Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Yngve Kippersund

A Recurrent Neural Network-Based Model Predictive Controller

Master's thesis in Cybernetics and Robotics
Supervisor: Lars Struen Imsland
Co-supervisor: John-Morten Godhavn
June 2023

NTNU
Norwegian University of
Science and Technology

Yngve Kippersund

A Recurrent Neural Network-Based Model Predictive Controller

Master's thesis in Cybernetics and Robotics
Supervisor: Lars Struen Imsland
Co-supervisor: John-Morten Godhavn
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Abstract

Effective and accurate process control for gas and oil extraction is essential for the safety of the rig crew, for minimizing environmental consequences from extraction as well as for maximizing profits of operation. Traditional control methods include model predictive control, and many applications use linear models. By instead utilizing neural networks' vast modelling capabilities, an increase in control performance may be gained if the process is inherently nonlinear. This thesis derives a mathematical formulation for one such model predictive controller. A data set is then synthesized, on which a neural network-based system dynamics model is trained. The neural network is then embedded into a self-developed model predictive controller framework. The resulting application proves, as a proof-of-concept, the viability of using a neural network as a model basis for a model predictive controller for the sake of flow control in a subsea gas and oil well, as some degree of target tracking is achieved. However, its resulting control performance unveils significant challenges related to its implementational steps. An unbalanced data set is identified as a severe source of predictive error in the resulting model, leading to degraded control performance, and teacher forcing is identified as an insufficient way of training the model. Suggested solutions and improvements for future similar work are provided.

Sammendrag

Effektiv og nøyaktig prosessregulering ved utvinning av gass og olje er avgjørende for riggmannskapets sikkerhet, for å minimere klima- og miljøkonsekvenser av utvinningen og for å maksimere profitt av drift. Tradisjonelle metoder for regulering inkluderer modellbasert prediktiv regulering, hvor mange applikasjoner benytter lineære modeller. Nevrale nettverk har omfattende modelleringskapasitet, og ved i stedet å utnytte dette, kan den påfølgende reguleringen forbedres i de tilfeller der prosessen er intrinsisk ulinear. Denne avhandlingen utleder en matematisk formulering for en slik modellbasert prediktiv regulator. Et datasett syntetiseres, og danner grunnlaget for å trene et nevralt nettverk for prediksjon av dynamikk. Det nevrale nettet blir deretter inkorporert i et selvutiklet rammeverk for modellbasert prediktiv regulering. Den ferdige applikasjonen fungerer så som et konseptbevis for nevrale nettverks anvendelighet som modellgrunnlag i en modellbasert prediktiv regulator for regulering av gass- og oljeflytrate i en undersjøisk olje- og gassbrønn, ettersom noen grad av referansefølging oppnåes. Imidlertid avdekker reguleringsoppførselen betydelige utfordringer knyttet til implementasjonsprosessen. Et ubalansert datasett identifiseres som en viktig kilde til prediktiv unøyaktighet i den trente modellen, hvilket fører til forringet regulering. Det påvises at teacher forcing (norsk: lærer-tvinging) er utilstrekkelig som metode for å trene modellen. Videre foreslåes det løsninger og forbedringer for fremtidige liknende arbeider.

Preface

This thesis is written as the final work of the author's degree of Master of Science in Cybernetics and Robotics at the Institute of Engineering Cybernetics, part of the Faculty of Information Technology and Electrical Engineering at the Norwegian Universe of Science and Technology (NTNU). The thesis is written in cooperation with Equinor, and Lars Struen Imsland (representing NTNU) and John-Morten Godhavn (representing Equinor) have provided supervision throughout the thesis work.

A list of the resources made available to the author in relation to the thesis, as well as a list of otherwise utilized resources, are listed in completion in appendix A. A list of acronyms is provided in appendix B, for the convenience of the reader. Note that the project work preceding this thesis has laid the foundation for the developments done in this thesis. Though some material is based on similar material in that project, all reused material is either rephrased and expanded in order to accommodate a wider scope or explicitly stated re-used in the case of full overlap. The linear step response model predictive controller later elaborated, developed and discussed in this thesis, is based on a collaborative work between the author, Simen Bergsvik and Amalie Gjersdal from that project, but is refactored and improved for the occasion of this thesis.

This thesis is written regarding subjects in control theory as well as artificial intelligence. Note then, that some material is elaborated on a level which to qualified people in either field might seem trivial. This is nevertheless necessary, as the basics of either field must be bridged in order to elaborate on a level which touches on both fields, in order to achieve the high-level goals of this thesis. That a figure or variable is non-scalar is indicated by bold font. Any figure signifying a prediction is, according to conventional notation, denoted $\hat{(\cdot)}$.

Acknowledgements

I would like to thank my advisors Lars and John-Morten for guiding me with constant optimism. I would like to thank my family for their ever-open arms and late-night cups of tea. I would like to thank the cinnamon bun crew for all the wonderful lunches, especially on Wednesdays. I would like to thank my study crew for great coffee, banter and distractions when they were needed the most. I would like to thank Jarle for his never-ending ambition, contagious drive and unrelenting friendship. Most of all, I would like to thank my beloved and ever-supporting Silje.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Thesis background and goal

In the extraction of oil and gas, the effectivization of flow rate control has the potential of yielding a multitude of benefits. When desired reference values for gas and oil flow rates are given, more effective and accurate control towards that reference yields both increased economic return and the potential of utilizing actuators more carefully. This in turn has the potential of increasing said actuators' lifespan, thus reducing required maintenance and associated costs. By controlling flow rates more effectively, redundant production may be avoided, thus sparing the climate from e.g. the burning of excessively produced gas. Controlling flow rates accurately also secures that the operating conditions of any given equipment in the system are upheld more effectively, thus increasing the safety in use of said equipment. Equinor operates multiple process industry locations, wherein effective and accurate flow rate control is extremely important.

A method of control which has been widely employed in industry across a multitude of domains for decades is *model predictive control* (MPC)[1]. This method is still popular today and is much used by Equinor as well[2]. Central in MPC is the mathematical formulation of the system model, which ensures that the dynamics of the system to be controlled are considered during the calculation of the optimal actuation sequence. Thus, the MPC has the potential for improved control performance when the accuracy of the system dynamics model increases. Since MPC is based on computing solutions to adequately posed optimization problems, associated computational demand may become heavy. Solving optimization problems is much easier and more effective for linear model-based optimization problems[3]. Consequently, many both traditional and modern industrially applied MPCs are based on a linear system dynamics model, an example of which is linear step response models[2]. However, in many cases, the system is in reality nonlinear, implying that a linear model-based MPC may be subject to significant modelling error, which leaves room for potentially significant control performance gain through improvement of the model.

Though there exist ways to somewhat remedy such accuracy loss, such as using different linear models for different working ranges of the system[4], known as gain scheduling, an alternative approach would be to model the system directly nonlinearly. A nonlinear model carries the potential of describing the system's full working ranges - not just around a neighbourhood. The field of nonlinear system dynamics modelling presents many available options[5], [6]. A central issue of nonlinear system dynamics modelling, however, is deriving a first principles-based model effectively, as this can involve time-consuming mathematical analyses, and thus be resource-costly and expensive. A data-driven model may then prove a favourable option in the case where data is readily available for the system in question. Among the nonlinear data-driven modelling paradigms, neural networks (NN) provide vast theoretical modelling potential due to their universal approximation capabilities[7].

1.2 Literature review

Many modern examples of MPC successfully demonstrate the feasibility, potential efficiency and accuracy of MPC using NN-based models (NNMPC) across a broad array of different applications, thus utilizing available process data effectively, and proving the applied modelling potential of NNs empirically. Indeed, in the context of control applications, they are often used for e.g. modelling a system's dynamics entirely (black-box modelling), modelling a system's dynamics partly as a supplement to a first principles-based model (grey-box modelling), or modelling the uncertainty of an already existing model[6]. NNs may also be utilized as emulators of optimization algorithms, or to replace the controller in full[6]. In [8], reinforcement learning is applied to adjust the parameters of a convex NN-based cost function adaptively, which effectively compensates for model uncertainties. As this thesis will focus on using NNs for black-box modelling, the interested reader is referred to [6] for further details on broader applications of NNs in MPC.

Feasibility of NNMPC in safety-critical domains such as human-machine interactions is demonstrated in [9], as they successfully implement an echo state gaussian process-based MPC for target tracking. Power-efficient temperature control of a building, surpassing prior control methods, is achieved in [10], where the NN-based model is even proven convex, thus reducing the complexity of the resulting MPC implementation. [11] achieved a well-performing and computationally feasible NNMPC on an embedded platform, controlling a quadrotor vehicle - in spite of the platform's restricted computational capacity. A complex paper production pipeline is successfully controlled in [12], using the *gated recurrent unit* (GRU)-variant[13] of the predecessing *long-short term memory* (LSTM) architecture[14] wrapped within a variational encoder-decoder layer.

In [15], multi-step predictions of a steel pickling process' dynamics are achieved by implementing a single-step prediction model by means of a simple *multilayer perceptron* (MLP), which takes in the values of the parameters known to affect the process. The MLP predicts the future value of one of the parameters, such that open-loop chaining the MLP for a number of time steps equalling the prediction horizon yields a multi-step prediction of that parameter, implemented as a *recurrent neural network* (RNN). This approach allowed training the NN-based model as a simple MLP, but deploying it as an RNN. When applied in MPC, significant improvements - relative to PI control performance on the same problem - are observed. Building on a similar approach, [16] trains an MLP to perform single-step predictions of the dynamics of a reactant's concentration in a continuously stirred tank reactor, and goes on to prove the input-to-state stability of the closed-loop controlled system under certain conditions once employing said model recurrently within the MPC. Differently, the model is here based on a *nonlinear autoregressive model with exogenous inputs* (NARX), which allows the model to predict future state of the process based on only historic and current values of its inputs and outputs. The resulting MPC yielded results comparable with an MPC based on the already-known first principles-based model for the same system, thus presenting a strong argument for the feasibility of the method. Note that, once embedded in the MPC, the NARX-model implicitly becomes a recurrent neural NARX-model (RNNARX), yielding multi-step process dynamics predictions[17].

Indeed, common to all the presented applications, is that they design the underlying model such that, once embedded within the MPC implementation, it explicitly or implicitly implements an RNN; the recurrence arises naturally or by design in order to implement the multi-step dynamics predictions required to accommodate the prediction horizon of an MPC. The term NNMPC is here used as an umbrella term for all MPC which use NNs in some way, while RNNMPC implies that the MPC uses an RNN as an underlying model, specifically. The approaches differ in the philosophical foundation on which dynamics are modelled. This is exemplified well by investigating [16] and [12]. Implementing an NN-based model with an underlying NARX state formulation (NNARX), [16] performs dynamics prediction by effectively integrating all transient effects relevant for the next time step in a *simultaneous* matter. Once deployed in the MPC, it becomes an RNNARX, thus implementing an RNNMPC. Differently, [12] accumulates all transient information *over time* and implement the model recurrently from the start, thus facilitating a more adaptive dynamics integration.

The applicability of NN-based models in MPC applications is proven across several different domains, indicating further promises for further applications. However, though derived for individual cases, such as in [16], RNNMPC still lacks *general* proofs of stability and robustness. Additionally, [18] argue that many current RNNMPC implementations have unanswered challenges with regard to interpretability of the model and consequently its consistency with respect to the underlying physical laws. They go on to demonstrate that designing an RNNMPC by considering the physical laws governing the system, i.e. creating a grey-box NN-based model, yields improved control performance on their case study, relative to the case of using a black-box NN-based model.

1.3 Contribution and problem description

The field of NNMPC yields a vast array of successful applications, providing a broad foundation upon which future similar applications may be based. Nevertheless - to the best knowledge of the author - NNMPC has not been employed in the control of flow rates in subsea gas and oil extraction using a well with gas lift capabilities. Based on the field's successes in implementing NNMPC in many different domains, the author of this thesis seeks to contribute to the field of NNMPC the following: a proof-of-concept application of RNNMPC based on an appropriation of the RNNARX multi-step process dynamics predictor employed in [16], designed to control the output gas and oil flow rates of a single subsea oil and gas well. The specific system is described in further detail in Section 1.4. The specific method of modelling is chosen for its conceptual simplicity relative to alternative methods presented, as it is deemed desirable that a solution to the control problem posed is retained as simple as possible. This is based on the author's hypothesis: a simpler implementation bears a stronger promise of being computationally *light* in the case of actual application in the future. A linear step response model-based MPC (LSRMPC), based on the work done in [2], is also implemented in order to attain a grounds of comparison for the implementation of this application.

Based on the motivation presented in Section 1.1, the goal tackled in this thesis revolves around determining whether such a method proves efficient and/or accurate in flow rate control. In the case where the application of this method proves successful, key factors underlying the success should be identified for the sake of reproducibility and usefulness for future similar applications. Conversely, in the case where the application does not prove successful, in addition to identifying the underlying factors causing the failure, mapping alternative approaches is of the utmost interest. Regardless of success or failure: identifying which central challenges must be addressed during the development of an RNNMPC is central. Presently, the problem description of this thesis is formalized as a main goal, facilitated by three sub-goals.

Main goal.

1. Investigate the feasibility of an RNNARX multi-step process dynamics predictor as a black-box model basis for an efficient and accurate MPC for the control of gas and oil flow rates of a single subsea gas and oil well compared to the performance of an LSRMPC.

Sub-goals. The above goal implies implementing a data-driven model of the process in question. As such, a foundation of data on which to train an RNNARX multi-step process dynamics predictor which may be employed in an MPC implementation is necessary. Accordingly, the main goal is facilitated by the following three sub-goals, which accommodate these required steps.

1. Perform data acquisition in order to create data sets suitable to train an RNNARX multi-step process dynamics predictor.
2. Identify and tackle central issues in training an RNNARX multi-step process dynamics predictor.
3. Implement and simulate the control performance of an MPC based on an RNNARX multi-step process dynamics predictor.

Note that implementing the LSRMPC is not defined as a sub-goal; with the exception of its tuning, the LSRMPC's implementation is a result of this thesis' preceding project work, see [19]. Its underlying theory, implementation and results are nevertheless briefly presented throughout this thesis alongside the theory relevant to the NNMPC. This is done to facilitate the understanding of their respective performances compared to each other.

1.4 Case study: the single gas and oil well

This section presents the case system around which this thesis revolves. Since the system is the same as in [19], this section is fetched directly from there, and altered slightly where needed to encapsulate the modifications specific to this thesis.

