A Sequential Optimisation Framework for Adaptive Model Predictive Control in Robotics

REL GUZMAN APAZA

Supervisor: Prof. Fabio Ramos
Associate Supervisor: Dr. Rafael Oliveira

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

Faculty of Engineering
The University of Sydney
Australia

November 2023

Authorship Attribution Statement

The work contained in the body of this thesis, except otherwise acknowledged, is the result of my own investigations.

Chapter 3 was published in the IEEE Robotics and Automation Letters (RA-L) journal (Guzman et al. 2021). Rel Guzman proposed a framework with the help of Fabio Ramos and Rafael Oliveira. The paper was written by Rel Guzman, but all co-authors contributed intellectually.

Chapter 4 was published at the 2022 International Conference on Robotics and Automation (ICRA) conference (Guzman et al. 2022b). Rel Guzman conducted the experiments and design of the proposed framework with the help of the same co-authors.

Chapter 5 was published at the 4th Annual Learning for Dynamics and Control Conference (L4DC) (Guzman et al. 2022a). Rel Guzman conducted the experiments and analysis of the paper. The same co-authors contributed intellectually with revisions and guidance.

Permission to include the published material has been granted by the author.

Prof. Fabio Ramos, in his capacity as the supervisor, confirms the correctness of the authorship attribution statements made above.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the University or other institute of higher learning, except where due acknowledgement has been made in the text.

Rel Guzman Apaza

April 2023

Abstract

State-of-the-art control and robotics problems have been solved for years using model-based control methods such as model predictive control (MPC) and reinforcement learning (RL). Both have shown promising results in easily handling complex dynamic domains, such as manipulation tasks. However, there is the issue that real-world systems are often subject to effects such as wear-and-tear, uncalibrated sensors, misspecifications, heteroscedasticity and so on. Such effects effectively perturb the system dynamics and can cause a learned controller to perform poorly in the real world. That and other factors lead to a well-known issue in robotics known as the reality gap, which comes up when transferring the robot from a simulator to the real-world environment. This work focuses on model-based methods with which the system can learn and adapt its own parameters to the environment. The main motivation of this thesis is to take advantage of both RL and control and propose a learning framework to deal with the reality gap and sequentially optimise robot performance by adapting MPC to the robot's decisions, simultaneously learning and finding an optimised controller under uncertainty in dynamics model parameters.

The first contribution is a reward-based framework for fine-tuning stochastic MPC, which in turn presents challenges in real-life situations concerning expensive evaluations and stochasticity. In order to deal with the data availability problem, fine-tuning is realised via a data-efficient Bayesian optimisation (BO) method that can handle the heteroscedastic noise across the MPC hyperparameter space to optimise the controller. The BO surrogate model is a GP that maps controller hyperparameters to the expected cumulative reward. The proposed optimisation framework is evaluated in simulated control problems and a robotic task.

In order to deal with the reality gap problem, the second contribution addresses the controller's ability to maximise rewards due to dynamics model misspecification. It extends the optimisation framework from the first contribution to obtain an adaptive stochastic MPC optimisation framework corresponding to optimising hyperparameters while jointly estimating probability distributions of physical parameters. The adaptation to the real world is performed by using a randomised dynamics model where the randomisation consists of the use of distribution-based physical parameters. The proposed optimisation framework is evaluated in simulated control problems and robotic manipulators.

Finally, this thesis explores the limitations of BO and how we can achieve improvements by using an alternative surrogate-based optimisation method to best adjust control hyperparameters to the current task in the presence of model parameter uncertainty and heteroscedastic noise. It proposes an adaptive optimisation framework that can automatically estimate control and model parameters by leveraging ideas from BO and supervised classification. The BO reformulation optimises expensive black-box functions by training a binary classifier. The final proposed framework is used to solve simulated control problems and simulated manipulators.

CONTENTS

List of Figures

List of Tables

Nomenclature

Basic Fonts

a	Scalar
$\mathbf{a}$	Vector
$\mathbf{A}$	Matrix
$\mathcal{A}$	Set

Linear Algebra

$[\mathbf{A}]_{ij}$	The element in the i'th row and j'th column of $\mathbf{A}$
$\mathbf{A}^\mathsf{T}$	Transpose
$\mathbf{A}^{-1}$	Inverse

Machine Learning and Optimisation

$g(\cdot)$	Function
$\hat{g}(\cdot)$	Function estimate
$\mathcal{D}$	Dataset
$\mathcal{D}_\mathbf{x}$	Input data points
$\mathcal{C}$	Label set
$h(\cdot)$	Predictive model
$\mathbf{w}$	Model parameters
$\mathbf{x}, y$	Input and output observations
$\mathbf{x}_{ij}$	j-th feature from input $\mathbf{x}_i$
$\ell(\cdot)$	Loss function
$L(\cdot)$	Expected loss
$\Phi(\cdot)$	Feature map
$k(\cdot, \cdot)$	Kernel function
k_ν	Noise kernel
Ω	Gaussian process hyperparameters

Π_{classif}	Probabilistic binary classifier
$\Pi(\cdot)$	Probability distribution of belonging to a positive class
$\Gamma(\cdot)$	Density ratio
τ	Threshold
γ, δ, ζ	Other hyperparameters

Probability Theory

$p(A)$	Probability of event A
$p(A \cap B)$	Probability of event A given event B
$\boldsymbol{\mu}$	Mean vector
$\boldsymbol{\Sigma}$	Covariance matrix
$\sim$	Distributed as
$\mathcal{N}$	Normal distribution
ν	Noise
σ_ν	Noise variance
$\mathbb{E}$	Expected value

Reinforcement Learning and Control

f	Dynamics model
$\hat{f}$	Simulated dynamics model
$\mathbf{a}_t$	Action or control sygnal at time t
$\mathbf{s}_t$	State at time t
r_t	Instant reward at time t
$\mathbf{o}$	Output signal at time t
$\bar{\mathbf{a}}$	Control trajectory
$\bar{\mathbf{s}}$	State trajectory
$\mathbf{s}_{\text{ref}}$	Reference or desired state
$\hat{\mathbf{s}}$	State estimate
$o(\cdot)$	Output function
$r(\cdot)$	Reward function
$q(\cdot), c(\cdot)$	Instant and terminal cost functions
ϵ	Perturbation

$\varrho(\cdot)$	Policy
J	Trajectory cost or trajectory episodic reward
T	Horizon or trajectory size

Special Symbols

$\approx$	Approximately equal to
$:=$	Equal by definition to
$\mathbb{R}$	Set of real numbers
$\mathbb{I}$	Indicator function
κ	Scaling factor
$\boldsymbol{\theta}$	Distribution-based physical variables
$\boldsymbol{\Psi}$	Parameters corresponding to the distribution-based variables
$\boldsymbol{\phi}$	Controller hyperparameters

CHAPTER 1

Introduction

The field of robotics has grown substantially with regard to its use in real-life settings, from industrial to healthcare applications. From a general perspective, robots are machines that perform a task that can be thought of as an arrangement of connected or related components that form a *robot system*. As opposed to *manual control* where a person decides actions for controlling the machine, a robotic system is aimed to perform the task autonomously using sensors and actuators as the way to interact with the real world, depending on its design and the environment surrounding it, e.g. manipulators as shown in Figure 1.1. Such an environment is known as the controlled system in the context of control theory, and automatic controllers are simply referred to as controllers (DiStefano III et al. 2014). In modern robotics, designing control systems is a complex task due to the dynamic and ever-changing nature of the real world. Addressing these challenges, a notable control method that has gained prominence is Model Predictive Control (MPC), particularly in its stochastic form. Stochastic MPC effectively incorporates probabilistic models to navigate real-world uncertainties, offering improved adaptability and intrinsic robustness (Fontes and Magni 2003). This approach demonstrates

(a) Jaco Manipulator (b) Humanoid (c) Self-Driving Car (d) Quadruped Robot

FIGURE 1.1. Robots in their environments. Source: IEEE ROBOTS catalogue.

how advanced control methods can autonomously manage the complexities of real-world environments, reducing the need for constant human oversight while enhancing the robot's ability to make independent decisions based on changing conditions.

The appealing idea is to develop autonomous robots in simulators before transferring them to the real world. A *physics simulator* or physics engine is a program that simulates the movement or interactions of objects in a virtual world. For example, physics simulators such as OpenAI and MuJoCo aim to provide robotics-focused physics simulations that serve as virtual playgrounds for robots (Collins et al. 2021). A mathematical model known as *dynamics model* mimics the behaviour of the robot dynamics in the simulator. Simulators are essential in robotics for designing physical parameters, prototyping algorithms, and in general, solving robotic tasks. Robotic systems developed in inaccurate simulators can produce incorrect results when deployed on real robots, emphasising the importance of precise simulators.

In the context of reinforcement learning (RL), the robot is seen as an autonomous decision-making agent. That decision-making is based on learning an input-output decision function based on trial and error. Rather than learning with the real robot, learning can be done using a dynamics model of the robotic system. The dynamics model is usually a probabilistic model known as Markov decision process (MDP), where, given an action, a performance measure known as reward is obtained as well as the next state (R. Sutton and Barto 2018). In an MDP, the robot is a decision-making agent that gets to observe the ground-truth state of the environment. This thesis assumes a *fully-observable* MDP to represent the robot's behaviour where it can fully observe the state of the environment at each timestep.

The intersection of control theory and RL is particularly evident in the application of stochastic MPC. In RL, the primary goal is to learn a policy that maximises a certain notion of cumulative reward, and stochastic MPC complements this by its ability to anticipate future states and adjust actions accordingly. Both control theory and RL areas of research have proposed approaches to tackle robotic tasks. Part of the objective of this thesis is to design a control system for robotics that resides in both control theory and RL. Control-theory solutions are usually model-driven, which means that they leverage the dynamics model, giving convergence guarantees at the

known operating conditions (Brunke et al. 2022). RL approaches are data-driven, which can make controllers adaptable to new contexts since only data can allow the controller to learn new behaviours.

Another point is regarding model-based optimisation. Modelling a Bayesian belief over the dynamics model is a way to model uncertainty arising from new contexts. Optimising a controller corresponds to *fine-tuning* it to fit the system behaviour, and it can be done by minimising the long-term reward of robot behaviours. For example, when RL and control are combined, a long-term reward can serve as a target output to be optimised, as it was done for car racing (Oliveira et al. 2018). Gaussian processes (GPs) are a state-of-the-art approach for Bayesian nonparametric regression of noisy functions. The regression method is known as Bayesian optimisation (BO) and is a well-known data-efficient method for black-box optimisation (X. Wang et al. 2023). GPs are also a popular choice for learning representations of dynamical systems (Scannell 2022) in the context of RL.

1.1 Motivations

1.1.1 The Reality Gap

The first motivation has to do with transferring controllers trained in simulators to the real world. The robot dynamics may not be perfectly modelled by an expert due to sensor measurement noise, disturbances being represented incorrectly, and the environment not being well-characterised, which leads to suboptimal controller designs when used in the real robot. Because simulators do not fully replicate real hardware behaviour, there is a problem known as *sim-to-real transfer*, which is still a very active area of research (Muratore et al. 2021b). The mismatch between simulation and reality is often called *reality gap*. Even though *simulated data* can result in a perfect controller in the simulator, there is no guarantee for an optimal solution in the real world, which raises the question of *how to design controllers to overcome the mismatch between simulation and reality?* It must be understood that by a simulator, one

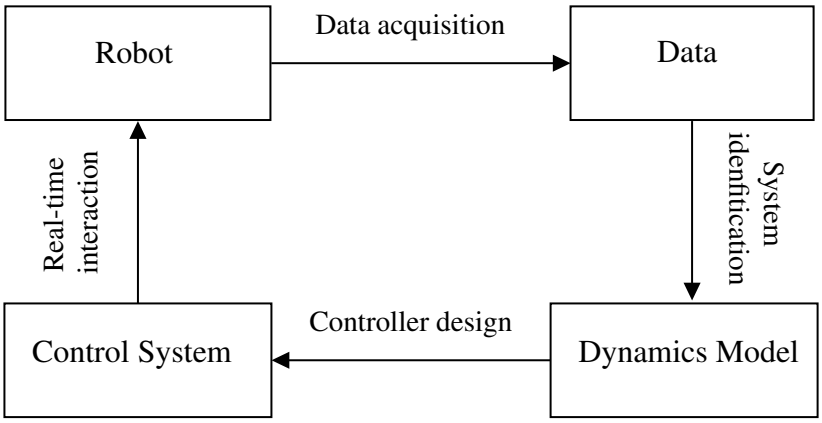


FIGURE 1.2. Data-driven control flow

can refer to the physics simulator or simply the dynamics model since, either way, a dynamics model is used for representing the robot's behaviour and interaction with the system.

1.1.2 Low Data Availability

The hope is that rather than needing to spend a lot of time understanding the environment the robot operates in, one should only collect a lot of experience and let the learning component handle the rest. In a data-driven approach Figure 1.2, lots of data can be collected and then used to learn a controller with highly effective approaches such as *deep learning*, but doing so requires the existence of millions of examples (Sünderhauf et al. 2018). Data availability is a common problem in robotics since collecting real data is needed to reduce the reality gap, but the issue is that *real data is expensive and time-consuming*. Real data is expensive and time-consuming due to the use of a real robot, as opposed to simulated data. That brings a second question: *how to get around the data availability problem in robotics?* Since the problem is that there is not enough data, a solution is to scale up data collection by collecting shared experiences from many robots, which can still be expensive due to mechanical system limitations. Another way is to use a more efficient learning algorithm. By following a *model-based RL* approach, learning can be done by simulations from an internally maintained dynamics model of the robot. Despite the well-known issues of simulation, model-based methods can cover more possible states of the robot without needing to use the actual robot.

1.1.3 Optimisation with Simulated Data

Given the advantages of working with simulated models, the question is *how to use simulated data to approximate real data and improve sim-to-real transfer?* Designing the world that the robot is going to interact with requires making assumptions about the environment, such as the physics parameters. That may bring the idea of updating the environment to make the simulator a better match for reality. If there is a mismatch between physics from the real world and physics from the simulator, a way to update the simulator can be to find the exact physics parameters from reality, which is a process known as *system identification*. However, a single best version of the environment can still be suboptimal. Considering there is a real data domain and a simulated data domain, an alternative solution can be domain randomisation, where instead of collecting the training data from a single simulated environment, the model is exposed to several variations of the environment, for example, for estimating the position and orientation of an object in a scene (Andrychowicz et al. 2020) by randomising parameters concerning those variables. As with any other optimisation problem, some parameters are more relevant than others. In general, randomising environments or robot parameters prepares the robot to adapt to different unknown scenarios.

1.2 Problem Statement

The problem in this thesis corresponds to a main research question:

How to properly optimise stochastic MPC to learn and adapt to the dynamics in real deployments?

Solving this question requires addressing two problems. The first sub-problem is about optimising the control system considering the expensive robot-environment interactions. A data-efficient approach can overcome the data availability problem. Besides using a simulated environment to cover more of the state space, a global optimisation method can be used. Then, the challenge is what parameters to explore and how to explore them to reduce the number of robot interactions with the real world.

The second sub-problem is about designing a controller that can deal with the sim-to-real transfer. The challenge is to find a way to optimise simulated data and real data together. The optimisation has to deal with the data availability problem while also minimising the reality gap. Preparing the robot for different scenarios has to be done efficiently.

1.3 Contributions

This thesis is composed of three contributing chapters, whose main contributions are summarised below.

The first contribution involves the development of a reward-based framework for fine-tuning stochastic MPC. This framework is directly applicable to RL, as it focuses on optimising the MPC strategy to maximise rewards, which is a core objective in RL. However, implementing such a framework in real-life scenarios introduces challenges, particularly concerning costly evaluations and inherent stochasticity. In order to deal with the data availability problem, fine-tuning is realised via a data-efficient Bayesian optimisation method that can handle the heteroscedastic noise across the MPC hyperparameter space to optimise the controller. The Bayesian optimisation surrogate model is a GP that maps controller hyperparameters to the expected cumulative reward. The proposed optimisation framework is evaluated in simulated control problems and a robotic task.

In order to deal with the reality gap problem, the second contribution addresses the controller's ability to maximise rewards due to dynamics model misspecification. It extends the optimisation framework from the first contribution to obtain an adaptive stochastic MPC optimisation framework corresponding to optimising hyperparameters while jointly estimating probability distributions of physical parameters. The adaptation to the real world is performed with the use of a randomised dynamics model where the randomisation consists of the use of distribution-based physical parameters. The proposed optimisation framework is evaluated in simulated control problems and robotic manipulators.

Finally, this thesis explores the limitations of Bayesian optimisation and how we can achieve improvements by using an alternative surrogate-based optimisation method to best adjust control hyperparameters to the current task in the presence of model parameter uncertainty and heteroscedastic noise. It proposes an adaptive optimisation framework that can automatically estimate control and model parameters by leveraging ideas from Bayesian optimisation and supervised classification. The Bayesian optimisation reformulation optimises expensive black-box functions by training a binary classifier. The final proposed framework is used to solve simulated control problems and simulated manipulators.

1.4 Thesis Outline

An overview of the following chapters presented in this thesis is presented below:

Chapter 2 presents necessary background concepts used throughout the thesis. Starting with control for robotics until the MPC formulation. Then supervised learning, optimisation, and Bayesian learning are introduced since they are essential concepts for learning from data and designing data-driven controllers under uncertainty. Finally, reinforcement learning concepts are introduced to allow reward-based optimisation. Concepts from all the background sections are used throughout the thesis. Next, the three contributing chapters (3 through 5) follow the same structure. After a motivating introduction that summarises the chapter's contributions, each chapter proceeds to describe related work, the methodology used to construct the proposed framework, and the experiments conducted. Chapter 3 presents the first contribution of this thesis: heteroscedastic optimisation for stochastic MPC, which addresses the data availability problem. Then, as a second contribution, Chapter 4 addresses the reality gap problem by extending the optimisation framework from the previous chapter to design a controller that adapts the controller simulated dynamics to the real world. Chapter 5 uses an alternative surrogate-based optimisation for the framework proposed in the previous chapter. Each chapter performs and analyses experiments in control and robotic tasks. This thesis closes in Chapter 6 with a review of the robotics problems addressed and a review of each chapter's contribution. Finally, some directions for future research are provided.

CHAPTER 2

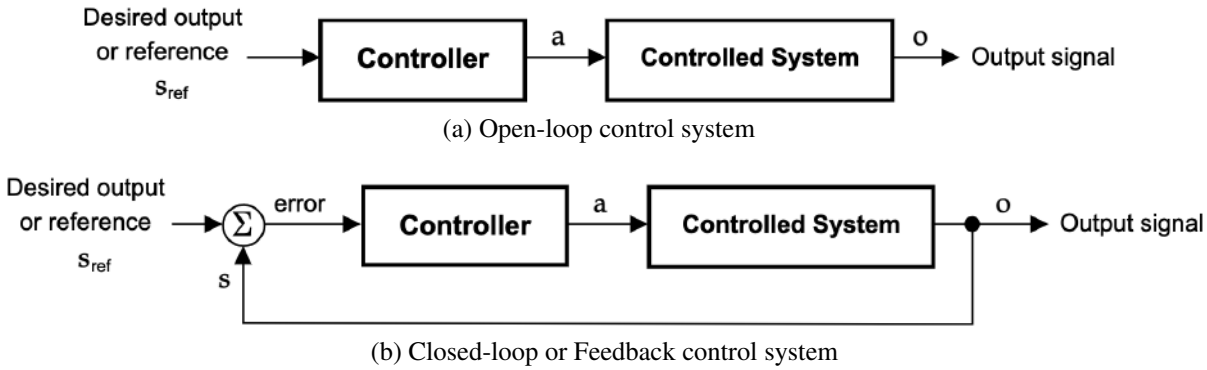
Background

This chapter presents a review of the necessary background that forms the basis for the methods used in this thesis. As stated in the previous chapter, the problem has to do with optimising control systems, specifically stochastic model predictive control, for which control theory and optimisation concepts are needed. Then, the problem is also about designing a controller that can learn and adapt the robot dynamics to the real world, for which machine learning concepts are needed, specifically supervised learning, Bayesian learning, and reinforcement learning.

First of all, this chapter starts by describing concepts from the control component. Section 2.1 introduces the areas of feedback control and optimal control focused on robotics. Starting from the concept of system dynamics until model predictive control. Secondly, for the learning component, supervised learning concepts are described in Section 2.2, and optimisation concepts are presented next in Section 2.3. Supervised learning is the basis for understanding machine learning, and it inherently depends on function optimisation concepts. Next, a probabilistic perspective to learning is presented as Bayesian learning in Section 2.4 in order to deal with real-world uncertainty, which leads to Bayesian optimisation described in Section 2.5. Finally, reinforcement learning is described in Section 2.6, which is presented as another type of learning, but since it is oriented toward data efficiency in robotics, it is mainly centred on model-based methods. All the concepts presented contribute to understanding the proposed framework.

2.1 Control for Robotics

In order to understand control theory, it is useful to understand what controlling means in the context of robotics. First of all, instead of manually controlling a system, plant, or process to perform some task for building a robotic system, it is helpful to think of a control system for automatic control. An example of a system that performs *automatic control* can be a thermostat, whose purpose is to maintain the temperature in a room. A *system* can be an arrangement, set, or collection of things connected or related in such a manner as to form an entirety or whole (DiStefano III et al. 2014). Therefore, a *control system* is an arrangement of components connected with the purpose of regulating another system or itself by regulating the system according to a desired behaviour $\mathbf{s}_{\text{ref}}$ by applying a *control signal* or control $\mathbf{a}$ to the system (Astrom and Murray 2008). The system's output is a controlled *output signal* $\mathbf{o}$. A controlled system can be represented as a block diagram as the ones in Figure 2.1.



(a) Open-loop control system

(b) Closed-loop or Feedback control system

FIGURE 2.1. Open-loop and closed-loop control systems. (a) The output u of the controller is used as the control signal for the controlled system. (b) The output $\mathbf{o}$ of the system, in this case, is the state $\mathbf{s}$ so that the controller computes an error $\mathbf{s}_{\text{ref}} - \mathbf{s}$ and corrects the system behaviour.

2.1.1 Feedback Control

There are open-loop and closed-loop control systems. Figure 2.1a is an example of an *open control system* where the desired output is planned beforehand, and the controller sends a corresponding control signal. An example of an open-loop control system is a microwave oven that

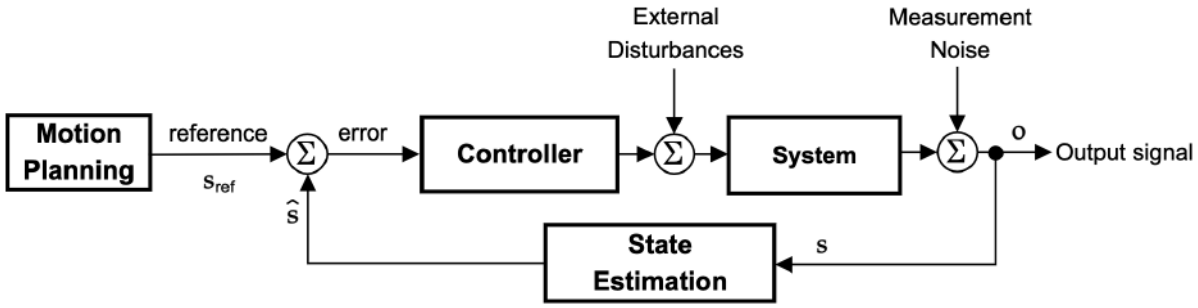


FIGURE 2.2. Typical components and variables in control system design.

operates for a given amount of time and then stops. Meanwhile, if the controlled output signal $\mathbf{o}$ is measured and fed back for use in the control computation, the system is called *closed-loop or feedback control system*. A closed-loop control system is shown in Figure 2.1b where $\mathbf{s}$ is known as the *state* of the system. An example would be a microwave oven that senses the temperature and takes it as feedback. Another example is an autonomous car that regulates its speed. The *reference* is the desired behaviour of the system represented as a *desired state* $\mathbf{s}_{\text{ref}}$. A desired state can be a steady state, which refers to a condition where the system variables have reached a constant value and remain unchanged over time (Edelstein-Keshet 2005). Then the error $\mathbf{s}_{\text{ref}} - \mathbf{s}$ is known as *steady-state error*. In most cases, the desired state of the system is assumed to be $\mathbf{0}$.

Feedback control systems are widely used in automation and robotics to enable the system itself to sense these malfunctions and to correct them in some manner with a *control signal* $\mathbf{a}$ without human intervention. In robotics, the *environment*, which is where the system is operating, does not have fully predictable behaviour because of *external disturbances* that can perturb the dynamics of the system as in Figure 2.2. When the wind is light, and visibility is good, the autonomous car's primary concern is to update its desired position in order to follow the road. However, a strong wind can blow the car off the course. To compensate for external disturbances, the car must make some adjustments to correct its position errors. Now, suppose the wind is accompanied by snow. Snow constitutes a visual noise that corrupts the position measurements taken by a sensor. *Measurement noise* is usually found in sensor reading. The car may not be able to see the road very well, so it may end up changing its course to compensate for the measurement noise.

Motion planning and state estimation are essential components for robotic systems that interact with the real world. In the field of artificial intelligence (AI), "planning" usually refers to finding a sequence of logical decisions or actions that transform an initial state of the system to the desired goal state where the system dynamics is often neglected or simplified (Russell and Norvig 2021). In this thesis, the motion planning problem is referred to as the problem of computing a feasible reference *motion plan, trajectory or path*, which is a sequence of states and controls that guide the robot from its initial state to the desired goal state. Such a trajectory must satisfy the robot's physically imposed constraints as well as constraints imposed by surrounding obstacles while at the same time minimising a specified performance measure. This thesis makes use of motion plans that are computed and updated in real time.

At the core of probabilistic robotics is the idea of *state estimation*, which is about estimating the robot state from sensor data since determining the exact pose of a robot is not feasible (Thrun et al. 2005). For example, moving a mobile robot is relatively easy if the exact location of the robot and all nearby obstacles are known. Unfortunately, these variables are not directly measurable. Instead, a robot has to rely on its sensors to gather this information. Sensors carry only partial information about those quantities, and their measurements are corrupted by measurement noise. This thesis does not deal explicitly with state estimation, although it emphasises the importance of taking state uncertainty into account by handling uncertain physical parameters and sources of uncertainty that can affect state estimation.

2.1.2 System Dynamics

Much of the impressive results in robotics in recent years have been achieved through the application of state space control in conjunction with rigid body mechanics and motion planning (Lynch and Park 2017). State space control refers to using a mathematical model known as *dynamical system* that specifies the temporal evolution of the behaviour or state of the system (R. C. Bishop 2011). The state of physical systems can be described with equations of motion, which are also dynamical systems. For example, the motion of a car in uniform rectilinear motion can be described as a system with linear dynamics. An equation of motion is usually denoted as a system of linear ordinary differential equations (ODE) with the form $\frac{d\mathbf{s}_t}{dt} = \mathbf{A}\mathbf{s}$

where $\mathbf{A}$ is some constant matrix, $\mathbf{s}_0$ is some initial state, and $\frac{d\mathbf{s}_t}{dt}$ corresponds to a vector of first-order time derivatives of the state at time t. Meanwhile, a discrete-time state-space model can be obtained as a result of the discretisation of a continuous-time system, which creates a correspondence of a next state $\mathbf{s}_{t+1}$ with $\frac{d\mathbf{s}_t}{dt}$. To simplify the analysis, such a system is usually assumed to be time-invariant, meaning that the output does not depend on when an input was applied. A *linear time-invariant (LTI)* discrete-time system has the standard form:

$$\begin{aligned} \mathbf{s}_{t+1} &= \mathbf{A}\mathbf{s}_t + \mathbf{B}\mathbf{a}_t \\ \mathbf{o}_t &= \mathbf{C}\mathbf{s}_t + \mathbf{D}\mathbf{a}_t \ , \end{aligned} \tag{2.1}$$

where $\mathbf{s}$, $\mathbf{a}$, and $\mathbf{o}$ are indexed by time. The vector $\mathbf{s}_t$ is the *state* of the system at time t. We denote the state space dimension by n so that $\mathbf{s}_t \in \mathbb{R}^n$. In general, there are multiple inputs to the system. We can define the control signal as a vector $\mathbf{a}_t \in \mathbb{R}^p$, so that $\mathbf{a}_t = \begin{bmatrix} a_{t1} & a_{t2} & \cdots & a_{tp} \end{bmatrix}^T$, and the output signal $\mathbf{o}_t = \begin{bmatrix} o_{t1} & o_{t2} & \cdots & o_{tq} \end{bmatrix}^T$. Each of these outputs represents a sensor measurement of some of the states of the system. The state (or system) matrix $\mathbf{A}$ is an $n \times n$ matrix representing how the states of the system affect each other. The input matrix $\mathbf{B}$ is an $n \times p$ matrix representing how the inputs to the system affect the states. The output matrix $\mathbf{C}$ is a $q \times n$ matrix representing the portions of the states that are measured by the outputs. Finally, the feedthrough (or feedforward) matrix $\mathbf{D}$, which is usually not considered, is a $q \times p$ matrix representing the portions of the control signal that are measured by the outputs. A, B, C, and D are constant matrices. C is usually the identity, making $\mathbf{o}_t = \mathbf{s}_t$.

2.1.3 Optimal Control

Optimisation refers to the problem of choosing a set of parameters that maximise or minimise a given function. In optimal control, we seek to optimise a given specification, choosing the parameters that maximise the performance (or minimise the cost) (Liberzon 2011) since, in most cases, simply achieving the desired objective of moving a robot to a desired final state $\mathbf{s}_{t_f}$ is insufficient. For example, the velocity and trajectory taken to reach the final state, the obstacles encountered along the way, and the energy consumed are all additional factors that

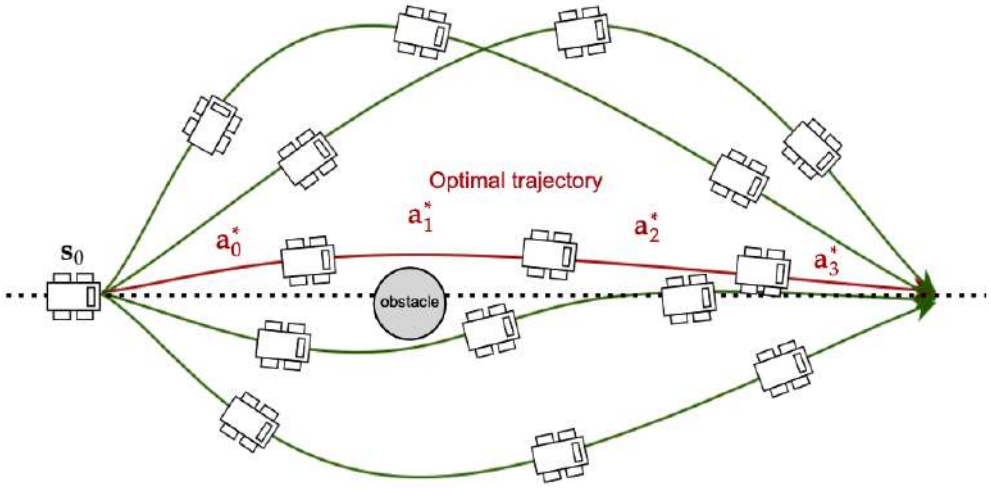


FIGURE 2.3. Optimal trajectory shown in red for an autonomous car in a single-obstacle environment. The dashed line in the reference path.

can be considered to fully evaluate the robot's performance. In robotics, optimal control refers to the optimisation of trajectories for a system over a period of time $[t_0, t_f]$, obtaining *optimal trajectories*. Trajectories are expressed as a sequence of states and controls

$$\left\{\mathbf{s}_{t_0}, \mathbf{a}_{t_0}, \mathbf{s}_{t_1}, \mathbf{a}_{t_0}, \ldots, \mathbf{s}_{t_f-1}, \mathbf{a}_{t_f-1}, \mathbf{s}_{t_f}\right\} \ , \tag{2.2}$$

where $\{\mathbf{a}_{t_0}, \ldots, \mathbf{a}_{t_f-1}\}$ is called a control trajectory that will be denoted as $\bar{\mathbf{a}}$, which in turn determines the state trajectory $\{\mathbf{s}_{t_0}, \ldots, \mathbf{s}_{t_f}\}$ that will be denoted as $\bar{\mathbf{s}}$. There are two essential concepts in optimal control: the dynamical system and the cost function. The controller decides *optimal controls* for the trajectory. Following the state-space notation, in control theory, the system behaviour is usually described by a dynamical system known as *dynamics model*

$$\begin{gathered} \mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t), \quad t \in [t_0, t_f] \\ \mathbf{s}_{t_0} : \text{ initial state } , \end{gathered} \tag{2.3}$$

where f is a function that depends on both the state and the control $(\mathbf{s}_t, \mathbf{a}_t)$ and describes the mechanism by which the state is updated from time t to time $t+1$. The admissible control trajectories are control sequences $\bar{\mathbf{a}} = \{\mathbf{a}_i\}_{i=t_0}^{t_f}$, and the objective is to allow numerical solutions to *trajectory optimisation* so as to harness the power of computational optimisation theory to solve analytically intractable control problems and find an optimal trajectory $\bar{\mathbf{a}}^* = \{\mathbf{a}_i^*\}_{i=t_0}^{t_f}$. An example of trajectory optimisation is shown in Figure 2.3. The fundamental idea in optimal

control is to formulate the goal of control as the long-term optimisation of a scalar cost function $J(\bar{\mathbf{a}})$. This cost function associates the cost to each possible behaviour parameterised by states $\bar{\mathbf{s}}$ and controls $\bar{\mathbf{a}}$, which are both functions of time. It turns out that the optimisation problem often becomes a lot more tractable (both analytically and computationally) if we are willing to design the cost with an additive form. The cost function takes the form

$$J(\bar{\mathbf{a}}) = q\left(\mathbf{s}_{t_f}\right) + \sum_{t=t_0}^{t_f-1} c(\mathbf{s}_t, \mathbf{a}_t) \,, \tag{2.4}$$

where c denotes an instant cost function and q denotes a terminal cost function. t_f is the terminal time. $\mathbf{s}_{t_f}$ is the terminal state. The optimal control problem consists of finding the optimal controls that minimise $J(\bar{\mathbf{a}})$. Some control problems have a duration called *time horizon or horizon length* T. For example, imagine a robot working on an assembly room floor and being allocated a fixed amount of time to complete each assembly (or pick-and-place) task. In general, finite horizon problems define a finite time interval, $t \in [0, T]$. The time horizon T can be given or left as a free variable to be optimised. Also, it is often wanted to achieve this objective at the lowest cost possible. The cost minimisation problem is defined as

$$\bar{\mathbf{a}}^* = \underset{\bar{\mathbf{a}}}{\operatorname{argmin}}\, J(\bar{\mathbf{a}}) \,. \tag{2.5}$$

2.1.4 Model Predictive Control

A modern optimal control scheme widely used for robot motion planning is *model predictive control (MPC)*, which is used for finding optimal trajectories over a finite time horizon (Ljungqvist 2020). Its definition can be understood by splitting the term "Model-based predictive control" into its meaningful parts: model-based and predictive control. The model-based aspect implies the necessity of a system model, while predictive control refers to the prediction of future system outputs and states.

In MPC, future values of the system outputs, denoted as $\mathbf{o}_t$, and the states, denoted as $\mathbf{s}_t$, are predicted. Specifically, a discrete-time MPC approach involves defining a mathematical model of the system as a discrete-time dynamical system, as described by Equation 2.3. Given the

current state $\mathbf{s}_t$, MPC seeks an optimal control trajectory $\{\mathbf{a}_t^*, \ldots, \mathbf{a}_{t+T-1}^*\}$ over a finite time horizon T, formulated as the following optimisation problem:

$$\begin{aligned}
&\min_{\substack{\mathbf{a}_t,\ldots,\mathbf{a}_{t+T-1} \\ \text{(planning horizon } T\text{)}}} q\left(\mathbf{s}_T\right) + \sum_{i=t}^{t+T-1} c\left(\mathbf{s}_i, \mathbf{a}_i\right) \\
&\quad \text{subject to } \mathbf{s}_{i+1} = f(\mathbf{s}_i, \mathbf{a}_i) \qquad \forall i = t, t+1, \ldots, t+T-1 \\
&\qquad\qquad \mathbf{s}_i \in \mathcal{X} \qquad\qquad \forall i = t, t+1, \ldots, t+T \\
&\qquad\qquad \mathbf{a}_i \in \mathcal{A} \qquad\qquad \forall i = t, t+1, \ldots, t+T-1 \\
&\qquad\qquad \mathbf{s}_i : \text{ initial state } ,
\end{aligned} \tag{2.6}$$

where the states and controls can be constrained by $\mathcal{X}$ and $\mathcal{A}$, e.g. magnitude constraints. The dynamics model in MPC can be either linear or nonlinear, with the latter case known as *nonlinear MPC*.

When the dynamics are linear, represented as $\mathbf{s}_{t+1} = \mathbf{A}\mathbf{s}_t + \mathbf{B}\mathbf{a}_t$, and the cost function is quadratic, the optimisation problem can be solved using a Linear Quadratic Regulator (LQR) (Liberzon 2011). LQR is commonly available as a built-in routine in many computational packages and provides an analytical solution. However, the computational cost of solving the Riccati equation, central to LQR, scales with the cube of the state space dimension. This cubic scaling makes LQR computationally expensive for real-time applications in robotics, particularly for systems with high-dimensional state spaces, relegating its use mostly to offline calculations. Unlike MPC, LQR does not inherently accommodate constraints, making it a specific, less flexible instance of MPC.

2.1.5 Practical Application of MPC

Building upon the theoretical foundations of MPC, its practical application involves computing feasible optimal trajectories iteratively. This process involves continually solving the optimal control problem from the current state $\mathbf{s}_t$. In this way, MPC is capable of generating open-loop control trajectories in real time. Upon computation, the first control element $\mathbf{a}_t^*$ is immediately applied to the system.

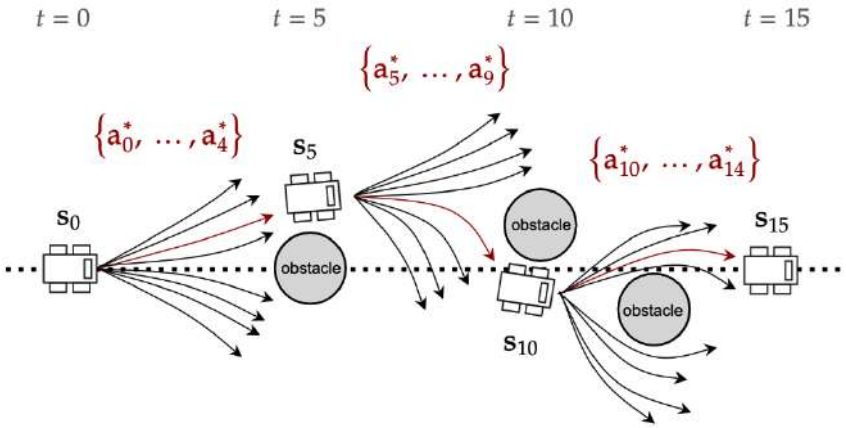


FIGURE 2.4. Example of MPC applied to an autonomous car in an environment with obstacles. The optimal trajectories are shown in red.

The dynamics model within MPC can be an estimate $\hat{f}$ of the *real dynamics model* f, allowing the control system to perform only simulations to find optimal controls. $\hat{f}$ can be called *simulated dynamics model* since it allows simulated interactions with the system. An example is shown in Figure 2.4, where the autonomous car should avoid the obstacles following an MPC procedure with a horizon of $T = 5$. The trajectory optimisation method has to be computationally cheap and fast to run in real time. Long simulation times necessarily lead to reducing the number of computer experiments obtained, which affects the system performance. Planning trajectories ahead in real time is possible when dealing with a closed-form optimisation scheme, such as with LQR or with GPU-based trajectory sampling (Williams et al. 2018).

A block diagram for an MPC method can be seen in Figure 2.5, and the way it is usually implemented is shown in Algorithm 1. The algorithm receives the simulated dynamics model, a finite length horizon, and instant and terminal cost functions to use them iteratively until some stopping condition is met, which in some cases is reaching T iterations or achieving a specific desired state. An MPC method starts by getting the current state of the system. Then, it plans a trajectory optimised with a method that can be LQR.

Besides linear and nonlinear MPC, there are also some data-driven MPC formulations for unknown dynamical systems and for disturbance adaptation. Data-driven control refers to the use of experimental data to learn algorithms that can adapt and improve control systems. Machine learning methods, which will be seen in Section 2.2, are used to extract patterns from data and

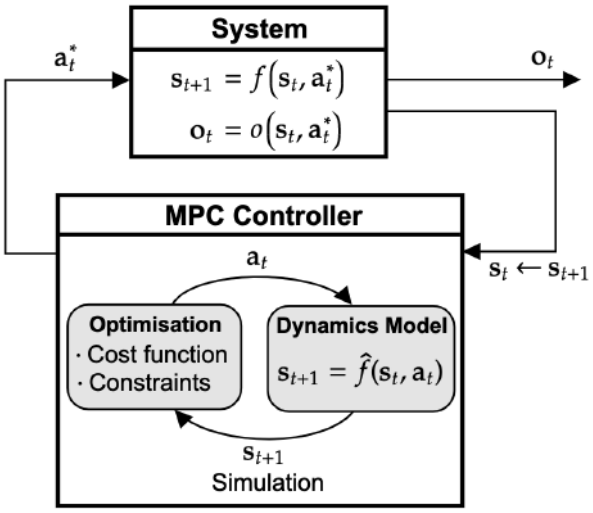


FIGURE 2.5. MPC diagram where the controller component makes use of an optimisation procedure plus the simulated dynamics model.

to learn components of the control system (Brunton and Kutz 2019). The main requirement for data-driven methods in robotics is to work well in real-world applications as they depend on data efficiency: how long it takes to learn. One of the main reasons for using data comes from some challenges of control systems that include unknown nonlinear dynamical systems since most real systems do not have well-defined equations of motion. Among the ways data is used in control, there is data-driven modelling that refers to learning the simulated dynamics model $\hat{f}$ using measurements of the input and output signals of the system, which can be thought of as *system identification* (Muratore 2021). An effective way is to also learn the controller, which is also known as *machine learning control*, besides the system dynamics. There is *learning-based MPC* for which most research has focused on the automatic data-based adaptation of the dynamics. Also, there are learning-based control methods that deal with the rest of the optimisation problem formulation that mostly constrain or regularise the controller favourably with respect to the underlying task, e.g. the cost function or the constraints (Hewing et al. 2020).

2.1.6 Model Predictive Path Integral Control

The methods discussed previously assumed the nonexistent external uncertainty in the system, and they are part of what is known as deterministic optimal control, e.g. deterministic MPC.

Algorithm 1: Model Predictive Control

input : $\hat{f}$ – Simulated dynamics model
T – Finite horizon length
c, q – Instant and terminal cost functions

while *stopping condition not met* **do**
  $\mathbf{s}_t \leftarrow$ GetCurrentState()
  $(\mathbf{a}_t, \ldots, \mathbf{a}_{t+T-1}) \leftarrow$ PlanTrajectory$(\hat{f}, \mathbf{s}_t, c, q)$ // Plan a trajectory as with LQR
  $\mathbf{a}^* \leftarrow \mathbf{a}_t$
  SendToActuators$(\mathbf{a}^*)$ // Execute first planned control signal

Stochastic optimal control, on the other hand, deals with uncertainty in system dynamics due to external disturbances, which makes it a more realistic approach. Disturbances can include inaccurate sensor measurements, inherent noise in the resultant state or imperfect control signals (Jacob Mathew 2020). It can be understood that since the dynamical system is subject to stochastic disturbances, described as $\mathbf{w}_t$, it is represented as a stochastic dynamical system

$$\begin{aligned} \mathbf{s}_{t+1} &= f(\mathbf{s}_t, \mathbf{a}_t, \mathbf{w}_t), \quad t \in [t_0, t_f] \\ \mathbf{s}_{t_0} &: \text{ initial state } , \end{aligned} \tag{2.7}$$

where the disturbance parameter can be additive. For example, an LTI stochastic dynamical system can be $\mathbf{s}_{t+1} = \mathbf{A}\mathbf{s}_t + \mathbf{B}\mathbf{a}_t + \mathbf{E}\mathbf{w}_t$ where $\mathbf{E}$ is constant. Such disturbance input is a noise term with unknown current and future values, and it is defined as a random variable with a known probability distribution. It is a random vector in a probability space with support $\mathbb{R}^{n_w}$. The noise term has the role of real disturbance acting on the system or can be used to represent an *unmodeled dynamics* (Giulioni 2015).

