Chair of Automatic Control

Technical University of Munich

# Design of Adaptive Nonlinear Controllers using Supervised Learning

Christian Dengler

Vollständiger Abdruck der von der Fakultät für Maschinenwesen der Technischen Universität München zur Erlangung des akademischen Grades eines Doktor-Ingenieurs genehmigten Dissertation.

| | |
|---|---|
| Vorsitzender: | Prof. Dr.-Ing. Michael W. Gee |
| Prüfende der Dissertation: | Prof. Dr.-Ing. Boris Lohmann |
| | Prof. Dr. Eyke Hüllermeyer |

Die Dissertation wurde am 30.06.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Maschinenwesen am 12.01.2021 angenommen.

# Abstract

This thesis deals with the design of adaptive controllers in the form of function approximators. The adaptability of the controller increases the control performance under uncertain model parameters. The approach in the thesis is based on optimized trajectories as a data basis, together with a new approach called disturbed oracle imitation that is required to train a controller with hidden states, e.g., a recurrent neural network. The approach is analyzed empirically using a simulation example and applied to the physical system of a mobile inverted pendulum.

# Zusammenfassung

Die vorliegende Arbeit befasst sich mit dem Entwurf von adaptiven Reglern in der Form von parametrierten Funktionen. Die Adaption des Reglers erhöht die Robustheit gegenüber unsicheren Modellparametern. Der Ansatz basiert auf optimierten Trajektorien als Datenbasis, zusammen mit einer neuen, gestörte Orakel-Nachahmung genannten Methode. Diese dient dazu, einen dynamischen Regler mit internen Zuständen zu trainieren. Der Ansatz wird empirisch an einem Simulationsbeispiel untersucht und am realen System inverses Pendel auf Rädern angewandt.

# Acknowledgements

The content of this thesis is the result of more than four years at the chair of automatic control at the Technical University of Munich, a time that was made possible and enjoyable by a bunch of people I would like to thank at this place. First of all, my thanks go to Prof. Boris Lohmann who allowed me not only to work at his chair, but also to freely chase the ideas that I found of interest. Also, he is a more than worthy opponent at table football. Second, I'd like to thank Prof. Vogel-Heuser for leading the research project CRC 768 that partly funded my time at Prof. Lohmann's chair and allowed me to widen my horizon on multiple topics that I would otherwise never have encountered. Then, I'd like to thank my former office college Ertug Olcay for many discussions, be it of technical or non-technical nature, and all the help on matters within the CRC 768. Then, all other colleagues at the chair as well as the students I got to supervise, that all influenced my work at the chair in some way. Last but not least, I'd like to thank all the people and companies that support, maintain or provide the free and open-source software that is used throughout the thesis, be it the operating system, the programming languages, the typesetting environment, the drawing program and probably many more.

# Contents

# Notation

## Abbreviations

A list of abbreviations used throughout the thesis is given in the following table:

| | |
|---|---|
| CMAES | Covariance matrix adaptation evolution strategy |
| CPU | Central processing unit |
| DAGGER | Data-set aggregation |
| DART | Disturbances for augmenting robot trajectories |
| DC | Direct current |
| DOI | Disturbed oracle imitation |
| EMF | Electromotive force |
| Epopt | Ensemble policy optimization |
| ES | Evolution strategy |
| GAE | Generalised advantage estimation |
| GPU | Graphics processing unit |
| MIP | Mobile inverted pendulum |
| MPC | Model predictive control |
| PPO | Proximal policy optimization |
| PWM | Pulse width modulation |
| RNN | Recurrent neural network |
| TBPTT | Truncated backpropagation through time |
| TPU | Tensor processing unit |

# General Notation

We name variables according to the common names used in the control community, rather than the names used in the machine learning community. For example, a state is represented by the variable $\mathbf{x}$ and the control action is represented by $\mathbf{u}$. The logarithm without indicated base $\log(x)$ is the natural logarithm with base $e$. We use common notations for vectors, matrices or operators that are indicated by the font, letter case, typography etc. :

| | |
|---|---|
| $x$ | A scalar variable |
| $\mathbf{x}$ | A vector |
| $\mathbf{X}$ | A matrix |
| $\mathsf{x}$ | A scalar random variable |
| $\mathbf{x}$ | A vector of random variables |
| $x_i$ | The i'th element of the vector $\mathbf{x}$ |
| $\mathbf{x}^*$ | A star superscript indicates a value that is optimal with respect to a given optimization problem |
| $f(x;\boldsymbol{\Theta})$ | A function of $x$ parameterized by $\boldsymbol{\Theta}$. The parameters are omitted when they are not important, e.g., after training. |
| $\mathbf{I}$ | The identity matrix. |

Chapter 1

# Introduction

## 1.1 Motivation for machine learning in control engineering

With the increase in the availability of computation power and efficient software libraries for neural networks, previous state of the art methods in fields like computer vision, text recognition, and computer games have been improved or even replaced by methods from the field of machine learning. A breakthrough example is the work of Krizhevsky et al. [55] who, in 2012, used a deep convolutional neural network for an image recognition challenge and about halved the previous best error rate. Another example, closer related to this work, is the software AlphaGo that in 2016 beat the champion Lee Sedol in the board game Go [101].

The previous examples show that expressive function approximators like neural networks can, if designed and trained the right way, surpass software that is hand-designed by humans or even surpass human decision-making itself. With the successes in computer vision, language processing or games, other researchers started to look into applications for methods of machine learning in their own field, which led to this thesis.

The field of control theory has been around since the 19th century, and is mostly concerned with feed forward and feedback control. We will be concerned with feedback controllers in this thesis, which are used in actuated

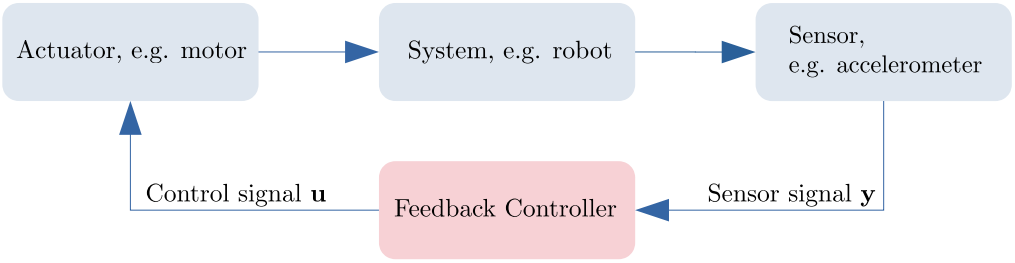



Figure 1.1: The general structure of a feedback controlled system.

and sensing machines. A feedback loop is shown in figure 1.1 with the controller highlighted in red. The feedback controller is in charge of providing a control signal $\mathbf{u}$ for the actuators of the system, e.g., the motor voltage, based on the current state of the system, e.g., the velocity and position, and possible external inputs, e.g., a target position. The actuators influence the system, e.g., by applying forces or moments. The sensors of the system measure the impact of the actuator, and provide a new input for the controller. The feedback controller is designed by an engineer to influence the system in a specified way.

Despite the ongoing research in control theory, deriving a control law in closed form can be a difficult task, especially for systems with unfavorable characteristics, e.g., unstable or nonlinear systems. Machine learning is presenting an opportunity towards automating the control design by optimizing the closed-loop behavior based on a cost function. With only a cost function defining the desired behavior, a class of methods called policy gradient methods have been used for the control of complex systems, e.g., in the work of Lillicrap et al. [65]. For a practical example, Abbeel et al. [2] used machine learning to design a controller for an autonomous helicopter able to perform maneuvers like flips or sideways rolls.

The transfer of machine learning based control from research into industrial applications is not considered successful up to date. For one thing, the methods gained popularity only recently and experts on the topic are still scarce. For another thing, the methods are known to not work out of the box easily. Problems include the design of an expressive cost function [29], large computation times or a deterioration of the control performance

when applied to a real system [73]. This thesis focuses on the last of the mentioned problems, i.e., a possible deterioration of the control performance after transferring the controller from simulation to application.

## 1.2 Goal of the thesis

The control design using machine learning is almost exclusively done in simulation, since the algorithms require a lot of data, which is costly to produce using a real system. Moreover, the system could be damaged during the data-generation phase. However, we know that "all models are wrong" [13] and some are useful. Therefore, the behavior in simulation is different from the behavior in the real application. This problem is not unique to controllers designed using machine learning, but also appears for an analytic control design and has led to the fields of robust and adaptive control. Robust control aims to find a static controller that meets performance criteria even in the case of bounded model errors. Adaptive control on the other hand aims to find a dynamic controller that adjusts its behavior depending on how the system behaves. The controller is adapted in a way that performance criteria are met for different system behaviors.

In recent developments, ideas from robust and adaptive control are investigated upon for the control design using machine learning. While the focus in current literature lies on policy gradient methods for a robust or adaptive control design using machine learning, in this thesis we take a first step towards adaptability for an indirect method that combines trajectory optimization and imitation learning. The goal of the thesis is to develop, implement and analyze a method for the design of adaptive controllers using machine learning that reliably and consistently produces good results. An advantage of the presented method is the easier tuning of the closed-loop behavior using equality constraints and the robustness against variations of the hyper-parameters.

We implement an adaptive controller using our novel method and compare the resulting control performance of our method with competing methods like policy gradient methods and evolution strategies. Afterwards, we validate the results in the application for a mobile inverted pendulum, which

is an under-actuated system with nonholonomic constraints, described by nonlinear dynamics.

## 1.3 Outline of the thesis

The thesis is divided into seven chapters with the first chapter being this introduction. The following chapter revisits the theoretical foundations required throughout the thesis, i.e., an introduction into machine learning, trajectory optimization and control design algorithms for training parameterized functions. The third chapter revisits existing works for the design of adaptive controllers using machine learning and describes a new approach to design the adaptive controller using trajectory optimization and supervised learning. The fourth chapter empirically analyzes the control performance of an adaptive controller designed using the new method on the example of a bridge crane in simulation. The analysis includes a comparison of the control performance with robust and adaptive controllers trained using a policy gradient algorithm and evolution strategies. The fifth chapter shows results for the adaptive controller on the mobile inverted pendulum, both in simulation and application. A short discussion of the results for both systems is found in chapter six and the thesis is summarized in chapter seven.

Chapter 2

# Theoretical foundations

This chapter provides fundamental theory required in later chapters. The addressed topics are machine learning, trajectory optimization and a short overview on existing methods to train a controller in the form of a function approximator.

## 2.1 Machine learning basics

Machine learning is a sub-field of artificial intelligence, defined by Murphy [75, p. 1] as

> ...a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision-making under uncertainty...

Additionally, as the name suggests, machine learning uses machines, in most cases computers, to solve specific problems.

One example of a problem that machine learning can be used for is to recognize objects in a digital image. The set of possible objects in the images has to be specified beforehand, e.g., cats and dogs. To solve the task, a large number of cat and dog pictures is provided to a machine learning algorithm, with information about which images contain cats and which images contain dogs. The machine learning algorithm then trains a model that can distinguish between dog and cat pictures, even for pictures that are not

contained in the initial set of pictures. The algorithms that deal with finding a relation between known input (e.g., a digital image) and an output (e.g., "it's a dog") pairs are classified under *supervised learning*. Data that comes in input-output pairs is also called *labeled data*. Supervised learning is an essential tool in this thesis, therefore a section of this chapter is dedicated to this topic.

Another class of machine learning algorithms is concerned with decision-making in a dynamic environment. The outcome of these algorithms is called the policy or controller. A way to train a controller is provided by *reinforcement learning* algorithms. Reinforcement learning algorithms generate their own training data which is why reinforcement learning is often considered its own class of machine learning algorithms, despite the fact that supervised learning is an essential part of most reinforcement learning algorithms as well.

Another category of machine learning that is not in the focus of this thesis is *unsupervised learning*. Unsupervised learning deals with *unlabeled data*, i.e., the model to train isn't provided with reference outputs for an input. Tasks using only input data include for example clustering [112], compression and decompression [8] or anomaly detection [94].

The mathematics behind most machine learning algorithms are based on probability theory, therefore a short introduction into basic probability theory is provided before continuing with supervised learning and reinforcement learning.

### 2.1.1 Probability theory

Probability theory is a basic building block for understanding and deriving many methods of machine learning. It is a mathematical framework for describing uncertainty or belief. A brief introduction to random variables and probability mass functions is given in the following, based on [91, p. 21–55], followed by an introduction to the Kullback-Leibler divergence, which is a common metric for the similarity of two probability distributions, and to the likelihood function, used to express how well a probability model explains existing data.

### Random Variables

The uncertainty of an event is expressed by a random variable. A random variable, written sans serifs in this work $\mathsf{x}$, is a mapping from a random event $\omega$, e.g., a coin toss, to a value. The random variable doesn't have a value, but rather describes the possible values depending on the outcome of the event $\omega$. For example, a random value $\mathsf{x}$ for the event of tossing a coin could be

$$\mathsf{x}(\omega) = \begin{cases} 0 \text{ if the coin shows head} \\ 1 \text{ if the coin shows tails} \end{cases} . \tag{2.1}$$

The dependence on the random event $\omega$ is no longer written explicitly in the following. A random variable can be discrete or continuous, i.e., it can represent a finite or infinite number of values. The random variable in our coin tossing example is discrete (0 or 1), whereas an estimate of the temperature of tomorrow would be represented by a continuous random variable.

### Probabilities of random variables and expected value

The probability of each value of a random variable is described by its related probability distribution $P_{\mathsf{x}}$, which is a function that takes a possible value of the random variable $\mathsf{x}$ as input and returns a scalar. For the coin example and the random variable as described in (2.1), the probability of each event, given that the coin is fair, is

$$\begin{aligned} P_{\mathsf{x}}(0) &= 0.5 \\ P_{\mathsf{x}}(1) &= 0.5. \end{aligned} \tag{2.2}$$

The function $P_{\mathsf{x}}$ is called a *probability mass function* if the random variable $\mathsf{x}$ is *discrete*, i.e., represents a finite number of possible values. All probabilities have to be greater or equal to zero and a probability mass function always fulfills the equality

$$\sum_{x \in \mathsf{x}} P_{\mathsf{x}}(x) = 1. \tag{2.3}$$

The *expected value*, also called *mean* or first moment, of a discrete random variable can be calculated from its probability mass function as

$$\mathbb{E}[\mathsf{x}] = \sum_{x \in \mathsf{x}} x \cdot P_{\mathsf{x}}(x). \tag{2.4}$$

If the probabilities used for the expected value have not been specified or are not clear from the context, the probability distribution can be clarified in the index of the operator. Furthermore, the expected value of a function $f$ applied to a random variable's value can be calculated as

$$\mathbb{E}_{\mathsf{x}\sim P_{\mathsf{x}}(x)}\left[f(\mathsf{x})\right] = \sum_{x\in\mathsf{x}} f(x)\cdot P_{\mathsf{x}}(x). \tag{2.5}$$

The previous concepts of probability distribution and mean can be extended for multiple random variables. Let $\mathsf{x}$ and $\mathsf{y}$ be two discrete random variables. The probabilities of observing the value $x$ for $\mathsf{x}$ and $y$ for $\mathsf{y}$ is expressed by the *joint probability mass function* $P_{\mathsf{x},\mathsf{y}}(x,y)$. As for the scalar case, the function $P_{\mathsf{x},\mathsf{y}}(x,y)$ only contains values greater or equal to zero and fulfills

$$\sum_{x\in\mathsf{x}}\sum_{y\in\mathsf{y}} P_{\mathsf{x},\mathsf{y}}(x,y) = 1. \tag{2.6}$$

The expected value of a random variable is calculated from the joint probability mass function as

$$\mathbb{E}\left[\mathsf{x}\right] = \sum_{x\in\mathsf{x}} x\cdot\sum_{y\in\mathsf{y}} P_{\mathsf{x},\mathsf{y}}(x,y). \tag{2.7}$$

The function $P_{\mathsf{x}}(x) = \sum_{y\in\mathsf{y}} P_{\mathsf{x},\mathsf{y}}(x,y)$ describes a proper distribution for the random variable $\mathsf{x}$ and is called the *marginal distribution*. It describes the probabilities for each possible value of $\mathsf{x}$ without consideration of $\mathsf{y}$. Another distribution for $\mathsf{x}$ that can be derived from the joint distribution is the *conditional probability distribution* $P_{\mathsf{x}|\mathsf{y}}(x|y)$. It describes the probability for each value of $\mathsf{x}$, given the observation $y$ for $\mathsf{y}$. Or in other words, the value for $\mathsf{y}$ is known to be $y$. The conditional probability mass function can be expressed as

$$P_{\mathsf{x}|\mathsf{y}}(x|y) = \frac{P_{\mathsf{x},\mathsf{y}}(x,y)}{\sum_{x\in\mathsf{x}} P_{\mathsf{x},\mathsf{y}}(x,y)} = \frac{P_{\mathsf{x},\mathsf{y}}(x,y)}{P_{\mathsf{y}}(y)} \tag{2.8}$$

$$\Leftrightarrow P_{\mathsf{x},\mathsf{y}}(x,y) = P_{\mathsf{x}|\mathsf{y}}(x|y)\cdot P_{\mathsf{y}}(y). \tag{2.9}$$

The random variables $\mathsf{x}$ and $\mathsf{y}$ are called *independent*, if $P_{\mathsf{x}|\mathsf{y}}(x|y) = P_{\mathsf{x}}(x)$. The joint probability distribution is in that case simply the product of the individual probability distributions $P_{\mathsf{x},\mathsf{y}}(x,y) = P_{\mathsf{x}}(x)\cdot P_{\mathsf{y}}(y)$.

For *continuous* random variables, the probability distribution is called the *probability density function*. A probability density function has to be at least positive semi-definite and fulfill

$$\int_{-\infty}^{\infty} P_{\mathsf{x}}(x)dx = 1. \tag{2.10}$$

An important distribution for this thesis is the *normal distribution*, also called Gaussian distribution. The distribution is characterized by its mean $\mu$ and standard deviation $\sigma$ and is abbreviated as $\mathcal{N}(\mu, \sigma^2)$. The standard deviation is a measure for the spreading of the possible values around the mean. The probability density function of the normal distribution is defined as

$$P_{\mathsf{x}}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}. \tag{2.11}$$

Another continuous distribution that is used in this work is the uniform distribution. The uniform distribution, abbreviated $\mathcal{U}(a, b)$ assigns the same probability to all values in a range $[a, b]$, and zero probability to all other values. The probability density function is

$$P_{\mathsf{x}}(x) = \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{else} \end{cases} \tag{2.12}$$

Yet another widely used distribution is the *Dirac distribution* that assigns all probability to a single value. This distribution is used to represent deterministic values in the context of probabilities, and the underlying random variable is not random for this special case. The distribution is equal to the Dirac delta function $\delta(x - \mu)$ as

$$\delta(x-\mu) = \begin{cases} +\infty & \text{if } x = \mu \\ 0 & \text{else} \end{cases} \tag{2.13a}$$

$$1 = \int_{-\infty}^{\infty} \delta(x-\mu)dx. \tag{2.13b}$$

Unlike in the discrete case for probability mass functions, evaluating the probability density function for a given $x$ does not return the probability of

the value $x$. Probabilities are evaluated in intervals rather than for specific values in the continuous case. The probability $p_{a<x<b}$ of a value of $\mathsf{x}$ being in the interval $[a, b]$ is given by the integral

$$p_{a<x<b} = \int_a^b P_{\mathsf{x}}(x)dx. \tag{2.14}$$

The *expected value* of a continuous random variable is computed as

$$\mathbb{E}[\mathsf{x}] = \int_{-\infty}^{\infty} x \cdot P_{\mathsf{x}}(x)dx \tag{2.15}$$

and furthermore

$$\mathbb{E}[f(\mathsf{x})] = \int_{-\infty}^{\infty} f(x) \cdot P_{\mathsf{x}}(x)dx. \tag{2.16}$$

The marginal and conditional probability distributions in the multivariate case are defined the same way as in the case of discrete random variables, with sums replaced by integrals.

$$P_{\mathsf{x}}(x) = \int_{-\infty}^{\infty} P_{\mathsf{x},\mathsf{y}}(x, y)dy \tag{2.17}$$

$$P_{\mathsf{x}|\mathsf{y}}(x|y) = \frac{P_{\mathsf{x},\mathsf{y}}(x, y)}{P_{\mathsf{y}}(y)} \tag{2.18}$$

For both discrete and continuous probability distributions, the expected value of the distribution can be approximated if samples $x_i$ of the random variable are available, with $i \in \{1, \ldots, N\}$. The *sample mean* $\bar{\mathsf{x}}$ is

$$\bar{\mathsf{x}} = \frac{1}{N} \sum_i x_i. \tag{2.19}$$

The sample mean returns the true expected value of the distribution for $N \to \infty$.

The number of random variables can be large, therefore the notation can be shortened by combining the random variables in a random vector with the bold symbol notation $\mathbf{x}$.

The normal distribution for a random vector $\mathbf{x}$ of dimension $n$ is called the *multivariate normal distribution* and is characterized by the mean $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$. The covariance matrix is a positive definite matrix that defines the spread of the distribution. An example of samples of

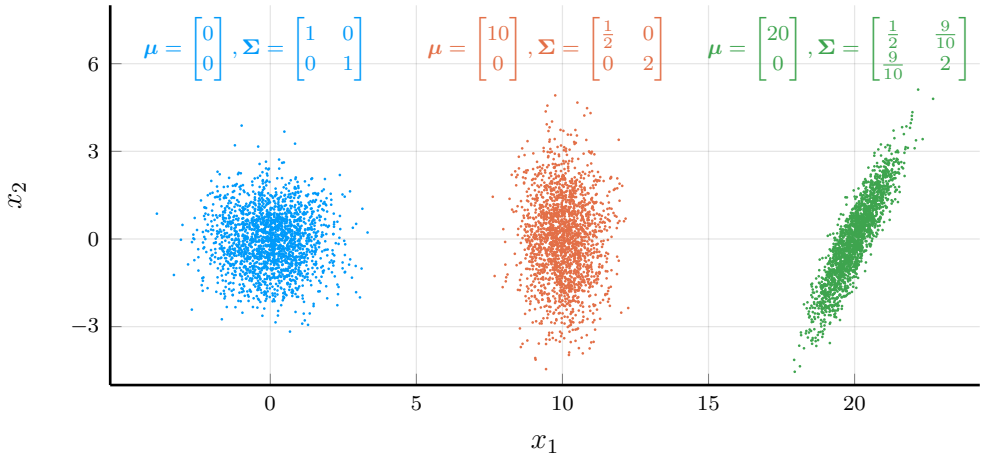



Figure 2.1: 1000 samples of $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ for different means $\boldsymbol{\mu}$ and covariance matrices $\boldsymbol{\Sigma}$.

a two-dimensional random vector for different covariance matrices is shown in figure 2.1. The short notation for this distribution is $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and the mathematical expression is

$$P_{\mathbf{x}}(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n \det(\boldsymbol{\Sigma})}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu})}. \tag{2.20}$$

### Kullback-Leibler divergence

In 1951, Kullback and Leibler introduced a measure for the similarity of two probability distributions [57]. This measure is known as relative entropy or Kullback-Leibler divergence. The Kullback-Leibler divergence from a probability distribution $P_2$ to a probability distribution $P_1$ is defined as

$$D_{KL}(P_1 || P_2) = \mathbb{E}_{\mathrm{x} \sim P_1} \left[ \log \frac{P_1(x)}{P_2(x)} \right]. \tag{2.21}$$

The Kullback-Leibler divergence is zero if $P_1(x) = P_2(x)$, otherwise positive. A smaller value of the divergence indicates a stronger similarity between distributions.

The Kullback-Leibler divergence can be used to create a distribution $P_2(x)$ that is similar to a known distribution $P_1(x)$. When $P_1(x)$ is known,

a parametric probability distribution $P_2(x;\boldsymbol{\Theta})$ can approximate $P_1(x)$ after minimizing the Kullback-Leibler divergence from $P_2$ to $P_1$. The best $\boldsymbol{\Theta}^*$ is found by solving

$$\begin{aligned}\boldsymbol{\Theta}^* &= \arg\min_{\boldsymbol{\Theta}} \mathbb{E}_{\mathsf{x}\sim P_1}\left[\log\frac{P_1(x)}{P_2(x;\boldsymbol{\Theta})}\right] \\ &= \arg\min_{\boldsymbol{\Theta}} \mathbb{E}_{\mathsf{x}\sim P_1}\left[\log P_1(x) - \log P_2(x;\boldsymbol{\Theta})\right] \\ &= \arg\max_{\boldsymbol{\Theta}} \mathbb{E}_{\mathsf{x}\sim P_1}\left[\log P_2(x;\boldsymbol{\Theta})\right]. \end{aligned} \tag{2.22}$$

### Likelihood and log-likelihood functions

The likelihood function for a parametric probability distribution $P(x;\boldsymbol{\Theta})$ is the function $L(\boldsymbol{\Theta};x)$, which is equal to the probability distribution. While having the same mathematical expression, the likelihood function is not a probability distribution in its first argument $\boldsymbol{\Theta}$ and $x$ is supposed to be a known value. In the case of a discrete probability distribution, the likelihood function evaluates the probability of a given $x$ for the parameters $\boldsymbol{\Theta}$. In the case of a probability density function, it expresses the density of a given $x$ for the parameters $\boldsymbol{\Theta}$. The log-likelihood function $l(\boldsymbol{\Theta};x)$ is the natural logarithm of the likelihood function.

The log-likelihood function is used to fit a parametric probability distribution $P(x;\boldsymbol{\Theta})$ to data $x_i$, for $i \in 1,\ldots,N$. If the samples $x_i$ are independent, the joint probability distribution $P(\mathbf{x};\boldsymbol{\Theta})$ of $N$ consecutive samples can be expressed as the product

$$P(\mathbf{x};\boldsymbol{\Theta}) = P(x_1;\boldsymbol{\Theta}) \cdot \ldots \cdot P(x_N;\boldsymbol{\Theta}). \tag{2.23}$$

The log-likelihood of the joint probability is

$$\begin{aligned} l(\boldsymbol{\Theta};\mathbf{x}) &= \log P(\mathbf{x};\boldsymbol{\Theta}) \\ &= \log P(x_1;\boldsymbol{\Theta}) + \ldots + \log P(x_N;\boldsymbol{\Theta}). \end{aligned} \tag{2.24}$$

Maximizing the log-likelihood can be seen as minimizing the Kullback-Leibler divergence from the parametric distribution $P(x;\boldsymbol{\Theta})$ and the true but unknown distribution of the data $P_1(x)$. This can be seen by comparing

equations (2.22) and (2.24). The sum and sample mean only differ by a constant factor $\frac{1}{N}$ which does not change the maximizing $\boldsymbol{\Theta}$.

$$\arg\max_{\boldsymbol{\Theta}} l(\boldsymbol{\Theta}; \mathbf{x}) = \arg\max_{\boldsymbol{\Theta}} \sum_{i}^{N} \log P(x_i; \boldsymbol{\Theta}) \tag{2.25a}$$

$$= \arg\max_{\boldsymbol{\Theta}} \frac{1}{N} \sum_{i}^{N} \log P(x_i; \boldsymbol{\Theta}) \tag{2.25b}$$

### 2.1.2 Supervised learning

The goal of supervised learning is to learn input to output relations, based on a set of training examples. A classification example for dog and cat pictures that can be solved using supervised learning was given at the beginning of section 2.1. Another example of a supervised learning problem is linear regression, i.e., estimating a linear function given input data $x_i$ and related output data $y_i$. It is assumed that the relation between the inputs $x_i$ and outputs $y_i$ can be described by a model $y_i \approx f(x_i; \boldsymbol{\Theta}) = \Theta_1 \cdot x_i + \Theta_2$ with a sufficiently small error. If we assume that the error between our best model and the real data is distributed according to a normal distribution with zero mean and a standard deviation of $\sigma$, i.e.,

$$y_i = \Theta_1 \cdot x_i + \Theta_2 + \varepsilon;\ \varepsilon \sim \mathcal{N}(0, \sigma^2) \tag{2.26}$$

then our outputs are distributed according to $P(y|x; \boldsymbol{\Theta}) = \mathcal{N}(f(x; \boldsymbol{\Theta}), \sigma^2)$. In order to find the parameters $\boldsymbol{\Theta}$ that best explain the data, we look for the parameters that give the highest probability density to the data. This is solved by maximizing the log-likelihood for our distribution $P(y|x; \boldsymbol{\Theta})$.

$$\boldsymbol{\Theta}^* = \arg\max_{\boldsymbol{\Theta}} \sum_{i} \log \left( \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(y_i - f(x_i;\boldsymbol{\Theta}))^2} \right) \tag{2.27a}$$

$$= \arg\max_{\boldsymbol{\Theta}} \sum_{i} \left( -\frac{1}{2\sigma^2} \left(y_i - f(x_i; \boldsymbol{\Theta})\right)^2 \right) - \log(\sigma\sqrt{2\pi}) \tag{2.27b}$$

$$= \arg\min_{\boldsymbol{\Theta}} \sum_{i} \left(y_i - f(x_i; \boldsymbol{\Theta})\right)^2 \tag{2.27c}$$

The formulation in equation (2.27c) is called the least squares problem and an analytic solution is known for linear models as the one used in our example [75, p. 221-222]. The assumption of independent and normally distributed

data with unknown mean and constant standard deviation is common in curve fitting. For other tasks, such as classification, the least squares optimization is unsuited as the output data that is subdivided in a finite number of classes clearly is not distributed according to a normal distribution. The distribution that is parameterized in that case is called a Bernoulli or Multinoulli distribution, and a cost function can be derived the same way that we showed for the least squares formulation.

Later in the thesis, we use function approximation together with the assumption that our model error is distributed according to a normal distribution. However, we will use nonlinear function approximators for the mean of the distribution that do not allow an analytic solution for the optimization problem (2.25b). In this work, we use neural networks and recurrent neural networks and solve (2.25b) by applying state of the art methods for this class of function approximators.

### Structure of a static neural network

A neural network is a function depending on an input $\mathbf{x}$ and a set of parameters $\boldsymbol{\Theta}$ with a specific structure. The name neural network results from early analogies that were drawn between the structure of the function and the structure of mammal brains.

A neural network can typically be divided into multiple *layers*, each of which consist of a linear transformation of the input to the layer, followed by an optional nonlinear transformation called activation function. The input to a layer is either the input to the neural network or the output of a previous layer. An example of a single layer with input $\mathbf{x}$ and output $\mathbf{y}$ is

$$\mathbf{y} = \mathbf{a}(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.28}$$

with $\mathbf{a}$ a generic nonlinear function. Most often $\mathbf{a}$ is an element-wise application of a simple nonlinear function, e.g., tanh. The matrix $\mathbf{W}$ and the vector $\mathbf{b}$ are parameters of the layer and need to be of appropriate dimension.

A neural network is typically made up of multiple concatenated layers that are evaluated consecutively. The last evaluated layer is called the output layer and all layers before are called hidden layers. The number of layers of a neural network is referred to as its *depth*. The dimension of the input

and output vectors of the neural network is specified by the data, however, the depth of the network and the dimension of the outputs of the hidden layer can be freely defined and are related to its *capacity*. The capacity of a parameterized function is its ability to fit a variety of functions [34].

As an example, a neural network with one hidden layer using the function $\mathbf{a}_1(\mathbf{x}; \boldsymbol{\Theta}_1)$ and an output layer $\mathbf{a}_2(\mathbf{x}; \boldsymbol{\Theta}_2)$ is evaluated as

$$\mathbf{y} = \mathbf{a}_2(\mathbf{a}_1(\mathbf{x}; \boldsymbol{\Theta}_1); \boldsymbol{\Theta}_2) = (\mathbf{a}_2 \circ \mathbf{a}_1)(\mathbf{x}). \tag{2.29}$$

### Underfitting and overfitting

The necessary capacity of a neural network depends on the task at hand and can only be guessed in a first step. If the capacity is chosen too low, the function can not approximate the data with a sufficient accuracy. If the capacity is chosen too high and data is insufficient or noisy, the function approximator will not interpolate well in regions between data. Training a function approximator with too low capacity leads to *underfitting*, i.e., a large error on the training data and bad interpolation quality, whereas a function approximator with too high capacity can lead to *overfitting*, i.e., a small error on the training data but bad interpolation quality. Figure 2.2 visualizes the effects on a one-dimensional example. The left plot in figure 2.2 shows a neural network with 7 parameters that is underfitting the data and the real function. The middle plot is a neural network with 13 parameters that achieved a better fit on both data and function compared to the smaller neural network. The last plot shows a neural network with 321 parameters that has the lowest error on the data, however, the approximation of the real function is poor in interpolating regions.

While underfitting is visible during training by observing the error on the training data, overfitting can only be observed by using multiple independent data sets. It is common practice in supervised learning to not use all data to fit the parameters of the function. Instead the data is split into a training data-set and a complementary test data-set. Only the training data-set is used to fit the parameters, e.g., by solving equation (2.27c). The error is then evaluated on the test data-set. If the error in the test data-set increases, the training is interrupted to avoid overfitting. This procedure is called *hold-*

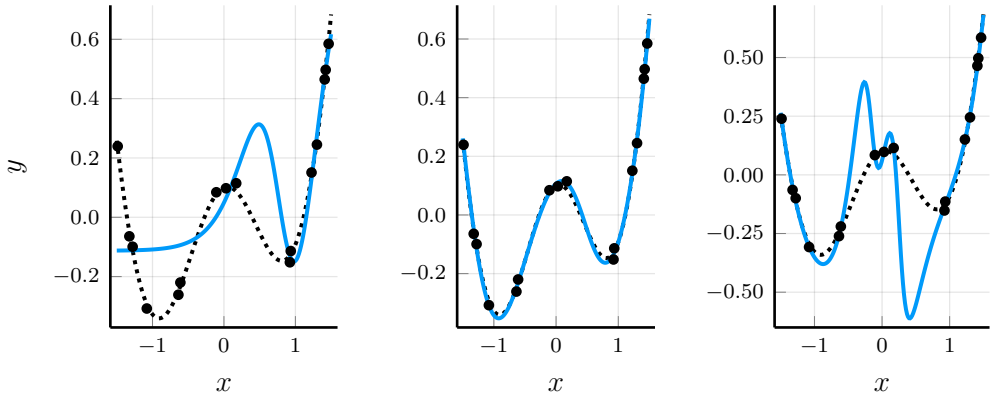



Figure 2.2: Visualization of data-based function approximation using neural networks with different capacities. Training data with small noise is shown as dots, the real function is a black dotted line. Neural network approximations are shown in blue.