As mentioned in Section 1.1, the goal of this thesis is to approximate the system dynamics of a single well. The well consists of a pipe connecting a reservoir of oil and gas beneath the seabed, to a valve - the so-called *production choke*, or *choke* for short - above the seabed. Also connected to this pipe is a valve that allows an influx of gas into the fluids flowing from the reservoir to above the seabed. An illustration of the system is given in Figure 1.1. The system will be modelled as a MIMO system, with two inputs and two outputs, where the two inputs are the variables available for control. The next paragraphs explain these inputs and outputs in further detail.

Since the single well consists of a single pipe, leading all flow of gas and oil through the sea floor, the function and purpose of the choke is to constrict the pipe's cross-section where it is placed. By controlling its degree of constriction, the flow rate through the pipe may be controlled. Since the choke can be either fully open or fully constricted, its working range is within $[0, 100]\,[\%]$. Though the choke's opening is in reality a discrete variable, able to change $\pm 2\,[\%]$ at each step, this thesis makes the simplification that its opening is a continuous variable. Controlling the choke's opening is an inexpensive means of control, as it simply involves constricting or expanding a valve at a specific point of the pipe.

The second means of controlling flow in the well's pipe is the *gas lift rate*. The gas lift rate describes the rate at which gas is injected through the gas lift choke into the flow transported from the reservoir through the sea floor. By injecting gas into the flow, the viscous properties are altered, changing the flow dynamics of the fluid within the pipe, thus ratifying the gas lift rate as a means of actuation. In reality, physical constraints to the tools used in controlling the gas lift rate disallow a gas lift rate in the range $(0, 2000)\,[\frac{m^3}{h}]$[1]. This thesis considers a simplified gas lift rate, which is not subject to this constraint. The working range of the gas lift rate is then for the scope of this thesis assumed to be $[0, 10000]\,[\frac{m^3}{h}]$. Note that this amount is from here-on assumed controlled externally with respect to the scope of this thesis; even though the gas lift rate is in reality controlled by the gas lift choke, the gas lift rate will be handled as a flow rate for the rest of this thesis, measured in $[\frac{m^3}{h}]$. Conversely to the choke, the gas lift rate is an expensive means of control, as it involves compressing large quantities of gas, which is a power-demanding process. Ideally, then, the choke valve should be the prioritized actuator, and the gas lift rate should only be used for control when necessary, i.e. for finer adjustments towards reference values or to further increase the flow rate when the choke is fully opened.

The two outputs are the gas rate and the oil rate flowing from the reservoir and through the choke, respectively. These flow as a mixed fluid, but are assumed separately measurable in this thesis.

The work done in this thesis has not interfaced with the described system directly, but has instead had available a digital model of the system. The digital model, as well as values specific to the system, such as lower and upper bounds on outputs, actuation and change in actuation, are all presented in further detail in Section 3.1.3.

1.5 Thesis outline

The further content of this thesis consists of six chapters. Chapter 2 presents the theoretical foundation relevant for the further implementations and gathered results of this thesis across five sections. Chapter 3 presents the specifications to which the MPC implementations' control performances should adhere, and details the considerations made during the implementation of the LSRMPC, the NNARX-model, from here-on referred to as the *model*, as well as the RNNMPC to which it is extended. The results from simulations of control sequences for both the MPC implementations, as well as predictive performance of the model are presented in Chapter 4. Chapter 5 discusses the observed results in light of the presented theory and implementational choices made and draws attention to factors

[1] $0[\frac{m^3}{h}]$ is allowed, as this simply means the gas lift valve is shut close.

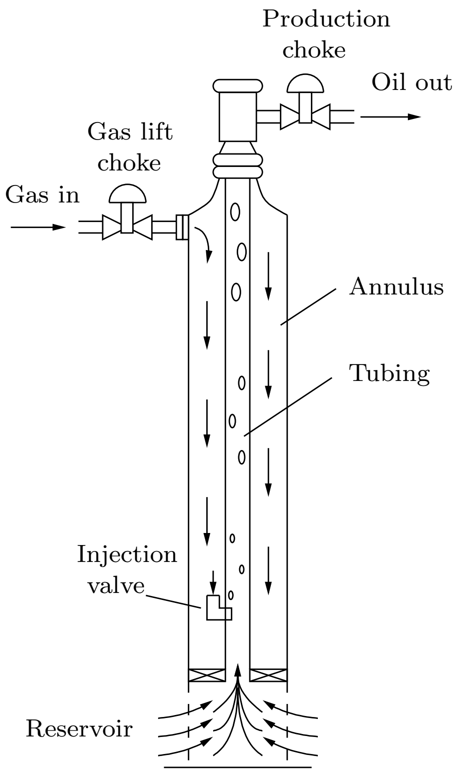


Figure 1.1: An illustration of the model to be discussed in this thesis: the single subsea oil- and gas well. Illustration is fetched with permission from [20].

that should and could be improved upon as well as why and how. Chapter 6 concludes the most important factors identified from both the implementation and testing of the two MPCs. Lastly, Chapter 7 provides suggestions as to how future work may succeed in similar future applications, based on these identified factors.

Chapter 2

Theory

This chapter presents theory relevant for this thesis, on which implementations in Chapter 3 are based, and results in Chapter 4 discussed in Chapter 5. Specifically, Section 2.1 presents general, linear and nonlinear MPC, Section 2.2 covers the basic neural network architecture, Section 2.3 the training of neural networks, Section 2.4 presents the extension from basic neural networks to recurrent neural networks, and Section 2.5 details an MPC problem formulation with a recurrent neural network-based model embedded within.

2.1 Model predictive control

While the introduction to MPC, Section 2.1.1, is fetched directly from [19] due to complete overlap, the further sections Section 2.1.2, Section 2.1.3 and Section 2.1.4, though conceptually very similar to content in [19], are reformulated and expanded to address the wider academic scope in this thesis.

2.1.1 Background

Model Predictive Control (MPC) is a class of control methods that calculate the optimal control sequence for some prediction horizon, given some description of the system's dynamics - a system model. Note that, even though this prediction horizon may be infinite, only finite horizon cases are considered in this project. In contrast to common control alternatives, such as LQ-control, the MPC methods have an industrial origin, rather than academic [21]. MPC bears similarities to LQ-control, of which this thesis does not cover the details. The interested reader is referred to [22].

MPC is implemented by calculating the optimal control sequence for some finite prediction horizon and applying only the first of these optimal control values to the process in question. This is illustrated for a single iteration of an MPC in Figure 2.1. Repeating the process for each time step, a feedback connection is implemented, making MPC a closed-loop control method. As the prediction horizon is kept constant for each time step, it recedes one step ahead into the future for each step ahead the system makes. This control principle is thus called the *Receding Horizon-principle*. It is an essential aspect of MPC, as it allows the trajectory of optimal control values to be adjusted according to continuously occurring sources of inaccuracy, such as model imperfections (including dynamics, kinematics and disturbance) and noise. The Receding Horizon-principle is an

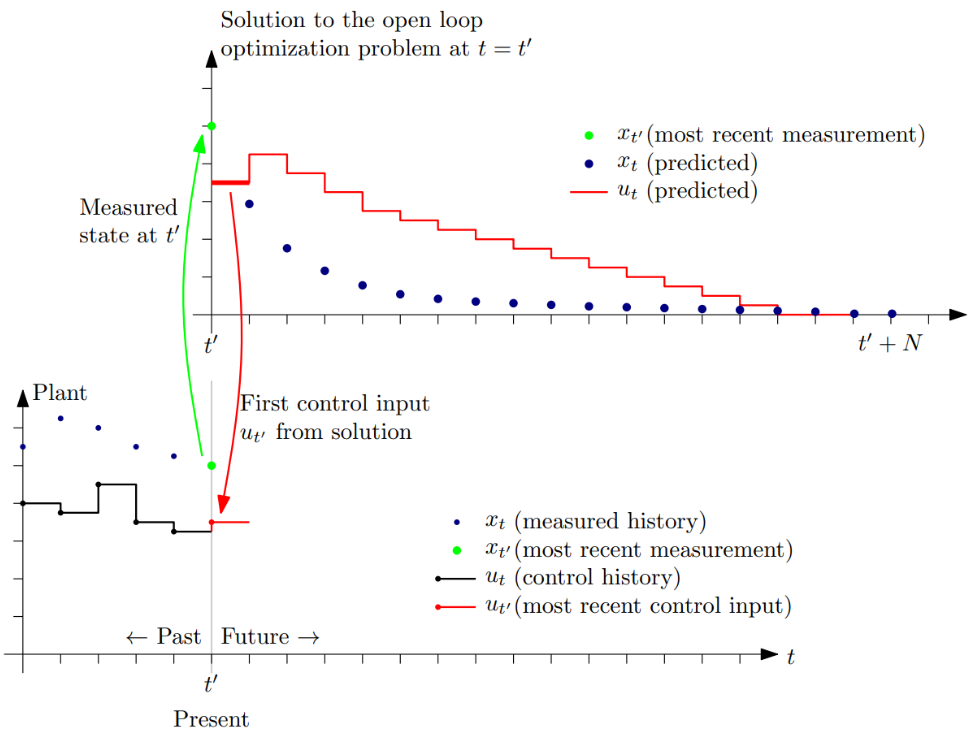


Figure 2.1: Illustration of the MPC principle for a single time step: The actuation for only the next time step is the first actuation value in the optimal trajectory as calculated from the optimization problem. The figure is fetched with permission from [24], p. 41.

important reason for the robustness qualities that MPC exhibits [23], and is also one of the main factors that distinguishes it as a control method from LQ-control.

Another essential aspect of MPC that distinguishes it from alternative control approaches, is its ability to take into account physical model constraints. Constraints may include upper and lower limits to state and/or actuation values, but importantly also enable the engineer to encode the system model into the optimization problem formulation as an equality constraint. The ability of MPC to take these into account stems from its use of an optimization problem solver (from here-on referred to as a "solver"). Solvers are algorithms that find a minimum - or maximum, depending on convention - in any given function, with the capability of restricting the accepted solution space to a sub-space defined by a set of inequality and equality constraints that are formulated as part of the optimization problem.

Even though the above-mentioned characteristics make MPC a widely used and state-of-the-art process control method, it has drawbacks, mainly associated with computational cost. This is briefly presented in Section 2.1.3 and Section 2.1.4.

Further details of the general MPC problem formulation and aspects regarding linearity and nonlinearity are presented in the following sections. Theoretical background for numerical optimization is not provided in this thesis. The interested reader is instead referred to [3].

2.1.2 General MPC

MPC determines the optimal control sequence by means of minimizing the value of some objective function $l(\cdot)$, often called the *cost function*:

$$\min_{\boldsymbol{x}} l(\boldsymbol{x}), \tag{2.1}$$

while also taking into account the constraints to which the physical system must adhere, by encoding said constraints as sets of equalities and inequalities:

$$\min_{\boldsymbol{x}} l(\boldsymbol{x}) \tag{2.2a}$$

$$s.t.$$

$$\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0}, \quad \boldsymbol{h}(\boldsymbol{x}) \geq \boldsymbol{0} \tag{2.2b}$$

The rest of this section covers relevant considerations for both the cost function and the constraints and elaborates on how to formulate them thereafter, before briefly covering optimization in MPC.

Cost function. Since the problem is one of minimization with respect to some set of variables, any values causing the cost function to go towards $-\infty$ would be preferred by the solver. However, this poses two main issues. Firstly, if the cost function tends towards $-\infty$, the optimization will not converge, and the optimizer will not find an optimum. Secondly, the cost function is simply a function designed to describe the degree of optimality in any given solution with respect to the set of optimization variables. This implies that there does not necessarily exist any direct link between a minimum in the cost function and a desired, or even tractable, state in the actual physical system for such an optimum, unless by intentional design. Thus, if the cost function design is poor, an optimum may yield poor performance in the system, or even be physically intractable.

To avoid this, the cost function must be designed adequately. Firstly, there must be a defined global minimum, such that the optimizer is able to converge. This is achieved easily by defining the cost function to be quadratic with respect to at least (but not limited to) the vector of optimization variables, here exemplified with a generic vector $\boldsymbol{x}$ and a matrix $\mathbf{Q}$ consisting of scaling factors synonymously called *weights.*

$$l(\boldsymbol{x}) = \boldsymbol{x}^{\mathsf{T}}\mathbf{Q}\boldsymbol{x} \tag{2.3}$$

It is important that the weight matrix $\mathbf{Q}$ is *positive semi-definite* (PSD), $\mathbf{Q} \succeq 0$, such that the cost function becomes monotonically increasing along all axes[3], and consequently that a global minimum is guaranteed to exist. Furthermore, PSD quadratic cost functions have the benefit of being *convex*, which ensures that any local optimum is a global optimum[3].

Note that a positive semi-definite function on the same form as (2.3) has its global minimum in the origin. By offsetting each variable with some value, for example the reference value $\boldsymbol{x}_{ref}$ in the target-tracking case, the global optimum can be shifted thereafter, such that it corresponds to the desired state in the true system:

$$l(\boldsymbol{x}; \boldsymbol{x}_{ref}) = (\boldsymbol{x} - \boldsymbol{x}_{ref})^{\mathsf{T}} \mathbf{Q} (\boldsymbol{x} - \boldsymbol{x}_{ref}) \tag{2.4}$$

Since the goal of optimization in MPC is to determine the optimal control action at the current time step, the optimization variables should either explicitly or implicitly determine the system's degrees of freedom - the input variables - through which the system state may be manipulated. In the explicit case, the input is used as the optimization variable directly. Conversely, in the implicit case, optimization is performed with respect to for example changes in input between time steps, $\boldsymbol{\Delta u}$, or state variables, $\boldsymbol{y}$, both of which implicitly determine the input's values through system modelling in the constraints. This is later exemplified in (2.8h) and (2.8c), respectively. Optimization is done based on open-loop predictions of future state for some amount of time steps into the future. An example of optimization that implicitly provides the optimal input may look like the following:

$$\min_{\boldsymbol{\Delta u}_{k:k+N-1}} l(\boldsymbol{y}_{k+1:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref,k+1:k+N}), \tag{2.5}$$

where $l(\boldsymbol{y}_{k+1:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref,k+1:k+N})$ is a generic scalar-valued cost function with respect to the sequence of vector-valued variables $\boldsymbol{y}_{k+1:k+N}$ and $\boldsymbol{\Delta u}_{k:k+N-1}$ as well as parameters representing a given reference $\boldsymbol{y}_{ref,k+1:k+N}$, and N is a number of time steps ahead into the future.