Moreover, since deterministic MPC allows constraint satisfaction and the trajectories are generated by optimising some criterion in a closed-loop scenario, deterministic MPC possesses some desirable intrinsic robustness properties (Fontes and Magni 2003). The term robust control is associated with the class of techniques that try to guarantee some worst-case performance or a worst-case bound by preserving performance and stability for each possible realisation of bounded disturbances or uncertainty (Tedrake 2023). However, MPC does not necessarily provide strict robustness guarantees. There are robust MPC methods that are beyond the scope of this work since the objective is not to design a robust controller. Anyhow, when the intrinsic robustness of deterministic MPC is not enough, stochastic MPC methods have been developed

to take into account uncertainties. Among the methods for stochastic optimal control, there is *Model Predictive Path Integral (MPPI)*, which is a gradient-free trajectory optimisation method that solves stochastic optimal control problems based on the (stochastic) sampling of the system trajectories. MPPI is also known as a sampling-based MPC algorithm (Williams et al. 2016). It has proven its potential in robotics due to its ability to handle uncertainty with minimal assumptions on the dynamics and cost function, which has to be dense. Also, due to its parallelisable sampling via Graphics Processing Unit (GPU) programming, it has been useful for complex robotic tasks from manipulators to autonomous vehicles (Kim et al. 2022; Manuelli 2020).

Since MPPI is an MPC method, it is a variation to Algorithm 1 where the trajectory optimisation is sampling-based. Algorithm 2 shows how MPPI works in detail. First of all, the algorithm receives the current state $\mathbf{s}_t$, which comes from the system, and we define the timestep t as the current *feedback-control loop iteration*. The controller receives an initial control trajectory $\bar{\mathbf{a}} = (\mathbf{a}_t, \ldots, \mathbf{a}_{t+T-1})$ which is defined as $\mathbf{0}$ at $t = 0$ as well as $\mathbf{a}_{\text{new}}$. Such a trajectory is updated and becomes the nominal control trajectory of size T at the end of the algorithm. It is called *nominal control trajectory* because it is an ideal or intended trajectory but not necessarily optimal since it is updated again at the next timesteps. The main feature of MPPI is that it performs Monte Carlo importance sampling to weight and optimise trajectories. It samples M sequences of perturbed control trajectories $\{\mathbf{a}_i + \epsilon_i\}_{i=t}^{t+T-1}$, where ϵ_i is an additive Gaussian noise $\epsilon_i \sim \mathcal{N}(0, \boldsymbol{\Sigma})$. $\boldsymbol{\Sigma}$ is a covariance matrix hyperparameter defined as a scalar matrix

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_\epsilon^2 & 0 & \ldots & 0 \\ 0 & \sigma_\epsilon^2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \sigma_\epsilon^2 \end{bmatrix}, \boldsymbol{\Sigma} \in \mathbb{R}^{p \times p}, \tag{2.8}$$

where p is the control vector size and σ_ϵ^2 is called the control variance, which leads to more varying and forceful actions when its value is high. The sampling of each control trajectory is known as a *rollout*. The rollout and evaluation of trajectory m from M trajectories correspond to lines 4 to 8. Next, it obtains the minimum cost β from all trajectories. β is used to compute the perturbation weights w from line 12 and the normalisation constant η from line 10. The computed weights are used for the importance sampling of trajectories. The temperature

hyperparameter $\lambda \in \mathbb{R}^+$ is also part of the calculation, and it corresponds to the Boltzmann distribution calculation from lines 10-12. Finally, the nominal control trajectory $\bar{\mathbf{a}}$ is created using the weights in line 14. The optimal action that is sent to the system actuators is the first element of $\bar{\mathbf{a}}$. As the last step, the new initial control trajectory for the next iteration is set in line 15. Finally, the first optimal control signal $\mathbf{a}^*$ is to be sent to the system actuators in line 17. The resulting controller is not only model-based but also sampling-based since it samples trajectories by generating control signals from a normal distribution.

MPPI performance is determined by its hyperparameters. For the temperature hyperparameter, $\lambda \to 0$ leads to a single trajectory having a higher probability of occurrence, and the control variance σ_ϵ^2 results in more varying and forceful actions as it increases (Williams et al. 2018). In this way, both λ and σ_ϵ can be considered to be balancing the exploration and exploitation of

Algorithm 2: Model Predictive Path Integral

input : $\hat{f}$ – Simulated dynamics model
M – Number of trajectories
T – Finite horizon length
λ – Temperature parameter for the sampling
Σ – Covariance of the noise ϵ_i for the perturbed actions
$\bar{\mathbf{a}} = (\mathbf{a}_t, \ldots, \mathbf{a}_{t+T-1})$ – Initial control trajectory
c, q – Instant and terminal cost functions
$\mathbf{a}_{\text{new}}$ – Control input for initialization

1 **while** *stopping condition not met* **do**
2 $\mathbf{s}_t \leftarrow$ GetCurrentState()
3 **for** $m \leftarrow 0$ **to** $M-1$ **do**
4 Sample $\left\{\epsilon_t^{(m)}, \epsilon_{t+1}^{(m)}, \ldots \epsilon_{t+T-1}^{(m)}\right\}$ $\epsilon_i^{(m)} \sim \mathcal{N}(0, \Sigma)$
5 **for** $i \leftarrow t$ **to** $t+T-1$ **do**
6 $\mathbf{s}_{i+1} = \hat{f}\left(\mathbf{s}_i, \mathbf{a}_i + \epsilon_i^{(m)}\right)$
7 $c^{(m)} \mathrel{+}= c(\mathbf{s}_{i+1}) + \lambda \mathbf{a}_i^T \Sigma^{-1} \epsilon_i^{(m)}$
8 $c^{(m)} \mathrel{+}= q(\mathbf{s}_{t+T})$
9 $\beta \leftarrow \min_m \left[c^{(m)}\right]$
10 $\eta \leftarrow \sum_{m=0}^{M-1} \exp\left(-\frac{1}{\lambda}\left(c^{(m)} - \beta\right)\right)$
11 **for** $m \leftarrow 0$ **to** $M-1$ **do**
12 $w^{(m)} \leftarrow \frac{1}{\eta} \exp\left(-\frac{1}{\lambda}\left(c^{(m)} - \beta\right)\right)$
13 **for** $i \leftarrow t$ **to** $t+T-1$ **do**
14 $\mathbf{a}_i \mathrel{+}= \sum_{m=1}^{M} w^{(m)} \epsilon_i^{(m)}$
15 $\mathbf{a}^* \leftarrow \mathbf{a}_t$
16 $\bar{\mathbf{a}} \leftarrow (\mathbf{a}_{t+1}, \ldots, \mathbf{a}_{t+T-2}, \mathbf{a}_{\text{new}})$
17 SendToActuators($\mathbf{a}^*$) // Execute first planned control signal

trajectories. Finally, the feedback-control loop can run until the system task is completed, e.g. a robot can run the cycle for a certain number of timesteps or until reaching a target position.

2.2 Supervised Learning

First of all, *machine learning (ML)* consists of building algorithms that learn some behaviour based on a collection of examples of some phenomenon. ML has countless important applications, including robotics, speech recognition, computer vision, and more. Within the field of ML, there is supervised learning, which starts with gathering data from a population of objects. Objects selected from the population are called *samples*, and each sample presented to a system returns some output associated with the sample. This makes *data* a collection of pairs (input, output). The inputs or input features can be, for example, email messages, pictures, or sensor measurements. The outputs are usually real numbers or labels (e.g. spam, cat, dog, mouse, and so on) (Burkov 2019). The learning problem can be either supervised if the true output is known or unsupervised if it is not. The purpose of learning (or training) is to use these outputs of the samples to build an output predictor or estimator. To formalise, the training data comes in pairs of input-output observations $(\mathbf{x}, y)$:

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subseteq \mathbb{R}^d \times \mathcal{C} \,, \tag{2.9}$$

where $\mathbf{x} \in \mathbb{R}^d$ is an input instance from a d-dimensional input space and $y \in \mathcal{C}$ its output. $\mathcal{C}$ is the target or output space. There are multiple scenarios for the output space, including binary classification, multi-class classification, and regression as in Table 2.1. In regression, the goal is to learn an unknown function that relates the input $\mathbf{x}$ to the output and predict its value over some domain. In classification, the goal is to learn to identify the class or category y of input data. This work uses supervised learning to improve the performance of a robotic system for both regression and binary classification.

The input space can contain discrete or continuous elements. For example, if it were a diagnosis dataset of patients in a hospital, the input can consist of d features $\mathbf{x}_i = [x_{i1}, x_{i2}, \cdots, x_{id}]$, where $x_{i1} \in 0, 1$ refers to patient i's gender, x_{i2} his height i, x_{i3} his age, and so on. The output

Learning type	Output
Binary classification	$\mathcal{C} = \{0, 1\}$ or $\mathcal{C} = \{-1, +1\}$
Multi-class classification	$\mathcal{C} = \{1, 2, \cdots, c_n\}$ where $c_n \geq 2$
Regression	$\mathcal{C} = \mathbb{R}$

TABLE 2.1. Types of learning in ML

y_i would be then 0 or 1, depending on whether the person needs to be diagnosed or not. The dimensionality of such input space is highly relevant since the problem becomes more complex as it increases.

2.2.1 Learning Model

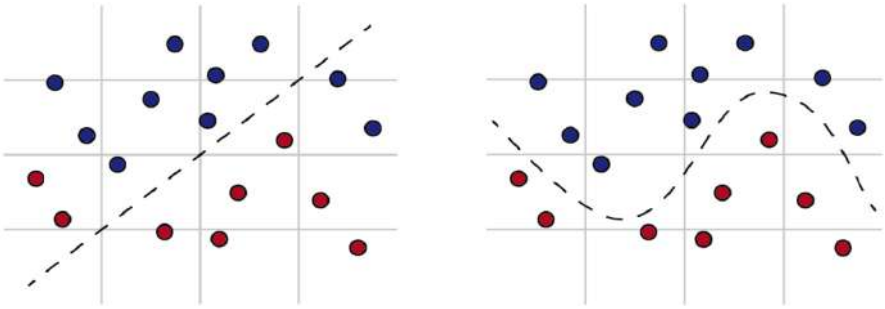
Supervised learning is not only about making predictions from data, but more formally, it is the construction of a *predictive model* $h : \mathbb{R}^d \to \mathcal{C}$ that maps the input to the output based on the collected dataset $\mathcal{D}$. In this work, the predictive model h used is either a classification model or a regression model that maps inputs to outputs as follows:

$$y_i \approx h(\mathbf{x}_i) \text{ for all } (\mathbf{x}_i, y_i) \in \mathcal{D} . \tag{2.10}$$

A predictive model for regression is usually modelled by also assuming inherent additive statistical noise known as *prediction noise* ν_i:

$$y_i = h(\mathbf{x}_i) + \nu_i , \tag{2.11}$$

where $\nu_1, \ldots, \nu_n$ are tipically assumed to be independent and normally distributed $\nu_i \sim \mathcal{N}(0, \sigma_\nu^2)$ for $i = 1, \ldots n$. Notice that here, the variance σ_ν^2 is considered constant throughout the domain of the function, which is known as the homoscedasticity assumption. Then, there is a learning step, also known as predictive model training. We can consider that there is a set of suitable predictive models, and we want to find the best $h(\cdot)$ for the dataset $\mathcal{D}$. The way to quantify best is by introducing the concept of a loss function. A loss function measures how wrong the predictive model h is. A simple loss function for regression problems is the squared loss $\ell(\mathbf{x}, y, h) = (h(\mathbf{x}) - y)^2$ that compares two real-valued inputs. Since there are many instances,



(a) (b)

FIGURE 2.6. Decision boundaries. (a) The linearly separable case where a linear decision boundary can be found. (b) Non-linearly separable case where a polynomial function can separate both classes.

a good measure is the average of all possible losses, which is known as the empirical expected loss

$$L(h, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(\mathbf{x}_i, y_i, h\right) \ . \tag{2.12}$$

Other similar losses are different in how they give penalties to mistakes if $|h(\mathbf{x}_i) - y_i|$ is large. For example, the absolute loss $\ell(\mathbf{x}, y, h) = |h\left(\mathbf{x}\right) - y|$ gives fewer mistake penalties than the squared loss where mistake penalties are magnified. Regarding classification, the loss measures the difference between a true output and a predicted output. A logistic loss is usually used for binary classification problems, while a cross-entropy loss can be used for more general cases such as multi-class classification. Some examples of loss functions are shown in Table 2.2.

Loss type	Usage	Loss function $\ell(\mathbf{x}, y, h)$
Squared loss	Regression	$\left(h\left(\mathbf{x}\right) - y\right)^2$
Absolute loss	Regression	$\left\|h\left(\mathbf{x}\right) - y\right\|$
Logistic loss	Classification	$\log\left(1 + e^{-h(\mathbf{x}_i)y_i}\right)$
Cross-entropy loss	Classification	$y_i \log h(\mathbf{x}_i) + (1 - y_i) \log(1 - h(\mathbf{x}_i))$

TABLE 2.2. Loss functions for binary classification

2.2.2 Predictive Model Optimisation and Complexity

The predictive model has to fit the input data $\mathbf{x}$ to the output, and it can be any function. For example, a straight line $h(x) = 9x - 7$ or a polynomial $h(x) = 5x^3 - 31x^2 + 3x$. That brings the concept of predictive model *complexity*, which refers to the number of parameters or variables included in a given predictive model, as well as whether the model is linear or non-linear. The polynomial is more complex than the straight line. The predictive model can be generalised as

$$h_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{w}^\top \mathbf{x}_i + b \ , \tag{2.13}$$

where a vector of predictive model parameters $\mathbf{w}$ is introduced, and b is a bias parameter that allows the model to fit patterns that do not pass through the origin. b is usually included in $\mathbf{w}$ as an extended parameter vector. In binary classification, $h_{\mathbf{w}}$ is usually a hyperplane called decision boundary that tries to separate the dataset into two groups as in Figure 2.6a. Then, the learning problem consists of estimating optimal predictive model parameters $\mathbf{w}^*$. Finally, the predictive model optimisation problem corresponds to the empirical expected loss minimisation:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \underbrace{l\left(\mathbf{x}_i, y_i, h_{\mathbf{w}}\right)}_{\text{Loss function}} \tag{2.14}$$

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} L(h_{\mathbf{w}}, \mathcal{D}) \ . \tag{2.15}$$

In ML, predictive model training corresponds to finding $\mathbf{w}^*$. After training, the predictive model $h_{\mathbf{w}}$ produces some generalisation error when inferring labels for unseen data. Such error can be decomposed into bias, variance, and irreducible error as

$$\text{Generalization error} \ = \ \text{bias}^{\,2} + \ \text{variance} \ + \ \text{irreducible error} \ , \tag{2.16}$$

where *bias* denotes an average error of the predictive model across different possible training datasets (C. M. Bishop 2006). As the term suggests, it also refers to how the predictive model can be biased to a particular solution that is one of many. Bias can be problematic in robotics since a robot has to learn and act with less data while the set of possible predictive models is vast. An example of a biased predictive model is shown in Figure 2.6b where the true decision boundary is substantially non-linear, so no matter how many training observations we are given,

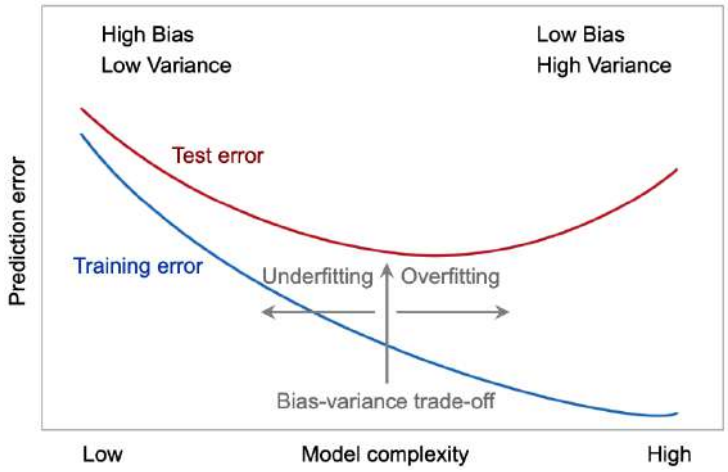


FIGURE 2.7. Bias and variance trade-off, and the behavior of training error (blue curve) and test error (red curve) as the predictive model complexity increases. The training and test errors correspond to the errors obtained by evaluating training and test data with the predictive model respectively.

it will not be possible to produce accurate output estimates using a linear predictive model. In other words, a linear predictive model results in high bias in this example. Meanwhile, in Figure 2.6a, the true decision boundary is very close to linear, and then a linear predictive model is low-biased. Generally, less complex predictive models result in less bias. Secondly, *variance* refers to how much the predictive model $h_{\mathbf{w}}$ sensitivity to small variations in the training dataset (Geron 2019), which means that if a predictive model has high variance, then small changes in the training data can result in large changes in the predictive model. In general, less complex predictive models have higher variance. For example, for the predictive model in Figure 2.6b, the decision boundary would vary more as the polynomial degree increased than if it were decreased until being linear. The irreducible error is related to the inherent noise ν, which can be due to missing variables or limited training data, and it cannot be removed with any predictive model.

The relationship between bias and variance is shown in Figure 2.7. A predictive model has lower bias and higher variance as its complexity increases. Bias and variance are strictly related to overfitting and underfitting. While overfitting is when the predictive model fits exactly the training data, underfitting is the opposite. The objective of training a predictive model is to have

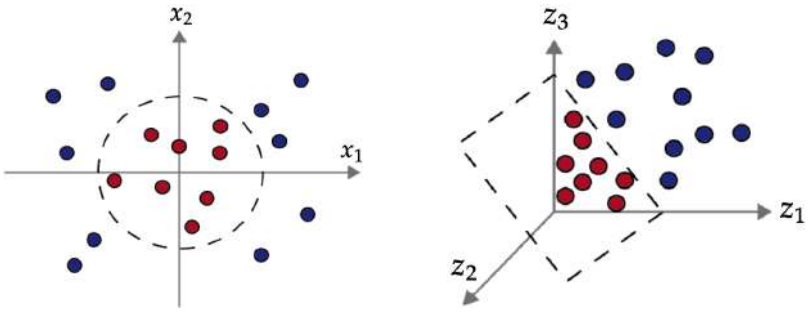
both no bias and no variance or at least low bias and low variance. Selecting the complexity of a predictive model corresponds to a well-known problem called *bias-variance trade-off*.

2.2.3 Feature Mapping

When having to model non-linear decision boundaries, a concept that comes up often is feature mapping. A *feature map* is a transformation for turning low-dimensional input into high-dimensional $\Phi : \mathbf{x} \in \mathbb{R}^d \rightarrow \Phi(\mathbf{x}) \in \mathcal{H}$ where $\mathcal{H}$ can be an m-dimensional space where $m >> d$. Higher dimensions capture more non-linear interactions between the features, and it leads to more predictive model complexity. The example below would capture interactions by performing non-linear operations between input values.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \rightarrow \Phi(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1 x_2 \\ \vdots \\ x_{d-1} x_d \\ \vdots \\ x_1 \dots x_d \end{bmatrix}. \tag{2.17}$$

A transformation can be, for example, $[x_1, x_2] \mapsto [z_1, z_2, z_3] := \left[x_1^2, \sqrt{2} x_1 x_2, x_2^2\right]$, which corresponds to $\Phi : R^2 \rightarrow R^3$ and is shown in Figure 2.8. However, the problem with using these transformations is that features may live in very high dimensional space, possibly infinite. Also, some learning algorithms make use of the inner product, dot product in a Euclidean Space, such as the large-margin classifier known as support vector machine (SVM) (Mohri et al. 2018). Determining the hyperplane decision boundary requires multiple inner product computations in high-dimensional spaces, which makes $\Phi\left(\mathbf{x}_i\right)^T \Phi\left(\mathbf{x}_j\right)$ become expensive to evaluate. However, usual implementations of SVM do not depend on the dimension d of the feature space but only on the margin and size n of the training data by applying what is known



(a) (b)

FIGURE 2.8. The effect of considering a feature map in a two-dimensional space. (a) The data is non-linearly separable. (b) A feature mapping is able to find a hyperplane decision boundary but in a higher dimensional space.

as *kernel trick* (Schlkopf et al. 2018). The concept of a kernel k is usually introduced as a function corresponding to the similarity measure between input features:

$$\begin{aligned} k\left(\mathbf{x}_i, \mathbf{x}_j\right) &= \Phi\left(\mathbf{x}_i\right)^T \Phi\left(\mathbf{x}_j\right) \\ k &: \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R} \, . \end{aligned} \tag{2.18}$$

Kernel-based learning algorithms make use of kernel functions that can even be constructed without knowing the feature mapping Φ associated with them as long as it is possible to prove that such a mapping exists.

2.2.4 Non-linear Classification with Neural Networks

A supervised learning algorithm that makes use of feature transformations is an artificial neural network (Montavon et al. 2011). *Artificial neural networks (ANNs)* were first suggested in the 40's (McCulloch and Pitts 1943). ANNs are inspired by biological neural networks and the way neurons in the brain function together to understand inputs from human senses. An ANN consists of neuron units and connections between them. The connections have weights that function as predictive model parameters that adjust to fit the data during learning. Neurons are usually divided into several groups, which are called layers, such as in a *multilayer perceptron (MLP)*, which is one of the most common ANN. Figure 2.9 shows the structure of an MLP. It considers only one output for the binary classification case, and it is a feed-forward ANN

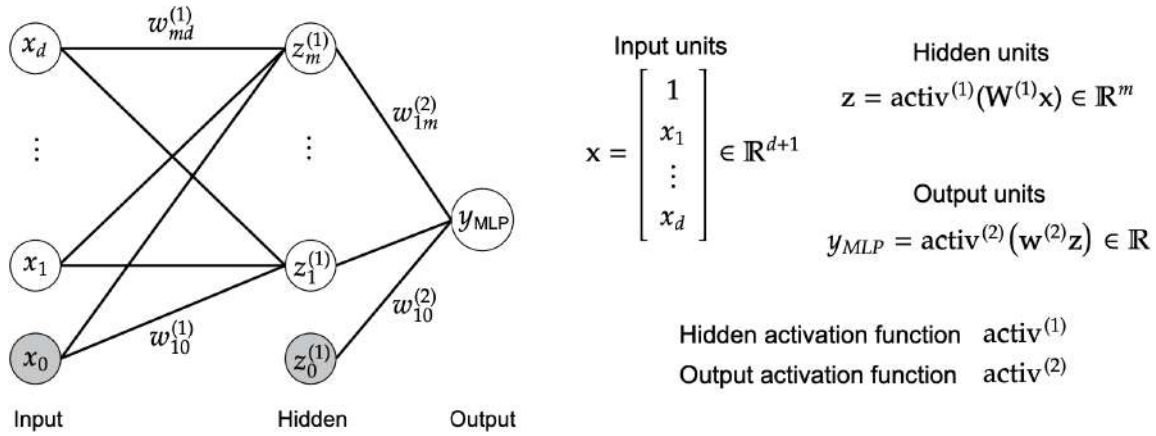


FIGURE 2.9. Fully-connected artificial neural network with a single output for binary classification and one hidden layer.

in which information passes from the input units to the output unit through the hidden units without forming cycles.

The computation flows from left to right. The nodes are units or neurons. The first left-most layer is called the input layer, the last right-most layer is the output layer, and any other layer is referred to as hidden layer. The dimensionalities of the input and output layers are determined by the predictive model $h_{\mathbf{W}^{(l)},\ldots,\mathbf{W}^{(1)}} : \mathbb{R}^d \to \mathcal{C}$, which corresponds to d input units and one output unit for the case of binary classification (Goodfellow et al. 2016).

Each hidden layer can have different numbers of hidden units. The number of hidden layers, the numbers of hidden units, and the ANN hyperparameters are to be defined a priori. Note that in the diagram, there are shaded bias units that have the purpose of fitting patterns that do not pass through the origin as in the linear predictive model from Equation 2.13. Also, each non-input layer has the following property: every node in a layer is connected to every node in the previous layer. A layer with such a property is called dense or fully connected. Each edge in the graph has an associated weight as follows:

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{00}^{(1)} & w_{01}^{(1)} & \cdots & w_{0m}^{(1)} \\ w_{10}^{(1)} & \ddots & & \vdots \\ \vdots & & & \\ w_{d0}^{(1)} & \cdots & & w_{dm}^{(1)} \end{bmatrix} \quad \ldots \quad \mathbf{W}^{(l)} = \begin{bmatrix} w_{00}^{(l)} & w_{01}^{(l)} & \cdots & w_{0e}^{(l)} \\ w_{10}^{(l)} & \ddots & & \vdots \\ \vdots & & & \\ w_{p0}^{(l)} & \cdots & & w_{pe}^{(l)} \end{bmatrix}, \tag{2.19}$$

where $\mathbf{W}^{(j)}$ is a matrix containing the weights of the edges connecting to layer j, and each matrix contains the weight of each edge, e.g. $w_{md}^{(1)}$ is the weight of the edge connecting the d-th unit from the input layer to the m-th unit of the first hidden layer. Each weight represents the importance of a corresponding feature since each node computes a weighted sum of its inputs, e.g. $\mathbf{W}^{(1)}\mathbf{x}$. An ANN applies a non-linear function called activation function $\mathrm{activ}(\cdot)$. For example, $\mathbf{z}^{(1)} = \mathrm{activ}^{(1)}(\mathbf{W}^{(1)}\mathbf{x})$ would be a vector of values of the first hidden layer units. Since the output of each layer is passed as input to the next layer, the predictive model corresponding to the ANN can be written as a composition of transformations:

$$\begin{aligned} h_{\mathbf{W}^{(l)},\ldots,\mathbf{W}^{(1)}}(\mathbf{x}) &= \mathrm{activ}^{(l)}\left(\mathbf{W}^{(l)}\,\mathrm{activ}^{(l-1)}\left(\ldots\,\mathrm{activ}^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right)\right) \\ &= \Phi_l\left(\Phi_{l-1}\left(\ldots\,\Phi_1\left(\mathbf{x}\right)\right)\right) \\ &= \Phi_l \circ \Phi_{l-1} \circ \ldots \circ \Phi_1(\mathbf{x})\;, \end{aligned} \tag{2.20}$$

where the repeated composition of non-linear functions makes the structure a *deep NN*, and it is what gives deep neural networks their remarkable expressive power. The neural network can be as deep as having l layers where l is the number of transformations Φ_j for $j = \{l, \ldots, 1\}$. If only a few hidden layers are used, then it is called *shallow NN or wide NN*. It can be proven that there is no function that can be learned with a deep NN that cannot be learned with a shallow NN (Poggio et al. 2017). However, under the same number of resources, deep neural networks can implement functions with higher complexity than shallow ones (Bianchini and Scarselli 2014) since one also has to define the number of hidden units, which could be exponential. The layer-dependent learning allowed deep NNs to solve more and more complex tasks in areas such as computer vision, speech recognition, and signal processing. In general, the number of hidden layers has to be defined according to the task. The activation function is usually an element-wise function that has to be non-linear since many layers of linear activation functions will only represent linear functions (Goodfellow et al. 2016). Some activation functions are shown in Table 2.3.

Since the weights are the predictive model parameters, they can be collectively denoted as $\overline{\mathbf{W}} = \left\{\mathbf{W}^{(l)}, \mathbf{W}^{(l-1)}, \ldots, \mathbf{W}^{(1)}\right\}$. Then the empirical expected loss minimisation corresponding to

an ANN can be written as follows:

$$\overline{\mathbf{W}}^* = \underset{\overline{\mathbf{W}}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} l\left(\mathbf{x}_i, y_i, h_{\overline{\mathbf{W}}}\right) \tag{2.21}$$

$$\overline{\mathbf{W}}^* = \underset{\overline{\mathbf{W}}}{\operatorname{argmin}} L(h_{\overline{\mathbf{W}}}, \mathcal{D}) \ . \tag{2.22}$$

For classification, the usual output activation function $\text{activ}^{(l)}$ is a logistic activation, while for regression, it is a linear activation (Goodfellow et al. 2016).

The exploration of neural networks for non-linear classification sets a foundation for their application in a complex control system discussed in "Adaptive Model Predictive Control by Learning Classifiers" (Chapter 5). This subsequent chapter will highlight how neural network classifiers are essential in optimising stochastic MPC, particularly in handling complex, non-linear relationships and improving model adaptability in varied scenarios.

Activation type	Equation
Linear	$\text{activ}(x) = x$
Binary step	$\text{activ}(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$
Logistic	$\text{activ}(x) = \frac{1}{1+\exp(-x)}$
TanH	$\text{activ}(x) = \tanh(x)$
Rectified linear unit (ReLU)	$\text{activ}(x) = \max(0, x)$

TABLE 2.3. Examples of activation functions

2.2.5 Optimising Predictive Models

In order to train a predictive model, there are traditional methods such as gradient descent, which is a gradient search for function minimisation. More details about optimisation methods will be seen in Section 2.3. In supervised learning, the commonly used optimisation methods are mainly gradient-based, where the empirical expected loss is the objective function (C. M. Bishop 2006). Then, the empirical expected loss minimisation problem from Equation 2.15 can be solved as long as the derivative of the empirical expected loss function with respect to the

Algorithm 3: Mini-Batch Gradient Search

input : l – Loss function
b – Batch size
γ – Learning rate
output : $\mathbf{w}^*$ – Optimal predictive model parameters

while *stopping condition not met* **do**
  $j_1, \ldots, j_b \leftarrow$ random indices between 1 and n
  $\nabla_{\mathbf{w}} L(h_{\mathbf{w}}, \mathcal{D}) \leftarrow \nabla_{\mathbf{w}} \frac{1}{b} \sum_{i=1}^{b} l\left(\mathbf{x}_{j_i}, y_{j_i}, h_{\mathbf{w}}\right)$ // Find direction to move to
  $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \gamma \nabla_{\mathbf{w}} L(h_{\mathbf{w}}, \mathcal{D})$ // Move to the next possibly optimal parameters

parameters can be computed. Then, new possibly optimal solutions can be found by calculating

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \gamma \nabla_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} l\left(\mathbf{x}_i, y_i, h_{\mathbf{w}}\right) \ , \tag{2.23}$$

which is commonly known as *batch gradient descent* since all the n data points from the training set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ are utilised. γ is a step length hyperparameter known as *learning rate* that determines how fast to move towards an optimal solution. Meanwhile, a less computationally expensive method consists of using subsets of the training dataset at each iteration as described in Algorithm 3, which, because of the random component, is known as a type of stochastic gradient descent.

An ANN can be trained the same way by optimising $\overline{\mathbf{W}}$ instead of $\mathbf{w}$, but to determine the search direction, instead of the gradient, a chain of derivatives is computed by using the chain rule since there can be many layers. Then, minimising the empirical expected loss is known as *back-propagation* since to calculate the derivative, a chain of derivatives is computed from the last layer to the first layer. Alternative optimisation methods for supervised learning include Adagrad (Duchi et al. 2011) and *adaptive moment estimation (Adam)* (Kingma and Ba 2015), which automatically adapt the learning rate γ to the predictive model parameters.

2.3 Optimisation

In general terms, an optimisation algorithm compares solutions to a problem over multiple iterations until a satisfactory solution is found. Such a problem is usually modelled as a mathematical function that receives problem variables as parameters. Moreover, the analytical form of the function to be optimised is often unknown and acts like a black box, which leads to *black-box optimisation*. This concept finds a parallel in the domain of supervised learning and optimising predictive models where the optimisation of complex, often non-explicit objective functions (such as loss functions) is crucial for adjusting predictive model parameters to minimise the error.

This thesis considers black-box optimisation problems since robotic tasks are too complex to have an analytical form. Ideally, the objective is to perform *global optimisation*, which refers to locating the *global minimum (or maximum)* of a function over a given set of solutions, where multiple locally optimal solutions might be present. The issues that come up when searching for a *global optimum* in a black-box function are usually high dimensionality and the topology of the search space, which makes global optimisation computationally demanding, and even so, global optimisation algorithms do not guarantee a global optimum. However, finding a solution close to an optimum within a short time is usually sufficient in real-time applications, such as robotics. The challenges in this domain are akin to those encountered in MPC hyperparameter optimisation, a key focus in the contributing chapters.

Consider the following black-box minimisation problem

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathcal{X}}{\operatorname{argmin}}\, g(\mathbf{x}) \,, \tag{2.24}$$

where $g : \mathbb{R}^d \to \mathbb{R}$ is a noise-free objective function that is continuous on the feasible domain $\mathcal{X}$. A maximum can be obtained by minimising $-g(\mathbf{x})$. We use the term optimum to mean either a maximum or a minimum. For instance, a global optimum is either a global maximum or a global minimum. A global optimiser is any point $\mathbf{x}^*$ that satisfies the optimisation problem. It is assumed that $\mathcal{X} \subseteq \mathbb{R}^d$ is continuous and high-dimensional. The *search space* is a set or domain of feasible solutions through which the optimisation algorithm searches, which in this

Algorithm 4: Gradient Search for Local Optimisation

input : g – Function to optimise
$\mathbf{x}_t$ – Initial solution
γ – Learning rate
output : $\mathbf{x}^*$ – Optimal solution in the search space

```
while stopping condition not met do
    Compute ∇_x g        // Find a direction to move to
    x_{t+1} ← x_t − γ∇_x g        // Move to the next possibly optimal solution
```

case would be the objective function domain. Such search space increases exponentially as the dimension d grows (Weise 2009, June). A *local maximum* is at $\mathbf{x}^*$ if $g(\mathbf{x}) \leq g(\mathbf{x}^*)$ for all $\mathbf{x}$ in some neighborhood of $\mathbf{x}^*$, and a *local minimum* at $\mathbf{x}^*$ if $g(\mathbf{x}) \geq g(\mathbf{x}^*)$ for all $\mathbf{x}$ in some neighborhood of $\mathbf{x}^*$. If the objective function has many local optima in the search space, then the objective function is *multimodal*.

2.3.1 Gradient-based Optimisation

A classic optimisation algorithm is gradient-descent for continuous differentiable functions. The objective function g is *continuous* if it does not break at any point c in its domain. Formally, it is said to be continuous if $g(c)$ is defined, and $\lim_{x \to c} g(x) = g(c)$ from both the left and from the right. The function is *differentiable* if its derivative exists at any point in its domain.

Suppose $\mathbf{x}_t$ is a point sampled from $\mathcal{X}$. The objective function value $g(\mathbf{x}_t)$ can be improved by applying several iterations of any local search method. A *local search method* comes from the idea of moving between configurations by performing local moves as in the gradient search for continuous functions shown in Algorithm 4, which belongs to a set of methods known as *gradient-based optimisation*. $\mathbf{x}_t \in \mathbb{R}$ is a point in the t-th iteration of the local search method, $\nabla_{\mathbf{x}} g$ is the gradient with respect to $\mathbf{x}$ corresponding to the search direction, and here γ also determines how fast to move towards an optimal solution. The concept of a learning rate, γ, is a critical hyperparameter in training neural networks, affecting the convergence rate and the quality of the trained model.

Finally, the stopping condition can be simply a maximum number of iterations or some convergence condition such as $\mathbf{x}_{t+1} - \mathbf{x}_t < \delta$, where δ is some convergence threshold. All in all,

Algorithm 5: Multi-Start Gradient Search

input : g – Function to optimise
γ – Learning rate
m – Number of starts
w $\mathcal{X}$ – Search space
output : $\mathbf{x}^*$ – Optimal solution in the search space

```
for i ← 1 to m do
    Sample x_t with uniform distribution on 𝒳
    while stopping condition not met do
        Compute ∇_x g
        x_{t+1} ← x_t − γ∇_x g
```

gradient search can be considered *deterministic local optimisation* since it can be done when information like exact function values and derivatives are available. It is referred to as "local" since it does not require to provide rigorous theoretical guarantees that the optimal solution is indeed global. If derivatives are unavailable as in black-box optimisation problems, *numerical differentiation* is used to approximate the derivatives empirically. Another method commonly used in deep learning is *automatic differentiation*, which decomposes the derivative into simpler operations, and it is aimed at computationally intensive tasks.

2.3.2 Stochastic Optimisation

Stochastic optimisation is a class of algorithms that employ some degree of randomness when selecting the search direction to find an optimal solution. This stochastic nature is pivotal in Bayesian optimisation, a central theme in the contributing chapters. Bayesian optimisation, which is detailed in Section 2.5, effectively uses randomness to efficiently explore and exploit the search space. The application of randomness in determining the search direction varies, including approaches such as pure random search and multi-start strategies. The latter consists of starting from multiple initial points in the search space as in the multi-start gradient search from Algorithm 5. *Multi-start methods* characteristically sample starting points which in some sense cover the search space assuming all of the best local minima can be detected (Salhi et al. 2000).

Algorithm 6: Multi Level Single Linkage

input : g – Function to optimise
γ, ζ – Sample reduction hyperparameters
$\mathcal{X}$ – Search space
output : $\mathbf{x}^*$ – Optimal solution in the search space

1 $\mathcal{X}^* \leftarrow \varnothing$
2 $k \leftarrow 0$
3 **while** *stopping condition not met* **do**
4 $\quad k \leftarrow k + 1$
5 $\quad$ Sample $\mathbf{x}_1, \ldots, \mathbf{x}_n$ with uniform distribution on $\mathcal{X}$
6 $\quad \mathcal{X}_r \leftarrow \text{ReducedSample}(x_1, \ldots, x_n; \gamma)$
7 $\quad r_k \leftarrow \frac{1}{\sqrt{\pi}} \left[\text{GammaFunction}\left(1 + \frac{d}{2}\right) m(\mathcal{X}) \frac{\zeta \log(kn)}{kn}\right]^{1/d}$
8 $\quad$ **foreach** $\mathbf{x} \in \mathcal{X}_r$ **do**
9 $\quad\quad$ **if** $\mathbf{x} \notin \mathcal{X}^*$ **and** $\nexists \mathbf{x}_j \in \mathcal{X}_r : ||\mathbf{x}_j - \mathbf{x}|| \leq r_k$ **and** $g(\mathbf{x}_j) < g(\mathbf{x})$ **then**
10 $\quad\quad\quad \mathbf{x}^* \leftarrow \text{LocalSearch}(\mathbf{x})$
11 $\quad\quad\quad \mathcal{X}^* \leftarrow \mathcal{X}^* \cup \{\mathbf{x}^*\}$
12 **return** $\mathbf{x}^* \leftarrow \text{BestLocalOptimum}(\mathcal{X}^*)$

A multi-start method used for global optimisation in robotics is *multi level single-linkage (MLSL)* (Rinnooy Kan and Timmer 1987), which is detailed in Algorithm 6. It is used to tune optimisation hyperparameters for the experiments in Section 4.6 and Section 5.8. MLSL enables the exploration of the whole search space through random sampling and the use of a local optimisation method. It consists of two phases: a global and a local search. In the *global phase*, random points are sampled from a probabilistic distribution on $\mathcal{X}$. In the *local phase*, selected points from the global phase are used as starting points for local searches. In line 5, n data points are sampled with uniform distribution on $\mathcal{X}$. Line 6 obtains a reduced sample $\mathcal{X}_r$ that consists of the kn best points in the sample according to $0 < \gamma \leq 1$. Then, line 7 computes a critical distance, which is based on clustering. Higher r_k leads to few local optimisations, thus increasing the risk of missing the global optimum, and lower r_k does the opposite. The critical distance r_k is reduced at each iteration. The critical distance calculation makes use of the gamma function GammaFunction, a Lebesgue measure of the search space $m(\mathcal{X})$, the π constant, and a hyperparameter $\zeta > 0$ that also regulates the reduced sample size. The reduction hyperparameters are defined as constants $\zeta = 2$, $\gamma = 0.3$ in global optimisation libraries such as NLopt[1]. Finally, a local search can be initiated from a reduced sample point if there

[1]NLopt: https://nlopt.readthedocs.io

is no other reduced sample point that has a lower function value within the distance r_k. The algorithm continues repeating the global and local phases until a stopping condition is satisfied.

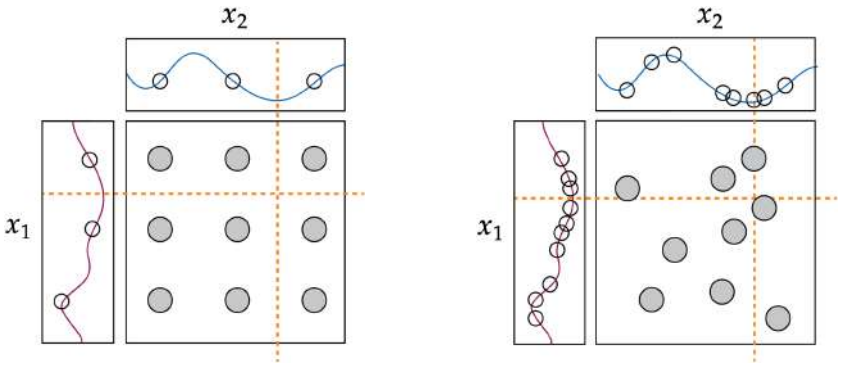
2.3.3 Hyperparameter Optimisation

Since this thesis makes use of optimisation methods for optimising hyperparameters and model parameters, it is essential to briefly know about their difference and what it consists of. First of all, as it can be understood from Section 2.2, model parameters are automatically estimated from data, while model hyperparameters are typically set manually to help estimate the model parameters. Generally, *hyperparameter optimisation or hyperparameter tuning* is the process of searching for optimal hyperparameters defined as $\mathbf{x}$ for an algorithm with a performance function denoted as g. For example, two common trivial hyperparameter optimisation methods are grid search and random search, which consist of going through a set of d sampled *hyperparameter values* $\mathbf{x} = [x_1, \ldots, x_d]$. Manual hyperparameter optimisation is clearly difficult since it requires a lot of data collection and data analysis. A trivial way to search hyperparameters automatically is to go through all possible combinations, which is usually infeasible. *Grid search optimisation* corresponds to exhaustively searching through a search space $\mathcal{X}$ of hyperparameters. Each hyperparameter is usually constrained. For example, the learning rate γ has to be positive $(0, +\infty]$. Grid search extracts a set of equally spaced hyperparameter values $\mathcal{D}_{\mathbf{x}}$ from the search space and evaluates each input.

Algorithm 7: Random Search Optimisation

input : g – function to optimise
$\mathcal{X}$ – Search space
output : $\mathbf{x}^*$ – Optimal solution in the search space

$\mathcal{D}_{\mathbf{x}} \leftarrow$ GetInitialDataset($\mathcal{X}$)
$\mathcal{D} \leftarrow \varnothing$
foreach $\mathbf{x} \in \mathcal{D}_{\mathbf{x}}$ **do**
    $y \leftarrow g(\mathbf{x})$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}, y)\}$
$\mathbf{x}^* \leftarrow$ GetOptimum($\mathcal{D}$)



(a) Grid search (b) Random search

FIGURE 2.10. Search space coverage for grid search and random search.