*out validation*. If training continues and the error is significantly higher on the test data-set than on the training data-set, overfitting occurred. Hold-out validation is used throughout the thesis for every supervised learning problem, even if it is not always explicitly mentioned.

### Training of neural networks

We have seen that we can approximate distributions by maximizing the log-likelihood in equation (2.25b) given data of the true distribution. In the following, we use a neural network as a function approximator that describes a characteristic of our distribution, e.g., the expected value. While no analytic solution of equation (2.25b) is available for generic neural networks, the unconstrained optimization problem can be solved numerically. Algorithms to solve this kind of optimization problem are known for centuries [53]. Most algorithms require the gradient of the objective function with respect to the parameters, i.e., $\nabla_{\boldsymbol{\Theta}} l(\boldsymbol{\Theta}, \mathbf{x}, \mathbf{y})$. Some algorithms additionally require the Hessian matrix. There are also algorithms that do not need a gradient information, e.g., [36, 109], however, those algorithms require more function evaluations and should not be used if the gradient can be evaluated efficiently.

As of today, first order methods, i.e., methods that only require the gradient, are state of the art for large neural networks. Computing the Hessian matrix is avoided as the number of entries in the Hessian matrix is equal to the number of parameters to the square. A simple yet common algorithm for training neural networks is *gradient ascent*. Pseudo-code for a gradient ascent algorithms for the training of a function approximator on data is given in algorithm 1. The approach consists in repeatedly calculating the gradient $\mathbf{g}(\boldsymbol{\Theta}; \mathbf{D}) = \nabla_{\boldsymbol{\Theta}} l(\boldsymbol{\Theta}; \mathbf{D})$, with $\mathbf{D}$ a matrix containing all input-output pairs used during training

$$\mathbf{D} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_N \\ \mathbf{y}_1 & \mathbf{y}_2 & \dots & \mathbf{y}_N \end{bmatrix}. \tag{2.30}$$

The parameters are updated as $\boldsymbol{\Theta}_{new} = \boldsymbol{\Theta}_{old} + \alpha \cdot \mathbf{g}(\boldsymbol{\Theta}_{old}, \mathbf{D})$ with a constant $\alpha$ that is called the step size. Note that in the case of a minimization, e.g., a problem formulation as in equation (2.27c), the method is called gradient descent, and the update becomes $\boldsymbol{\Theta}_{new} = \boldsymbol{\Theta}_{old} - \alpha \cdot \mathbf{g}(\boldsymbol{\Theta}_{old}, \mathbf{x})$. If a lot of

**Algorithm 1** Gradient ascent

**Require:** $\alpha, \mathbf{D} = \left[ [\mathbf{x}_1^T, \mathbf{y}_1^T]^T, \dots, [\mathbf{x}_N^T, \mathbf{y}_N^T]^T \right], l(\boldsymbol{\Theta}; \mathbf{D})$

1: Split $\mathbf{D}$ in $\mathbf{D}_{train}$ and $\mathbf{D}_{test}$
2: **while** (No stopping criterion active) **do**
3: $\quad \mathbf{g} \leftarrow \nabla_{\boldsymbol{\Theta}} l(\boldsymbol{\Theta}; \mathbf{D}_{train})$ ▷ Calculate gradient
4: $\quad \boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} + \alpha \cdot \mathbf{g}$ ▷ Update parameters
5: $\quad L \leftarrow l(\boldsymbol{\Theta}; \mathbf{D}_{test})$ ▷ Calculate the log-likelihood of test data
6: $\quad$ Check stopping criteria
7: **end while**
8: **return** $\boldsymbol{\Theta}$

data is used during training, using all the training data to compute a gradient direction can become inefficient. A subset of the training data, called a *mini-batch*, is extracted for the gradient direction. A different subset is used every iteration. When all mini-batches have been used, the training set is shuffled and new mini-batches are created. One use of all available data is called a training *epoch*. As such, one epoch consists of multiple parameter updates if the data is split in mini-batches. The gradient direction calculated using a

mini-batch is not the same as the direction that results when using the whole training set. However, the mean of the gradient direction is unbiased, i.e., for a subset $\mathbf{D}_m$ of uniformly sampled columns of a matrix of training data $\mathbf{D}$, we have

$$\mathbb{E}\left[\nabla l(\boldsymbol{\Theta}; \mathbf{D}_m)\right] = \nabla l(\boldsymbol{\Theta}; \mathbf{D}) \tag{2.31}$$

[96]. The modification of gradient ascent using mini-batches is called *mini-batch stochastic gradient ascend*, or only stochastic gradient ascent, e.g., [34, p. 147].

Guessing a good step size $\alpha$ for algorithm 1 is crucial for a fast convergence to a good optimum. Nowadays, more efficient update rules are used that either modify the update direction or the step size, or both. The most common modifications are called momentum [88], Nesterov momentum [77], Adagrad [26], Adadelta [116] and Adam [52]. We only use and present Adam in this work. Adam calculates moving averages of the mean and squared mean of the gradient, and uses those in its update. Pseudo-code for training a neural network using mini-batches and Adam is given in algorithm 2. The parameter $\alpha$ is the learning rate, the parameters $\beta_1$ and $\beta_2$ are responsible for the exponential decay of the moving average.

The pseudo-code of algorithm 2 only consists of a few lines, however, manually writing code to compute the gradient $\nabla_{\boldsymbol{\Theta}} l(\boldsymbol{\Theta}; \mathbf{D}_{train})$ can be tedious for deep neural networks. Luckily, algorithms exist that can automate the generation of the gradient, called *automatic differentiation*. Those algorithms track the operations that are used to evaluate the function $l(\boldsymbol{\Theta}; \mathbf{D}_{train})$ and use the partial derivatives of each operation and the chain rule of derivatives to compute the gradient. While there are different approaches for automatic differentiation, an efficient algorithm for neural networks is the *backpropagation* algorithm. The backpropagation algorithm is a reverse mode automatic differentiation algorithm, which means that it first evaluates the function completely while saving important intermediate results and only afterwards builds the gradient, starting with the last operations going backwards to the first operations. Reverse mode automatic differentiation is in general efficient for calculating the gradient of functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \ll n$. This is the case for the log-likelihood function $l(\boldsymbol{\Theta}; \mathbf{D})$ with input dimension $n$ equal

**Algorithm 2** Gradient ascent using Adam

**Require:** $\alpha, \beta_1, \beta_2, \varepsilon, \mathbf{D} = \left[[\mathbf{x}_1^T, \mathbf{y}_1^T]^T, \ldots, [\mathbf{x}_N^T, \mathbf{y}_N^T]^T\right], l(\boldsymbol{\Theta}; \mathbf{D})$

Split $\mathbf{D}$ in $\mathbf{D}_{train}$ and $\mathbf{D}_{test}$
$\mathbf{m} \leftarrow \mathbf{0}$
$\mathbf{v} \leftarrow \mathbf{0}$
$i \leftarrow 0$
**while** (No stopping criterion active) **do**
    Split $\mathbf{D}_{train}$ into multiple $\mathbf{D}_m$
    **for** each $\mathbf{D}_m$ in $\mathbf{D}_{train}$ **do**
        $i \leftarrow i + 1$
        $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\Theta}} l(\boldsymbol{\Theta}; \mathbf{D}_m)$ ▷ Calculate gradient
        $\mathbf{m} \leftarrow \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \mathbf{g}$ ▷ Moving average of the gradient
        $\mathbf{v} \leftarrow \beta_2 \cdot \mathbf{v} + (1 - \beta_2) \cdot \mathbf{g} \odot \mathbf{g}$ ▷ Moving avg. of squared gradient
        $\hat{\mathbf{m}} \leftarrow \mathbf{m}/(1 - \beta_1^i)$ ▷ Bias correction
        $\hat{\mathbf{v}} \leftarrow \mathbf{v}/(1 - \beta_2^i)$ ▷ Bias correction
        $\Theta_j \leftarrow \Theta_j + \alpha \cdot \hat{m}_j/(\sqrt{\hat{v}_j} + \varepsilon), \ \forall j$ ▷ Update parameters
    **end for**
    $L \leftarrow l(\boldsymbol{\Theta}; \mathbf{D}_{test})$ ▷ Calculate the log-likelihood of test data
    Check stopping criteria
**end while**
**return** $\boldsymbol{\Theta}$

to the number of parameters of the neural network and $m = 1$. Libraries for backpropagation are freely available in many popular programming languages. Throughout this work, we use the AutoGrad library, presented in [115], in the programming language Julia [11]. The history of the development of backpropagation is summarized in [95].

The runtime of training large neural networks can be reduced further by using appropriate hardware. The computationally most expensive part when training neural networks is the repeated computation of the gradient, which mostly consists of operations on matrices and vectors during the forward and backward evaluation of the layers in equation (2.28). Operations on large matrices can be accelerated by performing them on a graphics card (GPU for graphics processing unit) instead of using the central processing

unit (CPU). A GPU is optimized for performing one operation on a large amount of data whereas a CPU is treating data in very small chunks. Due to the large benefit of using appropriate hardware, special processing units for tensors (TPU) have recently been designed by Google. TPUs are more efficient still than current GPUs for training neural networks [46].

Training neural networks with a large amount of layers as they are currently used for computer vision or language processing, e.g., [16, 35], requires additional practices to achieve good training results which include batch normalization [45] or regularization using dropout [42] among other things. We found that our control problems did not require deep neural networks, therefore our training does not use deep learning methods.

### Recurrent neural networks

So far we have assumed that the data tuples $(\mathbf{x_i}, \mathbf{y_i})$ are independent from all other data tuples, i.e., $\mathbf{y_i}$ depends only on $\mathbf{x_i}$. However, the data can contain measurements over time, e.g., input data $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_T]$ with related outputs $\mathbf{Y} = [\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_T]$ and each $\mathbf{y}_t$ possibly depending on all past inputs $\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_0$. The index $t$ is used here instead of $i$ to indicate a value in discrete time. Thus, the order of the data matters in this case. While $\mathbf{y}_t$ is not independent from $\mathbf{y}_{t-1}$, i.e., $P(\mathbf{y}_t|\mathbf{y}_{t-1}) \neq P(\mathbf{y}_t)$ we assume that their conditional distributions given the past input data $\mathbf{X}$ are independent, i.e.,

$$P(\mathbf{y}_t|\mathbf{y}_{t-1}, \mathbf{x}_t, \ldots, \mathbf{x}_0) = P(\mathbf{y}_t|\mathbf{x}_t, \ldots, \mathbf{x}_0). \tag{2.32}$$

**Example 1.** Consider the dynamical system

$$x_t = x_{t-1} + \frac{3}{5}\tanh(x_{t-2}) + y_{t-1} \tag{2.33a}$$

$$y_t = -\frac{1}{6}(4x_t + x_{t-1}) + \varepsilon_t;\ \varepsilon_t \sim \mathcal{N}(0, 0.1^2). \tag{2.33b}$$

We assume $x_t = y_t = 0, \forall t < 0$. The system (2.33a) describes a dynamical system with the state variable $x$ and input variable $y$[1]. Equation (2.33b) describes a feedback control law of stochastic nature. Three sampled trajectories of the stochastic system are visualized in figure 2.3. The distribution of

[1]Later in the thesis, the input variable to dynamic systems is the variable $u$. We use $y$ here to be consistent with the notation used for probability distributions so far.

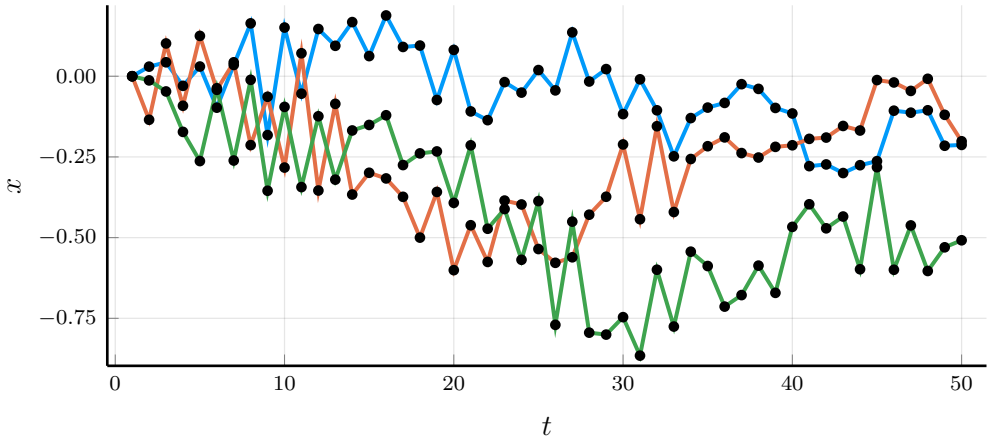



Figure 2.3: Sampled state trajectories $[x_0, x_1, \ldots, x_{49}]$ of the system (2.33).

$y$ is given in (2.33b) as $P(y_t|x_t, x_{t-1}) = \mathcal{N}(\frac{1}{6}(4x_t + x_{t-1}), 0.1^2)$ and depends only on the past two states. Condition (2.32) is fulfilled as knowing the value of $y_{t-1}$ doesn't provide additional information for the distribution of $y_t$.

Replacing (2.33b) with

$$y_t = -\frac{1}{6}(4x_t + y_{t-1}) + \varepsilon_t;\ \ \varepsilon_t \sim \mathcal{N}(0, 0.1^2). \tag{2.34}$$

would violate condition (2.32) as the resulting distributions

$$P(y_t|x_t, x_{t-2}, \ldots) = \mathcal{N}\left(\sum_{\tau=0}^{t} \frac{4x_\tau}{(-6)^{t-\tau+1}}, 0.1^2 \cdot \sum_{\tau=0}^{t} 6^{\tau-t}\right) \tag{2.35a}$$

$$P(y_t|y_{t-1}, x_t, x_{t-2}, \ldots) = \mathcal{N}\left(-\frac{1}{6}(4x_t + y_{t-1}), 0.1^2\right). \tag{2.35b}$$

are not equal. To derive (2.35a), we assumed $y_t = 0, \forall t < 0$ and recursively inserted the right hand side of expression (2.34) for $y_{t-1}, y_{t-2}, \ldots, y_0$. The resulting expression is then simplified using the fact that the sum of normally distributed random variables is also normally distributed [28]. To successfully train a recurrent neural network for the system using (2.34), a new state vector $\hat{\mathbf{x}}_t = [x_t, y_{t-1}]$ that fulfills the condition (2.32) could be used. ◇

Under the given assumption of independent distributions, the probability density of our data is

$$\begin{aligned} P(\mathbf{Y}|\mathbf{X}) =& P(\mathbf{y}_T|\mathbf{x}_T, \mathbf{x}_{T-1}, \ldots, \mathbf{x}_0) \cdot P(\mathbf{y}_{T-1}|\mathbf{x}_{T-1}, \mathbf{x}_{T-2}, \ldots, \mathbf{x}_0) \\ & \cdot \ldots \cdot P(\mathbf{y}_1|\mathbf{x}_1, \mathbf{x}_0) \cdot P(\mathbf{y}_0|\mathbf{x}_0). \end{aligned} \tag{2.36}$$

We could fit our data sequence by parameterizing characteristics of each distribution $P(\mathbf{y}_0|\mathbf{x}_0;\boldsymbol{\Theta}_1), P(\mathbf{y}_1|\mathbf{x}_1,\mathbf{x}_0;\boldsymbol{\Theta}_2),\ldots$ individually and maximizing the log-likelihood in equation (2.25b), however, this results in a model that is impractically large for long sequences. A more efficient parameterized representation is obtained by using a dynamic system of the form

$$\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t). \tag{2.37}$$

The vector $\mathbf{h}_t$ contains relevant information of the past history ($\mathbf{x}_t, \mathbf{x}_{t-1}$, $\mathbf{x}_{t-2}$, $\ldots, \mathbf{x}_0$). Since $\mathbf{h}_{t-1}$ contains all relevant information of the past, the following $\mathbf{h}_t$ only depends on the previous $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$

$$\begin{aligned}\mathbf{h}_t &= \tilde{\mathbf{f}}(\mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots, \mathbf{x}_0) \\ &= \mathbf{f}(\mathbf{h}_{t-1}, \mathbf{x}_t).\end{aligned} \tag{2.38}$$

In terms of recurrent neural networks, $\mathbf{h}_t$ is called the hidden state and the function $\mathbf{f}(.)$ is called the recurrent layer. The characteristics of the distributions in (2.36) can now be represented by reusing the same recurrent function for each distribution $P(\mathbf{y}_0|\mathbf{x}_0;\boldsymbol{\Theta}_1), P(\mathbf{y}_1|\mathbf{x}_1,\mathbf{x}_0;\boldsymbol{\Theta}_2),\ldots$. One or multiple layers are generally added to the recurrent layer, e.g., we could represent the mean of the distributions as

$$\mathbf{h}_t = \mathbf{a}_1(\mathbf{x}_t, \mathbf{h}_{t-1}; \boldsymbol{\Theta_1}) \tag{2.39a}$$

$$\mathbf{y}_t = \mathbf{a}_2(\mathbf{h}_t; \boldsymbol{\Theta_2}) \tag{2.39b}$$

with $\mathbf{a}_1(.)$ the recurrent layer and $\mathbf{a}_2$ the output layer. The output layer is typically a static layer following equation (2.28). Popular choices for the recurrent layer is the tanh-recurrent layer

$$\mathbf{h}_t = \mathbf{tanh}_{\odot}(\mathbf{W}_x\mathbf{x}_t + \mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{b}) \tag{2.40}$$

with the $\mathbf{tanh}_{\odot}(.)$ function applying tanh element-wise to its arguments. Other popular and more complex recurrent layers are, e.g., the long short-term memory layer [43] or the gated recurrent unit [22]. Those layers have shown improved results compared to the recurrent tanh layer in equation (2.40) for certain tasks like speech recognition [17].

Building the gradient of the log-likelihood given data sequences can also be automated by using backpropagation, however, backpropagation requires the forward and backward evaluation of the recurrent neural network over sequences of data. The algorithm is called *backpropagation through time* when used on data-sequences. The reuse of the parameters $\mathbf{\Theta_1}$ for each time-step in equation (2.39a) can lead to ill-conditioned gradients for long data sequences. The gradient can become very large, which is called the exploding gradient problem, or close to zero, which is called the vanishing gradient problem. In fact, training recurrent neural networks on long sequences is similar to training very deep neural networks; more details on problems with training recurrent neural networks are given in [10, 81]. Both exploding and vanishing gradients are undesirable during training. To avoid those problems, a correct initialization of the parameters is necessary. We initialize our parameters according to the Xavier-initialization, presented in Glorot and Bengio [31]. Further measures against ill-conditioned gradients include skip-connections for vanishing gradients [118], or gradient clipping for exploding gradients [81]. Gradient clipping is used later in this work. The general approach is to verify that a norm, usually the $L^2$ norm, is within a certain range $[0, \nu]$. If the norm exceeds the limit, the vector is scaled to have the norm $\nu$

$$\mathbf{g}_{clip} = \begin{cases} \nabla l(\mathbf{\Theta}) & \text{if } \|\nabla l(\mathbf{\Theta})\| \leq \nu \\ \nu \frac{\nabla l(\mathbf{\Theta})}{\|\nabla l(\mathbf{\Theta})\|} & \text{if } \|\nabla l(\mathbf{\Theta})\| > \nu \end{cases}. \tag{2.41}$$

Building gradients over long sequences not only leads to ill-conditioned gradients, but also to an increased computing time and memory usage. Therefore, long sequences are often truncated into multiple shorter sequences, i.e., a sequence $\mathbf{S}$ of length $T$ is truncated into multiple sequences $\mathbf{S}_i$ of a

reduced length $\tau < T$ with

$$\mathbf{S} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_T \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_T \end{bmatrix} \tag{2.42a}$$

$$\mathbf{S}_1 = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_\tau \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_\tau \end{bmatrix} \tag{2.42b}$$

$$\mathbf{S}_2 = \begin{bmatrix} \mathbf{x}_{\tau+1} & \mathbf{x}_{\tau+2} & \dots & \mathbf{x}_{2\cdot\tau} \\ \mathbf{y}_{\tau+1} & \mathbf{y}_{\tau+2} & \cdots & \mathbf{y}_{2\cdot\tau} \end{bmatrix} \tag{2.42c}$$

...

While this looks similar to the mini-batch idea for static neural networks, truncating long sequences leads to a bias in the gradient direction and thus convergence to a local optimum of (2.25b) is not guaranteed, as opposed to gradient descent using the complete sequence for training [102]. Despite the biased gradient, the advantages of truncating sequences are predominant and truncating long sequences is common practice. Backpropagation used in combination with truncated time sequences is called *truncated backpropagation through time* (TBPTT).

### 2.1.3 Reinforcement learning

Reinforcement learning is often considered a distinct category of machine learning algorithms next to supervised and unsupervised learning. We first introduce the main idea of reinforcement learning using the wording of the reinforcement learning community and then create the references to the wording of the control community that is used in the rest of the thesis.

The task of reinforcement learning is to find the optimal way to make decisions in a dynamic environment via trial and error. The learning setting contains the decision taking unit, called agent, that acts according to a policy in a possibly unknown, dynamic environment in discrete time. The interactions between the agent and the environment are visualized in figure 2.4. In each time-step, the agent perceives the state $\mathbf{x}_t$ of the environment upon which he chooses an action $\mathbf{u}_t$. The state $\mathbf{x}_t$ is assumed to be a Markov state, which means that the probabilities of the next state $\mathbf{x}_{t+1}$ only depends

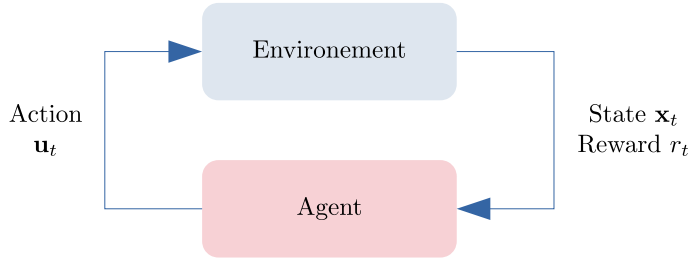



Figure 2.4: Reinforcement learning setting.

on $\mathbf{x}_t$ and $\mathbf{u}_t$ whereas the past states $\mathbf{x}_{t-1}, \ldots, \mathbf{x}_0$ are irrelevant. This can also be expressed as

$$P(\mathbf{x}_{t+1}|\mathbf{u}_t, \mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \ldots) = P(\mathbf{x}_{t+1}|\mathbf{u}_t, \mathbf{x}_t). \tag{2.43}$$

The agent also receives a scalar reward $r_t$ that can depend on the previous state and action, as well as the current state: $r_t = -c(\mathbf{x}_t, \mathbf{u}_{t-1}, \mathbf{x}_{t-1})$. The function $c(.)$ does not need to be known by the agent in order to improve his policy. In real problems, however, $c(.)$ is known as it is designed by the control engineer for the task.

The agent is trained using interactions with the environment by trying out different actions $\mathbf{u}_t$ and updating his policy. The states and actions can be either discrete and finite, e.g., board games like chess that allow a finite amount of possible positions, or continuous, e.g., the position of a point mass in a three-dimensional space. The goal is to find the optimal policy that is maximizing the accumulated reward over time, given a distribution on the starting state $\mathbf{x}_0 \sim P_{\mathbf{x}0}(\mathbf{x})$.

The algorithms and details of reinforcement learning are different in the case of discrete variables than in the case of continuous variables. We focus on the case of continuous states and actions, i.e., $\mathbf{x} \in \mathbb{R}^n, \mathbf{u} \in \mathbb{R}^m$ for which the problem is an optimal control problem. We will adopt the settings and notation of the control community and call the agent and policy a controller, the environment a system or plant and we will minimize accumulated costs $c_t = c(\mathbf{x}_t, \mathbf{u}_{t-1}, \mathbf{x}_{t-1})$ instead of maximizing rewards. The controller $\mathbf{u}_t \sim P(\mathbf{u}|\mathbf{x}_t; \boldsymbol{\Theta})$ is a function approximator, e.g., a neural network, that maps the state to an action. The control parameters are optimized to return the

actions that minimize the expected future costs. The optimization problem is

$$\boldsymbol{\Theta}^* = \arg\min_{\boldsymbol{\Theta}} J(\boldsymbol{\Theta}) \tag{2.44}$$

with

$$J(\boldsymbol{\Theta}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \cdot c_t\right] \tag{2.45a}$$

$$s.t.\ \mathbf{u}_t \sim P(\mathbf{u}|\mathbf{x}_t; \boldsymbol{\Theta}) \tag{2.45b}$$

$$\mathbf{x}_{t+1} \sim P(\mathbf{x}|\mathbf{x}_t, \mathbf{u}_t) \tag{2.45c}$$

$$\mathbf{x}_0 \sim P_{\mathsf{x}0}(\mathbf{x}). \tag{2.45d}$$

The constant $0 < \gamma \leq 1$ is called the discount factor and is used to bound the accumulated costs $\sum_{t=0}^{\infty} \gamma^t \cdot c_t$ that are possibly infinite for $\gamma = 1$, depending on the problem. Equations (2.45b) and (2.45c) indicate that the control law and system dynamics can be stochastic. Some randomness is required for reinforcement learning, as it is based on trying out different actions, however, the dynamics may also be deterministic. The algorithms of reinforcement learning for continuous variables compute an estimate of the gradient $\nabla_{\boldsymbol{\Theta}} J$ and are therefore called *policy gradient algorithms*. A specific policy gradient algorithm called proximal policy optimization is presented in section 2.3.

## 2.2 Trajectory optimization for discrete time systems

This section gives a short introduction into a method to optimize of trajectories for nonlinear, time-discrete systems of the form

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t) \tag{2.46}$$

on a finite time horizon $t \in 0, \ldots, T$. The goal is to find the trajectory, i.e., the best states $\mathbf{X}^* = [\mathbf{x}_1^*, \mathbf{x}_2^*, \ldots, \mathbf{x}_T^*]$ and inputs $\mathbf{U}^* = [\mathbf{u}_0^*, \mathbf{u}_1^*, \ldots, \mathbf{u}_{T-1}^*]$, that minimizes a given scalar cost function $J(\mathbf{X}, \mathbf{U})$ under given constraints. The initial state $\mathbf{x}_0$ is known and not included in the decision variables.

The methods to solve such problems can be divided in indirect and direct methods. Indirect methods focus on necessary conditions for the optimality of a trajectory and solve a boundary value problem. Direct methods formulate the dynamics of the system in equation (2.46) as equality constraints and solve the resulting static optimization problem. We are using a direct method in this thesis. Direct methods are known to converge better for poor initial guesses of the solution and an easier handling of constraints. However, indirect methods in general provide more accurate solutions. [80, p. 365]

For the presented approach, the decision variables consist of all the states $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T]$ and inputs $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{T-1}]$ of the trajectory. The static optimization problem to solve is

$$\mathbf{X}^*, \mathbf{U}^* = \arg\min_{\mathbf{X},\mathbf{U}} J(\mathbf{X}, \mathbf{U}) \tag{2.47a}$$

$$s.t.\ \mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t) \tag{2.47b}$$

$$\mathbf{ec}(\mathbf{X}, \mathbf{U}) = \mathbf{0} \tag{2.47c}$$

$$\mathbf{iec}(\mathbf{X}, \mathbf{U}) \geq \mathbf{0}. \tag{2.47d}$$

Equation (2.47c) contains equality constrains, additional to the $T$ equality constraints in equation (2.47b). Equation (2.47d) contains all the inequality constrains. In the following, we show how to numerically solve the constrained optimization problem.

### 2.2.1 Necessary conditions for optimality

To simplify the notation in this section, the problem is formulated in a simpler but equivalent form

$$\mathbf{x}^* = \arg\min_{x} J(\mathbf{x}) \tag{2.48a}$$

$$\mathbf{ec}(\mathbf{x}) = \mathbf{0} \tag{2.48b}$$

$$\mathbf{iec}(\mathbf{x}) \geq \mathbf{0} \tag{2.48c}$$

with the vector $\mathbf{x}$ containing all decision variables. The dimensions are assumed $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{ec}(\mathbf{x}) \in \mathbb{R}^m$ and $\mathbf{iec}(\mathbf{x}) \in \mathbb{R}^p$.

In a fist step, we only consider equality constrains $\mathbf{ec}(\mathbf{x}) = \mathbf{0}$. It can be shown that in every local minimum $\mathbf{x}^*$, the gradient of the loss function $J$

must be a linear combination of the gradients of each equality constraint [80, p. 69]:

$$\nabla_{\mathbf{x}} J(\mathbf{x}^*) = -\sum_{i=1}^{m} \lambda_i^* \nabla_{\mathbf{x}} \mathrm{ec}_i(\mathbf{x}^*) \tag{2.49}$$

The factors $\lambda_i$ are called the Lagrange multipliers. This necessary condition provides $n$ equations with $m$ new variables $\lambda_1, \ldots, \lambda_m$. We arrive at $n+m$ equations with $n+m$ variables when coupling the equations (2.48b) and (2.49). The system of equations that can be solved to find a local minimum to the optimization problem under equality constraints is

$$\nabla_{\mathbf{x}} J(\mathbf{x}) + \frac{\partial \mathbf{ec}(\mathbf{x})^T}{\partial \mathbf{x}} \boldsymbol{\lambda} = \mathbf{0} \tag{2.50a}$$

$$\mathbf{ec}(\mathbf{x}) = \mathbf{0}. \tag{2.50b}$$

Equations (2.50) can also be expressed by defining the Lagrange function $L(\mathbf{x}, \boldsymbol{\lambda})$ and looking for its local minimum, i.e., zero gradient with respect to its arguments

$$L(\mathbf{x}, \boldsymbol{\lambda}) = J(\mathbf{x}) + \boldsymbol{\lambda}^T \mathbf{ec}(\mathbf{x}) \tag{2.51a}$$

$$\nabla_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}) = \mathbf{0} \tag{2.51b}$$

$$\nabla_{\boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\lambda}) = \mathbf{0}. \tag{2.51c}$$

Any numerical method to solve systems of equations can be used to solve (2.51), e.g., gradient descent, Newtons method, etc. [80, p. 69–70]

Similar conditions can be established for local minima when inequality constraints (2.48c) are present, called Karush-Kuhn-Tucker conditions [80, p. 79-81] . The derivation is similar to the case that considers equality constraints only. The inequality constraints are split into active constraints, i.e., $\mathrm{iec}_i(\mathbf{x}) = 0$, and inactive constrains, i.e., $\mathrm{iec}_j(\mathbf{x}) > 0$ for the derivation. The resulting conditions are

$$\nabla_{\mathbf{x}} J(\mathbf{x}) + \frac{\partial \mathbf{ec}(\mathbf{x})^T}{\partial \mathbf{x}} \boldsymbol{\lambda} - \frac{\partial \mathbf{iec}(\mathbf{x})^T}{\partial \mathbf{x}} \boldsymbol{\mu} = \mathbf{0} \tag{2.52a}$$

$$\mathbf{ec}(\mathbf{x}) = \mathbf{0} \tag{2.52b}$$

$$\mathbf{iec}(\mathbf{x}) \geq \mathbf{0} \tag{2.52c}$$

$$\mathbf{iec}(\mathbf{x})^T \boldsymbol{\mu} = \mathbf{0} \tag{2.52d}$$

$$\boldsymbol{\mu} \geq \mathbf{0} \tag{2.52e}$$

with another vector of Lagrange multipliers $\boldsymbol{\mu}$. Note that (2.52d) can be replaced by

$$\text{iec}_i(\mathbf{x}) \cdot \mu_i = 0, \ \forall i \in \{1, \ldots, p\} \tag{2.53}$$

because of (2.52c) and (2.52e).

### 2.2.2 Barrier methods

We will now introduce a class of methods to solve constrained optimization problems of the form (2.48) and as such can be used to optimize trajectories in discrete time. First, the inequality constraints (2.48c) are eliminated by adding a logarithmic term in the objective function

$$\mathbf{x}^* = \arg\min_x J(\mathbf{x}) - \mu \sum_i \log(\text{iec}_i(\mathbf{x})) \tag{2.54a}$$

$$\mathbf{ec}(\mathbf{x}) = \mathbf{0}. \tag{2.54b}$$

As $\text{iec}_i(\mathbf{x}) \to 0$, the logarithmic barrier term $-\mu \sum_i \log(\text{iec}_i(\mathbf{x})) \to \infty$ for a barrier parameter $\mu > 0$, therefore the minimum of the new problem (2.54) must lie within the region satisfying the inequality constraints. As $\mu \to 0$ the barrier becomes steeper and the solution of the problem (2.54) approaches the solution of the original problem (2.48). The advantage of replacing inequality constraints with barrier terms in the objective function is that a distinction between an active and inactive set of constraints is not required.

Solving (2.54) using Lagrange multipliers for a small $\mu$ is, however, an ill conditioned problem [111]. The gradient of the new objective function

$$\nabla_{\mathbf{x}} \left( J(\mathbf{x}) - \mu \sum_i \log(\text{iec}_i(\mathbf{x})) \right) = \nabla_{\mathbf{x}} J(\mathbf{x}) - \mu \sum_i \frac{\nabla_{\mathbf{x}} \text{iec}_i(\mathbf{x})}{\text{iec}_i(\mathbf{x})} \tag{2.55}$$

contains $\frac{1}{\text{iec}_i(\mathbf{x})}$ that approaches infinity if the constraint becomes active. Therefore a primal dual method is generally used, that introduces additional dual variables $\nu_i = \frac{\mu}{\text{iec}_i(x)}$ expressing (2.55) as

$$\nabla_{\mathbf{x}} J(\mathbf{x}) - \frac{\partial \mathbf{iec}(\mathbf{x})^T}{\partial \mathbf{x}} \boldsymbol{\nu} = \mathbf{0} \tag{2.56a}$$

$$\text{iec}_i(\mathbf{x}) \cdot \nu_i - \mu = 0, \ \forall i \in \{1, \ldots, p\}. \tag{2.56b}$$

Adding the equality constraints (2.54b) using Lagrangian multipliers $\boldsymbol{\lambda}$ gives the necessary conditions for optimality of the new problem

$$\nabla_{\mathbf{x}} J(\mathbf{x}) + \frac{\partial \mathbf{ec}(\mathbf{x})^T}{\partial \mathbf{x}} \boldsymbol{\lambda} - \frac{\partial \mathbf{iec}(\mathbf{x})^T}{\partial \mathbf{x}} \boldsymbol{\nu} = \mathbf{0} \tag{2.57a}$$

$$\mathbf{ec}(\mathbf{x}) = \mathbf{0} \tag{2.57b}$$

$$\mathrm{iec}_i(\mathbf{x}) \cdot \nu_i - \mu = 0, \ \forall i \in \{1, \ldots, p\}. \tag{2.57c}$$

Equations (2.57) consist of $n + m + p$ equations for the same amount of variables and can be solved using common numerical methods like gradient descent, Newtons method etc., that iteratively approach the solution. The initial vector of decision variables $\mathbf{x}$ must be chosen to fulfill the inequality constraints $\mathrm{iec}_i(\mathbf{x})$ and the initial $\mu$ and $\nu_i$ must be strictly positive. The barrier parameter $\mu$ is reduced iteratively during the process [105].