The cost function is usually designed to give some measure of penalty to a candidate solution proportional to how well the solution follows the goals set for the MPC problem formulation [21]. In (2.5) this can, for example, be how much $\boldsymbol{y}_{k+1}$ deviates from $\boldsymbol{y}_{ref,k+1}$ - the less the better - and how much change in actuation $\boldsymbol{\Delta u}_k$ is required in order to minimize this deviance - the less the better. Intuitively, the cost function's global minimum will then provide the optimal solution. Note that, in this specific case, no considerations regarding the resulting value of the actuation itself are made, only its rate of change. In order to avoid divergence in actuation and the breaking of physical actuators' limits, upper and lower bounds on actuation should be encoded into the optimization problem's constraints. This is later exemplified in (2.8f).

Formulating (2.5) in the quadratic fashion of (2.4) may provide a cost function as follows:

$$\begin{aligned} l(\boldsymbol{y}_{k+1:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref,k+1:k+N}) = \sum_{i=0}^{N-1} &(\boldsymbol{y}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^{\mathsf{T}} \mathbf{Q} (\boldsymbol{y}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) \\ &+ \boldsymbol{\Delta u}_{k+i}^{\mathsf{T}} \mathbf{R} \boldsymbol{\Delta u}_{k+i}, \end{aligned} \tag{2.6}$$

Note that, while it suffices that $\mathbf{Q} \succeq 0$, $\mathbf{R}$ has to be *positive definite* (PD), $\mathbf{R} \succ 0$. This is in order to avoid potentially cancelling the cost function's actuation rate-term in (2.6), which would imply the actuation could change at any arbitrary rate without impacting the solution, resulting in arbitrary actuation values and quite possibly damage to the physical actuation equipment. Intuitively, arbitrary values in actuation are not acceptable as a solution, as this would cause arbitrary system behaviour. Furthermore, the cost function as a whole must remain at least positive semi-definite, such that a defined global minimum is guaranteed to exist, for the sake of convergence. Note that the strict requirement of positive definiteness is not made for $\mathbf{Q}$ because the acceptable solution space is defined by means of inequality constraints to be such that any value therein is acceptable with respect to tractability in the physical system, later exemplified (2.8d). Furthermore, coherence between input and state is enforced by means of encoding the system model as an equality constraint, later exemplified in (2.8d) and (2.8c). Intuitively, any solution then existing within the defined solution space is acceptable since it is not arbitrary, but comes as a direct consequence of the calculated optimal actuation values.

The global optimum of the quadratic cost function will in every unconstrained case, as in (2.1), be the (potentially shifted) origin, due to the cost function's positive semi-definiteness. This is not necessarily the case for any constrained case, as in (2.2), in which the global, unconstrained optimum may be outside the legal bounds defined by the constraints. Where the constrained global optimum lies depends on both the specific set of constraints as well as the surface shape of the cost function. While the set of constraints is usually derived from physical considerations, and not subject to alterations, this is not the case for parameters defined as part of the cost function. In (2.6), these are the optimization horizon N, as well as the weights $\mathbf{Q}$ and $\mathbf{R}$, all of which are *tunable*; their values may be determined by the user according to what yields the best behaviour. Consequently, the control performance is directly affected by the specific values of the optimization horizon and the weights.

The size of the optimization horizon N defines how many time steps, with which to achieve its goals, the solver has at its disposition. A small N then indicates that the reference values must be reached in fewer time steps, yielding the solution fewer degrees of freedom. This will result in a more crude and aggressive optimal control sequence. Conversely, if the prediction horizon is larger, the solution will have more degrees of freedom, and a more optimal control sequence may be found. Indeed, a larger prediction horizon *does* result in a better control performance, as indicated in [24]. However, solving an optimization problem with more degrees of freedom involves higher computational costs. The exact size of the prediction horizon thus presents a compromise between computational cost and performance and is a parameter that must be tuned to fit any specific MPC scheme.

In (2.6), the *control horizon* is implicitly equal to the *prediction horizon.* However, separating the two is perfectly viable, in which case the expression may be altered:

$$
\begin{aligned}
l(\boldsymbol{y}_{k+1:k+H_p}, \boldsymbol{\Delta u}_{k:k+H_u}; \boldsymbol{y}_{ref,k+1:k+H_p}) = &\sum_{i=0}^{H_p-1} (\boldsymbol{y}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^\mathsf{T} \mathbf{Q} (\boldsymbol{y}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) \\
&+ \sum_{i=0}^{H_u} \boldsymbol{\Delta u}_{k+i}^\mathsf{T} \mathbf{R} \boldsymbol{\Delta u}_{k+i},
\end{aligned}
\tag{2.7}
$$

Note that calculating any actuations for time steps beyond the prediction horizon is superfluous, as the corresponding outputs would in such a case not be predicted, and the extra actuation values will then have yielded no extra information. Coherent with the definition in (2.7), any implementation should always define $H_p > H_u$.

The values of the elements of $\mathbf{Q}$ and $\mathbf{R}$ scale the variables in the cost function, thus affecting the steepness of the cost function's slopes along each axis. The higher the values of the weights, the steeper the slope of the cost function along the corresponding variable's axis, the more impactful changes in that variable become. Thus, the magnitudes of the weights imply the penalty contributed to the cost function from the corresponding variable's deviations from the value at the unconstrained optimum, and tuning $\mathbf{Q}$ and $\mathbf{R}$ changes the behaviour of minima found during minimization. This can be used to alter the MPC scheme's effectiveness with respect to different aspects. In the case of quadratic cost functions, such as (2.6), the weight matrices $\mathbf{Q}$ and $\mathbf{R}$ are usually implemented as diagonal matrices, such that the cost function becomes a simple sum of a weighted square of all variables. Then, high-valued elements of $\mathbf{Q}$ will penalize deviations from the reference, $\boldsymbol{y}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}$, whereas high-valued elements of $\mathbf{R}$ will penalize rapid changes in actuation, $\boldsymbol{\Delta u}_{k+i}$. Intuitively, the weights are a means of encoding priorities in the constrained optimum with respect to different aspects of the system. Finding the ideal tuning of any cost function must be done on a case-by-case basis, as it will depend highly on the system and desired behaviour.

Constraints. The true strength of MPC as a method of process control comes from its ability to take into account constraints, which allows physical regards of the true system to be facilitated. These are regards of the physical system, such as limits to actuation, limits to the rate of change in actuation as well as consistency with the system dynamics. Note that the constraints may encode more abstract concepts as well, such as mathematical consistency with initial state. The encoding may be performed by reformulating the generic equalities and inequalities in (2.2b), such that they instead represent individual constraints of the system.

To exemplify this, a basic case of MPC is presented, where (2.7) is used as a cost function. This example later lays the foundation for MPC problem formulations (2.11) and (2.30), used in Section 3.2 and Section 3.3, respectively.

$$\begin{aligned} \min_{\boldsymbol{\Delta u}_{k:k+H_u}} \sum_{i=0}^{H_p-1} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^{\mathsf{T}} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) \\ + \sum_{i=0}^{H_u} \boldsymbol{\Delta u}_{k+i}^{\mathsf{T}} \mathbf{R} \boldsymbol{\Delta u}_{k+i} \end{aligned} \tag{2.8a}$$

$$s.t.$$

$$\hat{\boldsymbol{x}}_k = \boldsymbol{x}_k \tag{2.8b}$$
$$\hat{\boldsymbol{x}}_{k+1+i} = \boldsymbol{f}(\hat{\boldsymbol{x}}_{k+i}, \boldsymbol{u}_{k+i}) \qquad \forall\ i \in [0, H_p - 1] \tag{2.8c}$$
$$\hat{\boldsymbol{y}}_{k+1+i} = \hat{\boldsymbol{x}}_{k+1+i} + \boldsymbol{v}_{k+i} \qquad \forall\ i \in [0, H_p - 1] \tag{2.8d}$$
$$\boldsymbol{y}_{lb} \leq \hat{\boldsymbol{y}}_{k+1+i} \leq \boldsymbol{y}_{ub} \qquad \forall\ i \in [0, H_p - 1] \tag{2.8e}$$
$$\boldsymbol{u}_{lb} \leq \boldsymbol{u}_{k+i} \leq \boldsymbol{u}_{ub} \qquad \forall\ i \in [0, H_p - 1] \tag{2.8f}$$
$$\boldsymbol{\Delta u}_{lb} \leq \boldsymbol{\Delta u}_{k+i} \leq \boldsymbol{\Delta u}_{hb} \qquad \forall\ i \in [0, H_u] \tag{2.8g}$$
$$\boldsymbol{\Delta u}_{k+i} = \mathbf{0} \qquad \forall\ i \in (H_u, H_p - 1] \tag{2.8h}$$
$$\boldsymbol{u}_{k+i} = \boldsymbol{u}_{k-1+i} + \boldsymbol{\Delta u}_{k+i} \qquad \forall\ i \in [0, H_p - 1] \tag{2.8i}$$
$$\boldsymbol{v}_{k+i} = \boldsymbol{y}_k - \hat{\boldsymbol{y}}_k \qquad \forall\ i \in [1, H_p - 1] \tag{2.8j}$$

Given (2.8c) and (2.8d), the minimization problem (2.8a) is forced to comply with the system dynamics, here assumed to be described by some state space formulation. Furthermore, (2.8e) describes the region, with respect to predicted system state, within which any solution is required to exist. Lastly, limits to actuation and rates of change in actuation, as well as the consistency of change in actuation, are upheld by (2.8f), (2.8g) and (2.8i), respectively. Note that (2.8h) is enforced as a requirement, to ensure consistency, as $\boldsymbol{u}_{k+i:k+H_p-1}\ \forall\ i > H_u$ would otherwise not be defined.

By adhering to the requirement in (2.8c) and (2.8d), the optimizer is performing open-loop predictions of future state. Consistency with current state at step k is ensured by the constraint in (2.8b). As is convention, this project will denote any prediction $\hat{(\cdot)}$, exemplified with $\hat{\boldsymbol{y}}_{k+1+i}$.

Since any system model has inaccuracy, this should be possible to account for also in the MPC problem formulation. This is exemplified in (2.8i), where the prediction error $\boldsymbol{v}_{k+i} = \boldsymbol{v}_k$ is assumed constant as a default case where no modelling inaccuracy dynamics are known. Though the expression for $\boldsymbol{v}_k$ should be tailored to any specific case, it is here exemplified as an assumed constant difference between the last measured output $\boldsymbol{y}_k$ and the corresponding predicted output $\hat{\boldsymbol{y}}_{k|k-1}$. The predicted output is denoted $\hat{\boldsymbol{y}}_{k|k-1}$ to indicate that the prediction was made *for*, but *prior* to, the current timestep.

The set of all the constraints in a constrained MPC problem formulation spans a subset of the solution space of the unconstrained variant of the same MPC problem. We call this subset the *feasible set* [24]. In the case of (2.8), the feasible set consists of only *hard* constraints: the constraints are absolute, and the solution *must* adhere. When subject to hard constraints, a minimization problem does not necessarily produce a solution within the feasible set. Thus, it may be advantageous to allow the feasible set some degree of flexibility. This can be done by introducing *slack variables* to the lower and upper bounds of the constraints causing infeasibility, after which these constraints are now called *soft* - they are no longer absolute. The flexibility is implemented by adding the slack variables to the set of optimization variables, such that the feasible set may be expanded if required:

$$\begin{aligned}\min_{\boldsymbol{\Delta u}_{k:k+H_u},\boldsymbol{\epsilon_y}} \sum_{i=0}^{H_p-1} &(\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^\mathsf{T}\mathbf{Q}(\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) \\ &+ \sum_{i=0}^{H_u} \boldsymbol{\Delta u}_{k+i}^\mathsf{T}\mathbf{R}\boldsymbol{\Delta u}_{k+i} \\ &+ \boldsymbol{\rho}^\mathsf{T}\boldsymbol{\epsilon_y}\end{aligned} \tag{2.9a}$$

$$s.t.$$

$$\hat{\boldsymbol{x}}_k = \boldsymbol{x}_k \tag{2.9b}$$

$$\hat{\boldsymbol{x}}_{k+1+i} = \boldsymbol{f}(\hat{\boldsymbol{x}}_{k+i}, \boldsymbol{u}_{k+i}) \qquad \forall\ i \in [0, H_p-1] \tag{2.9c}$$

$$\hat{\boldsymbol{y}}_{k+1+i} = \hat{\boldsymbol{x}}_{k+1+i} + \boldsymbol{v}_{k+i} \qquad \forall\ i \in [0, H_p-1] \tag{2.9d}$$

$$\boldsymbol{y}_{lb} - \boldsymbol{\epsilon_y} \leq \hat{\boldsymbol{y}}_{k+1+i} \leq \boldsymbol{y}_{ub} + \boldsymbol{\epsilon_y} \qquad \forall\ i \in [0, H_p-1] \tag{2.9e}$$

$$\boldsymbol{u}_{lb} \leq \boldsymbol{u}_{k+i} \leq \boldsymbol{u}_{ub} \qquad \forall\ i \in [0, H_p-1] \tag{2.9f}$$

$$\boldsymbol{\Delta u}_{lb} \leq \boldsymbol{\Delta u}_{k+i} \leq \boldsymbol{\Delta u}_{hb} \qquad \forall\ i \in [0, H_u] \tag{2.9g}$$

$$\boldsymbol{\Delta u}_{k+i} = \mathbf{0} \qquad \forall\ i \in (H_u, H_p-1] \tag{2.9h}$$

$$\boldsymbol{u}_{k+i} = \boldsymbol{u}_{k-1+i} + \boldsymbol{\Delta u}_{k+i} \qquad \forall\ i \in [0, H_p-1] \tag{2.9i}$$

$$\boldsymbol{v}_{k+i} = \boldsymbol{y}_k - \hat{\boldsymbol{y}}_{k|k-1} \qquad \forall\ i \in [0, H_p-1] \tag{2.9j}$$

$$\boldsymbol{\epsilon_y} \geq \mathbf{0} \qquad \forall\ i \in [0, H_p-1] \tag{2.9k}$$

where the dimensionalities of $\boldsymbol{\epsilon_y}$ and $\boldsymbol{\rho}$ match that of $\mathbf{y}$. Subject to linear constraints, (2.9) is still convex [24] (p. 42). Like $\mathbf{Q}$ and $\mathbf{R}$, $\boldsymbol{\rho} > 0$ is a vector containing weights for the slack variables and may be tuned, in order to alter the behaviour of the cost function with respect to the slack variables. The values of $\boldsymbol{\rho}$ define how much the cost function increases in value through adding slack to the constraints. If the solver is able to find a minimum without applying slack, i.e. $\boldsymbol{\epsilon_y} = \mathbf{0}$, then no slack will be added, as this is the minimal and optimal contribution the slack-term can give. This is true when (2.9k) is upheld and under the assumption that $\boldsymbol{\rho} > \mathbf{0}$. Like $\mathbf{R}$, $\boldsymbol{\rho}$ must be positive definite, such that the slack variables may not be varied arbitrarily. Arbitrary values in the slack variables would result in arbitrary alterations of the corresponding constraints, defeating their purpose. In the example of (2.9), the constraints are only made soft for $\boldsymbol{y}_{k+1+i}$ in (2.9e), since allowing slack in actuations would mean potentially allowing violation of physical constraints, such as actuators' saturations.