Another method is a random search method shown in Algorithm 7, where the data points $\mathcal{D}_{\mathbf{x}}$ are uniformly distributed across the search space. Random search is more asymptotically efficient for hyperparameter optimisation than grid search (Bergstra and Bengio 2012). Then, as with multi-start methods, the search space coverage is essential for increasing the chance of getting to an optimal configuration. Random search can cover the search space better as in Figure 2.10 where the hyperparameter space consists of two elements $\mathbf{x} = [x_1, x_2]$, and the dashed line intersection is where the global optimum is. *Random search* involves uniformly sampling hyperparameter combinations through the search space. In random search, the trials are i.i.d., so there is no need to know about previous trials. This means that the same trials will probably not be repeated if starting over with the search for new data points is required.

The literature includes comprehensive comparisons of various hyperparameter optimisation strategies, akin to those used in black-box optimisation, but primarily tailored for enhancing the performance of supervised learning algorithms (Bergstra et al. 2011; Shekhar et al. 2021). Among other hyperparameter optimisation methods that will be useful for MPC hyperparameter optimisation, there is Bayesian optimisation and other surrogate-based optimisation methods, including its variations (Tiao et al. 2021). These methods adeptly handle complex optimisation landscapes, making them highly suitable for MPC hyperparameter tuning, a key focus in this thesis.

2.4 Bayesian Learning

This section introduces Bayesian learning and Bayesian inference for Gaussian processes, which lays the foundation for Bayesian optimisation, which will be introduced in Section 2.5. Bayesian learning is a statistical approach to machine learning that is based on probability theory and the Bayes' theorem, and one of its main advantages is that it provides a framework for dealing with uncertainty and ambiguity in the data (C. M. Bishop 2006). As explained in Section 2.2, the learning goal is to obtain a predictive mathematical model that maps inputs to desired outputs, and this is done via parameter optimisation. In order to introduce uncertainty in the modelling process, it is worth mentioning that there are two different approaches for building a predictive model: discriminative and generative learning.

Discriminative learning consists of directly learning the mapping between inputs $\mathbf{x}$ and outputs y in a regression problem, and it directly learns boundaries in classification, so it discriminates with respect to the inputs. An example of a discriminative method is the neural network described in Section 2.2.4. Meanwhile, *generative learning* consists of building a probabilistic model of the mapping in regression and a probabilistic model for the categories in classification (Jebara 2003). A *probabilistic model* is a mathematical model that uses probability theory to model *uncertainty*. From a statistical point of view, the input $\mathbf{x}$ and the output y are considered *random variables*. Random variables are a way to measure uncertainty. Dealing with uncertainty occurs when we are doubtful about some situation that may happen because of the presence of partial observability and non-determinism, which are both characteristics of real-life robotic environments. In machine learning (ML), the outcomes observed from a random variable are known as *observations*, *realisations* or *samples*. The *support* of a random variable is the set of possible values that the observations can take. A dataset can provide observations that can be used to determine the value of a variable y, for example. If we cannot be completely sure about its value, we can consider it as a random variable and provide a degree of belief known as probability. The *probability density function (PDF)* of a single variable is denoted as $p(y)$, which is a function that maps the support of the variable to probability densities. The PDF completely characterises the distribution of a continuous random variable.

2.4.1 Bayes' Rule

Considering a multivariate random variable $\mathbf{x}$, probabilistic modelling usually consists of learning $p(y|\mathbf{x})$, which describes a conditional probability distribution of the output y given the input $\mathbf{x}$, and such probability distribution $p(y|\mathbf{x})$ can be learned either discriminatively or generatively. In this section, we care about estimating $p(y|\mathbf{x})$ using a generative predictive model. From a Bayesian perspective, the probability distribution $p(y|\mathbf{x})$ is the quantity of interest called the *posterior*, and it can be estimated through the application of Bayes' rule, considering the variable dependency where $p(y)$ is the *prior* that encapsulates the subjective prior knowledge

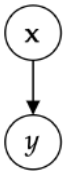


FIGURE 2.11. Variable dependency. The output y depends on the input $\mathbf{x}$.

$$\underbrace{p(y \mid \mathbf{x})}_{\text{posterior}} = \frac{\overbrace{p(\mathbf{x} \mid y)}^{\text{likelihood}}\overbrace{p(y)}^{\text{prior}}}{\underbrace{p(\mathbf{x})}_{\text{evidence}}}, \tag{2.25}$$

of the variable y. The likelihood $p(\mathbf{x} \mid y)$ describes how $\mathbf{x}$ and y are related given y. Once data becomes available, the likelihood allows us to update the posterior. The denominator quantity is the *marginal likelihood or evidence*

$$p(\mathbf{x}) = \int p(\mathbf{x} \mid y)p(\mathbf{x})\mathrm{d}\mathbf{x}\,. \tag{2.26}$$

2.4.2 Bayesian Modelling

Consider a linear regression problem where the objective is to learn an input-output mapping that consists of optimising predictive model parameters $\mathbf{w}$ given a dataset of size n

$$y_i = \mathbf{w}^T\mathbf{x}_i + \nu_i, \qquad i = 1, \cdots, n\,, \tag{2.27}$$

where the prediction noise ν_i is assumed to be Gaussian $\nu_i \sim \mathcal{N}(0, \sigma_\nu^2)$, and $\mathbf{w} \in \mathbb{R}^d$. The Bayesian approach can be applied to parameter learning, which is called *parameter estimation* from a statistical inference perspective, and since it requires defining a probabilistic predictive

model, it can also be called Bayesian learning. By considering variable dependencies, a probabilistic model can be defined by describing the parameters $\mathbf{w}$ as a random variable for the function dependency between $\mathbf{x}$ and y. This distribution of parameters can be interpreted as uncertainty due to insufficient knowledge about the generative process. Then, a probabilistic model can be formulated as a dependency between $\mathbf{w}$ and the training data $\mathcal{D}$, which is also considered a random variable consisting of the data points $(y_i, \mathbf{x}_i)$. To make statements about the interaction of the parameters $\mathbf{w}$ and the observations $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, we consider their joint probability distribution $p(\mathbf{w}|\mathcal{D})$, where the Bayesian approach can be applied. The assumptions formulated via the generative model allow the evaluation of the likelihood $\mathrm{p}(\mathcal{D} \mid \mathbf{w}) = \mathrm{p}(\mathbf{X}, \mathbf{y}|\mathbf{w})$ where $\mathcal{D}$ consists of a vector of input multi-dimensional random variables $\mathbf{X} = [\mathbf{x}_1, \ldots \mathbf{x}_n]$ and a vector of output random variables $\mathbf{y} = [y_1, \ldots y_n]$. The prior $p(\mathbf{w})$ is a distribution over all possible parameters $\mathbf{w}$, which usually assumes a normal distribution. Then, the posterior is formulated as

$$\mathrm{p}(\mathbf{w} \mid \mathcal{D}) = \frac{\mathrm{p}(\mathcal{D} \mid \mathbf{w})\mathrm{p}(\mathbf{w})}{\mathrm{p}(\mathcal{D})}\,, \tag{2.28}$$

where $\mathrm{p}(\mathcal{D}) = \int \mathrm{p}(\mathcal{D} \mid \mathbf{w})\mathrm{p}(\mathbf{w})\mathrm{d}\mathbf{w}$ can be calculated because it is a combination of the likelihood and prior, although it is usually intractable because it has to integrate over all possible parameters $\mathbf{w}$. When all terms on the right-hand side are known, the posterior can be evaluated. *Bayesian inference* in the context of ML can be performed by computing the posterior $\mathrm{p}(\mathbf{w} \mid \mathcal{D})$. However, finding the posterior usually entails a combination of modelling assumptions and observations. If enough assumptions are considered, then the posterior can have a closed-form solution. If there is no closed-form solution, approximation techniques such as sampling or *variational inference* (Zhang et al. 2019) can be used.

Finally, with the posterior found, a *predictive posterior distribution* for new inputs can be defined as $p(y_* \mid \boldsymbol{x}_*, \mathcal{D})$ as shown in Equation 2.29. $\mathbf{x}_*$ and $\mathcal{D}$ are both shaded, indicating that both variables are observed, and the others are latent (or unobserved). Such posterior integral is intractable if no assumptions are made. A parametric method known as *Bayesian linear regression* takes advantage of the convenience of normal distribution operations and solves the inference problem analytically. Bayesian linear regression assumes a Gaussian prior $p(\mathbf{w}|\mathcal{D})$,

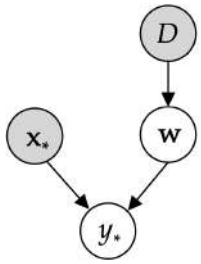


FIGURE 2.12. Variable dependency for computing a predictive posterior.

$$p\left(y_* \mid \mathbf{x}_*, \mathcal{D}\right) = \int_{\mathbf{w}} p\left(y_* \mid \mathbf{x}_*, \mathbf{w}\right) p(\mathbf{w} \mid \mathcal{D}) d\mathbf{w} \,. \tag{2.29}$$

and since the noise is Gaussian, a Gaussian likelihood $p(y_i|\mathbf{x}_i, \mathbf{w}) = \mathcal{N}(\mathbf{w}^T\mathbf{x}_i, \sigma_\nu^2)$ is also assumed. The integral becomes tractable with such assumptions. More details on Bayesian linear regression can be found in C. M. Bishop (2006).

2.4.3 Gaussian Processes

Before introducing Bayesian inference for Gaussian processes, it is helpful to elaborate on the Gaussian uncertainty. As it was seen in previous sections, a common probability distribution introduced in modelling is the noise variable $\nu \sim \mathcal{N}(0, \sigma_\nu^2)$ where σ_ν^2 is known as *noise variance*. To understand what a Gaussian process does, it is helpful to give more details on the Gaussian distribution. First and foremost, a *Gaussian or normal distribution* is widely used across the many algorithms for ML, including BO, to represent uncertainty. Its PDF is parameterised by a mean $\mu \in \mathbb{R}$ and a variance $\sigma^2 \in \mathbb{R}_{>0}$. We say that if a one-dimensional random variable x follows a Gaussian distribution $x \sim \mathcal{N}(\mu, \sigma^2)$, its PDF is denoted as $p\left(x; \mu, \sigma^2\right) = \mathcal{N}(\mu, \sigma^2)$ and defined as follows:

$$p\left(x; \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \,, \tag{2.30}$$

where its support is the real numbers: $\operatorname{supp}(x) = \mathbb{R}$. The *mean* μ of the random variable is a measure of centrality that reflects the midpoint probability distribution in a single measure. It can also be described as the weighted average of its possible values i, which turns out to be the *expectation or expected value* $\mathbb{E}[x] = \sum_{i \in \operatorname{supp}(x)} i\, p\left(i\right)$ if x is a discrete random variable. For a Gaussian-distributed random variable, the support is continuous. In that case, the expectation is

defined as $\mathbb{E}[x] = \int_{-\infty}^{\infty} xp(x)dx$. Meanwhile, the *variance* is a measure of how spread out the distribution is. The squared root of the variance is called standard deviation σ, and it is usually used to measure the spread from the mean. For example, for Gaussian-distributed variables, it is known that 95% of its observations lie within two standard deviations from the mean (Blitzstein and Hwang 2019).

A *multivariate random variable* can be expressed in vector notation as an ordered set of d random variables $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ with a joint PDF $p(\mathbf{x}) = p(x_1, x_2, \ldots, x_d)$. If $\mathbf{x}$ follows a multivariate Gaussian distribution, its PDF is defined as follows:

$$p(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right) , \tag{2.31}$$

where the distribution is parameterized by a mean $\boldsymbol{\mu} \in \mathbb{R}^d$ and a *positive-definite* covariance matrix $\boldsymbol{\Sigma} \in \mathcal{S}_{++}^d$, which means that

$$\mathcal{S}_{++}^d = \left\{\mathbf{M} \in \mathbb{R}^{d \times d} : \mathbf{M} = \mathbf{M}^T \text{ and } \mathbf{z}^T\mathbf{M}\mathbf{z} > 0 \text{ for all } \mathbf{z} \in \mathbb{R}^d \text{ such that } \mathbf{z} \neq \mathbf{0}\right\} . \tag{2.32}$$

The mean $\boldsymbol{\mu}$ is now multi-dimensional, representing a vector containing the expected values of the elements $\mathbb{E}[\mathbf{x}] = (\mathbb{E}[x_1], \ldots, \mathbb{E}[x_d]) = (\mu_1, \ldots, \mu_d)$, and the covariance matrix

$$\boldsymbol{\Sigma} = \begin{bmatrix} \Sigma_{11} & \cdots & \Sigma_{1d} \\ \vdots & \ddots & \vdots \\ \Sigma_{d1} & \cdots & \Sigma_{dd} \end{bmatrix} \tag{2.33}$$

is a square matrix that gives the covariance for every pair (x_i, x_j), denoted as Σ_{ij}. Every element in $\mathbf{x}$ is normally distributed according to $x_i \sim N(\mu_i, \Sigma_{ii})$, and every linear combination of x_i is normally distributed (Kaiser 2021). The *covariance* is a single-number summary of the joint distribution of two random variables. It can also be seen as a measure of their tendency relative to their means (Blitzstein and Hwang 2019).

This thesis mostly uses multivariate random variables to represent predictive model variables and parameters in order to build probabilistic models. Probabilistic models incorporate probability distributions into the model.

Gaussian Process Regression

A Gaussian process (GP) is a stochastic process that generalises multivariate Gaussian distributions over finite dimensional vectors to infinite dimensionality (Sammut and Webb 2011). As a *stochastic or random process*, a GP involves a state that changes in a random way over time, which is why it can be seen as a sequence of random variables $\{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots\}$. The subindex is often interpreted as time, and each element $\mathbf{x}_\tau$ is known as the *state of the process* at time τ. For example, $\mathbf{x}_{\tau+1} = \mathbf{x}_\tau + \mathcal{N}(0, 1)$ is an example of a stochastic process where the change in the state depends on the previous one and a random number. The time can also be continuous, but the main focus in this thesis is discrete-time stochastic processes. A GP is used for modelling distributions over functions (Rasmussen and Williams. 2006). Therefore, it can be used for regression. For a given training dataset, there are potentially infinitely many functions that can fit. With Bayesian linear regression, a posterior over the parameters can be derived, which leads to several potentially optimal linear functions. Meanwhile, a GP can directly represent a posterior over the functions.

An advantage of *GP regression* is that it is a non-parametric way of approximating nonlinear functions. The previously seen predictive learning models were *parametric models* since they assumed a mapping function with a fixed number of parameters $\mathbf{w}$ as in $y \approx \mathbf{w}^T\mathbf{x}$. A GP is a *non-parametric model* that does not define an explicit parameterised formula for function modelling.

First of all, a probability distribution over functions consists of a set $\mathcal{H}$ of all possible function mappings $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \rightarrow \mathbb{R}$. A function can be compactly represented as an n-dimensional vector $\mathbf{g} = [g(\mathbf{x}_1), \ldots g(\mathbf{x}_n)]$. Then, we can specify a Gaussian PDF for each function in $\mathcal{H}$ as

$$\mathbf{g} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{K}) \,, \tag{2.34}$$

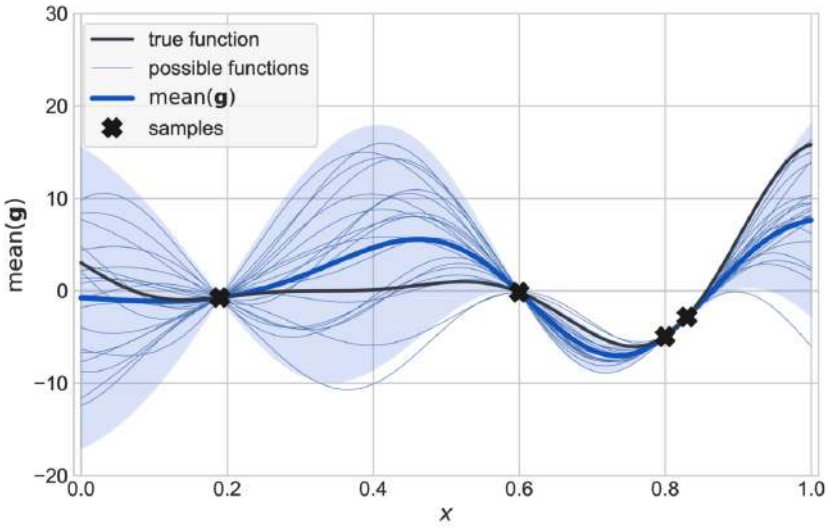


FIGURE 2.13. Gaussian process example with four initial observations and a set of potential function fits. In regions with fewer observations, there is greater uncertainty in the predicted functions.

which is used as a prior distribution for $\mathbf{g}$ since a GP performs Bayesian inference to approximate a posterior. The covariance matrix K is now the matrix

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & \cdots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}, \tag{2.35}$$

which contains *kernel or covariance functions* $k(\cdot, \cdot)$ that represent similarity measures between input features as explained in Equation 2.18. The kernel functions have to be defined beforehand. Examples of kernel functions will be described in Section 2.4.4.

The next quantity of interest is the likelihood. As with Bayesian linear regression, a Gaussian likelihood $p(y_i|\mathbf{x}_i, \mathbf{w}) = \mathcal{N}(\mathbf{w}^T\mathbf{x}_i, \sigma_\nu^2)$ where $g(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i$ is also assumed. Therefore, the evaluation y_i at the point $\mathbf{x}_i$ satisfies the *GP likelihood*

$$\begin{gathered} y_i \mid g(\mathbf{x}_i) \sim \mathcal{N}\left(g(\mathbf{x}_i), \sigma_\nu^2\right) \quad i = 1, \ldots, n \\ \text{vectorized as} \\ \mathbf{y} \mid \mathbf{g} \sim \mathcal{N}\left(\mathbf{g}, \boldsymbol{\sigma}_\nu^2 \mathbf{I}\right) . \end{gathered} \tag{2.36}$$

Finally, a posterior distribution can be derived by applying a Gaussian distribution property known as conditioning (Murphy 2012). Since the output vector $\mathbf{y}$ and the function $\mathbf{g}$ are jointly normal, their joint distribution is given by

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{g} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu} \end{bmatrix}, \begin{bmatrix} \mathbf{K} + \sigma_\nu^2 I & \mathbf{K} \\ \mathbf{K} & \mathbf{K} \end{bmatrix} \right) , \tag{2.37}$$

where the posterior distribution for the function $\mathbf{g}$ is computed as follow

$$\begin{aligned} \mathbf{g} \mid \mathbf{y} &\sim \mathcal{N}\left(\text{mean}(\mathbf{g}), \text{cov}(\mathbf{g})\right) \\ \text{mean}(\mathbf{g}) &= \boldsymbol{\mu} + \mathbf{K}^T \mathbf{K}^{-1}(\mathbf{y} - \boldsymbol{\mu}) \\ \text{cov}(\mathbf{g}) &= \mathbf{K} - \mathbf{K}(\mathbf{K} + \sigma_\nu^2 \mathbf{I})^{-1}\mathbf{K} \, . \end{aligned} \tag{2.38}$$

As a means of demonstrating a Gaussian fit, Figure 2.13 shows a Gaussian process example with four initial observations. The shaded region represents the confidence interval, which is determined by the Gaussian process variance and is typically set as twice the standard deviation, serving to quantify the uncertainty around the true function. Alongside the confidence interval, you can observe possible functions generated from the posterior distribution, showcasing a range of potential function fits. When aiming to approximate the true function, it is desirable for the mean, denoted as $\text{mean}(\mathbf{g})$, to closely align with the true function.

Since the regression model can infer new functions given some dataset, we can define train and test data. During training, the GP learns by using a dataset $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$. The test data will be used as prior knowledge composed of also inputs $\mathbf{X}_* = [\mathbf{x}_{*1}, \ldots \mathbf{x}_{*m}]$ and outputs $\mathbf{g}_* = [g_{*1}, \ldots, g_{*m}]$. Then, the goal is to formulate and infer a posterior distribution that can be defined as

$$\begin{aligned} & p\left(\mathbf{g}_* \mid \mathbf{X}_*, \mathcal{D}\right) \\ & p\left(\mathbf{g}_* \mid \mathbf{X}_*, \mathbf{X}, \mathbf{y}\right) \, . \end{aligned} \tag{2.39}$$

As well as with Equation 2.37, the joint distribution of the output vector $\mathbf{y}$ and the test function vector $\mathbf{g}_*$ is given by the *GP prior*

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{g}_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{bmatrix}, \begin{bmatrix} \mathbf{K}_{\mathbf{XX}} + \sigma^2 I & \mathbf{K}_{\mathbf{X}\mathbf{X}_*} \\ \mathbf{K}_{\mathbf{X}_*\mathbf{X}} & \mathbf{K}_{\mathbf{X}_*\mathbf{X}_*} \end{bmatrix} \right) , \tag{2.40}$$

and using the conditioning property again, we can get the *GP predictive posterior*

$$\mathbf{g}_* \mid \mathbf{X}_*, \mathbf{X}, \mathbf{y} \sim \mathcal{N}\left(\text{mean}(\mathbf{g}_*), \text{cov}(\mathbf{g}_*)\right) \tag{2.41a}$$

$$\text{mean}(\mathbf{g}_*) = \boldsymbol{\mu}_* + \mathbf{K}_{\mathbf{X}_*\mathbf{X}}[\mathbf{K}_{\mathbf{XX}} + \sigma_\nu^2 \mathbf{I}]^{-1}(\mathbf{y} - \boldsymbol{\mu}) \tag{2.41b}$$

$$\text{cov}(\mathbf{g}_*) = \mathbf{K}_{\mathbf{X}_*\mathbf{X}_*} - \mathbf{K}_{\mathbf{X}_*\mathbf{X}}(\mathbf{K}_{\mathbf{XX}} + \sigma_\nu^2 \mathbf{I})^{-1}\mathbf{K}_{\mathbf{X}\mathbf{X}_*} \tag{2.41c}$$

where

$$\begin{aligned} &\mathbf{K}_{XX} \in \mathbf{R}^{n\times n} \text{ such that } (\mathbf{K}_{XX})_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) \\ &\mathbf{K}_{XX_*} \in \mathbf{R}^{n\times m} \text{ such that } (\mathbf{K}_{XX_*})_{ij} = k(\mathbf{x}_i, \mathbf{x}_{*j}) \\ &\mathbf{K}_{X_*X} \in \mathbf{R}^{m\times n} \text{ such that } (\mathbf{K}_{X_*X})_{ij} = k(\mathbf{x}_{*i}, \mathbf{x}_j) \\ &\mathbf{K}_{X_*X_*} \in \mathbf{R}^{m\times m} \text{ such that } (\mathbf{K}_{X_*X_*})_{ij} = k(\mathbf{x}_{*i}, \mathbf{x}_{*j}) \ . \end{aligned} \tag{2.42}$$

2.4.4 Model Selection

Approximating the true function is a hard problem because the possibilities are essentially unlimited. A GP can approximate the true function by fitting a normal distribution. Since the GP is used for black-box optimisation problems, there is often no prior knowledge for selecting the function structure. The GP is usually assumed to start with a prior function $\mathbf{0} = (0_1, \ldots 0_n)$ as $\boldsymbol{\mu}$, which does not produce any loss of generality in case the mean is unknown (Rasmussen and Williams. 2006). The GP posterior can be computed analytically, but as in most ML algorithms, there are hyperparameters to optimise. The GP formulation allows us to consider sets of function structures determined according to the *GP hyperparameters*. Model selection is usually referred to as *GP training*, and it corresponds to finding the set of hyperparameters that can best describe the function to approximate (Mchutchon and Rasmussen 2011). Those hyperparameters can be selected by analysing some sample dataset if expert knowledge is available, but they are usually fit to sample dataset $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ collected a priori.

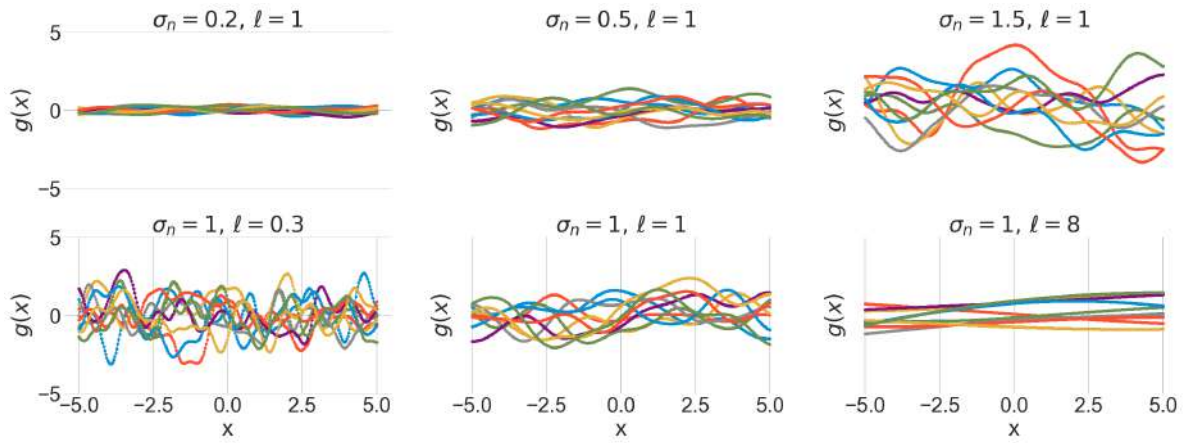


FIGURE 2.14. Typical components and variables in control system design.

The first and most common GP hyperparameter is the noise variance σ_ν from the noise variable $\nu \sim \mathcal{N}(0, \sigma_\nu^2)$. The noise variance σ_ν specifies how much noise is expected to be present in the data. Other hyperparameters depend on the choice of kernel function for the GP covariance matrix. There are several kernel functions $k(\cdot, \cdot)$ in the literature, starting with the *squared exponential*, also known as radial basis function or Gaussian kernel. The figure Figure 2.14 shows sampled functions $f \sim \mathcal{N}(\mathbf{0}, \mathbf{K})$ where $f : \mathbb{R} \to \mathbb{R}$ and $(\mathbf{K})_{ij} = k_{\text{SE}}(x_i, x_j)$. The squared exponential kernel is defined as follows:

$$k_{\text{SE}}(x_i, x_j) = \sigma_n^2 \exp\left(-\frac{(x_i - x_j)^2}{2\ell^2}\right) , \tag{2.43}$$

where its hyperparameters are a signal variance σ_n^2 and a lengthscale ℓ. The signal variance σ_n^2 can increase or decrease the variance of the function approximation without affecting the function smoothness. The lengthscale ℓ can regulate the smoothness of the function approximation. Small ℓ values would make the function approximation less smooth, characterising a function that changes quickly. It also determines how far it can reliably extrapolate from the training data. Large ℓ values produce a smoother function approximation, and that would make sure that training points that are far away remain strongly correlated. The squared exponential kernel function from Equation 2.43 can be extended to multi-dimensional input vectors by replacing the squared difference by $||\mathbf{x}_i - \mathbf{x}_j||^2 = (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j)$.

The squared exponential kernel described can only be used if a unique lengthscale is used for all the input features. *Automatic relevance determination (ARD)* is an extension to such a kernel. The ARD kernel from Equation 2.44 is able to implicitly determine the relevance of each feature by using a lengthscale ℓ_m per input feature $\mathbf{x}_m$ where $m \in \{1, \ldots, d\}$. A large lengthscale ℓ_m makes the covariance almost independent of the corresponding input feature $\mathbf{x}_m$, which makes the feature less relevant for inference (Rasmussen and Williams. 2006). All features do not have the same relevance in a real dataset, so the optimal solution should consist of only strong relevant features.

$$k_{\text{SE-ARD}}(x_i, x_j) = \sigma_n^2 \exp\left(-\frac{1}{2}\sum_{m=1}^{d}\frac{\left(\mathbf{x}_{im} - \mathbf{x}_{jm}\right)^2}{\ell_m^2}\right) . \tag{2.44}$$

The kernel functions can be classified into stationary and non-stationary. They are known as stationary if they depend on the distance separating the two input vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, but not on the input themselves (Duvenaud 2014). They are non-stationary if they do not depend on the distance. Table 2.4 shows a list of other kernel functions commonly used for a GP, their mathematical expressions, and their corresponding hyperparameters. The linear and polynomial kernel functions are examples that do not depend on the distance. To compute kernel functions for multi-dimensional inputs, a diagonal matrix $\mathbf{L} \in \mathbb{R}^{d\times d}$ is used where $\mathbf{L}_{(i,i)} = \frac{1}{\ell_i^2}$. The covariance matrix is often defined as a diagonal matrix. Also, the distance is defined as

$$\text{dist}(\mathbf{x}_\text{i}, \mathbf{x}_\text{j}) = (\mathbf{x}_i - \mathbf{x}_j)^\top \mathbf{L} (\mathbf{x}_i - \mathbf{x}_j) . \tag{2.45}$$

We can define a set of GP hyperparameters as $\Omega = \{\sigma_\nu, \sigma_n, \ell_1, \ell_2, \ldots, \ell_d\}$. Then, there are two main approaches to training the GP using a sample dataset: the first one is a purely Bayesian approach where a prior is placed on the hyperparameters $p(\Omega)$, and then the posterior $p(\Omega \mid \mathbf{X}, \mathbf{y})$ is inferred using Bayesian inference algorithms (Svensson et al. 2015). The other one is the empirical Bayes approach, where the optimal hyperparameters are optimised by maximising the marginal likelihood (Ganjali 2016), which is the one discussed in this section. The marginal

Kernel name	Expression $k(\mathbf{x}_i, \mathbf{x}_j)$	Hyperparameters
Linear	$\sigma_n^2\left(\sigma_0^2 + \mathbf{x}_i^T\mathbf{L}\mathbf{x}_j\right)$	$\Omega = \{\sigma_n, \sigma_0, \boldsymbol{\ell}\}$
Matern	$\sigma_n^2(1+\sqrt{3\,\mathrm{dist}(\mathbf{x}_i,\mathbf{x}_j)})\exp(-\sqrt{3\,\mathrm{dist}(\mathbf{x}_i,\mathbf{x}_j)})$	$\Omega = \{\sigma_n, \boldsymbol{\ell}\}$
Polynomial	$\sigma_n^2\left(\sigma_0^2 + \mathbf{x}_i^T\mathbf{L}\mathbf{x}_j\right)^p$	$\Omega = \{\sigma_n, \sigma_0, \boldsymbol{\ell}, p\}$
Squared Exponential	$\sigma_n^2 \exp\left(-\frac{\mathrm{dist}(\mathbf{x}_i,\mathbf{x}_j)}{2}\right)$	$\Omega = \{\sigma_n, \boldsymbol{\ell}\}$

TABLE 2.4. A list of typical kernel functions taken from (Marchant Matus 2015). All kernel functions shown, but the linear and polynomial kernel functions depend on the distance between input vectors, p is the degree of the polynomial, and $\boldsymbol{\ell} = \{\ell_1, \ell_2, \ldots, \ell_d\}$.

likelihood of the GP is given by

$$p(\mathbf{y} \mid \mathbf{X}) = \int p(\mathbf{y} \mid \mathbf{g}, \mathbf{X}) p(\mathbf{g} \mid \mathbf{X}) d\mathbf{g} \,, \tag{2.46}$$

where $p(\mathbf{y} \mid \mathbf{g}, \mathbf{X}) = \mathcal{N}(\mathbf{g}, \sigma_\nu^2\mathbf{I})$ is the likelihood and $p(\mathbf{g} \mid \mathbf{X}) = \mathcal{N}(\mathbf{0}, \mathbf{K})$ is the prior, considering the observed data $\mathbf{X}$. Then, instead of maximising the marginal likelihood directly, the negative logarithm of the marginal likelihood (Rasmussen and Williams. 2006) is known as *log-marginal likelihood* and is the objective function to be optimised:

$$\begin{aligned} \mathcal{L}(\Omega) &= -\log p(\mathbf{y} \mid \mathbf{X}; \Omega) \\ &= -\frac{1}{2}\mathbf{y}^\top\left(\mathbf{K} + \sigma_\nu^2\mathbf{I}\right)^{-1}\mathbf{y} - \frac{1}{2}\log\left|\mathbf{K} + \sigma_\nu^2\mathbf{I}\right| - \frac{n}{2}\log 2\pi \,, \end{aligned} \tag{2.47}$$

where π is the mathematical constant, the covariance matrix $\mathbf{K}$ is calculated using $\mathbf{X}$ and the hyperparameters Ω. The maximum likelihood estimate is the solution to the problem

$$\Omega^\star \in \arg\min_{\Omega} \mathcal{L}(\Omega) \,, \tag{2.48}$$

which can be calculated using some black-box optimisation approach. The result is a set of suitable hyperparameters for the GP model.

2.5 Bayesian Optimisation

This section describes Bayesian optimisation (BO) as a GP-based and surrogate-based method for solving stochastic optimisation problems. One of the uses of the GP described in Section 2.4.3 is to act as an approximation model that mimics the behaviour of some unknown function since it is used to perform regression, starting from some prior assumptions. A particular method that uses GPs is BO, which has been widely applied in robotics and control to optimise expensive black-box functions (Marco-Valle 2020, July). The way this is done is by sequentially optimising an approximated model of the true function, which corresponds to a class of optimisation methods known as surrogate-based methods.

Sequential-based Model Optimisation

BO is part of a class of sequential optimisation methods known as *sequential model-based optimisation (SMBO)* that consists of using an approximated function model known as *surrogate model* (Jiang et al. 2020) to optimise costly-to-evaluate black-box functions. The general idea of dealing with expensive functions due to resources of some kind is also related to *active learning*, which is optimisation oriented to classification for supervised learning but done sequentially. Also, since a surrogate model is used, BO is also a *surrogate-based optimisation* method. An SMBO method consists of the following main components

- BO sequentially fits a surrogate model denoted as $\mathcal{M}$ to the current set $\mathcal{D}$ of evaluated data points. $\mathcal{M}$ is assumed to be cheaper to evaluate, such as a GP.
- A *utility measure* uses the surrogate model $\mathcal{M}$ to select the next input $\mathbf{x}_t$, which may get better performance. That selection is made with some black-box optimisation algorithm.

As in Algorithm 8, an SMBO algorithm starts with defining the dataset of evaluated points $\mathcal{D}$ can start as empty $\mathcal{D} = \varnothing$, but it also can start with the points already evaluated used to train the GP, for example. In line 3, the surrogate model is fit to the dataset. In line 4, a utility measure is maximised, which gives the next point to evaluate $\mathbf{x_t}$. Then in line 5, the function to optimise

is evaluated at such a point $\mathbf{x_t}$. Finally, the dataset is updated with the new input-output pair $(\mathbf{x}_t, y_t)$. The whole procedure can be done n times, in which n new data points are found.

Algorithm 8: Sequential-based Model Optimisation

input : y – Function to optimise
n – Number of iterations
$\mathcal{M}$ – Surrogate model
$\mathcal{X}$ – Search space
output : $\mathbf{x}^*$ – Optimal solution in the search space

1 $\mathcal{D} \leftarrow$ InitialiseDataset()
2 **for** $t \leftarrow 1$ **to** n **do**
3 FitModel($\mathcal{M}, \mathcal{D}$)
4 $\mathbf{x}_t \leftarrow \text{argmax}_{\mathbf{x}\in\mathcal{X}}$ UtilityMeasure($\mathbf{x}, \mathcal{M}, \mathcal{D}$)
5 $y_t \leftarrow y(\mathbf{x}_t)$
6 $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_t, y_t)\}$
7 $\mathbf{x}^* \leftarrow$ GetOptimum($\mathcal{D}$)

2.5.1 Bayesian Optimisation Formulation

As an optimisation algorithm, BO's objective is to choose points to evaluate in order to locate global optimisers, but with some more advantages, starting with the *robustness to noisy objective function evaluations*. A noisy function corresponds to an objective function g under some external noise ν. The optimisation problem is defined as

$$y = g(\mathbf{x}) + \nu \tag{2.49}$$

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathcal{X}}{\text{argmin}}\, g(\mathbf{x}) \,, \tag{2.50}$$

where $g(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}$, and $\nu \sim \mathcal{N}(0, \sigma_\nu^2)$ as seen before. The BO algorithm is shown in Algorithm 9. It is assumed that a set of random observations was collected (e.g. using the random search seen in Section 2.3.3) to obtain optimal GP hyperparameters Ω. The method starts by initialising a dataset $\mathcal{D}$ that can also be empty in line 2. Then a GP prior is obtained by fitting the GP model in line 3. Both previously evaluated points and new points $\mathbf{x}_*$ are denoted as $\mathbf{x}$ from now on. The method returns the optimal input $\mathbf{x}_t$ if the function y were noiseless. However, it corresponds to the point with the largest GP posterior mean $\mathbf{x}^* = \text{argmax}_{\mathbf{x}\in\mathcal{D}}\, \text{mean}(\mathbf{g}(\mathbf{x}))$ since we can only observe the function g through noisy point-wise observations y (Shahriari et al. 2016).

Algorithm 9: Bayesian Optimisation

input : y – function to optimise
n – Number of iterations
$\mathcal{M}$ – Surrogate model
$\mathcal{X}$ – Search space
α – Acquisition function
Ω – GP hyperparameters
output : $(\mathbf{x}^*)$

$\mathcal{D} \leftarrow$ GetInitialDataset$(\mathcal{X})$
for $t = 1$ **to** n **do**
  Fit a GP model $\mathcal{M}_\Omega$ with the data $\mathcal{D}$
  $\mathbf{x}_t \leftarrow \text{argmax}_{\mathbf{x}\in\mathcal{S}}\, \alpha(\mathbf{x}, \mathcal{M}_\Omega, \mathcal{D})$
  $y_t \leftarrow y(\mathbf{x}_t)$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_t, y_t)\}$
$\mathbf{x}^* \leftarrow$ GetOptimum$(\mathcal{D})$

Evaluating new points $\mathbf{x}$ in a search space $\mathcal{S}$ can be done by using the utility measure α known as *acquisition function*, which is maximised to give the next input $\mathbf{x}_t$, which is not necessarily the optimum so far since BO follows the exploration-exploitation paradigm. In the context of function optimisation, the *exploration-exploitation* paradigm consists of either exploring new unseen function regions or exploiting regions where the optimal is so far (Luke 2013). Hence, there is a trade-off between exploring and exploiting. For example, Figure 2.15 shows a function with homoscedastic noise $\sigma_\nu^2 = 1$. The objective is to minimise the true function in yellow, and this is done by running BO for n iterations. Figure 2.16 shows the first two iterations where the regions with higher variance are observed first since it uses an acquisition function that does more exploration. At the last iteration, BO tends to converge to an optimal region.

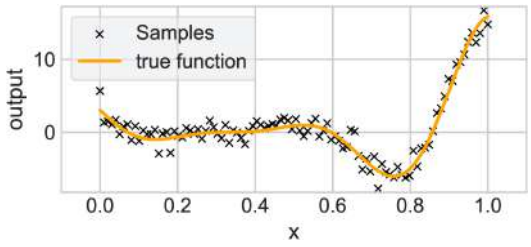


FIGURE 2.15. Noisy Forrester

$$g(x) = (6x-2)^2 \sin(12x-4) + \mathcal{N}\left(0, \sigma_\nu^2\right) \qquad \sigma_\nu^2 = 1\,. \tag{2.51}$$

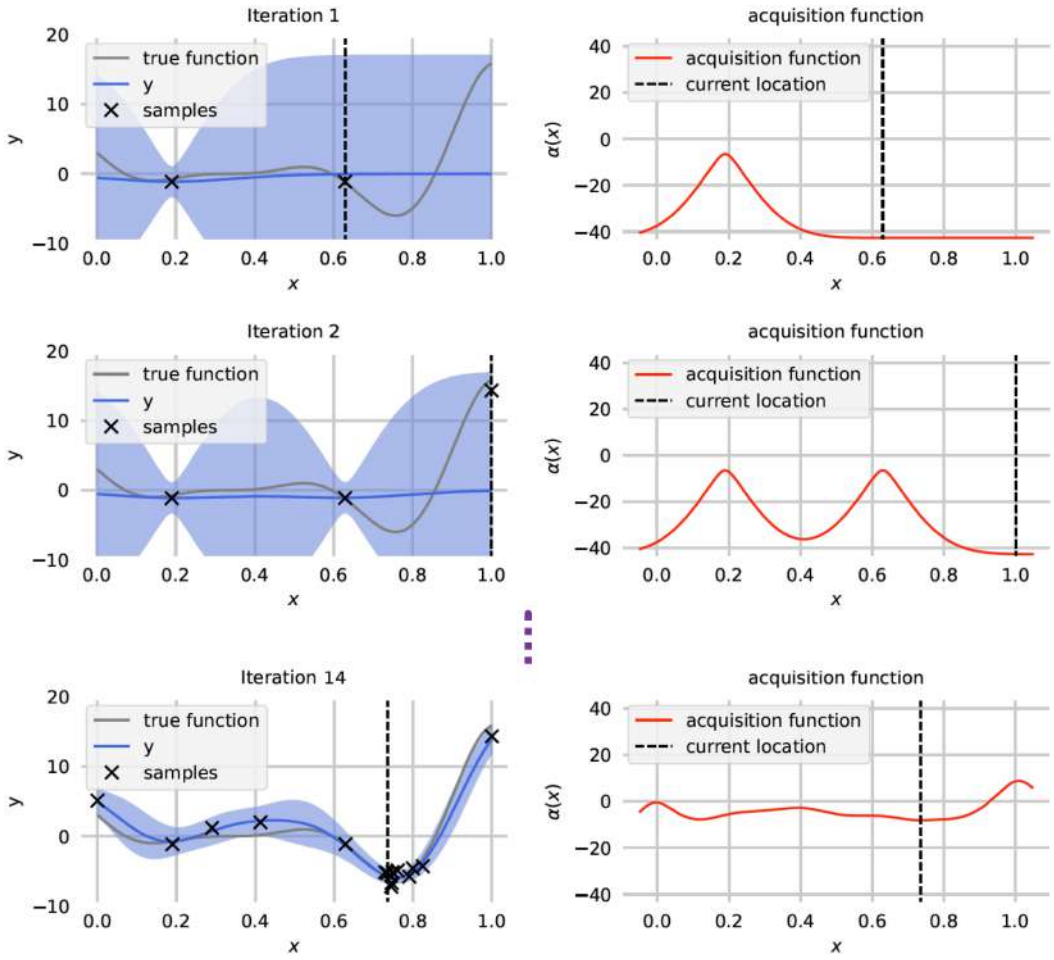


FIGURE 2.16. BO iterations starting with a single initial observation

2.5.2 Acquisition Functions for Bayesian Optimisation

The acquisition function is typically an inexpensive function since it has to be maximised at each iteration. In an ideal situation, it is approximately the true function and minimising it returns a point close to the optimal, but it is done by taking into account previously evaluated data points. Since the function parameter is a single input and not a matrix, given an input $\mathbf{x}$, the mean $\text{mean}(\mathbf{g}(\mathbf{x}))$ and variance $\text{var}(\mathbf{g}(\mathbf{x}))$ are obtained analytically by computing the GP posterior from *Equation* 2.41. The standard deviation is defined as $\text{std}(\mathbf{g}(\mathbf{x})) = \sqrt{\text{var}(\mathbf{g}(\mathbf{x}))}$.

The acquisition functions described next assumed a noiseless setting as BO is commonly described, but when the objective function is noisy, BO considers an approximation $\hat{g} \approx g$ where $\hat{g}(\mathbf{x}^*) = \text{mean}(\mathbf{x}^*)$.

