	1 (1)	`horizon`
iLQR	1 (1)	`horizon`
MPPI	4 (4)	`horizon, numtrajs, noisemagn, costscale`

to select the basis functions, which can include polynomial terms x^s with $2 \le s \le 8$ and trigonometric terms $\sin(\omega x)$ and $\cos(\omega x)$ where $1 \le \omega \le 8$.

Sparse Identification of Nonlinear Systems (SINDy) represents dynamics in the form

$$f(x_t, u_t) = \sum_{i=1}^{N} a_i f_i(x_t, u_t),$$

where $f_1, \dots, f_N$ are nonlinear basis functions [7]. SINDy uses a fixed set of candidate functions $g_1, \dots, g_M$ and performs sparse linear regression to identify a subset of $N < M$ basis functions for the dynamics. We use the pySINDy library [10] in our implementation and use hyperparameters to select the set of candidate functions in the same way that we select basis functions for the Koopman operator.

Gaussian Processes (GPs) are a non-parametric model that has been commonly used for MPC [24]. Standard GPs use the full training set for inference, so they do not scale well to larger data sets. Instead, we use an approximate variational GP [17] implemented by the `gPyTorch` library [16], which selects a learnable subset of training points to use for inference. This subset is referred to as the inducing set. The hyperparameter we use for GP is the size of the inducing set.

Multi-layer Perceptrons (MLP) A feed-forward neural network architecture. We use hyperparameters to control the number of hidden layers in the network, the number of neurons in each layer, the choice of activation function (from ReLU, SeLU, tanh and sigmoid), and the learning rate during training. The number of training iterations are currently fixed.

B. Objective Functions

The next tunable component is the optimizer's objective function L. `AutoMPC` allows users to specify custom objective functions with tunable hyperparameters. We provide several pre-defined objectives which can be used as building blocks to define an objective. First, we consider the simple **quadratic objective**

$$L_Q(\mathbf{x}_{t:t+H}, \mathbf{u}_{t:t+H-1}; h_L) = (x_{t+H} - x_{\text{targ}})^T F (x_{t+H} - x_{\text{targ}}) + \sum_{i=t}^{t+H} [(x_i - x_{\text{targ}})^T Q (x_i - x_{\text{targ}}) + u_i^T R u_i],$$

with hyperparameters $h_1 \dots h_{2n+m}$ dictating the matrices

$$Q = \text{diag}(h_1, \dots, h_n), F = \text{diag}(h_{n+1}, \dots, h_{2n}), \text{ and } R = \text{diag}(h_{2n+1}, \dots, h_{2n+m}).$$

This objective can be effective for simple tasks which drive the system to a target state, such as the pendulum and cart-pole swing-up tasks.

The objective function can also include a **regularization term** to guide the optimization toward regions of the state space where model accuracy is better. We implement an option to penalize deviation from the data distribution of $\mathcal{D}$ as modeled by a multivariate Gaussian with mean μ_x and covariance matrix Σ_x. Similarly we model the controls in $\mathcal{D}$ with mean μ_u and covariance Σ_u. We add a regularization term to the objective:

$$L_R(\mathbf{x}_{t:t+H}, \mathbf{u}_{t:t+H-1}; h_L) = \sum_{i=t}^{t+H} h_1 (x_i - \mu_x)^T \Sigma_x^{-1} (x_i - \mu_x) + h_2 (u_i - \mu_u)^T \Sigma_u^{-1} (u_i - \mu_u)$$

with hyperparameters h_1, h_2.

C. Optimizers

We currently implement four optimizers. Each method has a hyperparameter for the planning horizon.

Linear Quadratic Regulators (LQR) solve the optimal control problem for tasks with linear models and quadratic cost functions [23]. We introduce a categorical hyperparameter to choose between finite and infinite horizon LQR.

Direct Transcription (DT) is a common method for formulating trajectory optimization as a nonlinear programming (NLP) problem [4]. DT requires the system model and the cost function to be differentiable. Our implementation uses IPOPT [33] to solve the NLP.

Iterative Linear Quadratic Regulator (iLQR) is a popular method for trajectory optimization similar to DT [32]. iLQR requires the system model to be differentiable and the cost function to be twice-differentiable.