Optimization. The solver is the algorithm that finds the solution to the MPC problem formulation as it is presented in (2.9) at every time step. Most numerical optimization methods are gradient-based[3]. Delving further into the theoretical foundations and concrete implementations of available solvers is beyond the scope of this project. The interested reader is referred to [3]. Considerations made when choosing the solvers used in this thesis are instead presented where relevant in Section 3.2 and Section 3.3.

A significant drawback of MPC is the computational cost associated with solving an optimization problem at each time step, as this in many cases becomes non-trivial. This, however, depends on the MPC problem formulation. Broadly speaking, MPC problem formulations fall under one of two categories: *linear MPC* and *nonlinear MPC*, which are covered in the following two sections.

2.1.3 Linear MPC

MPC problem formulations fall under the category of linear MPC if their cost functions are *convex* and their constraints *linear*[3]. Linear MPC-schemes observe vastly reduced computational complexity because they bear the benefit of being solvable with convex solvers; for convex functions, local extrema are guaranteed to be global extrema[24], meaning that solving the MPC optimization problem becomes a matter of seeking a point where the cost function's gradient becomes zero. The interested reader is referred to [3] for a detailed discussion regarding both convex and non-convex solving methods.

While a cost function is subject to design, and may be designed convex, nonlinearity in the constraints is enough to make the whole MPC problem formulation nonlinear. Such cases pose a tradeoff. If the constraints, e.g. the system model, are linearized such that the full MPC problem formulation becomes linear, the computational complexity is reduced, potentially at the cost of some control performance[25]. This might be both viable and desirable in smaller systems with reduced computational capacity, such as embedded systems[2], and especially if control performance is deemed good enough.

Linear Step Response MPC (LSRMPC). One example of a much-used method of linearization is *linear step-response modelling*[2], used in linear step-response MPC (LSRMPC). The following text derives the LSRMPC and is directly fetched from "Section 2.1.5 Linear Step Response MPC" in [19], with some typographical alterations as well as improvements in the mathematical formulations.

The distinguishing factor between LSRMPC and the generic MPC derived in (2.9) is the model used to describe the system dynamics. Linear step response modelling is a way of modelling processes as input-output relations by linearizing around some working point for the inputs. By applying a step on an input of the system and measuring the response of an output of the system, a proportional relationship between the two may be deduced. Deducing each such proportionality coefficient when applying a step input from the beginning until the system has settled, results in a series of coefficients that map the effects over time which changes in an input have on a corresponding output in a linear fashion [21]. The first of these coefficients represents the effect a change in input has from the moment it occurs to the next time step. The last coefficient represents the lasting effect an input has had after the dynamics have settled. Since the step response model is made around some working point, the linearization is only a viable approximation around this working point. Depending on the system's degree of nonlinearity, this linearization can not be expected to yield accurate predictions far away from the linearization point. For more detailed examples and literature, the reader is referred to [21] and [22].

The series of coefficients is the full step response model for the SISO relationship between an input and an output [21], and is denoted $S \in \mathbb{R}^N$, where N is the number of steps before the dynamics settle. Since the step response model describes a system's dynamics, it may be used to predict future output values at any step j into the future. As given in [21] (equation (20-10)) for a SISO system:

$$\hat{y}_{k+j} = \sum_{i=1}^{j}[S_i \Delta u_{k+j-i}] + \sum_{i=j+1}^{N-1}[S_i \Delta u_{k+j-i}] + S_N u_{k+j-N} + v_{k+j} \tag{2.10a}$$

$$v_{k+j} = y_k - \hat{y}_{k|k-1}, \tag{2.10b}$$

where $\hat{y}_{k+j}$ is the predicted difference in output between timesteps $k+j-1$ and $k+j$, and N is the settling time of the SISO relationship. The first two sums in (2.10a) represent the contribution of future changes in input and the contribution of past changes in input, respectively. The third term represents the lasting contribution to the output after it has settled from some past input, and the last term corrects for a bias resulting from modelling error, that is assumed constant - (2.10b). Altering (2.9) to instead comply with the model of system dynamics as described in (2.10), we arrive at a new MPC problem formulation:

$$\min_{\Delta u_{k:k+N-1}} \sum_{j=0}^{N-1}[Q(\hat{y}_{k+1+j} - y_{ref,k+1+j})^2 + R\Delta u_{k+j}^2] + \rho \epsilon_y \tag{2.11a}$$

$$s.t.$$

$$\hat{y}_k = y_k \tag{2.11b}$$

$$\hat{y}_{k+1+j} = \sum_{i=1}^{1+j}[S_i \Delta u_{k+1+j-i}] + \sum_{i=j+2}^{N-1}[S_i \Delta u_{k+1+j-i}] + S_N u_{k+j-N} + v_{k+j} \quad \forall\ j \in [0, N-1] \tag{2.11c}$$

$$y_{lb} - \epsilon_y \leq \hat{y}_{k+1+j} \leq y_{lb} + \epsilon_y \quad \forall\ j \in [0, N-1] \tag{2.11d}$$

$$u_{lb} \leq u_{k+j} \leq u_{ub} \quad \forall\ j \in [0, N-1] \tag{2.11e}$$

$$\Delta u_{lb} \leq \Delta u_{k+j} \leq \Delta u_{ub} \quad \forall\ j \in [0, N-1] \tag{2.11f}$$

$$u_{k+j} = u_{k-1+j} + \Delta u_{k+j} \quad \forall\ j \in [0, N-1] \tag{2.11g}$$

$$v_{k+j} = y_k - \hat{y}_{k|k-1} \quad \forall\ j \in [0, N-1] \tag{2.11h}$$

The system is here assumed to be SISO - i.e. scalar input and output - for simplicity. The SISO step response model may be generalized to MIMO by first identifying each SISO relationship - as explained above - before combining them in a matrix to represent the full MIMO step response model; the vector S is generalized to the matrix $\mathbf{S} \in \mathbb{R}^{n_{out}N \times n_{in}}$:

$$\mathbf{S} = \begin{bmatrix} S_{1,1} & S_{1,2} & \cdots & S_{1,n_{in}} \\ S_{2,1} & S_{2,2} & \cdots & S_{2,n_{in}} \\ \vdots & \vdots & \ddots & \vdots \\ S_{n_{out},1} & S_{n_{out},2} & \cdots & S_{n_{out},n_{in}} \end{bmatrix}, \tag{2.12}$$

where each row represents the SISO relationships between all inputs and a single output, and each column represents the SISO relationships between a single input and all outputs.

Integrating a MIMO step response model into an MPC problem formulation is covered in the implementational details of the LSRMPC, Section 3.2.1.

In the cases where linearization is not an option, loss of control performance is not desirable, or computational power is not a limiting resource, the MPC problem formulation may be retained as nonlinear and handled thereafter.

2.1.4 Nonlinear MPC

MPC problem formulations fall under the category of *nonlinear MPC* if their cost functions are non-convex and/or any of their constraints are nonlinear. Where convexity guarantees that a local optimum is also a global one, the lack of convexity implies that there is no such guarantee, and different methods of optimization must be utilized. Such nonlinear optimization is necessarily more computationally demanding.

Nevertheless, nonlinear MPC is shown to perform better than linear MPC, e.g. in cases where the system dynamics are fast and nonlinear, one example being autonomous drones[25]. Whether the added computational cost is a worthwhile investment or not is subject to consideration on a case-by-case basis.

Nonlinear MPC is in its most general form described by (2.2), but a specific example is later provided in Section 2.5.

2.2 Artificial neural networks

Though the following sections are based on similar sections from [19], they are altered to better suit this thesis. Specifically, all sections are reformulated and expanded to be more generally applicable as well as address the wider academic scope of this thesis.

Artificial Neural Networks (NNs) is the name of a class of machine learning models whose structure is inspired by the brain's (hence the name), processing information by mimicking signals firing between interconnected neurons in order to produce some output[17].

Throughout this thesis, the term *model* refers to an NN trained to perform tasks relevant to the thesis.

2.2.1 Machine learning and deep learning

A high-level explanation of machine learning is presented in [26](p. 2):

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Paraphrasing this explanation in more intuitive terms, machine learning algorithms are algorithms that perform their tasks better when given increased amounts of relevant data.

There are mainly two ways of using data to facilitate algorithms in modelling processes by means of machine learning. One is to model processes based on the information directly available in the data presented. This is called *lazy learning*[27], as the data is simply fed through a model, yielding some output; it is a static operation, and exposure to data does not provide opportunity to alter the model itself. Instead, the algorithm can be seen to learn from the data, in the sense that it models the desired process more accurately, given a more densely populated data set. The algorithm is able to utilize available data, but has no inherent understanding of the process it models.

The complement to lazy learning is *eager learning*[27], a type of machine learning in which the algorithm uses the given data to *train* some model of the process which the given data represents. The process of training involves actively shaping the model, meaning that exposure to data creates a lasting impact on how the algorithm models the process in the future. The process of training a machine learning algorithm is covered in Section 2.3.

Deep learning represents the vast class of eager machine learning methods that are implemented as NNs. Deep learning algorithms can be employed in a vast array of tasks, such as classification, regression, language translation and anomaly detection, to name a few[17]. This thesis will tackle problems of regression, further elaborated in Section 2.5 and Section 3.3. Consequently, all further derivations are based on NNs for regression.

Regression by means of deep learning is presented in [17](p. 99) as the following:

> "[...] the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. [...]"

The dimensionality of the function to be approximated must of course be considered. In general terms, one may alter the above statement to apply generally and say that the learning algorithm should implement a function $\hat{\boldsymbol{f}} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ that predicts $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}_{in}) \in \mathbb{R}^m$ for $\boldsymbol{x}_{in} \in \mathbb{R}^n$. When training a deep learning model to approximate a true system by means of regression, it will always remain an *approximation* of the true system, meaning that its predictions must always be expected to deviate to some extent from the true values. Generally:

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}_{in}) \approx \hat{\boldsymbol{f}}(\boldsymbol{x}_{in}) = \hat{\boldsymbol{y}}, \tag{2.13}$$

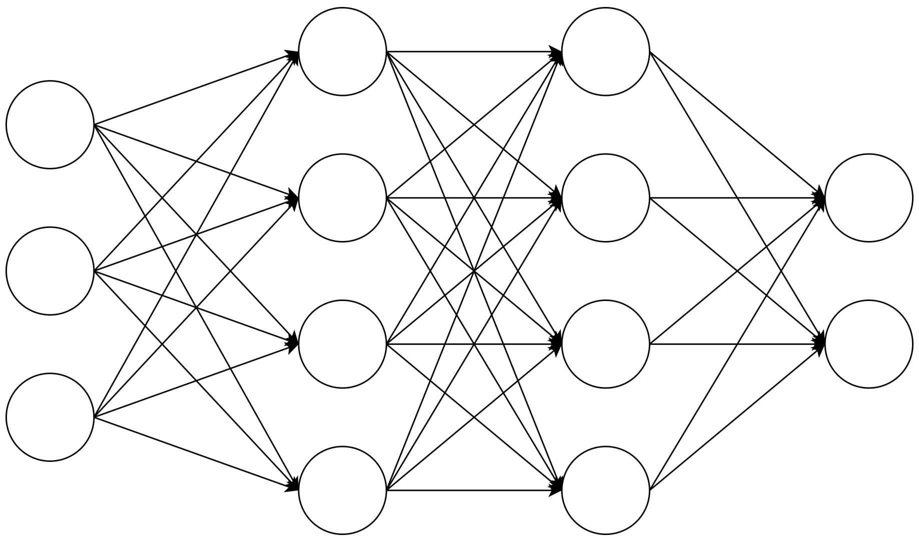


Figure 2.2: An illustration of a simple MLP. The input layer has 3 neurons, the two hidden layers have 4 neurons each, and the output layer has 2 neurons.

2.2.2 The multilayer perceptron

The NN in its most basic form consists of an *input layer*, some number of *hidden layers* and an *output layer*, each of which consists of some number of *neurons*. The input layer holds all given input values - $\boldsymbol{x}_{in}$ in (2.13) - and the output layer delivers the final function value - $\hat{\boldsymbol{f}}(\boldsymbol{x}_{in})$ in (2.13). In the case of regression, the NN's goal is to provide an output that matches the ground truth - $\boldsymbol{f}(\mathbf{x}_{in})$ in (2.13) - as closely as possible. The hidden layers are implemented as intermediate steps between the input and the output, in order to facilitate the approximation. All of the hidden layers and the output layer each implements a set of parameters, collectively denominated as $\boldsymbol{\theta}$. These parameters shape the function $\hat{\boldsymbol{f}}(\mathbf{x}_{in};\boldsymbol{\theta})$ and how it approximates. Note that only the final result given by the output layer is of interest, and thus we do not care about the specific values of the hidden layers, as long as the final output is satisfactory - hence the name *hidden*. The layers' parameters are described in further detail below.

The type of NN here described is called a *Multilayer Perceptron* (MLP), an example of which is illustrated in Figure 2.2.