Probability of Improvement

Assuming a noiseless function $y = g(\mathbf{x})$, the function evaluation at the best point so far is denoted as $g(\mathbf{x}^*)$. The acquisition function is able to measure the improvement of a point $\mathbf{x}$ to be evaluated by computing

$$\alpha_{\text{Imp}}(\mathbf{x}) = \max(g(\mathbf{x}^*) - g(\mathbf{x}), 0) \ . \tag{2.52}$$

A *probability of improvement (PI)*, introduced in Kushner (1964), can be expressed as the probability

$$\alpha_{\text{PI}}(\mathbf{x}) = p(\alpha_{\text{Imp}}(x) > 0) \Leftrightarrow p(g(\mathbf{x}) > g(\mathbf{x}^*)) \ . \tag{2.53}$$

Then to express $\alpha_{\text{PI}}(\mathbf{x})$ it in terms of variables already seen and knowing that the function evaluated at a single input is expressed as $g(\mathbf{x}) \sim \mathcal{N}(\text{mean}(\mathbf{g}(\mathbf{x})), \text{var}(\mathbf{g}(\mathbf{x})))$, the probability of improvement is

$$\alpha_{\text{PI}}(\mathbf{x}) = \Phi\left(\frac{\text{mean}(\mathbf{g}(\mathbf{x})) - g(\mathbf{x}^*)}{\text{std}(x)}\right) \ , \tag{2.54}$$

where Φ is the cumulative distribution function of the standard normal distribution. The acquisition function does not balance exploration and exploitation automatically. For that reason, an extra hyperparameter $\delta \geq 0$ is added, which allows more exploitation as it decreases and more exploration as it increases:

$$\alpha_{\text{PI}}(\mathbf{x}) = \Phi\left(\frac{\text{mean}(x) - g(\mathbf{x}^*) - \delta}{\text{std}(x)}\right) \ . \tag{2.55}$$

Expected Improvement

Adding functionality to PI, there is another acquisition function known as *expected improvement (EI)* (Bull 2011) that additionally considers the expected magnitude of the improvement. The expected improvement $\alpha_{\text{EI}}(\mathbf{x}) := \mathbb{E}[\alpha_{\text{Imp}}(\mathbf{x})]$ is then defined as

$$\alpha_{\text{EI}}(\mathbf{x}) = \int_{-\infty}^{\infty} \alpha_{\text{Imp}}(\mathbf{x})\varphi(z)dz \ = \int_{-\infty}^{\infty} \underbrace{\max\left(g(\mathbf{x}^*) - g(\mathbf{x}), 0\right)}_{\alpha_{\text{Imp}}(\mathbf{x})} \varphi(z)dz \ , \tag{2.56}$$

where φ is the PDF of the standard normal distribution $\mathcal{N}(0,1)$. To solve the integral, it breaks into two components: one where $g(\mathbf{x}^*) - g(\mathbf{x})$ is positive and one where it is negative. The point where the switch happens is $z_0 = \frac{g(\mathbf{x}^*) - \text{mean}(\mathbf{g}(\mathbf{x}))}{\text{std}(\mathbf{g}(\mathbf{x}))}$. Then the integral becomes

$$\alpha_{\text{EI}}(\mathbf{x}) = \underbrace{\int_{-\infty}^{z_0} \alpha_{\text{Imp}}(\mathbf{x})\varphi(z)dz}_{\text{Zero since } \alpha_{\text{Imp}}(\mathbf{x})=0} + \int_{z_0}^{\infty} \alpha_{\text{Imp}}(\mathbf{x})\varphi(z)dz \,. \tag{2.57}$$

The resulting expected improvement can be calculated by reducing the expression in terms of $\text{mean}(\mathbf{g}(\mathbf{x}))$ and $\text{std}(\mathbf{g}(\mathbf{x}))$ and doing integration by parts:

$$\begin{aligned}
\alpha_{\text{EI}}(\mathbf{x}) &= \int_{z_0}^{\infty} \max\left(g(\mathbf{x}^*) - g(\mathbf{x}), 0\right) \Phi(z)dz \\
&= \int_{z_0}^{\infty} \left(\text{mean}(\mathbf{g}(\mathbf{x})) + \text{std}(\mathbf{g}(\mathbf{x}))z - g(\mathbf{x}^*)\right) \Phi(z)dz \\
&= \left(\text{mean}(\mathbf{g}(\mathbf{x})) - g(\mathbf{x}^*)\right) \Phi\left(\frac{\text{mean}(\mathbf{g}(\mathbf{x})) - g(\mathbf{x}^*)}{\text{std}(\mathbf{g}(\mathbf{x}))}\right) \\
&\quad + \text{std}(\mathbf{g}(\mathbf{x}))\varphi\left(\frac{\text{mean}(\mathbf{g}(\mathbf{x})) - g(\mathbf{x}^*)}{\text{std}(\mathbf{g}(\mathbf{x}))}\right) \,.
\end{aligned} \tag{2.58}$$

Then $\alpha_{\text{EI}}(x)$ takes high values when $\text{mean}(\mathbf{g}(\mathbf{x})) > g(\mathbf{x}^*)$, which is when the mean value of the GP is high at $\mathbf{x}$. An extra hyperparameter $\delta \geq 0$ can allow more exploitation as it decreases and vice versa:

$$\begin{aligned}
\alpha_{\text{EI}}(\mathbf{x}) &= \left(\text{mean}(\mathbf{g}(\mathbf{x})) - g(\mathbf{x}^*) - \delta\right) \Phi\left(\frac{\text{mean}(\mathbf{g}(\mathbf{x})) - g(\mathbf{x}^*) - \delta}{\text{std}(\mathbf{g}(\mathbf{x}))}\right) \\
&\quad + \text{std}(\mathbf{g}(\mathbf{x}))\varphi\left(\frac{\text{mean}(\mathbf{g}(\mathbf{x})) - g(\mathbf{x}^*) - \delta}{\text{std}(\mathbf{g}(\mathbf{x}))}\right) \,.
\end{aligned} \tag{2.59}$$

Upper Confidence Bound

The acquisition function known as *upper confidence bound (UCB)* was proposed in Cox and John (1992), and it explicitly deals with the exploitation–exploration trade-off with a hyperparameter $\delta \geq 0$ that also allows more exploitation as it decreases and vice versa. In a function minimisation case, it is defined as

$$\alpha_{\text{UCB}}(\mathbf{x}) = \text{mean}(\mathbf{g}(\mathbf{x})) - \delta\,\text{std}(\mathbf{g}(\mathbf{x})) \,. \tag{2.60}$$

The way δ is optimised depends on the task, and it can only be determined via data analysis. Nevertheless, in most cases, a higher δ is usually more advantageous since it makes use of the GP, and more regions with higher uncertainty will be explored. Meanwhile, as $\delta \to 0$, it does more exploitation, which leads to converging into a local minimum.

UCB's ability to explore more thoroughly reduces the risk of missing global optima, a significant advantage over PI and EI that might converge prematurely to local optima. Furthermore, the flexibility offered by the δ hyperparameter in UCB allows more adaptation to the specific characteristics of the MPC parameter space. Finally, in terms of time performance, UCB's computational simplicity, compared to more complex acquisition functions, translates into faster evaluations, beneficial in the context of MPC, where time efficiency is often as critical as achieving optimal performance.

2.5.3 Bayesian Optimisation Alternatives

Despite the non-parametric advantages of Bayesian inference for optimisation, BO is hindered by the GP surrogate model, which has limitations such as a cubic computational cost when training and does not directly handle variable noise structures such as heteroscedasticity. BO extensions are focused on addressing the several issues of BO, and they are mainly restrained by the necessity to ensure analytical tractability of the predictive posterior distributions and typically make strong and oversimplifying assumptions. For example, (Kuindersma et al. 2012) proposed a heteroscedastic BO approach that uses a variational approximation that is expensive to compute.

Tree-Structured Parzen estimator

Some BO alternatives bypass the challenges of analytical tractability in GP-based approaches by selecting points according to a density ratio. One alternative is called *tree-structured Parzen estimator (TPE)*, which was proposed in Bergstra et al. (2011). TPE also assumes a noiseless

function $y = g(\mathbf{x})$. TPE uses a quantile hyperparameter $0 < \gamma < 1$ that provides a threshold

$$\tau = \Phi^{-1}(\gamma) \,, \tag{2.61}$$

for splitting the data into a first group that gave the best scores and a second group containing the rest as in Figure 2.17. Then, the goal is to find the next inputs that are more likely to be in the first group, which will serve as a utility measure. In order to propose a new input $\mathbf{x}$, TPE computes a density ratio between the probability $a(\mathbf{x})$ of being in the first group and the probability $b(\mathbf{x})$ of being in the second group. Unlike BO, TPE models $p(\mathbf{x} \mid y, \mathcal{D})$ instead of $p(y \mid \mathbf{x}, \mathcal{D})$ as Equation 2.62.

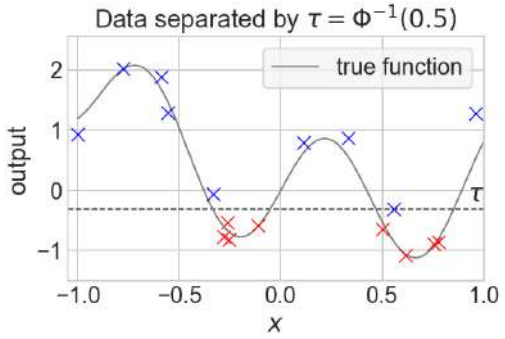
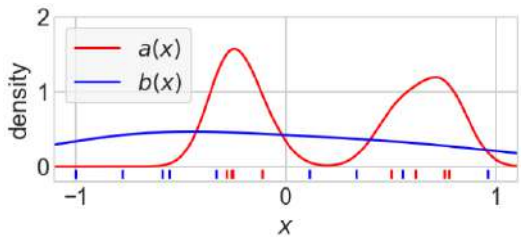
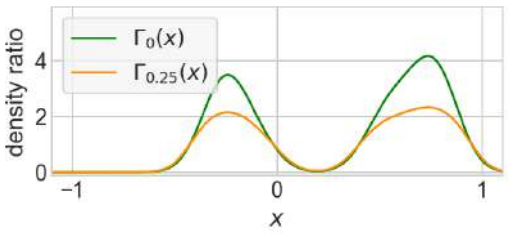


FIGURE 2.17. Observations divided two groups: the red data points where $y < \tau$ and the blue data points where $y \geq \tau$.

$$\begin{aligned} p(\mathbf{x} \mid y < \tau) &= a(\mathbf{x}) \\ p(\mathbf{x} \mid y \geq \tau) &= b(\mathbf{x}) \,. \end{aligned} \tag{2.62}$$



(a) Ordinary density ratio $\Gamma(\mathbf{x})$



(b) γ-relative density ratio $\Gamma\gamma(\mathbf{x})$

FIGURE 2.18. Density ratio example

Algorithm 10: Tree-Structured Parzen Density Estimator

input : y – Function to optimise
n – Number of iterations
$\mathcal{X}$ – Search space
γ – Quantile hyperparameter
output : $\mathbf{x}^*$ – Optimal solution in the search space

$\mathcal{D} \leftarrow \text{InitialiseDataset}()$
for $t \leftarrow 1$ **to** n **do**
  $\tau \leftarrow \text{SplitData}(\mathcal{D}, \gamma)$
  $\Gamma_\gamma \leftarrow \text{GetDensityRatio}(\mathcal{D}, \tau)$
  $\mathbf{x}_t \leftarrow \text{argmax}_{\mathbf{x} \in \mathcal{X}} \Gamma_\gamma(\mathbf{x})$
  $y_t \leftarrow y(\mathbf{x}_t)$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_t, y_t)\}$
$\mathbf{x}^* \leftarrow \text{GetOptimum}(\mathcal{D})$

Then, the utility measure for selecting a new input is a ratio between the probabilities $a(\mathbf{x})$ and $b(\mathbf{x})$:

$$\Gamma_0(\mathbf{x}) = a(\mathbf{x})/b(\mathbf{x})\,, \qquad (2.63) \qquad \Gamma_\gamma(\mathbf{x}) = \frac{a(\mathbf{x})}{\gamma a(\mathbf{x}) + (1-\gamma) b(\mathbf{x})}\,. \qquad (2.64)$$

A relative density ratio parameterised by γ was proven to be proportional to the expected improvement α_{EI} (Bergstra et al. 2011). For example, Figure 2.18b shows that optimal regions are the ones where the density ratio is higher, and the exploration-exploitation trade-off can be regulated by γ. The whole TPE optimisation method is shown in Algorithm 10.

Bayesian Optimisation by Density Ratio Estimation

As well as TPE, another approach that makes use of the density ratio is called *Bayesian optimisation by density ratio estimation (BORE)* (Tiao et al. 2021). Unlike TPE, BORE trains a probabilistic classifier to obtain a utility measure. Based on a reformulation of the α_{EI} acquisition function from Equation 2.58, BORE approximates the γ-relative density ratio to a binary class posterior probability as

$$\Gamma_\gamma(\mathbf{x}) := \gamma^{-1} \Pi(\mathbf{x})\,, \qquad (2.65)$$

where $\Pi(\mathbf{x})$ computes the probability of $\mathbf{x}$ belonging to a positive class $\Pi(\mathbf{x}) = p(z = 1 \mid \mathbf{x})$. The binary label $z \in \{0, 1\}$ introduced here denotes a negative or positive class, and its meaning corresponds to whether the point should be selected or not. Then given a maximisation

objective, BORE sets $z := \mathbb{I}[y \geq \tau]$, indicating whether the corresponding observation y at a point $\mathbf{x}$ is above the γ-th quantile τ of the (empirical) observations distribution, i.e., $\gamma = p(y \geq \tau)$. In the end, computing the acquisition function h from the classical BO method is reduced to classification, as shown in Algorithm 11.

Algorithm 11: Bayesian Optimisation by Density Ratio Estimation

input : y – function to optimise
n – Number of iterations
$\mathcal{M}$ – Surrogate model
$\mathcal{X}$ – Search space
γ – Exploration-exploitation hyperparameter
Π_{classif} – Probabilistic binary classifier to train
output : $(\mathbf{x}^*)$

$\mathcal{D} \leftarrow$ GetInitialDataset$(\mathcal{X})$
for $t = 1$ **to** n **do**
  $\tau \leftarrow$ SplitData$(\mathcal{D}, \gamma)$
  $\Pi \leftarrow$ TrainBinaryClassifier$(\mathcal{D}, \tau, \Pi_{\text{classif}})$
  $\mathbf{x}_t \leftarrow \operatorname{argmax}_{\mathbf{x} \in \mathcal{X}} \Pi(\mathbf{x})$
  $y_t \leftarrow y(\mathbf{x}_t)$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_t, y_t)\}$
$\mathbf{x}^* \leftarrow$ GetOptimum$(\mathcal{D})$

As it can be seen, BORE also follows the steps from SMBO paradigm Algorithm 8 steps where the utility measure is the probability distribution $\Pi(\mathbf{x})$. The optimisation depends on the hyperparameter $\gamma \in (0, 1)$, which influences the exploration-exploitation trade-off. A smaller γ encourages exploitation. Intuitively, it leads to fewer modes and sharper peaks in the acquisition function. Figure 2.19 shows an example of minimising Figure 2.15 where the class probability $\Pi(\mathbf{x})$ is shown at the right, starting with a single observation. The observations are divided into blue and red markers. The optimal region is exploited as more iterations are run.

2.6 Reinforcement Learning

In simple terms, *reinforcement learning (RL)* is learning through reinforcement. *Reinforcement* is the act of strengthening whatever causes the frequency of a given behaviour to increase. That desired behaviour can be interpreted as the desired state of a decision-maker. From a learning perspective, the *decision-maker* learns how to choose particular *actions* $\mathbf{a}$ to achieve

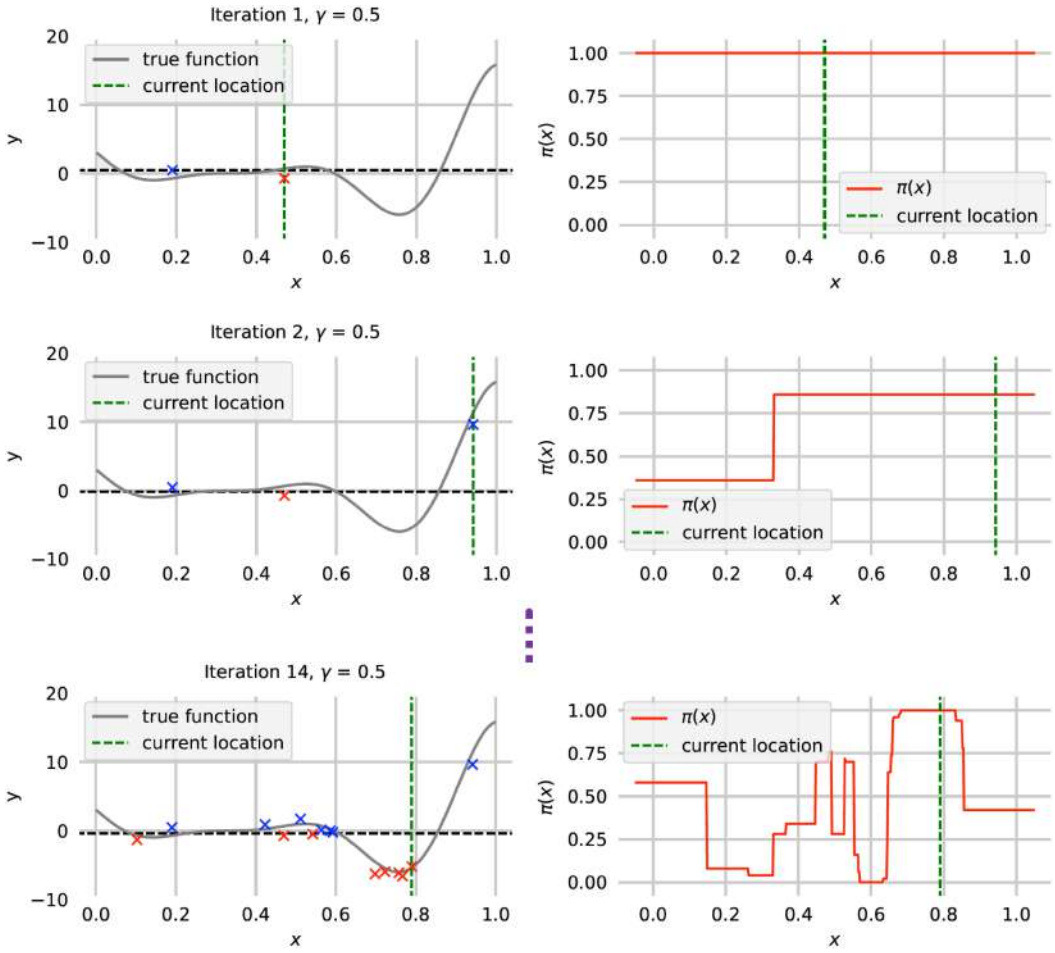


FIGURE 2.19. BORE iterations starting with a single initial observation. The horizontal dashed line represents the threshold τ.

that desired *state* $\mathbf{s}$ by reinforcing optimal actions. In RL, the decision-maker is known as *agent*, which can be a robot, and the thing it interacts with, comprising everything outside the agent, is called the *environment* (R. Sutton and Barto 2018). RL and optimal control researchers have been studying very similar problems for intelligent devices from thermostats to autonomous cars to robotic manipulators (Antsaklis and Rahnama 2018), which is why it has connections with control systems. The controlled system can be a robot, e.g. an autonomous car. Then there is the control signal that regulates the car speed, which is the corrective action $\mathbf{a}$, and the state $\mathbf{s}$ of the system is still its current state defined by its sensor readings. Moreover, as in optimal control, where an action or state has a cost associated, in RL, they have a *reward* associated

that measures the agent's performance. Then, instead of minimising costs, RL problems are usually formulated as maximising rewards.

RL's connection with supervised learning resides in how the output is presented. In supervised learning, there is an output y and a predictive model that is trained based on input features $\mathbf{x} = [x_1, \ldots, x_p]$ from a given observation. That *observation* is analogous to a given state $\mathbf{s}$ of the environment. The difference is that RL does not receive the right output. The robot must learn how to obtain that output by trial and error, and it must do it by performing actions $\mathbf{a}$. Regarding the learning process, in general, RL consists of learning an optimal *policy*, which is a function that maps from states of the environment to actions.

2.6.1 Markov Decision Process

As a stochastic process, a *Markov process (MP)* or Markov chain involves a state $\mathbf{s}$ that changes in a random way over time according to a *Markov property*, which is the assumption that the future depends only on the present and not on the past (R. Sutton and Barto 2018). For example, if the objective is about predicting rainy days, the state space can be $\mathcal{S} = \{\text{raining}, \text{not raining}\}$ and a state $s \in \mathcal{S}$. With the Markov property, raining in the present can lead to rain in the future, which is desired since rainy days come in sequences. The simplest and most common formulation assumes the environment can be modelled as a *discrete-time Markov process* where $t \in 0, 1, 2, \ldots$, the Markov property can be represented as

$$p(\mathbf{s}_{t+1}|\mathbf{s}_t) = p(\mathbf{s}_{t+1}|\mathbf{s}_1, \ldots, \mathbf{s}_t) \ , \tag{2.66}$$

where moving from one state $\mathbf{s}$ to $\mathbf{s}_{t+1}$ is called a *transition*, and the probability of moving to such a state $p(\mathbf{s}_{t+1}|\mathbf{s}_t)$ is called *transition probability* or state-transition probability. Then, achieving a state can have a reward known as *immediate reward* r in order to evaluate the agent's behaviour. That leads to an extension called *Markov reward process (MRP)*. While an MP consists of states and transition probabilities, an MRP consists of states, transition probabilities, and also a reward function $r(\mathbf{s})$.

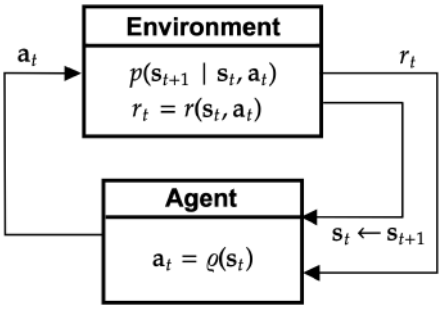


FIGURE 2.20. Agent-environment interaction. The policy ϱ is deterministic.

In an RL environment, the agent makes decisions known as actions $\mathbf{a}$ in order to reach the desired state, and it does it sequentially in time, for which the problem is known as *sequential decision making*. Since it follows the Markov property, such a decision-making process is known as *Markov decision process (MDP)*. The dynamics of the MPD is defined by

$$p(\mathbf{s}_{t+1}, r_{t+1}|\mathbf{s}_t, \mathbf{a}_t) = p(\mathbf{s}_{t+1}, r_{t+1}|\mathbf{s}_1, \mathbf{a}_1, r_1 \ldots, \mathbf{s}_t, \mathbf{a}_t, r_t) \,, \tag{2.67}$$

where at each timestep t, the agent reaches a state $\mathbf{s}_t \in \mathcal{S}$ and executes an action $\mathbf{a}_t \in \mathcal{A}$ according to a policy ϱ. The environment then provides a feedback signal in the form of a reward $r \in \mathbb{R}$. The agent's interaction with the environment is shown in Figure 2.20. Interacting with the environment gives rise to a sequence or *trajectory* from time t_0 to t_f:

$$\left\{\mathbf{s}_{t_0}, \mathbf{a}_{t_0}, r_{t_1}, \mathbf{s}_{t_1}, \mathbf{a}_{t_1}, r_{t_2}, \ldots \mathbf{s}_{t_f-1}, \mathbf{a}_{t_f-1}, r_{t_f}, \mathbf{s}_{t_f}\right\} \,. \tag{2.68}$$

This thesis considers only finite-horizon problems where the main objective is to maximise the cumulative reward known as *return* for a trajectory starting at *timestep* t up to timestep $t + T$. An *episode* is known as the whole set of trials (agent-environment interactions) that return the trajectory. The return for the current episode can also be called *episodic reward* and is computed as $J_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_{t+T}$. The states, actions, and rewards can be sampled as in Figure 2.21.

Moreover, given the MDP framework, the probability of reaching $\mathbf{s}_{t+1}$ can be marginalised, giving the transition probability from Equation 2.69a. The expected reward for r_{t+1} can be

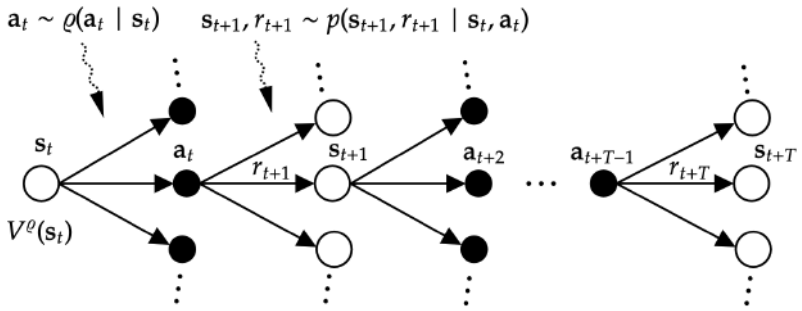


FIGURE 2.21. States and interactions sampled by an MDP. $V^{\varrho}(\mathbf{s}_t)$ is the value at state $\mathbf{s}_t$, and $\varrho(\mathbf{a}_t|\mathbf{s}_t)$ is a stochastic policy.

calculated as follows:

$$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \tag{2.69a}$$

$$r(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}\left[r_{t+1} \mid \mathbf{s}_t, \mathbf{s}_{t+1}, \mathbf{a}_t\right] \ . \tag{2.69b}$$

2.6.2 Policy and Value Function

Two related concepts in RL are the policy and the value function. While the policy helps the robot decide the next action, the value function or state-value function depends on the policy to evaluate states. The *value function* $V(\mathbf{s})$ estimates the goodness of the state $\mathbf{s}$ so as to select between actions. *Value-based methods* are those algorithms that only learn the value function for policy optimisation. Formally, a *policy* is any state-action mapping function $\varrho : \mathcal{S} \rightarrow \mathcal{A}$ that can be either a *deterministic policy* $\mathbf{a}_t = \varrho(\mathbf{s}_t)$ or a *stochastic policy* $\varrho(\mathbf{a}_t|\mathbf{s}_t) = p(\mathbf{a}_t|\mathbf{s}_t)$. A policy can be a set of rules that the robot can use to determine which action $\mathbf{a}_t$ to take from a current state $\mathbf{s}_t$. In the case of the stochastic policy, there is a probability distribution for actions to take, so actions can be sampled from it $\mathbf{a}_t \sim \varrho(\mathbf{a}_t|\mathbf{s}_t)$ as in Figure 2.21. Future states are evaluated to compute the value function of a policy ϱ, which is defined as

$$V^{\varrho}(\mathbf{s}_t) = \mathbb{E}_{\varrho}\left[J_t \mid \mathbf{s}_t\right] = \mathbb{E}_{\varrho}\left[\sum_{i=0}^{T-1} \zeta^i r_{t+i+1} \mid \mathbf{s}_t\right] \ , \tag{2.70}$$

where $0 \leq \zeta \leq 1$ determines the rate at which future rewards are discounted relative to immediate outcomes. Optimising the value function $V^{\varrho^*}(\mathbf{s}) = \max_{\varrho} V^{\varrho}(\mathbf{s})$ corresponds to finding the best possible cumulative reward that can be obtained with respect to ϱ, which also leads to finding the optimal policy $\varrho^* = \operatorname{argmax}_{\varrho} V^{\varrho}(\mathbf{s})$. Value optimisation is done by evaluating a recursive structure known as *Bellman's equation* efficiently. Assuming a *discrete state space* $\mathcal{S}$, the Bellman's equation for the value function is defined as

$$\begin{aligned} V^{\varrho}(\mathbf{s}) &= \mathbb{E}_{\mathbf{a}\sim\varrho(\mathbf{s})} \sum_{\mathbf{s}'\in\mathcal{S}} p(\mathbf{s}'|\mathbf{s},\mathbf{a}) \left[r(\mathbf{s},\mathbf{a}) + \zeta V^{\varrho}(\mathbf{s}')\right] \\ V^{\varrho}(\mathbf{s}) &= \sum_{\mathbf{s}'\in\mathcal{S}} p(\mathbf{s}'|\mathbf{s},\mathbf{a}) \left[r(\mathbf{s},\mathbf{a}) + \zeta V^{\varrho}(\mathbf{s}')\right] \qquad \text{for a fixed policy .} \end{aligned} \tag{2.71}$$

Optimisation problems that satisfy Bellman's equation can be solved with *dynamic programming (DP)*, which is an approach that transforms a complex problem into a sequence of subproblems so as to avoid repeated computations (Nayak 2021). Given the Bellman's equation, the optimal value function and the optimal policy are defined as

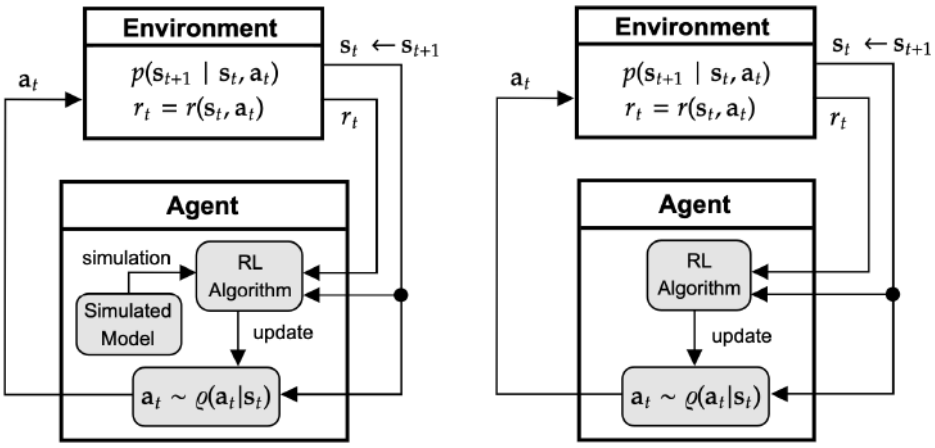
$$V^*(\mathbf{s}) = \max_{\mathbf{a}\in\mathcal{A}} \left\{ \sum_{\mathbf{s}'\in\mathcal{S}} p(\mathbf{s}'|\mathbf{s},\mathbf{a}) \left[r(\mathbf{s},\mathbf{a}) + \zeta V^*(\mathbf{s}')\right] \right\} \tag{2.72a}$$

$$\varrho^*(\mathbf{s}) = \operatorname*{argmax}_{\mathbf{a}} V^*(\mathbf{s}) \,, \tag{2.72b}$$

where the transition probability $p(\mathbf{s}_{t+1} = \mathbf{s}'|\mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a})$ and the reward function $r(\mathbf{s},\mathbf{a})$ are the only elements that remain to be defined.

2.6.3 Model-Based and Model-Free Reinforcement Learning

When an agent interacts with a real-world environment, it encounters an unknown *real transition model* where physical interactions can be both expensive and time-consuming. For instance, in a robotic manipulator reaching task, the mechanical system can suffer wear and tear from extensive action trials, in addition to the processing time required for sampling trajectories (M. P. Deisenroth et al. 2011). In this context, two primary approaches emerge in RL: model-based and model-free methods.



(a) Model-based RL (b) Model-free RL

FIGURE 2.22. Model-based and model-free RL. In Model-based RL, a simulated transition model is used.

A *model-free* method, as illustrated in Figure 2.22b, learns an optimal policy primarily through direct interactions with the environment. This approach does not rely on an explicit model of the environment. Instead, it focuses on learning from the rewards and consequences of actions taken. For example, in *temporal difference learning*, a model-free method, the value of a state $V^{\varrho}(\mathbf{s})$ is updated based on the rewards received and the estimated values of future states. The update rule in such a scenario is given by:

$$V^{\varrho}(\mathbf{s}) \leftarrow V^{\varrho}(\mathbf{s}) + \gamma \cdot (r(\mathbf{s}, \mathbf{a}) + \zeta V^{\varrho}(\mathbf{s}') - V^{\varrho}(\mathbf{s})) \ , \tag{2.73}$$

where γ represents the learning rate, and ζ is the discount factor.

Conversely, a model-based approach, depicted in Figure 2.22a, employs an approximation of the real transition model known as the *simulated transition model*. This model approximates the robot-environment interactions via $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$. By using a simulated transition model, thousands of simulated trajectories can be generated, potentially covering more of the state space. As a result, fewer real interactions with the environment are needed to learn an effective policy (R. S. Sutton and Barto 2018).

Algorithm 12: Model-Based Reinforcement Learning

input : p_θ – Simulated transition model
$\varrho_{\mathbf{w}}$ – Base policy (e.g. random policy)
T – Horizon or trajectory size
output : $p_\theta^*, \varrho_{\mathbf{w}}^*, \mathcal{D}^p$

$\mathcal{D}^p \leftarrow$ GetInitialDataset() // Get an initial supervised dataset of $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
while *stopping condition not met* **do**
  $\mathbf{s}_t \leftarrow$ GetCurrentState()
  $p_\theta \leftarrow$ LearnTransitionModel$(\mathcal{D}^p)$
  $\varrho_{\mathbf{w}} \leftarrow$ LearnPolicy$(\mathcal{D}^p)$
  $(\mathbf{a}_t, \ldots, \mathbf{a}_{t+T-1}) \leftarrow$ PlanTrajectory$(p_\theta, \varrho_{\mathbf{w}}, \mathbf{s}_t)$
  $\mathbf{a}^* \leftarrow \mathbf{a}_t$
  $\mathcal{D}^\varrho \leftarrow$ SendToActuators$(\mathbf{a}^*)$ // Execute first planned control signal
  $\mathcal{D}^p \leftarrow \{\mathcal{D}^p \cup \mathcal{D}^\varrho\}$

In a model-based RL approach, the simulated transition model is defined a priori or learned iteratively. Two classic model-based methods that are also value-based methods are *value-iteration* and *policy-iteration*. They receive a simulated transition model and instant reward function and compute a table that maps states to values $V^*(\mathbf{s})$, and a table that maps states to actions $\varrho^*(s)$ in the whole state space $\mathcal{S}$. Those kinds of methods are known as *tabular methods*. Tabular methods are used when the possible number of states and actions are small enough (small $\mathcal{S}$ and $\mathcal{A}$) to form a state-action table, which is the policy.

In robotics, however, states are usually continuous, such as a state consisting of sensor measurements like the position and velocity of a robot. In fact, probabilistic trajectories mainly consist of a *continuous state space* $\mathcal{S}$ consisting of real-valued state vectors $\mathbf{s}$ and also a *continuous action space* $\mathcal{A}$ consisting of real-valued vector actions $\mathbf{a}$. When the state space $\mathcal{S}$ is too large, it can be transformed via the feature mapping $\Phi : \mathbf{s} \in \mathcal{S} \rightarrow \Phi(\mathbf{s}) \in \mathcal{H}$, which, in this particular case, is used for reducing the state space size: $|\mathcal{H}| < |\mathcal{S}|$. The transformed state space is then used for approximating the value function with a parameterised value function $\hat{V}(\mathbf{s}, \mathbf{w}) \approx \hat{V}(\mathbf{s})$. The parameterised value function at a single state can be defined as $\hat{V}(\mathbf{s}, \mathbf{w}) = \mathbf{s}^T\mathbf{w}$ if a linear model approximation of the value function is reasonable. Such a method is known as *function approximation*, and it can also work for continuous state spaces (van Hasselt 2012) since the parameterised value function value is not used as a lookup table.

As with the value function, the policy can also be parameterised and defined as $\varrho^{\mathbf{w}}$. *Policy search methods* are an alternative to value-based methods. In a value-based method, the policy is derived indirectly by finding actions that maximise the value function. A policy search method examines different policy parameterisations, bypassing value function assignment (Peshkin 2002). From a supervised learning approach, we can obtain a dataset of states and actions from real data to empirically compute $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$. Following usual model-based RL algorithms proposed in the literature (Oliveira et al. 2018; Scannell 2022), a general model-based RL approach follows the steps in Algorithm 12 where the agent incrementally explores its environment, collecting states and actions to improve its simulated transition model. Line 1 collects some initial data using the base policy $\varrho_{\mathbf{w}}$. The initial data $\mathcal{D}^p$ can consist of n tuples $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$, where s is the current state, a is the next action, and s' is the next state. Line 2 obtains the current state. Then, line 3 is a simulated transition model learning step where $\mathcal{D}^p = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)_i\}_{i=0}^n$ is used to update the simulated transition model p_θ by optimising θ.

The policy learning step corresponds to optimising the policy parameters $\mathbf{w}$ for $\varrho_{\mathbf{w}}$ in line 5. Considering T timesteps, a trajectory of size T is planned using $\varrho_{\mathbf{w}}$ and p_θ in line 6. A first optimal action is obtained and sent to the real robot actuators in lines 7 and 8. Finally, the new data $\mathcal{D}^\varrho$ obtained from the single trial is appended to $\mathcal{D}^p$, which is then used in the next iteration.

2.6.4 Policy Learning and Connection with Optimal Control

Learning a simulated transition model can be done with a GP or any other supervised learning method, such as a deep neural network (Gal et al. 2016), which is capable of scaling to high dimensional state spaces. The policy weights can be optimised using a *gradient-based* approach. For example, there is *probabilistic inference for learning control (PILCO)* (M. Deisenroth and Rasmussen 2011), which handles uncertainty requiring few data points to learn a parameterised policy. PILCO learns the simulated transition model with a GP, which yields normally distributed predictions about the state of the robot. The GP models the difference between the current $\mathbf{s}_{t+1}$ and the previous one $\mathbf{s}_{t-1}$ given the action $\mathbf{a}_{t-1}$. Then a difference function is defined as $\Delta_t = \mathbf{s}_{t+1} - \mathbf{s}_t + \nu$ where $\nu \sim \mathcal{N}(0, \sigma_\nu^2 \mathbf{I})$. The GP yields one-step predictions via the

predictive posterior distribution

$$
\begin{aligned}
p\left(\mathbf{s}_t \mid \mathbf{s}_{t-1}, \mathbf{a}_{t-1}\right) &= \mathcal{N}\left(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t\right) \\
\boldsymbol{\mu}_t &= \mathbf{s}_t + \operatorname{mean}(\Delta_t) \\
\boldsymbol{\Sigma}_t &= \operatorname{cov}(\Delta_t) \ ,
\end{aligned}
\tag{2.74}
$$

where mean and cov are the mean and covariance for the GP for Δ_t as in Equation 2.41c. The authors define a parameterized policy $\varrho(\mathbf{s}, \mathbf{w})$. Having an instant cost function $\mathrm{c}(\mathbf{s})$ defined beforehand, *policy learning* can be done by minimizing the expected return:

$$
J(\mathbf{w}) = \sum_{t=0}^{T} J_t = \sum_{t=0}^{T} \mathbb{E}_{\mathbf{s}_t}\left[\mathrm{c}\left(\mathbf{s}_t\right)\right] \tag{2.75}
$$

$$
\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} J(\mathbf{w}) \ , \tag{2.76}
$$

where $\mathbb{E}_{\mathbf{s}_t}\left[c\left(\mathbf{s}_t\right)\right] = \int c\left(\mathbf{s}_t\right) \mathcal{N}\left(\mathbf{s}_t \mid \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t\right) \mathrm{d}\mathbf{s}_t$. The minimisation can be computed by applying the chain rule $\frac{\mathrm{d}J_t}{\mathrm{d}\theta} = \frac{\mathrm{d}J_t}{\mathrm{d}p(\mathbf{s}_t)} \frac{\mathrm{d}p(\mathbf{s}_t)}{\mathrm{d}\theta}$, and a gradient-based approach to optimise the policy parameters $\mathbf{w}$.

Some methods do not include the simulated transition model learning step. For example, a BO approach can be used to update parameters for car racing in a model-free way (Oliveira et al. 2018). Other related methods include using trajectory optimisation methods from other areas, such as optimal control. The first connection is how the robot's behaviour is modelled. The dynamics model from Equation 2.3 $\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t)$ has the same interpretation as the transition model $p\left(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t\right)$.

The second connection between optimal control and RL is that in the context of RL, a *prediction problem* has the goal of estimating the value function V^{ϱ} for a given policy ϱ, while in a *control problem*, the goal is to find an optimal policy (R. Sutton and Barto 2018). The other connection has to do with what is known as data-driven control and learning-based control. As described in Section 2.1.4, planning a trajectory can be done with LQR, MPPI, or any other algorithm from optimal control. The controller is part of the agent in RL. In fact, the policy itself is often called *control policy* in surveys of RL-based solutions for optimal control (Buşoniu et al. 2018; Kiumarsi et al. 2018).

2.7 Summary

This chapter provided the necessary background used as a foundation for developing a framework for optimising MPC. Each section presented only specific concepts. Control for robotics was introduced from an RL perspective since both areas are highly related. Then, supervised learning, optimisation, and Bayesian learning were presented. The optimisation section focused on optimising noiseless functions, while the Bayesian learning section introduced BO as a way to handle noisy observations. Finally, RL concepts focused on model-based RL were described.

CHAPTER 3

Heteroscedastic Optimisation for Stochastic Model Predictive Control

3.1 Introduction

This chapter aims to build a framework for optimising stochastic model predictive control (MPC) methods for robotic systems. The challenge is the usually insufficient real data available for learning controllers since the system is to be used in the real world. The way this chapter deals with a real-world environment is by making use of reinforcement learning (RL), control theory and Bayesian learning to develop a framework that performs controller optimisation while handling both expensive and noisy observations. The method proposed in this chapter and experiments are part of a research paper published at RA-L[1].

Aiming at solving the data availability problem, experiments on hyperparameter optimisation for stochastic MPC lead to not only noisy observations but also heteroscedastic noise, which brings the objective of proposing a BO-based method that can be robust to such type of noise. In general, this chapter investigates the effect of heteroscedasticity on finding an optimal controller. Bayesian optimisation (BO) was chosen among all the optimisation methods in the literature because it was widely used for optimising robotic tasks with few interactions with the real world (Marco-Valle 2020, July). Taking into account the aforementioned peculiarities, this chapter makes the following contributions:

- a framework to optimise stochastic MPC via heteroscedastic BO;

[1] Guzman, R., Oliveira, R., & Ramos, F. (2021). Heteroscedastic bayesian optimisation for stochastic model predictive control. *IEEE Robotics and Automation Letters*, *6*(1), 56–63

- a class of parametric models for the heteroscedastic noise in the controller's performance; and
- experimental results on benchmark continuous control problems in simulated and real scenarios.

The remainder of this chapter is structured as follows. Section 3.2 reviews relevant prior work in the areas of stochastic MPC and BO. Section 3.3 introduces cumulative reward computation due to interactions with the real-world environment, which allows a formulation for modelling reward-based problems. This is done by following an RL approach to designing learning-based controllers. Section 3.4 visually shows and describes the heteroscedastic behaviour of the cumulative reward with respect to the controller hyperparameters and formulates a way to optimise the robot performance by optimising the expected cumulative reward. Section 3.5 describes the Gaussian process (GP) formulation as a surrogate model for inferring controller hyperparameters.

The first main contribution is described in Section 3.6 where a parametric heteroscedastic noise model for the expected cumulative reward is described. Next, given the formulation described before, the controller optimisation framework is detailed in Section 3.7. A heteroscedastic GP regression model is used as a surrogate model for the hyperparameter search space. BO allows the optimal selection of search space locations to be evaluated by following an exploration-exploitation paradigm. Finally, Section 3.8 presents experimental evaluations of the proposed framework in simulated classic control environments and a real robot. Finally, Section 3.9 summarises the results obtained and the chapter's contributions.

3.2 Related Work

This section reviews some relevant work from RL for control, stochastic MPC and BO-based optimisation focused on robotics applications.

3.2.1 Model Predictive Control

Within the areas of control and RL, model-based control and model-based RL have a noticeable similarity, mainly in the use of a dynamics model of the robot as noted in Görges (2017). In an RL context, the tendency to produce an optimal action is reinforced, which is related to the objective of classic feedback control by cleverly exploiting information from interactions. However, due to the data availability problem, model-based approaches are preferable for real-world applications (T. Wang et al. 2019) since they make use of a simulated dynamics model to simulate interactions with the real world.