Barrier methods are also called interior point methods, since the solution is approached from within the area of satisfied inequality constraints. Together with the conditions $\mathrm{iec}_i(\mathbf{x}) \geq 0$ and $\nu_i \geq 0$ that are always fulfilled during the optimization, equations (2.57) approach the KKT condition (2.52) for $\mu \to 0$.

We use an existing implementation of a barrier method called Ipopt for interior point optimizer [106] interfaced through JuMP [27] in the Julia language.

## 2.3 Optimal control using function approximators

In this section, the different methods that can be used to optimize a feedback controller in the form of a function approximator are presented. We consider the control of dynamical systems in the form of nonlinear, time discrete state-space models as described in equation (2.46). As we deal with the control of physical systems, we only consider the case of continuous control signals $\mathbf{u}$ with input saturation, i.e., $\mathbf{u} \in \{\mathbf{u} \in \mathbb{R}^m | u_{i,\min} \leq u_i \leq u_{i,\max}, \ \forall i \in \{1, \ldots, m\}\}$.

The addressed control design problem is to find parameters $\boldsymbol{\vartheta}_{\mathrm{c}}$ of a feed-

back control law

$$\mathbf{u}_t = \mathbf{g}(\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c}) \tag{2.58}$$

such that the control acts optimally with respect to some loss function $J(\boldsymbol{\Theta})$. The optimization setting can contain different elements depending on the method, e.g., a finite or infinite time-horizon, inequality constraints etc. For some algorithms, it is useful to use a non-deterministic control law. In that case, the control parameters $\boldsymbol{\vartheta}_\mathrm{c}$ describe a probability distribution conditioned on the state $\mathbf{x}$ and the control inputs are sampled:

$$\mathbf{u}_t \sim \mathbf{g}(\mathbf{u}|\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c}). \tag{2.59}$$

The available methods that can be used to find the optimal, or at least close to optimal, $\boldsymbol{\vartheta}_\mathrm{c}$ can be classified into the three categories 1) black-box optimization, 2) policy gradient algorithms and 3) imitation learning. Every method has different strengths and weaknesses, as such the choice of the right method can lead to faster convergence or better results. The most commonly used methods of the different categories are presented in the following.

### 2.3.1 Black-box optimization methods

Black-box optimization is a term used for optimization algorithms that do not use gradient information and rely solely on function evaluations. The algorithms are generally used to solve unconstrained static optimization problems. The optimization problem for our controller can be written down as a static optimization problem of the decision variables $\boldsymbol{\vartheta}_\mathrm{c}$. With a finite time horizon $T$ and the simulated states $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T]$ and inputs $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{T-1}]$, we write the optimization as

$$\boldsymbol{\vartheta}_\mathrm{c}^* = \arg\min_{\boldsymbol{\vartheta}_\mathrm{c}} J(\boldsymbol{\vartheta}_\mathrm{c}) = \mathbb{E}\left[c(\mathbf{X}, \mathbf{U})\right] \tag{2.60a}$$

$$s.t.\ \mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t) \tag{2.60b}$$

$$\mathbf{u}_t \sim \mathbf{g}(\mathbf{u}|\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c}) \tag{2.60c}$$

$$\mathbf{x}_0 \sim P_{x0}(x). \tag{2.60d}$$

The costs of a trajectory $c(\mathbf{X}, \mathbf{U})$ is in most cases designed as a sum of costs per time-step as $c(\mathbf{X}, \mathbf{U}) = \sum_{t=0}^{T} c(\mathbf{x}_t, \mathbf{u}_t, t)$, but is not limited to that form for black-box optimization algorithms.

The controller and initial state $\mathbf{x}_0$ can be deterministic or defined as probability distribution as indicated in (2.60). One evaluation of $J(\boldsymbol{\vartheta}_\mathrm{c})$ requires at least one simulation the system for $T$ steps and the evaluation of the cost function $c(\mathbf{X}, \mathbf{U})$ for each trajectory. Using black-box optimization methods is reasonable if the gradient $\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} J(\boldsymbol{\vartheta}_\mathrm{c})$ is either expensive to compute or difficult to derive. We will limit ourselves in the following to a method called finite differences that is known for its simplicity, and the current state of the art methods of the class of evolution strategies that have shown promising results in the context of optimizing control laws for complex systems [93]. Other black-box optimization methods include, e.g., the Nelder-Mead method [76], particle swarm optimization [86] or genetic algorithms [108].

### Finite differences

Finite differences is a way to estimate the gradient of a function by evaluating the function multiple times at specific points in each dimension. The analytic derivative of a scalar function is defined as

$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{2.61}$$

An example of a finite difference formula is to use a sufficiently small $h$ to approximate the derivative in (2.61) as

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x)}{h}. \tag{2.62}$$

Estimating the derivative according to (2.62) requires two function evaluations. Different formulas exist for the approximation of first and higher order derivatives. E.g., the central first order difference is

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2 \cdot h}. \tag{2.63}$$

and the central difference of second order is

$$\frac{\partial^2 f(x)}{\partial x^2} \approx \frac{f(x+h) - 2 \cdot f(x) + f(x-h)}{h^2}. \tag{2.64}$$

The formulas can be derived using a Taylor series around the base points. Details on the derivations can be found, e.g., in [61].

Approximating the gradient of a vector value function $\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} J(\boldsymbol{\vartheta}_\mathrm{c})$ requires evaluating a finite difference formula for each $\vartheta_{\mathrm{c},i}$. The approximation can then be used with gradient-based optimization algorithms, e.g., gradient descent or Newton's method. As such, finite differences can turn gradient-based methods into black-box optimization methods, however the number of function evaluations is growing with the number of parameters, therefore this approach is inefficient for large problems.

### Evolution strategies

Evolution strategies (ES) are stochastic methods that optimize a function based on mutation and selection of solution candidates. The mutation of individuals is based on probability densities and the selection of individuals is based on a loss function. Solution candidates $\boldsymbol{\vartheta}_{\mathrm{c},i}$ for the problem (2.60) are sampled from a known probability distribution $P_\Psi(\boldsymbol{\vartheta}_\mathrm{c}; \boldsymbol{\Psi})$. The distribution of solution candidates is updated iteratively in order to minimize the expected value of the loss. The intermediate goal is to find a good distribution by optimizing

$$\boldsymbol{\Psi}^* = \arg\min_{\boldsymbol{\Psi}} \mathbb{E}_{\boldsymbol{\vartheta}_\mathrm{c} \sim P_\Psi} \left[ J(\boldsymbol{\vartheta}_\mathrm{c}) \right]. \tag{2.65}$$

Thus, while we are interested in finding good parameters $\boldsymbol{\vartheta}_\mathrm{c}$, evolution strategies optimize parameters $\boldsymbol{\Psi}$ of a distribution $P_\Psi(\boldsymbol{\vartheta}_\mathrm{c}; \boldsymbol{\Psi})$ such that the distribution provides good solution candidates. The gradient of the objective can be derived as

$$\begin{aligned}
\nabla_{\boldsymbol{\Psi}} \mathbb{E}_{\boldsymbol{\vartheta}_\mathrm{c} \sim P_\Psi} \left[ J(\boldsymbol{\vartheta}_\mathrm{c}) \right] &= \nabla_{\boldsymbol{\Psi}} \int J(\boldsymbol{\vartheta}_\mathrm{c}) \cdot P_\Psi(\boldsymbol{\vartheta}_\mathrm{c}; \boldsymbol{\Psi}) d\boldsymbol{\vartheta}_\mathrm{c} && \text{(2.66a)} \\
&= \int J(\boldsymbol{\vartheta}_\mathrm{c}) \cdot \nabla_{\boldsymbol{\Psi}} P_\Psi(\boldsymbol{\vartheta}_\mathrm{c}; \boldsymbol{\Psi}) d\boldsymbol{\vartheta}_\mathrm{c} && \text{(2.66b)} \\
&= \int \left( J(\boldsymbol{\vartheta}_\mathrm{c}) \cdot \nabla_{\boldsymbol{\Psi}} \log P_\Psi(\boldsymbol{\vartheta}_\mathrm{c}; \boldsymbol{\Psi}) \right) P_\Psi(\boldsymbol{\vartheta}_\mathrm{c}; \boldsymbol{\Psi}) d\boldsymbol{\vartheta}_\mathrm{c} && \text{(2.66c)} \\
&= \mathbb{E}_{\boldsymbol{\vartheta}_\mathrm{c} \sim P_\Psi} \left[ J(\boldsymbol{\vartheta}_\mathrm{c}) \cdot \nabla_{\boldsymbol{\Psi}} \log P_\Psi(\boldsymbol{\vartheta}_\mathrm{c}; \boldsymbol{\Psi}) \right]. && \text{(2.66d)}
\end{aligned}$$

For (2.66a), we used the expression of the expected value (2.15) and for (2.66c), we used the equality

$$\nabla_x f(x) = f(x) \cdot \nabla_x \log f(x) \tag{2.67}$$

that holds for all positive definite functions. The advantage of the expression(2.66d) is that the gradient operator is inside the mean, compared to the initial expression that took the gradient of a mean.

The expected value in (2.66d) can be approximated using the sample mean over $N_{pop}$ samples

$$\mathbf{g} = \frac{1}{N_{pop}} \sum_{i=1}^{N_{pop}} J(\boldsymbol{\vartheta}_{\mathrm{c},i}) \cdot \nabla_{\boldsymbol{\Psi}} \log P_\Psi(\boldsymbol{\vartheta}_{\mathrm{c},i}; \boldsymbol{\Psi}) \tag{2.68}$$

that requires sampling multiple solution candidates $\boldsymbol{\vartheta}_{\mathrm{c},i}$ and evaluating their costs $J(\boldsymbol{\vartheta}_{\mathrm{c},i})$ through simulation. The number of sampled parameter candidates $N_{pop}$ is called the population size. In the following, we present two algorithms, starting with an algorithm that has been used for control problems in [93].

Salimans et al. [93] choose $P_\Psi$ to be a normal distribution with constant covariance matrix $\sigma^2\mathbf{I}$ and mean $\bar{\boldsymbol{\vartheta}}_{\mathrm{c}}$. Equation (2.66d) then simplifies to

$$\nabla_{\hat{\boldsymbol{\vartheta}}_{\mathrm{c}}} \mathbb{E}_{\boldsymbol{\vartheta}_{\mathrm{c}} \sim P_\Psi} \left[ J(\boldsymbol{\vartheta}_{\mathrm{c}}) \right] = \frac{1}{\sigma} \mathbb{E}_{\varepsilon \sim \mathcal{N}(0,I)} \left[ J(\bar{\boldsymbol{\vartheta}}_{\mathrm{c}} + \sigma\boldsymbol{\varepsilon}) \cdot \boldsymbol{\varepsilon} \right]. \tag{2.69}$$

Moreover, Salimans et al. [93] use mirrored sampling [14], i.e., for each noise sample two parameter candidates $\bar{\boldsymbol{\vartheta}}_{\mathrm{c}} + \sigma\boldsymbol{\varepsilon}$ and $\bar{\boldsymbol{\vartheta}}_{\mathrm{c}} - \sigma\boldsymbol{\varepsilon}$ are created. They optimize the mean $\bar{\boldsymbol{\vartheta}}_{\mathrm{c}}$ using simple gradient descent with learning rate $\alpha$. Pseudo-code for the main approach of Salimans et al. [93] is shown in algorithm 3. They add small changes for a more efficient parallel implementation and use neural network specific improvements that are not shown in our pseudo-code. The algorithm can easily be parallelized and is memory efficient. However, a large number of function evaluations is required for complex problems using algorithm 3. The algorithms is later referred to simply as ES.

Other algorithms exist that internally estimate the covariance matrix of the distribution and adapt the learning rate. The covariance matrix adaptation evolution strategy (CMAES) [38, 39] and the distance-weighted exponential natural evolution strategy (DXNES) [30] are two examples of such

**Algorithm 3** Evolution strategy with mirror sampling.

**Require:** $\bar{\boldsymbol{\vartheta}}_{\mathrm{c}}$, $\sigma$, $\alpha$, $N_{pop}$

**while** (No stopping criterion active) **do**
  $\mathbf{g} = \mathbf{0}$
  **for** $i = 1$ to $N_{pop}/2$ **do**
    $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\mathbf{g} \leftarrow \mathbf{g} + \left(J(\bar{\boldsymbol{\vartheta}}_{\mathrm{c}} + \sigma\boldsymbol{\varepsilon}) - J(\bar{\boldsymbol{\vartheta}}_{\mathrm{c}} - \sigma\boldsymbol{\varepsilon})\right) \cdot \boldsymbol{\varepsilon}$
  **end for**
  $\bar{\boldsymbol{\vartheta}}_{\mathrm{c}} \leftarrow \bar{\boldsymbol{\vartheta}}_{\mathrm{c}} - \frac{\alpha}{\sigma N_{pop}}\mathbf{g}$
**end while**
**return** $\bar{\boldsymbol{\vartheta}}_{\mathrm{c}}$

algorithms. However, these algorithms are limited by the number of control parameters due to the memory required to store the covariance matrix. Hence, they are not suited for deep neural networks. CMAES is used later, therefore an explanation of the main ideas is given in the following.

The solution candidates of CMAES are sampled from a distribution $\boldsymbol{\vartheta}_{\mathrm{c},i} = \bar{\boldsymbol{\vartheta}}_{\mathrm{c}} + \sigma\boldsymbol{\varepsilon};\ i \in \{1, \ldots, N_{pop}\}$ with $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{C})$. Different from algorithm 3, CMAES uses a dense covariance matrix $\mathbf{C}$ and a scaling factor $\sigma$ that are both updated iteratively during the optimization. The covariance matrix is adapted to primarily sample parameter candidates in promising parts of the parameter space. To sample more efficiently from $\mathcal{N}(\bar{\boldsymbol{\vartheta}}_{\mathrm{c}}, \sigma^2\mathbf{C})$, the positive definite matrix $\mathbf{C}$ is decomposed using eigenvalue decomposition into $\mathbf{C} = \mathbf{B}\mathbf{E}^2\mathbf{B}^T$, with $\mathbf{E}$ the diagonal matrix that contains the square root of the eigenvalues as its diagonal. Sampling $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{C})$ can be achieved by sampling $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ followed by the transformation $\boldsymbol{\varepsilon} = \mathbf{B}\mathbf{E}\mathbf{z}$.

The parameter candidates $\boldsymbol{\vartheta}_{\mathrm{c},i}$ are evaluated by computing $J(\boldsymbol{\vartheta}_{\mathrm{c},i})$ and sorted in increasing order according to their loss. After sorting, we have $J(\boldsymbol{\vartheta}_{\mathrm{c},1}) \leq J(\boldsymbol{\vartheta}_{\mathrm{c},2}) \leq \cdots \leq J(\boldsymbol{\vartheta}_{\mathrm{c},N_{pop}})$. A subset of the $\mu < N_{pop}$ best candidates $\boldsymbol{\vartheta}_{\mathrm{c},i}$, i.e., the candidates with the smallest loss, is used in a weighted sum to generate the mean of the distribution for the next epoch

$$\bar{\boldsymbol{\vartheta}}_{\mathrm{c,new}} = \sum_{i=1}^{\mu} w_i \boldsymbol{\vartheta}_{\mathrm{c},i}. \tag{2.70}$$

The weights $w_i$ fulfill the conditions $\sum_i w_i = 1$ and $w_i > 0, \forall i$.

The algorithm then adapts the covariance matrix $\mathbf{C}$ and the scaling factor $\sigma$ individually. The covariance matrix of the current best parameters candidates is estimated based on the samples and used together in an exponentially weighted mean with past covariance matrices in an update term called the rank-$\mu$ update. Another term, called rank-one update, that considers the path of the mean $\bar{\boldsymbol{\vartheta}}_\mathrm{c}$ over the last epochs is added to the rank-$\mu$ update for the adaptation of $\mathbf{C}$. The update of the scaling factor $\sigma$ also considers the path of the mean over the last epochs. If the updates are strongly correlated, $\sigma$ is increased, else decreased. The formulas are lengthy and contain several heuristics, we therefore refer to [37] for a summary of the update equations. The recommended values for all parameters that are used in our implementation as well are given in the appendix of [37]. The procedure of CMAES is summarized in algorithm 4.

**Algorithm 4** Covariance matrix adaptation evolution strategy (CMAES).

**Require:** $\bar{\boldsymbol{\vartheta}}_\mathrm{c}$, $\sigma$, $N_{pop}$

$\mathbf{C} \leftarrow \mathbf{I}$
**while** (No stopping criterion active) **do**
  **for** $i = 1$ to $N_{pop}$ **do** ▷ Sample parameter candidates
    $\mathbf{z}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\boldsymbol{\varepsilon}_i \leftarrow \mathbf{B}\mathbf{E}\mathbf{z}_i$
    $\boldsymbol{\vartheta}_{\mathrm{c},i} \leftarrow \bar{\boldsymbol{\vartheta}}_\mathrm{c} + \sigma\boldsymbol{\varepsilon}_i$
  **end for**
  Compute $J(\boldsymbol{\vartheta}_{\mathrm{c},i})$ and sort $\boldsymbol{\vartheta}_{\mathrm{c},i}$ in increasing order.
  $\bar{\boldsymbol{\vartheta}}_\mathrm{c} \leftarrow \sum_{i=1}^{\mu} w_i \boldsymbol{\vartheta}_{\mathrm{c},i}$ ▷ Update the mean of the distribution
  Adjust step-size $\sigma$
  Adjust covariance matrix $\mathbf{C}$ and compute $\mathbf{B}$ and $\mathbf{E}$
**end while**
**return** $\bar{\boldsymbol{\vartheta}}_\mathrm{c}$

A comparison of black-box optimization algorithms on simple control problems can be found in [104].

### 2.3.2 Policy gradient methods

Policy gradient methods constitute a branch of reinforcement learning methods and are used for problems with continuous state and action spaces. The methods allow estimating the gradient of the loss $\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} J(\boldsymbol{\vartheta}_\mathrm{c})$ by trying different actions. The policy gradient methods sample in the action space instead of in the parameter space as is the case for evolution strategies, and adjust the controller to increase the probability of good actions. They tend to be more efficient than black-box optimization methods when the number of parameters is large. However, the optimization problem in (2.60) is limited to cost functions of the form

$$J(\boldsymbol{\vartheta}_\mathrm{c}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t c(\mathbf{x}_t, \mathbf{u}_t)\right];\ \gamma \in ]0, 1] \tag{2.71}$$

for many of the algorithms of this class. An exception is the Reinforce algorithm that is introduced in the next section. After the reinforce algorithm, actor critic algorithms are introduced and a specific algorithm called proximal policy optimization is shown, which is used later in the thesis.

#### Reinforce algorithm

The Reinforce algorithm was introduced in [110]. The algorithm adjusts the parameters of the controller to increase the probability of trajectories with small accumulated costs. We use a random variable composed of states and actions $\boldsymbol{\tau} = [\mathbf{x}_0, \mathbf{u}_0, \ldots, \mathbf{x}_{T-1}, \mathbf{u}_{T-1}, \mathbf{x}_T]$ to describe possible trajectories of $T$ steps, with the probability of observing a specific $\boldsymbol{\tau}$ defined by the probability distributions of the controller $\mathbf{g}(\mathbf{u}|\mathbf{x}_t)$ and the system dynamics $\mathbf{f}(\mathbf{x}|\mathbf{x}_t, \mathbf{u}_t)$. For deterministic dynamics, the Dirac distribution in equation (2.13) can be used for the following derivation. The state $\mathbf{x}_t$ is assumed to be a Markov state. The probability distribution of $\boldsymbol{\tau}$ is then

$$\begin{aligned} P_\tau(\boldsymbol{\tau}; \boldsymbol{\vartheta}_\mathrm{c}) =& f(\mathbf{x}_T|\mathbf{x}_{T-1}, \mathbf{u}_{T-1}) \cdot \mathbf{g}(\mathbf{u}_{T-1}|\mathbf{x}_{T-1}; \boldsymbol{\vartheta}_\mathrm{c}) \cdot \ldots \cdot \\ & f(\mathbf{x}_1|\mathbf{x}_0, \mathbf{u}_0) \cdot \mathbf{g}(\mathbf{u}_0|\mathbf{x}_0; \boldsymbol{\vartheta}_\mathrm{c}) \cdot P_{\mathrm{x}0}(\mathbf{x}_0). \end{aligned} \tag{2.72}$$

The accumulated costs over a single trajectory $\boldsymbol{\tau}$ is abbreviated as $c(\boldsymbol{\tau})$. The gradient of the log probability of a trajectory with respect to the control

parameters can be simplified to

$$
\begin{aligned}
&\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \log P_\tau(\boldsymbol{\tau}; \boldsymbol{\vartheta}_\mathrm{c}) \\
=&\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \left(\log(f(\mathbf{x}_T|\mathbf{x}_{T-1}) + \log(\mathbf{g}(\mathbf{u}_{T-1}|\mathbf{x}_{T-1}; \boldsymbol{\vartheta}_\mathrm{c})) + \ldots\right) && \text{(2.73a)} \\
=&\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \left(\log(\mathbf{g}(\mathbf{u}_{T-1}|\mathbf{x}_{T-1}; \boldsymbol{\vartheta}_\mathrm{c})) + \log(\mathbf{g}(\mathbf{u}_{T-2}|\mathbf{x}_{T-2}; \boldsymbol{\vartheta}_\mathrm{c})) + \ldots\right) && \text{(2.73b)} \\
=&\sum_{t=0}^{T} \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \log(\mathbf{g}(\mathbf{u}_t|\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c})) && \text{(2.73c)}
\end{aligned}
$$

With this results, we derive an expression for the gradient $\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} J(\boldsymbol{\vartheta}_\mathrm{c}) = \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \mathbb{E}_{\tau\sim P_\tau}\left[c(\boldsymbol{\tau})\right]$ in a similar way as for the derivation of the gradient for the evolution strategies

$$
\begin{aligned}
\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \mathbb{E}_{\tau\sim P_\tau}\left[c(\boldsymbol{\tau})\right] &= \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \int_\tau c(\boldsymbol{\tau}) \cdot P_\tau(\boldsymbol{\tau}; \boldsymbol{\vartheta}_\mathrm{c}) d\boldsymbol{\tau} && \text{(2.74a)} \\
&= \int_\tau c(\boldsymbol{\tau}) \cdot \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} P_\tau(\boldsymbol{\tau}; \boldsymbol{\vartheta}_\mathrm{c}) d\boldsymbol{\tau} && \text{(2.74b)} \\
&= \int_\tau c(\boldsymbol{\tau}) \cdot \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} (\log P_\tau(\boldsymbol{\tau}; \boldsymbol{\vartheta}_\mathrm{c})) \cdot P_\tau(\boldsymbol{\tau}; \boldsymbol{\vartheta}_\mathrm{c}) d\boldsymbol{\tau} && \text{(2.74c)} \\
&= \int_\tau \left(\sum_{t=0}^{T} \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \log \mathbf{g}(\mathbf{u}_t|\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c})\right) \cdot c(\boldsymbol{\tau}) \cdot P_\tau(\boldsymbol{\tau}; \boldsymbol{\vartheta}_\mathrm{c}) d\boldsymbol{\tau} && \text{(2.74d)} \\
&= \mathbb{E}_{\tau\sim P_\tau}\left[\left(\sum_{t=0}^{T} \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \log \mathbf{g}(\mathbf{u}_t|\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c})\right) \cdot c(\boldsymbol{\tau})\right] && \text{(2.74e)}
\end{aligned}
$$

The expression (2.74e) can be approximated using the sample mean over multiple simulated trajectories. The parameters are then updated using gradient descent. Pseudo-code for the resulting algorithm, called Reinforce, is given in algorithm 5.

### Actor critic methods

Algorithm 5 often requires large amounts of simulations in order to produce a good estimation of the gradient. Actor critic methods are methods that can reduce the variance of the sampled gradient and thus provide a meaningful gradient using less simulations. Actor critic methods use at least two function approximators: the first function approximator is the controller $\mathbf{g}(\mathbf{u}_t|\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c})$ also called the actor, the second function approximator is the critic that is representing either a value-function, action-value function or advantage

**Algorithm 5** Reinforce

**Require:** $\boldsymbol{\vartheta}_\mathrm{c}$, $N$

**while** (no stopping criteria met) **do**
  **for** $i = 1$ to $N$ **do**
    $\boldsymbol{\tau}_i \leftarrow \boldsymbol{\tau} \sim P_\tau$ ▷ Sample Trajectories, e.g., in simulation
  **end for**
  $\mathbf{g} \leftarrow \frac{1}{N} \sum_{i=1}^{N} J(\boldsymbol{\tau}_i) \cdot \left( \sum_{t=0}^{T} \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \log \mathbf{g}(\mathbf{u}_{i,t} | \mathbf{x}_{i,t}; \boldsymbol{\vartheta}_\mathrm{c}) \right)$ ▷ Gradient
  $\boldsymbol{\vartheta}_\mathrm{c} \leftarrow \boldsymbol{\vartheta}_\mathrm{c} - \alpha \cdot \mathbf{g}$ ▷ Gradient descent
**end while**
**return** $\boldsymbol{\vartheta}_\mathrm{c}$

function. The value-function $V(\mathbf{x})$ is defined as the function that returns the expected accumulated future costs, given the starting state $\mathbf{x}$ and a controller $\mathbf{g}(\mathbf{u}_t|\mathbf{x}_t)$

$$V(\mathbf{x}) = \mathbb{E}_{\mathbf{u} \sim \mathbf{g}(.)} \left[ \sum_{t=0}^{\infty} \gamma^t c(\mathbf{x}_t, \mathbf{u}_t) | \mathbf{x}_0 = \mathbf{x} \right]. \tag{2.75}$$

The value function evaluates each state $\mathbf{x}$. The lower the value of a state is, the better it is to be in this state. A recursive formula for the value function can be derived by unfolding the sum in equation (2.75). The recursive formula, given by

$$V(\mathbf{x}) = \mathbb{E}_{\mathbf{u} \sim \mathbf{g}(.)} \left[ c(\mathbf{x}_0, \mathbf{u}_0) + \gamma V(\mathbf{x}_{t+1}) | \mathbf{x}_0 = \mathbf{x} \right], \tag{2.76}$$

is called the Bellman equation in the context of reinforcement learning[2].

The action-value function also represents the expected future costs given a starting state $\mathbf{x}$ and controller $\mathbf{g}(.)$, however, the first action is not sampled from the controller, but is given as input to the function

$$Q(\mathbf{x}, \mathbf{u}) = \mathbb{E}_{\mathbf{u} \sim \mathbf{g}(.)} \left[ \sum_{t=0}^{\infty} \gamma^t c(\mathbf{x}_t, \mathbf{u}_t) | \mathbf{x}_0 = \mathbf{x}, \mathbf{u}_0 = \mathbf{u} \right]. \tag{2.77}$$

The advantage of the action-value function compared to the value function is that it contains information about which action leads to small or high costs in each state.

[2]The name Bellman equation has a different meaning in the control community, where the name is always related to optimality. In the reinforcement learning community, the same equation is called Bellman optimality equation.

Recently, the advantage function $A(\mathbf{x}, \mathbf{u})$ has gained popularity and is used in most current algorithms. The advantage function represents the expected decrease or gain in costs when choosing an action $\mathbf{u}$ in a given state, compared to choosing an action according to the current policy $\mathbf{g}(.)$. It can be expressed as the difference between the action value function for that action $\mathbf{u}$ and the value function

$$A(\mathbf{x}, \mathbf{u}) = Q(\mathbf{x}, \mathbf{u}) - V(\mathbf{x}). \tag{2.78}$$

The advantage function can be trained using the generalized advantage estimation approach, presented in Schulman [97]. The gradient can be expressed using the advantage-function as

$$\nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \mathbb{E}_{\tau \sim P_\tau} \left[ c(\boldsymbol{\tau}) \right] = \mathbb{E}_{\tau \sim P_\tau} \left[ \sum_{t=0}^{T} \left( \nabla_{\boldsymbol{\vartheta}_\mathrm{c}} \log \mathbf{g}(\mathbf{u}_t | \mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c}) \right) \cdot A(\mathbf{x}_t, \mathbf{u}_t) \right]. \tag{2.79}$$

Algorithms using the gradient in (2.79) are called advantage actor critic algorithms [66]. A recursive formulation can be derived for the action-value function and the advantage function in the same way as for the value function.

Current popular actor critic algorithms are the deterministic policy gradient [100], truncated natural policy gradient [25], trust region policy optimization [98] and proximal policy optimization algorithms (PPO) [99].

PPO is used later in the thesis. It is an actor critic algorithm that uses the advantage function $A(\mathbf{x}, \mathbf{u})$ for the gradient calculation and allows multiple gradient descent steps using one batch of sampled trajectories. The parameter update is calculated by minimizing a clipped loss function $\tilde{J}(\boldsymbol{\vartheta}_\mathrm{c})$ instead of just calculating a gradient direction. The clipped loss function is

$$\tilde{J}(\boldsymbol{\vartheta}_\mathrm{c}) = \mathbb{E}_{\boldsymbol{\tau} \sim P_{\boldsymbol{\vartheta}_\mathrm{c,old}}} \left[ \max \left( r_t(\boldsymbol{\vartheta}_\mathrm{c}) A(\mathbf{x}_t, \mathbf{u}_t), r_{\mathrm{clip,t}}(\boldsymbol{\vartheta}_\mathrm{c}) A(\mathbf{x}_t, \mathbf{u}_t) \right) \right] \tag{2.80a}$$

$$r_t(\boldsymbol{\vartheta}_\mathrm{c}) = \frac{g(\mathbf{u}_t | \mathbf{x}_t, \boldsymbol{\vartheta}_\mathrm{c})}{g(\mathbf{u}_t | \mathbf{x}_t, \boldsymbol{\vartheta}_\mathrm{c,old})} \tag{2.80b}$$

$$r_{\mathrm{clip,t}}(\boldsymbol{\vartheta}_\mathrm{c}) = \mathrm{clip}(r_t(\boldsymbol{\vartheta}_\mathrm{c}), 1 - \varepsilon, 1 + \varepsilon). \tag{2.80c}$$

In practice, the mean in equation (2.80a) is evaluated as a sample mean using simulated trajectories. Note that in equation (2.80a) the variables $\mathbf{x}_t$

and $\mathbf{u}_t$ are sampled using the parameters $\boldsymbol{\vartheta}_{\mathrm{c,old}}$, therefore no re-sampling of trajectories is required for each gradient step during the minimization of $\tilde{J}(\boldsymbol{\vartheta}_{\mathrm{c}})$.

The value $r_t(\boldsymbol{\vartheta}_{\mathrm{c}}) > 0$ represents a change in probability of an action for new parameters $\boldsymbol{\vartheta}_{\mathrm{c}}$, compared to the same action for the parameters of the last epoch $\boldsymbol{\vartheta}_{\mathrm{c,old}}$. If the probability of the action is unchanged, e.g., $\boldsymbol{\vartheta}_{\mathrm{c}} = \boldsymbol{\vartheta}_{\mathrm{c,old}}$, we have $r_t(\boldsymbol{\vartheta}_{\mathrm{c}}) = 1$, if the probability of $\mathbf{u}_t$ is increased for $\boldsymbol{\vartheta}_{\mathrm{c}}$ compared to $\boldsymbol{\vartheta}_{\mathrm{c,old}}$ then $r_t(\boldsymbol{\vartheta}_{\mathrm{c}}) > 1$ and vice versa in case the probability is decreased. The loss $\hat{J}(\boldsymbol{\vartheta}_{\mathrm{c}}) = \mathbb{E}_{\boldsymbol{\tau}\sim\boldsymbol{\vartheta}_{\mathrm{c,old}}}\left[r_t(\boldsymbol{\vartheta}_{\mathrm{c}})A(\mathbf{x}_t, \mathbf{u}\right]$ is used by [98] in the trust-region policy optimization together with a constraint of the Kullback-Leibler divergence between the old and new policy. Instead of a constraint, PPO uses clipping in the loss function.

The policy update is constrained by clipping $r_t(\boldsymbol{\vartheta}_{\mathrm{c}})$ if the change of probability exceeds a threshold defined by $\varepsilon$. The clipped probability quotient $r_{\mathrm{clip,t}}(\boldsymbol{\vartheta}_{\mathrm{c}})$ restraints the change in probability that is allowed.

The final loss function of PPO only limits the policy improvement and allows surpassing the threshold if the change in probability leads to an increase in the costs. This is achieved by using the largest value of $r_t(\boldsymbol{\vartheta}_{\mathrm{c}})A(\mathbf{x}_t, \mathbf{u}_t)$ and $r_{\mathrm{clip,t}}(\boldsymbol{\vartheta}_{\mathrm{c}})A(\mathbf{x}_t, \mathbf{u}_t)$ in the mean of the final loss function $J(\boldsymbol{\vartheta}_{\mathrm{c}})$. The minimization in each epoch of PPO is solved, e.g., using gradient descent.

### 2.3.3 Imitation learning approach

Another approach to train a parameterized controller is to couple trajectory optimization with imitation learning. Imitation learning is a technique that can be used if demonstrations on how to solve the task are available. A controller or agent is trained to behave similarly to the behavior in the demonstrations. The technique is mostly used to train machines to imitate humans. A survey on methods with human demonstrations are given in [44]. The data, however, can also result from simulations or optimizations. We focus on the case of non-human data throughout the thesis.

### Behavioral cloning with optimal trajectories

The simplest form of imitation learning, called behavioral cloning [117], uses collected state-action pairs for supervised learning. If a model is available, the required training data can be generated as optimal trajectories $(\mathbf{X}^*, \mathbf{U}^*)$. The procedure is as follows

1. Optimize multiple trajectories to receive training tuples $(\mathbf{x}_i^*, \mathbf{u}_i^*)$.
2. Use supervised learning to train a parameterized controller $\mathbf{g}(.)$ on the data tuples, such that $\mathbf{u}_i^* \approx \mathbf{g}(\mathbf{x}_i^*; \boldsymbol{\Theta})$
3. Apply the controller on the system.

Early applications of this indirect approach to optimal control can be found, e.g., in Pomerleau [87] on an autonomous driving example, in Ortega and Camacho [79] for the obstacle avoidance of a mobile robot, in [3, 4] for the control of a chemical process and more recently in [21] for a semi-active suspension system.