Each neuron represents a function applied to its input and consists of three components. That each neuron represents a function, means that each layer can be viewed as a vector-valued function of the previous layer; the network as a whole is a sequential convolution of every layer. In the specific case of the MLP illustrated in Figure 2.2:

$$\boldsymbol{f}(\boldsymbol{x}_{in};\boldsymbol{\theta}) = \boldsymbol{f}_3(\boldsymbol{f}_2(\boldsymbol{f}_1(\boldsymbol{x}_{in};\boldsymbol{\theta}_1);\boldsymbol{\theta}_2);\boldsymbol{\theta}_3), \tag{2.14}$$

where the input layer $\boldsymbol{x}_{in}$ provides values that pass through $\boldsymbol{f}_1$, then $\boldsymbol{f}_2$, before they are output from the output layer $\boldsymbol{f}_3$. The following paragraphs aim to explain the vector-valued function in each layer and provide rationale as to their structure.

Values are given to the MLP's input layer by some external interface, for example the user. The output is then calculated as a result of these input values' propagation through the MLP, illustrated by the arrows in Figure 2.2. Importantly, values are weighted as they propagate through the MLP, meaning that every arrow represents some *weight* on the value passing from a node in layer $l-1$ to a node in layer l. The number of neurons in any layer l - its *width* - is defined to be η_l in this thesis. A neuron in layer l takes in the weighted sum of all values in layer $l-1$, and so the description for a general layer l becomes:

$$\boldsymbol{f}_l(\boldsymbol{x}) = \mathbf{W}_l\boldsymbol{x}, \tag{2.15}$$

where $\mathbf{W} \in \mathbb{R}^{\eta_l \times \eta_{l-1}}$ and $\boldsymbol{x} \in \mathbb{R}^{\eta_{l-1}}$ is the collection of weights on the form:

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,\eta_{l-1}} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,\eta_{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{\eta_l,0} & w_{\eta_l,1} & \cdots & w_{\eta_l,\eta_{l-1}} \end{bmatrix}$$

Additionally, a neuron contains a *bias* added to the weighted sum, rendering the layer l as:

$$\boldsymbol{f}_l(\boldsymbol{x}) = \mathbf{W}_l\boldsymbol{x} + \boldsymbol{b}_l, \tag{2.16}$$

where of course $\boldsymbol{b}_l \in \mathbb{R}^{\eta_l}$. Note that this function approximation is only a linear transformation of the input, and is thus not able to approximate nonlinear dynamics. Furthermore, linear transformations applied to linear transformations also remain linear[28], such that a full network with layers as defined in (2.16) is only able to approximate linear behaviour.

The third and last component, the *activation function*, remedies this by passing the linear transformation at each neuron through a nonlinear function, called the *activation function*, $a(\cdot)$. The linear combination (2.16) is passed through the activation function at each neuron, meaning the full vector-valued formulation of a layer becomes:

$$\boldsymbol{f}_l(\boldsymbol{x}) = a(\mathbf{W}_l\boldsymbol{x} + \boldsymbol{b}_l) \tag{2.17}$$

Where the sum of all the neurons' linear functions previously resulted in a linear function, simply performing linear regression, now having introduced nonlinearity at every neuron makes the resulting function (2.17) nonlinear. More specifically, the introduced nonlinearity can be understood to make each neuron's linear function active for some regions of input and inactive for others. The sum of such variously active linear functions creates a manifold, which, with appropriate parameters (weights and biases) shaping the linear functions, may approximate nonlinear functions. An example is provided in Figure 2.3. It has been shown that the MLP can learn its parameters such that an arbitrary nonlinear function may be approximated with as little as only one hidden layer[7], given a sufficient amount of hidden neurons.

There exist many different activation functions, such as the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$, and the hyperbolic tangent function, $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$. Maybe the most influential of today, however, is the *rectified linear unit* ($ReLU$), defined as:

$$ReLU(\boldsymbol{x}) = \max\{0, \boldsymbol{x}\}, \tag{2.18}$$

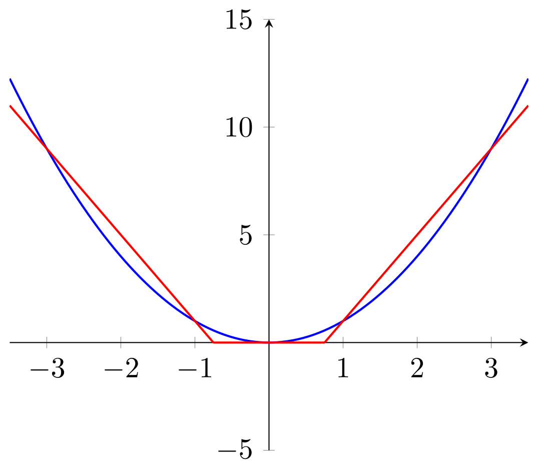


Figure 2.3: A second-order polynomial and three linear functions approximating it. Illustration is fetched from [19].

where vector-valued input is passed through $ReLU$ element-wise.

$ReLU$ usually provides great empirical results and is often the default choice of activation function in neural network design[17]. One disadvantage of the $ReLU$ is that any negative semi-definite input value will result in a zero-output. This may cause issues during training, as later elaborated in Section 2.3.3. That the zero-output of $ReLU$ may prove problematic has motivated variants that avoid the issue of zero-output for negative semi-definite input values.

One important such variant is the *leaky rectified linear unit* ($LReLU$), first introduced by [29], which avoids dying neurons by slightly modifying (2.18), such that negative input-values are mapped to non-zero outputs:

$$LReLU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{2.19}$$

where the so-called *leak rate* $\alpha > 0$ is a value which must be determined by the user. In [29], the authors propose the value $\alpha = 0.01$, such that $LReLU$ has a non-zero gradient also for negative input-values, while still closely mimicking $ReLU$. Which value for α works best must however be decided on a case-by-case basis.

2.2.3 MLP-based function dynamics approximation

A system's discrete-time dynamics may be represented on the standard state space format:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k) + \boldsymbol{w}_k \\ \boldsymbol{y}_k &= \boldsymbol{x}_k \end{aligned} \tag{2.20}$$

where $\boldsymbol{x}_k$ is assumed directly mapped to $\boldsymbol{y}_k$. Though any system usually must be assumed affected by some noise $\boldsymbol{w}_k$, this thesis makes the simplification of not considering the noise in further derivations, $\boldsymbol{x}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k)$. This is done, as the system basis for the implementations in chapter 3 suffers no noise - see Section 3.1.3. Reformulating to match the format of the basic model in (2.13), we then get a prediction model on the following form:

$$\hat{\boldsymbol{y}}_{k+1} = \hat{\boldsymbol{f}}(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{2.21}$$

Recalling that any nonlinear function may be approximated by an NN, it is reasonable to assume that e.g. an MLP may suffice in system dynamics approximation if given the state and current input, $\boldsymbol{x}_k$ and $\boldsymbol{u}_k$.

Note that any non-Markovian system experiences transient responses. A static relation between input and output is not sufficient to capture such transient responses, and so the state vector $\boldsymbol{x}_k$ must be formulated to reflect this by including data far enough into the past to reflect the lengths of any transient effects present. Such data is well encapsulated in a *nonlinear autoregressive model with exogenous inputs* (NARX) state-formulation:

$$\begin{aligned} \boldsymbol{x}_k \triangleq [&y_{0,k}, y_{0,k-1}, \cdots, y_{0,k-m_y}, \\ &\quad\vdots \qquad\qquad , \\ &y_{n_{out},k}, y_{n_{out},k-1}, \cdots, y_{n_{out},k-m_y}, \\ &u_{0,k-1}, u_{0,k-2}, \cdots, u_{0,k-m_u}, \\ &\quad\vdots \qquad\qquad , \\ &u_{n_{in},k}, u_{n_{in},k-1}, \cdots, u_{n_{in},k-m_u}], \end{aligned} \tag{2.22}$$

Defining an MLP's input layer to take in data in the form of the NARX-state vector (2.22) as well as the current input values, $\boldsymbol{u}_k$, implements a neural NARX (NNARX) prediction model. Such an NNARX prediction model yields single-step prediction of the output of a discrete-time system's dynamic behaviour.

Note that, even though NARX formulation is static - in that the NARX state vector (2.22) takes in all values simultaneously - it is tightly linked to the system's corresponding state space formulation of the system's dynamics, as detailed in [6] (eq. (4)-(7)). However, for the NNARX prediction model to predict accurately by capturing all transient behaviour, the (2.22) must contain data far enough into the system's past to capture all *still* relevant information regarding actions performed on the system.

Alternatively to taking in all relevant past data simultaneously, as described above, a model may instead be designed cumulatively, in the sense that it gathers all relevant data during application, and all predictions are valid once it has accumulated data from many enough steps that all transients are accounted for. This may be implemented by *recurrent neural networks* (RNNs), and is covered in more detail in Section 2.4.

2.3 Training neural networks

This section is based on material found in [17], and presents the basic theory of training deep learning models, with some special emphasis on the theory relevant to training NNs for the purpose of regression. Though Section 2.3.1 is *based on* similar sections from [19], it is fully reworked and expanded to accommodate the wider academic scope in this thesis. The same applies to Section 2.3.4.

2.3.1 Training in deep learning

As indicated by (2.13), a model will always be an approximation of the true process it models. Deep learning algorithms are eager learning algorithms, implying that the user may alter the algorithm and its behaviour by providing data. Since the goal is to model a process as accurately as possible, the process of training a model may be summarized as the process of altering its parameters such that the gap between it and the true process is minimized.

After having defined the task to be solved by the NN, e.g. regression, its structure may be determined by specifying its inputs and outputs, with which the data must match. More specifically, the data used in training is sampled from the true process and is always coupled with a *label*, which describes the true output from the process given the input represented by the data. The data is then used as input, and the labels to judge the performance of the resulting output. Such data may for instance be actuation-data for a dynamic system, coupled with the corresponding measured outputs as labels.

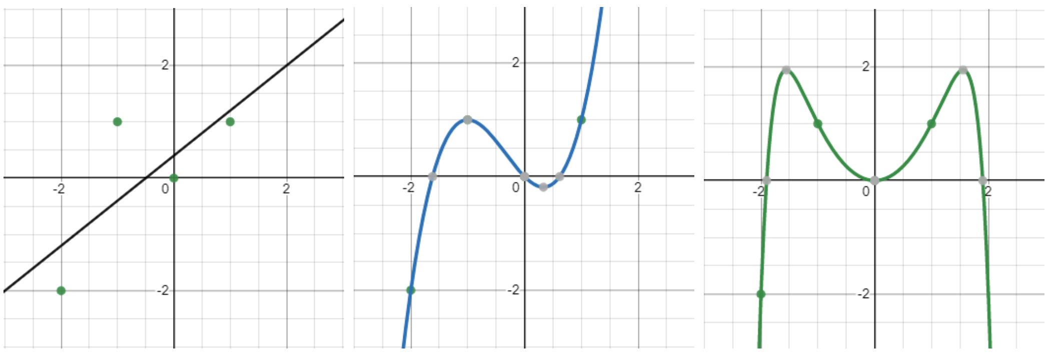
Since the true process may be subject to disturbances and uncertainties, the samples collected from it will be distributed around the true process with stochasticity dependent on said disturbances and uncertainties. This distribution, $p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})$, is called the *data-generating process*, and outputs the labels $\boldsymbol{y}$ given input-data $\boldsymbol{x}_{in}$. Though the type of distribution depends on the specific system, the expectation value may still be stated generally, as:

$$\mathbb{E}[p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})] = \boldsymbol{f}(\boldsymbol{x}_{in}) \tag{2.23}$$

This implies that driving a model to best model this process means shaping its parameters such that it implements a function $\hat{\boldsymbol{f}}$ which most closely matches $\boldsymbol{f}$, as previously indicated in (2.13). If done successfully, the model will be able to predict samples drawn from $p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})$ accurately, even if they were not part of the previously seen data in the training data set; the model has then achieved *generalization.*

Presently, considerations regarding requirements on the model, shaping the training process and requirements on the data sets are presented.

Capacity. An NN's goal is to predict the value of the samples drawn from some data-generating process, $p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})$. As indicated by (2.23), these samples' expectation value may be described by some function, whose degree of nonlinearity depends on the true process described by $p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})$. The NN's ability to attain some degree of nonlinearity to accurately model such a function is called its *capacity*; an NN with low capacity is only able to capture low degrees of nonlinearity, whereas an NN with high capacity is able to capture high degrees of nonlinearity. Figure 2.4 illustrates this, presenting data points sampled from a noise-free third-order polynomial, and different approximations of the underlying data-generating process. Figure 2.4a illustrates the case where an NN has too low capacity, and very poorly models the underlying process, Figure 2.4b, in which case the NN is considered *underfitted.* Conversely, Figure 2.4c illustrates the case where an NN has quite high capacity relative to the underlying process. Note that, even though



(a) Underfitting a 1st-order polynomial to the data-points. (b) Correctly fitting a 3rd-order polynomial to the data-points. (c) Overfitting a 10th-order polynomial to the data-points.

Figure 2.4: The illustration shows three different polynomials of order 1, 3 and 10, respectively, all performing regression to fit 4 data points to a function. The points are drawn from a 3rd-order polynomial. The illustrations are fetched from [19].

this model correctly fits all data points present in the data set, it would not generalize to any new, previously unseen sample drawn from the underlying process. Thus, the model may intuitively be described as overdetermined, as it represents only one of many functions that will correctly fit to the data points, but not generalize. In the regions around the data points, the model is more nonlinear than the underlying process; the NN is *too* nonlinear and is considered *overfitted*. Thus, the *optimal capacity* is neither too low nor too high, but rather that which matches the degree of nonlinearity seen in the true, underlying process.

The NN's capability of universal approximation, given enough hidden neurons - see Section 2.2.2 - directly gives that an NN's potential capacity is determined by its structure. The structure is determined by parameters such as the amount of hidden layers and neurons. These are called *hyperparameters*, as they are determined on a higher level than the parameters within the actual model. The degree of nonlinearity achieved depends on the execution of the training, though actually measuring the degree of nonlinearity in a model after training is non-trivial. For discussions regarding measures of nonlinearity, the interested reader is referred to [30].

Note that a model may, in theory, become perfectly fitted to any problem, if the training set is exhaustive of all possible samples. Modeling digital logic, such as the XOR-function[17], is an example of this. However, for continuously-valued problems of regression, any finite set of samples will never be exhaustive with respect to the problem, and overfitting should be avoided, as it will only lead to great performance on the training data set, but not great performance in general.