Model Predictive Path Integral (MPPI) is a sampling-based optimizer that can be used with non-differentiable system models, and has been demonstrated to work effectively with neural networks [34]. We introduce hyperparameters

for the number of trajectories sampled on each iteration, the magnitude of the noise, and the cost scaling factor.

Note that the choice of optimizer is constrained by the choices of system model and objective function, and vice versa. For example, if the model is linear and the objective function is quadratic, then an LQR controller may be used, but a nonlinear system model requires the use of a more advanced optimizer. iLQR and Direct Transcription require the system model to be differentiable, while MPPI can work with non-differentiable models. At the moment, the model and optimizer classes must be chosen manually, but in future work we are exploring auto-tuning both.

V. Results

We consider three robotics tasks: pendulum swing-up, cart-pole swing-up, and half-cheetah running as implemented in the HalfCheetah-v2 environment in OpenAI Gym [6]. Each training set consists of 1,000 trajectories generated by applying, at each time step, controls chosen randomly from a uniform distribution. Each trajectory lasts for 10 s and the time step is 0.05 s. The train-test split is 50-50, and a MLP is trained as the surrogate model. The pendulum and cart-pole tasks are to move from the pole-down state to the pole-up state x_{goal}, and use the following rollout performance metric

$$J(\mathbf{x}_{1:T}, \mathbf{u}_{1:T-1}) = \sum_{i=1}^{T} \begin{cases} 1 & |\, x_i - x_{\text{goal}} \,|^2 > \delta \\ 0 & \text{otherwise} \end{cases}$$

where $\delta = 0.1$ for the pendulum and $\delta = 0.2$ for the cart-pole. For the half-cheetah, we use the same initial state and reward R as defined in OpenAI gym, and set

$$J(\mathbf{x}_{1:T}, \mathbf{u}_{1:T-1}) = 200 - R(\mathbf{x}_{1:T}, \mathbf{u}_{1:T-1}).$$

A. Surrogate Function Tuning

The first experiment, shown in Fig. 2 evaluates the correlation between the surrogate performance $\hat{\mathbb{J}}$ and the true performance $\mathbb{J}$. Each point in the scatter plot corresponds to a configuration h for a MLP-iLQR-Quad pipeline on the cart-pole task. The surrogate value varies depending on the random draw of $\mathcal{D}_H$, so the surrogate estimation procedure was repeated 10 times for each configuration. The error bars in Fig. 2 indicate the inter-quartile range. For some configurations the inter-quartile range is 0. The correlation between $\mathbb{J}$ and $\hat{\mathbb{J}}$ is not perfect and some configurations have considerably higher variance in $\hat{\mathbb{J}}$ than others. However, the relationship between $\mathbb{J}$ and $\hat{\mathbb{J}}$ is still mostly monotonic. This suggests that $\hat{\mathbb{J}}$ is a useful metric for tuning.

B. System ID Tuning

Our next experiment compares the performance of each system ID method on each sample system. The system ID data set $\mathcal{D}_I$ is split into a training set, validation set, and testing set containing 70%, 15%, and 15% of the trajectories respectively. Each method is tuned for 100 iterations (or until the search space is exhausted) by SMAC based on the 1 s rollout RMSE prediction error on the validation set.

Tab. II shows the 1 s error on the testing set for the tuned models. Note that the performance of the methods considered

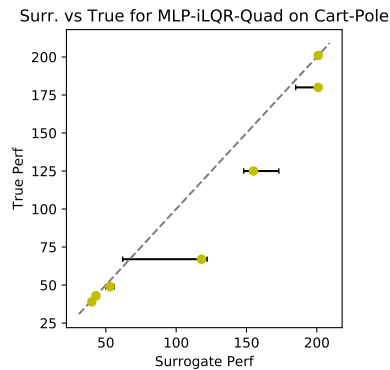

Fig. 2: Comparing true performance $\mathbb{J}$ and the surrogate performance $\hat{\mathbb{J}}$. Error bars indicate interquartile range over 10 random draws of the dataset. Equality is indicated by the dotted line.

here varies significantly, both across methods for a given task and across tasks for a given method. For example, SINDy is the best performing scheme for the pendulum and cart-pole tasks, but is significantly worse than all other methods for the half-cheetah test. This is because the dynamics of the pendulum and cart-pole systems can be represented as a sum of the nonlinear basis functions implemented for SINDy, while the dynamics of the half-cheetah are more complicated. This variability in performance highlights that methods are typically optimal only with respect to certain systems, motivating the need for auto-tuning to avoid the tedious process of searching among models.