A problem with this approach is that, for a given distribution of starting states $\mathbf{x}_0 \sim P_{\mathsf{x}0}(x)$, the distribution of states included in the training data $\mathbf{X}^*$ can be different from the distribution of states visited later during the execution of the controller. This results from approximation errors during supervised learning and model errors if the controller is used outside of the simulation environment. When the controller encounters states that were not covered by the state distribution of the training data, the generated actions $\mathbf{u}_t$ can lead to undesirable or dangerous behavior.

The severity of the problem depends on the amount of available data and is most pronounced if data is scarce, e.g., when data is collected on the real system. The problem of mismatching distributions has been solved using different approaches. Two approaches are presented in the following.

### Dataset aggregation

A solution to the mentioned problem that is important for this work is presented in Ross et al. [92]. To make sure the distributions in the training data and the application are identical, they generate training tuples for the states visited in the closed-loop using a teacher. The procedure repeats the steps

1. Generate trajectories of states $\mathbf{X}$ with actions $\mathbf{U}$ produced by the controller $\mathbf{g}(\mathbf{x}_t, \boldsymbol{\vartheta}_\mathrm{c})$.

2. Ask an expert to propose actions $\hat{\mathbf{U}}$ for the states $\mathbf{X}$, and add the new training tuples $(\mathbf{x}_t, \hat{\mathbf{u}}_t)$ to a data set.

3. Train the control parameters $\boldsymbol{\vartheta}_\mathrm{c}$ using supervised learning on all available data $(\mathbf{x}_t, \hat{\mathbf{u}}_t)$.

The approach is called DAGGER for dataset aggregation, as old data is not discarded during the process.

#### Disturbances for augmenting robot trajectories

Laskey et al. [59] propose to tackle the problem by disturbing the expert during its demonstrations. The noise distribution used for the disturbances is optimized to account for the difference between the data distribution and the distribution of states visited during the application. The optimization of the noise distribution is a minimization of the Kulback-Leibler divergence of the disturbed expert demonstrations from the distribution of states in the application of the controller. The approach is called DART (Disturbances for Augmenting Robot Trajectories).

### Other approaches to imitation learning with trajectory optimization

We will shortly mention further approaches that combine imitation learning and trajectory optimization. However, these approaches are only mentioned to give a better overview over the topic and are not used in this work.

Mordatch and Todorov [67] combine the trajectory optimization and the supervised learning steps into a single objective. The optimization problem is solved by iterating between a regularized trajectory optimization and a supervised learning step. In simulation, they show a decrease in the expected costs for their method compared to the simpler behavioral cloning approach. In [68], they use their approach on multiple simulation tasks of nonlinear models and add a small number of recurrent states and sampled noise to increase the robustness of the final controller.

A similar approach called guided policy search is presented by Levine and Koltun [62, 63]. The objective is to minimize the Kulback-Leibler divergence of the state distribution generated by the controller and a distribution of states with small costs. To solve this problem, two steps are repeated iteratively. The first step consists in optimizing trajectories using a regularized cost function and the second step is the policy training using weighted supervised learning. Guided policy search is used in [119] and [47] with small changes for the feedback control of a simulated quadrotor navigation task and in [64] for the vision based control of a robot arm.

Chapter 3

# Adaptive and adjustable nonlinear control design

This chapter deals with adding robustness to controllers, trained using methods presented in section 2.3.1. First, we elaborate the notions of adaptability and robustness and present existing methods for training robust parameterized controllers. Afterwards, we present our methodology that allows training an adaptive feedback controller using imitation learning with trajectory optimization. Finally, we show a trivial extension to our method that allows small adjustments of the behavior during the application without retraining the controller. The method is also published in our previous paper [20].

In the following, the dynamic model (2.46) is extended to depend on a set of model parameters $\mathbf{p}$, e.g., the mass of system components or friction coefficients etc., that are considered uncertain

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t; \mathbf{p}). \tag{3.1}$$

## 3.1 Preliminaries

### 3.1.1 Adaptive or robust control

The design of a feedback controller is in general based on a mathematical model. A model, however, never exactly represents the real system. The difference between a simulation model and the real system is often referred

to as the *reality gap*. A controller designed or trained based on a simulation environment usually degrades in performance when applied to the real system. To reduce the degradation of the control performance, the controller can be designed to be robust against model errors. This can be achieved via adaptive control theory or robust control theory.

An adaptive controller can adapt its behavior during the application to the system. This type of controller is used if model parameters are not constant or are initially uncertain. Their adaptive behavior makes adaptive controllers robust against model errors as long as the model structure is sufficiently correct [7, p. 426]. Adaptive control and robust control are, however, two separate areas. Robust control is dealing with designing static controllers that can maintain a certain performance for a multitude of different models. As such, adaptive control and robust control are not competing methods in the presence of model uncertainties, but rather complementary methods [54, 58]. In fact, ideas from adaptive and robust control have been combined for certain model classes [9].

As a rule of thumb, Aström and Wittenmark [7, p. 426] argue that robust control can respond more quickly to changes in the system, but is more sensitive to noise due to high gains. Adaptive control on the other hand responds slower to changes in the system but can handle larger parameter variations.

### 3.1.2 Existing approaches for parameterized controllers

Due to the high cost of real-world data, most parameterized controllers are trained in simulation. The gap between the simulation model and the real system can lead to poor control performance in the application. To reduce the loss of performance, methods have been developed to either reduce the reality gap or to train a controller that performs well despite the reality gap.

In order to reduce the reality gap, data from the real system is necessary. Abbeel et al. [1] use a policy gradient method on an iteratively corrected model to compute a gradient direction for the controller. The step size, however, is determined on the real system. This assures an improvement of the control performance on the real system, and requires fewer data than com-

puting both gradient direction and step size on the real system. Deisenroth and Rasmussen [18] iteratively update a model in the form of a Gaussian process that is then used to optimize the controller offline. Golemo et al. [33] use a recurrent neural network to estimate the model error. The policy gradient algorithm PPO [99] is then used on the updated model to learn a controller.

The following works do not try to reduce the reality gap, but rather to learn a controller that is robust against modeling errors. The methods can roughly be divided into methods that train a controller on a multitude of different models, and methods that use a disturbing agent, which is called the adversary.

Mordatch et al. [69] use varying model parameters to compute a robust feed forward control for a humanoid robot. They also add a feedback control to the robot, to correct deviations from the robust trajectories, however, the robustness is induced in the trajectories rather than in the feedback controller in this work. Rajeswaran et al. [89] present a method that uses a policy gradient algorithm to train robust controllers. They sample a multitude of trajectories, but only use a percentile of the worst trajectories to compute a gradient direction. This leads to a controller that focuses on improving on trajectories with high costs, rather than improving on the mean costs. This can be seen as a relaxation of the min-max problem formulation for robust control, used, e.g., in robust model predictive control [60]. Pinto et al. [85] approach the min-max problem by using an adversary that is disturbing the agent during execution. The model parameters are kept constant and the controller is a parameterized function trained using a policy gradient algorithm to minimize the simulation costs, whereas the adversary is a parameterized function trained to maximize the simulation costs. Pattanaik et al. [83] also use adversarial attacks and a policy gradient algorithm, however, they derive the adversarial actions using the gradient of the action value function of the controller. Since the action value function contains information about the expected costs for each action, its gradient $\nabla_{\mathbf{u}} Q(\mathbf{x}, \mathbf{u})$ points in the direction of actions that lead to increased costs. Muratore et al. [72, 73] vary the parameters of the model and use a policy gradient algorithm for the policy training. They introduce the simulation optimization bias as

a stopping criterion for the policy optimization to avoid overfitting to the simulation environment. Bousmalis et al. [12] also use randomized model parameters and a policy gradient algorithm to improve the robustness of their feedback controller on a vision based grasping problem. Chebotar et al. [15] again randomize model parameters, but iteratively adapt the distribution of the model parameters to the real system by including the real system in the loop. The policy is trained using a policy gradient algorithm.

All previously mentioned methods developed a static control law. The first work that we present that develops an adaptive controller is by Yu et al. [114]. In simulation, they train a control law that has both the state and the system parameters as input. As the system parameters are unknown during the real application, they add an online system identification model that uses the recent history of states and actions to estimate the system parameters. The estimation of model parameters coupled with a control law that acts based on the estimated parameters is called a self-tuning regulator in the adaptive control community [7]. Different from most self-tuning regulators, Yu et al. [114] use a data-based approach with neural networks for both the control and parameters estimation tasks. A similar approach is presented by Peng et al. [84]. They train an adaptive control law using a policy gradient algorithm. However, instead of splitting the controller in two parts, they use a single recurrent neural network. The recurrent neural network is able to adjust its behavior based on the past history of states and actions by updating its internal, hidden state.

Looking back at the presented literature, the methods to train robust or adaptive parameterized controllers so far are limited to policy gradient methods. However, the control performance resulting from policy gradient methods has been shown to largely depend on the choice of hyper-parameters and, in some cases, to produce inconsistent results, even when using the same hyper-parameters multiple times [41]. Therefore, algorithms for robust control design that are not based on reinforcement learning are valuable alternatives. In the next section, we present a new method to train an adaptive controller using imitation learning with trajectory optimization.

## 3.2 Adaptive control via imitation learning

This section presents the main method of the thesis. The method allows training an adaptive controller in the form of a recurrent neural network based on trajectory optimization. As was shown in section 2.3.3, a parameterized controller can be trained by generating data in the form of optimal trajectories followed by supervised learning on the state-action tuples. However, this approach is limited to constant model parameters and static controllers as is shortly explained in the following.

In our previous work [19], we showed empirically that a static controller $\mathbf{g}(\mathbf{x}; \boldsymbol{\vartheta}_\mathrm{c})$, trained using supervised learning on optimized trajectories, degrades significantly in performance if the trajectories are generated using randomized model parameters $\mathbf{p}$. In the cited work, multiple approaches to imitation learning with trajectory optimization are compared. The naive approach is as follows

1. Generate multiple optimal trajectories using randomized model parameters $\mathbf{p}$ and receive tuples $(\mathbf{x}_t, \mathbf{u}_t)$

2. Train a parameterized controller $\mathbf{g}(\mathbf{x}; \boldsymbol{\vartheta}_\mathrm{c})$ by minimizing the mean squared error of $\|\mathbf{u}_t - \mathbf{g}(\mathbf{x}_t; \boldsymbol{\vartheta}_\mathrm{c})\|_2$.

This approach, however, does not generally lead to a good control law. This is caused by the fact that the optimal action $\mathbf{u}_t$ depends in this case on both the state $\mathbf{x}_t$ and the model parameters $\mathbf{p}$. The control law $\mathbf{g}(\mathbf{x}; \boldsymbol{\vartheta}_\mathrm{c})$ is not depending on the model parameters and has to interpolate between the data, which leads to a bad closed-loop performance. A controller with additional information about the model parameters, i.e., a control law $\mathbf{g}(\mathbf{x}, \mathbf{p}; \boldsymbol{\vartheta}_\mathrm{c})$ per contra showed close to optimal control performance in our empirical study [19]. The control law $\mathbf{g}(\mathbf{x}, \mathbf{p}; \boldsymbol{\vartheta}_\mathrm{c})$ can, however, only be used in simulation as we assume the real $\mathbf{p}$ to be unknown.

For the method presented in this section, the controller with additional information about the model parameters is used as an intermediate result for the training of the adaptive controller. We call that controller the oracle controller, as it contains additional information that is passed on to the

recurrent controller. Moreover, the term oracle is used in computer linguistics in similar sequence problems [32].

As was mentioned earlier, the simple method of using trajectory optimization with supervised learning is limited to training static controllers $\mathbf{u}_t = \mathbf{g}(\mathbf{x}_t)$. In the following we consider a recurrent controller of the form $\mathbf{u}_t, \mathbf{h}_t = \mathbf{r}(\mathbf{x}_t, \mathbf{h}_{t-1})$ that was trained using supervised learning on optimized trajectories. In the ideal case, the controller would reproduce the optimal trajectory in the closed-loop for a given initial state $\mathbf{x}_0$ of the training data. However, due to model inaccuracies and/or approximation errors of the controller, the trajectory at some point inevitably deviates from the optimal trajectory. At that point, the controller needs to correct this error to get closer to the optimal behavior again. However, the controller was only trained on optimal state-action sequences, and as such, it will extrapolate from the data as soon as the system deviates from optimal trajectories. In other words, the data distribution during training is different from the data distribution during testing. The problem is well known in sequence generating language models [90]. In our case, this leads to unpredictable behavior and generally poor control performance. For unstable systems, the controller is usually not able to stabilize the system, even if all trajectories in the training data converged to an equilibrium state. Therefore, it is necessary to introduce suboptimal trajectories with recovering actions into the training data. Our approach uses ideas similar to DAGGER and DART, introduced in section 2.3.3.

In our approach, the state distribution of the training data is generated by the recurrent controller in closed-loop, and training data is generated for the visited states, as is the case for DAGGER. In contrast to DAGGER, we use the intermediate controller $\mathbf{g}(\mathbf{x}_t, \mathbf{p})$ to generate the training targets. Moreover, since the evaluation of $\mathbf{g}(\mathbf{x}_t, \mathbf{p})$ is computationally inexpensive, the data is not aggregated into a large dataset, but discarded after each training epoch. To further explore non-optimal state sequences, we add noise during the simulations, similar to DART. In contrast to DART, the noise is added to the trajectories generated by the recurrent controller and not to the trajectories generated by the teacher. This removes the necessity to adapt the noise distribution as is done in DART.

Our method, that allows training recurrent controllers on trajectories with varying model parameters consists of three steps:

1. Generate optimal trajectories for a multitude of starting states and a multitude of different model parameters

2. Train an oracle controller in the form of a static control law $\mathbf{g}(\mathbf{x}, \mathbf{p})$.

3. Train the recurrent controller $\mathbf{r}(\mathbf{x}, \mathbf{h})$ on state sequences generated by the recurrent controller, but action sequences generated by the oracle controller.

In the following, we give details for each of the three steps.

### 3.2.1 Generating optimal trajectories

In the first step, optimal trajectories are generated, e.g., using barrier methods as described in section 2.2. Since both the starting state $\mathbf{x}_0$ and model parameters $\mathbf{p}$ are varied for each optimization, we define probability distributions $P_{\mathbf{x}0}(\mathbf{x})$ and $P_{\mathbf{p}}(\mathbf{p})$ a priori, e.g., as uniform distributions. The range of the possible model parameters must be estimated by the engineer, especially parameters that are difficult to measure should be varied. The distribution of starting states $P_{\mathbf{x}0}(\mathbf{x})$ is also important for the performance of the final controller. The training data should cover important states that are likely to be visited during execution. Moreover, states that are avoided by optimal trajectories due to high costs should be given non-zero probability in $P_{\mathbf{x}0}(\mathbf{x})$ to allow the controller to recover from those states. As an example, for the wheeled inverted pendulum, it is important to include starting states $\mathbf{x}_0$ of an already falling pendulum to assure that the trained controller can stabilize the system in a case of, e.g., a disturbance.

Another problem that can lead to bad closed-loop performance is to generate ambiguous training data, i.e., generating state-action pairs $(\mathbf{x}_t, \mathbf{u}_t)$ where similar states $\mathbf{x}_t$ are associated with different actions $\mathbf{u}_t$. Consider the exam-

ple trajectory generated by solving the optimization problem

$$\mathbf{x}^*, \mathbf{u}^* = \arg\min_{\mathbf{x},\mathbf{u}} \sum_{t=0}^{3} x^2 \tag{3.2a}$$

$$s.t.\ x_{t+1} = x_t + u_t \tag{3.2b}$$

$$x_0 = 1;\ x_3 = 1. \tag{3.2c}$$

The constraint $x_3 = 1$ forces the trajectory to return to a region of high costs. The optimized trajectory corresponds of the states $\mathbf{x}^T = [1, 0, 0, 1]$ and actions $\mathbf{u}^T = [-1, 0, 1]$. Matching the states with the actions produces the state-action tuples $(1, -1), (0, 0), (0, 1)$. We see that the state $x = 0$ is matched with actions $u = 0$ as well as $u = 1$. A static controller trained using supervised learning would interpolate and produce the control action $u = g(0) = 0.5$.

In order to avoid ambiguous control actions, the optimal action must be dependent only on the state $x$ and not on the time $t$. This can be assured by eliminating all constraints in the optimization problem and using a time independent cost function of the form $\sum_t c(\mathbf{x}_t, \mathbf{u}_t)$. This is a strong restriction for the formulation of the optimization problem. Another way to avoid ambiguity without changing the optimization setting is to only use the very first state-action tuple $(\mathbf{x}_0, \mathbf{u}_0)$ during supervised learning. This comes with the disadvantage of being data inefficient, as only one data tuple is generated per trajectory instead of $T$ tuples.

We will show empirically with the example of the wheeled inverted pendulum, that in practice, small ambiguities can be included in the training data, and that the whole trajectory should be used as training data for the sake of efficiency. Still, it is in the interest of the control engineer to avoid creating data as in the example problem (3.2).

Another consideration is that the relationship between state and control signal will be approximated by a smooth function approximator, therefore smooth optimal trajectories lead to a better data fit of the oracle controller in the next step. If the optimal control signal in a trajectory is not smooth or even noisy, the function approximator might not be able to reproduce the signal with the required accuracy. A smooth control signal can be achieved by tuning the cost function (2.47a) or by adding inequality constraints (2.47d).

The outcome of this first step are the optimal trajectories with associated model parameters, i.e., tuples $(\mathbf{X}^*, \mathbf{U}^*, \mathbf{p})$.

### 3.2.2 Training an oracle controller

As was mentioned earlier, training a controller $\mathbf{g}(\mathbf{x}, \mathbf{p}; \boldsymbol{\theta}_\mathrm{c})$ that takes both the state $\mathbf{x}_t$ and model parameters $\mathbf{p}$ as input results in a controller that achieves good control performance over a variety of parameters, while being limited to an application in simulation. This limitation results from the fact, that the real system parameters are assumed unknown or at least inaccurate. This controller, called oracle controller, is nevertheless useful, as it can efficiently generate near optimal actions for any combination of state and model parameters, independent of the past trajectory. As such, it can be used as a teacher for a recurrent neural network.

Before training the oracle network, the $N$ optimal trajectories are split into input-output pairs $([\mathbf{x}_{t,k}, \mathbf{p}_k], \mathbf{u}_{t,k})$ with $t \in \{0, \dots, T\}$ and $k \in \{1, \dots, N\}$. Subsequently, the control parameters are obtained by minimizing the mean squared error

$$\boldsymbol{\theta}_\mathrm{c}^* = \arg\min_{\boldsymbol{\theta}_\mathrm{c}} \frac{1}{N} \sum_{k=1}^{N} \sum_{t=0}^{T-1} \left\| \mathbf{u}_{t,k} - \mathbf{g}(\mathbf{x}_{t,k}, \mathbf{p}_k; \boldsymbol{\theta}_\mathrm{c}) \right\|^2 . \tag{3.3}$$

The control performance of the oracle controller should be verified in simulation before continuing to the next step.

### 3.2.3 Training a recurrent controller

After the oracle controller $\mathbf{g}(\mathbf{x}, \mathbf{p}; \boldsymbol{\theta}_\mathrm{c})$ was tested in simulation, a recurrent controller $\mathbf{r}(\mathbf{x}_t, \mathbf{h}_t, \boldsymbol{\vartheta}_\mathrm{c})$ that is able to adapt to the model parameters by using its internal state is created. As was shown in section 2.1.2 recurrent neural networks are an efficient alternative to functions that use a long history of past states as input.

The recurrent neural network should be trained on a data distribution that is close to the data distribution in the application. To ensure that the distributions are close, the recurrent neural network is trained on states that are generated in a closed-loop by the recurrent neural network itself.

This is called on policy learning, in contrast to off policy learning that uses states not generated by the policy itself, e.g., using the data from optimized trajectories. [59]

We create an on-policy algorithm for the training of a recurrent controller that uses ideas of DAGGER [92] and DART [59]. DAGGER is designed to be data efficient in that it accumulates all past demonstrations in order to reduce the use of the human expert. This is unnecessary in our work, as our expert is the oracle network that provides computationally efficient training targets. We therefore discard previous trajectories every episode of our algorithms. Since both our oracle and recurrent controller have deterministic behavior, the algorithm could converge to singular trajectories. We add small noise to the simulation environment to ensure an exploration in the state space even after convergence of the algorithm. This allows the controller to recover from disturbances. The noise level can be calibrated before the training of the recurrent controller in simulations using the oracle controller. If the trajectories are disturbed too much, the dependency of the states sequence on the model parameters might be lost. Therefore, a noise level that leads to subjectively small disturbed trajectories should be aimed for. The DART algorithm [59] also used noise to account for model errors, however, the authors use it in an off-policy setting and repeatedly adjusted the noise level to fit the distribution of visited states between the controller and the expert teacher. Since we use the noise in an on-policy setting, we do not have to adjust the noise level.

The final algorithm, customized for our use case, consists of the following steps

1. Creating $N_{traj}$ simulations using the current recurrent controller with different model parameters $\mathbf{p}$ and with small noise added to the dynamics.

2. Using the oracle controller $\mathbf{g}(\mathbf{x}_{t,k}, \mathbf{p}_k)$ to create target control signals $\hat{\mathbf{u}}_t$ for all states $\mathbf{x}_t$ visited during the simulation.

3. Truncate the state trajectories from 1. and target control signals from 2. into sequences suited for the training of a recurrent neural network.

4. Adjust the control parameters $\boldsymbol{\vartheta}_\mathrm{c}$ of the recurrent controller by supervised learning using TBPTT.

5. Check stopping criteria, e.g., maximum number of iterations reached. Eventually go to 1. or return parameters $\boldsymbol{\vartheta}_\mathrm{c}$.

In the following, we abbreviate this approach as DOI for disturbed oracle imitation. A visual comparison between DAGGER, DART and DOI is shown in figure 3.1. The algorithm is depicted as pseudo-code in algorithm 6 with the subroutine for the data generation in algorithm 7.

**Algorithm 6** Disturbed Oracle Imitation (DOI)

Inputs: $N_{epoch}, N_{traj}, N_{gd}, \boldsymbol{\vartheta}_\mathrm{c}$
**for** epoch $= 1$ to $N_{epoch}$ **do**
  $\mathcal{D} \leftarrow \emptyset$
  **for** traj $= 1$ to $N_{traj}$ **do**
    sample sequence $\mathbf{X}_\mathrm{traj}, \hat{\mathbf{U}}_\mathrm{traj}$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \left(\mathbf{X}_\mathrm{traj}, \hat{\mathbf{U}}_\mathrm{traj}\right)$
  **end for**
  **for** gd $= 1$ to $N_{gd}$ **do**
    Update $\boldsymbol{\vartheta}_\mathrm{c}$ using TBPTT.
  **end for**
**end for**

**Algorithm 7** Generating training data for the recurrent neural network

Inputs: $\mathbf{g}(.), P_{\mathsf{x}0}, P_p, \varepsilon$
$\mathbf{x}_0 \sim P_{\mathsf{x}0}(\mathbf{x}), \mathbf{p} \sim P_p(\mathbf{p}), \mathbf{h}_{(-1)} = \mathbf{0}$
**for** t $= 0$ to $T$ **do**
  $\hat{\mathbf{u}}_t = \mathbf{g}(\mathbf{x}_t, \mathbf{p})$ ▷ Evaluate oracle
  $\mathbf{u}_t, \mathbf{h}_t = \mathbf{r}(\mathbf{x}_t, \mathbf{h}_{t-1}; \boldsymbol{\vartheta}_\mathrm{c})$ ▷ Evaluate controller
  $\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t; \mathbf{p}) + \mathcal{N}(\mathbf{0}, \varepsilon^2\mathbf{I})$ ▷ Disturbed dynamics
**end for**
return $\mathbf{X} = [\mathbf{x}_0, \ldots, \mathbf{x}_{t-1}], \hat{\mathbf{U}} = [\hat{\mathbf{u}}_0, \ldots, \hat{\mathbf{u}}_{t-1}]$

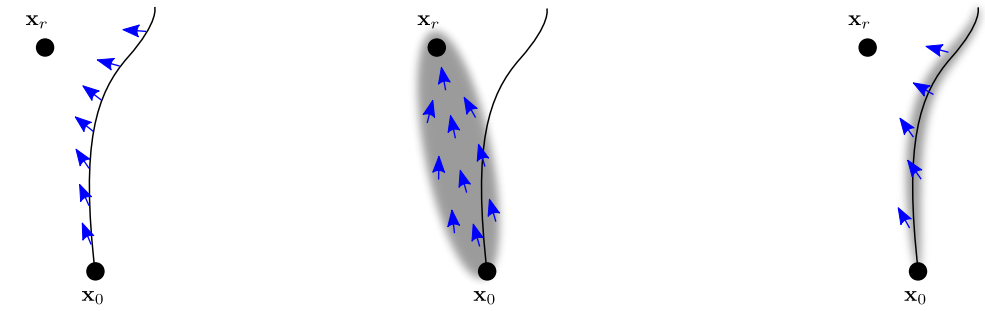



Figure 3.1: A visual comparison of DAGGER, DART and DOI. $\mathbf{x}_0$ indicates the initial state and $\mathbf{x}_r$ a reference state with low costs. The black line indicates a trajectory sampled using the parameterized controller in the loop. Blue arrows indicate the training data created for each approach, possibly generated for a distribution of trajectories, indicated by a blurred area.

As new training data is generated every epoch, it is not necessary to fully train the recurrent controller to convergence in the supervised learning step. It is more efficient to train for a small number of supervised learning epochs $N_{gd}$ only before generating new data.

We also use a fixed number of epochs $N_{epoch}$ for DOI. A stopping criterion could be used instead, e.g., checking for improvements of the accumulated costs in simulation.

## 3.3 Adjusting the control behavior

A drawback of most machine learning based control designs is the large amount of computing resources that are required for the training. While these controllers have a large amount of parameters $\boldsymbol{\vartheta}_c$, they cannot be tuned a posteriori, as the effect of changing parameters is in general unpredictable. As such, in case a small change in the behavior of the system is desired, the resource heavy training of a controller has to be repeated with a new cost function. For simple control structures, e.g., PID controllers, it is common practice to fine-tune the final behavior on the real system [5].

To allow fine-tuning in our case, a small change in the design method of parameterized controllers is introduced. In the same way that the controller is trained on different model parameters, we add a parameter $\lambda$ to the cost function $c(\mathbf{x}, \mathbf{u}, \lambda)$, that is varied during the trajectory generation. $\lambda$ should be bounded in a predefined range, e.g., $\lambda \in [-1, 1]$. Adding this new parameter can be interpreted as augmenting the state $\mathbf{x}_t$ with an additional state with $\lambda_{t+1} = \lambda_t$. Both oracle controller and recurrent controller are then trained with $\lambda$ as an additional input, e.g., the recurrent controller is then of the form

$$[\mathbf{u}_t^T, \mathbf{h}_t^T]^T = \mathbf{r}_\lambda(\mathbf{x}_t, \lambda, \mathbf{h}_{t-1}). \tag{3.4}$$

During execution, the parameter $\lambda$ is constant, but can be adjusted if small changes in the behavior are required.

Chapter 4

# Analysis using the simulation example of a bridge crane

In this chapter, an adaptive controller is trained for a bridge crane in simulation and compared in terms of control performance and robustness with other controllers. The focus is on the comparison with competing approaches, i.e., approaches for a robust and adaptive control design using reinforcement learning and evolution strategies. The control performance is analyzed with respect to changes in the model parameters. The task is chosen to bring the system at halt at a desired position without strong oscillations of the crane. Before the presentation of the control design steps, the model equations are introduced and the task is described in more detail.

## 4.1 Model equations and task description

The bridge crane used in this thesis is a system moving in a two-dimensional space, depicted in figure 4.1. The system consists of a mass that is attached via a stiff, massless rod to a moving cart. The cart is restricted to move horizontally. The system can be described by four states $\mathbf{x}^T = [s, \dot{s}, \varphi, \dot{\varphi}]$ with $s$ the horizontal position of the cart, $\dot{s}$ the horizontal velocity of the crane, $\varphi$ the angle between the rod and the vertical axis and $\dot{\varphi}$ the angular velocity of the rod. The system input is the voltage $u_a$ applied to an electric motor, limited to $|u_a| \leq 40$. The motor is modeled as a DC motor based on

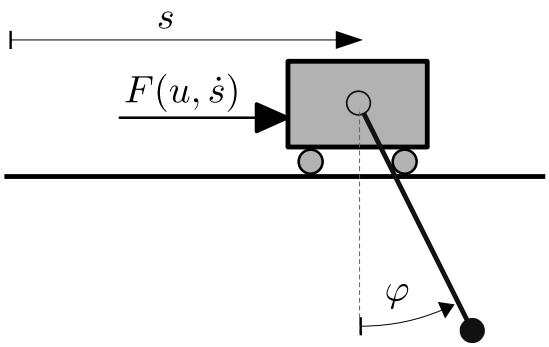



Figure 4.1: A schematic drawing of the bridge crane.

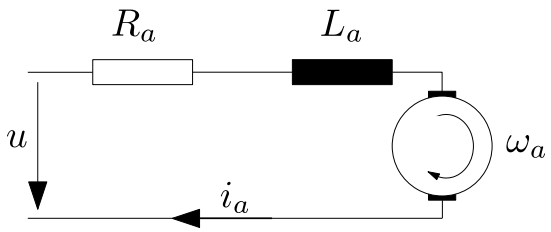



Figure 4.2: Schematic diagram of a DC motor circuit.

the circuit shown in figure 4.2 that is used throughout literature, e.g., [48, 107] or [78, p. 94–95]. The torque generated by the motor is proportional to the armature current $i_a$

$$\tau = k_a \cdot i_a \tag{4.1}$$

with the constant $k_a$ called the torque constant. The current dynamics can be expressed using Kirchhoff's circuit laws as

$$u_a = R_a \cdot i_a + L_a \cdot \frac{\partial i_a}{\partial t} + k_v \omega_a \tag{4.2a}$$

$$\Leftrightarrow \frac{\partial i_a}{\partial t} = \frac{u_a}{L_a} - \frac{R_a}{L_a} i_a - \frac{k_v}{L_a} \omega_a. \tag{4.2b}$$

The constant $k_v$ is called the velocity constant. Its product with the angular velocity of the anchor expresses a voltage called the back electromotive force (EMF) that opposes the change in the magnetic flux induced by the rotation of the anchor.

We furthermore simplify our model by ignoring the current dynamics, i.e., setting $\frac{\partial i_a}{\partial t} = 0$, and using the steady state relations in the coupled system. This simplification is sustained by the fact that the current dynamics are much faster than the cart and crane dynamics that we are interested in. With our assumption, we express the current as

$$i_a = \frac{1}{R_a} \left( u_a - k_v \omega_a \right). \tag{4.3}$$

and thus the motor torque becomes

$$\tau = \frac{k_a}{R_a} \left( u_a - k_v \omega_a \right). \tag{4.4}$$

The torque is transformed into a lateral force via a gearbox and belt system. The relation between the rotational velocity and translational velocity as well as between the torque and the force on the belt are determined by the gear ratio $i$ and radius $r$ of the wheel that the belt is attached to. The relations are

$$F = \frac{i}{r} \tau \tag{4.5a}$$

$$\dot{s} = \frac{r}{i} \omega_a. \tag{4.5b}$$

As such, we can express the force exerted from the motor on the cart as a function of the motor voltage and the cart velocity. The motor model is

$$F(u_a, \dot{s}) = \frac{i k_a}{r^2 R_a} \left( r u_a - i k_v \dot{s} \right). \tag{4.6}$$

We adopt the equations derived in [50] for the rigid body dynamics of our bridge crane system and repeat them in the following without derivation. We model sliding friction for the lateral movement of the cart using a friction coefficient $\mu$. Friction in the link between the cart and the cord is neglected.

$$\ddot{s} = \frac{m L \dot{\varphi}^2 \sin\varphi + m g \sin\varphi \cos\varphi - \mu \dot{s} + F(u_a, \dot{s})}{M + m \sin\varphi} \tag{4.7a}$$

$$\ddot{\varphi} = \frac{-m L \dot{\varphi}^2 \sin\varphi \cos\varphi - (M+m) g \sin\varphi + (\mu \dot{s} - F(u_a, \dot{s})) \cos\varphi}{L(M + m \sin^2\varphi)} \tag{4.7b}$$

Table 4.1: Model parameters of the bridge crane example.

| Variable | SI-Unit | Value | Description |
|---|---|---|---|
| $M$ | kg | 30 | Mass of the cart |
| $m$ | kg | 3 | Mass of the load |
| $L$ | $\mathrm{m} \pm 25\%$ | 1.2 | Length of the crane cable |
| $\mu$ | $\mathrm{N\,s\,m^{-1}}$ | 0.001 | Friction coefficient |
| $i$ | / | 3 | Gear box ratio |
| $r$ | m | $2.39 \cdot 10^{-2} \pm 25\%$ | Radius of the drive wheel |
| $R_a$ | $\Omega$ | 1.35 | Armature resistance |
| $L_a$ | H | $1.35 \cdot 10^{-3}$ | Armature inductance |
| $k_a$ | $\mathrm{N\,m\,A^{-1}}$ | 0.268 | Motor torque constant |
| $k_v$ | $\mathrm{V\,s\,rad^{-1}}$ | 0.263 | Motor velocity constant |

The model parameters with the value used in our simulation experiments are given in table 4.1. The model parameters do not belong to a physical bridge crane system. The motor parameters correspond to a physical hanging chain system at the chair of automatic control of the Technical University of Munich and were identified in [103]. While fictional, the value of the remaining parameters are also inspired by the previously mentioned work. For the use with neural networks, we normalize the system input as $u = \frac{u_a}{40}$ to have $-1 \leq u \leq 1$.

The system is equivalent to a pendulum system. A common task for this type of system is to balance the pendulum in its unstable position of rest at $\varphi = \pm\pi$ or to swing the pendulum up to the mentioned position of rest. The system is in that case often refereed to as the inverted pendulum. We do not solve the inverted pendulum task, as the success of the inverted pendulum problem for reinforcement learning is reliant on laborious tuning of hyper-parameters. In the student thesis of Krottenthaler [56], the inverted pendulum task was examined among other systems and the swing up task is found to be a challenging problem for policy gradient algorithms.

A simpler task, used in the following, is to drive the crane to a desired position, e.g., $s = 0$ and dampen remaining oscillations of the cable. We solve that task for a varying or uncertain length of the cable $L$ and radius

of the drive wheel $r$. Those parameters were chosen, as the system behavior is sensitive to their variation, compared to variations of the other model parameters.