Regarding controllers, there is classical control where PID controllers are common in applications such as heating, ventilation, and air conditioning (HVAC) (Fiducioso et al. 2019). Within what is known as modern control, methods such as model-based control, learning-based control and data-driven control are related and are commonly used in robotics applications due to advancements concerning physics simulators and data processing (Collins et al. 2021). The MPC problem solved with classic optimal control methods such as a linear quadratic regulator (LQR) usually becomes inefficient when dealing with highly non-linear dynamics and non-convex reward functions (T. Wang et al. 2019; Williams et al. 2017a). Meanwhile, state-of-the-art approaches can efficiently adapt to challenging stochastic environments with the introduction of sampling-based approaches for trajectory optimisation such as a flexible data-driven sampling-based MPC method known as model predictive path integral (MPPI) (Williams et al. 2016) with applications from autonomous driving (Williams et al. 2018) to manipulators (Bhardwaj et al. 2022) where trajectories are determined by optimising gaussian distributed controls. Among the MPPI variations, some realise variations in the weighting of trajectories (Williams et al. 2016, 2017b).

Like any other optimisation algorithm, MPPI has hyperparameters for which some hyperparameter optimisation method is essential. Hyperparameters often have to be optimised according to the task and learning behaviours, leading to settings that are not necessarily transferable across tasks (Mahmood et al. 2018). To exemplify, online MPPI hyperparameter optimisation can

be done by learning the dynamics offline and adjusting to disturbances online by varying an adaptive temperature hyperparameter (Liang et al. 2019).

3.2.2 Bayesian Optimisation

In order to perform policy search, the reward function can be defined as a direct function of policy parameters. A parametric policy can be defined to allow policies of different shapes depending on the kernel function (Oliveira et al. 2018). Another way is to use a model-based policy search method where the dynamics is modelled with a GP to deal with inherent dynamics model errors, which helps achieve a data-efficient framework (M. P. Deisenroth et al. 2015). Other methods use BO not only for dealing with dynamics model learning. Bayesian optimisation has been widely applied to hyperparameter tuning (Shahriari et al. 2016; Snoek et al. 2012). Essentially, any global optimisation method from evolutionary-based methods such as evolutionary strategies (Hansen et al. 2003) to surrogate-based optimisation methods like BO can be used for hyperparameter tuning. A framework similar to the one described in this chapter is proposed in Lu et al. (2020), where BO is used to identify the optimal hyperparameters of an MPC controller for HVAC systems.

A common choice for a Bayesian surrogate model is a GP that assumes that the observation noise is i.i.d. Gaussian across the search space. However, such an assumption is unrealistic because the noise can also be input-dependent, which leads to heteroscedastic noise. The heteroscedastic noise with parametric noise models can be learnt via maximum likelihood as in Wilson and Williams (2017), which makes use of less computational resources than other approaches in the literature. For example, one approach is to add a second GP prior to the log-variance of the noise model (Goldberg et al. 1997), which results in a stochastic process that is no longer Gaussian and requires Markov chain Monte Carlo for inference (Andrieu et al. 2003).

Approximate inference methods have also been proposed to reduce the computational overhead by using variational inference to compute the GP posterior distribution (Kersting et al. 2007; Kuindersma et al. 2012; Lázaro-Gredilla and Titsias 2011), which can still be expensive. One

of the objectives of this chapter is to take a simpler approach and use a flexible parametric noise model to encode prior knowledge about the noise.

3.3 Episodic Reward Formulation

Following the very popular paradigm known as *sense-plan-act*, a robotic system receives input sensor measurements and produces output signals that are sent to the actuators. For example, an autonomous vehicle may have position sensors and servomotors as actuators for rotary motions. An action $\mathbf{a}$ might produce changes in the *state of the robot* $\mathbf{s}$, which can consist of sensor measurements. The robot planning is done internally in a *control component* that must be able to regulate itself according to a desired state. The robot's desired state can be reinforced by following an RL approach. As opposed to a cost function used in usual controllers described in Section 2.1, an instant reward function is used: $r(\mathbf{s}, \mathbf{a})$ where $\mathbf{s} \in \mathcal{S}$ is a continuous state space, and $\mathbf{a} \in \mathcal{A}$ a continuous action space. The reward can be thought of as the output signal coming from the system due to the controller-system interaction as in Figure 3.1. To realise trajectory optimisation, the proposed framework also measures a cumulative reward or episodic reward y that consists of accumulating instant rewards over episodes:

$$y = \sum_{i=1}^{n_s} r(\mathbf{s}_i, \mathbf{a}_i) \ , \tag{3.1}$$

where n_s is the number of timesteps, which is considered fixed. The proposed framework consists of improving the system performance by only observing the cumulative rewards, hence

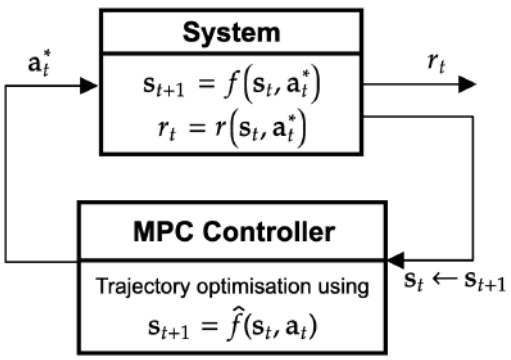


FIGURE 3.1. MPC controller diagram where the output signal is the reward.

Algorithm 13: Reward-Based MPC

input : $\hat{f}$ – Simulated dynamics model
n_s – Number of timesteps
T – Finite horizon length
$\mathbf{x}$ – Controller hyperparameters
output : y – Cumulative Reward

```
y = 0
for i ← 1 to n_s do
    s_t ← GetCurrentState()
    a* ← MPC(f̂, s_t, x)
    r_i ← SendToActuators(a*)        // Execute first planned control signal
    y += r_i                         // Accumulate rewards
```

the name reward-only optimisation. At each timestep, the controller returns an optimal next action a^* that is then sent to the system actuators. Considering the dynamics model notation from Section 2.1.4, the system corresponds to the actual robot that has its real dynamics model f, and to allow few controller-system interactions, we make use of a simulated dynamics model $\hat{f}$ for simulated trajectories.

3.4 Hyperparameter Optimisation and Heteroscedasticity

The controller we choose to optimise is an MPC controller described in Section 2.1.4 and treated as a black-box system that receives hyperparameters $\mathbf{x}$, a simulated dynamics model $\hat{f}$, and a time horizon n_s to plan trajectories. As a black-box system, we deal with hyperparameter optimisation by considering $\hat{f}$ and n_s fixed. We also consider a control perspective for the policy. The parameterised policy from Algorithm 12 can be seen as a control policy $\pi_{\mathbf{x}}$, but we consider the controller as a black box. Then, the controller function MPC$(\hat{f}, \mathbf{s}_t, \mathbf{x})$ outputs the next optimal action. The resulting controller optimisation is learning-based and follows the steps from Algorithm 13 to compute the episodic reward.

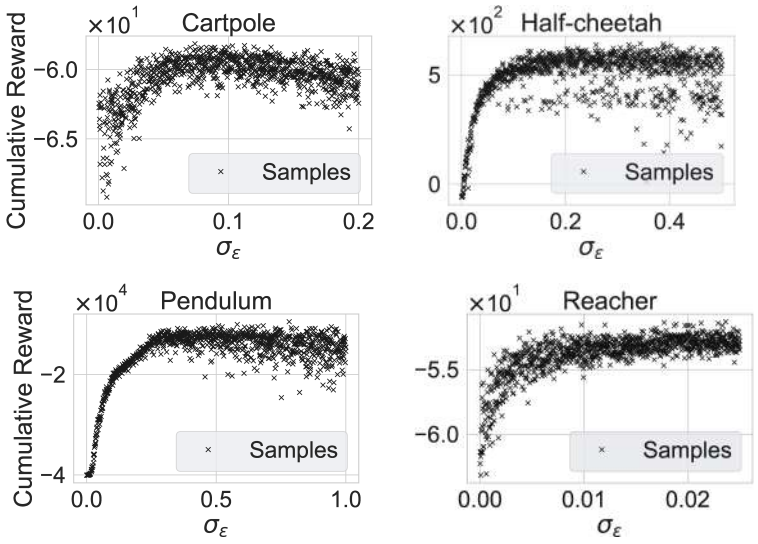


FIGURE 3.2. Heteroscedastic behaviour in classic control tasks and across a range of values for the control variance σ_ϵ of an MPPI controller.

Considering a realistic noisy cumulative reward function, the noisy black-box function y is defined as follows:

$$y(\mathbf{x}) = g(\mathbf{x}) + \nu(\mathbf{x}) \tag{3.2}$$

$$g : \mathcal{X} \to \mathbb{R}\,, \tag{3.3}$$

where the input to optimise is the hyperparameters $\mathbf{x}$. The input-dependent noise formulation is used due to initial experiments seen in Figure 3.2 regarding the cumulative reward with the MPPI stochastic controller. For each classic control problem, the plots show the obtained cumulative reward as a function of the control variance hyperparameter σ_ϵ. The cumulative reward variance across the range of control variance values is not uniform. The noise in the highest regions tends to be significantly higher than elsewhere, evidencing an input-dependent noise known as *heteroscedasticity*.

Then, the problem there is to find the actual hyperparameters that maximise the cumulative reward, which is noisy and heteroscedastic. The objective should be to maximise the *noiseless cumulative reward* g as a function that depends on $\mathbf{x}$. This problem corresponds to noisy optimisation as seen in Section 2.5. The way we deal with that situation is by maximising

the function corresponding to the actual expected cumulative reward $g := \mathbb{E}[y]$, which is intractable as it requires marginalising over many variables, including the stochastic behaviour of the controller itself. A way to approximate g is by empirically averaging cumulative rewards over a certain number of episodes: $\hat{g} := \frac{1}{n_e} \sum_{j=1}^{n_e} y_j(\mathbf{x})$ based on a finite number of episodes n_e.

Another way is to use a method that can deal with noisy functions such as BO, which was seen before in Section 2.5. $\hat{g}$ will be called *expected cumulative reward* from now on. Then, the noisy black-box optimisation problem is defined as

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathcal{X}}{\operatorname{argmax}}\, \hat{g}(\mathbf{x}) \,. \tag{3.4}$$

3.5 Gaussian Process Formulation

The optimisation method we use is BO since it is data-efficient and can handle the uncertainty of noisy functions. By using a GP as a surrogate model, we are given a design matrix of n training data points $\mathbf{X}$. Then we can have a normally distributed function g that evaluated in $\mathbf{X}$ results in $g(\mathbf{X}) = [g(\mathbf{x}_1), \ldots, g(\mathbf{x}_n)]^\mathsf{T}$ where each input belongs to a search space $\{\mathbf{x}_i\}_{i=1}^n \subset \mathcal{X}$. The prior for g can be defined with the zero-mean assumption since the mean is unknown, as explained in Section 2.4.4. Then, the GP is trained with observed data to obtain the GP prior:

$$\begin{bmatrix} \mathbf{y} \\ g_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K}_{\mathbf{XX}} + \sigma^2 I & \mathbf{K}_{\mathbf{X}\mathbf{x}_*} \\ \mathbf{K}_{\mathbf{x}_*\mathbf{X}} & \mathbf{K}_{\mathbf{x}_*\mathbf{x}_*} \end{bmatrix}\right) \,, \tag{3.5}$$

where $\mathbf{K}_{\mathbf{XX}}$, $\mathbf{K}_{\mathbf{X}\mathbf{x}_*}$, $\mathbf{K}_{\mathbf{x}_*\mathbf{X}}$, and $\mathbf{K}_{\mathbf{x}_*\mathbf{x}_*}$ are covariance matrices defined as in Equation 2.42. By denoting a test point as $\mathbf{x}_*$, we condition $g_* = g(\mathbf{x}_*)$ on the observations as $\mathbf{y} = g(\mathbf{X}) + \boldsymbol{\nu}$, where $\boldsymbol{\nu} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_\nu)$ represents the observation noise. Then, considering the zero-mean GP prior, the GP predictive posterior from Equation 2.41c can be computed analytically as follows:

$$g_* \mid \mathbf{x}_*, \mathbf{X}, \mathbf{y} \sim \mathcal{N}\left(\text{mean}(g_*), \text{cov}(g_*)\right) \tag{3.6a}$$

$$\text{mean}(g_*) = \mathbf{K}_{\mathbf{x}_*\mathbf{x}}[\mathbf{K}_{\mathbf{XX}} + \boldsymbol{\Sigma}_\nu]^{-1}\mathbf{y} \tag{3.6b}$$

$$\text{cov}(g_*) = \mathbf{K}_{\mathbf{x}_*\mathbf{x}_*} - \mathbf{K}_{\mathbf{x}_*\mathbf{x}}(\mathbf{K}_{\mathbf{XX}} + \boldsymbol{\Sigma}_\nu)^{-1}\mathbf{K}_{\mathbf{X}\mathbf{x}_*} \ . \tag{3.6c}$$

Under a homoscedastic noise assumption, $\boldsymbol{\Sigma}_\nu = \sigma_\nu^2\mathbf{I}$. In this chapter, however, we use a heteroscedastic, i.e. input-dependent, noise formulation where $[\boldsymbol{\Sigma}_\nu]_{ij} = k_\nu(\mathbf{x}_i, \mathbf{x}_j)$ and $k_\nu : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a positive-definite kernel function that needs to be formulated.

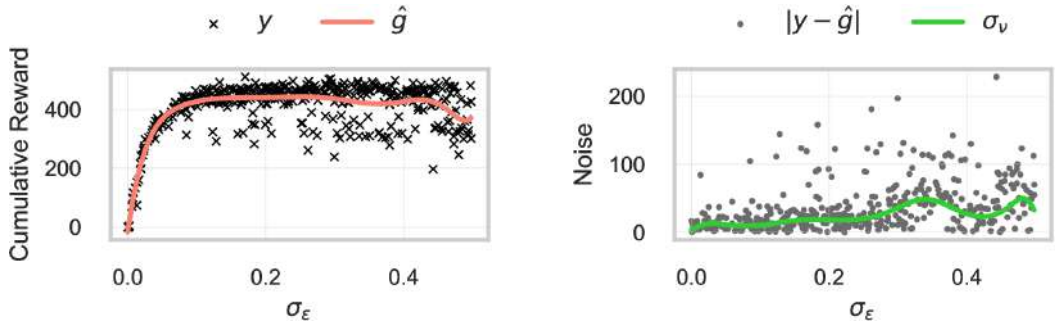
3.6 Heteroscedastic Noise Model

In order to handle noisy heteroscedastic observations $\mathbf{x}$, we make use of a variation to the traditional GP known as heteroscedastic GP. The main issue is how to model the external noise $\nu(\mathbf{x})$. We consider that the cumulative reward evaluation is corrupted by a jointly Gaussian noise $\boldsymbol{\nu} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_\nu)$. The way to compute K_ν for the heteroscedastic case can go from modelling it with another GP as in Kersting et al. (2007) to variational inference (Kuindersma et al. 2012), but this chapter proposes to obtain $\boldsymbol{\Sigma}_\nu$ by introducing a kernel function k_ν that fits the empirical noise given the data. Then, both the covariance matrix kernel k and the noise kernel k_ν can be summed up as follows:

$$k(\mathbf{x}_i, \mathbf{x}_j) + k_\nu(\mathbf{x}_i, \mathbf{x}_j) \ . \tag{3.7}$$

The kernel function k should be chosen according to the task, and it should preferably be one with the fewest kernel hyperparameters. Some possible kernel functions were shown in Table 2.4. Now, to define k_ν, we use a flexible parametric form for the noise model with the assumption that episodes are executed independently, which leads to $k_\nu(\mathbf{x}, \mathbf{x}') = 0$. Then the main concern is to model $k_\nu(\mathbf{x}, \mathbf{x}) \approx \sigma_\nu^2(\mathbf{x})$.

The standard deviation can be computed given observed data. Using the MPPI controller from Section 2.1.6, Figure 3.3a was obtained by experimenting with a control task where cumulative



(a) Expected cumulative reward function (b) Noise function

FIGURE 3.3. Example of a 10-degree polynomial regression model $\hat{g}$ fitting the expected cumulative reward function in (a), while an estimate for the noise model σ_ν is represented as the blue curve in (b).

rewards y were obtained for several values of $\mathbf{x} := \sigma_\epsilon$. The expected cumulative reward is computed with a 10-degree polynomial regression. In this context, the noise corresponds to the difference between the observed cumulative reward y and the expected cumulative reward for a given setting $\mathbf{x}$, i.e. $\nu(\mathbf{x}) := y - \hat{g}(\mathbf{x})$. The noise function $\nu(\mathbf{x})$ is modeled with $k_\nu(\mathbf{x}, \mathbf{x})$. The parametric model for the noise function is defined as:

$$\sigma_\nu(\mathbf{x}) = z \cdot \exp\left(\boldsymbol{\beta}^\mathsf{T}\Phi(\mathbf{x})\right), \tag{3.8}$$

where $\boldsymbol{\beta} \in \mathbb{R}^m$ and $\Phi : \mathcal{X} \to \mathbb{R}^m$ is a feature map. Similar to what the error noise σ_ν does for homoscedastic BO Section 2.5, a scalar factor z is added in case of variations of the cumulative reward around its expected value. Small values of z produce expected cumulative reward functions that are close to their mean, and larger values allow more variation. If z is large, the modelled expected cumulative reward function will be able to account for more outliers. A smooth feature map Φ allows the noise model to fit the gradually changing noise variance, as in Figure 3.3, with no sharp changes across the search space. The exponential term ensures the positive-definiteness of k_ν, according to the covariance matrix definition, and it includes a generalised linear model $\boldsymbol{\beta}^\mathsf{T}\Phi(\mathbf{x})$ that allows it to vary the noise distribution as a function of the input. β are regression model parameters to be learned. The feature mapping is used to transform the input $\mathbf{x}$ into a higher dimensional space so as to capture more non-linear interactions. The choice of the feature map Φ is arbitrary.

Having a parametric form for k_ν, the problem is how to learn a suitable noise model from data. The way σ_ν is learned for traditional homoscedastic BO is by optimising the GP log-marginal likelihood seen in Equation 2.47. The noise model is learned in a two-stage regression problem. From a set of randomly sampled inputs $\mathbf{x}_i \in \mathcal{S}$, we can approximate the expected cumulative reward $\hat{g}$ with a flexible generalised linear regression model based on the feature map $\Phi : \mathcal{X} \to \mathbb{R}^m$ and learn weights $\mathbf{w} \in \mathbb{R}^m$ as

$$\hat{g}(\mathbf{x}) \approx \mathbf{w}^\mathsf{T}\Phi(\mathbf{x}) \,. \tag{3.9}$$

With the estimate $\hat{g}$ we then fit the residuals $|g(\mathbf{x}) - \hat{g}(\mathbf{x})|$ with (3.8) as a regression problem.

3.7 Bayesian Controller Optimisation

Optimising the controller with BO is realised via hyperparameter optimisation in a procedure described in Algorithm 14. The search space $\mathcal{X}$ input is essential since it requires expert knowledge about the task by considering how each controller hyperparameter affects the result. The search space can be defined as intervals for each hyperparameter. A reward function r is also received as input. The other important input is the GP model $\mathcal{M}$. Beforehand, a collected dataset $\mathcal{D}$ can be used as initial observations for optimising the GP hyperparameters by optimising the GP log-marginal likelihood Equation 2.47, which results in optimal GP hyperparameters Ω.

The algorithm loops through n iterations corresponding to the number of evaluations performed by BO. Then it receives the current state of the robot in line 3. The GP model is trained with the collected dataset $\mathcal{D}$ in line 4. Since the objective is for the control component to decide which hyperparameters to try next in the real robot, an acquisition function received as input is maximised in line 5. The predictive posterior distribution from Equation 3.6c is used by the acquisition function, which allows us to balance exploration and exploitation of the objective function regions as explained in Section 2.5.2. Then, the input to try next is $\mathbf{x}_t$. The loop that starts at line 6 is for collecting n_e episodic or cumulative rewards. The loop that starts at line 8 is for collecting a single episodic reward after interacting with the real robot n_s times.

Algorithm 14: Bayesian Controller Optimisation

input : $\hat{f}$ – Simulated dynamics model
$\mathcal{M}$ – GP model
Ω – GP hyperparameters
$\mathbf{x}$ – Controller hyperparameters
α – Acquisition function
r – Dense reward function
n – Number of BO iterations
n_s – Number of timesteps in an episode
n_e – Number of episodes to average the cumulative reward
T – Finite horizon length
$\mathcal{X}$ – Controller hyperparameter search space

output : $(\mathbf{x}^*, \hat{g}^*)$

1 $\mathcal{D} \leftarrow$ GetInitialDataset() // Get an initial supervised dataset consisting of $(\mathbf{x}, \hat{\mathbf{g}})$
2 **for** $t \leftarrow 1$ **to** n **do**
3 $\quad \mathbf{s}_t \leftarrow$ GetCurrentState()
4 $\quad$ Fit a GP model $\mathcal{M}_\Omega$ with the data $\mathcal{D}$
5 $\quad \mathbf{x}_t \leftarrow \text{argmax}_{\mathbf{x} \in \mathcal{X}}\, \alpha(\mathbf{x}, \mathcal{M}_\Omega, \mathcal{D})$
6 $\quad$ **for** $j \leftarrow 1$ **to** n_e **do**
7 $\quad\quad y_j(\mathbf{x}_t) = 0$
8 $\quad\quad$ **for** $i \leftarrow 1$ **to** n_s **do**
9 $\quad\quad\quad \mathbf{a}_i^* \leftarrow$ MPC$(\hat{f}, \mathbf{s}_t, \mathbf{x}_t, r)$
10 $\quad\quad\quad r_i \leftarrow$ SendToActuators$(\mathbf{a}_i^*, r)$
11 $\quad\quad\quad y_j(\mathbf{x}_t)$ += r_i
12 $\quad \hat{g}_t = 1/n_e \sum_j [y_j(\mathbf{x}_t)]$
13 $\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_t, \hat{g}_t)\}$

Inside the innermost loop, the controller with updated hyperparameters $\mathbf{x}_t$ performs trajectory optimisation and executes the next optimal action $\mathbf{a}_i^*$ in lines 9 and 10. It goes without saying that the controller performs simulations with the simulated dynamics model $\hat{f}$, starting from the current state $\mathbf{s}_t$. The controller receives the reward function r to optimise trajectories, but since MPC minimises a cost function, it can minimise the negative reward function.

Next, applying the action $\mathbf{a}_i^*$ returns a reward that is accumulated in y in line 11. Finally, since the cumulative rewards y are noisy, either $\text{mean}(g_*)$ or an averaged cumulative reward can be used as a performance measure, as in line 12. In this work, we use the second option since, after experimenting with control tasks, the mean obtained by the GP does not necessarily reflect noiseless function evaluations. It is also done so as to fairly compare the proposed heteroscedastic BO framework against other optimisation methods that will be seen in the next

TABLE 3.1. Reward functions used in the experiments. The Cartpole and Pendulum reward functions were taken from experiments in Gardner et al. (2017), and the rest from T. Wang et al. (2019).

Control Problem	Instant Reward
Cartpole	$-(s_{1,t}^2 + 500 * \sin {s_{3,t}}^2 + s_{2,t}^2 + s_{4,t}^2)$
Half-Cheetah	$\dot{s}_t - 0.01 \left\|\mathbf{a}_t\right\|_2^2 - inclination_t$
Pendulum	$-(50(\cos s_t - 1)^2 + \dot{s}^2) + 4000$
Reacher	$-(distance_t - \left\|\mathbf{a}_t\right\|_2^2)$

section. The final output of the Bayesian controller optimisation method is both the optimised controller hyperparameters $\mathbf{x}^*$ and the approximated optimal noiseless function value $\hat{g}^*$.

3.8 Experiments in Control and Robotic Problems

Considering the proposed heteroscedastic controller optimisation framework. This section presents experiments to assess the effectiveness of performing cumulative reward evaluations. We evaluate the framework by optimising the stochastic MPC controller known as MPPI for continuous control problems in both simulated problems and a physical robot platform. We address two main questions: (Q1) Is there a gain over homoscedastic BO denoted as BO$_{homo}$? (Q2) How does heteroscedastic BO (BO$_{hetero}$) perform against a non-BO baseline that does not take heteroscedasticity into account?

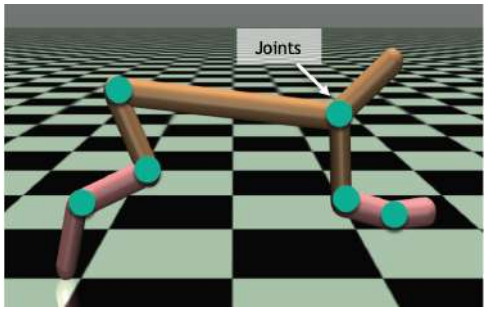
3.8.1 Control Problems

Starting with control problems, there are several simulated environments used in the literature. Two of the most commonly used simulators for RL are OpenAI Gym[2], and Mujoco (Todorov et al. 2012). Some of the most representative problems were selected to conduct experiments. The benchmark control problems used in this section are Cartpole, Pendulum, Half-Cheetah, and Reacher, which are shown in Figure 3.4. We used the instant reward functions $r(\mathbf{s}, \mathbf{a})$ to measure the performance of each control problem. Each one has a particular instant reward

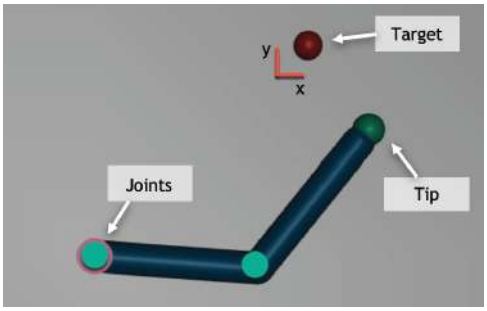
[2]OpenAI Gym: https://gym.openai.com

(a) Cartpole

(b) Pendulum



(c) Half-Cheetah



(d) Reacher

FIGURE 3.4. Control problems from OpenAI Gym and Mujoco.

function used in benchmark model-based RL method comparisons (T. Wang et al. 2019). The instant reward functions used in this section are listed in Table 3.1. Some slight modifications were made for Reacher and Half-Cheetah. We reduced the effect of actions and gave more priority to the distance to the target in the case of the Reacher problem. For Half-Cheetah, we added more priority to the inclination since Half-Cheetah would tend to turn upside down as its speed increases. The actuation is then set to finish when such inclination is greater than $\pi/2$ or lower than $-\pi/2$. These modifications make the rewards more informative for MPPI. Each control problem is described as follows:

- Cartpole consists of a pushable cart with a pole attached to it. The goal is to keep the pole in an upright position. The state space consists of the cart position from the centre $s_{1,t}$, the cart velocity $s_{2,t}$, the pole angle away from the upright vertical position $s_{3,t}$, and the pole velocity at the tip $s_{4,t}$. The reward function measures how far the pole angle is from a straight position.

- Pendulum is similar to Cartpole, but the cart is fixed, and the pole starts in a downright position. The state space consists of the joint angle ϑ and the pole velocity at the tip $\dot{\vartheta}$. The reward function also measures how far the pole angle is from a straight position.
- Half-Cheetah is a robot with the shape of half of a cheetah. The state space consists of 3 values representing the root joint position, 6 angular positions, and 9 velocities ($\dot{s}_t$ is a joint angular velocity). The goal is to move forward as fast as possible. The reward function measures how fast and forward Half-Cheetah moves without turning upside down plus a penalty for control inputs $\mathbf{a}_t$.
- Reacher is a 2-joint arm fixed at one end. The state space consists of 4 values representing joint angles, 2 values for the target position, 2 values for the velocity at the tip of the arm, and 3 more values denoting a vector to the target. The goal is to reach a target randomly positioned in a 2D plane. The reward function measures the distance from the tip of the arm to the target plus a penalty for control inputs $\mathbf{a}_t$.

The expected cumulative reward represents the expected time the pendulum stays in an upright position for the Cartpole and Pendulum problems. It represents the distance traversed in Half-Cheetah, and the speed to reach the target in Reacher. High expected cumulative rewards are the result of motions that increase the reward accordingly, e.g. Half-Cheetah would be expected to reach farther distances.

3.8.2 Optimisation Settings

To evaluate the expected cumulative rewards for each problem, the time horizon T and the number of trajectory rollouts M are set as fixed for the controller. The two hyperparameters to optimise are the temperature λ and the control variance σ_ϵ, which means that the variable to optimise is $\mathbf{x} = [\lambda, \sigma_\epsilon]$. We can use grid search to find search intervals for each hyperparameter. Figure 3.5 shows examples of evaluated expected cumulative rewards $\hat{g}$ for several hyperparameters values. Smaller search spaces can be found by first evaluating a grid of values within large enough intervals and then manually narrowing it down.

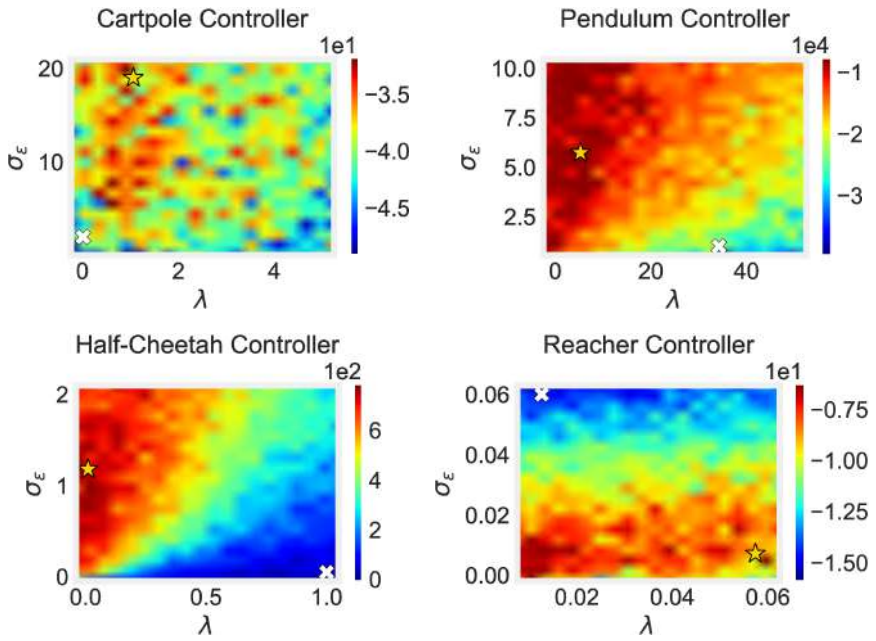
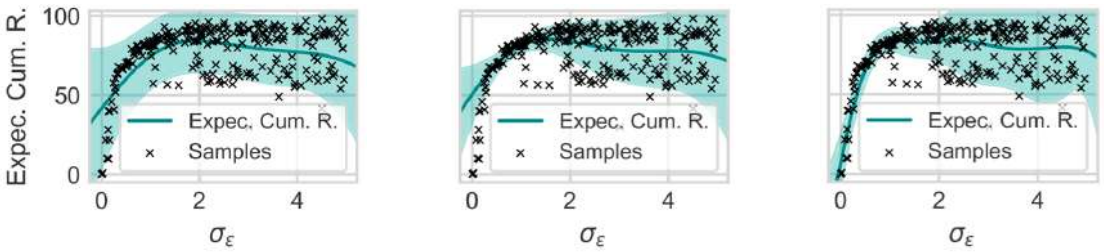


FIGURE 3.5. Search space in classic control tasks for varying control variance σ_ϵ and temperature λ values. The cross denotes the minimum location, while the star denotes the maximum location.

TABLE 3.2. MPPI hyperparameter search spaces and optimal values within the intervals per control problem.

Problem	n_e	T	M	λ interval	σ_ϵ interval
Cartpole	40	10	100	$[10^{-10}, 1.2]$	$[10^{-10}, 3.0]$
Pendulum	15	10	10	$[10^{-10}, 1.2]$	$[10^{-10}, 3.0]$
Half-Cheetah	25	14	10	$[10^{-10}, 0.1]$	$[10^{-10}, 2.5]$
Reacher	20	10	15	$[10^{-10}, 0.1]$	$[10^{-10}, 2.5]$

The search spaces for each MPPI hyperparameter and the other hyperparameter values considered as fixed are shown in Table 3.2. The number of timesteps considered for every task is $n_s = 200$. These were found by taking into account usual values for the hyperparameters that tend to be close to 0 in other control problems (Liang et al. 2019; Williams et al. 2016). The table also shows optimal values found within these narrowed intervals via grid search.



(a) Degree 1 (b) Degree 5 (c) Degree 10

FIGURE 3.6. Noise model with different polynomial degrees.

3.8.3 Gaussian Process Training and Data Scaling

For better comparison, both BO_{homo} and BO_{hetero} were implemented using the same squared exponential kernel from Equation 2.43 and UCB acquisition function from Equation 2.60 with $\delta = 1.2$. For the optimisation of the acquisition function h, we used L-BFGS-B (Byrd et al. 1995), which is a variation of the multi-start gradient search method for global optimisation. The GP log-marginal likelihood was also maximised to find optimal GP hyperparameters $\Omega := \{\sigma, \sigma_n, \ell\}$ for BO_{homo} and $\Omega := \{z, \sigma_n, \ell\}$ for BO_{hetero} also using L-BFGS-B. Ω was kept fixed after it was optimised. Both BO variations were optimised with 100 previously observed sample points that were generated from the defined hyperparameter intervals from Table 3.2 for each control problem.

A polynomial basis function $\mathbf{\Phi}$ was used for BO_{hetero} and evaluated for different degrees. A polynomial degree of 1 as in Figure 3.6a and 5 as in Figure 3.6b would result in a noise model ignoring small variances, while a higher degree would not. We then set a 10-degree polynomial model Figure 3.6c because it is the first high degree that correctly handles the increasing variance. The noise model was computed using the regression model in Equation 3.9. Something else to note about the GP hyperparameters is that their values have to be proportional to the range of the expected cumulative reward function in order to model for standard comparison among functions. The expected cumulative rewards were scaled to $[0, 100]$ and the input variable $\mathbf{x}$ values were scaled to $[0, 5]$ so as to make the GP training easier and also allow the use of the squared exponential kernel with a single lengthscale ℓ.

3.8.4 Heteroscedastic Noise Evaluation

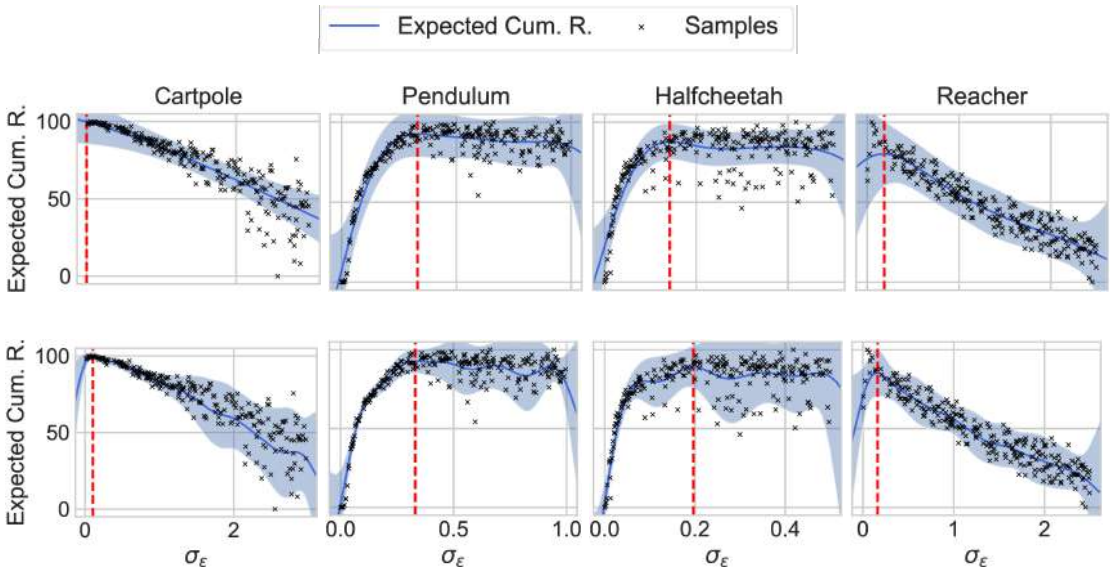


FIGURE 3.7. Expected cumulative rewards for hyperparameters sampled via grid search. A homoscedastic GP approximated the expected cumulative reward in the upper row, and a heteroscedastic GP in the lower row. The red dashed lines indicate the maximum expected cumulative reward found by the GP. The shaded regions correspond to two standard deviations from the mean.

To demonstrate how BO_{homo} and BO_{hetero} fit the expected cumulative reward function, Figure 3.7 shows the varying noise behaviour of the expected cumulative reward for intervals of the MPPI hyperparameter σ_ϵ. We can see that the noise around the mean increases with the x-axis hyperparameter. The noise heteroscedasticity is evident in all the control problems, so we can answer to **Q1**. There is a gain over BO_{homo} as more noise is captured, which means that the hyperparameter optimisation would be able to sample more hyperparameters from noisy regions without over-exploring. However, in Half-Cheetah and Pendulum, the noise is quite skewed around the mean, which means that the expected cumulative reward may not be Gaussian in those cases. The framework still captures a Gaussian noise for the rest of the control problems.

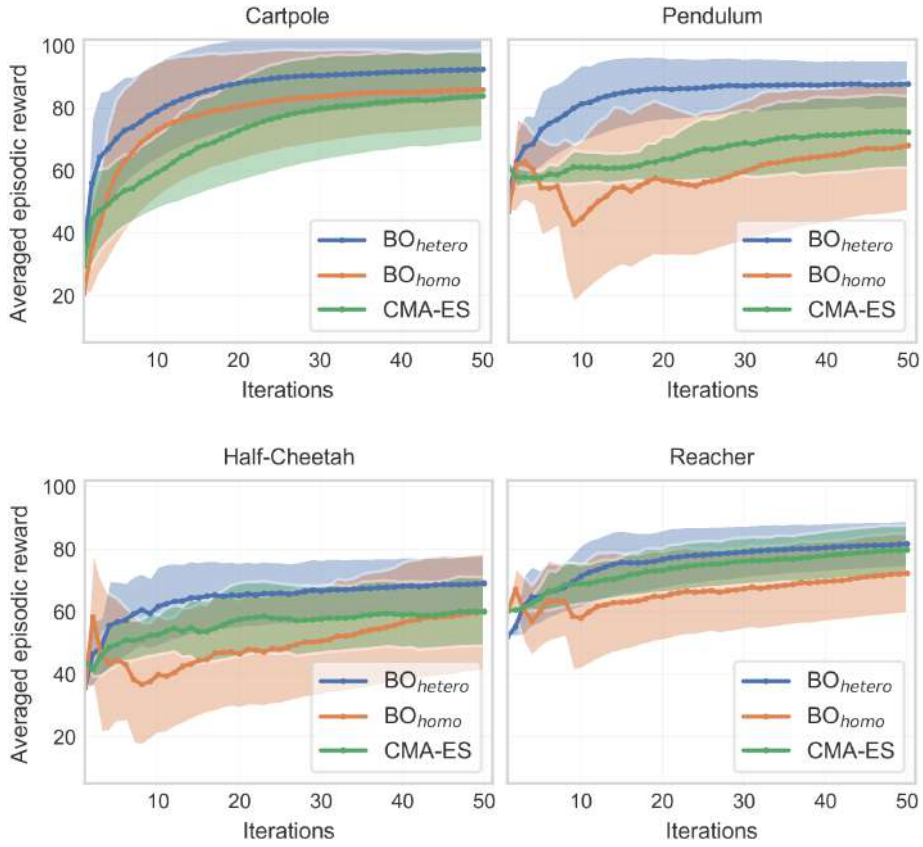


FIGURE 3.8. Optimisation performance. These results were averaged over 50 episodes with shaded areas and error bars corresponding to two standard deviations. Each method started at the same predefined point in the search space.

3.8.5 Method Comparison

To answer **Q2**, in Figure 3.8, we compare BO$_{hetero}$, BO$_{homo}$, and covariance matrix adaptation evolution strategy (CMA-ES) (Arnold and Hansen 2010), which is a non-BO baseline that does not take heteroscedasticity into account. In all experiments, only continuous search spaces were utilised. Discretisations were applied solely for visualisation purposes as in Figure 3.5. Now, to allow for proper comparison, each method started at a single defined point in the search space where there is a minimum. We use CMA-ES with $\sigma_0 = 1$ and a population size of 2. CMA-ES has been used for hyperparameter tuning and is considered to be a data-efficient black-box optimisation method that is also used in the robotics literature (Golovin et al. 2017; Loshchilov and Hutter 2016).

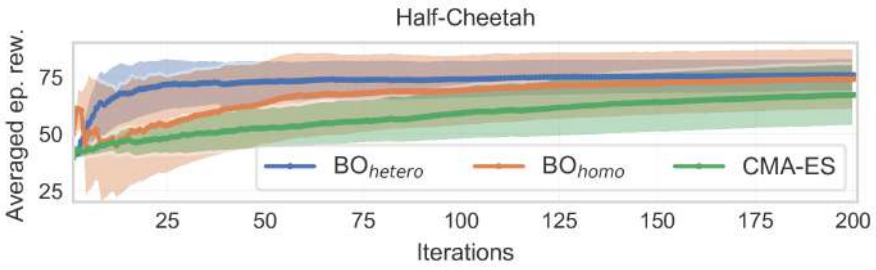


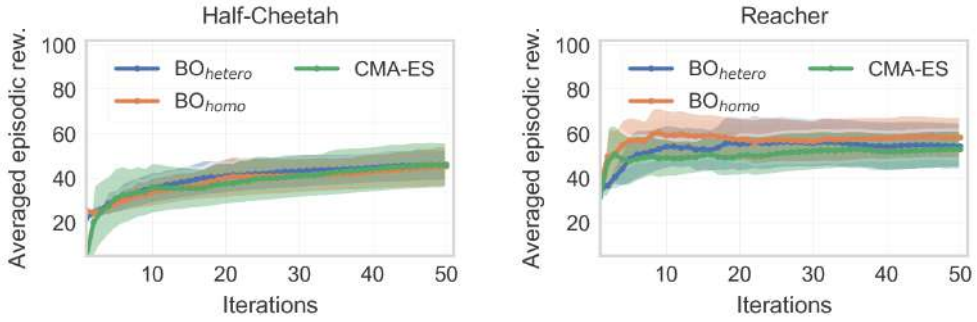
FIGURE 3.9. Performance for Half-Cheetah in 200 iterations.

As expected, BO$_{hetero}$ overcomes BO$_{homo}$ and CMA-ES. For BO$_{homo}$, the standard deviation reflects incorrect noise modelled in some regions as also shown in Figure 3.7. BO$_{homo}$ ends up with a higher standard deviation in most cases. For the Reacher task, there was not much improvement due to mostly homoscedastic regions in the sample collected. To assess long-term performance, we let the optimisation continue for 200 iterations for Half-Cheetah in Figure 3.9. In general, since BO$_{hetero}$ describes the noise behaviour, it finds optimal regions faster than CMA-ES.

Another aspect was the computational time required for each episode across different environments. In simpler tasks such as Pendulum, Cartpole, and Reacher, all methods, including BO and CMA-ES, demonstrated relatively low computational times, reflecting the lower complexity of these environments. However, it's important to highlight that in more complex tasks such as Half-Cheetah, the BO methods, particularly BO$_{hetero}$ required the longest computational time. This increased time demand is attributed to the intrinsic complexity of BO, which requires more computational resources for effective modelling and optimisation.

It is important to note that CMA-ES does not run inference from prior data, so it has to start without knowing anything about the search space. Meanwhile, because the GP hyperparameters were optimised beforehand, BO is able to apply prior knowledge encoded in the noise model to outperform CMA-ES in fewer iterations. BO approaches do global optimisation in fewer trials, which is the desired behaviour to deal with the data availability problem.

We experimented optimising Half-Cheetah and Reacher with their unmodified reward functions in Figure 3.10. The unmodified reward functions make the tasks challenging to optimise by all



(a) Unmodified Half-Cheetah (b) Unmodified Reacher

FIGURE 3.10. Performance using the unmodified reward functions.