C. Optimizer Tuning

Next, we compare the performance of each optimizer on all three tasks. For each system, we test with a hand-tuned system ID model (SINDy for pendulum and cart-pole and MLP for half-cheetah). For the LQR optimizer, the dynamics are linearized around the target state. The objective and optimizer hyperparameters are tuned for 100 iterations by SMAC based on the surrogate performance. Since the outcome of the tuning process vary, we tune each setting three times, randomizing the dataset, model weights, and SMAC seed. Tab. III reports the best tuning outcome for each task and optimizer. We also report the results of the iLQR optimizer with the Gaussian regularization objective. For the pendulum task, we observe that all methods perform comparably, though iLQR is slightly worse. For the cart-pole, we observe that all methods perform identically except for LQR which fails to complete the task. For the half-cheetah, iLQR achieves the best performance, while MPPI also performs decently. We note that as with system ID, the best optimizer varies from system to system.

D. End-to-end Tuning

Next, we evaluate the end-to-end system performance of several tuning approaches, and compare with a hand-tuned baseline. We use the example of MLP-iLQR-Quad on the cart-pole, and tune using the following approaches: 1) We tune the system ID for accuracy in the same manner as in Sec. V-B, while the objective and optimizer are fixed to the hand-tuned baseline; 2) We tune the system ID model based on the surrogate performance, keeping objective and

TABLE II: Performance of SysID methods for three systems. Each combination is tuned and evaluated based on the RMSE at a 1 s time horizon.

Sys. ID	Pendulum	Cart-pole	Half-cheetah
ARX	1.97	1.77	5.39
Koopman	2.37	1.94	5.35
SINDy	0.03	0.00	5,115,567
GP	0.36	1.13	2.45
MLP	0.10	0.15	1.57

TABLE III: Performance of optimizers for three systems. For each system, we fix a hand-tuned SysID, and tune the optimizer and objective function hyperparameters by an MLP surrogate. for each combination, we show the best result out of three tunes. Lower values indicate better performance.

Optimizer	Pendulum	Cart-pole	Half-cheetah
LQR	31.0	201.0	261.5
iLQR	35.0	21.0	-29.5
iLQR w/ Gauss. Reg.	31.0	21.0	134.1
Direct Transcription	30.0	21.0	221.8
MPPI	31.0	21.0	52.2

optimizer fixed; 3) Using a system ID pre-tuned on data (i.e. the result of tuning under the first mode), we tune the optimizer and objective hyperparameters; 4) We perform full pipeline tuning of all hyperparameters simultaneously.

Fig. 3 compares the results against the hand-tuned baseline over 100 tuning iterations. We run each tuning method five times and plot the median scores. Except for tuning system ID for accuracy, all methods perform comparably and are able to exceed the baseline.

E. Comparison to Offline RL

Next, we compare the true performance of full pipeline tuning compared to the offline RL algorithm Batch-constrained Deep Q Learning (BCQ) [15]. We train BCQ for one million iterations with the same dataset size and distribution as used in prior sections. To limit the hyper-parameter space for the half-cheetah, we use a custom quadratic objective which only tunes the weights for the vertical position, horizontal velocity, and front thigh, shin, and foot angles. The custom objective also allows the target velocity to be tuned. For each system, we repeat the tuning procedure ten times, randomizing the dataset, model weights, and SMAC seed. For the half-cheetah system, we observe that the surrogate model evaluation occasionally produces highly implausible estimates, and so we modify the tuning procedure to reject performance estimates which fall outside of a predefined plausible range. Results in Fig. 4 demonstrate that in the median case, our method achieves

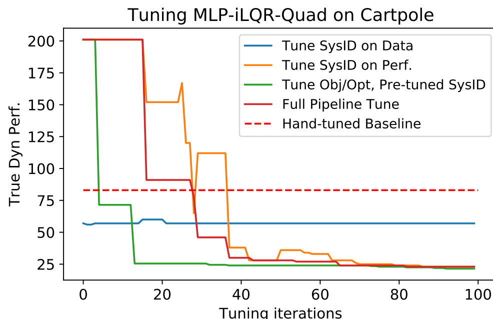

Fig. 3: Comparing auto-tuning procedures. The y-axis plots true performance on the ground truth dynamics, median of 5 trials.