## 4.2 Adaptive control design

The specific settings, considerations and hyper-parameters for the training of the adaptive controller using DOI are given in the following.

### 4.2.1 Trajectory optimization

The first step required for the DOI approach is to generate training data for the oracle controller by optimizing trajectories. The cost function is designed to punish large control signals, as well as deviations from the desired state $\mathbf{x} = \mathbf{0}$. The cost function is

$$c(\mathbf{x}, u) = s^2 + 2 - \cos(\varphi) - e^{-5^2\left(s^2+\varphi^2\right)^2} + 4\ln\left(1 + e^{s\cdot v}\right) + 4u^2 \tag{4.8}$$

with the terms $s^2$ and $1-\cos(\varphi)$ punishing states stronger, the further away they are from $s = 0, \varphi = 0$. The term $1 - e^{-5^2\left(s^2+\varphi^2\right)^2}$ is added to improve the behavior close to the origin. It adds a cost of 1 to all states, except states close to $s = 0, \varphi = 0$. The term $\ln\left(1 + e^{s\cdot v}\right)$ is an approximation of $\max(0, s\cdot v)$ with a continuous derivation. It punishes velocities that drive the crane away from the desired position. As such the term reduces overshooting over the desired position. The last term $u^2$ simply punishes large control signals.

The trajectories are optimized over 300 discrete time steps with a step size of $\delta_t = 0.05$s. A constraint is added to limit the applied motor voltage $-1 \leq u_t \leq 1$. We generate 5000 trajectories with different model parameters and initial states, sampled from uniform distributions. The initial states $\mathbf{x}_0 \sim P_{\mathsf{x}0}(\mathbf{x})$ are sampled independently from $s \sim \mathtt{U}(-1, 1), \dot{s} \sim \mathtt{U}(-0.5, 0.5), \varphi \sim \mathtt{U}(-\pi, \pi)$ and $\dot{\varphi} \sim \mathtt{U}(-0.3, 0.3)$. The model parameters are samples in the parameter range $L \sim P_{\mathsf{L}}(L) = \mathtt{U}(1.2 \cdot 0.75, 1.2 \cdot 1.25)$ and $r \sim P_{\mathsf{r}}(r) = \mathtt{U}(0.0239 \cdot 0.75, 0.0239 \cdot 1.25)$. The full optimization problem, solved 5000

times, is

$$\mathbf{X}^*, \mathbf{U}^* = \arg\min_{\mathbf{X},\mathbf{U}} \sum_{t=0}^{T} c(\mathbf{x}_t, u_t) \tag{4.9a}$$

$$s.t.\ \mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, u_t; \mathbf{p}) \tag{4.9b}$$

$$\mathbf{x}_{t=0} = \mathbf{x}_0 \tag{4.9c}$$

$$-1 \leq u_t \leq 1. \tag{4.9d}$$

The task can also be solved with less trajectories as will be shown later. Another 2000 trajectories are optimized as a validation set and as a reference for the closed-loop performance of the controllers. The starting states $\mathbf{x}_{0,i};\ i \in \{1, \ldots, 2000\}$ of the validation trajectories are used to approximate the mean accumulated costs of the controllers throughout this chapter as

$$\mathbb{E}\left[\sum_t c(\mathbf{x}, u)\right] \approx \frac{1}{|\mathcal{X}_0|} \sum_{\mathbf{x}_0 \in \mathcal{X}_0} \sum_t c(\mathbf{x}_t, u_t) \tag{4.10}$$

with $\mathcal{X}_0 = \{\mathbf{x}_{0,1}, \ldots, \mathbf{x}_{0,2000}\}$ the set of starting states used in the validation trajectories.

An optimized trajectory from the validation trajectories is shown in figure 4.3. The example trajectory starts with $\varphi \approx \pi$ and $s$ close to one, nevertheless, the state is close to the target state $\mathbf{x} = 0$ before the trajectory ends at $T = 300$ steps.

### 4.2.2 Oracle training

The oracle is a static neural network $g(\mathbf{x}, \mathbf{p}; \boldsymbol{\vartheta}_\mathrm{c})$ with 2 hidden layers and 32 neurons each. The hidden layers have tanh-activation functions and the output layer is linear. The oracle network is trained using supervised learning on input-output tuples from the optimized trajectories $([\mathbf{x}^T, L, r]^T, u)$. Moreover, since many trajectories had converged to the target state $\mathbf{x} = \mathbf{0}$ long before $T = 300$ steps, we only took the first 150 training tuples from each trajectory, to aim for an increase in the variance in the state distribution, or, in other words, to avoid including too many data-tuples near the target state. The data is split into a training set of 80% of the data and a test set of the remaining data. The training set is again split in mini-batches of 8192 tuples during training.

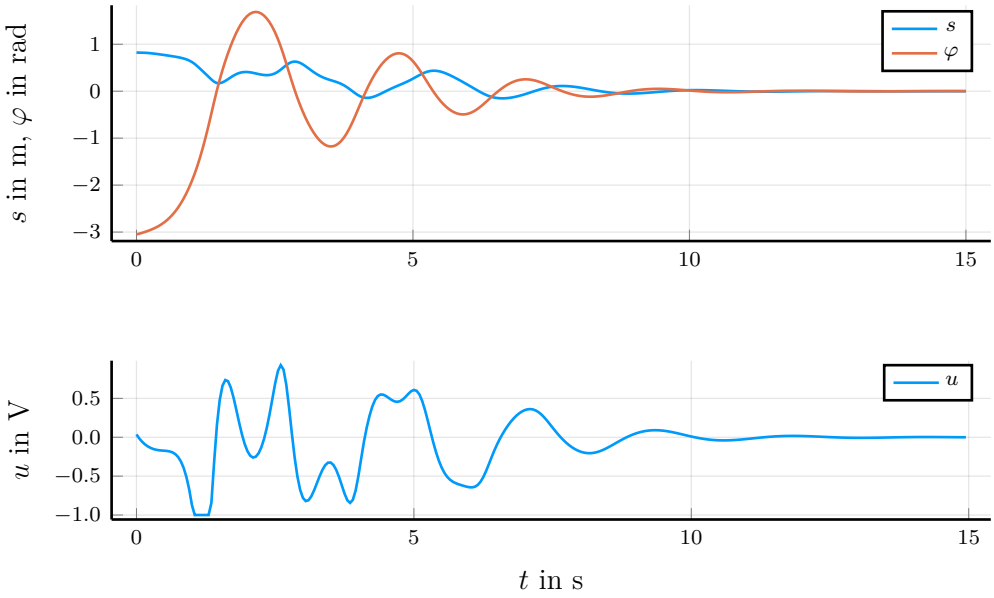



Figure 4.3: An example of an optimized trajectory for the bridge crane. Shown are the states $s$ and $\varphi$ in the top plot, and the normalized control signal $u$ in the bottom plot.

To analyze the influence of the number of trajectories on the closed-loop performance, the neural network is trained for different numbers of trajectories, and the closed-loop performance is evaluated every 250 training epochs during training. The mean costs per training episode and for different numbers of trajectories is shown in figure 4.4. The figure shows that the closed-loop performance increases with an increasing number of trajectories. This can be expected, as the state-space is better covered with more available data. Including more than 1000 trajectories, however, did not lead to noticeable increases in the control performance.

For a small number of trajectories, in this case 20 or 100, the closed-loop performance is decreasing after reaching a minimum at around 1000 training epochs. This is not to be confused with over-fitting on the training data during supervised learning, as the data was split in training and testing data as explained in section 2.1.2 and over-fitting on the data was not observed. An explanation could be a different kind of over-fitting to the states visited in the training trajectories, as the states in the training data are correlated due

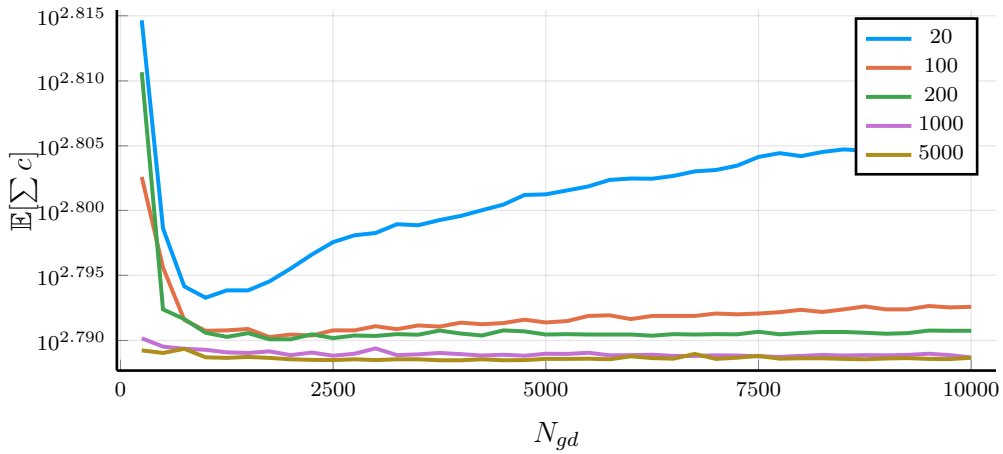



Figure 4.4: Closed loop performance of the oracle controller $g(\mathbf{x}, \mathbf{p}; \boldsymbol{\vartheta}_c)$ per training epoch for different numbers of optimal trajectories used as training data.

to the simulation. For the next step, i.e., for training the recurrent neural network, we use the oracle controller that was trained on 5000 trajectories.

### 4.2.3 Adaptive controller

The adaptive controller is parameterized as a recurrent neural network, consisting of one recurrent layer with 16 hidden states, followed by a tanh hidden layer and a linear layer with 8 neurons each. The recurrent neural network is trained using DOI, i.e., algorithm 6, and uses the hyper-parameters depicted in table 4.2.

To ascertain improvements of the controller, the average closed-loop performance as well as the match between the oracle and the recurrent controller are evaluated. The mean accumulated costs $\mathbb{E}\left[\sum_t c(\mathbf{x}, u)\right]$ are calculated every 5 training epochs of the DOI algorithm. The mean is approximated as the sample mean over simulated trajectories using the starting states in $\mathcal{X}_0$. Furthermore, the error on the training data, i.e., the error between the proposed control action by the oracle, and the chosen control action during simulation by the recurrent neural network, is evaluated every epoch of DOI

Table 4.2: Hyper-parameters used for DOI for the bridge crane example.

| Variable | Value | Description |
|---|---|---|
| $N_{\text{epoch}}$ | 500 | Number of epochs of DOI |
| $N_{\text{traj}}$ | 200 | Number of trajectories simulated per epoch |
| $N_{\text{gd}}$ | 50 | Number of gradient descent steps per epoch |
| $T$ | 300 | Number of discrete steps per simulation |
| $\tau$ | 50 | Length of truncated sequences for TBPTT |
| $\varepsilon$ | $5 \cdot 10^{-3}$ | Standard deviation of noise added to the simulation |
| $\delta_t$ | 0.05 | Discrete time step size in seconds |

before the supervised learning step as

$$E_{rnn} = \mathbb{E}\left[\|u_t - \hat{u}_t\|^2\right] \tag{4.11}$$

with $u_t$ the control signal computed by the recurrent controller and used during simulation and $\hat{u}_t$ the control signal proposed by the oracle controller and used as training data. Both measures are depicted per training epoch $N_{\text{epoch}}$ in figure 4.5.

The figure shows a clear correlation between both values. In the early stages of the training at around 30 and 50 training epochs of DOI, both the loss $E_{rnn}$ and the accumulated simulation costs peak shortly before converging. Possible reasons for the spike could be, e.g., a large gradient during supervised learning.

## 4.3 Control design using alternative methods

In this section, we train a controller based on the reinforcement learning algorithm proximal policy optimization (PPO) [99], a controller using evolution strategies [93] and a controller using CMAES [38] as a comparison for the adaptive controller trained in the previous section. The first controller is a static controller trained to be robust against uncertainties in the model parameters $L$ and $r$. The second and third controllers are adaptive controllers in the form of recurrent neural networks. We also trained a recurrent controller using reinforcement learning as proposed in Heess et al.

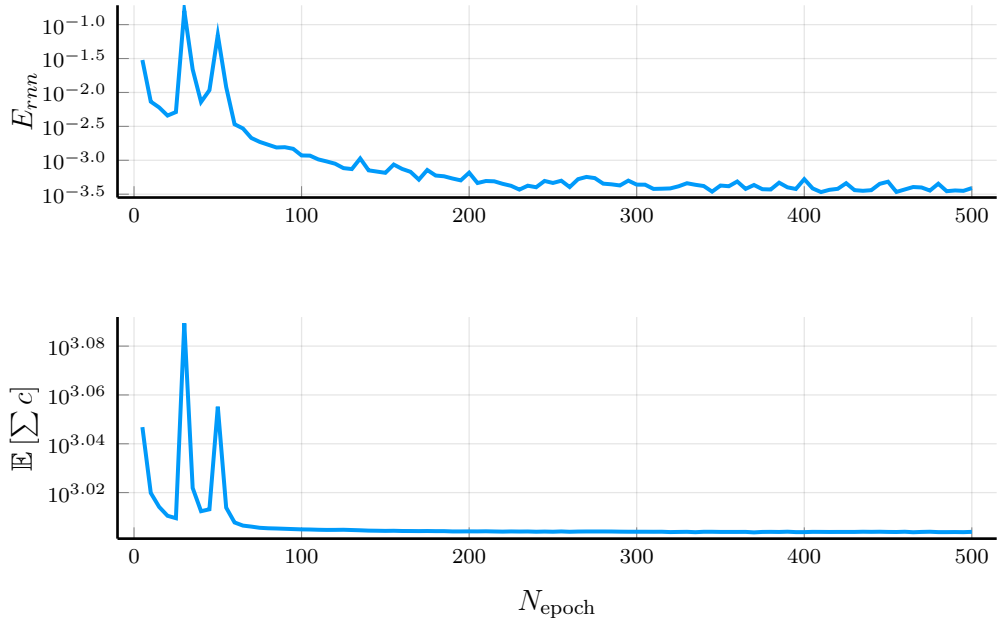



Figure 4.5: The loss $E_{rnn}$ evaluated before supervised learning on the new data is shown in the top figure over the number of training epochs $N_{\text{epoch}}$, and the mean accumulated costs $\mathbb{E}\left[\sum_t c(\mathbf{x}_t, u_t)\right]$ in simulation are shown in the bottom subfigure.

[40], however, we didn't manage to receive a competing control performance using the algorithm, therefore we excluded it from the end results.

### 4.3.1 Robust control using reinforcement learning

A robust controller $g_{\text{Epopt}}(\mathbf{x})$ is trained using the ensemble policy optimization (Epopt) framework presented by Rajeswaran et al. [89]. The approach approximates the min-max problem of robust control by improving on a subset of the worst trajectories. A policy gradient algorithm is required to determine the gradient direction for EPOPT. The policy gradient algorithm is adapted to use only the worst $\xi$-percentile of sampled trajectories for the gradient calculation. The approach, as presented in [89], also contains an adaptation of the distribution of the uncertain parameters to the real parameters of the system, which is omitted here as the bridge crane example is a simulation study. A study on Epopt using a different policy gradient algorithm on a

**Algorithm 8** Epopt with PPO

**Require:** $P_{\mathbf{p}}(\mathbf{p})$, $P_{\mathrm{x0}}(\mathbf{x}_0)$, $g(\mathbf{u}|\mathbf{x};\boldsymbol{\vartheta}_\mathrm{c})$

**for** epoch $= 1 \dots N_{\mathrm{epoch}}$ **do**
  **for** traj $= 1 \dots N_{\mathrm{traj}}$ **do**
    $\mathbf{p} \sim P_{\mathbf{p}}(\mathbf{p})$
    $\mathbf{x}_0 \sim P_{\mathrm{x0}}(\mathbf{x}_0)$
    Sample trajectory $\boldsymbol{\tau}$ using $g(\mathbf{u}_t|\mathbf{x}_t;\boldsymbol{\vartheta}_\mathrm{c})$
  **end for**
  Train Value-Function $V(\mathbf{x})$ using sampled trajectories
  Choose the $\xi$-percentile of highest cost trajectories and discard other data
  Compute advantage value $A_t$ using generalized advantage estimation
  Update $\boldsymbol{\vartheta}_\mathrm{c}$ using PPO
**end for**
**return** $\boldsymbol{\vartheta}_\mathrm{c}$

modified version of the bridge crane can be found in the students thesis [49]. Pseudo-code for Epopt with PPO is given in algorithm 8.

We tune the hyper-parameters by hand to achieve convergence of the Epopt-PPO algorithm. The value function is approximated by a neural network of two tanh layers with 32 neurons each, followed by a linear layer. The value function is trained using the Adam optimizer using the recommended parameters in [52], whereas the policy is trained using simple stochastic gradient descent. The values for important hyper-parameters are given in table 4.3. To improve convergence of the algorithm, we choose a discount factor $\gamma < 1$, thus slightly changing the initial objective. The average sum of discounted costs $\mathbb{E}\left[\sum_t \gamma^t c(\mathbf{x}_t, u_t)\right]$ is shown per training epoch in figure 4.6. Training is stopped after 4000 episodes, when the loss starts to slightly increase.

### 4.3.2 Adaptive control using evolution strategies

A robust or adaptive control design using evolution strategies (ES) and neural networks has not yet been published to the best of our knowledge. However,

Table 4.3: Hyper-parameters used for the Epopt-PPO algorithm.

| Variable | Value | Description |
|---|---|---|
| $N_{epoch}$ | 1000 | Number of training epochs |
| $N_{traj}$ | 2000 | Number of sampled trajectories per epoch |
| $T$ | 300 | Number of discrete time-steps used in the trajectory samples |
| $\gamma$ | 0.999 | Discount factor |
| $\sigma$ | 0.05 | Standard deviation of the policy |
| $\varepsilon$ | 0.05 | Clipping factor for PPO |
| $\lambda$ | 0.5 | Factor used in the GAE to compromise between bias and variance |
| $\xi$ | 0.1 | Percentile of the highest cost trajectories used to determine an improvement direction |
| $\nu$ | 10 | Gradient clipping threshold |
| $\alpha$ | $1 \cdot 10^{-4}$ | Learning rate for the policy |

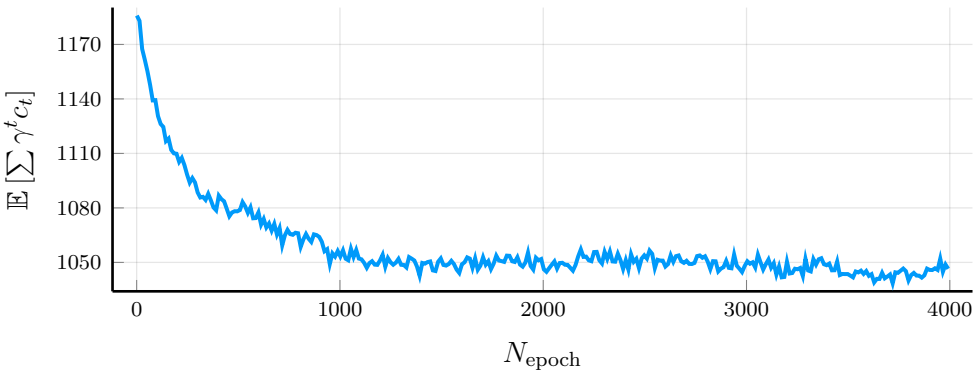



Figure 4.6: Accumulated discounted costs per training epoch of Epopt for the bridge crane example.

Table 4.4: Hyper-parameters used to train a recurrent neural network using evolution strategies.

| Variable | Value | Description |
|---|---|---|
| $N_{epoch}$ | 1000 | Number of training epochs |
| $N_{pop}$ | 200 | Population size. |
| $N_{traj}$ | 50 | Number of sampled trajectories for one evaluation of the loss $J(\boldsymbol{\vartheta}_{\mathrm{c}})$. |
| $T$ | 300 | Number of discrete time-steps used in the trajectory samples |
| $\sigma$ | 0.1 | Initial standard deviation of the parameter distribution. |
| $\alpha$ | 0.01 | Step size for the gradient descent. |

training a recurrent controller is not different from training a static controller. A scalar loss function $J(\boldsymbol{\vartheta}_{\mathrm{c}})$ that use the control parameters as input is defined and optimized without the need of providing a gradient. We train a recurrent controller $r_{\mathrm{ES}}(\mathbf{x}_t, \mathbf{h}_t; \boldsymbol{\vartheta}_{\mathrm{c}})$ using the simple evolution strategy in algorithm 3 and a recurrent controller $r_{\mathrm{CMAES}}(\mathbf{x}_t, \mathbf{h}_t; \boldsymbol{\vartheta}_{\mathrm{c}})$ using CMAES in algorithm 4. The neural networks use the same structure as the recurrent controller trained using DOI.

The loss $J(\boldsymbol{\vartheta}_{\mathrm{c}})$ has to include costs over multiple model parameters and multiple starting states. We define the loss function $J(\boldsymbol{\vartheta}_{\mathrm{c}})$ as the average performance over randomly sampled model parameters and starting states for the recurrent neural network. For each evaluation of $J(\boldsymbol{\vartheta}_{\mathrm{c}})$, i.e., once per epoch for each parameter candidate $\boldsymbol{\vartheta}_{\mathrm{c},i}$, $N_{traj}$ simulations are carried out. Thus the total number of simulations per epoch is thus $N_{traj} \cdot N_{pop}$.

First, we use the algorithm as presented in algorithm 3 with the hyper-parameters in table 4.4. The standard deviation $\sigma$ is halved every 100 epochs in order to converge closer to the local minimum. The estimated simulation costs are shown per training epoch in figure 4.7 for the first 300 training epochs. The algorithm converges after less than 200 episodes. The estimate of the accumulated costs is noisy as it is estimated as a sample mean over $N_{traj}$ trajectories.

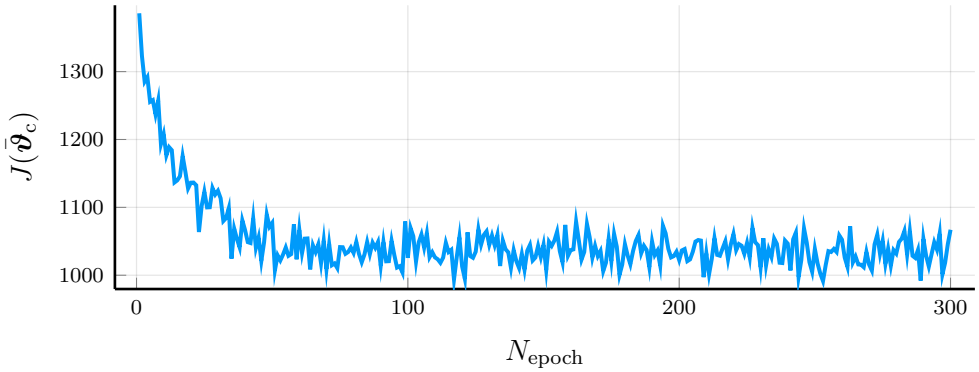



Figure 4.7: Estimate of the average accumulated costs per epoch during the training of $r_{\text{ES}}(\mathbf{x}_t, \mathbf{h}_t)$ using ES.

Table 4.5: Hyper-parameters used to train a recurrent neural network using CMAES.

| Variable | Value | Description |
|---|---|---|
| $N_{epoch}$ | 500 | Number of training epochs |
| $N_{pop}$ | 200 | Population size. |
| $N_{traj}$ | 50 | Number of sampled trajectories for one evaluation of the loss $J(\boldsymbol{\vartheta}_c)$. |
| $T$ | 300 | Number of discrete time-steps used in the trajectory samples |
| $\sigma$ | 0.1 | Initial standard deviation of the parameter distribution. |

Next, we use CMAES on the same problem. The number of parameters of the recurrent neural network used for this problem is small (497 parameters), therefore, this second order method can be used. We use our own implementation of algorithm 4 with the hyper-parameters given in table 4.5. The average costs over all parameter candidates are shown per training epoch in figure 4.8 for the first 300 epochs. The algorithm converges after around 120 epochs.

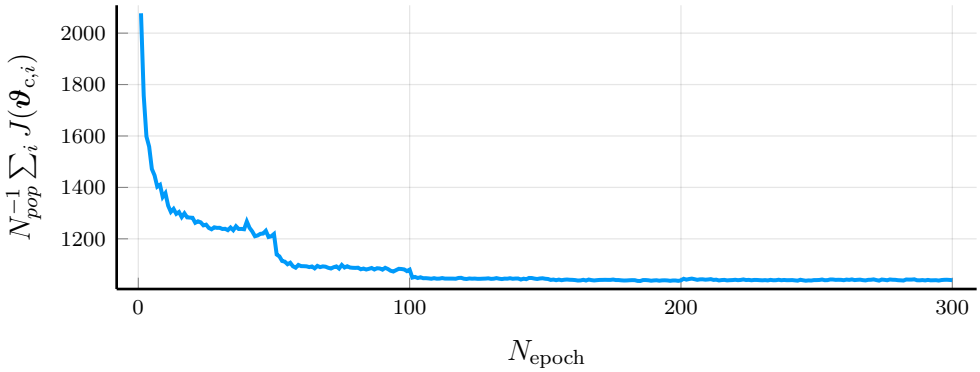



Figure 4.8: Estimate of the average accumulated costs per epoch during the training of $r_{\text{CMAES}}(\mathbf{x}_t, \mathbf{h}_t)$ using CMAES.

## 4.4 Analysis and results

In the following, we evaluate the oracle controller $g(\mathbf{x}_t, \mathbf{p})$ and the recurrent controller $r(\mathbf{x}_t, \mathbf{h}_t)$ with respect to their closed-loop performance and compare their costs to the costs achieved using trajectory optimization. The question that we want to answer by this analysis is how much the function approximation is affecting the closed-loop performance and which step of the DOI approach leads to degenerations of the performance.

Furthermore, we include four controllers for a comparison. The first controller $g(\mathbf{x}_t)$ is trained using imitation learning with trajectory optimization, without varying the model parameters. The controllers $g_{\text{Epopt}}(\mathbf{x})$ is trained using reinforcement learning, specifically using Epopt with PPO, as explained in section 4.3.1. The controller $r_{\text{ES}}(\mathbf{x}, \mathbf{h})$ is trained using evolution strategies and finally the controller $r_{\text{CMAES}}(\mathbf{x}, \mathbf{h})$ is trained using CMAES, both designs are described in section 4.3.2 . In the following, an overview over the controllers included in the comparison is provided.

- $g(\mathbf{x}, \mathbf{p})$ is an oracle controller, an intermediate result used as a teacher in DOI. It is included in the results to clarify the difference in costs for each step of our method in section 3.

- $r(\mathbf{x}, \mathbf{h})$ is a recurrent controller, trained using DOI, and is in the focus of the comparison.

- $g(\mathbf{x})$ is a static controller, trained using behavioral cloning on trajectories generated using constant model parameters.
- $g_{\text{Epopt}}(\mathbf{x})$ is a robust controller, trained using EPOPT.
- $r_{\text{ES}}(\mathbf{x}, \mathbf{h})$ is an adaptive controller, trained using ES.
- $r_{\text{CMAES}}(\mathbf{x}, \mathbf{h})$ is an adaptive controller, trained using CMAES.

We use a validation set of 2000 validation trajectories for the evaluation of the control performance. We perform simulations for each controller given the initial states $\mathbf{x}_0 \in \mathcal{X}_0$ in the validation set and evaluate the accumulated costs for each trajectory. Statistics of the costs are then presented to compare the controllers. The first statistic is the sample mean over the trajectory costs

$$J_1 = \frac{1}{|\mathcal{X}_0|} \sum_{\mathbf{x}_0 \in \mathcal{X}_0} \sum_t c(\mathbf{x}_t, u_t) \approx \mathbb{E}_{\mathbf{x}_0 \sim P_0, \mathbf{p} \sim P_p} \left[ \sum_t c(\mathbf{x}_t, u_t) \right]. \tag{4.12}$$

The metric $J_1$ evaluates the average control performance. However, to assess robustness against model parameters, we are interested in the worst case scenario. We cannot pick the highest trajectory costs, since the costs largely depend on the starting state and therefore evaluating the highest cost for each controller would not take trajectories with favorable initial states into account. Instead, we subtract the accumulated costs that are calculated using the interior point algorithm from the accumulated costs of the controller. With state action pairs $(\mathbf{x}_t^*, u_t^*)$ belonging to optimized trajectories in the validation set, and $(\mathbf{x}_t, u_t)$ simulated states and actions using one of the controllers, our second metric is

$$J_2 = \max_{\mathbf{x}_0 \in \mathcal{X}_0} \left( \sum_t c(\mathbf{x}_t, u_t) - c(\mathbf{x}_t^*, u_t^*) \right). \tag{4.13}$$

The metric $J_2$ is 0 only if the control law can perfectly reproduce the optimal trajectories for each starting state in the validation set, otherwise positive.

The values for both metrics are given for the different controllers in table 4.6. As a reference, the metric $J_1$ evaluates to 1008.91 for the trajectories in the validation set that are optimized using an interior point solver.

Table 4.6: Evaluation of the control performance with regards to both metrics $J_1$ and $J_2$ for different controllers.

| | $g(\mathbf{x}, \mathbf{p})$ | $r(\mathbf{x}, \mathbf{h})$ | $g(\mathbf{x})$ | $g_{\text{Epopt}}(\mathbf{x})$ | $r_{\text{ES}}(\mathbf{x}, \mathbf{h})$ | $r_{\text{CMAES}}(\mathbf{x}, \mathbf{h})$ |
|---|---|---|---|---|---|---|
| $J_1$ | 1009.36 | 1008.74 | 1024.62 | 1200.41 | 1037.0 | 1043.93 |
| $J_2$ | 77.59 | 77.43 | 168.91 | 245.91 | 314.85 | 162.58 |

As a first result, we see that the average accumulated costs of the controller trained using reinforcement learning, i.e., $g_{\text{Epopt}}(\mathbf{x})$, is higher than the costs of the controllers trained using supervised learning or evolution strategies. While reinforcement learning optimizes the control parameters in a direct manner and can in theory reach a local minimum for the accumulated costs, the sensitivity of the final performance on the hyper-parameters often leads to sub-optimal controllers as is the case here. Moreover, $g_{\text{Epopt}}(\mathbf{x})$ is trained to be robust in the worst case scenario, which can lead to an increase in the average accumulated costs compared to a training without Epopt. The metric $J_2$ is also higher for $g_{\text{Epopt}}(\mathbf{x})$ compared to all other controllers except $r_{\text{ES}}(\mathbf{x}, \mathbf{h})$. Comparing the performance with $g(\mathbf{x})$, we see an increase in the average accumulated costs by about 171 and an increase in the worst case costs by 77, as such, we conclude that the higher costs of $g_{\text{Epopt}}(\mathbf{x})$ are not caused by the lack of information on the model parameters, but rather on the algorithm not fully converging to a local minimum.

The adaptive controllers trained using evolution strategies, $r_{\text{ES}}(\mathbf{x}, \mathbf{h})$ and $r_{\text{CMAES}}(\mathbf{x}, \mathbf{h})$, perform better in terms of average costs than $g_{\text{Epopt}}(\mathbf{x})$, but worse than the controllers trained using supervised learning. There is no clear winner between both controllers, as $r_{\text{ES}}(\mathbf{x}, \mathbf{h})$ has lower average costs indicated by $J_1$, but higher costs in the worst case scenario indicated by $J_2$.

Comparing the controllers trained using imitation learning with trajectory optimization, i.e., $g(\mathbf{x})$, $g(\mathbf{x}, \mathbf{p})$ and $r(\mathbf{x}, \mathbf{h})$, we see that $g(\mathbf{x})$ produces significantly higher average costs, with more than 1.5% above the costs of the trajectories optimized using the interior point solver. The controllers $g(\mathbf{x}, \mathbf{p})$ and $r(\mathbf{x}, \mathbf{h})$ that include information about the model parameters or the state history deviate from the costs produced by the interior point solver by less than 0.05%. This is to be expected due to the increase in information

available to the controller. Surprisingly, the recurrent controller produces lower costs than the oracle controller that was its teacher and even lower than the optimal trajectories in the validation set. Since $r(\mathbf{x}, \mathbf{h})$ is trained using DOI and uses $g(\mathbf{x}, \mathbf{p})$ as its reference instead of the cost function, we believe that its superior control performance is a coincidence and that the costs might as well have turned out higher than the costs of the oracle controller. The average costs of the optimized trajectories reveal that some of the trajectories converged to a local optimum during the optimization using the interior point solver.

Both controllers $g(\mathbf{x}, \mathbf{p})$ and $r(\mathbf{x}, \mathbf{h})$ perform similarly in terms of robustness as is indicated by values $J_2$ of 77.59 and 77.43 respectively. The value is lower than the values of the controllers trained using reinforcement learning or evolution strategies, indicating a higher robustness against variations of the model parameters.

We conclude that for this problem, the best control performance, both in terms of average and worst case performance is achieved by the oracle controller $g(\mathbf{x}, \mathbf{p})$ and the adaptive controller $r(\mathbf{x}, \mathbf{h})$. Both have very similar performance and the average costs even compete with the costs of trajectories generated using an interior point optimizer. For this simulation problem, we see that neither the training of an oracle using behavioral cloning or the training of an adaptive controller using DOI significantly changes the final performance. Of the two controllers, $g(\mathbf{x}, \mathbf{p})$ cannot be used in an application example with uncertain or varying $\mathbf{p}$, whereas $r(\mathbf{x}, \mathbf{h})$ has no such restrictions.

As a additional remark without rating, we provide the wall clock time required to generate the controllers. In our case, the controllers were trained on a computer equipped with an Intel i7-4770 CPU and Nvidia GTX 1060 GPU. The simulation and trajectory optimizations were run in parallel on four threads and all supervised learning steps were executed on the GPU. The approximate computing times for the control design steps as measured for the amount of training epochs used in the results are given in table 4.7. Since most of the algorithms converged earlier than the predefined number of epochs and the number of trajectories used for supervised learning is larger than required, the minimal time to train a controller using each method is smaller than indicated. Moreover, the computing times depend on many

factors like hyper-parameters, programming skills, and the model and task at hand. Therefore, the values need to be interpreted with care and cannot easily be generalized.