Training, validation and testing. The matter of finding the parameters providing the ideal model must be addressed. Then, a measure of what is optimal must be determined. By defining a metric measuring the model's error during training, E_{train}, variable with respect to the model's parameters, its gradient may be derived and used to seek the optimum, which then provides the optimal set of parameters. In the case of multidimensional regression tasks, E_{train} is often defined to be the mean of the *mean square error* (MSE):

$$E_{\text{train}}(\boldsymbol{\theta}; \mathbb{X}_{\text{train}}) = \frac{1}{I}\sum_{i=1}^{I} ||\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i||_2^2 = \frac{1}{I}\sum_{i=1}^{I} ||\hat{\boldsymbol{f}}(\boldsymbol{\theta}; \boldsymbol{x}_i) - \boldsymbol{y}_i||_2^2, \tag{2.24}$$

where $\mathbb{X}_{\text{train}} \triangleq \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\} \ \forall \ i \in [1, ..., I]$ is the invariant set of training data with I sample-label pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, passed as a set of parameters. Each sample-label pair i contains a vector-valued sample of J elements, $\boldsymbol{x}_i = [x_{i,0}, x_{i,1}, ..., x_{i,J}]$, and the corresponding labels $\boldsymbol{y}_i = [y_{i,0}, y_{i,1}, ..., y_{i,J}]$. $E_{\text{train}}(\boldsymbol{\theta}; \mathbb{X}_{\text{train}})$, from here-on referred to as simply E_{train} for simplicity, then describes the mean of the squared euclidean distance between the predicted output $\hat{\boldsymbol{f}}(\boldsymbol{\theta}; \boldsymbol{x}_i) = \hat{\boldsymbol{y}}_i$ and the true output $\boldsymbol{y}_i$ over *all* samples $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ in the data set, as a measure of how *close* the predictions come to the ground truths in general. In this case, the optimum is the minimum, as lower values of E_{train} imply better performance of the model.

Containing the term $\hat{\boldsymbol{f}}(\boldsymbol{\theta}; \boldsymbol{x}_i)$, E_{train} is highly nonlinear with respect to $\boldsymbol{\theta}$, i.e. also non-convex, and no general closed-form solution for its optimum can be assumed to exist. In practice, this often means that E_{train} will have several local minima. While finding the global minimum would supply the ideal solution, any decrease in E_{train} still means a better-performing set of parameters. Thus, the problem instead becomes one of optimization. Finding the optimum is nevertheless non-trivial, due to the nonlinearity of E_{train}. Optimization techniques tailored to machine learning have been derived, and are covered in more detail in Section 2.3.2.

So far presented, the problem of training in machine learning is simply a problem of unconstrained optimization by iteratively seeking the optimum. Each iteration is called an *epoch*. However, the goal of finding the optimum during training as described above, is in reality two-fold; as previously mentioned, in addition to minimizing E_{train}, we want the resulting model to be able to *generalize*. As stated in [17](p. 108):

> "What separates machine learning from optimization is that we want the generalization error, also called the test error, to be low as well."

The generalization error E_{gen} is in general not available, as it would have to be computed over every theoretically possible sample - infinitely many for continuously-valued problems. Thus, we approximate E_{gen} with an alternative metric, $E_{\text{gen}} \approx E_{\text{test}}(\boldsymbol{\theta}; \mathbb{X}_{\text{test}})$ (from here-on simply "E_{test}"), being the measure of how well a trained model performs on previously unseen data $\mathbb{X}_{\text{test}}$. This approximation holds under the assumption that the test data set $\mathbb{X}_{\text{test}}$ contains only samples distributed according to the data-generating process $p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})$. Since the ideal case would be $E_{\text{train}} = E_{\text{test}}$, E_{test} is defined equivalently to E_{train} as the mean of the MSE, variable with respect to the model parameters, given $\mathbb{X}_{\text{test}}$. In reality we instead have $E_{\text{train}} \approx E_{\text{test}}$, where $E_{\text{train}} < E_{\text{test}}$, since the training data set can not contain infinitely highly resoluted information about $p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})$. Given that $\mathbb{X}_{\text{train}}$ and $\mathbb{X}_{\text{test}}$ are identically and independently distributed (i.i.d.) according to $p_{\text{true}}(\boldsymbol{y}|\boldsymbol{x}_{in})$, as well as completely disjoint, the assumption that E_{test} is a measure of the model's generalization capabilities is viable[17], reasons for which are elaborated further below.

Since $\mathbb{X}_{\text{train}}$ and $\mathbb{X}_{\text{test}}$ must be i.i.d. and disjoint, E_{train} and E_{test} must be assumed differently shaped, meaning their minima will not lie at the same points. Consequently, during training, both E_{train} and E_{test} tend to decay up to some point, after which E_{test} starts rising while E_{train} continues decreasing. This causes E_{train} and E_{test} to diverge from one

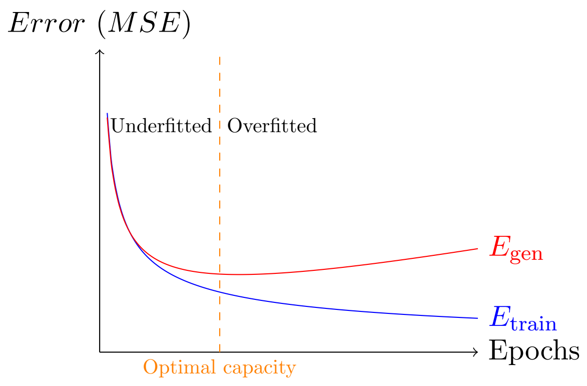


Figure 2.5: An illustration of how generalization error and training error develop over time, sinking initially, before E_{gen} starts rising again, diverging from E_{train}.

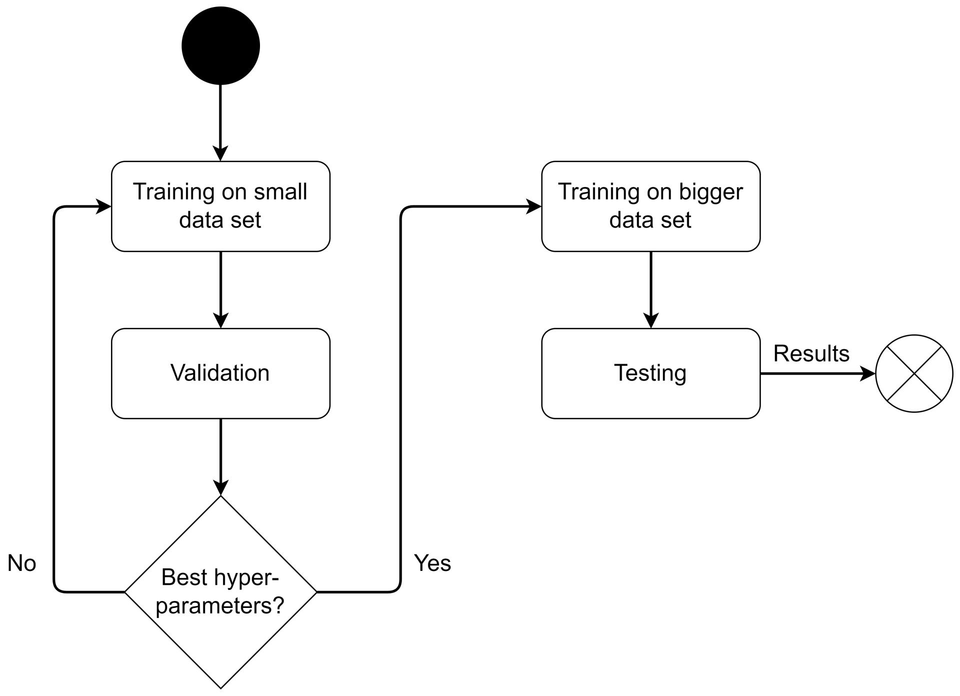


Figure 2.6: The procedure of training an NN.

another, illustrated in Figure 2.5. While E_{train} continues to decrease, indicating that the model eventually becomes nearly perfectly fitted to the samples in $\mathbb{X}_{\text{train}}$, this does not indicate a decrease in E_{test} as training goes on. Instead, the parameters $\boldsymbol{\theta}^*$, for which a minimum is found for E_{train}, give rise to a sufficiently different NN than the parameters at the optimum for E_{test} would. $\boldsymbol{\theta}^*$ then implement an overfitted model, analogous to the simplified case illustrated in Figure 2.4c; the model has low error for the training data, but can not be assumed to perform well on previously unseen data. Since E_{test} is the best approximation available for E_{gen}, and generalization is the goal, it follows that training an NN for indefinite times is not desirable.

Avoiding overfitting by reducing E_{test} may be achieved by regularization. This is looked into in Section 2.3.4.

Training, as currently described, does not directly address the matter of the model's structure, defined by hyperparameters such as the amount of hidden layers and the amount of neurons. Finding the ideal set of hyperparameters is done by defining different candidate sets, and performing the training process for each of them. Each resulting model's predictive accuracy, measured by the validation error E_{val} over a data set $\mathbb{X}_{\text{val}}$, may be calculated in the exact same fashion as the previously described E_{test} as a measure of the model's ability to generalize. The chosen model is that which minimizes E_{val}.

If $\mathbb{X}_{\text{train}}$ is large enough to cause prohibitively long computational times during the search for hyperparameters, a smaller subset may be used instead, since the procedure is performed only to isolate a set of hyperparameters likely to later yield good results during training, not to train the final model. The process of determining both hyperparameters and parameters of a model then becomes as described by Figure 2.6. Note that the requirements of i.i.d. and disjointness also apply between $\mathbb{X}_{\text{train}}$ and $\mathbb{X}_{\text{val}}$.

Since E_{val} is used to optimize the structure for which the model yields the lowest E_{train}, the model is fitted not only to $\mathbb{X}_{\text{train}}$, but partly also to $\mathbb{X}_{\text{val}}$. Thus, most often, the following holds:

$$E_{\text{train}} < E_{\text{val}} < E_{\text{test}} \approx E_{\text{gen}}$$

Indicating the need for a separate testing phase after determining the optimal hyperparameters, as illustrated in Figure 2.6.

Optimization: stochastic gradient descent. The material here presented regarding stochastic gradient descent is heavily based on material found in [17], chapters 4 and 8.

Since training in machine learning is a problem of unconstrained optimization, a popular way of tackling the problem is to reduce it to finding the gradient of the cost function with respect to the model parameters, $\boldsymbol{\nabla}_{\boldsymbol{\theta}} E_{\text{train}}(\boldsymbol{\theta}; \mathbb{X}_{\text{train}})$; the negative gradient directly gives the direction which lowers the cost function value the most. Once determined, a step of some size may be made in that direction in order to lower the cost function value. This process is repeated until the difference in the cost function between each epoch e, $\Delta E_{\text{train}} \triangleq E_{\text{train}}(\boldsymbol{\theta}_{e-1}; \mathbb{X}_{\text{train}}) - E_{\text{train}}(\boldsymbol{\theta}_{e}; \mathbb{X}_{\text{train}})$, is reduced below some user-defined threshold value τ, or the training has gone through the designated amount of epochs. This method of optimization is called *gradient descent*[17] (GD). Numerically deriving the gradient is somewhat elaborated in Section 2.3.2.

Finding the gradient for E_{train} can be very costly for a large data set $\mathbb{X}_{\text{train}}$. Though the gradient will be more precise if calculated over all samples in $\mathbb{X}_{\text{train}}$, the average of the gradients of some randomly chosen subset of the full set of samples is an unbiased estimator for the gradient of the full data set[17]. Thus, the computational demand of the gradient-based optimization may be reduced vastly by calculating the gradient with respect to only some subset of the full data set; a *mini-batch*, the size of which is called the *batch size*. Since the subset must be chosen uniformly randomly among the full data set, this variant of gradient descent is called *stochastic gradient descent* (SGD).

Note that the estimated gradient will most likely deviate somewhat from the true gradient it estimates. While some precision is lost, the optimum may nevertheless be found faster, as simply training for more iterations may compensate for the lost precision. Training for more iterations becomes feasible when the computational demand of each iteration is vastly reduced by the batch size being significantly smaller than the size of the full data set[17].

Algorithm 1: Stochastic Gradient Descent.

```
Input: training data set $\mathbb{X}_{\text{train}}$, learning rate $\epsilon$, batch size $\beta$, initial parameter
       values $\boldsymbol{\theta}_0$, threshold for required improvement at each step $\tau$ or maximal
       amount of epochs $e$
Output: Optimal set of parameters $\boldsymbol{\theta}^*$
$k \leftarrow 0$;
$\Delta E_{\text{train}} \leftarrow \infty$;
while $\Delta E_{train} > \tau$ or $k > e$ do
    $k \leftarrow k + 1$;
    Randomly sample a subset of from $\mathbb{X}_{\text{train}}$: $\mathbb{X}_{\text{mini-batch}} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), ..., (\boldsymbol{x}_\beta, \boldsymbol{y}_\beta)\}$;
    $\nabla_{\boldsymbol{\theta}} E_{\text{train}}(\boldsymbol{\theta}; \mathbb{X}_{\text{mini-batch}}) \leftarrow \frac{1}{\beta} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{\beta} ||\hat{\boldsymbol{f}}(\boldsymbol{\theta}; \boldsymbol{x}_i) - \boldsymbol{y}_i||_2^2$;
    $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \epsilon \nabla_{\boldsymbol{\theta}} E_{\text{train}}(\boldsymbol{\theta}; \mathbb{X}_{\text{mini-batch}})$;
    $\Delta E_{\text{train}} \leftarrow E_{\text{train}}(\boldsymbol{\theta}_{k-1}; \mathbb{X}_{\text{mini-batch}}) - E_{\text{train}}(\boldsymbol{\theta}_k; \mathbb{X}_{\text{mini-batch}})$;
end
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}_k$
```

Since E_{train} is nonlinear, it is not given that a step in the direction of the negative gradient will result in a decrease of function value for any arbitrary step size. In fact, without further measures, convergence is not even guaranteed at the optimum, since the gradient computed in algorithm 1 is noisy, due to the samples being sampled randomly. Consequently, the learning rate must be chosen adequately. In most modern training contexts, this is performed automatically by the algorithm used, typically a variant of SGD (see further below).

In addition to the learning rate, variants of SGD may employ the concept of *momentum*. The idea is that previous iterations' gradients should carry over to the current iteration to some degree, affecting the direction of the current iteration's gradient with some level of "momentum". This is a way of reducing the amount of noise introduced by the random sampling at each iteration, as the final direction of the step made for the parameter update becomes an exponentially decaying moving average of the history of gradients.