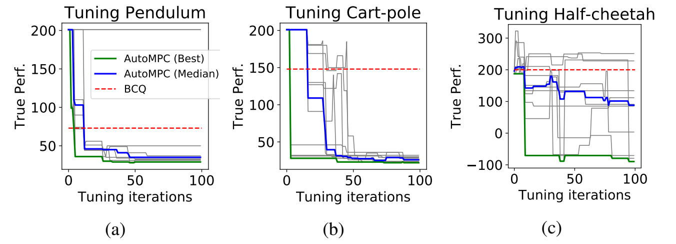

Fig. 4: Tuning curves for three robotics tasks compared the offline RL algorithm BCQ. For each system, we run ten tuning trials, plotted in grey. The median and best performances for each system are highlighted. Performance is evaluated with respect to ground truth dynamics.

superior performance to BCQ on each task. We remark that BCQ was originally presented as learning on data produced from a moderately good controller. In these experiments, we consider the much more challenging task of learning on data generated by a completely uninformed (random) controller. Even with uninformed initial data, `AutoMPC` achieves reasonable performance in almost all cases. In the more challenging half-cheetah system, the controllers produced by `AutoMPC` exhibit larger variations in performance, but at least some perform well. For such problems, we suggest to run the tuning process multiple times on the same dataset to obtain different controllers, then acquire a new dataset on the physical system using those controllers, and finally re-run tuning on the new dataset.

F. Analysis of Selected Hyperparameters

We observe a few interesting trends in the hyperparameters selected by `AutoMPC`. First, `AutoMPC` rarely selects ReLU activations for MLP system ID models. Instead, it more often selects tanh or SeLU activations. This may suggest that the discontinuities of ReLU cause problems for optimizations. Additionally, `AutoMPC` typically selects at least two MLP layers at least one of which has more than 150 neurons. Second, we observe that `AutoMPC` typically selects longer prediction horizons for the half-cheetah system, but prediction horizon appears to be less relevant for the simpler systems. Finally, for the pendulum and cart-pole systems, `AutoMPC` typically places a large objective weight on pendulum angle compared to other state dimensions.

VI. Conclusion & Discussion

In this work, we introduce a pipeline for tuning data-driven MPC controllers and present the associated software package `AutoMPC`. Contrary to existing methods that focus on certain stages of data-driven control, our approach jointly optimizes the hyperparameters of system identification, task specification, and control synthesis of several available schemes. Our framework is extensible, allowing users to add their own models, optimizers, or objective functions. We experimentally demonstrate that `AutoMPC` can produce good controllers even when learning on data produced by a totally uninformed method. In future work, we plan to extend `AutoMPC` to perform automatic model and optimizer selection in addition to hyperparameter tuning. We are also interested in exploring methods such as meta-learning to further improve speed and reliability.

References

[1] I. Abraham, G. De La Torre, and T. D. Murphey, "Model-based control using Koopman operators," in *Robotics: Science and Systems (RSS)*, 2017.

[2] S. Bansal, R. Calandra, T. Xiao, S. Levine, and C. J. Tomiin, "Goal-driven dynamics learning via Bayesian optimization," in *Conference on Decision and Control (CDC)*, 2017, pp. 5168–5173.

[3] A. Bemporad and M. Morari, "Robust model predictive control: A survey," in *Robustness in identification and control*. Springer, 1999, pp. 207–226.

[4] J. T. Betts, "Survey of numerical methods for trajectory optimization," *Journal of Guidance, Control, and Dynamics*, vol. 21, no. 2, pp. 193–207, 1998.

[5] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2017.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *Arxiv preprint arXiv:1606.01540*, 2016.

[7] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems," *National Academy of Sciences*, vol. 113, no. 15, pp. 3932–3937, 2016.

[8] M. Budišić, R. Mohr, and I. Mezić, "Applied Koopmanism," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 22, no. 4, p. 047510, 2012.

[9] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," in *Neural Information Processing Systems (NeurIPS)*, 2018, pp. 4754–4765.