Table 4.7: Computation times in minutes used to train the controllers in this chapter.

| Optimization | Time in minutes |
|---|---|
| Optimizing 5000 trajectories | 270 |
| Training the oracle $g(\mathbf{x}, \mathbf{p})$ (10000 epochs) | 13 |
| Training $r(\mathbf{x}, \mathbf{h})$ using DOI (500 epochs) | 4 |
| Total time to train $r(\mathbf{x}, \mathbf{h})$ | 287 |
| EPOPT for $g_{\text{Epopt}}(\mathbf{x})$ (4000 epochs) | 511 |
| ES for $r_{\text{ES}}(\mathbf{x}, \mathbf{h})$ (1000 epochs) | 195 |
| CMAES for $r_{\text{CMAES}}(\mathbf{x}, \mathbf{h})$ (500 epochs) | 110 |

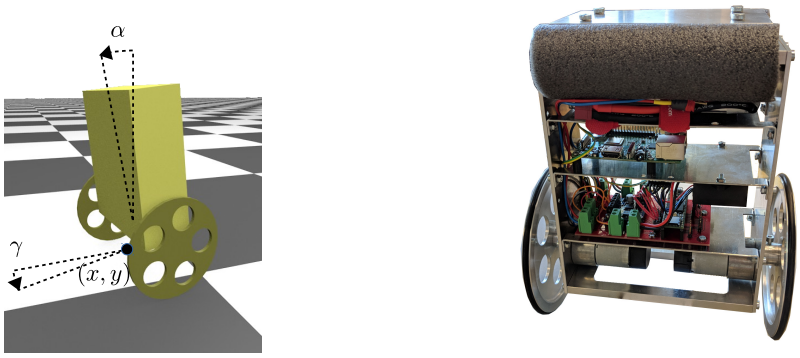
Chapter 5

# Application example of a mobile inverted pendulum

This section presents the application of an adaptive controller, trained using DOI, to the mobile inverted pendulum (MIP). The structure of the chapter is similar to chapter 4. First the system and its model is presented. Afterwards the control design details are provided before showing results both in simulation and application. The application results are published in our previous paper [20].

## 5.1 System and model

The MIP, shown in figure 5.1, consists of a rigid rectangular body that is supported by two wheels. The concept of the robot is known in the control community for over three decades [113]. The specific robot shown in figure 5.1b was built as a part of this thesis. A requirement for the robot was to have enough computing power to evaluate simple neural networks. The body and wheels consist of aluminum constructions with padding added at the top of the robot to allow it to fall over without being damaged. The robot is propelled by two DC motors that are connected via a gearbox to the wheels. The energy is provided by a single 12V battery with a capacity of 3000mAh, powering the motors as well as the rest of the electronics. The sensors of the system consists of two motor encoders measuring each motor's




(a) Schematic view of the MIP (b) The real MIP

Figure 5.1: The mobile inverted pendulum (MIP).

rotation and an inertial measurement unit (type MPU6050) measuring the acceleration and rotation speed of the robot's body in all axes. The sensor signals are read and processed by a micro-controller board (type Teensy 3.6) and then sent in the form of a state vector $\mathbf{x}_t$ to a single-board computer (type Raspberry Pi 3B+). The single-board computer evaluates the control law, i.e., the neural network, and sends the control signal $\mathbf{u}_t$, consisting of the pulse width modulation (PWM) duty cycle for each motor, back to the micro-controller. From there, the signal is forwarded to a motor driver board and finally to the motors. The cycle time, which corresponds to the discrete time step size in the simulation, is set to $\delta_t = 0.01$s.

The model of the system consists of the motor model and the rigid body dynamics. We adopt the model of Pathak et al. [82] for the rigid body dynamics and repeat the model equations without derivation in the appendix A.1. The system is described by the seven dimensional state vector $\mathbf{x} = \left[x, y, \varphi, \alpha, \dot{\alpha}, v, \dot{\varphi}\right]^T$ consisting of the position coordinates $x, y$ of the MIP in the horizontal plane, the forward velocity $v$, the yaw angle $\varphi$ and its derivative $\dot{\varphi}$ as well as the tilt angle $\alpha$ and its derivative $\dot{\alpha}$.

The input to the model in [82] is the torques $\boldsymbol{\tau}$ generated by the two motors. For each motor, we use the DC motor model already presented in section 4.1. We add a heuristic friction model with three parameters $c_{\text{fric},1-3}$ generating a torque $\tau_{\text{fric}}$ that depends on the angular velocity of the wheels $\omega$ and reduce the number of parameters of the motor torque model in equation

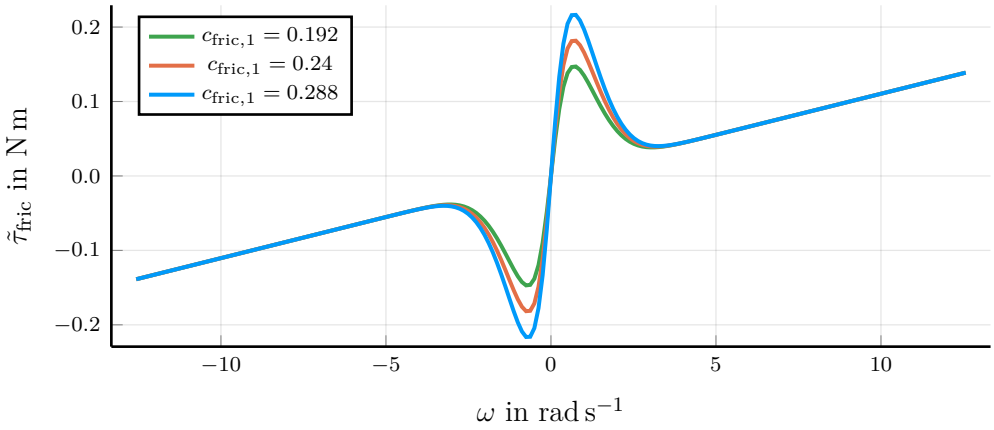



Figure 5.2: The friction model in equation (5.2) for different $c_{\text{fric},1}$ with $c_{\text{fric},2}, c_{\text{fric},3}, k_1, k_2$ as given in table 5.1.

(4.4) to only contain two unknown parameters $k_1$ and $k_2$. The friction model is heuristically chosen to be a smooth approximation of the Stribeck friction [6]. Viscous friction is not modelled, as its effect on the torque is the same as the back EMF and is therefore already considered in the motor parameter $k_1$. The reduction of the number of parameters is done in preparation for a parameter identification, since the real motor parameters are not provided. The final motor model used for the MIP is

$$\tau_{\text{fric}} = c_{\text{fric},1} \cdot \tanh(c_{\text{fric},2} \cdot \omega) \cdot e^{-c_{\text{fric},3} \cdot \omega^2} \tag{5.1a}$$

$$\tau = k_2 \cdot (u - k_1 \cdot \omega) - \tau_{\text{fric}}. \tag{5.1b}$$

Since we identify the parameters and the physical interpretation is lost, one could also consider the product $k_1 \cdot k_2 \cdot \omega$ as the viscous friction moment. The so obtained total friction moment

$$\tilde{\tau}_{\text{fric}} = \tau_{\text{fric}} + k_1 k_2 \omega \tag{5.2}$$

is visualized in figure 5.2 for different values of $c_{\text{fric},1}$ with $c_{\text{fric},2}, c_{\text{fric},3}, k_1, k_2$ as given in table 5.1. Note that $\omega$ is the angular velocity of the wheels, instead of the angular velocity of the motor. Both are related to each other by a factor depending on the gearbox of the motor. The angular velocity of the wheels $\omega$ can be expressed as a function of the states $v$ and $\dot{\varphi}$ together with

the distance between the two wheels $2 \cdot b$ and the wheel radius $R$, as

$$\omega_l = \frac{1}{R}\left(v - b\dot{\varphi}\right) \tag{5.3a}$$

$$\omega_r = \frac{1}{R}\left(v + b\dot{\varphi}\right). \tag{5.3b}$$

The indices $l$ and $r$ are used in the following to distinguish between left and right wheel or motor, should it be necessary. We use system identification to estimate the motor parameters $c_{\text{fric},1-3}$, $k_1$ and $k_2$, assuming the same parameters for both motors. The data for the system identification was provided by stabilizing the robot using a hand-tuned PID-controller with small noise added to the control signal. For the other parameters, masses and distances are measured directly, whereas the moments of inertia are approximated using formulas from classical mechanics, which can be found, e.g., in [70, p. 318-321].

All relevant model parameters for our MIP model are given in table 5.1. The parameters $c_z$, $I_{yy}$ and $c_{\text{fric},1}$ will be varied for the control design for the following reasons. The parameter $c_z$ is difficult to measure exactly and has a strong influence on the dynamic behavior. The approximate calculation of $I_{yy}$ assumes a uniform density that is not reflected in the real system, e.g., the battery at the top is heavier than the circuit board in the middle of the robot. $c_{\text{fric},1}$ is included due to its strong influence on the behavior close to the position of rest.

## 5.2 Task description

Mobile inverted pendula have been around for more than three decades as a popular test system for control design methods [113], therefore, the existing literature dealing with this system is vast. For an extensive literature review, we refer to the work of Murdock [74, p. 7–9]. The control design for the MIP is often done using a linearized model [24, 51, 71] and as such limited to small deviations of the tilt angle and position error.

The task we solve in our work is to design a controller that allows driving to arbitrary positions $(x_r, y_r)$ within a certain radius using only feedback control, i.e., without the need to generate a feasible trajectory online. The

Table 5.1: Model parameters of the MIP.

| Variable | Value | Unit | Description |
|---|---|---|---|
| $M_\mathrm{b}$ | 1.76 | kg | Mass of the body |
| $M_\mathrm{w}$ | 0.147 | kg | Mass of a wheel |
| $R$ | 0.07 | m | Radius of a wheel |
| $c_z$ | $0.07 \pm 20\%$ | m | Height of the center of mass above wheel axis |
| $b$ | 0.09925 | m | Half length between wheels |
| $I_{xx}$ | 0.0191 | $\mathrm{kg\,m^2}$ | Moment of inertia, x-axis |
| $I_{yy}$ | $0.0158 \pm 20\%$ | $\mathrm{kg\,m^2}$ | Moment of inertia, y-axis |
| $I_{zz}$ | 0.0048 | $\mathrm{kg\,m^2}$ | Moment of inertia, z-axis |
| $I_{wa}$ | $3.6 \cdot 10^{-4}$ | $\mathrm{kg\,m^2}$ | Moment of inertia. Wheel, y-axis |
| $I_{wd}$ | $1.45 \cdot 10^{-3}$ | $\mathrm{kg\,m^2}$ | Moment of inertia. Wheel, z-axis |
| $k_1$ | 0.018131 | $\mathrm{V\,s}$ | Motor constant |
| $k_2$ | 0.61 | $\mathrm{N\,m\,A^{-1}}$ | Motor constant |
| $c_{\mathrm{fric},1}$ | $0.24 \pm 20\%$ | $\mathrm{N\,m^{-1}}$ | Friction model constant |
| $c_{\mathrm{fric},2}$ | 2.0 | / | Friction model constant |
| $c_{\mathrm{fric},3}$ | 0.4 | / | Friction model constant |

advantage of not using a trajectory generation online is that less computing power is necessary and the computation time is constant, which is not the case for nonlinear model predictive control [23]. We choose the radius in which the target positions $(x_r, y_r)$ must lie to be 1m, however, larger radii can be chosen at the expense of generating more training data.

For the task at hand, it is sufficient to train a controller to drive into the origin of the coordinate system $(x_r, y_r) = (0, 0)$ with $\gamma_r = 0$. To drive to a different target position and orientation $(x_r, y_r, \gamma_r)$ in the inertial coordinate system $I$, we use a coordinate transformation into a coordinate system $B$ that has its origin at the target position and is rotated by $\gamma_r$. Figure 5.3 visualizes the coordinate transformation. The position and orientation of

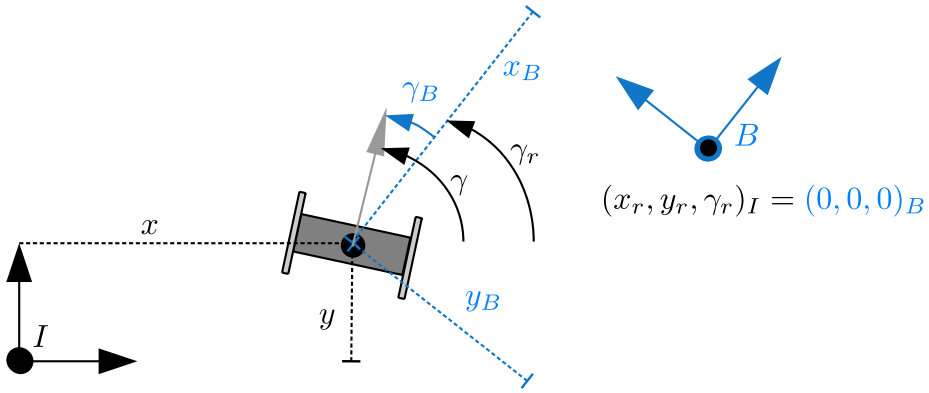



Figure 5.3: Visualization of the coordinate transformation, placing the origin of a new coordinate system $B$ in the target $(x_r, y_r, \gamma_r)_I$ in the inertial coordinate system $I$.

the MIP $(x, y, \gamma)$, expressed in the coordinate system $B$ is $(x_B, y_B, \gamma_B)$ with the transformation

$$[\delta x, \delta y]^T = [x - x_r, y - y_r]^T \tag{5.4a}$$

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} \cos(\gamma_r) & \sin(\gamma_r) \\ -\sin(\gamma_r) & \cos(\gamma_r) \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \tag{5.4b}$$

$$\gamma_B = \gamma - \gamma_r. \tag{5.4c}$$

The remaining states $\alpha, \dot{\alpha}, v, \dot{\gamma}$ do not differ between the two coordinate systems. Thus, after the controller has been trained to steer the MIP into the origin of the inertial coordinate system, any other reference point can be targeted by applying the coordinate transformation (5.4) and using the control input $\tilde{\mathbf{x}} = [x_B, y_B, \gamma_B, \alpha, \dot{\alpha}, v, \dot{\gamma}]^T$.

## 5.3 Adaptive control design

In the following, the control design using DOI is presented for an adaptive controller of the form $[\mathbf{u}_t^T, \mathbf{h}_{t+1}^T]^T = \mathbf{r}(\mathbf{x}_t, \mathbf{h}_t)$. Additionally, a non-adaptive controller and an adaptive and adjustable controller are also trained using imitation learning and trajectory optimization.

Other methods like reinforcement learning or evolution strategies did not converge to a usable controller for this system. Moreover, we added equality constraints to ensure a convergence to the target position that cannot be included by any of the other mentioned methods.

### 5.3.1 Trajectory generation

In the first step, the trajectory optimization for the given task is implemented. We use a cost function that punishes deviations from the origin $(x, y) = (0, 0)$ as well as deviations in the yaw angle. To allow full rotations, the yaw angle is punished using a trigonometric function as $(1-\cos\gamma)$. High angular velocities or forward velocities as well as high control signals $u_l, u_r$ are also punished, and for the tilt angle, we add the term $\alpha \cdot \dot{\alpha}$ to further punish falling over while rewarding angular velocities that help the MIP get back up. The full cost function is

$$
\begin{aligned}
c(\mathbf{x}, \mathbf{u}) =& c_{\mathbf{x}}(\mathbf{x}) + c_{\mathbf{u}}(\mathbf{u}) && \text{(5.5a)}\\
c_{\mathbf{x}}(\mathbf{x}) =& x^2 + y^2 + 0.5 \cdot (1 - \cos\gamma) + \alpha^2 \\
&+ 2 \cdot \left(\alpha \cdot \dot{\alpha} + 0.005 \cdot \dot{\alpha} + 0.1 \cdot v^2 + 0.1 \cdot \dot{\gamma}^2\right) && \text{(5.5b)}\\
c_{\mathbf{u}}(\mathbf{u}) =& u_l^2 + u_r^2. && \text{(5.5c)}
\end{aligned}
$$

The constant weights for each term were tuned manually to produce a subjectively appealing control behavior.

For the given optimization horizon $T$, the cost function itself is not able to produce trajectories that drive consistently into the origin. Instead, the MIP drives to a position close to the target with an offset largely depending on the start state. Therefore, we add end constraints that ensure that the MIP reaches the target position after $T = 501$ discrete time steps

$$\begin{aligned} &x_T = y_T = \alpha_T = \dot{\alpha}_T = v_T = \dot{\gamma}_T = 0 \ ; \ \cos\gamma_T = 1 \\ &u_{l,T-1} = u_{r,T-1} = 0. \end{aligned} \tag{5.6}$$

The constraints on the control input $\mathbf{u}_{T-1}$ lead to a smoother control signal near the end position. Depending on the initial state, the optimizer returns a control signal with sudden peaks, or a noisy control signal in some parts. Reproducing this signal is difficult for a function approximator. Therefore, we add further constraints to restrict the second derivative of the control signal. The constraints are expressed as a finite difference expression, cf. equation 2.64, and assure a smooth control signal and trajectory. The constraints are

$$\left| \frac{u_{t-1} - 2 \cdot u_t + u_{t+1}}{\delta_t^2} \right| \leq k, \ \forall t \in \{1, \ldots, T-2\} \tag{5.7}$$

The optimization problem for the trajectories thus accounts to:

$$\mathbf{X}^*, \mathbf{U}^* = \underset{\mathbf{X},\mathbf{U}}{\arg\min} \sum_{t=0}^{T-1} c(\mathbf{x}_t, \mathbf{u}_t) \tag{5.8a}$$

$$\text{s.t. } \mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t, \mathbf{p}) \tag{5.8b}$$

$$\mathbf{x}_{t=0} = \mathbf{x}_0 \tag{5.8c}$$

$$|u_r| < 1; \ |u_l| < 1 \tag{5.8d}$$

Constraints in (5.6) and (5.7).

The initial state $\mathbf{x}_0$ is randomized using uniform distributions in polar coordinates as follows

$$r \sim \mathcal{U}(0, 1.1); \ \varphi_0 \sim \mathcal{U}(0, 2\pi) \tag{5.9a}$$

$$x_0 = r\cos\varphi; \ y_0 = r\sin\varphi \tag{5.9b}$$

$$\alpha_0 \sim \mathcal{U}(-\frac{\pi}{9}, \frac{\pi}{9}); \ \dot{\alpha}_0 \sim \mathcal{U}(-\frac{1}{2}, \frac{1}{2}) \tag{5.9c}$$

$$v \sim \mathcal{U}(-\frac{3}{2}, \frac{3}{2}); \ \dot{\varphi} \sim \mathcal{U}(-2, 2). \tag{5.9d}$$

The radius of the initial positions is chosen 1.1m instead of 1m as is indicated in the original task description to make sure that the states at a distance of 1m are covered by the generated data and the oracle controller can later interpolate instead of extrapolate. The model parameters $c_z$, $I_{yy}$ and $c_{\text{fric},1}$ are sampled for each trajectory in a uniform distribution of $\pm 20\%$ around the estimated value as

$$I_{yy} \sim \mathcal{U}(0.8 \cdot 0.0158, 1.2 \cdot 0.0158) \tag{5.10a}$$

$$c_{\text{fric},1} \sim \mathcal{U}(0.8 \cdot 0.24, 1.2 \cdot 0.24) \tag{5.10b}$$

$$c_z \sim \mathcal{U}(0.8 \cdot 0.07, 1.2 \cdot 0.07). \tag{5.10c}$$

For the adjustable controller, we modify the state costs $c_{\mathbf{x}}(\mathbf{x})$ to contain an adjustable parameter $\lambda \in [-1, 1]$

$$\begin{aligned} c_{\mathbf{x}}(\mathbf{x}, \lambda) =& 10^{\lambda} \cdot \left(x^2 + y^2 + 0.5 \cdot (1 - \cos\gamma) + \alpha^2\right) \\ &+ 2 \cdot 10^{-\lambda} \cdot \left(\alpha \cdot \dot{\alpha} + 0.005 \cdot \dot{\alpha} + 0.1 \cdot v^2 + 0.1 \cdot \dot{\gamma}^2\right). \end{aligned} \tag{5.11}$$

For $\lambda = 0$, the cost function is equal to (5.5c). For $\lambda > 0$, deviations from the target position are punished stronger, and velocities are punished less, thus, the behavior is changed towards a faster approach of the target position. For $\lambda < 0$ the opposite is true, the behavior is changed towards a slower approach of the target position. The influence of $\lambda$ on the transition behavior is visualized in figure 5.4 for $\lambda \in \{-1, 0, 0.5, 1\}$.

We optimize 10000 trajectories for both cost functions $c_{\mathbf{x}}(\mathbf{x})$ and $c_{\mathbf{x}}(\mathbf{x}, \lambda)$, however, we show later that the task can be solved with fewer trajectories. Another 2000 optimal trajectories are created as a validation set that is used as a reference for the achievable control performance. Different from the training trajectories, the initial states for the validation trajectories are sampled in a radius of 1m around the origin, i.e., $r \sim \mathcal{U}(0, 1)$ and given initial velocities $\dot{\alpha}_0 = v_0 = \dot{\gamma}_0 = 0$. The initial states of the trajectories in the validation set are used throughout this chapter whenever the average control performance is estimated. The set containing the initial states of the validation trajectories is denoted $\mathcal{X}_0$ in the following.

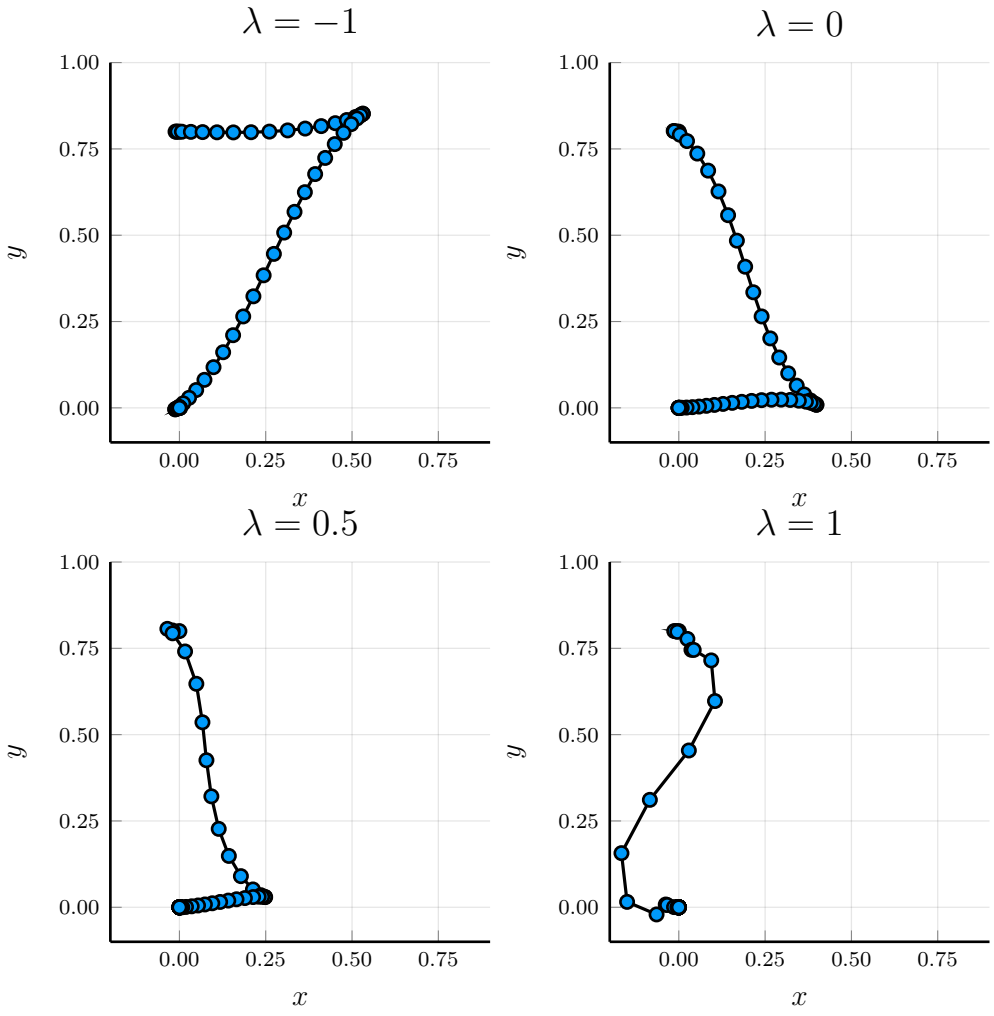



Figure 5.4: Position coordinates $(x_t, y_t)$ for solving (5.8) using different values for $\lambda$ and $(x_0, y_0, \gamma_0) = (0, 0.8, 0)$ with zero initial velocities. A marker is placed every 0.1s of the trajectory.

### 5.3.2 Oracle training

After the training data in the form of state-action tuples is created, the oracle controller $\mathbf{g}(\mathbf{x}, \mathbf{p}; \boldsymbol{\vartheta}_\mathrm{c})$ is trained using supervised learning as described in section 3.2.2.

The oracle is parameterized as a neural network with two hidden layers of 128 neurons each. The activation functions for the hidden layers are the tanh function and the output layer is linear. The inputs for the neural network are both the state $\mathbf{x}_t$, with $\gamma$ split up in $\cos\gamma$ and $\sin\gamma$ to be indifferent of full rotations, and the model parameters $\mathbf{p}$.

The data generated in the previous step is split in a training set of 80% of the data, and a testing set of the remaining data. We use a mini-batch size of 8192 tuples and randomize the data sequence every training episode. We use Adam as our optimizer and run the training for a maximum of 10000 episodes. The control performance of the oracle controller is evaluated afterwards in simulation. To check if we need more data, the influence of the number of optimized trajectories on the control performance is analyzed. Multiple controllers are trained using different numbers of trajectories and evaluated every 250 training epochs. The average control performance $\mathbb{E}\left[\sum_t^T c(\mathbf{x}_t, \mathbf{u}_t)\right]$ over the number of training epochs $N_{gd}$ is shown in figure 5.5 for the different controllers.

The final control performance is increasing with the number of trajectories, whereas no noticeable improvement is observed for more than 5000 optimal trajectories. Using as few as 20 optimal trajectories as training data leads to significantly higher costs than for all other controllers, because the resulting controller is not able to stabilize the system.

We observe that for 100 optimized trajectories, the average accumulated costs reach their minimum after around 1200 epochs of supervised learning. Further training reduces the error on the training and testing data-sets, however, the closed-loop performance worsens. This effect was already observed on the bridge crane example. The states visited during simulation are not covered sufficiently by the training data. The data is correlated in space because it results from simulations. Therefore, the controller is fitted to the data distribution that is not sufficiently related to the state distribution for

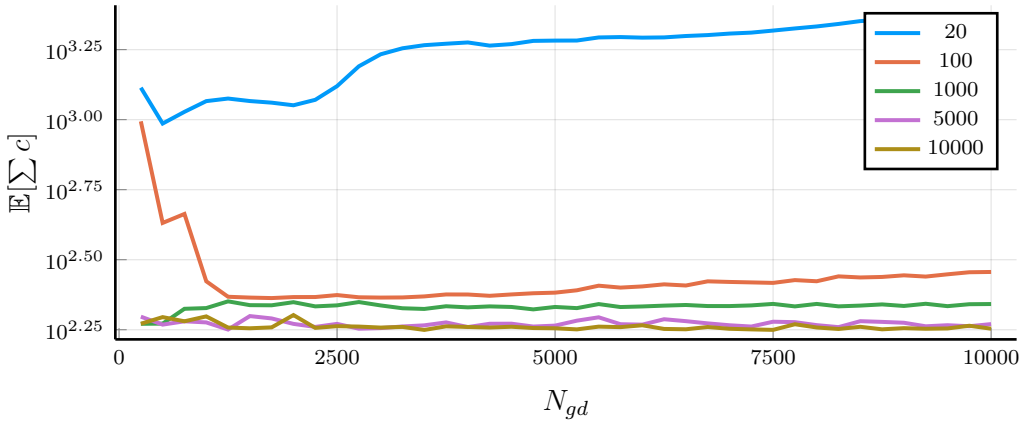



Figure 5.5: Mean accumulated costs of oracle controllers $\mathbf{g}(\mathbf{x}, \mathbf{p})$ trained on different numbers of trajectories over $N_{gd}$ epochs of supervised learning. The number of trajectories used as training data is given in the line label.

simulations for initial states $\mathbf{x}_0 \sim P_{\mathsf{x}0}(\mathbf{x})$. Apart from the oracle controller $\mathbf{g}(\mathbf{x}, \mathbf{p}; \boldsymbol{\vartheta}_\mathrm{c})$, we also train an adjustable oracle controller $\mathbf{g}_\lambda(\mathbf{x}, \mathbf{p}, \lambda; \boldsymbol{\vartheta}_\mathrm{c})$ with the same training settings.

### 5.3.3 Adaptive control

With the oracle controller available, the adaptive controller $\mathbf{r}(\mathbf{x}, \mathbf{h}; \boldsymbol{\vartheta}_\mathrm{c})$ that is also used later on the physical system is trained using DOI.

The controller is parameterized as a recurrent neural network consisting of a recurrent tanh layer followed by two non-recurrent hidden layers using the tanh function as their activation function and a linear output layer. We choose the number of output neurons to be 32 for the recurrent layer and 64 and 32 for the following hidden layers. Different from the oracle controller, the neural network size of the adaptive controller is limited by the computing power available on the robot as well as the time-step size $\delta_t$.

Each epoch of DOI, 500 trajectories are created by simulating the system using the recurrent controller for $T = 501$ discrete time-steps. The states are disturbed in each time-step by additive noise $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, 10^{-6} \cdot \mathbf{I})$. The initial states and model parameters are sampled from the same distribution as for

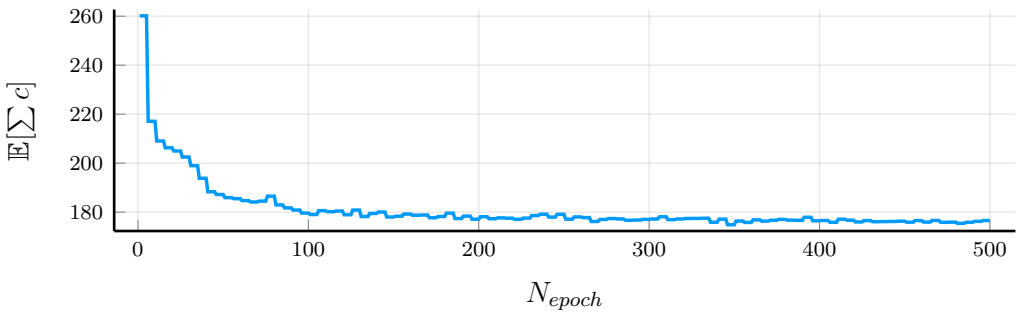



Figure 5.6: Mean accumulated costs of the recurrent controller $\mathbf{r}(\mathbf{x}, \mathbf{h}; \boldsymbol{\vartheta}_c)$ trained using DOI over the number of training epochs $N_{epoch}$

Table 5.2: Approximate computation times in minutes for different optimizations used to create an adaptive controller using DOI for the MIP.

| Optimization | Time in minutes |
|---|---|
| Optimizing 10000 trajectories | 571 |
| Training the oracle $g(\mathbf{x}, \mathbf{p})$ (10000 epochs) | 27 |
| Training $r(\mathbf{x}, \mathbf{h})$ using DOI (500 epochs) | 5 |
| Total time to train $r(\mathbf{x}, \mathbf{h})$ | 603 |

the trajectory optimization, given in equations (5.9), (5.10). For each state $\mathbf{x}_t$ in the simulated trajectories, the oracle controller generates a target action $\hat{\mathbf{u}}_t$. The trajectories are then split in sequences of 50 time steps that are used to train the recurrent neural network using truncated backpropagation through time. The algorithms is run for 500 episodes, after which no further improvement is observed. A plot of the simulation costs per training episode of DOI is shown in figure 5.6. An adjustable version of the recurrent controller $\mathbf{r}_\lambda(\mathbf{x}, \mathbf{h}, \lambda; \boldsymbol{\vartheta}_c)$ is also trained using the same settings.

As an additional resource, we provide the approximate computation times that would be required to train the recurrent controller on an i7-4770 CPU and GTX 1060 GPU using our implementation in the Julia programming language. The actual optimal trajectories were computed on a different machine with 20 CPU cores for a greatly reduced computing time. The time values depend on many factors and are provided only as a rough estimate of the required computing power.

## 5.4 Control performance

In the following, the adaptive controller $\mathbf{r}(\mathbf{x}, \mathbf{h})$ and the adaptive and adjustable controller $\mathbf{r}_\lambda(\mathbf{x}, \mathbf{h}, \lambda)$ generated using DOI are evaluated. For a comparison of the control performance, we include a static controller $\mathbf{g}(\mathbf{x})$ that is trained using the same settings as the oracle controllers for this system, but uses training data that is generated using constant model parameters $I_{yy} = 0.0158, c_{\text{fric},1} = 0.24, c_z = 0.07$. For the results in simulation, we also add the oracle controllers $\mathbf{g}(\mathbf{x}, \mathbf{p})$ and $\mathbf{g}(\mathbf{x}, \mathbf{p}, \lambda)$ as well as the costs generated using the interior point optimizer to the comparison. An overview of the controllers that are used in the comparison is given in figure 5.7.