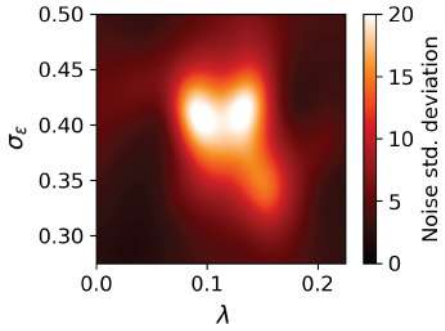
the methods used, which suggests MPPI has difficulties in solving these tasks. A possible reason is that the unmodified reward function is too uninformative for the MPPI controller.

3.8.6 Experiments with a Physical Robot

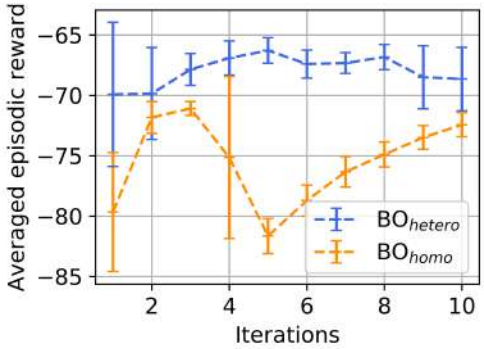
To assess the effects of real heteroscedastic noise in a physical system, some experiments are performed concerning MPPI controller optimisation for a physical robot. The four-wheel-drive skid-steer robot from Figure 3.11a was tasked with following a circular path at a set speed. The cost function was formulated as $c(s_t) = \sqrt{d_t^2 + (v_r - v_t)^2}$, where d_t represents the robot's distance to the edge of the circle, $v_r = 0.2$ m/s is a reference linear speed, and v_t is the current speed. The robot was localised using a particle filter on a prebuilt map. Internally, MPPI employed a simulated dynamics model $\hat{f}$ of the robot (Kozłowski and Pazderski 2004) for trajectory optimisation. The controller was configured with $M = 50$ rollouts and a time horizon $T = 400$. The episodes lasted 20 seconds, with the robot starting from a fixed initial position. The search space $\mathcal{S}$ for BO was defined by the intervals $\sigma_\epsilon \in [0.3, 0.5]$ and $\lambda \in [0.01, 0.21]$.

Figure 3.11b shows the learnt heteroscedastic noise model. Data from preliminary runs revealed that the noise in the episode rewards had a concentrated region of high variance in roughly the middle of the search space. As previously discussed, both the temperature λ and the control variance σ_ϵ^2 influence MPPI's exploration-exploitation trade-off. For this experiment, the bounds for σ_ϵ were chosen as settings that lead to acceptable performance in practice,

(a) Robot



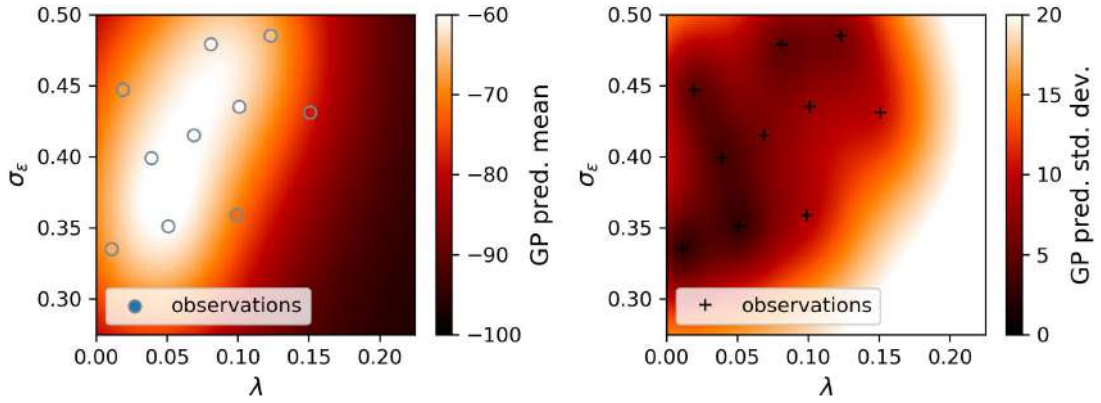
(b) Noise model



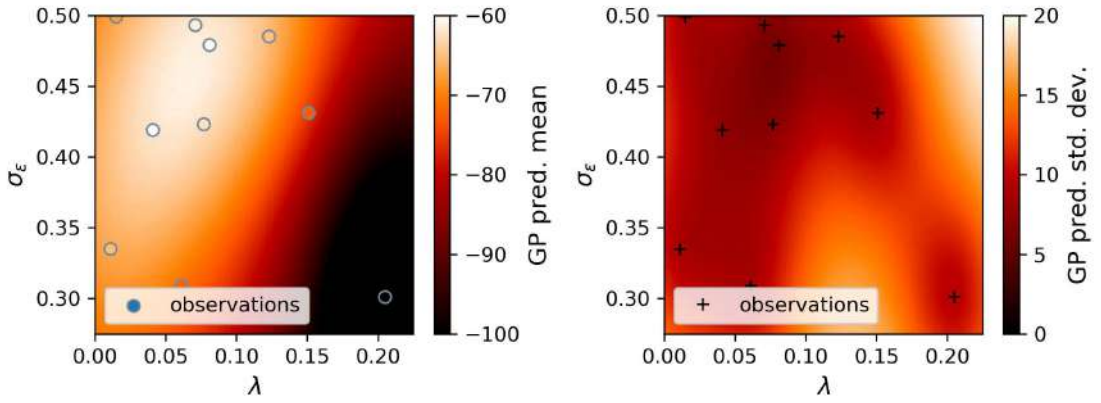
(c) Results

FIGURE 3.11. (a) Experiments with a physical robot known as Wombot, (b) the learnt heteroscedastic noise model and (c) the resulting performance for each BO method. The results were averaged over 3 independent trials for each algorithm, totalling 60 runs of MPPI in 20-second episodes on the robot.

but we allowed for a λ range that could cause instability. In the real robot, high temperatures λ cause excessive exploration in the action space of MPPI, which leads to an almost-sure failure in execution. Conversely, low temperatures force MPPI to take actions that are close to optimal, leading to mostly high rewards. The region at the centre appears to be optimal. However, that is the region where the robot's behaviour is unstable. MPPI's control variance σ_ϵ^2 contributes to this behaviour in a similar fashion by determining the spread of the exploration.



(a) Heteroscedastic GP



(b) Homoscedastic GP

FIGURE 3.12. GP models fit with data from one of the trials in the experiments with a physical robot.

To appropriately model the aforementioned noise concentration behaviour, we set the GP noise model from Equation 3.8 as a mixture of stationary kernels by defining the following:

$$\Phi(\mathbf{x}) := [k_q(\mathbf{x}, \mathbf{x}_1), \ldots, k_q(\mathbf{x}, \mathbf{x}_m)]^\mathsf{T} , \tag{3.10}$$

where the weights $\boldsymbol{\beta} \in \mathbb{R}^m$, the hyperparameters $\mathbf{x}_i \in \mathcal{S}$, alongside the other GP hyperparameters, were optimised offline by maximum a posteriori estimation. As reasonable choices for the priors, we set log-Gaussian priors for positive GP hyperparameters and Gaussian priors for the rest. As kernel k_q, we used a rational quadratic kernel (Rasmussen and Williams. 2006).

Performance results are presented in Figure 3.11c. We compared BO_{hetero} against BO_{homo}. As the results show, both algorithms are eventually able to find regions of high reward. However, due to its uniform noise model, BO_{homo} is led to a more exploratory behaviour instead of concentrating on promising regions, as evidenced by the query locations in Figure 3.12. Consequently, we observe a significant drop in performance during the optimisation, as shown in Figure 3.11c. In contrast, the heteroscedastic noise model allows BO_{hetero} to maintain steady performance improvements, which means lower tracking error with respect to the circular path specified by the cost function.

3.9 Summary

This chapter presented a framework for optimising stochastic MPC hyperparameters using BO with a parametric heteroscedastic noise model. A reward-based controller optimisation was proposed according to an expected cumulative reward formulation from RL. Then, a heteroscedastic noise behaviour was notably usual in the studied control problems when searching for optimal controller hyperparameters. Such heteroscedastic behaviour raised the question of how to model the noise. A heteroscedastic BO method was proposed using a parametric noise model that allowed the GP to correctly model the uncertainty that homoscedastic BO cannot handle. In the experiments section, The proposed approach was seen to outperform homoscedastic BO and non-BO approaches in most of the classic control and robotics problems considered. The experimental results were obtained using a flexible polynomial noise model and MPPI as a stochastic MPC controller. The experiments were only empirical and showed the effectiveness of Bayesian hyperparameter optimisation due to the significant performance improvement of the MPC control system and, therefore, in the robotic system.

CHAPTER 4

Adaptive Model Predictive Control under Model Parameter Uncertainty

4.1 Introduction

The previous chapter described a reward-based framework for optimising stochastic model predictive control (MPC) hyperparameters and also introduced a way to handle the heteroscedastic noise found in benchmark control problems. This chapter extends the framework by also addressing the reality gap problem with a type of domain randomisation. This chapter makes use of reinforcement learning (RL), control theory and Bayesian learning to build an adaptive MPC controller where the trajectory optimisation part is done through the randomisation of physical parameters. Due to the randomised setting, the controller is able to adapt to the real world, approximately solving the reality gap problem. Part of this chapter has been previously published and presented at ICRA[1].

Aiming to solve the reality gap problem when learning stochastic MPC controllers, this chapter derives a BO framework to obtain an optimal controller using a randomised dynamics model. This is done by defining distribution-based physical parameters. Therefore, the optimisation framework automatically estimates probability distributions over dynamics model parameters for optimal control solely based on performance data. This chapter makes the following contributions:

- a framework for optimising stochastic MPC while jointly estimating probability distributions of the dynamics model parameters, which is based only on observing rewards;

[1] Guzman, R., Oliveira, R., & Ramos, F. (2022b). Bayesian optimisation for robust model predictive control under model parameter uncertainty. *2022 International Conference on Robotics and Automation (ICRA)*, 5539–5545

- an analysis on whether capturing the uncertainty of dynamics model parameters leads to better performance;
- experimental results on benchmark simulated classic control and robotic problems.

The remainder of this chapter is structured as follows. Section 4.2 reviews relevant prior work in the areas of domain randomisation for robotics and stochastic MPC optimisation. Next, Section 4.3 establishes the difference between simulated data and real data and how randomisation helps overcome real uncertainties. Then Section 4.4 describes the dynamics model randomisation and the use of distribution-based physical parameters. Section 4.5 describes the adaptive MPC optimisation framework, considering that both the controller hyperparameters and dynamics model parameters are jointly optimised. The optimisation is reward-based, given that the expected cumulative reward is established as the performance measure. Then experiments in control and robotics problems are described in Section 4.6. The experimental results are analysed according to the expected cumulative reward, considering the uncertainty handled by the distribution-based physical parameters. Finally, Section 4.7 conveys a summary of the results obtained and the contributions of the chapter.

4.2 Related Work

This section reviews some relevant work on stochastic MPC and the reality gap problem in robotics applications.

4.2.1 Stochastic Model Predictive Control

To define an adaptive MPC controller, it is essential to know its use in a robotics context to relate it to the reality gap problem. Stochastic MPC methods have been successfully used in applications, from steady-state control to path planning in robotics (Pravitra et al. 2020). For example, in Carron et al. (2019), stochastic MPC, together with Gaussian processes, allows the system to adapt to disturbances using the GP to model uncertainty where little data is available for a robotic manipulator. This is because MPC is model-based, which means it

relies on a model of the system dynamics, which will be called simulated dynamics model. The main advantage of using physics and robotic simulation tools is undeniable because they make it possible to generate large amounts of data without much interaction with the real world (Cranmer et al. 2020; Todorov et al. 2012). Unfortunately, such a simulated dynamics model has to deal with the reality gap or sim-to-real problem because its robustness degrades when it is different from the true dynamics (Muratore et al. 2021a; Peng et al. 2018).

4.2.2 Domain Randomisation

Domain randomisation is a concept that comes from the machine learning (ML) area of transfer learning. However, in the context of robotics, two domains are taken into account: a simulated domain and a real-world domain (Forsberg 2022). Domain randomisation became a common alternative to deal with the reality gap since it consists of randomising simulators to expose the robot to different scenarios. For a manipulator grasping problem, (Tobin et al. 2017) performs domain randomisation regarding visual data for pose prediction, allowing deep neural networks to handle pose ambiguity for real-world scenarios. Conditions such as camera poses, scene layout, and wall textures can be randomised (Hsu et al. 2023).

Dynamics randomisation has been applied to find approximations for robust dynamics in common control and robotics problems. For example, there are adaptive dynamics models that address environment contexts where a robot's dynamics could change due to some robotic part malfunctioning (Lee et al. 2020). Other approaches propose inferring simulation parameters based on data instead of uniform parameter randomisation (Peng et al. 2018; Ramos et al. 2019).

Bayesian learning has also been applied to adapt distribution-based dynamics model parameters during learning, improving sim-to-real transfer for classic control problems (Oliveira et al. 2021). Domain randomisation has been used to adapt physics parameters to the real world as in Muratore et al. (2021a), where a Bayesian domain randomisation (BayRn) method was proposed as a way to adapt parameter distributions during learning. Semage et al. (2022) uses

BO to craft robust policies for simulated environments. None of the aforementioned adaptive methods directly account for heteroscedasticity in the reward function.

4.3 Real Data and Randomised Simulated Data

It can be understood that simulators do not precisely reflect reality, but simulated data is necessary to deal with the data availability problem. We start by describing the common assumptions in robotics and then introduce modifications to the simulated dynamics model previously denoted as $\hat{f}$. This chapter assumes that the state is fully-observable, so the robot can fully observe the state of the environment. Then, by following the model-based RL approach, the robot maintains a simplified model of the environment $\hat{f}$. The traditional way to describe the robot behaviour is with a deterministic dynamics model that follows the Markov decision process (MDP) assumption seen in Equation 2.67, but here we also use the usual control systems notation. The simulated dynamics model of the robot is defined as

$$s_{t+1} = \hat{f}(s_t, a_t) \, , \tag{4.1}$$

where the state of the robot $s \in \mathcal{S}$ lies in a continuous state space $\mathcal{S}$, and $a \in \mathcal{A}$ is an action specified by a continuous action space $\mathcal{A}$. Upon the execution of an action, the system transitions to the next state s_{t+1}, producing an instant reward measured by $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that measures the system performance at a given state and action.

In order to improve sim-to-real transfer from the simulated domain to the real-world domain, the proposed framework considers approximating the simulated dynamics model $\hat{f}$ to the real dynamics model f by making the controller observe data from randomised versions of $\hat{f}$. Such data can be thought of as the data coming from interactions with the environment, which are data tuples $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$. Real data can be collected due to real robot interactions coming from f, while simulated data is collected from a robotics simulator or from the simulated dynamics model $\hat{f}$. As briefly mentioned in Section 1.1.3, there are two main domains: one that outputs real data and one that outputs simulated data, as shown in Figure 4.1. Now, simulated data are

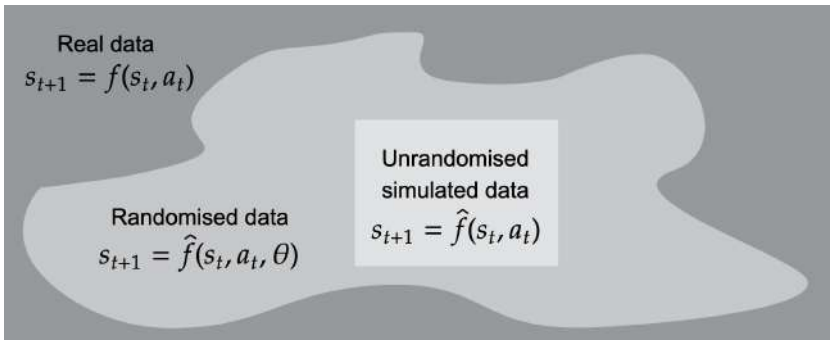


FIGURE 4.1. Data domains. Random simulated data are able to approximate real data.

considered unrandomised and would only cover a very small portion of the space. However, by using randomised data, the robot would cover more data from the real data domain.

4.4 Dynamics Model Randomisation

Randomised data is obtained by introducing a randomised parameter θ to $\hat{f}$. The proposed framework follows a domain randomisation approach (Peng et al. 2018) where randomised dynamics are used to train policies in simulation. Randomising dynamics corresponds to randomising physical parameters. For instance, in the Pendulum problem from OpenAI Gym that consists of swinging the pole up as in Figure 4.2, two main physical parameters are the pole mass with true value $m = 1.0$ and its length with true value 1.0. In order to deal with uncertainties regarding the physical parameters, each one is defined as a random variable. For example, the parameter mass $\theta = m$ could be beta-distributed then $m \sim \text{Beta}(\alpha, \beta)$ with a probability distribution from Figure 4.3b. Considering gamma-distributed mass and length. Figure 4.4

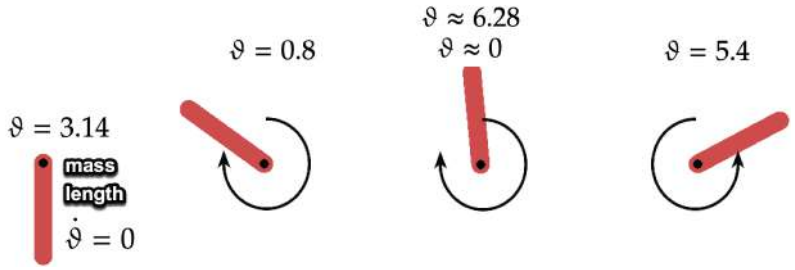
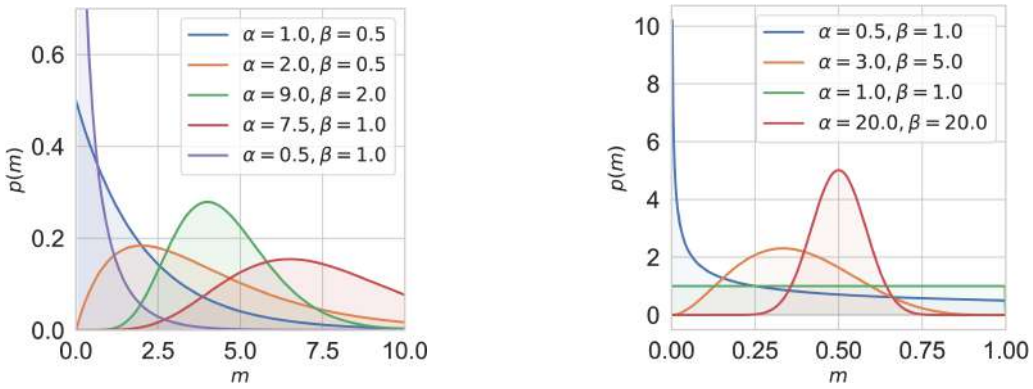


FIGURE 4.2. Pendulum problem where the goal is to swing up a pole from $\vartheta = 3.14$ degrees to $\vartheta = 6.28$ or $\vartheta = 0$ according to the swing direction.



(a) Gamma distributions (b) Beta distributions

FIGURE 4.3. Examples of probability distributions for different distribution parameters. α and β are the parameters for gamma and beta-distributed variables.

shows expected cumulative rewards for several possible distribution mean μ and standard deviation σ. The regions where $\mu = 1$ and $\sigma \approx 0$ are where the mass and length are close to their true values. Notice how those regions in red tend to obtain the highest expected cumulative rewards.

In the context of the MPC controller, randomisation can be done when optimising trajectories since it is where the interaction with the simulated dynamics model is done. We define a random vector of b dynamics model parameters $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_b]$ to be used at each interaction with the simulated dynamics model. Each dynamics model parameter follows a probability distribution parameterised by certain parameters denoted as $\boldsymbol{\psi}_i$ for $i = 1, \ldots, b$. For example, the pendulum mass is parameterised by $\boldsymbol{\psi} = \{\mu_m, \sigma_m\}$. Then at state $\mathbf{s}_t$ the following calculations are performed:

$$\theta_1 \sim p_{\theta_1}(\theta_1; \boldsymbol{\psi}_1) \tag{4.2}$$

$$\vdots \tag{4.3}$$

$$\theta_b \sim p_{\theta_b}(\theta_b; \boldsymbol{\psi}_b) \tag{4.4}$$

$$\mathbf{s}_{t+1} = \hat{f}\left(\mathbf{s}_t, \mathbf{a}_t, [\theta_1, \ldots, \theta_b]\right) \ . \tag{4.5}$$

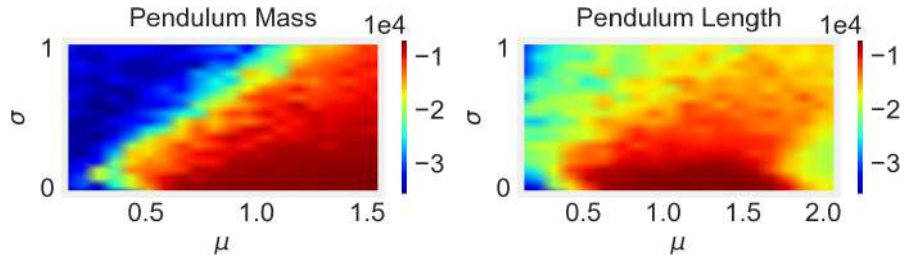


FIGURE 4.4. Expected cumulative rewards for different model parameter distributions. The regions in red correspond to regions where the controller achieved the highest expected cumulative rewards.

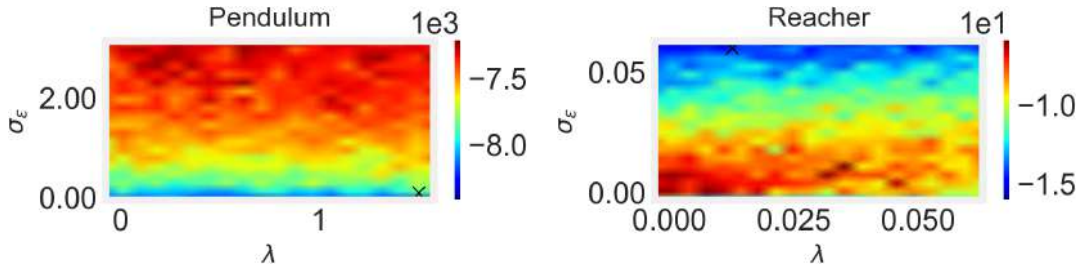


FIGURE 4.5. Expected cumulative rewards for MPPI hyperparameter combinations.

4.5 Adaptive Model Predictive Control

As mentioned before, in this chapter, the proposed framework does not directly aim at finding parameters that match the observed dynamics as it is done for system identification (Romeres et al. 2016). Instead, it adapts model parameter distributions to the controller, which means that those resulting distributions may not be close to their true values. They are used for optimising the controller. We make sure that we are using the right MPC hyperparameters by optimising them together with the dynamics model parameters. In the case of the MPPI controller described in Section 2.1.6, the hyperparameters are the temperature λ, control noise σ_ϵ, horizon T, and the number of rollouts M. They can be collectively described as ϕ. In this chapter, we only work with λ and σ_ϵ as in Chapter 3. The hyperparameter search spaces can be seen in Figure 4.5, which shows expected cumulative rewards for grid search hyperparameter combinations after running the controller for the Pendulum and Reacher problems.

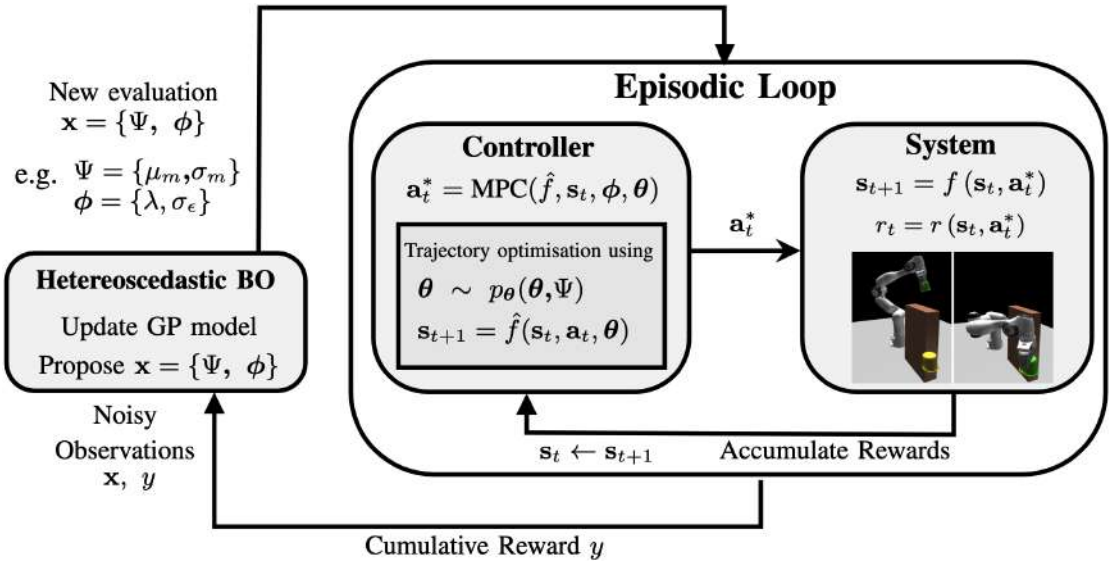


FIGURE 4.6. Overview of the adaptive MPC optimisation framework proposed.

Considering a realistic noisy cumulative reward function as previously seen in Chapter 3, the noisy black-box function y is defined as

$$g : \mathcal{X} \to \mathbb{R} \tag{4.6}$$

$$y(\mathbf{x}) = g(\mathbf{x}) + \nu(\mathbf{x}) , \tag{4.7}$$

where we can only maximise an approximation of the expected cumulative reward $\hat{g} \approx g$, where y is a cumulative reward variable, and the input $\mathbf{x}$ consists of the collection of distribution parameters $\boldsymbol{\Psi}$ that parameterise the model parameters $\boldsymbol{\theta}$, and the controller hyperparameters $\boldsymbol{\phi}$. Therefore, we define $\mathbf{x} = \{\boldsymbol{\Psi}, \boldsymbol{\phi}\}$, and the optimisation problem is described as follows:

$$\boldsymbol{\Psi}^\star, \boldsymbol{\phi}^\star = \underset{\{\boldsymbol{\Psi}, \boldsymbol{\phi}\}}{\text{argmax}}\ \hat{g}(\boldsymbol{\Psi}, \boldsymbol{\phi}) . \tag{4.8}$$

In order to handle noisy heteroscedastic observations, we maximise the expected cumulative reward as performance measure $\hat{g}$ with the heteroscedastic BO method described in Chapter 3. The framework computes the BO posterior inference $g_* \mid \mathbf{x}_*, \mathbf{X}, \mathbf{y} \sim \mathcal{N}(\text{mean}(g_*), \text{cov}(g_*))$ where $\mathbf{X}$ and $\mathbf{y}$ are observations , and $(\mathbf{x}_*, g_*)$ are new optimal points inferred by BO. This way, we are able to find optimal MPC hyperparameters adapted to the simulated dynamics model. A general overview of the adaptive MPC framework is shown in Figure 4.6. It shows how BO optimises $\mathbf{x}$ by receiving the accumulated reward y, which is averaged to obtain the expected

cumulative reward $\hat{g}$. The robotic task presented in the figure is a manipulator called Franka, which will be described in the experiments section.

4.6 Experiments

In order to test the performance of the proposed adaptive MPC framework, some experiments are evaluated from simulated control problems and to robotic tasks. The first subsection specifies the tasks, the randomised dynamics model, and the controller settings used in the experiments. Next, the framework is evaluated by comparing it with other optimisation methods to support the use of adaptive MPC. Finally, some experiments with robotic tasks are presented.

4.6.1 Simulators and Parameterisation

The proposed adaptive MPC was evaluated by solving OpenAI Gym benchmark tasks: Cartpole, Pendulum, Half-Cheetah, Reacher, and Fetchreach with the dense reward functions shown. The same reward functions used in previous chapters are used to compute instant rewards. The additional simulated environment is Fetchreach shown in Figure 4.7. Fetchreach consists of a robotic manipulator with seven degrees of freedom and a paddle gripper as the end effector. The objective is to move the gripper to a 3-dimensional target position, and the dense reward function measures the Euclidean distance from the gripper to the target position. First of all,

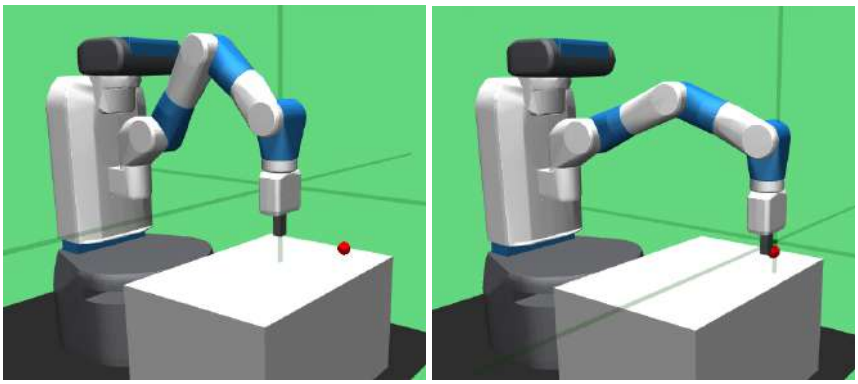


FIGURE 4.7. Fetchreach. Manipulator reaching task.

different tasks have different constraints for model parameters $\boldsymbol{\theta}$. Some parameters have to be

positive, like the mass and length, and others can be defined only within a specific range. We considered using a probability distribution with positive support. The beta distribution's support is $[0, 1]$, which is inconvenient since one has to scale sampled model parameters if values outside the range are needed. We also need to specify ranges for each distribution parameter $\boldsymbol{\psi}$. For example, if we define dynamics model parameters as gamma-distributed, we need to specify ranges for their parameters α and β as in Figure 4.3.

Rather than using the distribution parameters α and β for better visualisation and interpretability, the distribution mean μ and standard deviation σ are used to represent each distribution-based variable. α and β can be derived given μ and σ. We use gamma distributions for the simplicity of such conversion. Each model parameter is assumed to be gamma-distributed, and the following calculation is applied:

$$\theta \sim \text{Gamma}(\alpha, \beta) \tag{4.9}$$

$$\alpha = \frac{\mu^2}{\sigma^2} \tag{4.10}$$

$$\beta = \frac{\mu}{\sigma^2} \,. \tag{4.11}$$

The stochastic controller used here was MPPI Section 2.1.6, where the dynamics randomisation was done at each rollout when interacting with the simulated dynamics model. Considering that there are distribution parameters $\boldsymbol{\psi}_i$ for each dynamics model parameter θ_i for $i = 1, \ldots, b$, dynamics model randomisation is performed as follows

$$\theta_1 \sim p_{\theta_1}(\theta_1; \boldsymbol{\psi}_1) \tag{4.12}$$

$$\vdots \tag{4.13}$$

$$\theta_b \sim p_{\theta_b}(\theta_b; \boldsymbol{\psi}_b) \tag{4.14}$$

$$\mathbf{s}_{j+1} = \hat{f}\left(\mathbf{s}_j, \mathbf{a}_j + \boldsymbol{\epsilon}_j^{(m)}, [\theta_1, \ldots, \theta_b]\right) \,, \tag{4.15}$$

where j is the current timestep when sampling a trajectory, m is the current sampled trajectory, and $\mathbf{a}_j + \boldsymbol{\epsilon}_j^{(m)}$ is a vector of perturbed actions. After performing MPPI's trajectory optimisation steps, the next optimal action found by MPC is sent to the system actuators.

Next, a search space can be defined for each parameterisation ψ_i, and the controller hyperparameters $\phi = \{\lambda, \sigma_\epsilon\}$ by narrowing down large enough intervals until noticeable optimal regions are found. Then, BO$_{hetero}$ can be used to perform the expected cumulative reward optimisation from Equation 4.8 where $\mathbf{\Psi}$ and ϕ are jointly optimised. Therefore, the optimisation method consists of processing data tuples $(\mathbf{x}, \hat{g})$ where the input is defined as $\mathbf{x} = \{\mathbf{\Psi}, \phi\}$, and the output is the expected cumulative reward $\hat{g}$.

4.6.2 Experiments in Classic Control Tasks

The objective is to find optimal hyperparameter settings ϕ together with model distribution parameters $\mathbf{\Psi}$. For Cartpole and Pendulum, μ_l and μ_m are the mean pole length and mass. For Half-Cheetah and Reacher, we consider the scaling factor κ as a random variable with mean and standard deviation denoted by subscripts. κ multiplies the model parameters, e.g. κ with mean $\mu = 1.0$ and $\sigma \to 0$ means that the model parameter corresponds exactly to its true value. κ_m and κ_d are the scaling factors for the masses and damping ratios for Half-Cheetah and Fetchreach, and κ_{d1} and κ_{d2} are scaling factors for two Reacher damping ratios. Regarding the true parameter values, for Cartpole, the true mass and pole length were both 0.5, and for Pendulum, they were 1.0. For the rest, a scaling factor of 1.0 corresponds to the parameter's true value.

Problem	n_e	T	M	Distribution parameter range	
Cartpole	40	10	250	$\mu_l \in [0.2, 1.5]$ $\mu_m \in [0.1, 1.5]$	$\sigma_l \in [1e-5, 0.1]$ $\sigma_m \in [1e-5, 0.1]$
Pendulum	15	20	400	$\mu_l \in [0.2, 2.0]$ $\mu_m \in [0.2, 2.0]$	$\sigma_l \in [1e-5, 0.1]$ $\sigma_m \in [1e-5, 0.1]$
Half-Cheetah	25	14	10	$\kappa_{m,\mu} \in [0.6, 1.2]$ $\kappa_{d,\mu} \in [0.6, 1.4]$	$\kappa_{m,\sigma} \in [1e-5, 0.1]$ $\kappa_{m,\sigma} \in [1e-5, 0.1]$
Reacher	20	12	18	$\kappa_{d1,\mu} \in [0.1, 8.0]$ $\kappa_{d2,\mu} \in [0.1, 8.0]$	$\kappa_{d1,\sigma} \in [0.001, 0.6]$ $\kappa_{d2,\sigma} \in [0.001, 0.6]$
Fetchreach	120	3	12	$\kappa_{d,\mu} \in [1.0, 50.0]$	$\kappa_{d,\sigma} \in [0.001, 0.6]$

TABLE 4.1. Search spaces for the control tasks. Intervals for the dynamics model parameters and scaling factors.

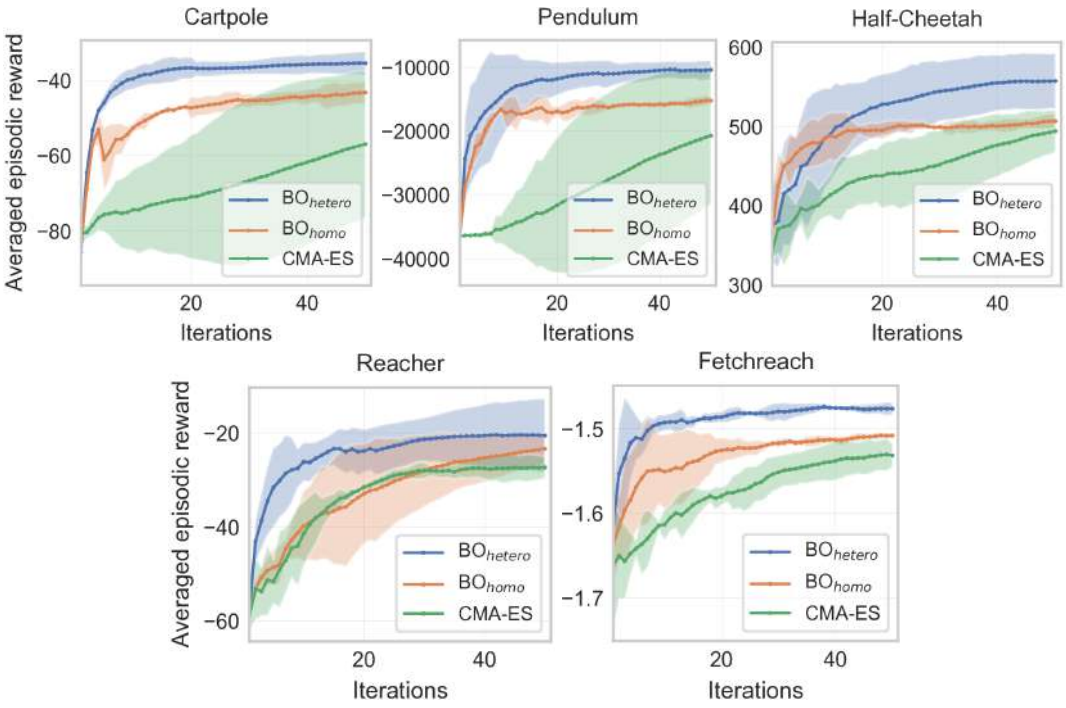


FIGURE 4.8. Expected cumulative rewards per iteration. The shaded areas denote two standard deviations. Each method started at a point with a minimum expected cumulative reward.

For the controller hyperparameters, search spaces within $\lambda \in [1e-5, 2.5]$, $\sigma_\epsilon \in [1e-5, 4.0]$. The search spaces for the distribution parameters are shown in Table 4.1. The number of timesteps was $n_s = 200$ for every task. Results in Figure 4.8 show expected cumulative rewards for 50 iterations by running both BO versions and covariance matrix adaptation evolution strategy (CMA-ES) (Arnold and Hansen 2010), which was configured as in the previous chapter. Each BO method has a GP model trained by optimising the log-marginal likelihood Equation 2.47 with a batch of 150 observations $(\mathbf{x}, \hat{g})$ to obtain optimal kernel settings. We used an exponential kernel for both BO versions. The log-marginal likelihood optimisation was done by using the multi-start method known as multi-level single-linkage (MLSL) Algorithm 6. For both BO versions, we used the exponential kernel defined in Equation 2.43. The expected cumulative rewards were scaled to $[0, 100]$, and the input variable $\mathbf{x}$ values were scaled to $[0, 5]$ in order to make the GP training easier and allow the use of the squared exponential kernel with a single lengthscale ℓ. We set $\Omega := \{\sigma_\nu, \sigma_n, \ell\}$ for homoscedastic BO (BO_{homo})

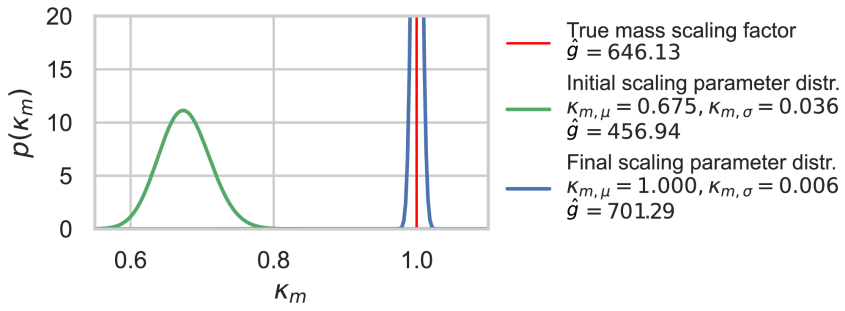


FIGURE 4.9. Mass parameter scaling factor κ_m for Half-Cheetah.

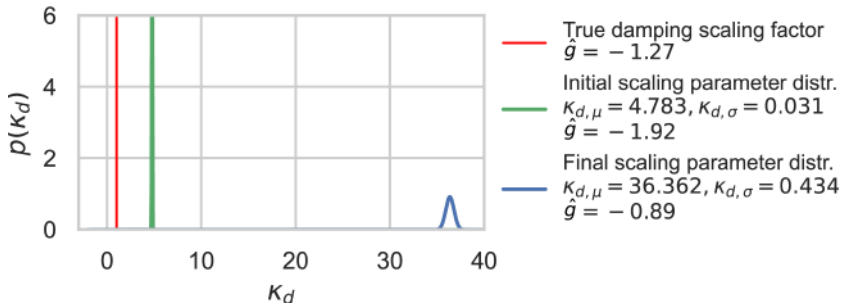


FIGURE 4.10. Optimised scaling factor κ_d for Fetchreach.

and $\Omega := \{z, \sigma_n, \ell\}$ for heteroscedastic BO (BO$_{hetero}$). For both BO versions, we used a UCB acquisition function defined in Equation 2.60 with $\delta = 2.0$ optimised with the same MLSL method.

We can see that BO$_{hetero}$ can outperform the others due to the noisy nature. Each optimisation starts at points that give a minimum reward. On average, adaptive MPC with BO$_{hetero}$ can find optimal $\boldsymbol{\Psi}$ and $\boldsymbol{\phi}$ in fewer iterations and, in most cases, with less variance.

4.6.3 Adapted Parameter Distributions

For Half-Cheetah, the optimised distribution of the scaling factor of the mass κ_m yields an expected cumulative reward of 701.29, which is higher than the reward obtained with the true value. This can be explained because we optimise the reward to optimise the controller and not for system identification. In Figure 4.9, we show the expected cumulative reward of using the true parameters and the optimised distribution. The estimated optimal parameter distribution shows the approximation to the true value. In the case of Fetchreach, the damping ratio plays

a major role. Even wrong dynamics model parameters can lead to better performance. In Figure 4.10, we start at an initial parameter distribution denoted in blue for the damping ratio κ_d that gives a maximum reward found via random search. Then, we use adaptive MPC with BO_{hetero} to optimise the parameter distribution. We found that a high damping ratio allows faster movements, improving reward performance with the true model. That is because, for an overdamped model, the controller would be prone to apply higher torques by choosing a larger control variance σ_ϵ. However, that leads to uncertain arm movements when the gripper is close to the target. Similar behaviour occurs for Reacher, but it is not so noticeable since the task is simpler.

4.6.4 Experiments in a Robotic Simulator

We used a simulated Franka robotic arm[2] with the task of reaching a yellow target with a gripper in a single-obstacle environment shown in Figure 4.11, which was implemented only for experiments with control systems. The MPPI-based motion planning framework from Bhardwaj et al. (2022) was used. MPPI trajectory optimisation performed GPU-based trajectory sampling since we used the STORM[3] simulator adapted to have only one obstacle and be reward-based as in the proposed adaptive MPC framework. We defined the number of episodes $n_e = 6$, number of timesteps $n_s = 480$, horizon length $T = 20$, number of trajectories to sample $M = 150$, and the control variance $\sigma_\epsilon = 0.5$. Having defined the reaching task and the simulator, the variables to be randomised need to be defined. In this case, the obstacle dimension sizes are randomised for the simulated dynamics model $\hat{f}$. We adapt the controller hyperparameter $\phi = \lambda \in [0.01, 2.0]$ to the environment by defining distribution-based length x, width y and height z defined by $\mathbf{\Psi} = \{(x_\mu, x_\sigma), (y_\mu, y_\sigma), (z_\mu, z_\sigma)\}$ with true values (0.3, 0.1, 0.6) respectively. The distribution parameters for the dynamics model parameters are shown in Table 4.2.