[10] B. de Silva, K. Champion, M. Quade, J.-C. Loiseau, J. Kutz, and S. Brunton, "Pysindy: A python package for the sparse identification of nonlinear dynamical systems from data," *Journal of Open Source Software*, vol. 5, no. 49, p. 2104, 2020. [Online]. Available: https://doi.org/10.21105/joss.02104

[11] M. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *International Conference on Machine Learning (ICML)*, 2011, pp. 465–472.

[12] M. Fazel, R. Ge, S. M. Kakade, and M. Mesbahi, "Global convergence of policy gradient methods for the linear quadratic regulator," in *International Conference on Machine Learning (ICML)*, vol. 80, 2018.

[13] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Neural Information Processing Systems (NeurIPS)*, 2015, pp. 2962–2970.

[14] M. Forgione, D. Piga, and A. Bemporad, "Efficient calibration of embedded MPC," in *IFAC World Congress*, 2020.

[15] S. Fujimoto, D. Meger, and D. Precup, "Off-policy deep reinforcement learning without exploration," in *International Conference on Machine Learning (ICML)*, 2019, pp. 2052–2062.

[16] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, and A. G. Wilson, "Gpytorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration," in *Neural Information Processing Systems (NeurIPS)*, 2018, pp. 7576–7586.

[17] J. Hensman, A. Matthews, and Z. Ghahramani, "Scalable variational Gaussian process classification," in *International Conference on Artificial Intelligence and Statistics*, vol. 38, 2015, pp. 351–360.

[18] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International Conference on Learning and Intelligent Optimization*, 2011, pp. 507–523.

[19] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 1946–1956.

[20] S. Kamthe and M. Deisenroth, "Data-efficient reinforcement learning with probabilistic model predictive control," in *International Conference on Artificial Intelligence and Statistics*, 2018, pp. 1701–1710.

[21] R. Kidambi, A. Rajeswaran, P. Netrapalli, and T. Joachims, "Morel: Model-based offline reinforcement learning," *arXiv preprint arXiv:2005.05951*, 2020.

[22] A. Kumar, J. Fu, M. Soh, G. Tucker, and S. Levine, "Stabilizing off-policy q-learning via bootstrapping error reduction," in *Neural Information Processing Systems (NeurIPS)*, 2019, pp. 11 784–11 794.

[23] H. Kwakernaak and R. Sivan, *Linear Optimal Control Systems*. Wiley-interscience New York, 1972, vol. 1.

[24] G. Lee, S. S. Srinivasa, and M. T. Mason, "Gp-ilqg: Data-driven robust optimal control for uncertain nonlinear dynamical systems," *arXiv preprint arXiv:1705.05344*, 2017.

[25] I. Lenz, R. A. Knepper, and A. Saxena, "DeepMPC: Learning deep latent features for model predictive control." in *Robotics: Science and Systems (RSS)*. Rome, Italy, 2015.

[26] L. Ljung, "Perspectives on system identification," *Annual Reviews in Control*, vol. 34, no. 1, pp. 1–12, 2010.

[27] G. Mamakoukas, M. L. Castano, X. Tan, and T. Murphey, "Local Koopman operators for data-driven control of robotic systems." in *Robotics: Science and Systems (RSS)*, 2019.

[28] A. Marco, P. Hennig, J. Bohg, S. Schaal, and S. Trimpe, "Automatic LQR tuning based on gaussian process global optimization," in *International Conference on Robotics and Automation (ICRA)*, 2016, pp. 270–277.

[29] D. Piga, M. Forgione, S. Formentin, and A. Bemporad, "Performance-oriented model learning for data-driven MPC design," *Control Systems Letters*, vol. 3, no. 3, pp. 577–582, 2019.

[30] S. Ross and J. A. Bagnell, "Agnostic system identification for model-based reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2012, p. 1905–1912.

[31] D. H. Shim, H. J. Kim, and S. Sastry, "Decentralized nonlinear model predictive control of multiple flying robots," in *Conference on Decision and Control (CDC)*, vol. 4, 2003, pp. 3621–3626.

[32] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *International Conference on Intelligent Robots and Systems*, 2012, pp. 4906–4913.

[33] 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.

[34] 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 *International Conference on Robotics and Automation (ICRA)*, 2017, pp. 1714–1721.

[35] Y. Wu, G. Tucker, and O. Nachum, "Behavior regularized offline reinforcement learning," *arXiv preprint arXiv:1911.11361*, 2019.