### 5.4.1 Simulation results

We use the cost function (5.5) that is independent of $\lambda$ for the evaluation. The evaluation uses four metrics, of which two were already used in the bridge crane example. The first metric is the mean costs over 2000 simulations, using the initial states of the validation trajectories

$$J_{\mathbb{E},c} = \frac{1}{|\mathcal{X}_0|} \sum_{\mathbf{x}_0 \in \mathcal{X}_0} \sum_{t=0}^{T-1} c(\mathbf{x}_t, \mathbf{u}_t) \approx \mathbb{E}\left[\sum_{t=0}^{T-1} c(\mathbf{x}_t, \mathbf{u}_t)\right]. \tag{5.12}$$

For the evaluation of the robustness of the controllers, the highest costs over all model parameters is of interest. Identifying the worst model parameters and the respective costs is intractable for this problem, however. As a sample estimate of the worst performance we use a metric that evaluates the highest above optimal costs on our validation set

$$J_{\max,c} = \max_{\mathbf{x}_0 \in \mathcal{X}_0} \left( \sum_{t=0}^{T-1} c(\mathbf{x}_t, \mathbf{u}_t) - c(\mathbf{x}_t^*, \mathbf{u}_t^*) \right). \tag{5.13}$$

The behavior of our controller is also affected by the end constraints that were used during the trajectory optimization. The end constraint violation is therefore used as another metric. Similar to merit functions in the optimization community, see, e.g., Wächter [105, p. 16], we use the $L^2$ norm of the constraints in equation (5.6), evaluated for the final state of the simulation

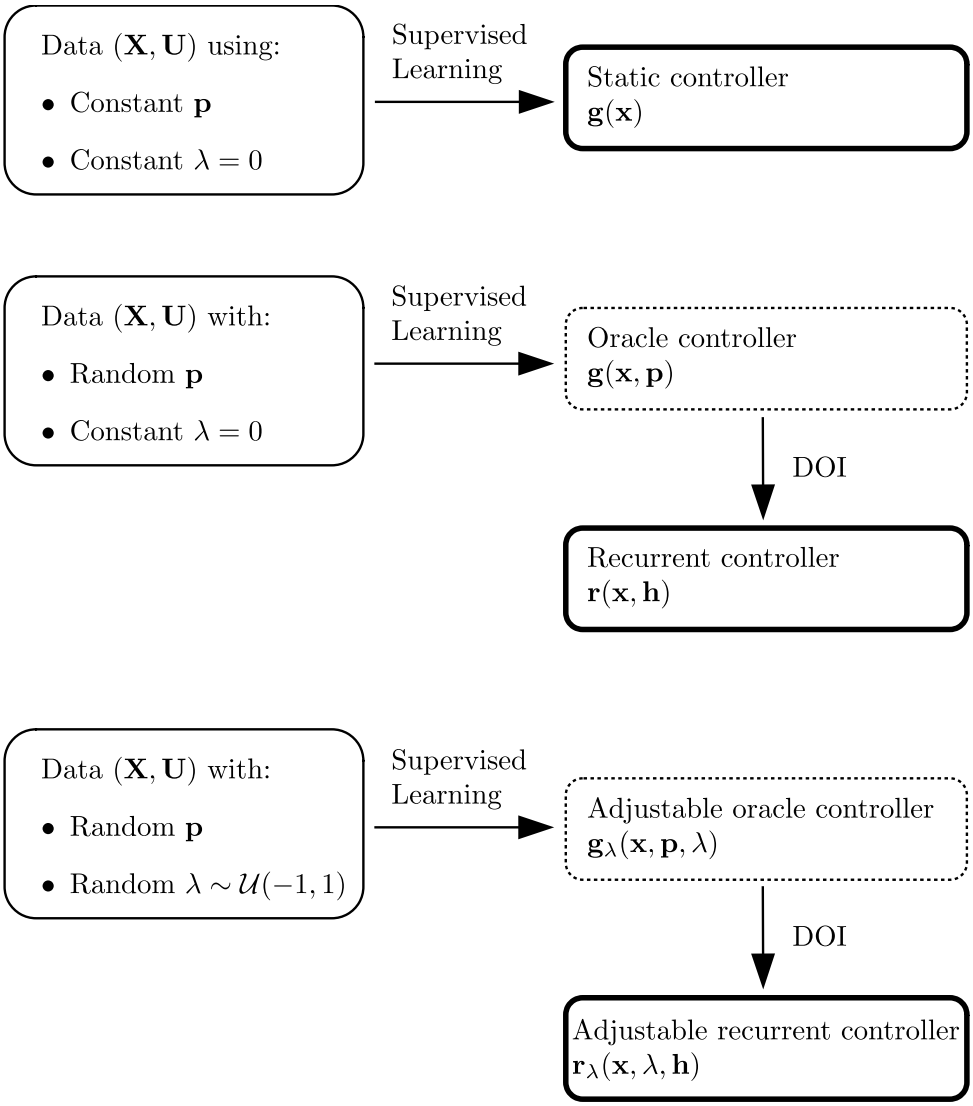



Figure 5.7: An overview of the different parameterized controllers that are compared for the MIP. The data that is used to train the controllers is also indicated. The controllers enclosed in dashed blocks are used in simulation only.

as another performance metric

$$J_{\mathbb{E},c_T} = \frac{1}{|\mathcal{X}_0|} \sum_{\mathbf{x}_0 \in \mathcal{X}_0} x_T^2 + y_T^2 + (1 - \cos\gamma_T)^2 + \alpha_t^2 + \dot{\alpha}_T + v_T + \dot{\gamma}_T^2$$
$$\approx \mathbb{E}\left[x_T^2 + y_T^2 + (1 - \cos\gamma_T)^2 + \alpha_t^2 + \dot{\alpha}_T + v_T + \dot{\gamma}_T^2\right]$$

As a final performance metric, we use a sample estimate of the largest violation of the end constraint. The metric is again evaluated through simulations using the initial states of the validation trajectories.

$$J_{\max,c_T} = \max_{\mathbf{x}_0 \in \mathcal{X}_0} \left(x_T^2 + y_T^2 + (1 - \cos\gamma_T)^2 + \alpha_t^2 + \dot{\alpha}_T + v_T + \dot{\gamma}_T^2\right). \quad (5.14)$$

The evaluation of the four metrics for a static controller $\mathbf{g}(\mathbf{x})$ without parameter information, the oracle controllers $\mathbf{g}(\mathbf{x}, \mathbf{p})$, $\mathbf{g}(\mathbf{x}, \mathbf{p}, \lambda)$, the recurrent controllers $\mathbf{r}(\mathbf{x}, \mathbf{h})$, $\mathbf{r}(\mathbf{x}, \lambda, \mathbf{h})$ and for trajectories that are optimized using the interior point optimizer are depicted in table 5.3. The adjustable controllers are used with $\lambda = 0$ for the comparison, as the cost function $c_{\mathbf{x}}(\mathbf{x}, \lambda)$ is in accordance with the metrics (5.12) and (5.13) only for $\lambda = 0$.

Table 5.3: Mean and maximal accumulated costs for different controllers in simulation. Controllers with gray font cannot be used in the application and are only given as a reference.

| | $\mathbf{g}(\mathbf{x})$ | $\mathbf{g}(\mathbf{x}, \mathbf{p})$ | $\mathbf{r}(\mathbf{x}, \mathbf{h})$ | $\mathbf{g}_\lambda(\mathbf{x}, 0, \mathbf{p})$ | $\mathbf{r}_\lambda(\mathbf{x}, 0, \mathbf{h})$ | traj. opt. |
|---|---|---|---|---|---|---|
| $J_{\mathbb{E},c}$ | 181.47 | 178.33 | 176.58 | 185.81 | 185.75 | 155.163 |
| $J_{\max,c}$ | 712.03 | 520.95 | 121.38 | 494.27 | 182.67 | 0 |
| $J_{\mathbb{E},c_T}$ | 0.133 | 0.102 | 0.090 | 0.070 | 0.051 | 0 |
| $J_{\max,c_T}$ | 1.10 | 2.13 | 1.34 | 1.092 | 0.524 | 0 |

Looking at the values of $J_{\mathbb{E},c}$ in the table, we can see which step of our DOI approach led to the largest increase of the closed-loop costs for the final controller. The mean costs of the directly optimized trajectories is the lowest with a value of 155.163 whereas all the parameterized controllers have average costs in a range from 176-186. The training of the oracle controllers $\mathbf{g}(\mathbf{x}, \mathbf{p})$ and $\mathbf{g}_\lambda(\mathbf{x}, \lambda, \mathbf{p})$ using supervised learning led to an increase of the mean costs of between 13% and 20%. A possible reason for the increase in costs is the

use of equality and inequality constraint in the trajectory optimization that produce ambiguous training data. The increase in costs is smaller for the oracle controller that was trained on trajectories that were optimized using a constant cost function. The oracle controller that is adjustable through $\lambda$ was trained on data that includes this additional parameter, therefore the input data is of higher dimension and the relation between input and output, i.e, between $(\mathbf{x}_t, \mathbf{p}, \lambda)$ and $\mathbf{u}_t$, is more difficult to learn.

Still looking at $J_{\mathbb{E},c}$, we see that the mean costs between the oracle controllers and the recurrent controllers vary less than 1% in this case. The value for the metric is smaller for the adaptive controllers than for their individual oracle controllers. This has already been observed for the bridge crane example, nevertheless, we believe this to be a random deviation from the oracle performance that can also lead to an increase in the costs.

The static controller $\mathbf{g}(\mathbf{x})$ that is given as a reference performs slightly worse than the non-adjustable oracle and recurrent controllers, but slightly better than the adjustable controllers. A possible explanation for the larger mean costs of the adjustable controllers is that the data used for this controller is the richest in information, while the neural network size and amount of data is the same for all controllers.

Looking at $J_{\max,c}$, which indicates the largest above optimal costs in simulation for initial states $\mathbf{x}_0 \in \mathcal{X}_0$, the recurrent controllers $\mathbf{r}(\mathbf{x}, \mathbf{h})$ and $\mathbf{r}(\mathbf{x}, \lambda, \mathbf{h})$ perform better than the non-recurrent controllers with values of 121.38 and 182.67. The static controller $\mathbf{g}(\mathbf{x})$ without information about the model parameters has the largest value of 712.03, followed by the oracle controller $\mathbf{g}(\mathbf{x}, \mathbf{p})$ with a value of 520.95 and the adjustable oracle controller $\mathbf{g}_\lambda(\mathbf{x}, 0, \mathbf{p})$ with a value of 494.27. While $J_{\max,c}$ is largely depending on the choice of the initial states, the difference in the values between the recurrent and the static controllers indicates that the recurrent controllers are more robust than the static controllers with regard to varying model parameters and varying initial states. A visualization of simulations for the initial state that leads to the largest costs $J_{\max,c}$ for $\mathbf{g}(\mathbf{x})$ is shown in figure 5.8. As is shown in the figure, the static controllers produce trajectories that do not end in a close region around the target position after 501 time-steps for certain initial states and model parameter combinations. Simulations for the worst initial states for

the oracle and recurrent controller are shown in the appendix A.2.

The metrics $J_{\mathbb{E},c_T}$ gives smaller values for controllers with information about the model parameters, which indicates that, in average, the controllers are closer in terms of the $L^2$ norm to the position of rest in the origin after 501 time-steps. The recurrent controllers even slightly outperform the oracle controllers. The maximum value over all trajectories for each controller, given by $J_{\max,c_T}$, also shows that the recurrent controllers outperform the oracle controllers. This is again unexpected, because the recurrent controllers use the oracle controllers as their teacher and the end constraint is only indirectly included in their training. A possible reason could be that the recurrent controllers produce a smoother control signal due to their dependency on the past history of states, which could be favorable for the problem at hand.

In the following, the controllers are applied to the real system and the control performance is evaluated again for a test trajectory.

### 5.4.2 Results in the application

In the application, we compare the static controller $\mathbf{g}(\mathbf{x})$ and both recurrent controllers $\mathbf{r}(\mathbf{x}, \mathbf{h})$, $\mathbf{r}(\mathbf{x}, \lambda, \mathbf{h})$. It is impractical to estimate mean costs over a multitude of trajectories in the application, as the amount of experiments and therefore the time-effort is too large. Instead, we define multiple target positions and use the controllers to drive to the target positions and back to the origin in a 20s cycle. The accumulated costs are evaluated on the measurement data. Let $\mathrm{mod}(a, b)$ denote the remainder and $\mathrm{div}(a, b)$ the quotient of the Euclidean division of $a$ and $b$. The target positions are defined as follows:

$$\tau = \mathrm{mod}(t, 20) \tag{5.15a}$$

$$i = \mathrm{div}(t, 20) \tag{5.15b}$$

$$\begin{bmatrix} x_r(t) \\ y_r(t) \\ \gamma_r(t) \end{bmatrix} = \begin{cases} \mathbf{0} & \text{if } \tau < 10 \text{ or } i > 9 \\ [x_i, y_i, \gamma_i]^T & \text{else} \end{cases}. \tag{5.15c}$$

with randomly sampled targets $[x_i, y_i, \gamma_i]$. The target positions and orientations are visualized in a 2D plane in figure 5.9 and are given as numerical

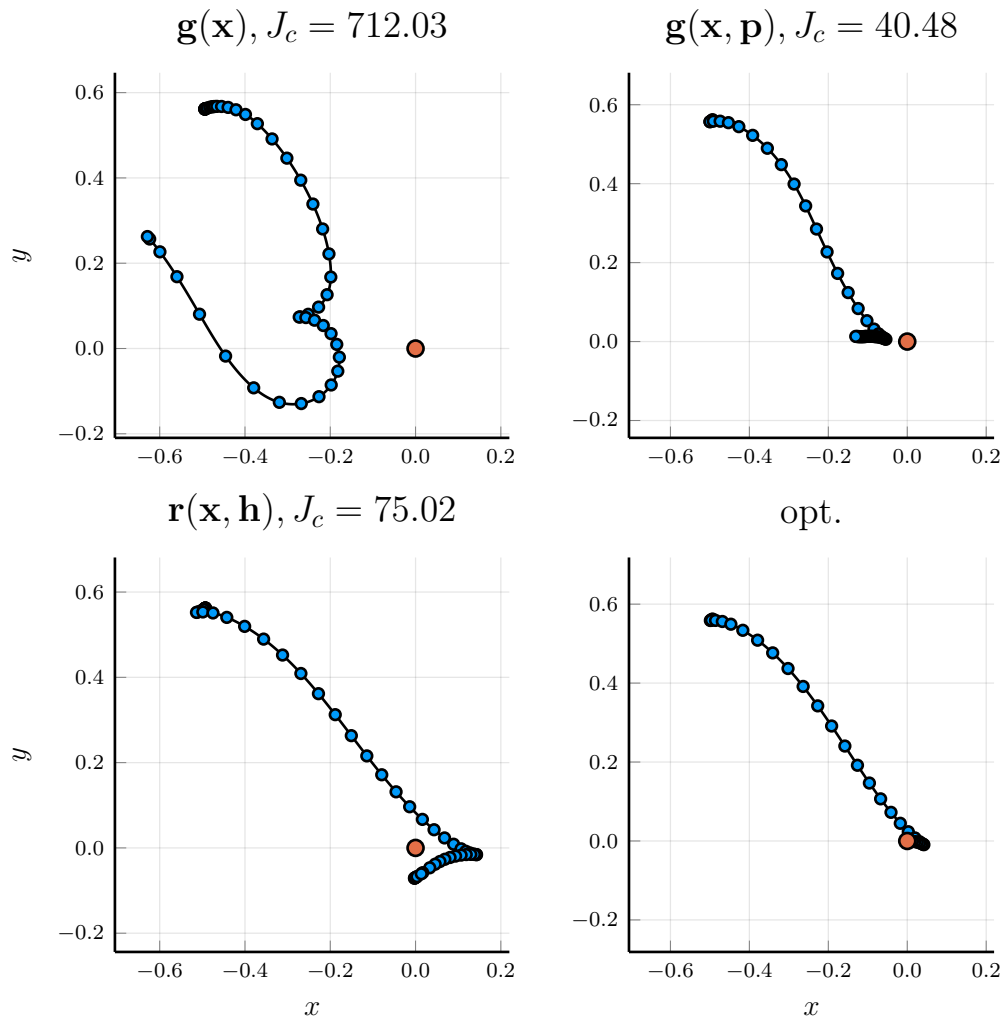



Figure 5.8: The position coordinates $(x_t, y_t)$ for simulations, starting at an initial state that produces high costs $J_c = \sum_{t=0}^{T-1} c(\mathbf{x}_t, \mathbf{u}_t) - c(\mathbf{x}_t^*, \mathbf{u}_t^*)$ for the controller $\mathbf{g}(\mathbf{x})$. One marker is placed every 0.1s of the trajectory.

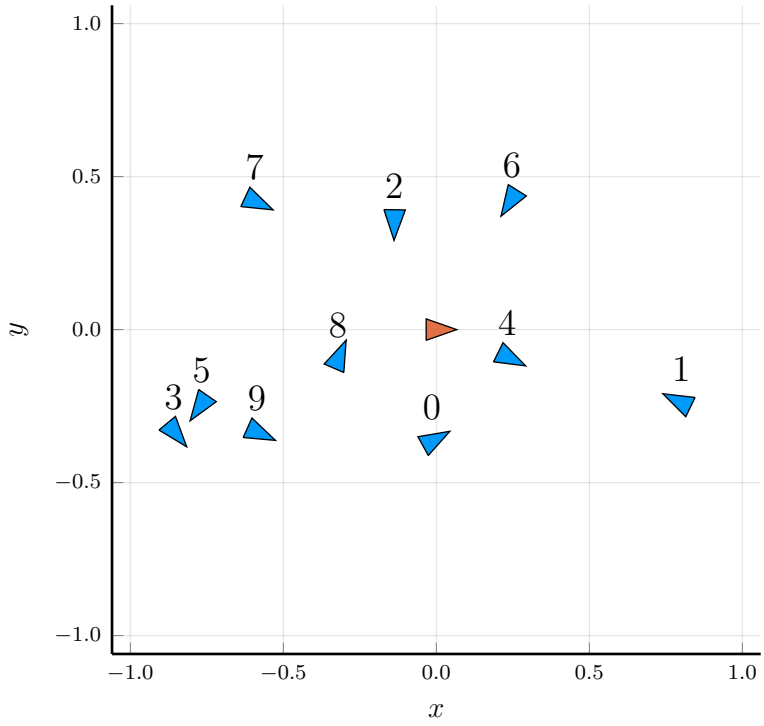



Figure 5.9: The target positions and orientations $(x, y, \gamma)$ for the evaluation of different controllers in the application. The origin is depicted as an orange arrow.

values in the appendix A.3. An experiment is carried out for each controller $\mathbf{g}(\mathbf{x})$, $\mathbf{r}(\mathbf{x}, \mathbf{h})$ and $\mathbf{r}(\mathbf{x}, \lambda, \mathbf{h})$ and the measurement data of the first 220s is saved. The cost function (5.5) is evaluated for the transformed coordinates $\tilde{\mathbf{x}}$ as described in section 5.2 in order to assign the lowest cost to the current target position $[x_i, y_i, \gamma_i]^T$. The obtained accumulated costs are given for each controller in table 5.4.

Table 5.4: Accumulated costs for different controllers on a test trajectory in the application.

| | $\mathbf{g}(\mathbf{x})$ | $\mathbf{r}(\mathbf{x}, \mathbf{h})$ | $\mathbf{r}_\lambda(\mathbf{x}, 0, \mathbf{h})$ | $\mathbf{r}_\lambda(\mathbf{x}, 0.3, \mathbf{h})$ |
|---|---|---|---|---|
| $\sum c_x$ | 5221.86 | 3957.26 | 4062.04 | 4131.76 |
| $\sum c_u$ | 1674.85 | 1566.88 | 1265.35 | 1431.55 |
| $\sum c$ | 6896.71 | 5524.14 | 5327.39 | 5563.31 |

The worst performance with respect to the accumulated costs is rendered by the static controller $\mathbf{g}(\mathbf{x})$ with a value of 6896.71. The recurrent controller $\mathbf{r}(\mathbf{x}, \mathbf{h})$ achieves a reduction of the costs of about 20% with a value of 5524.14 and the recurrent and adaptive controller with $\lambda$ shows similar results with a reduction of more than 22% and a value of 5327.39 . Adjusting $\lambda$ to a value of 0.3 leads to an increase in the accumulated costs, as is expected since the cost function that is used for the evaluation is only in accordance with the cost function used for training in the case of $\lambda = 0$.

The measurement data of the experiments is shown in figure 5.10 for the static controller, in figure 5.11 for the recurrent controller and in figure 5.12 for the adjustable recurrent controller with $\lambda = 0.3$. In the application, the target position is approached slower than in simulation, as the controllers have more difficulty to stabilize the tilt angle. This can be seen in small forward backward movements in all three figures in the position graph. These small movements, which are not measurement noise, appear when the system is close to the target position.

Comparing the static controller $\mathbf{g}(\mathbf{x})$ shown in figure 5.10 with the recurrent controller $\mathbf{r}(\mathbf{x}, \mathbf{h})$ in figure 5.11, the recurrent controller shows a smaller error on the target yaw angle. Moreover, it avoids spikes in the yaw angle that can be seen for the static controller at around second 72 and 192 of the trajectory. A position error remains for both of these controllers, and is also present for the adjustable controller with $\lambda = 0$, therefore we increase $\lambda$ to 0.3, resulting in a stronger importance on the target position, and less importance on the velocities. The results for this setting, shown in figure 5.12 reveal an improvement in the accuracy of the target position and a faster approach of the target position that is especially visible at seconds 30-40, 110-120 and 150-160 in the trajectory. The error in the yaw angle is also reduced as can be seen comparing the bottom plots of figures 5.11 and 5.12. Due to the higher velocities, however, the accumulated costs increase for $\lambda = 0.3$ compared to $\lambda = 0$ as can be seen in table 5.4. The state costs $\sum_t c_x(\mathbf{x}_t)$ as well as the total accumulated costs are the highest of all the recurrent controllers for $\lambda = 0.3$. Nevertheless, the behavior for $\lambda = 0.3$ is subjectively the most appealing for the task at hand. The behavior of the controller is shown qualitatively in figure 5.13 as well, and the behavior in simulation is added in figure 5.14 for

a comparison. Another qualitative impression of the control performance is given in a video at `https://youtu.be/MwVZgRJSnXg`.

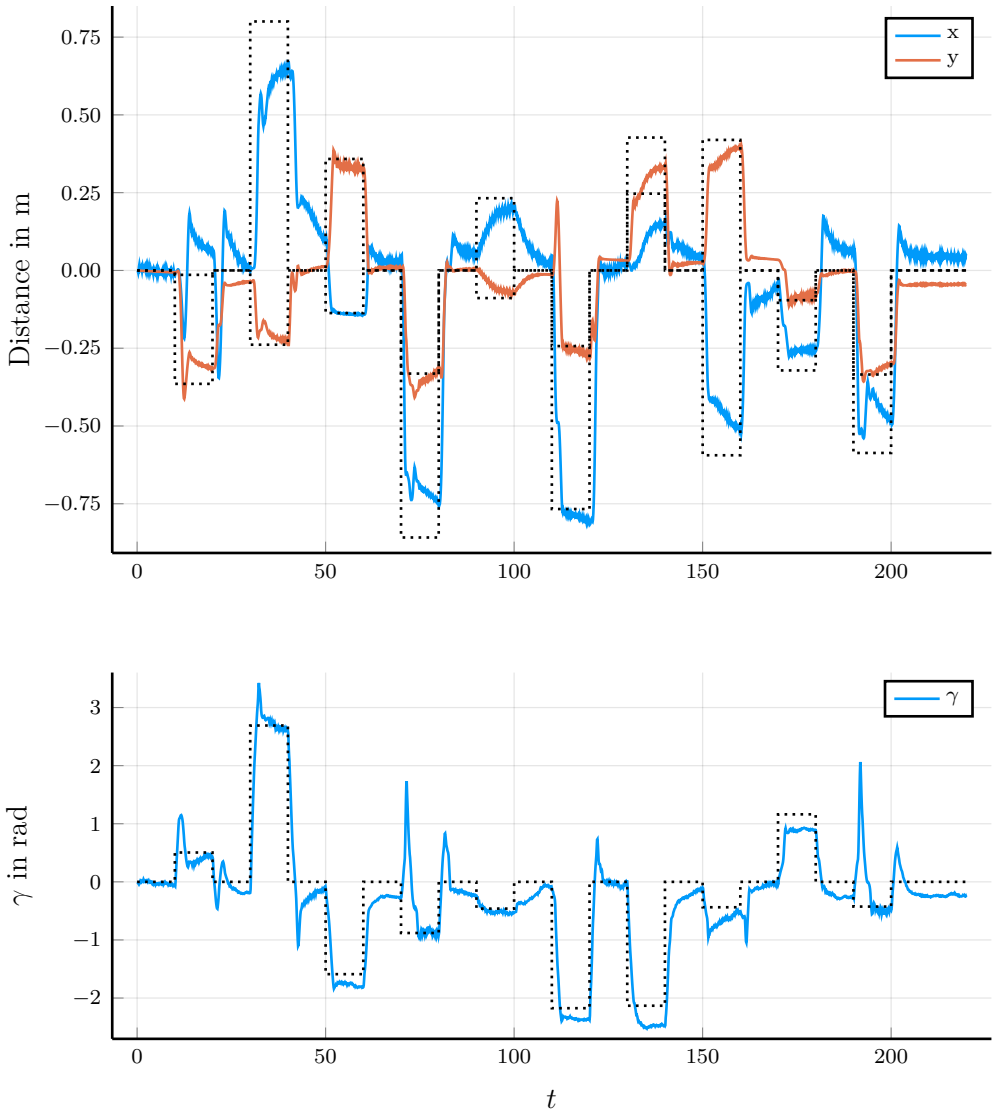



Figure 5.10: Measurement data for an application of a static neural network controller $\mathbf{g}(\mathbf{x})$. Units are in meters for the position coordinates $x$ and $y$ (top plot) and radian for $\gamma$ (bottom plot).

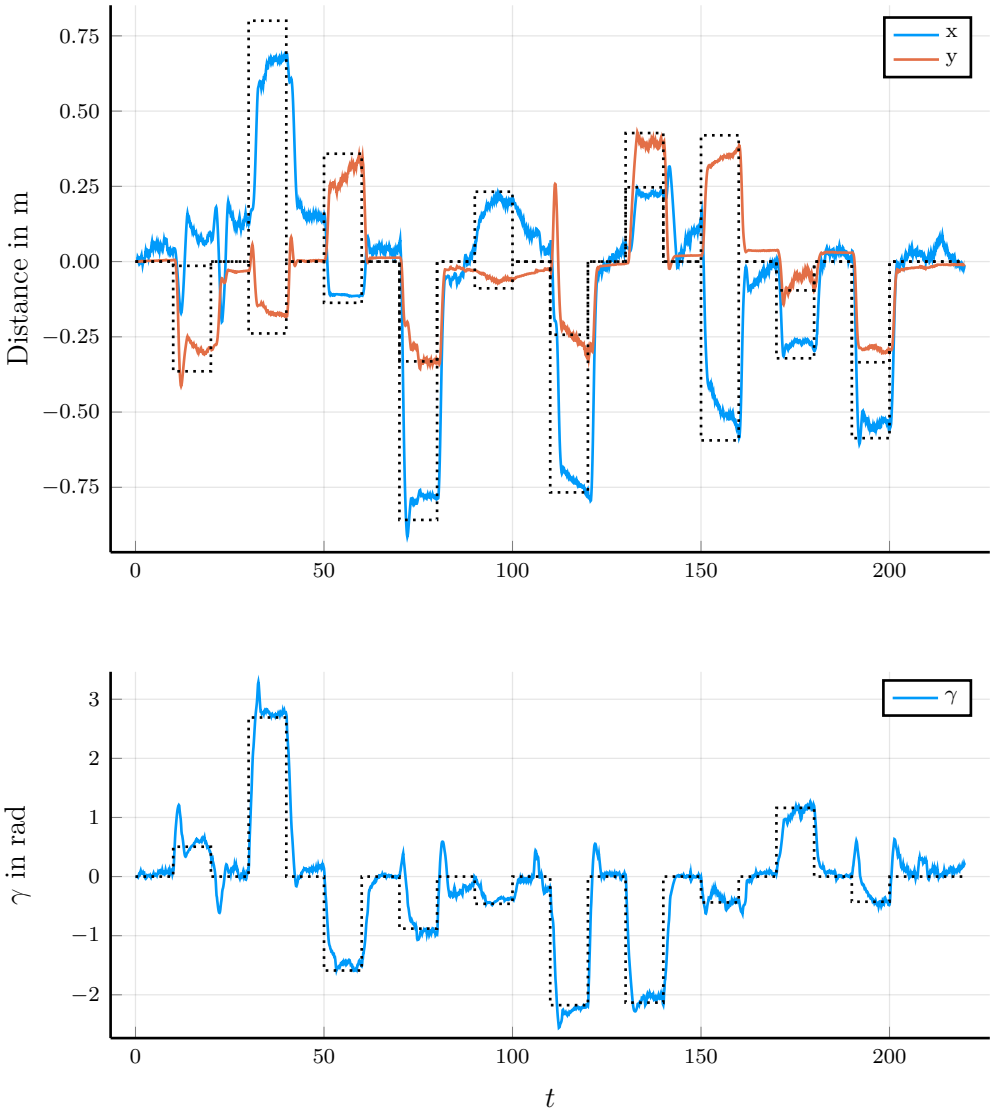



Figure 5.11: Measurement data for an application of a recurrent neural network controller $\mathbf{r}(\mathbf{x}, \mathbf{h})$. Units are in meters for the position coordinates $x$ and $y$ (top plot) and radian for $\gamma$ (bottom plot).

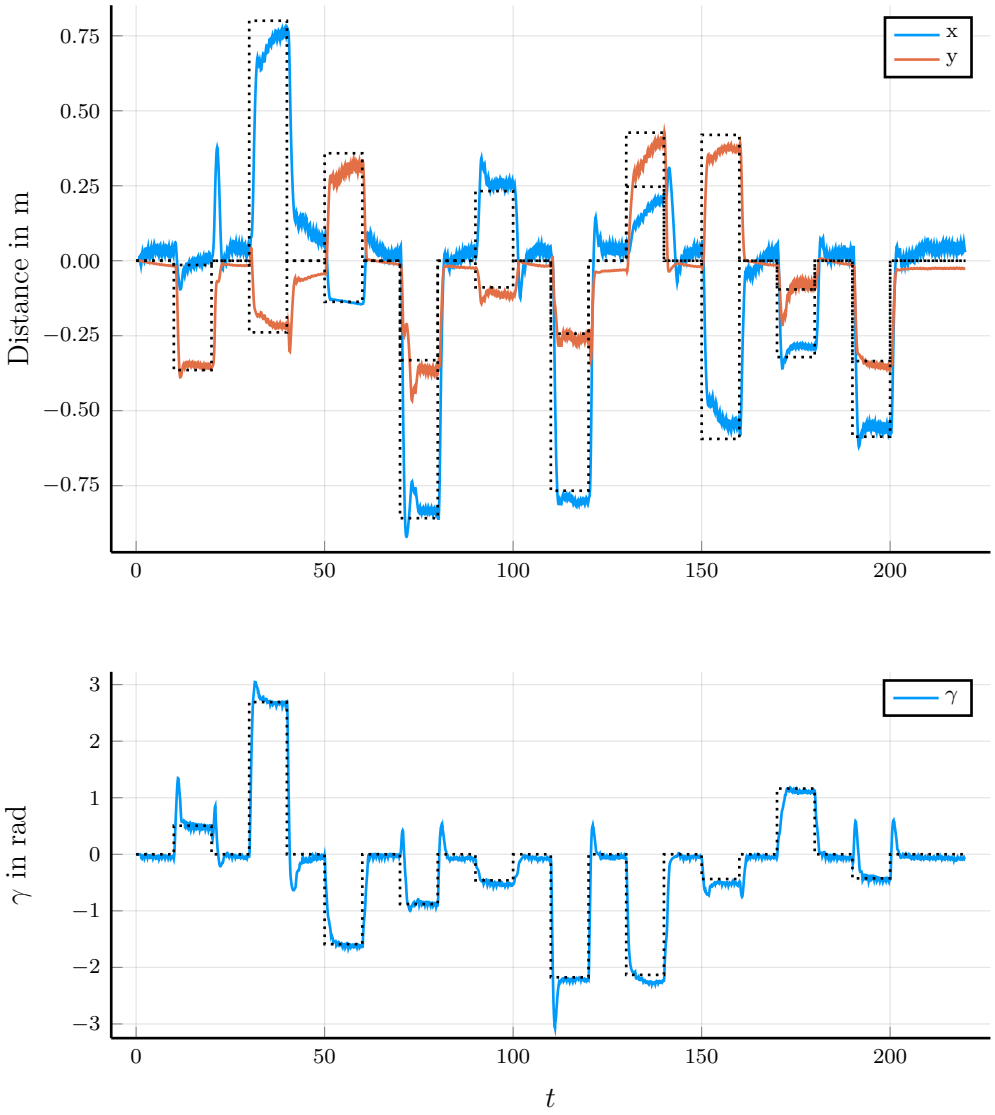



Figure 5.12: Measurement data for an application of an adjustable recurrent neural network controller $\mathbf{r}_\lambda(\mathbf{x}, \lambda, \mathbf{h})$ with $\lambda = 0.3$. Units are in meters for the position coordinates $x$ and $y$ (top plot) and radian for $\gamma$ (bottom plot).

## 5.5 Further applications

So far, we trained controllers to drive the MIP to specific locations. Another common application for mobile robots is to follow a moving target or move on

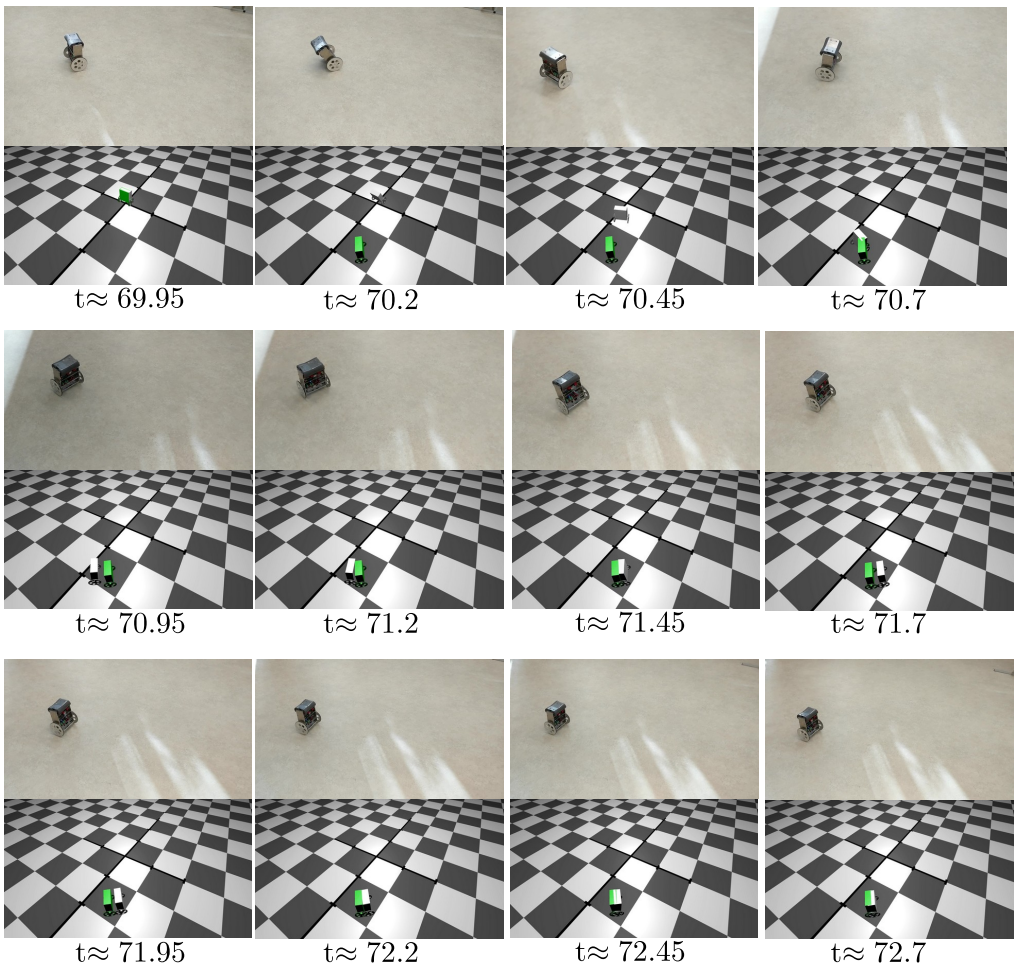



Figure 5.13: Qualitative visualization of a maneuver on the test trajectory of the robot in the application.