[2]IssacGym: https://developer.nvidia.com/isaac-gym
[3]STORM: https://github.com/NVlabs/storm

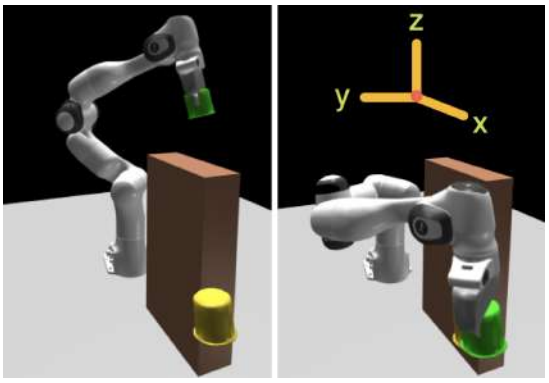
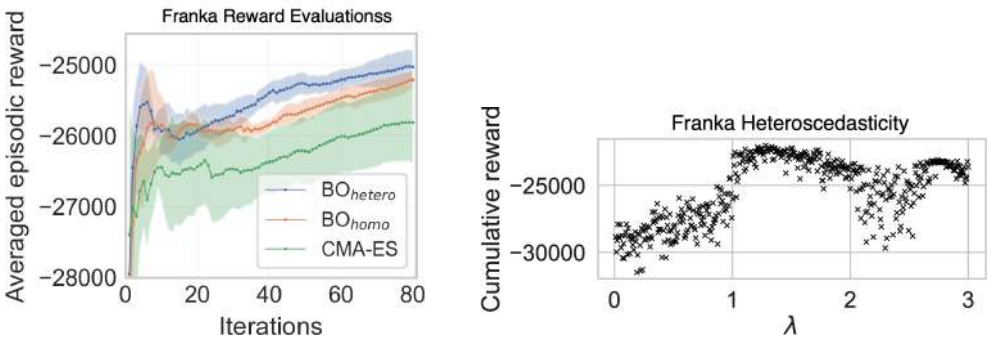


FIGURE 4.11. Franka manipulator reaching task in a single-obstacle environment.

Length	Width	Height
$x_\mu \in [0.3, 0.32]$ $x_\sigma \in [0.001, 0.05]$	$y_\mu \in [0.1, 0.12]$ $y_\sigma \in [0.001, 0.01]$	$z_\mu \in [0.6, 0.62]$ $z_\sigma \in [0.001, 0.03]$

TABLE 4.2. Search spaces for the Franka manipulator. Intervals for the dynamics model parameters and scaling factors.

A starting point for the parameter distributions was set at a point where the expected cumulative reward was minimum, using collected observations. As with the control problems, each BO method has a GP model trained by optimising the log-marginal likelihood with 500 collected observations $(\mathbf{x}, \hat{g})$, but with a squared exponential kernel from Equation 2.44 that allows optimising a lengthscale for each input feature without needing to do scaling. Then in order



(a) Learning curves (b) Heteroscedastic behavior

FIGURE 4.12. Learning curves when optimising Franka (a), and the heteroscedastic behaviour of the search space (b).

to observe the performance of the adaptive MPC framework, BO_{homo}, BO_{hetero}, and CMA-ES were compared. Figure 4.12a shows the expected cumulative rewards obtained against the number of iterations for each method. Each method was run for 80 iterations. The heteroscedastic behaviour for λ is shown in Figure 4.12b, which is what is exploited by BO_{hetero}. CMA-ES was able to optimise the noisy optimisation problem as in the previous chapter. However, it is still outperformed by BO_{homo} and BO_{hetero} by doing more exploration but still getting stuck in local optima. Using the BO methods, the expected cumulative reward is maximised while exploiting and exploring new regions, so the optimal dynamics model parameter distributions are not necessarily found at the last iteration.

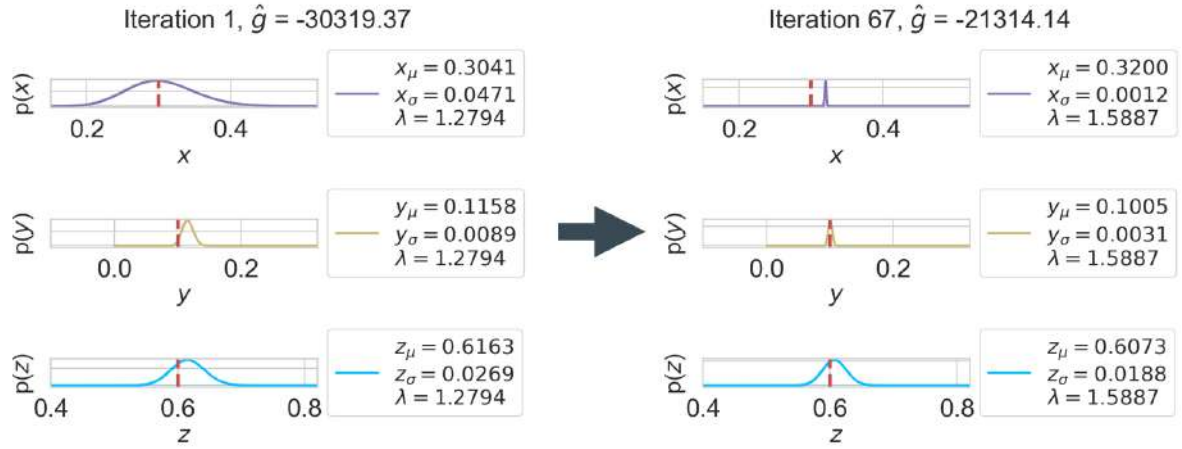


FIGURE 4.13. Initial distributions for Franka (left) and best inference found by BO_{hetero} (right).

To evaluate if the optimisation proposed converges to an optimum, Figure 4.13 shows the optimum setting found by BO_{hetero}, starting from initial distributions until convergence, which in this case is where distributions are close to the true values. This happens because a greater obstacle width presents a more challenging scenario for the robot to maneuver around, while a lower width increases the risk of collisions. Consequently, more accurate knowledge of the obstacle's width is essential for devising effective trajectories. Meanwhile, the length x and height z of the obstacle are often less critical for path planning, as they do not typically constrain the robot's immediate movement space in the same way. The results imply that, by maximising the expected cumulative reward, better optimal regions can be found by randomising the dynamics model parameters as proposed.

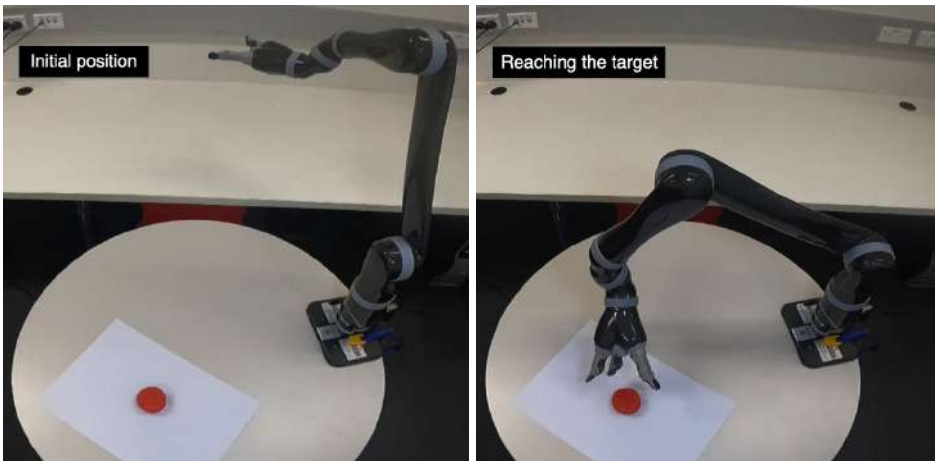


FIGURE 4.14. Jaco manipulator reaching task.

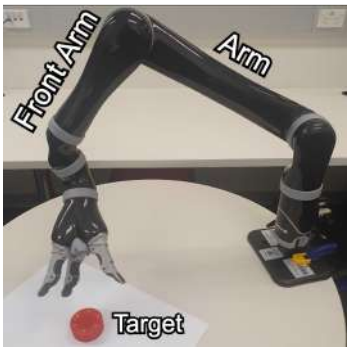


FIGURE 4.15. Jaco arm components.

Arm Length	Front Arm length
$d_{1,\mu} \in [0.3100, 0.5100]$ $d_{1,\sigma} \in [0.001, 0.03]$	$d_{2,\mu} \in [0.1073, 0.3073]$ $d_{2,\sigma} \in [0.001, 0.03]$

TABLE 4.3. Search spaces for the Jaco manipulator. Intervals for the dynamics model parameters.

4.6.5 Experiments with a Real Robot

We configured a reaching task using a Kinova Jaco[2] manipulator shown in Figure 4.14. Jaco[4] has 6 degrees of freedom because of 6 joints connecting rigid links that allow versatile arm movements. The objective is to reach the red target in a fixed location, always starting at the predefined initial position. Assuming a fully-observable environment, we define a state space that consists of the 6 joint angles, the gripper location, and the target location, while the action space consists of the 6 joint angles.

The same adaptive MPC framework used for the control tasks was used in this case. We defined the number of episodes $n_e = 8$, number of timesteps $n_s = 1$, horizon length $T = 20$, number

[4]Specification guide: https://assistive.kinovarobotics.com/uploads/EN-UG-007-Jaco-user-guide-R05.pdf

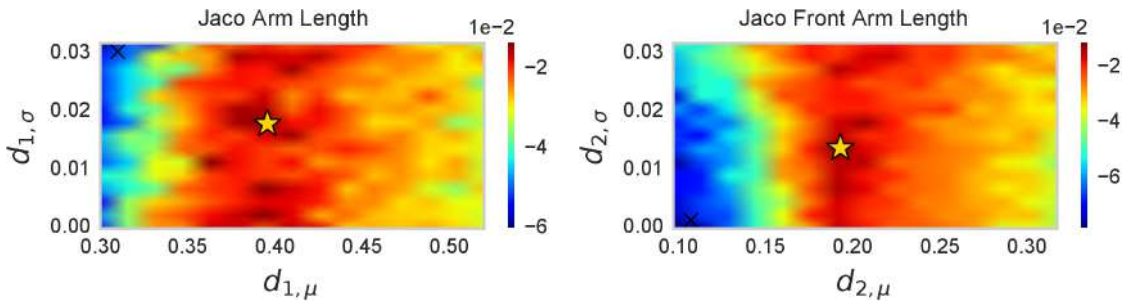
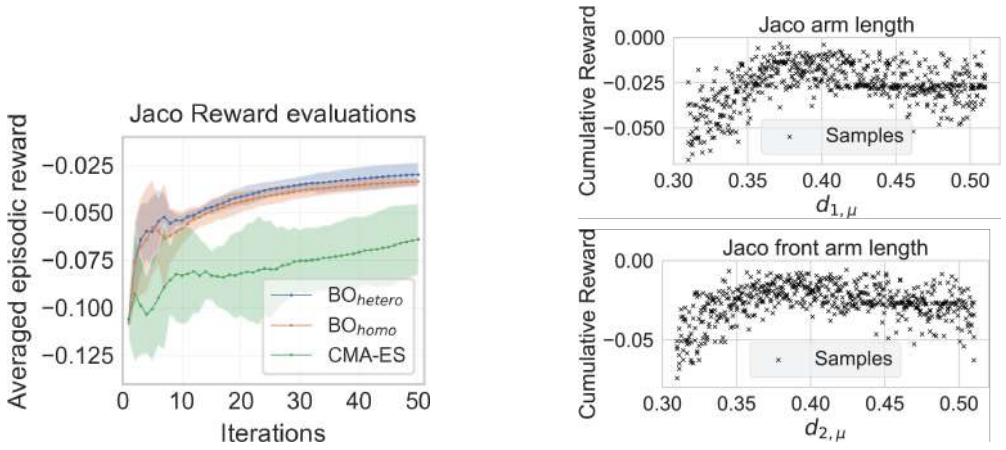


FIGURE 4.16. Expected cumulative rewards by running the Jaco reaching task for several dynamics model parameter values.

of trajectories to sample $M = 150$, control variance $\sigma_\epsilon = 1.0$, and temperature $\lambda = 0.01$. The reward was computed as the negative distance from the gripper to the target. The gripper position was obtained via forward kinematics, given the joint angles. The experiments with the Jaco robot consider uncertainty about the length of the arm and the front arm shown in Figure 4.15. The arm length d_1 and the front arm length d_2 with true values $(0.4100, 0.2073)$ are defined as randomised variables parameterised by $\mathbf{\Psi} = \{(d_{1,\mu}, d_{1,\sigma}), (d_{2,\mu}, d_{2,\sigma})\}$. The objective now is to adapt them to the fixed MPC hyperparameters $\{\lambda, \sigma_\epsilon\}$. The search space was defined as intervals of ± 0.10 the true lengths, and standard deviations $d_{1,\sigma}, d_{2,\sigma} \in (0.001, 0.03)$.

The search spaces for each distribution-based variable can be seen in Figure 4.16, which shows expected cumulative rewards for grid search combinations after running the MPPI controller with other variables fixed. The regions in red correspond to optimal distributions. Each BO method has a GP model trained by minimising the log-marginal likelihood with 400 collected observations $(\mathbf{x}, \hat{g})$, and with the squared exponential kernel from Equation 2.44. In order to compare the optimisation methods, Figure 4.12a shows the expected cumulative rewards obtained against the number of iterations for each method. Each method was run for 50 iterations. The noisy behaviour of $d_{1,\mu}$ and $d_{1,\sigma}$ is shown in Figure 4.17b, which is what is exploited by BO_{hetero}. In this case, both BO methods optimise the controller similarly but still better than CMA-ES. The proposed optimisation converges to an optimum as in Figure 4.18. It shows the optimum setting found by BO_{hetero} for the Jaco reaching task. The distributions get close to the true values in this case. Optimising point estimates is out of the scope of this work. Both BO versions inferred similar optimal distributions.



(a) Learning curves (b) Noisy behaviour

FIGURE 4.17. Learning curves when optimising Jaco (a), and the heteroscedastic behaviour of the search space (b).

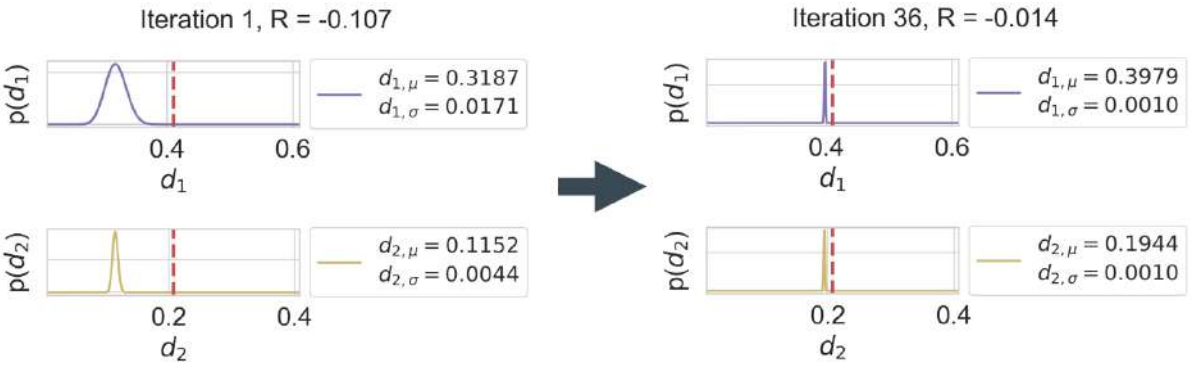


FIGURE 4.18. Initial distributions for Jaco (left) and best inference found by BO$_{hetero}$ (right).

4.7 Summary

This chapter presented an extension to the heteroscedastic BO framework proposed in Chapter 3 for dealing with the reality gap problem in robotics. It started by separating real data and simulated data in order to come up with a solution to make simulated data closer to real data. Then, a randomised dynamics model is formulated by introducing distribution-based physical parameters to the dynamics model. The optimisation framework is introduced as reward-based, where the expected cumulative reward is the performance measure. Finally, the proposed adaptive MPC optimisation framework was utilised where heteroscedastic BO was able to perform

as well or better than the traditional BO in simulated environments and a real robotic manipulator. The manipulator tasks optimised made use of two robots: Franka and Jaco. Results showed that having parameter and object dimension distributions can lead to improved performance in a few iterations.

CHAPTER 5

Adaptive Model Predictive Control by Learning Classifiers

5.1 Introduction

In the previous chapter, a robust framework optimising MPC was developed where robustness comes from robustness to noisy heteroscedastic observations and robustness in terms of adaptability to real-world settings. This chapter deals with the data availability problem but regarding the optimisation method. The adaptive model predictive control (MPC) framework described in Chapter 4 used Bayesian optimisation (BO) to perform controller hyperparameter optimisation because of its data efficiency advantage. However, BO's main problem was that its Gaussian process (GP) surrogate model needed to be optimised with a dataset collected from interactions with the real robot. This chapter proposes a way to optimise the controller without needing to train the surrogate model. The method and experiments are part of a research paper published at L4DC[1].

An alternative to reducing interactions with the real world is to make use of a simulated dynamics model for trajectory optimisation. However, modelling an accurate dynamics model inevitably leads to errors. Even so, by using data-driven approaches, it is possible to reduce the error produced in a real-world setting. This chapter is focused on the optimisation method to help solve the data availability problem. Instead of using BO, this chapter evaluates the use of alternative surrogate-based optimisation methods, specifically, one that makes use of data classification to build a surrogate model: Bayesian optimisation by ratio estimation (BORE). This chapter makes the following contributions:

[1] Guzman, R., Oliveira, R., & Ramos, F. (2022a). Adaptive model predictive control by learning classifiers. *Proceedings of The 4th Annual Learning for Dynamics and Control Conference*, *168*, 480–491

- a framework to optimise stochastic MPC by learning classifiers;
- evaluations of the proposed framework when there is heteroscedastic noise; and
- experimental results on benchmark continuous control problems in simulated environments.

The remainder of this chapter is structured as follows. Section 5.2 reviews relevant prior work in the areas of stochastic MPC, MPC optimisation, and surrogate-based optimisation. Section 5.3 provides a comparison between BO-based methods and alternative surrogate-based optimisation methods. Next, Section 5.4 introduces the BORE as the surrogate-based method used for optimisation. A way to deal with the noise is also proposed. Section 5.5 focuses on the classification problem and lists the problems regarding classification. This is done to correctly choose a binary classifier. Then Section 5.6 presents the MPC optimisation problem that consists of optimising both the controller hyperparameters and the dynamics model parameters. Section 5.7 introduces the classification part to the optimisation framework. The experiments are described in Section 5.8. Finally, Section 5.9 conveys a summary of the results obtained and the contributions of the chapter.

5.2 Related Work

This section reviews some relevant work on stochastic MPC and the reality gap problem in robotics applications.

5.2.1 Stochastic Model Predictive Control

As mentioned in Chapter 4, the proposed framework is particularly focused on a particular stochastic MPC controller known as model predictive path integral control (MPPI), which was introduced in Williams et al. (2016) as a sampling-based controller. There is a recent robust variation denominated robust model predictive path integral control (RMPPI) proposed in Gandhi et al. (2021), which alleviates issues of MPPI related to the lack of robustness to unknown external disturbances. Besides MPPI's use for autonomous vehicle driving, it has

also recently been used for dexterous manipulation tasks such as the task of learning to spin a pen between the fingers of a robotic hand (Charlesworth and Montana 2021), and for Franka manipulation tasks where online control and adaptation is necessary (Abraham et al. 2020).

5.2.2 Model Predictive Control Optimisation

Among the recent research on MPC optimisation, there is Sorourifar et al. (2021), where a constrained variant of BO is used to optimise stochastic MPC formulations dependent on dynamics model parameters. That approach is able to satisfy state and output constraints such as robot physical limitations and visibility constraints. Edwards et al. (2021) proposes a method to jointly optimise the data-driven system identification, task specification, and control synthesis of unknown dynamical systems for automatic MPC tuning. MPPI hyperparameters such as the horizon and the number of trajectories are optimised using a BO-based method known as sequential model-based algorithm configuration (SMAC) (Hutter et al. 2011), which uses tree-based models forest to predict the performance of configurations.

5.2.3 Surrogate-Based Optimisation

Regarding surrogate-based optimisation methods, tree-structured parzen estimator (TPE) as a tree-based method has been compared against BO for black-box optimisation (Turner et al. 2021). TPE has been used for optimising CMA-ES hyperparameters in a noiseless setting (Zhao and Li 2018), and it has also been used for multi-objective optimisation in computationally expensive optimisation problems in Ozaki et al. (2020). Something particularly important about it is that the quantile hyperparameter is empirically investigated, and results in benchmark problems indicate that a quantile hyperparameter close to zero provides more optimal results. Meanwhile, Bayesian optimisation by density ratio estimation (BORE), which is an optimisation problem highly related to TPE, has been proposed and compared against other optimisation methods in Tiao et al. (2021), where it performs decently against the traditional BO. More research on BORE includes Oliveira et al. (2022), where a variation denoted as BORE++ addressed BORE's suboptimal performance by quantifying uncertainty.

5.3 Surrogate-Based Optimisation Problem

This chapter provides a variation to the adaptive MPC framework proposed in Chapter 4. The main issue to solve about the framework is the need for an initial dataset to train the GP hyperparameters, which goes against the data availability problem in robotics. The proposed variation has the objective of dealing with such an issue while also handling a search space with heteroscedastic behaviour.

As in Chapter 3, we start by defining the noisy minimisation problem

$$y = g(\mathbf{x}) + \nu \qquad \text{where } \nu = \sigma_\nu(\mathbf{x}) \tag{5.1}$$

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathcal{X}}{\operatorname{argmin}}\, \mathbb{E}[g(\mathbf{x})]\ , \tag{5.2}$$

where only an expectation of the true function can be approximated, which is denoted as $\hat{g} = \mathbb{E}[g(\mathbf{x})]$. In Chapter 4, the method used for solving such an optimisation problem was BO, which is a surrogate optimisation method described in Section 2.5. BO possesses major advantages when used to optimise costly objective functions. BO places a prior over the space of functions and combines it with noisy samples to produce an approximation to the unknown function g. BO can also approximate $\hat{g}$ by using the GP predictive posterior mean.

The key component for the effectiveness of BO is the use of an acquisition function α that guides the search for the optimum. That acquisition function acts like a surrogate model that receives the GP model $\mathcal{M}$ that fits the true function mean and is maximised as follows:

$$\mathbf{x}_t \leftarrow \underset{\mathbf{x}\in\mathcal{S}}{\operatorname{argmax}}\, \alpha(\mathbf{x}, \mathcal{M}_\Omega, \mathcal{D})\ , \tag{5.3}$$

which corresponds to the step where the method selects the input to evaluate at the current iteration t. The GP hyperparameters Ω are received as input and are obtained by optimising the log-marginal likelihood from Equation 2.47 with respect to the hyperparameters. Misspecifying the GP hyperparameters commonly leads to a suboptimal choice of GP and potentially slower convergence rates (Bull 2011). Methods for correctly learning the GP hyperparameters correspond to an active subfield of research. Some methods propose robustness with respect to the hyperparameters by handling model misspecification (Wynne et al. 2021) or by introducing

adaptive bounds on the hyperparameter values (Z. Wang and de Freitas 2014). Considering alternative surrogate-based optimisation methods, there are two that were discussed before: Tree-structured Parzen estimator (TPE), which was detailed in Algorithm 10 and BORE, which was detailed in Algorithm 11. Unlike BO, TPE and BORE are not kernel-based, which means they are not hindered by hyperparameter misspecification. For TPE, the surrogate model is specified by

$$\tau := \text{SplitData}(\mathcal{D}, \gamma) \tag{5.4}$$

$$\Gamma_\gamma := \text{GetDensityRatio}(\mathcal{D}, \tau) \tag{5.5}$$

$$\mathbf{x}_t := \underset{\mathbf{x}\in\mathcal{X}}{\operatorname{argmax}}\, \Gamma_\gamma(\mathbf{x}) \,, \tag{5.6}$$

where τ is a threshold that results from the TPE quantile hyperparameter $0 < \gamma < 1$ and is obtained by calculating $\tau = \Phi^{-1}(\gamma)$. Γ_γ is a relative density ratio from Equation 2.64, and $\mathcal{D}$ is the data observed so far. BORE's surrogate model comes from a probability distribution $\Pi(\mathbf{x})$ obtained by training a probabilistic binary classifier Π_{classif} as follows:

$$\tau := \text{SplitData}(\mathcal{D}, \gamma) \tag{5.7}$$

$$\Pi := \text{TrainBinaryClassifier}(\mathcal{D}, \tau, \Pi_{\text{classif}}) \tag{5.8}$$

$$\mathbf{x}_t := \underset{\mathbf{x}\in\mathcal{X}}{\operatorname{argmax}}\, \Pi(\mathbf{x}) \,, \tag{5.9}$$

where $\Pi(\mathbf{x}) = p(z = 1 \mid \mathbf{x})$. Both methods depend on an exploration-exploitation hyperparameter γ. While TPE is a widely known ML method for hyperparameter optimisation, obtaining decent performance when optimising classification tasks (Bergstra et al. 2011), there has not been research concerning hyperparameter optimisation for controllers with BORE, which is why this chapter is focused on BORE.

5.4 Search Space Exploration

BORE's exploration-exploitation trade-off is balanced by γ, with small γ encouraging more exploitation. As expected, a fixed γ brings up some issues regarding exploration. These challenges become pronounced when BORE is applied to robotic tasks where heteroscedastic noise is a prevalent factor, as seen in the experiments from Chapter 3. To see how γ affects BORE's optimisation when dealing with heteroscedastic noise, a heteroscedastic toy function shown in Figure 5.1 will allow a better understanding of the issue.

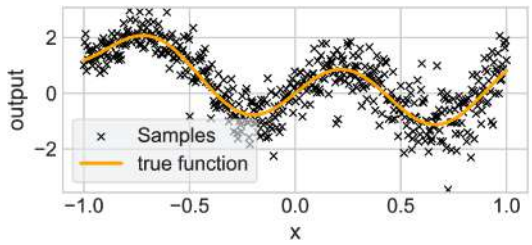


FIGURE 5.1. Heteroscedastic function where the optimal value is at $x = 0.6635$.

$$g(x) = \sin(7x) + x^2 - 0.85x + \mathcal{N}\left(0, \sigma_\nu^2(x)\right)$$
$$\sigma_\nu^2(x) = (0.3 * (x + 4.3))^4/4\,. \tag{5.10}$$

A low γ would lead to considering only the region where it finds the current optimal as in Figure 5.2 where, after 20 iterations, BORE does only exploitation as expected. The BORE optimisation example uses a random forest classifier Π_{classif}. If γ is too large, BORE would realise more exploration, but a simple example from Figure 5.3 with $\gamma = 0.99$ shows that the method can also get stuck in a certain region. This happens because BORE reaches a point where almost all observations are considered optimal (red markers) due to the fixed threshold τ represented by the black dashed line. The threshold appears to be increasing because BORE divides the evaluations into two groups. In this division, a higher γ results in approximately 99% of the evaluations being concentrated near the region where the local optimum is located.

Moreover, BORE does not consider uncertainty, so unseen regions may never be explored. Therefore, BORE keeps jumping through locations within the same region, which in this case, leads to only local optima. It is required to find an ideal value of γ that would do less and less exploration throughout the iterations.

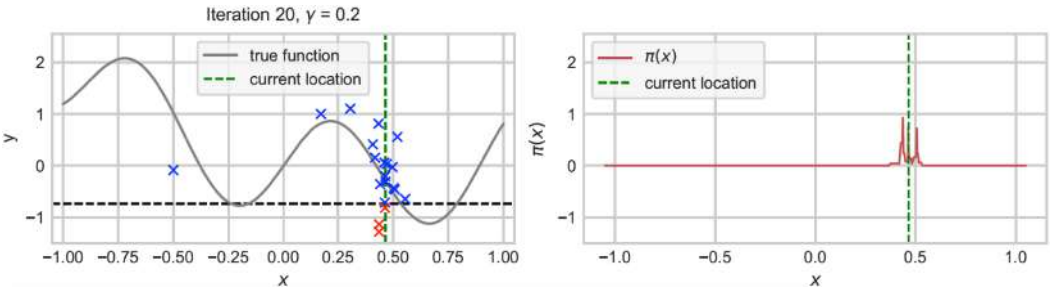


FIGURE 5.2. BORE iteration when $\gamma = 0.2$.

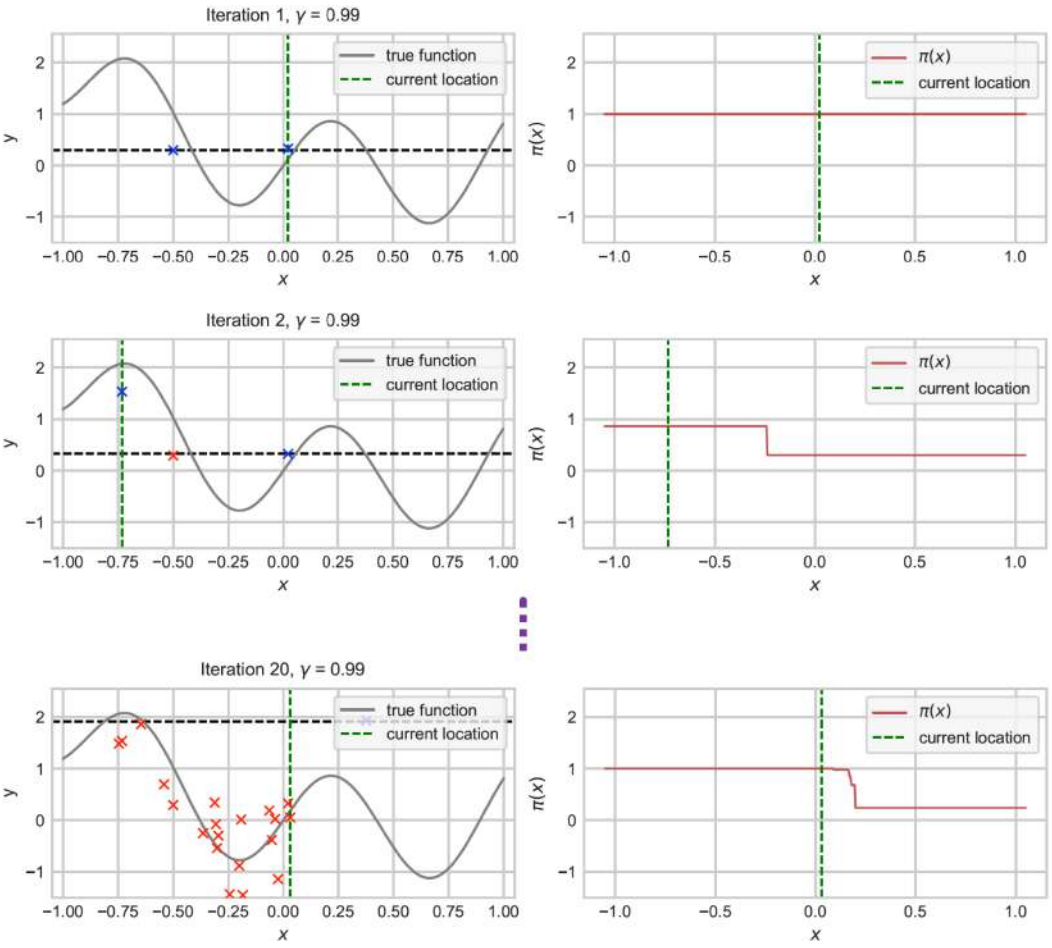


FIGURE 5.3. BORE iterations when $\gamma = 0.99$.

This chapter uses a decaying γ throughout BORE iterations. An example is shown in Figure 5.4, where γ is linearly decayed from 0.99 to 0.05, which means that the values of γ are equally spaced. The linearly decaying value of γ per iteration t is defined as

$$\gamma_t = \gamma_1 - \frac{t-1}{n-1}(\gamma_1 - \gamma_n) \tag{5.11}$$

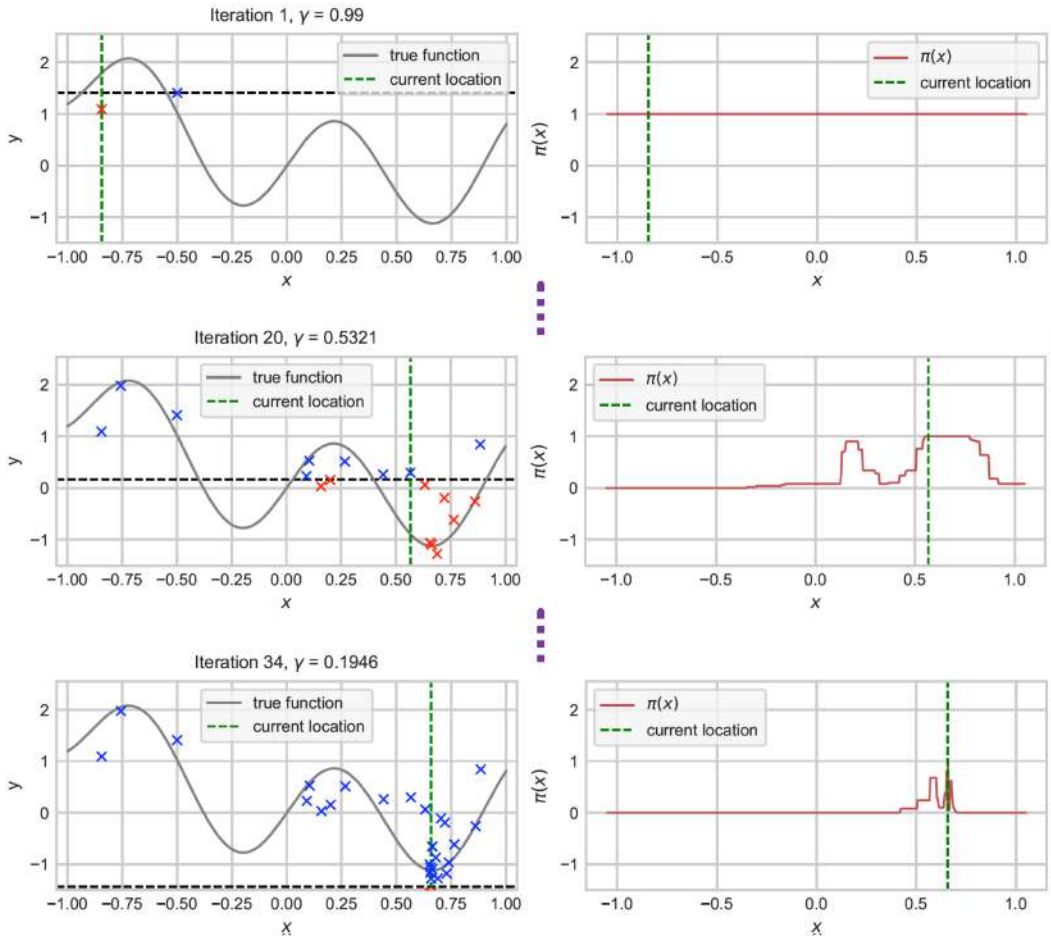


FIGURE 5.4. BORE iterations when linearly decaying γ.

where γ_1 and γ_n are an initial value an final value of γ, and n is the number of iterations. Linear decay performs decently for controller optimisation, as will be shown in the experiments section.

5.5 Classification Problem

As it can be seen in Figure 5.4, the noisy observations are separated by a decision boundary denoted as a black dashed line at $\tau \leftarrow \Phi^{-1}(\gamma_t)$. The observations are labelled according to the threshold. An observation $\mathbf{x}$ is assigned a positive label if the observation obtained an expected cumulative reward higher than τ and negative otherwise. Therefore, the classification problem

is supervised. Then the main issues regarding classification for controller optimisation are as follows:

- In a one-dimensional setting, the observations can be very close to the decision boundary, which could produce wrongly classified inputs since the two classes are not well defined.
- Observations are noisy with heteroscedastic behaviour. That can be more evident when classifying observations in higher dimensional spaces, having to obtain non-linear decision boundaries. The binary classifier chosen would have to be robust to outliers.
- Since the number of observations to classify corresponds to the current iteration t, there are few data points representing each class, which might produce high-bias classifiers.

5.6 Model Predictive Control Tuning

Having defined the optimisation method, the next objective is to establish the controller optimisation problem. As in Chapter 4, we can only maximise an approximation of the expected cumulative reward $\hat{g} \approx g$, where y is a cumulative reward variable, and the input $\mathbf{x}$ consists of the collection of distribution parameters $\boldsymbol{\Psi}$ and the controller hyperparameters $\boldsymbol{\phi}$. $\boldsymbol{\Psi}$ parameterises the distribution-based dynamics model parameters $\boldsymbol{\theta}$. The input variable is defined as $\mathbf{x} = \{\boldsymbol{\Psi}, \boldsymbol{\phi}\}$, and the black-box heteroscedastic optimisation problem is described as follows:

$$y = g(\boldsymbol{\Psi}, \boldsymbol{\phi}) + \nu \qquad \text{where } \nu = \sigma_\nu(\boldsymbol{\Psi}, \boldsymbol{\phi}) \tag{5.12}$$

$$\boldsymbol{\Psi}^\star, \boldsymbol{\phi}^\star = \underset{\{\boldsymbol{\Psi}, \boldsymbol{\phi}\}}{\operatorname{argmax}} \; \hat{g}(\boldsymbol{\Psi}, \boldsymbol{\phi}) \; . \tag{5.13}$$

where the simulated dynamics model $\hat{f}$ depends on b physical parameters $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_b]$ to be randomised at each interaction with the simulated dynamics model. Each distribution-based dynamics model parameter is parameterised by $\boldsymbol{\psi}_i$ for $i = 1, \ldots, b$. Then at state $\mathbf{s}_t$ the

following calculations are performed:

$$\theta_1 \sim p_{\theta_1}(\theta_1; \boldsymbol{\psi}_1) \tag{5.14}$$

$$\vdots \tag{5.15}$$

$$\theta_b \sim p_{\theta_b}(\theta_b; \boldsymbol{\psi}_b) \tag{5.16}$$

$$\mathbf{s}_{t+1} = \hat{f}\left(\mathbf{s}_t, \mathbf{a}_t, [\theta_1, \ldots, \theta_b]\right) \ . \tag{5.17}$$

The uncertainty of the physical parameters of the dynamics model allows the controller to adapt under different environment circumstances or characteristics, such as the size of an obstacle or the length of a robot component.

5.7 Stochastic Model Predictive Control Optimisation by Learning Classifiers

BORE is used for iteratively optimising the controller. The whole proposed method is described in Algorithm 15. The goal is to estimate the optimal $\mathbf{x}^* = \{\boldsymbol{\Psi}^*, \boldsymbol{\phi}^*\}$ using a binary classifier. Following the approach used in Chapter 3, we define the cumulative reward $y = \sum_{i=1}^{n_s} r_i$, where n_s is the number of timesteps in an episode, and set our goal as maximising an expected cumulative reward $\hat{g} := \frac{1}{n_e} \sum_{j=1}^{n_e} y_j(\mathbf{x})$ based on a finite number of episodes n_e. The classifier Π_{classif} is trained by first assigning labels $\{z_k\}_{k=1}^{t-1}$ to the data observed so far until the current iteration t. For training, the classifier uses an auxiliary dataset that consists of observed data until the previous iteration $t-1$ denoted as

$$\{(\boldsymbol{\Psi}_k, \boldsymbol{\phi}_k, z_k)\}_{k=0}^{t-1} \ , \tag{5.18}$$

where the labels are obtained by separating the observed data according to $\gamma \in (0, 1)$ by computing the γth quantile of $\{\hat{g}_k\}_{k=0}^{t-1}$ as follows:

$$\tau \leftarrow \Phi^{-1}(\gamma), \qquad z_k \leftarrow \mathbb{I}[\hat{g}_k \geq \tau] \quad \text{for } k = 0, \ldots, t-1 \ . \tag{5.19}$$

The exploration-exploitation trade-off is balanced by γ, where small γ encourages more exploitation. Instead of keeping γ fixed, we use the strategy of the quantile hyperparameter γ_1 that decays linearly across the iterations until a final γ_n. Inputs predicted as positive labels $z = 1$ are considered to have a higher reward, and one of them is selected by maximising the classifier probability of belonging to the positive class:

$$\boldsymbol{\Psi}_t, \boldsymbol{\phi}_t = \underset{\{\boldsymbol{\Psi}, \boldsymbol{\phi}\} \in \mathcal{X}}{\operatorname{argmax}} \, \Pi_{t-1}(\boldsymbol{\Psi}, \boldsymbol{\phi}) \, . \tag{5.20}$$

Note that this is equivalent to the acquisition function maximisation in the traditional BO. For better performance, the maximisation can be carried out with a global optimisation method like the ones detailed in Section 2.3.

Algorithm 15: Adaptive MPC by Learning Classifiers

input : $\hat{f}$ – Dynamics model
$\mathcal{M}$ – GP model
Ω – GP hyperparameters
$\boldsymbol{\phi}$ – Controller hyperparameters
α – Acquisition function
r – Dense reward function
n – Number of BORE iterations
n_s: Number of timesteps in an episode
n_e – Number of episodes to average the cumulative reward
T – Finite horizon length
$\mathcal{X}$ – Optimisation search space
γ_1, γ_n – Initial and final quantile hyperparameter value
$\Pi_{\text{classif}} : \mathcal{X} \to [0, 1]$ – Probabilistic binary classifier

output : $(\boldsymbol{\Psi}^*, \boldsymbol{\phi}^*, \hat{g}^*)$

$\mathcal{D} \leftarrow$ GetInitialDataset() // Get an initial supervised dataset, e.g. $(\boldsymbol{\Psi}_0, \boldsymbol{\phi}_0, \hat{g}_0)$
for $t \leftarrow 1$ **to** n **do**
  $\mathbf{s}_t \leftarrow$ GetCurrentState()
  $\gamma_t \leftarrow \gamma_1 - \frac{t-1}{n-1}(\gamma_1 - \gamma_n)$ // Linear γ decay
  $\tau \leftarrow \Phi^{-1}(\gamma_t)$ // Compute the γ_t-th quantile of $\{\hat{g}_k\}_{k=0}^{t-1}$
  $z_k \leftarrow \mathbb{I}\left[\hat{g}_k \geq \tau\right]$ for $k = 0, \ldots, t-1$ // Assign labels to the observed data points
  $\Pi_{t-1} \leftarrow$ TrainBinaryClassifier$(\{(\boldsymbol{\Psi}_k, \boldsymbol{\phi}_k, z_k)\}_{k=0}^{t-1}, \Pi_{\text{classif}})$ // BORE's acquisition function
  $\boldsymbol{\Psi}_t, \boldsymbol{\phi}_t = \arg\max_{\{\boldsymbol{\Psi}, \boldsymbol{\phi}\} \in \mathcal{X}} \Pi_{t-1}(\boldsymbol{\Psi}, \boldsymbol{\phi})$ // Estimate new input
  for $j \leftarrow 1$ **to** n_e **do**
    $y_j(\boldsymbol{\Psi}_t, \boldsymbol{\phi}_t) = 0$
    for $i \leftarrow 1$ **to** n_s **do**
      $\mathbf{a}_i^* \leftarrow$ MPC$(\hat{f}, \mathbf{s}_t, r, \boldsymbol{\phi}_t, p_{\boldsymbol{\theta}}(\boldsymbol{\theta}; \boldsymbol{\Psi}_t))$ // Use parameter distributions
      $r_i \leftarrow$ SendToActuators$(\mathbf{a}_i^*, r)$ // Evaluate the optimal action
      $y_j(\boldsymbol{\Psi}_t, \boldsymbol{\phi}_t)$ += r_i // Accumulate rewards
  $\hat{g}_t = 1/n_e \sum_j \left[y_j(\boldsymbol{\Psi}_t, \boldsymbol{\phi}_t)\right]$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{\Psi}_t, \boldsymbol{\phi}_t, \hat{g}_t)\}$

5.8 Experiments

Given the optimisation problem described in Equation 5.13, surrogate-based optimisation methods were evaluated in control and robotic tasks. The proposed adaptive MPC framework was evaluated in several experiments described below.

5.8.1 Simulation Experiments

The optimised control and robotics tasks were carried out only in simulated environments: Pendulum, Half-Cheetah, and Fetchreach from OpenAI Gym with the same dense instantaneous reward functions used in Section 4.6. We also experimented with the reaching task for the Franka robot environment from Bhardwaj et al. (2022) with a single obstacle, a fixed target location, and a fixed initial robot position. The stochastic MPC controller used was MPPI for trajectory optimisation, and it used GPU processing, which helped overcome efficiency issues.