There exist many variations of SGD. Some notable examples are *AdaGrad*, *RMSProp* and *Adam*, based on different ways of adaptively choosing the learning rate and implementing momentum. Adam implements, in addition to adaptive learning rates specific to each parameter, adaptive moments - from which the name derives - estimating the mean and the variance of the gradient, respectively. While the moments are implemented as exponentially decaying moving averages, what sets Adam apart is that it uses *both* the first and second moments, as well as bias-correcting them over time. This has shown improvements in performance over predecessors, such as RMSProp[31].

Their specific details surpass the scope of this thesis, and the interested reader is instead referred to [17](chapter 8) for further information.

Requirements on the data. Qualitative data is as important as large amounts of data. Specifically, three main requirements are set for data sets used in training models.

Firstly - as previously argued - the data sets must all be *identically and independently distributed* (i.i.d.), i.e. all samples must stem from the same data-generating process and be independent from each other. Given that all data sets stem from the same data-generating process, then a model trained on such data will learn patterns also present in the test data, yielding increased predictive accuracy for previously unseen data, i.e. improved *generalization*. Conversely, if all data is *not* identically distributed, a well-executed training procedure will still result in a model that is unable to predict accurately for the test data set, as it will have learnt different patterns than the ones present in the test data. Furthermore, if the samples are correlated, then the model will not be able to predict *isolated* samples drawn from the data-generating process.

Secondly, all data sets must be strictly *disjoint*. Recall that the goal of training is to minimize the model's generalization error E_{gen}, which may only be estimated by using previously unseen data. Thus, if the data sets are not disjoint, and data from either the validation and/or test sets are included in the training data set, then later using the validation and test sets to measure the model's ability to generalize will not be a true test of generalization, and of no actual interest. Knowing to some degree the model's predictive accuracy is important, even if the available data is limited. While the specific ratios may be subject to tuning, the split of data into 70% for training, 15% for validation and 15% for testing is one commonly used option - see for example[16].

Lastly, in the case of regression, normalizing the labels prior to training is important to avoid that different outputs scale E_{train} differently if they are of different magnitude; it is the error *relative* to each output that is interesting. An example is provided by application in Section 3.3.1.

A final, less strict, but nevertheless important aspect: the quality of the data is important. The procedure of training can be intuitively thought of as algorithmically teaching each layer in the model which patterns and features exist in the inputs and carry the most important information with respect to the model's output and shaping the parameters thereafter, such that these features are detected. It is then important that the data-generating process from which the data is sampled does not carry patterns irrelevant to the task the model is designed to solve. If $\mathbb{X}_{\text{train}}$ contains patterns between data and labels that do not generally apply outside $\mathbb{X}_{\text{train}}$, the model will generalize poorly, as the features in $\mathbb{X}_{\text{train}}$ will not be generally applicable. Generally: If all data-label pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ used in training, validation and testing are sampled from a non-general data-generating process, i.e.

$$p_{\text{general}}(\boldsymbol{y}_i|\boldsymbol{x}_i) \neq p_{\text{data sets}}(\boldsymbol{y}_i|\boldsymbol{x}_i), \tag{2.25}$$

then the results during training may be good, but the model will still perform poorly once exposed to general data. A notable example from research is that in which a model was trained to distinguish between wolves and huskies in pictures. All training pictures containing wolves also contained snow, whereas all pictures containing huskies contained no snow. When asked to classify a picture of a husky containing snow in the background, the model incorrectly classified the picture as a picture of a wolf, as it had "learnt" from the poorly chosen data that patterns of snow indicated a wolf. Effectively, the model had learnt to detect snow in the background of pictures[32].

Also regarding the quality of the data: it is important that the data set is balanced in its representation of the data-generating process' behaviour. Less represented behaviours will be learnt less effectively.

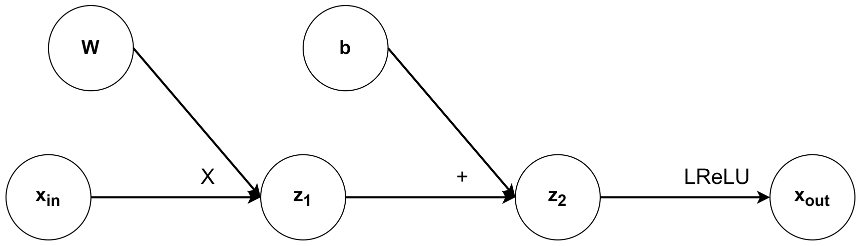


Figure 2.7: The computation graph for the operations being applied on *one* input in any generic node. The input x_{in} is multiplied with a weight w, resulting in an intermediate placeholder variable z_1, which is added together with a bias b, creating another intermediate placeholder variable z_2, which is finally passed through the activation function - in this case *LReLU*. The result of the computations results in x_{out}. Note that the final output of the node is the sum of such operations applied on *all* the inputs to the node, which is omitted from this illustration for simplicity.

2.3.2 Finding the gradient and backpropagation

In order to perform SGD during training, the expression for the cost function's gradient with respect to the model's parameters, $\boldsymbol{\nabla}_{\boldsymbol{\theta}} E_{\text{train}}(\boldsymbol{\theta}; \mathbb{X}_{\text{train}})$, must be derived explicitly. Although this in theory may be done manually, the potential amount of parameters contained within the vector $\boldsymbol{\theta}$ would quickly prove the process prohibitively time-consuming. This is especially true for a testing procedure, such as in Figure 2.6, where testing for hyperparameters causes many different structures to be evaluated during one training process, and each structure would have its distinct corresponding expression for the cost's gradient. Instead, ways of finding the gradient algorithmically have been derived. This thesis briefly explains the method of *backpropagation*[33], which has been implemented in open-source libraries for machine learning in Python, such as PyTorch[34].

Backpropagation is a way of using the calculated value of the cost function during one step of SGD, to calculate what effect changes in $\boldsymbol{\theta}$ would cause in the cost function value. The algorithm is based on the chain rule for derivatives in calculus. More specifically, since NNs are convolutions of layers - see (2.14) - any change to the parameters in early layers affect the numerical outcomes in the consequent layers. As shown in (2.17), each node in each layer applies a set of operations on the input. In order to trace the effect from each variable's value and each associated operation on the input, an NN may be viewed as a *computation graph.* A computation graph is implemented as a directed acyclic graph (DAG) with nodes holding variables and connections between nodes indicating operations[17]. A small example of a computation graph, illustrating the operations performed on one input to a single, generic node, is presented in Figure 2.7.

By decomposing the full model into a computation graph, each node may be seen as a function of all parts of the computation graph that eventually lead into it. Then, each node's gradient may be formulated by means of the chain rule, applied to all elements in the part of the computation graph that leads into it, which in turn leads to the full expression for the cost function's gradient. Finding the specific gradient at each node may be done by either symbolic or automatic differentiation. There are several different implementations of backpropagation; the implementation later employed in this thesis, see Section 3.3.2, is that of PyTorch, which uses automatic differentiation, details of which are beyond the scope of this thesis. For specific details regarding the numerical considerations in deriving the gradient, the interested reader is referred to [35].

Note that, while especially suited for finding the gradient of models, backpropagation is in fact entirely general, and may be used to numerically derive any function's derivate[17].

2.3.3 Gradient-related issue: ReLU and neuron death

Though ReLU usually provides great performance, it suffers the weakness of *neuron death.* By neuron death is meant the phenomenon that a neuron's parameters attain values during training, which drives its output to a negative value, and consequently it outputs only 0. When a neuron's output is always 0, the gradient of the cost function with respect to that neuron's associated parameters is also always 0. Since the gradient becomes 0 with respect to that set of parameters, they will no longer be altered during further training. Thus, the neuron has stopped learning and effectively died.

2.3.4 Regularization

Due to the fact that a reduction in E_{train} does not necessarily give a reduction in E_{test}, methods have been devised to lower E_{test} specifically, namely *regularization* methods [17]. As there exist many methods of regularization, this thesis focuses only on the ones utilized in Section 3.3.2: *early stopping* and *weight decay.*

Early stopping. Early stopping is meant to specifically remedy the issue of the overfitting that occurs when training for extended periods of time. This is done, as the name suggests, simply by terminating the training early, even if the designated amount of epochs has not passed. Specifically, early stopping terminates the training at the point of optimal capacity - at the minimum of E_{val} for hyperparameter searching or E_{test} for training the final model.

In practice, E_{val} is not guaranteed to develop monotonously; some iterations of training may yield higher values for E_{val} than others, even if the trend is a decreasing value, as will later be exemplified in Figure 4.8. This can lead to several local minima for E_{val} as the amount of epochs increases. Thus, the concept of *patience* is introduced to the early-stopping scheme. Patience defines how many iterations the training should continue after the lowest value in E_{val} has been recorded, in order to avoid stopping early due to "flukes".

Weight decay. Weight decay is a regularization method designed to incentivize the training process to approach lower-valued parameters and thus reduce overfitting by injecting some added variance to the model's certainty of its predictions over the data set during training. This is achieved by adding a quadratic term with respect to the parameters, scaled with a weight decay coefficient α, to the training error. Extending the error proposed earlier, (2.24), we get:

$$E_{\text{train}}(\boldsymbol{\theta}; \mathbb{X}_{\text{train}}) = \frac{1}{I}\sum_{i=1}^{I} ||\hat{\boldsymbol{f}}(\boldsymbol{\theta}; \boldsymbol{x}_i) - \boldsymbol{y}_i||_2^2 + \lambda\frac{1}{2}\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\theta} \tag{2.26}$$

The scaling factor $\frac{1}{2}$ is included for simplicity in the expression of the resulting gradient, and makes no difference in the outcome of the algorithm, as its presence, or lack thereof, is trivially compensated for by setting λ accordingly.

The added term causes a linear penalty on the gradient for increasing parameter values. Informally, for functions quadratic with respect to the parameters - such as (2.26) - the weight decay coefficient λ expresses an *added variance*, which alters the optimum with respect to the parameters. The added variance serves to reduce overfitting, as the model is forced to become less "certain". Recall that the goal of regularization is reducing $E_{\text{gen}} \approx E_{\text{test}}$, even if at the cost of an increased E_{train}. This explanation of the effect of weight decay is an informal one and given as such for the sake of brevity, as the mathematical details are non-trivial. The interested reader is referred to [17] (7.1.2, p. 227) for the specific mathematical details.

2.4 Recurrent neural networks

2.4.1 General remarks on recurrent neural networks

In broad terms, *recurrent neural networks* (RNNs) are a class of NNs that are structurally tailored to both processing and outputting sequences of data. While this makes them especially suited and popular for e.g. language modelling, they can also be applied to system dynamics prediction.

While an MLP may be easily designed to predict system state one step ahead - see Section 2.2.3 - they may also be designed to both take in and output larger sequences of data; an MLP's input and output layers may be designed and trained such that the model outputs the sequence of the next predicted N system state values, or just the system state N steps ahead. However, this grants little flexibility to alter the value of N after the model has been specified, as any MLP is fixed to tasks taking in and outputting data on specifically the specified format. MLPs have strongly limited flexibility with respect to sequence length, and are thus not considered especially suited to processing sequential data[17].

Such generality may instead be achieved by modelling single-step system dynamics prediction (with e.g. an MLP), furthering the resulting outputs' system state information into a new step of dynamics prediction. Structurally, this equals recurring a model's output back into its input. The recurrence may be repeated as many N times as desired, implementing open-loop multi-step system dynamics prediction for that many steps. Throughout the following details of RNN theory, the term "time step" refers to the iteration of recurrence in the RNN; an RNN of N recurrences has time steps $\{k, k+1, ..., k+N\}$. Importantly, N may be chosen freely without loss of validity in the model, as each step contains a model with the same shared parameters as at each other step. While this may resemble a simply very deep MLP, this type of network still differs from an MLP due to the shared parameters at each chained instance. Note that even though the model is equally valid at each time step ahead, it must be assumed to contain some modelling error. Any such mismatch between the true system and the model may accumulate the farther into the future the predictions are made - unless the mismatch is proven to have an expectation value centered exactly on the true system's expectation value. This is further discussed in Section 5.2. Nevertheless, the recurrent structure enables modelling of both sequences of *arbitrary length* as well as single values *arbitrarily far* into the future.

In the described case of multi-step system dynamics prediction, only the output from each time step is passed on from one time step to the next, effectively discarding the latent

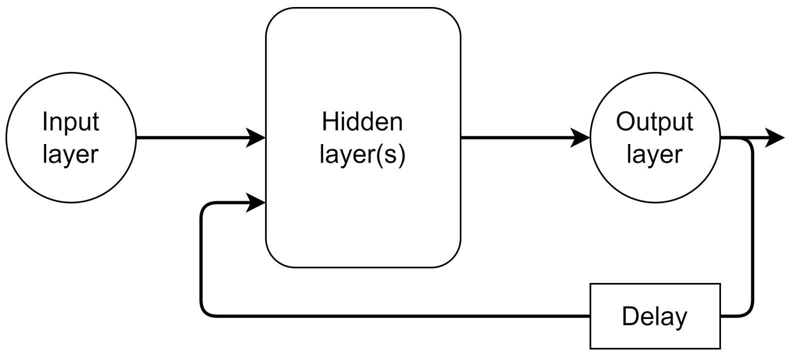


Figure 2.8: The Jordan network, illustrated in a simplified fashion with general presentations of the layers for simplicity.

information within the hidden layers. This yields a structure as illustrated in Figure 2.8, and is commonly called a *Jordan network* after it was described in [36].

Such RNNs are vulnerable to loss of information at each step; data entering the network through the input must always pass through the output layer before recurring back as input to the hidden layer(s). Unless specifically designed to do so, the output layer carries no guarantee that no information is lost by reducing the latent information within the hidden layers to the desired format of the output. Building on the example of language modelling: if a model is designed to predict the next word in a sentence, it may output the same word for many different sequences of input. Thus the output information can not unambiguously be reversed to latent information, and the output layer carries strictly less information than the input and hidden layers. Though this loss of information may not be an issue, depending on the use-case, it is an important observation, as it may mean that the model in practice has less available data than may be desirable.