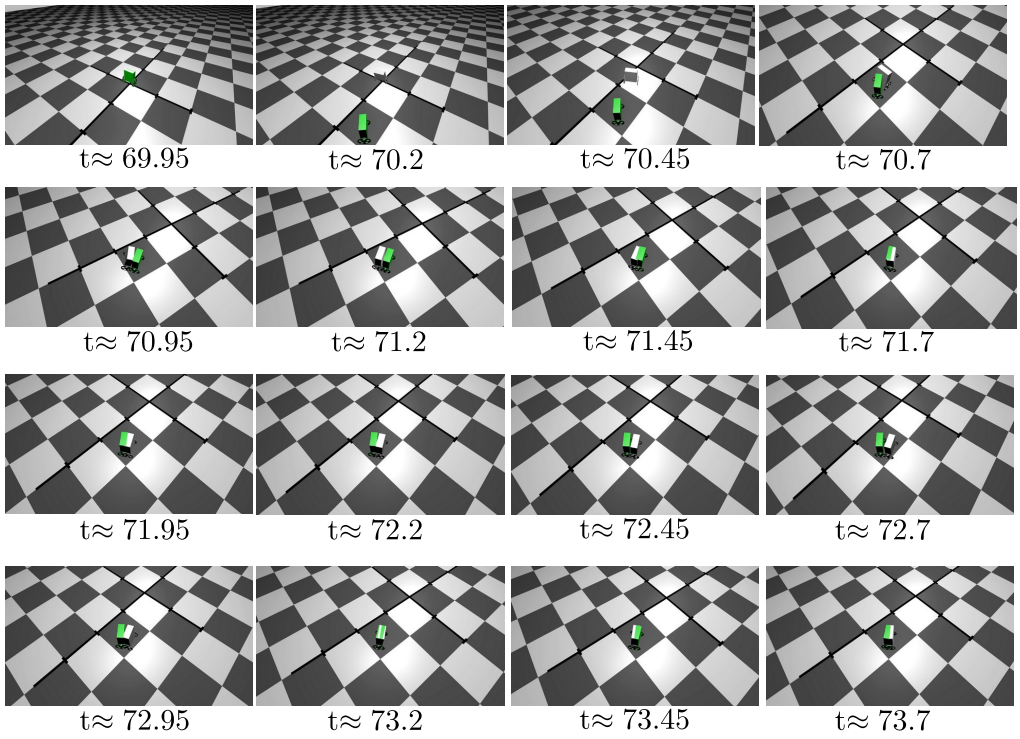



Figure 5.14: Qualitative visualization of a maneuver on the test trajectory of the robot in simulation.

a predefined trajectory. The presented controllers are able to follow a moving target by continuously shifting the target position. However, the system is driving behind the target within a certain distance as is seen in a video at `https://youtu.be/MwVZgRJSnXg`. The controllers are only trained to move to a target with zero velocity. To close the distance between the system and the target, training data including a moving target has to be included. For other tasks, e.g., ascending slopes or avoiding collisions, new training data and therefore new trajectories for that task is required as well.

# Chapter 6

# Discussion

This chapter discusses the results of the last two chapters and clarifies the advantages and disadvantages of the method presented in chapter 3.

Chapter 4 demonstrates through an empirical analysis that trajectory optimization followed by supervised learning and DOI can lead to a superior control performance compared to controllers that are trained using popular statistical methods like reinforcement learning. The statistical methods optimize the accumulated costs of the trajectories in a direct manner, which, in theory, allows them to converge to a local minimum. Due to their statistical nature, reaching a local minimum generally requires a lot of samples and learning can be slow. In practice, the algorithms don't fully converge to the local minimum unless a lot of time is spent tuning hyper-parameters.

Trajectory optimization with supervised learning on the other side indirectly trains a controller based on demonstrations. Therefore, no theoretical guarantees for the closed-loop performance can be provided. However, the results in this thesis, as well as previous works mentioned in section 2.3.3 that use similar methods, provide empirical evidence that the method can be used successfully to design nonlinear parameterized controllers. The control design using our method requires trajectory optimization and supervised learning, which are both established methods that generally provide good results without requiring a lot of hand tuning. Moreover, the method is divided into multiple successive steps, which adds transparency into the training. It is easier to identify mistakes during the process and a preview of the control

performance is provided after each step. Therefore, we believe that trajectory optimization with supervised learning is currently the method that should be used to train parameterized controllers if there is low tolerance for trial and error.

Moreover, the different methods to train parameterized controllers have different requirements on the problem setting. One advantage of trajectory optimization with imitation learning is that the behavior can be defined using equality constraints. This may lead to ambiguous training data, as is mentioned in section 3.2.1. On the other hand, including equality constraints can be necessary to achieve convergence, as was the case, e.g., for the presented wheeled inverted pendulum problem. A disadvantage of the method is that trajectory optimization requires the model and cost function to be differentiable, which is not required for the statistical methods.

In the previous two chapters, an adaptive controller trained using DOI, was compared amongst others against a static controller trained using behavioral cloning. Chapter 4 shows that the adaptive controller performs better in average as well as in the worst case if the model parameters are uncertain. Chapter 5 provides an application example for which the adaptive controller also shows better performance than the static controller. DOI provides an extension to the behavioral cloning approach that allows an adaptation of the controller to the physical system and can be used, as is done in this thesis, to provide robustness.

The presented extension to make the controller adjustable is aiming towards the application outside of a simulation environment. It adds to the overall package of a method that is suitable for practical applications.

Chapter 7

# Conclusion and outlook

This thesis deals with the design and application of control laws in the form of nonlinear function approximators. As of writing this thesis, most published research focuses on the field of reinforcement learning to design such controllers. This thesis investigates a different approach, using optimized trajectories and supervised learning. Previous approaches based on those methods were limited to static function approximators.

The main contribution of this thesis is a novel approach to train function approximators with internal state variables for the design of feedback controllers. The internal states allow the controller to adapt to its environment. In this thesis, we use the adaptability to deal with uncertain parameters of the controlled plant. The method is based on the idea of a teaching and a learning agent, found in imitation learning. Both agents are function approximators. The teaching agent has access to the full information that is required to perform optimal actions, whereas the learning agent is trained to perform similar actions without access to the full information, relying on its internal state instead. The method, called disturbed oracle imitation (DOI), is empirically investigated using the example of an overhead crane in simulation and finally used in the practical application for a mobile inverted pendulum.

The empirical evaluations for the simulation example show that the method can outperform algorithms from the fields of reinforcement learning and evolution strategies with regard to the achieved control performance under un-

certain model parameters. The adaptive controller trained using DOI also outperforms a static controller, trained using trajectory optimization and imitation learning, indicating that adding hidden states increased the controller's robustness.

In the practical application on the mobile inverted pendulum, the adaptive controller trained using our method achieved a better control performance than a static controller trained using a similar method but without DOI. An advantage of our method, that results from using trajectory optimization as an interim step is that equality constraints can easily be included in the optimization to define the behavior of the system. Including equality constraints is not possible with statistical methods like reinforcement learning or evolution strategies. None of the tested statistical methods converged for the mobile inverted pendulum. Furthermore, the new method is less sensitive to changes in the algorithm's hyper-parameters, making it more suitable for industrial applications.

Additionally, we introduced an extension of the method to allow manual adjustments to the behavior after training the controller. Controllers with a high number of parameters, as is the case for function approximators, usually do not allow simple adjustments after training. We introduce an additional, scalar control input that can be used to manually modify the behavior of the controller during application. The additional input is included in the design of the adaptive controller for the mobile inverted pendulum. The input is adjusted to subjectively enhance the control behavior.

A drawback of the method is that the training using supervised learning does not guarantee convergence to a local minimum for the accumulated costs in the closed-loop. While the method worked reliably in all of our applications, a theoretical foundation would increase the acceptance of the method and could be the topic of further research.

In classic adaptive control, design methods have been coupled with robust control theory to add robustness against unmodeled uncertainties to the adaptive controller, e.g., [54]. Pursuing similar ideas for the use with function approximators could further improve the control performance.

Appendix A

# Additional resources for chapter 5

## A.1 Rigid body dynamics model for the mobile inverted pendulum

The rigid body dynamics model of a MIP that is derived in Pathak et al. [82] is repeated in the following. The input is a vector of torques applied to the right and left wheels $\boldsymbol{\tau} = [\tau_r, \tau_l]$. The model parameters with description are depicted in table 5.1. The state space model is

$$\dot{x} = \cos(\alpha) \cdot v \tag{A.1a}$$
$$\dot{y} = \sin(\alpha) \cdot v \tag{A.1b}$$
$$\dot{\gamma} = \dot{\gamma} \tag{A.1c}$$
$$\dot{\alpha} = \dot{\alpha} \tag{A.1d}$$
$$\ddot{\alpha} = f_5(\mathbf{x}) + g_5(\mathbf{x})(\tau_r + \tau_l) \tag{A.1e}$$
$$\dot{v} = f_6(\mathbf{x}) + g_6(\mathbf{x})(\tau_r + \tau_l) \tag{A.1f}$$
$$\ddot{\gamma} = f_7(\mathbf{x}) + g_7(\mathbf{x})(\tau_r - \tau_l) \tag{A.1g}$$

with the abbreviations

$$\begin{aligned} D(\alpha) =& \left((-M_b^2 - 2M_wM_b)c_z^2 - 2I_{yy}M_w - Iyy M_b\right) R^2 \\ &+ M_b^2\cos^2(\alpha)c_zR^2 - 2M_bc_z^2I_{wa} - 2I_{yy}I_{wa} \end{aligned} \tag{A.2a}$$

$$\begin{aligned} G(\alpha) =& (M_bc_z^2 + I_{xx} + 2I_{wd} + 2b^2M_w)R^2 + 2b^2I_{wa} \\ &+ (-M_bc_z^2 + I_{zz} - I_{xx})R^2\cos^2(\alpha) \end{aligned} \tag{A.2b}$$

$$\begin{aligned} K(\alpha) =& M_bR^2c_z\sin(\alpha)\left(-4I_{yy} - 3M_bc_z^2 + I_{xx} - I_{zz}\right) \\ &+ M_bR^2c_z\sin(3\alpha)\left(I_{xx} - I_{zz} + M_bc_z^2\right) \end{aligned} \tag{A.2c}$$

$$\begin{aligned} H =& \left(\frac{1}{2}M_bR^2 + M_wR^2 + I_{wa}\right)(I_{zz} - I_{xx}) \\ &- M_bc_z^2(M_wR^2 + I_{wa}) \end{aligned} \tag{A.2d}$$

$$\begin{aligned} f_5(\mathbf{x}) =& \frac{\sin(2\alpha)\dot{\gamma}^2H - M_bc_zg\sin(\alpha)\left(M_bR^2 + 2(I_{wa} + M_wR^2)\right)}{D(\alpha)} \\ &+ \frac{M_b^2c_z^2R^2\sin(2\alpha)\dot{\alpha}^2}{2D(\alpha)} \end{aligned} \tag{A.3a}$$

$$\begin{aligned} f_6(\mathbf{x}) =& K(\alpha)\dot{\gamma}^2 \\ &+ \frac{M_b^2c_z^2R^2g\sin(2\alpha) - 2\sin(\alpha)\dot{\alpha}^2M_bR^2c_z\left(I_{yy} + M_bc_z^2\right)}{2D(\alpha)} \end{aligned} \tag{A.3b}$$

$$f_7(\mathbf{x}) = \frac{-R^2\dot{\gamma}}{G(\alpha)}\left(\sin(2\alpha)\dot{\alpha}(I_{xx} - I_{zz} + M_bc_z^2) + \sin(\alpha)M_bc_zv\right) \tag{A.3c}$$

$$g_5(\mathbf{x}) = \frac{M_bR^2 + 2M_wR^2 + 2I_{wa} + M_b\cos(\alpha)c_zR}{D(\alpha)} \tag{A.4a}$$

$$g_6(\mathbf{x}) = -\frac{R\left(M_b\cos(\alpha)c_zR + I_{yy} + M_bc_z^2\right)}{D(\alpha)} \tag{A.4b}$$

$$g_7(\mathbf{x}) = \frac{Rb}{G(\alpha)}. \tag{A.4c}$$

## A.2 Simulation plots for high cost trajectories

Figures A.1 and A.2 show the simulated trajectories that provide the worst above optimal costs $J_{\max,c}$ for the controllers $\mathbf{g}(\mathbf{x}, \mathbf{p})$ and $\mathbf{r}(\mathbf{x}, \mathbf{h})$ respectively.

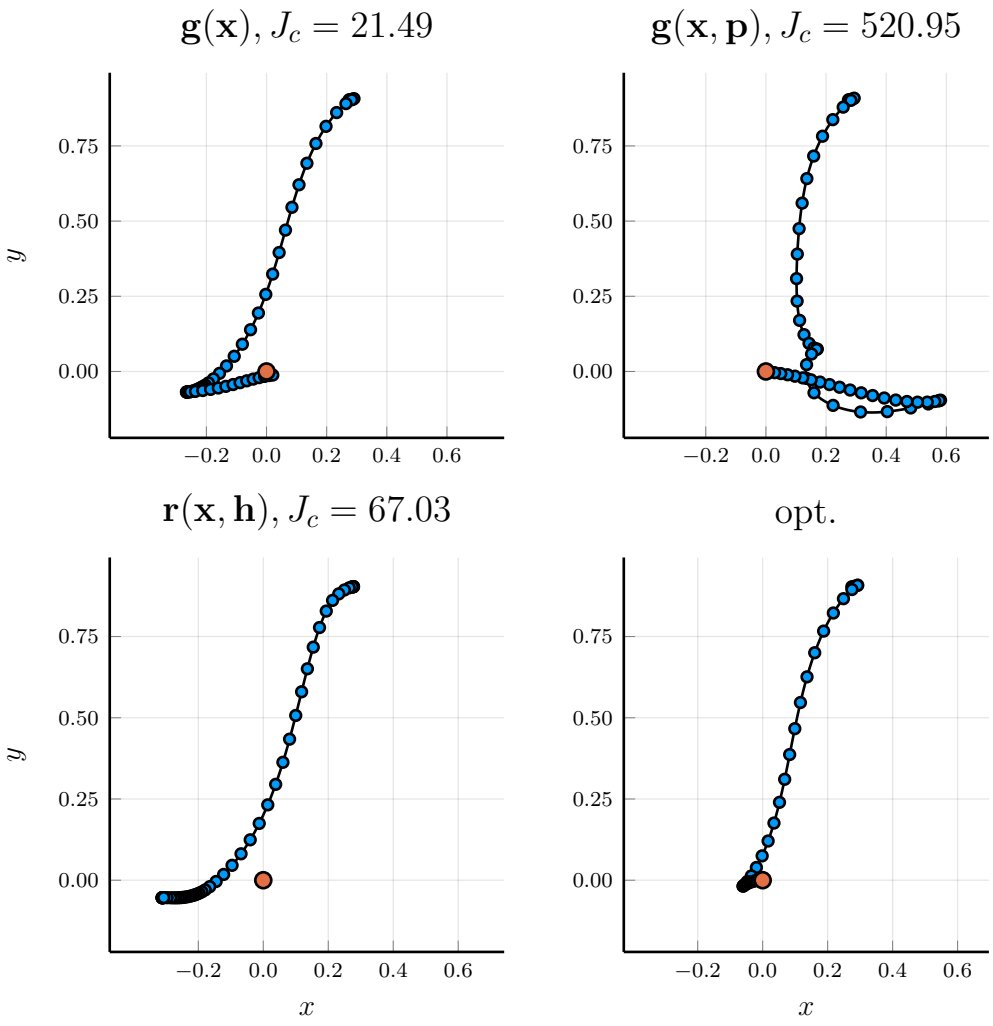



Figure A.1: The position coordinates $(x_t, y_t)$ for simulations, starting at an initial state that produces high costs for the controller $\mathbf{g}(\mathbf{x}, \mathbf{p})$. One marker is placed every 0.1s of the trajectory.

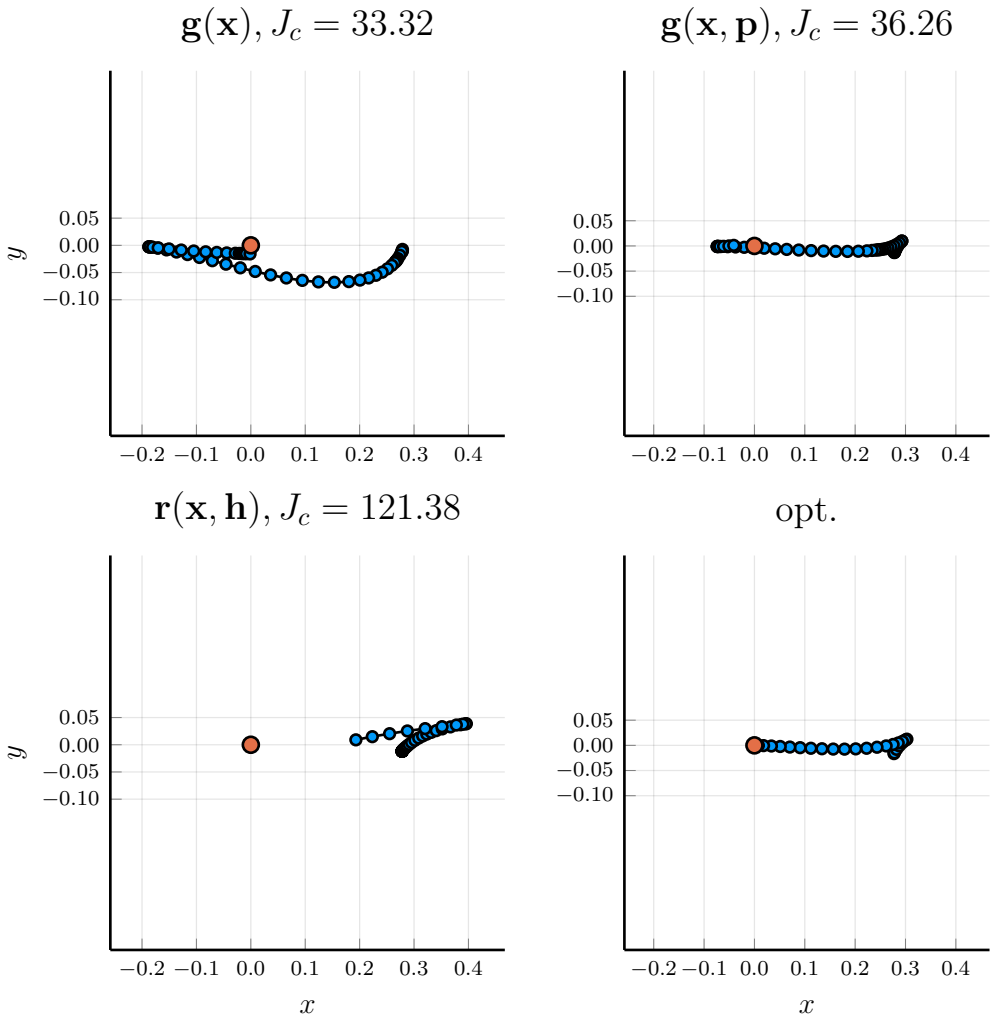



Figure A.2: The position coordinates $(x_t, y_t)$ for simulations, starting at an initial state that produces high costs for the controller $\mathbf{r}(\mathbf{x}, \mathbf{h})$. One marker is placed every 0.1s of the trajectory.

## A.3 Target positions for the application comparison

The target positions for the application, used in section 5.4.2. The values were generated randomly and rare displayed rounded to the 3rd decimal place.

$$
\begin{aligned}
[x_0, y_0, \gamma_0] &= [-0.014, -0.365, 0.505] \\
[x_1, y_1, \gamma_1] &= [0.800, 0.239, 2.691] \\
[x_2, y_2, \gamma_2] &= [-0.137, 0.358, -1.589] \\
[x_3, y_3, \gamma_3] &= [-0.858, -0.332, -0.880] \\
[x_4, y_4, \gamma_4] &= [0.232, -0.089, -0.461] \\
[x_5, y_5, \gamma_5] &= [-0.767, -0.243, -2.175] \\
[x_6, y_6, \gamma_6] &= [0.247, 0.427, -2.132] \\
[x_7, y_7, \gamma_7] &= [-0.594, 0.420, -0.437] \\
[x_8, y_8, \gamma_8] &= [-0.321, -0.096, 1.162] \\
[x_9, y_9, \gamma_9] &= [-0.587, -0.335, -0.426]
\end{aligned}
$$

# Bibliography


[1] Pieter Abbeel, Morgan Quigley, and Andrew Y Ng. Using inaccurate models in reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 1–8. ACM, 2006.

[2] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems 19*, pages 1–8. MIT Press, 2007.

[3] Bernt M Åkesson and Hannu T Toivonen. A neural network model predictive controller. *Journal of Process Control*, 16(9):937–946, 2006.

[4] Bernt M Åkesson, Hannu T Toivonen, Jonas B Waller, and Rasmus H Nyström. Neural network approximation of a nonlinear model predictive controller applied to a ph neutralization process. *Computers & chemical engineering*, 29(2):323–335, 2005.

[5] Kiam Heong Ang, Gregory Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE transactions on control systems technology*, 13(4):559–576, 2005.

[6] Brian Armstrong-Hélouvry, Pierre Dupont, and Carlos Canudas De Wit. A survey of models, analysis tools and compensation methods for the control of machines with friction. *Automatica*, 30(7):1083–1138, 1994.

[7] Karl J. Aström and Björn Wittenmark. *Adaptive control*. Dover Publications, Mineola, N.Y, 2008. ISBN 9780486462783.

[8] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012.

[9] Charalampos P Bechlioulis and George A Rovithakis. Robust adaptive control of feedback linearizable mimo nonlinear systems with prescribed performance. *IEEE Transactions on Automatic Control*, 53(9):2090–2099, 2008.

[10] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[11] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1): 65–98, 2017.

[12] Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4243–4250. IEEE, 2018.

[13] George EP Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

[14] Dimo Brockhoff, Anne Auger, Nikolaus Hansen, Dirk V Arnold, and Tim Hohm. Mirrored sampling and sequential selection for evolution strategies. In *International Conference on Parallel Problem Solving from Nature*, pages 11–21. Springer, 2010.

[15] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019*

*International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

[16] Lu Chi and Yadong Mu. Deep steering: Learning end-to-end driving model from spatial and temporal visual cues. *arXiv preprint arXiv:1708.03798*, 2017.

[17] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

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

[19] Christian Dengler and Boris Lohmann. Multi-Modell-Ansatz für die Nachjustierung von Black–Box–Reglern in der Praxis. *Proceedings. 29. Workshop Computational Intelligence*, pages 217–236, November 2019.

[20] Christian Dengler and Boris Lohmann. Adjustable and adaptive control for an unstable mobile robot using imitation learning with trajectory optimization. *Robotics*, 9(2):29, Apr 2020.

[21] Ronnie Dessort and C. Chucholowski. Explicit model predictive control of semi-active suspension systems using artificial neural networks (ann). In Prof. Dr. Peter E. Pfeffer, editor, *8th International Munich Chassis Symposium 2017*, pages 207–228, Wiesbaden, 2017. Springer Fachmedien Wiesbaden. ISBN 978-3-658-18459-9.

[22] Rahul Dey and Fathi M Salemt. Gate-variants of gated recurrent unit (gru) neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1597–1600. IEEE, 2017.

[23] Moritz Diehl, Hans Joachim Ferreau, and Niels Haverbeke. Efficient numerical methods for nonlinear mpc and moving horizon estimation. In *Nonlinear model predictive control*, pages 391–417. Springer, 2009.

[24] N. Dini and V. J. Majd. Model predictive control of a wheeled inverted pendulum robot. In *2015 3rd RSI International Conference on Robotics and Mechatronics (ICROM)*, pages 152–157, Oct 2015. doi: 10.1109/ICRoM.2015.7367776.

[25] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778, 2016. URL `http://arxiv.org/abs/1604.06778`.

[26] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[27] Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *arXiv:1508.01982 [math.OC]*, 2015. URL `http://arxiv.org/abs/1508.01982`.

[28] Eberly College of Science, PennState. Sums of Independent Normal Random Variables. `https://online.stat.psu.edu/stat414/node/172/`. Accessed: 02.03.2020.

[29] Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. In *International Conference on Machine Learning*, pages 49–58, 2016.

[30] Nobusumi Fukushima, Yuichi Nagata, Sigenobu Kobayashi, and Isao Ono. Proposal of distance-weighted exponential natural evolution strategies. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 164–171. IEEE, 2011.

[31] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[32] Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976,

Mumbai, India, December 2012. The COLING 2012 Organizing Committee. URL https://www.aclweb.org/anthology/C12-1059.

[33] Florian Golemo, Adrien Ali Taiga, Aaron Courville, and Pierre-Yves Oudeyer. Sim-to-real transfer with neural-augmented robot simulation. In Aude Billard, Anca Dragan, Jan Peters, and Jun Morimoto, editors, *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 817–828, Zurich, Switzerland, 29–31 Oct 2018. PMLR. URL http://proceedings.mlr.press/v87/golemo18a.html.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL http://www.deeplearningbook.org.

[35] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International conference on machine learning*, pages 1764–1772, 2014.

[36] Nikolaus Hansen. The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.

[37] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *CoRR*, abs/1604.00772, 2016. URL http://arxiv.org/abs/1604.00772.

[38] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2): 159–195, 2001.

[39] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11 (1):1–18, 2003.

[40] Nicolas Heess, Jonathan J. Hunt, Timothy P. Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *CoRR*, abs/1512.04455, 2015. URL http://arxiv.org/abs/1512.04455.

[41] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[42] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[44] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):21, 2017.

[45] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[46] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[47] Gregory Kahn, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. PLATO: policy learning using adaptive trajectory optimization. *CoRR*, abs/1603.00622, 2016. URL `http://arxiv.org/abs/1603.00622`.

[48] Tolgay Kara and Ilyas Eker. Nonlinear modeling and identification of a dc motor for bidirectional operation with real time experiments. *Energy Conversion and Management*, 45(7-8):1087–1106, 2004.

[49] Florian Karg. *Erlernen eines robusten Reglers für ein schwingendes Pendel über EPOpt*. Term paper, Technical University of Munich, 2020.

[50] Denis Kartachov. Pendulum-cart system – analysis of the equations of motion, May 2016. Accessed 27.04.2020.

[51] Yeonhoon Kim, Soo Hyun Kim, and Yoon Keun Kwak. Dynamic analysis of a nonholonomic two-wheeled inverted pendulum robot. *Journal of Intelligent and Robotic Systems*, 44(1):25–46, 2005.

[52] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

[53] Nick Kollerstrom. Thomas Simpson and Newton's method of approximation: an enduring myth. *The British journal for the history of science*, 25(3):347–354, 1992.

[54] Gerhard Kreisselmeier and Brian Anderson. Robust model reference adaptive control. *IEEE Transactions on Automatic Control*, 31(2): 127–133, 1986.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[56] Julia Krottenthaler. Implementierung und Evaluation eines Aktor-Kritiker-Algorithmus. Bachelor's thesis, Technical University of Munich, September 2017.

[57] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[58] ID Landau. From robust control to adaptive control. *Control Engineering Practice*, 7(9):1113–1124, 1999.

[59] Michael Laskey, Jonathan Lee, Roy Fox, Anca Dragan, and Ken Goldberg. Dart: Noise injection for robust imitation learning. *arXiv preprint arXiv:1703.09327*, 2017.

[60] M Lazar, D Muñoz De La Peña, WPMH Heemels, and T Alamo. Min-max nonlinear model predictive control with guaranteed input-tostate

stability. In *17th Symposium on Mathematical Theory for Networks and Systems. Kyoto, Japan*, 2006.

[61] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, volume 98. Siam, 2007.

[62] Sergey Levine and Vladlen Koltun. Guided policy search. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1–9, 2013.

[63] Sergey Levine and Vladlen Koltun. Learning complex neural network policies with trajectory optimization. In *International Conference on Machine Learning*, pages 829–837, 2014.

[64] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[65] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL `http://arxiv.org/abs/1509.02971`.

[66] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[67] Igor Mordatch and Emo Todorov. Combining the benefits of function approximation and trajectory optimization. In *Robotics: Science and Systems*, pages 5–32, 2014.

[68] Igor Mordatch, Kendall Lowrey, Galen Andrew, Zoran Popovic, and Emanuel V Todorov. Interactive control of diverse complex characters with neural networks. In *Advances in Neural Information Processing Systems*, pages 3132–3140, 2015.

[69] Igor Mordatch, Kendall Lowrey, and Emanuel Todorov. Ensemble-cio: Full-body dynamic motion planning that transfers to physical humanoids. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5307–5314. IEEE, 2015.

[70] David Morin. *Introduction to Classical Mechanics*. Cambridge University Pr., 2008. ISBN 0521876222. URL `https://www.ebook.de/de/product/6801020/david_morin_introduction_to_classical_mechanics.html`.

[71] V. Muralidharan and A. D. Mahindrakar. Position stabilization and waypoint tracking control of mobile inverted pendulum robot. *IEEE Transactions on Control Systems Technology*, 22(6):2360–2367, Nov 2014. ISSN 2374-0159. doi: 10.1109/TCST.2014.2300171.

[72] Fabio Muratore, Felix Treede, Michael Gienger, and Jan Peters. Domain randomization for simulation-based policy optimization with transferability assessment. In Aude Billard, Anca Dragan, Jan Peters, and Jun Morimoto, editors, *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 700–713. PMLR, 29–31 Oct 2018. URL `http://proceedings.mlr.press/v87/muratore18a.html`.

[73] Fabio Muratore, Michael Gienger, and Jan Peters. Assessing transferability from simulation to reality for reinforcement learning. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[74] Daniel Murdock. *Modeling, identification and control of a wheeled balancing system*. PhD thesis, Georgia Institute of Technology, 2016.

[75] Kevin P. Murphy. *Machine Learning*. MIT Press Ltd, 2012. ISBN 0262018020. URL `https://www.ebook.de/de/product/19071158/kevin_p_murphy_machine_learning.html`.

[76] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[77] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[78] Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, London, 5th edition, 2010. ISBN 978-0-136-15673-4.

[79] J Gomez Ortega and EF Camacho. Mobile robot navigation in a partially structured static environment, using neural predictive control. *Control Engineering Practice*, 4(12):1669–1679, 1996.

[80] Markos Papageorgiou, Marion Leibold, and Martin Buss. *Optimierung: Statische, dynamische, stochastische Verfahren für die Anwendung*. Springer-Verlag, 2012. ISBN 978–3–540–34012–6.

[81] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[82] Kaustubh Pathak, Jaume Franch, and Sunil Kumar Agrawal. Velocity and position control of a wheeled inverted pendulum by partial feedback linearization. *IEEE Transactions on robotics*, 21(3):505–513, 2005.

[83] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary. Robust deep reinforcement learning with adversarial attacks. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2040–2042, Stockholm, Sweden, 10–15 July 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[84] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.

[85] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. *arXiv preprint arXiv:1703.02702*, 2017.

[86] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[87] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[88] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[89] Aravind Rajeswaran, Sarvjeet Ghotra, Balaraman Ravindran, and Sergey Levine. Epopt: Learning robust neural network policies using model ensembles. *ICLR*, 2017.

[90] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*, 2015.

[91] Sheldon Ross. *Introduction to probability models*. Academic Press, San Diego, CA, 2003. ISBN 9780124079489.

[92] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[93] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[94] Thomas Schlegl, Philipp Seeböck, Sebastian M Waldstein, Georg Langs, and Ursula Schmidt-Erfurth. f-anogan: Fast unsupervised anomaly detection with generative adversarial networks. *Medical image analysis*, 54:30–44, 2019.

[95] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[96] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

[97] John Schulman. *Optimizing expectations: from deep reinforcement learning to stochastic computation graphs*. PhD thesis, University of California, Berkeley, 2016.

[98] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL `http://arxiv.org/abs/1502.05477`.

[99] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

[100] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, Beijing, China, June 2014.

[101] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529 (7587):484, 2016.

[102] Corentin Tallec and Yann Ollivier. Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*, 2017.

[103] Francois Thibeau. Model Extension and Identification of a Chain attached to a Cart. Bachelor's thesis, Technical University of Munich, October 2018.

[104] Christophe Vogel. Black-Box Optimization as an Alternative to an Analytical Controller Design. Bachelor thesis, Technical University of Munich, September 2018.

[105] Andreas Wächter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering.* PhD thesis, Carnegie Mellon University, 2002.

[106] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006.

[107] Siri Weerasooriya and Mohamad A El-Sharkawi. Identification and control of a dc motor using back-propagation neural networks. *IEEE transactions on Energy Conversion*, 6(4):663–669, 1991.

[108] Darrell Whitley and Andrew M. Sutton. *Genetic Algorithms — A Survey of Models and Methods*, pages 637–671. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-540-92910-9. doi: 10.1007/978-3-540-92910-9_21.

[109] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008.

[110] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[111] Margaret H Wright. Why a pure primal newton barrier step may be infeasible. *SIAM Journal on Optimization*, 5(1):1–12, 1995.

[112] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005.

[113] Kazuo Yamafuji and Takashi Kawamura. Postural control of a monoaxial bicycle. *Journal of the Robotics Society of Japan*, 7(4):74–79, August 1988.

[114] Wenhao Yu, C. Karen Liu, and Greg Turk. Preparing for the unknown: Learning a universal policy with online system identification. *CoRR*, abs/1702.02453, 2017. URL `http://arxiv.org/abs/1702.02453`.

[115] Deniz Yuret. Knet: beginning deep learning with 100 lines of julia. In *Machine Learning Systems Workshop at NIPS 2016*, 2016.

[116] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[117] Eric Zhan, Stephan Zheng, Yisong Yue, Long Sha, and Patrick Lucey. Generative multi-agent behavioral cloning. *arXiv*, 2018.

[118] Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Russ R Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. In *Advances in neural information processing systems*, pages 1822–1830, 2016.

[119] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 528–535. IEEE, 2016.