The goal of the Franka task shown was the same as in the previous chapter: reaching a yellow target while avoiding obstacle collision. We assume partial observability for the obstacle dimension sizes and attempt to infer them as part of the dynamics model parameters for which we define search spaces shown in Table 5.1. Some search spaces are different from the ones used in the previous chapter. Each episode consisted of $n_s = 480$ timesteps for Franka and $n_s = 200$ timesteps for the other tasks. The dynamics model parameter l is the rod length for Pendulum, κ_m is the mass scaling factor for all the links in Half-Cheetah, and κ_d is a damping ratio scaling factor for all components in Fetchreach. For Franka, we optimise the obstacle dimensions x, y, and z. Each dynamics model parameter is a random variable parameterised by $\boldsymbol{\psi}$. For example, $\boldsymbol{\psi} = \{\kappa_{d,\mu}, \kappa_{d,\sigma}\}$ are damping ratio mean and damping ratio standard deviation for the Fetchreach.

Environment	n_e	T	M	Control hyp.	Distribution parameter range		True parameter
Pendulum	1	10	10	$\lambda \in [0.01, 50]$ $\sigma_\epsilon \in [1.0, 10]$	$\mu_l \in [0.5, 1.6]$	$\sigma_l \in [0.001, 0.1]$	$l = 1.0$
Half-Cheetah	18	15	10	$\lambda \in [0.01, 1.0]$ $\sigma_\epsilon \in [0.05, 2.0]$	$\kappa_{m,\mu} \in [0.2, 2.0]$	$\kappa_{m,\sigma} \in [0.001, 0.1]$	$\kappa_m = 1.0$
Fetchreach	90	12	3	$\lambda \in [0.01, 0.03]$ $\sigma_\epsilon \in [0.001, 0.5]$	$\kappa_{d,\mu} \in [1.0, 50]$	$\kappa_{d,\sigma} \in [0.001, 0.6]$	$\kappa_d = 1.0$
Franka	10	150	20	$\lambda \in [0.01, 2.0]$	$x_\mu \in [0.3, 0.32]$ $y_\mu \in [0.1, 0.12]$ $z_\mu \in [0.6, 0.62]$	$x_\sigma \in [0.001, 0.05]$ $y_\sigma \in [0.001, 0.01]$ $z_\sigma \in [0.001, 0.03]$	$x = 0.3$ $y = 0.1$ $z = 0.6$

TABLE 5.1. Search spaces for the control and robotic tasks. Intervals for the dynamics model parameters and scaling factors.

5.8.2 Method configuration

We compare the proposed adaptive MPC framework against other surrogate-based methods used in robotics. The configuration for the optimisation and the compared methods are described below.

Adaptive MPC Configuration

In configuring our adaptive MPC framework, a key component is the implementation of Bayesian Optimisation by Ratio Estimation (BORE), which leverages a probabilistic binary classifier to perform optimisation. The classifier, denoted as Π_{classif} should be able to deal with the stochasticity of robotic tasks. Within this scope, we explore several classifier options as detailed in Tiao et al. (2021), including XGBoost (Chen and Guestrin 2016), multi-layer perceptron (MLP) described in Section 2.2.4, and random forest (RF) (Breiman 2001). For example, as an ensemble method, RF combines decision trees via bagging. The number of decision trees should be sufficiently large to reduce classification variance without increasing the bias. We compare two probabilistic classifiers: RF with 50 decision trees denoted as BORE-RF, and Multi-layer Perceptron (MLP) classifier denoted as BORE-MLP with 2 hidden layers, each with 32 units, ReLU activations and sigmoid for the output layer. The weights were optimised for 1000

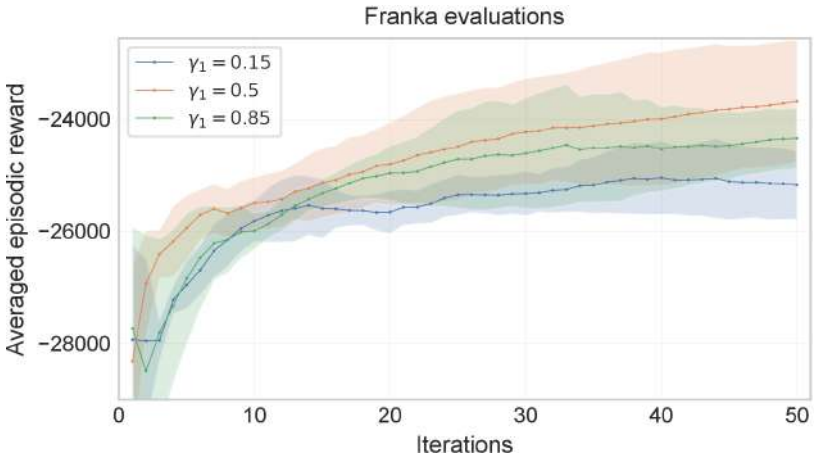


FIGURE 5.5. Some evaluations using different starting values γ_1

epochs using the mini-batch gradient descent described in Algorithm 3, cross-entropy loss, and ADAM optimiser (Kingma and Ba 2015) with a batch size of 32.

Next, we start by exploring with a $\gamma_1 = 0.5$ that decays linearly across the iterations until a reasonable final $\gamma_n = 0.05$. With a low γ_1, BORE could stay stuck in some local minimum, and with a higher γ_1, BORE would do more exploration first before exploiting some region. A reasonable initial value is $\gamma_1 = 0.5$, corresponding to the median, which has shown an optimal performance according to the preliminary results in Figure 5.5 for the Franka reaching problem. Specifically, in these results, $\gamma_1 = 0.5$ consistently outperformed other values in terms of convergence speed and stability, indicating its effectiveness in balancing exploration and exploitation. Finally, we set a parameter distribution $p_{\boldsymbol{\theta}}$ with positive support since we deal with physics variables (mass, damping ratio) and sizes. We choose the gamma distribution $\Gamma(\alpha, \beta)$, and the provided mean μ and standard deviation σ are transformed by computing $\alpha = \frac{\mu^2}{\sigma^2}$ and $\beta = \frac{\mu}{\sigma^2}$.

BO-Based Methods

The proposed method is compared against the traditional homoscedastic BO (BO_{homo}) and the heteroscedastic BO (BO_{hetero}) detailed in Chapter 3. We collected 400 data points via random search for the control and robotic tasks over the search spaces shown in Table 5.1. Then, using such data, optimal GP hyperparameters are found by optimising the GP log-marginal likelihood

using MLSL Algorithm 6. We used a UCB acquisition function $\alpha_{\text{UCB}}(\mathbf{x})$ from Equation 2.60 with balance factor $\delta = 3.0$. An anisotropic squared exponential kernel from Equation 2.44 was used to optimise separate lengthscales for each component of $\mathbf{x}$.

Other Optimisation Methods

Other methods for hyperparameter optimisation used for these experiments are TPE optimisation, which was detailed in Algorithm 10 with quantile value 0.5 since it is what BORE is based on, and finally, we use covariance matrix adaptation evolution strategy (CMA-ES) (Arnold and Hansen 2010) as a non-BO baseline set with $\sigma_0 = 10$ and population size 2. This choice is driven by several considerations. Firstly, the focused search strategy afforded by a smaller population is beneficial in our context, where the search space is not overly complex. Secondly, CMA-ES's efficiency in adapting its search distribution with fewer samples makes it well-suited for scenarios with costly evaluations. CMA-ES has been widely used for hyperparameter tuning in robotics (Modugno et al. 2016; Sharifzadeh et al. 2021).

Optimisation Assessment

To quantitatively assess optimisation performance, we report averaged cumulative rewards, defined as $\frac{1}{t'}\sum_{i=1}^{t'} \hat{g}_i$ from $t' = 1$ to $t' = n$. To obtain the uncertainty about the mean, we repeat those 50 iterations 5 times. The result of repeating gives averaged cumulative rewards with their respective standard deviations. We compare the averaged cumulative reward against the number of iterations. Figure 5.6 shows that BO$_{homo}$ and BO$_{hetero}$ perform similarly mainly in the Pendulum problem because of homoscedastic noise across the search space. However, BO$_{homo}$ tends to converge to a local minimum in the other problems, which is expected since BO$_{homo}$ does not account for heteroscedasticity. It is possible to achieve better or equal results with TPE, although it also seems to get stuck since it only divides observations based on the output and chooses the best next point without considering unseen regions. Both BO versions are being outperformed by BORE-MLP and BORE-RF. Interestingly, the standard deviations observed in the results indicate that the performance consistency of BORE-MLP and BORE-RF varies across different domains. In some cases, BORE-MLP shows much tighter bounds, potentially

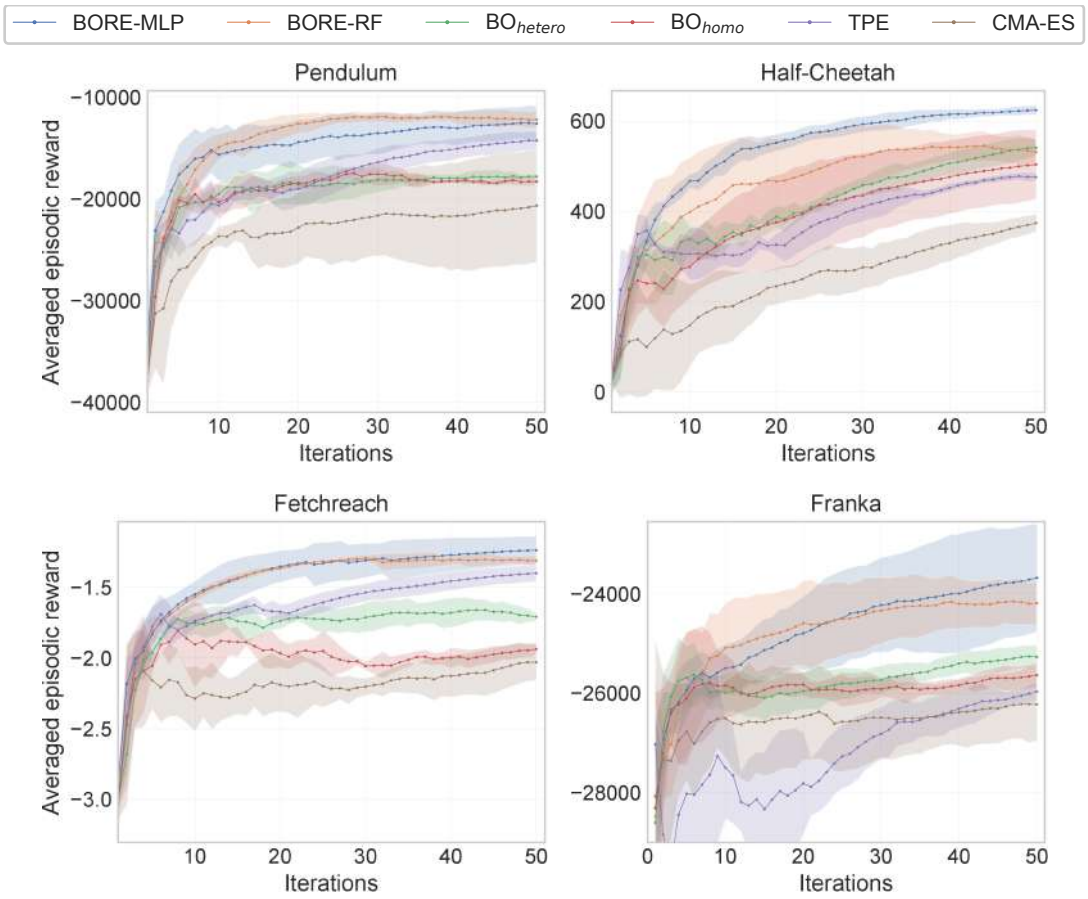


FIGURE 5.6. Expected cumulative rewards $\hat{g}$ per iteration where the shaded areas correspond to 1.5 standard deviations. Each method started at a point with minimum expected cumulative reward obtained via random search.

due to its ability to model complex, non-linear relationships more effectively in those specific domains. Conversely, BORE-RF exhibits lower variation in other domains, which could be attributed to its robustness against overfitting and its effectiveness in handling diverse data types.

This variability in performance consistency underscores the importance of domain-specific characteristics when selecting an optimisation method. It highlights that while BORE-MLP and BORE-RF generally outperform the BO versions, their relative effectiveness can vary depending on the specific nature of the problem being addressed. BORE-MLP converges faster to an optimum in most tasks. In the Franka environment, the difference is higher, and it suggests that the proposed framework performs better in higher-dimensional problems.

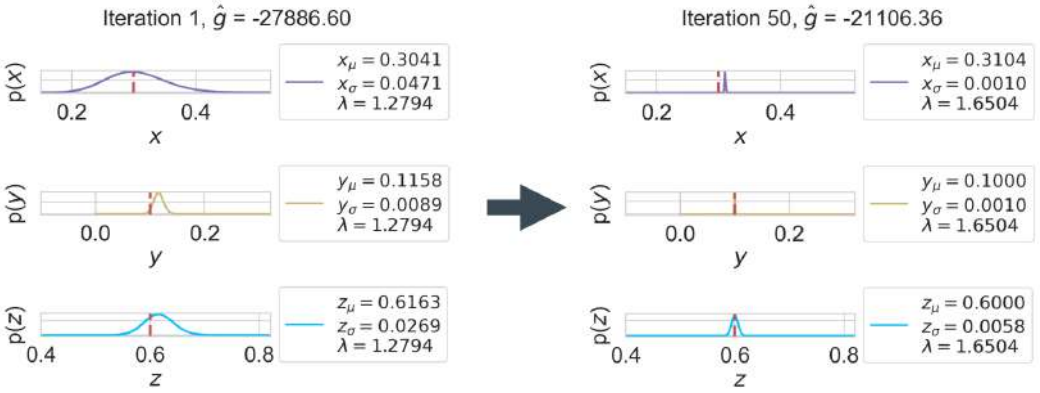
Method	$\hat{g}_{max}$	$\hat{g}_\sigma$	λ	x_μ	x_σ	y_μ	y_σ	z_μ	z_σ
BORE-MLP	-21106.36	724.10	1.65	0.3104	0.0010	0.1000	0.0010	0.6000	0.0058
BORE-RF	-21891.82	291.64	1.67	0.3036	0.0431	0.1001	0.0013	0.6145	0.0234
BO_{hetero}	-23025.47	162.22	1.73	0.3185	0.0500	0.1200	0.0028	0.6145	0.0118
BO_{homo}	-22870.14	158.46	2.00	0.3200	0.0010	0.1000	0.0010	0.6200	0.0010
TPE	-23438.34	121.26	1.63	0.3124	0.0159	0.1152	0.0045	0.6047	0.0067
CMA-ES	-23779.35	481.55	1.47	0.3200	0.0010	0.1200	0.0010	0.6200	0.0142

TABLE 5.2. Maximum reward found at the last iteration for the Franka task.

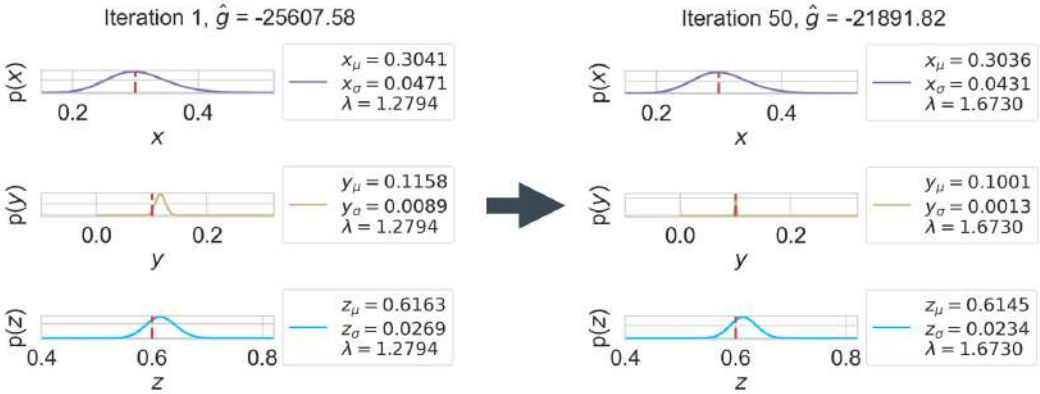
5.8.3 Evaluating the Optima

The previous section emphasised the proposed MPC framework and its ability to explore efficiently compared to other optimisation methods. This section describes the optima found by the methods in the Franka environment, where the improvement is more noticeable. The results obtained with the methods taken into account are shown in Table 5.2, we show the control hyperparameters $\boldsymbol{\phi} = \{\lambda\}$ and the dynamics model parameters $\boldsymbol{\Psi} = \{(x_\mu, x_\sigma), (y_\mu, y_\sigma), (z_\mu, z_\sigma)\}$ that give the maximum reward $\hat{g}_{max}$ at the last iteration after running each method for 50 iterations. $\hat{g}_\sigma$ is the observed standard deviation of the reward at the respective iteration. BORE-MLP is able to find an optimum close to the one found by BORE-RF. $\hat{g}_\sigma$ is higher for BORE-MLP as the method is still exploring new unseen regions at the end, and it can still improve its current maximum. The table also shows the optimised parameters for the distribution-based sizes: violet for the length x, yellow for the width y, and cyan for the height z. To better visualise the optimised distribution-based parameters obtained by the methods, Figure 5.7 shows the initial and final distributions found by each method. There is improvement with respect to the BO_{hetero} proposed in Chapter 4. Besides, it is worth mentioning again that both TPE and BORE do not have to optimise their hyperparameters, unlike BO.

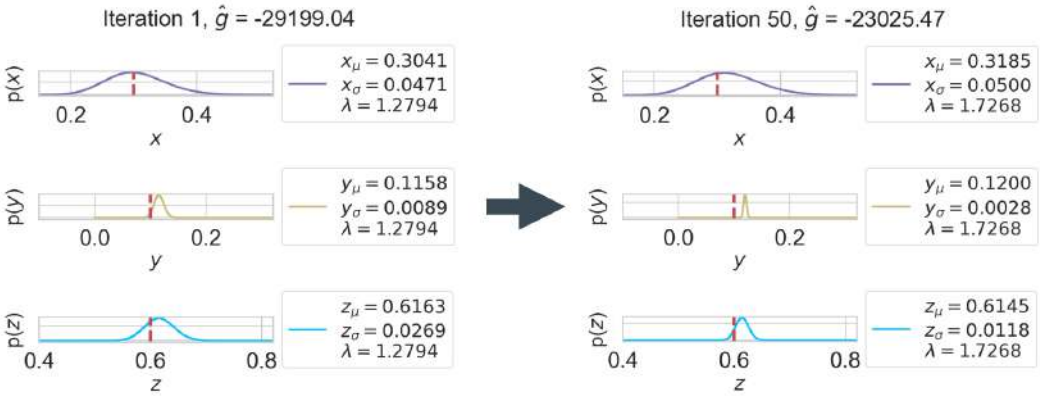
It is also shown that BORE-MLP and almost all the other methods found that considering more uncertainty in the obstacle height z would provide a higher reward, which is understandable considering that the gripper could find convenient trajectories by moving over the obstacle. The most relevant dimension size is the width y since a wrong y would result in obstacle collision. Meanwhile, all methods can allow more uncertainty about the length of the obstacle as it does not affect the collision. Most methods converge to a similar controller hyperparameter λ.



(a) Dynamics model parameters inferred by BORE-MLP.



(b) Dynamics model parameters inferred by BORE-RF.



(c) Dynamics model parameters inferred by BO_{hetero}.

FIGURE 5.7. Initial Franka parameters and best inference at the last iteration.

5.9 Summary

This paper described an adaptive variant of MPC that automatically estimates model parameter distributions and optimises MPC hyperparameters within a BORE optimisation framework. In contrast to previous approaches, the proposed framework shows that global optimisation can be accomplished by learning a binary classifier as a surrogate model of the true function. After evaluating the use of BORE as a surrogate-based optimisation method, this chapter formulated an adaptive MPC framework that deals with the data availability problem better than with BO and also handles heteroscedastic noise settings.

Performance evaluations were realised with the proposed framework with different classifiers and against benchmark BO versions. BORE was able to surpass the performance of the traditional BO and a heteroscedastic BO variation. The experiments were only empirical and showed the effectiveness of BORE hyperparameter optimisation due to significant performance improvement of simulated control and robotic tasks.

CHAPTER 6

Conclusions and Future Work

This thesis addressed the problem of optimising stochastic model predictive control (MPC) while learning and adapting the robot dynamics to the real world. An optimisation framework proposed in the last contributing chapter was built up according to research and experiments from the previous contributing chapters. The subproblems considered were the reality gap problem, which consisted of the error produced when transferring a simulated robot to the real world and the low data availability for designing robotic systems. Since simulated data were used to optimise trajectories, the framework developed handled both a data-efficient optimisation method and a controller that can be adapted to real-world scenarios.

6.1 Contributions

The following is a review of the contributions of this thesis organised by topic.

6.1.1 Heteroscedastic Bayesian Optimisation

Chapter 3 proposed a framework for optimising stochastic MPC hyperparameters based only on observed rewards using Bayesian optimisation (BO). In particular, this thesis makes use of the stochastic MPC method known as model predictive path integral (MPPI). The problem addressed in this chapter was the low data availability in robotics, for which a data-efficient method had to be used. In general, the method that outperformed the rest was heteroscedastic BO due to the input-dependent noise behaviour in various search spaces selected. Since the BO proposed used a GP as a surrogate model, a noise model for the variance was introduced.

A simple but flexible parametric noise model was used to model uncertainty, which led to exploiting the right optimal regions rather than getting stuck in local optima. Experiments against the traditional homoscedastic BO and non-BO methods were realised. Also, simulated control problems that were used throughout the thesis were introduced.

6.1.2 Model Predictive Control Under Parameter Uncertainty

Chapter 4 addressed the reality gap problem by optimising stochastic MPC with randomised simulated data. Therefore an adaptive stochastic MPC optimisation framework that adapts to real-world scenarios was proposed. It consisted of randomising the dynamics model due to the use of distribution-based physical parameters. Both the dynamics model parameter distributions and the controller hyperparameters were jointly optimised in a reward-based objective function. The heteroscedastic noise model used in Chapter 3 was used here. The experiments showed that heteroscedastic BO also outperforms the other methods, and some solutions do not necessarily find the true optimal dynamics model parameters to be optimal. Robotic manipulators (Franka and Jaco) with reaching tasks were also introduced.

6.1.3 Controller Optimisation by Learning Classifiers

Chapter 5 addressed the data availability problem by modifying the surrogate-based optimisation method. A surrogate model that depends on learning a probabilistic binary classifier was used. Bayesian optimisation by ratio estimation (BORE) was introduced with a variation that allowed better handling of exploration and exploitation. An adaptive MPC optimisation framework was also used in similar control and robotic problems. The difference here was that BORE hyperparameters did not need to be optimised with real data evaluations: BORE allowed better data efficiency. The experiments showed that the proposed framework also works decently in similar settings with heteroscedastic noise. Simulated environments were used to empirically test the performance and convergence of the proposed framework.

6.2 Future work

This section presents a few areas for future work to which the methods in this thesis can contribute.

6.2.1 Modeling Uncertainty

There are few theoretical guarantees for BORE's optimality, and it is evident that BORE does not consider uncertainty. The density ratio obtained with a probabilistic classifier does not model the variance as the predictive posterior distribution in BO. Therefore, BORE tends not to explore unseen locations. It would rather exploit regions where a local optimum is found. Another point is that BORE is meant for noiseless objective functions. For example, in a noisy minimisation problem, BORE may consider regions where the noise is high as optimal instead of optimising an approximation of the true function, which also leads to issues regarding heteroscedastic noise. Meanwhile, BO approaches can approximate the true function with the posterior mean. Some future work may include proposing a variation that accounts for uncertainty and analysing it for controller optimisation.

6.2.2 Optimising the Horizon and Number of Trajectories

Although MPPI's performance depends on two main hyperparameters: the control variance and the temperature, the horizon length and the number of trajectories affect the controller's performance in terms of processing efficiency. In most of the applications that use MPPI, both hyperparameters are determined as fixed with high values, such as 10000 trajectories, in order to cover much of the state space. BO for discrete search spaces can be used to optimise both hyperparameters online, lowering the horizon length and the number of trajectories in order to obtain an optimised controller that would perform more efficiently. BORE can be adapted to discrete inputs since it is based on TPE, which can handle discrete data.

6.2.3 Distribution-Based Actions

MPPI samples control signal perturbations from a normal distribution parameterised by the control signal. With the purpose of minimising the reality gap, such a distribution can become dependent on physical limitations. For example, in a manipulator task with action space corresponding to joint angles, the perturbation distribution can be optimised according to them. By defining beta-distributed perturbations, the action signals produced would be limited to the interval $[0, 1]$, which can be mapped to manipulator joint angles. This will produce more realistic action signals if the perturbation distribution is optimised iteratively.

Bibliography

Abraham, I., Handa, A., Ratliff, N., Lowrey, K., Murphey, T. D., & Fox, D. (2020). Model-based generalization under parameter uncertainty using path integral control. *IEEE Robotics and Automation Letters*, *5*(2), 2864–2871.

Andrieu, C., de Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to mcmc for machine learning. *Machine Learning*, *50*(1), 5–43.

Andrychowicz, M., Baker, B., Chociej, M., Józefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L., & Zaremba, W. (2020). Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, *39*(1), 3–20.

Antsaklis, P. J., & Rahnama, A. (2018). Control and machine intelligence for system autonomy. *Journal of Intelligent & Robotic Systems*, *91*(1), 23–34.

Arnold, D. V., & Hansen, N. (2010). Active covariance matrix adaptation for the (1+1)-cma-es. *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, 385–392.

Astrom, K. J., & Murray, R. M. (2008). *Feedback systems: An introduction for scientists and engineers*. Princeton University Press.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*(10), 281–305.

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, *24*.

Bhardwaj, M., Sundaralingam, B., Mousavian, A., Ratliff, N. D., Fox, D., Ramos, F., & Boots, B. (2022). Storm: An integrated framework for fast joint-space model-predictive control for reactive manipulation. *Proceedings of the 5th Conference on Robot Learning*, *164*, 750–759.

Bianchini, M., & Scarselli, F. (2014). On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE Transactions on Neural Networks and Learning Systems*, *25*(8), 1553–1565.

Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Springer.

Bishop, R. C. (2011). Metaphysical and epistemological issues in complex systems. In *Philosophy of complex systems* (pp. 105–136, Vol. 10). North-Holland.

Blitzstein, J. K., & Hwang, J. (2019). *Introduction to probability* (Second). Chapman; Hall/CRC.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

Brunke, L., Greeff, M., Hall, A. W., Yuan, Z., Zhou, S., Panerati, J., & Schoellig, A. P. (2022). Safe learning in robotics: From learning-based control to safe reinforcement learning. *Annual Review of Control, Robotics, and Autonomous Systems*, *5*(1), 411–444.

Brunton, S. L., & Kutz, J. N. (2019). *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press.

Bull, A. D. (2011). Convergence rates of efficient global optimization algorithms. *Journal of Machine Learning Research*, *12*(88), 2879–2904.

Burkov, A. (2019). *The hundred-page machine learning book*. Andriy Burkov.

Buşoniu, L., de Bruin, T., Tolić, D., Kober, J., & Palunko, I. (2018). Reinforcement learning for control: Performance, stability, and deep approximators. *Annual Reviews in Control*, *46*, 8–28.

Byrd, R. H., Lu, P., Nocedal, J., & Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, *16*(5), 1190–1208.

Carron, A., Arcari, E., Wermelinger, M., Hewing, L., Hutter, M., & Zeilinger, M. N. (2019). Data-Driven Model Predictive Control for Trajectory Tracking With a Robotic Arm. *IEEE Robotics and Automation Letters*, *4*(4), 3758–3765.

Charlesworth, H. J., & Montana, G. (2021). Solving challenging dexterous manipulation tasks with trajectory optimisation and reinforcement learning. *Proceedings of the 38th International Conference on Machine Learning*, *139*, 1496–1506.

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794.

Collins, J., Chand, S., Vanderkop, A., & Howard, D. (2021). A review of physics simulators for robotic applications. *IEEE Access*, *9*, 51416–51431.

Cox, D. D., & John, S. (1992). A statistical method for global optimization. *[Proceedings] 1992 IEEE International Conference on Systems, Man, and Cybernetics*, 1241–1246 vol.2.

Cranmer, K., Brehmer, J., & Louppe, G. (2020). The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences*, *117*(48), 30055–30062.

Deisenroth, M., & Rasmussen, C. E. (2011). Pilco: A model-based and data-efficient approach to policy search. *Proceedings of the 28th International Conference on machine learning (ICML-11)*, 465–472.

Deisenroth, M. P., Fox, D., & Rasmussen, C. E. (2015). Gaussian processes for data-efficient learning in robotics and control. *IEEE Trans. Pattern Anal. Mach. Intell.*, *37*(2), 408–423.

Deisenroth, M. P., Rasmussen, C. E., & Fox, D. (2011). Learning to control a low-cost manipulator using data-efficient reinforcement learning. *Robotics: Science and Systems VII, University of Southern California, Los Angeles, CA, USA, June 27-30, 2011*.

DiStefano III, J. J., Stubberud, A. R., & Williams, I. J. (2014). *Schaum's outline of feedback and control systems* (2nd Edition). McGraw-Hill Education.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(61), 2121–2159.

Duvenaud, D. (2014). *Automatic model construction with gaussian processes* [Doctoral dissertation, University of Cambridge].

Edelstein-Keshet, L. (2005). *Mathematical models in biology*. Society for Industrial; Applied Mathematics.

Edwards, W., Tang, G., Mamakoukas, G., Murphey, T., & Hauser, K. (2021). Automatic tuning for data-driven model predictive control. *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 7379–7385.

Fiducioso, M., Curi, S., Schumacher, B., Gwerder, M., & Krause, A. (2019). Safe contextual bayesian optimization for sustainable room temperature pid control tuning. *Proceedings*

of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, 5850–5856.

Fontes, F. A., & Magni, L. (2003). Min-max model predictive control of nonlinear systems using discontinuous feedbacks. *IEEE Transactions on Automatic Control*, *48*(10), 1750–1755.

Forsberg, F. (2022). Domain adaptation to meet the reality-gap from simulation to reality, 57.

Gal, Y., McAllister, R., & Rasmussen, C. E. (2016). Improving PILCO with Bayesian neural network dynamics models. *Data-Efficient Machine Learning workshop, International Conference on Machine Learning*.

Gandhi, M. S., Vlahov, B., Gibson, J., Williams, G., & Theodorou, E. A. (2021). Robust model predictive path integral control: Analysis and performance guarantees. *IEEE Robotics and Automation Letters*, *6*(2), 1423–1430.

Ganjali, D. (2016). *Efficient reinforcement learning with bayesian optimization*. University of California, Irvine.

Gardner, J., Guo, C., Weinberger, K., Garnett, R., & Grosse, R. (2017). Discovering and Exploiting Additive Structure for Bayesian Optimization. *International Conference on Artificial Intelligence and Statistics*, *54*, 1311–1319.

Geron, A. (2019). *Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to build intelligent systems* (2nd). O'Reilly Media, Inc.

Giulioni, L. (2015). *Stochastic model predictive control with application to distributed control systems* [Doctoral dissertation, Polytechnic University of Milan].

Goldberg, P., Williams, C., & Bishop, C. (1997). Regression with input-dependent noise: A gaussian process treatment. *Advances in Neural Information Processing Systems*, *10*.

Golovin, D., Kochanski, G., & Karro, J. E. (2017). Black box optimization via a bayesian-optimized genetic algorithm. *Advances in Neural Information Processing Systems*.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Görges, D. (2017). Relations between model predictive control and reinforcement learning. *IFAC-PapersOnLine*, *50*(1), 4920–4928.

Guzman, R., Oliveira, R., & Ramos, F. (2021). Heteroscedastic bayesian optimisation for stochastic model predictive control. *IEEE Robotics and Automation Letters*, *6*(1), 56–63.

Guzman, R., Oliveira, R., & Ramos, F. (2022a). Adaptive model predictive control by learning classifiers. *Proceedings of The 4th Annual Learning for Dynamics and Control Conference*, *168*, 480–491.

Guzman, R., Oliveira, R., & Ramos, F. (2022b). Bayesian optimisation for robust model predictive control under model parameter uncertainty. *2022 International Conference on Robotics and Automation (ICRA)*, 5539–5545.

Hansen, N., Müller, S. D., & Koumoutsakos, P. (2003). Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evol. Comput.*, *11*(1), 1–18.

Hewing, L., Wabersich, K. P., Menner, M., & Zeilinger, M. N. (2020). Learning-based model predictive control: Toward safe learning in control. *Annual Review of Control, Robotics, and Autonomous Systems*, *3*(1), 269–296.

Hsu, K.-C., Ren, A. Z., Nguyen, D. P., Majumdar, A., & Fisac, J. F. (2023). Sim-to-lab-to-real: Safe reinforcement learning with shielding and generalization guarantees. *Artificial Intelligence*, *314*, 103811.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. *Learning and Intelligent Optimization*, 507–523.

Jacob Mathew, M. (2020). *Learning forward-models for robot manipulation* [Doctoral dissertation, University of Birmingham].

Jebara, T. (2003). *Machine learning: Discriminative and generative*. Springer.

Jiang, P., Zhou, Q., & Shao, X. (2020). *Surrogate model-based engineering design and optimization*. Springer.

Kaiser, M. (2021). *Structured models with gaussian processes* [Doctoral dissertation, Technische Universität München].

Kersting, K., Plagemann, C., Pfaff, P., & Burgard, W. (2007). Most likely heteroscedastic gaussian process regression. *Proceedings of the 24th International Conference on Machine Learning*, 393–400.

Kim, T., Park, G., Kwak, K., Bae, J., & Lee, W. (2022). Smooth model predictive path integral control without smoothing. *IEEE Robotics and Automation Letters*, *7*(4), 10406–10413.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization.

Kiumarsi, B., Vamvoudakis, K. G., Modares, H., & Lewis, F. L. (2018). Optimal and autonomous control using reinforcement learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, *29*(6), 2042–2062.

Kozłowski, K., & Pazderski, D. (2004). Modeling and control of a 4-wheel skid-steering mobile robot. *International Journal of Applied Mathematics and Computer Science*, *14*(4), 477–496.

Kuindersma, S., Grupen, R. A., & Barto, A. G. (2012). Variational bayesian optimization for runtime risk-sensitive control. In *Robotics: Science and systems viii*.

Kushner, H. J. (1964). A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise. *Journal of Basic Engineering*, *86*(1), 97–106.

Lázaro-Gredilla, M., & Titsias, M. K. (2011). Variational heteroscedastic gaussian process regression. *Proceedings of the 28th International Conference on International Conference on Machine Learning*, 841–848.

Lee, K., Seo, Y., Lee, S., Lee, H., & Shin, J. (2020). Context-aware dynamics model for generalization in model-based reinforcement learning. *Proceedings of the 37th International Conference on Machine Learning*, *119*, 5757–5766.

Liang, C., Wang, W., Liu, Z., Lai, C., & Zhou, B. (2019). Learning to guide: Guidance law based on deep meta-learning and model predictive path integral control. *IEEE Access*, *7*, 47353–47365.

Liberzon, D. (2011). Calculus of variations and optimal control theory: A concise introduction.

Ljungqvist, O. (2020). *Motion planning and feedback control techniques with applications to long tractor-trailer vehicles*. Linköping University Electronic Press.

Loshchilov, I., & Hutter, F. (2016). CMA-ES for hyperparameter optimization of deep neural networks. *CoRR*, *abs/1604.07269*.

Lu, Q., Kumar, R., & Zavala, V. M. (2020). MPC controller tuning using bayesian optimization techniques. *CoRR*, *abs/2009.14175*.

Luke, S. (2013). *Essentials of metaheuristics* (second). Lulu.

Lynch, K. M., & Park, F. C. (2017). *Modern robotics*. Cambridge University Press.

Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W., & Bergstra, J. (2018). Benchmarking reinforcement learning algorithms on real-world robots. *Proceedings of The 2nd Conference on Robot Learning*, *87*, 561–591.

Manuelli, L. (2020). *Robot manipulation with learned representations* [Doctoral dissertation]. Massachusetts Institute of Technology.

Marchant Matus, R. (2015). *Bayesian optimisation for planning in dynamic environments* [Doctoral dissertation, University of Sydney].

Marco-Valle, A. (2020, July). *Bayesian optimization in robot learning - automatic controller tuning and sample-efficient methods* [Doctoral dissertation, Eberhard Karls Universität Tübingen].

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Mchutchon, A., & Rasmussen, C. (2011). Gaussian process training with input noise. *Advances in Neural Information Processing Systems*, *24*.

Modugno, V., Neumann, G., Rueckert, E., Oriolo, G., Peters, J., & Ivaldi, S. (2016). Learning soft task priorities for control of redundant robots. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 221–226.

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of machine learning* (2nd). MIT Press.

Montavon, G., Braun, M. L., & Müller, K.-R. (2011). Kernel analysis of deep networks. *Journal of Machine Learning Research*, *12*(78), 2563–2581.

Muratore, F. (2021). *Randomizing physics simulations for robot learning* [Doctoral dissertation, Technische Universität Darmstadt].

Muratore, F., Eilers, C., Gienger, M., & Peters, J. (2021a). Data-efficient domain randomization with Bayesian optimization. *IEEE Robotics and Automation Letters*, *6*(2), 911–918.

Muratore, F., Ramos, F., Turk, G., Yu, W., Gienger, M., & Peters, J. (2021b). Robot learning from randomized simulations: A review. *CoRR*, *abs/2111.00956*.

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. MIT Press.

Nayak, S. (2021). *Fundamentals of optimization techniques with algorithms*. Academic Press.

Oliveira, R., Ott, L., & Ramos, F. (2021). No-regret approximate inference via bayesian optimisation. *37th Conference on Uncertainty in Artificial Intelligence (UAI 2021)*.

Oliveira, R., Tiao, L., & Ramos, F. (2022). Batch bayesian optimisation via density-ratio estimation with guarantees.

Oliveira, R., M. Rocha, F. H., Ott, L., Guizilini, V., Ramos, F., & Grassi, V. (2018). Learning to race through coordinate descent bayesian optimisation. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 6431–6438.

Ozaki, Y., Tanigaki, Y., Watanabe, S., & Onishi, M. (2020). Multiobjective tree-structured parzen estimator for computationally expensive optimization problems. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 533–541.

Peng, X. B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2018). Sim-to-real transfer of robotic control with dynamics randomization. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 1–8.

Peshkin, L. M. (2002). *Reinforcement learning by policy search* [Doctoral dissertation]. Brown University.

Poggio, T., Mhaskar, H., Rosasco, L., Miranda, B., & Liao, Q. (2017). Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *International Journal of Automation and Computing*, *14*(5), 503–519.

Pravitra, J., Ackerman, K. A., Cao, C., Hovakimyan, N., & Theodorou, E. A. (2020). $\mathcal{L}1$-adaptive MPPI architecture for robust and agile control of multirotors. *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, 7661–7666.

Ramos, F., Carvalhaes Possas, R., & Fox, D. (2019). BayesSim : adaptive domain randomization via probabilistic inference for robotics simulators. *Robotics: Science and Systems (RSS)*.

Rasmussen, C. E., & Williams., C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Rinnooy Kan, A. H. G., & Timmer, G. T. (1987). Stochastic global optimization methods part ii: Multi level methods. *Mathematical Programming*, *39*(1), 57–78.

Romeres, D., Prando, G., Pillonetto, G., & Chiuso, A. (2016). On-line bayesian system identification. *15th European Control Conference, ECC 2016, Aalborg, Denmark, June 29 - July 1, 2016*, 1359–1364.

Russell, S., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach (Fourth).

Salhi, A., Proll, L. G., Rios Insua, D., & Martin, J. I. (2000). Experiences with stochastic algorithms for a class of constrained global optimisation problems. *RAIRO - Operations Research - Recherche Opérationnelle*, *34*(2), 183–197.

Sammut, C., & Webb, G. I. (2011). *Encyclopedia of machine learning*. Springer.

Scannell, A. (2022). Bayesian learning for control in multimodal dynamical systems.

Schlkopf, B., Smola, A. J., & Bach, F. (2018). *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT Press.

Semage, B. L., Karimpanal, T. G., Rana, S., & Venkatesh, S. (2022). Uncertainty aware system identification with universal policies, 2321–2327.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & de Freitas, N. (2016). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, *104*(1), 148–175.

Sharifzadeh, M., Jiang, Y., Lafmejani, A. S., Nichols, K., & Aukes, D. (2021). Maneuverable gait selection for a novel fish-inspired robot using a cma-es-assisted workflow. *Bioinspiration & Biomimetics*, *16*(5), 056017.

Shekhar, S., Bansode, A., & Salim, A. (2021). A comparative study of hyper-parameter optimization tools. *2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, 1–6.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*, *25*.

Sorourifar, F., Makrygirgos, G., Mesbah, A., & Paulson, J. A. (2021). A data-driven automatic tuning method for mpc under uncertainty using constrained bayesian optimization. *IFAC-PapersOnLine*, *54*(3), 243–250.

Sünderhauf, N., Brock, O., Scheirer, W., Hadsell, R., Fox, D., Leitner, J., Upcroft, B., Abbeel, P., Burgard, W., Milford, M., & Corke, P. (2018). The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, *37*(4-5), 405–420.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, R., & Barto, A. (2018). *Reinforcement learning, second edition: An introduction*. MIT Press.

Svensson, A., Dahlin, J., & Schön, T. B. (2015). Marginalizing gaussian process hyperparameters using sequential monte carlo. *6th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing, CAMSAP 2015, Cancun, Mexico, December 13-16, 2015*, 477–480.

Tedrake, R. (2023). Underactuated robotics: Algorithms for walking, running, swimming, flying, and manipulation. Course Notes for MIT 6.832.

Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic robotics*. MIT Press.

Tiao, L. C., Klein, A., Seeger, M. W., Bonilla, E. V., Archambeau, C., & Ramos, F. (2021). Bore: Bayesian optimization by density-ratio estimation. *139*, 10289–10300.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 23–30.

Todorov, E., Erez, T., & Tassa, Y. (2012). Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033.

Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z., & Guyon, I. (2021). Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, *133*, 3–26.

van Hasselt, H. (2012). Reinforcement learning in continuous state and action spaces. In *Reinforcement learning: State-of-the-art* (pp. 207–251). Springer.

Wang, T., Bao, X., Clavera, I., Hoang, J., Wen, Y., Langlois, E. D., Zhang, S., Zhang, G., Abbeel, P., & Ba, J. (2019). Benchmarking model-based reinforcement learning. *CoRR*, *abs/1907.02057*.

Wang, X., Jin, Y., Schmitt, S., & Olhofer, M. (2023). Recent advances in bayesian optimization. *ACM Computing Surveys*.

Wang, Z., & de Freitas, N. (2014). Theoretical analysis of bayesian optimisation with unknown gaussian process hyper-parameters. *CoRR*, *abs/1406.7758*.

Weise, T. (2009, June). *Global optimization algorithms - theory and application* (Second). Self-Published.

Williams, G., Aldrich, A., & Theodorou, E. A. (2017a). Model predictive path integral control: From theory to parallel computation. *Journal of Guidance, Control, and Dynamics*, *40*(2), 344–357.

Williams, G., Drews, P., Goldfain, B., Rehg, J. M., & Theodorou, E. A. (2016). Aggressive driving with model predictive path integral control. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 1433–1440.

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

Williams, G., Drews, P., Goldfain, B., Rehg, J. M., & Theodorou, I. A. (2018). Information-Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving. *IEEE Transactions on Robotics*, *34*(6), 1603–1622.

Wilson, T., & Williams, S. B. (2017). Active sample selection in scalar fields exhibiting non-stationary noise with parametric heteroscedastic gaussian process regression. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 6455–6462.

Wynne, G., Briol, F.-X., & Girolami, M. (2021). Convergence guarantees for gaussian process means with misspecified likelihoods and smoothness. *Journal of Machine Learning Research*, *22*.

Zhang, C., Bütepage, J., Kjellström, H., & Mandt, S. (2019). Advances in variational inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *41*(8), 2008–2026.

Zhao, M., & Li, J. (2018). Tuning the hyper-parameters of cma-es with tree-structured parzen estimators. *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, 613–618.