The alternative case is that in which information is passed directly between the hidden layers across time steps. This still allows sampling the output for each step, but does not require the information within the network to be filtered through the format of the output before it is furthered to the next time step. Then, input data never actually leaves the network, as any data that is fed into the network will continue circulating within the hidden layer(s) across time steps. This implies that all information present in the data is in theory available at any arbitrary time step. Returning to the language modelling example: the next word in a partially-finished sentence is dependent on not singularly the word that comes before, but rather a broader context, i.e. the rest of the sentence, or even every piece of text preceding the word to be predicted. Retaining such latent information within the hidden layers across time steps may thus be of utmost importance. This variant of RNN is commonly called an *Elman network* after it was described in [37]. The mitigation of information loss provided by this structure actually makes the RNN universally capable - as described in [17](p.372):

> ”The [Elman network] is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size.”

Training RNNs is done with the backpropagation algorithm, as backpropagation is universally applicable on any function represented as a computational graph. When applied to RNNs, however, some special concerns must be made. Firstly, since the RNN in concept

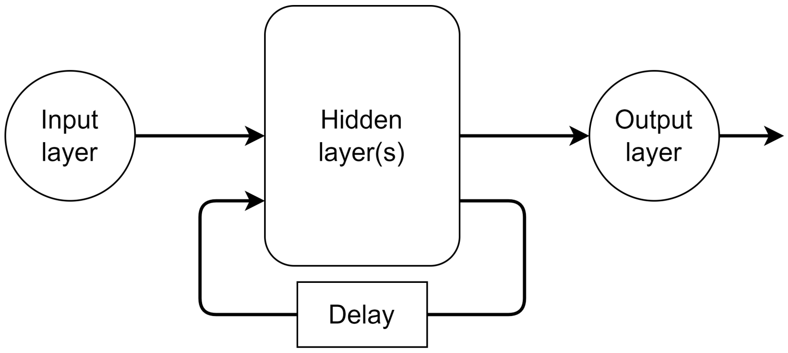


Figure 2.9: The Elman network, illustrated in a simplified fashion with general presentations of the layers for simplicity.

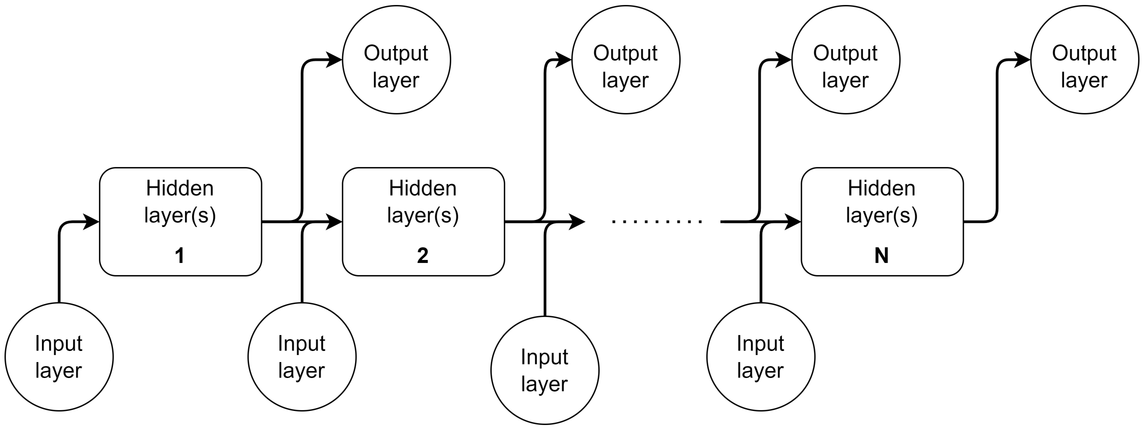


Figure 2.10: An RNN unfolded for N recurrences; the unfolded RNN consists of N MLP-cells.

can recur indefinitely, it must be unfolded for as many iterations of recurrence as is desirable for the modelling task. This turns the RNN into a finite DAG, see Figure 2.10, which may be represented by a computational graph. Importantly, however, the shared parameters of the original RNN have now caused the same parameters to repeat throughout the DAG. As previously explained, this is a desirable quality, which should remain also after the training. Secondly, BPTT must thus ensure that the parameters change in unison during training, such that they remain equal also after the training. In order to adhere to this requirement, the gradients with respect to each parameter must be defined equally for each repetition of the individual parameter. When respecting this requirement, BPTT is in truth simply an application of the generalized backpropagation algorithm, as derived and presented in [38], and the gradient may be calculated accordingly. A concise presentation of the generalized backpropagation algorithm is also presented in [17] (Algorithm 6.5, p. 213), and the more in-depth presentation of the RNN's general gradients' expressions are presented in [17] (10.2.2, p. 380). Note that BPTT is a significantly more resource and computationally demanding algorithm than the simplified backpropagation presented in Section 2.3.2.

The main disadvantage of parameter sharing is that it may easily cause issues in the training of RNNs due to an either *vanishing* or *exploding gradient*. When the computational graph for the network becomes sufficiently deep, and the same weight $w_{i,j}$ (on layer i's input j of each time step) is thus repeated as many times as the number of recurrences N, meaning the model contains weight terms that are exponential with respect to N: $w_{i,j}^N$.

The gradient of the cost function with respect to any such weight will necessarily "vanish" towards 0 or "explode" towards $\pm\infty$ for sufficiently high-valued N when $w_{i,j}$ is sufficiently different from 1. As such, the RNN's ability to learn based on its prediction error deep into the network becomes lessened, as the gradient of the error will become either close to zero or immense once sufficiently deep into the network. A zero gradient causes no further change to parameters, and a too-high-valued gradient causes too-large parameter changes. Both cases sabotage proper parameter learning for deep recurrent networks, effectively sabotaging the RNN's ability to consider long-term dependencies. The issue of a weak understanding of long-term dependency in RNNs is further addressed in Section 2.4.3.

For networks on the Jordan form, the time steps of the RNN may be viewed as disconnected from each other during training; since only the output is passed on between time steps, training may be performed by furthering the label instead of the current time step's prediction. This is called *teacher forcing*, and allows for much quicker training since training does not have to be done on the full recurrent model with BPTT, but instead on the underlying MLP with regular backpropagation[17].

Training by means of teacher forcing does introduce issues with respect to performance. Specifically, the model is taught with respect to the ground truth - the samples presented as inputs to the model are drawn from the distribution of the true data-generating process. Even though the model may predict well for one step, problems arise when chaining single-step predictions: since the model can not be assumed to predict perfectly after training, the distribution of the ground truth can not be assumed to exactly equal that of the predicted data. Consequently, when the model is later applied, and at each time step receives input in the form of its own previous predictions, it is receiving inputs from an essentially different data-generating process, than that of the data on which it was trained. Recalling that a model will not necessarily perform well on data drawn from a distribution not matching that of its training data set, (2.25), it must be assumed that the model will perform worse than observed during training. The gap between data observed during training and application resulting from teacher forcing is called *exposure bias*, and may cause significantly decreased performance[17].

Scheduled sampling[39] has been proposed as a method of mitigating exposure bias from teacher-forcing during training. The method is implemented by gradually replacing labels from the data set with previously predicted values from the RNN itself. Starting from no predicted values as labels and ending with only predicted values as labels, the method teaches the model to predict based on both data from the true process, and incorrect predictions, such that the model will be prepared to predict adequately, also in the face of inaccurate inputs during application.

2.4.2 Recurrent neural NARX-Model

As detailed in Section 2.2.2, single-step dynamics prediction may be achieved by viewing the process as an NARX model and implementing it by means of an MLP, (2.22) and (2.21). As briefly explained in Section 1.2, this is done successfully in [16].

Extending the neural NARX-model for multi-step dynamics predictions may be done trivially by recurring a single-step predicting MLP's output to its input, creating a *recurrent neural NARX-model* (RNNARX) on the Jordan form as in Figure 2.8. By providing the model an initial state $\boldsymbol{x}_k$ for the current time step k, all future dynamics $\hat{\boldsymbol{y}}_{k+i}$ may be predicted for $i \in [1, N]$, given some defined, finite N:

$$\hat{\boldsymbol{y}}_{k+1+i} = \hat{\boldsymbol{f}}_{MLP}(\hat{\boldsymbol{y}}_{k+i:k+i-m_y}, \boldsymbol{u}_{k+i-1:k+i-m_u}, \boldsymbol{u}_{k+i}) \tag{2.27}$$

This predictive structure applied to the oil- and gas well system presented in Section 1.4 is later illustrated in Figure 2.13.

Special regards must be considered for the cases where $i > 1$. Firstly, all $\boldsymbol{u}_{k+i:k+1}$ are future actuation values and must be planned and supplied by the user or e.g. the optimization algorithm during optimization in MPC. Secondly, $\hat{\boldsymbol{y}}_{k+i:k+i-m_y}$ similarly consists of previously predicted values $\hat{\boldsymbol{y}}_{k+i:k+1}$, as well as historical, measured values $\boldsymbol{y}_{k:k+i-m_y}$. An implementation using the model (2.27) must then ensure that the vectors $\boldsymbol{u}_{k+i:k+i-m_u}$ and $\hat{\boldsymbol{y}}_{k+i:k+i-m_y}$ contain the appropriate combination of future and past historical values in the correct sequence, relative to the future time step $k+i, i > 1$.

2.4.3 Gated RNNs

The so-far described RNNs are trivial extensions of the regular MLP; except for the added connections which implement plain recurrence, no architectural changes are made. A class of more advanced RNN architectures is that of the *gated RNNs.* In the way the so-far described RNNs consist of a single *cell* with recurrent connections to itself, so do the gated RNNs. Differently, however: the cells in the gated RNNs contain several more units of computation, each creating and controlling their own part of the total flow of information. While they are all contrived of linear transformations and an activation function - similar to a hidden layer in a regular MLP - they are interconnected with simple mathematical operations, such that the different flows of information are combined. Each unit of linear transformation and activation function is called a *gate*, and may be regarded as a separate hidden layer, parallel to the other gates.

The following paragraphs present two of the most significant gated RNN variants on a relatively superficial level, such as to facilitate an intuitive understanding. This choice of level of detail is made for brevity, as gated RNNs play no *direct* role in this thesis, and only superficial knowledge of these models is relevant to the scope of this thesis as a background for better understanding the survey in Section 1.2. The explanations are based on material in [17]. Also provided are the associated mathematical formulations of the architectures, though [6] (eq. (10)) provides a more concise formulation.

Long short-term memory. The most successful of the gated RNNs is the *long short-term memory* architecture (LSTM) first introduced in [14], which was designed to mitigate the vanishing and exploding gradient problems. It achieves this by structurally providing the model with capabilities of understanding a *context* $\boldsymbol{c}_k$ from the totality of all information that is input over time, which is recurred with no external interference nor being passed through any altering layer. The previously described RNNs suffered information to either explode or vanish, due to the weights causing an exponential effect from being multiplied with the initial input as many times as the number of recurrences. For the LSTM cell, $\boldsymbol{c}_k$ is, due to its unhindered recurrence, *unweighted* between time steps, and the disruptive exponentiality is removed. This may be intuitively viewed as the structure of the LSTM cell allowing its "short-term" memory to be retained for longer periods of time - hence the name. The further paragraphs explain the inner workings of an LSTM cell. For a short-hand visual introduction, refer to the full LSTM cell architecture illustration in Figure 2.11.

Though it can be a strength that RNNs with connections between the hidden layers retain *all* information they ever receive, it is not given that all previous data is of interest for a prediction arbitrarily far into the future. The *forget*, *state candidate*, *input* and *output* gates implement the information flow control within the LSTM cell, which in turn determines the impact each flow of information has in shaping the context $\boldsymbol{c}_k$. As $\boldsymbol{c}_k$ carries the LSTM cell's contextual understanding of the totality of all given input, it is this variable that is the basis for the final output.

The gates are implemented the same way as a standard MLP hidden layer; a linear transformation is applied to its input, a bias is added, and the result is passed through an activation function - see (2.17). The activation function for the state candidate gate may be whichever suits the relevant prediction problem, as it plays the main role of shaping the to-be output - completely similar to the hidden layers of a basic RNN. The input, forget and output gates all instead employ the sigmoid activation function, $\sigma(x) = \frac{1}{1+e^{-x}}$, as they all serve the purpose of determining a factor $f_g \in [0, 1]$ which scales different parts of the information flow in order to shape the impact each of the information flows has on $\boldsymbol{c}_k$ before it is output from the LSTM cell.

The forget gate outputs a factor f_f, determining how much of the context from the previous time step, $\boldsymbol{c}_{k-1}$, should be furthered into this time step. The state candidate gate determines a candidate contextual understanding $\tilde{\boldsymbol{c}}_k$, which serves as an updating term to $\boldsymbol{c}_k$. The input gate outputs the factor f_i, determining how impactful $\tilde{\boldsymbol{c}}_k$ should be on the updated context. Consequently, the updated context $\boldsymbol{c}_k$ is a weighted sum of old and new information:

$$\boldsymbol{c}_k = f_f \boldsymbol{c}_{k-1} + f_i \tilde{\boldsymbol{c}}_k \tag{2.28}$$

The output gate outputs a factor f_o, determining how much of the context should be output from the cell.

The final output of the LSTM cell is denominated the *hidden state* $\boldsymbol{h}_k$, as this variable carries hidden information passed as input along with any externally given input to the next cell of the unfolded LSTM network. This recurrent connection creates a structure analogous to that of an Elman network. The hidden state must be fed through an output layer if it is to be utilized as a model prediction. The form of the output layer depends on the application.

While taking in the $\boldsymbol{h}_k$ alongside $\boldsymbol{x}_k$ allows the state candidate gate to predict $\tilde{\boldsymbol{c}}_k$ the same way a cell of a basic RNN would calculate predictions, it also allows the forget, input and output gates to determine the factors controlling the information flow *adaptively*; the LSTM automatically learns what data to retain and what data to disregard by means of f_f, f_i and f_o as a result of training.

Note that, since all gates take in two variables, each of them may be mathematically described as follows:

$$\boldsymbol{f}_g(\boldsymbol{x}_k, \boldsymbol{h}_{k-1}) = a(\mathbf{W}_g \boldsymbol{x}_k + \mathbf{U}_g \boldsymbol{h}_{k-1} + \boldsymbol{b}_g)$$

where g denominates the specific gate - forget, state candidate, input or output - and a represents the activation function of the respective gate. Consequently, the set of parameters to be trained becomes significantly larger than for a basic RNN:

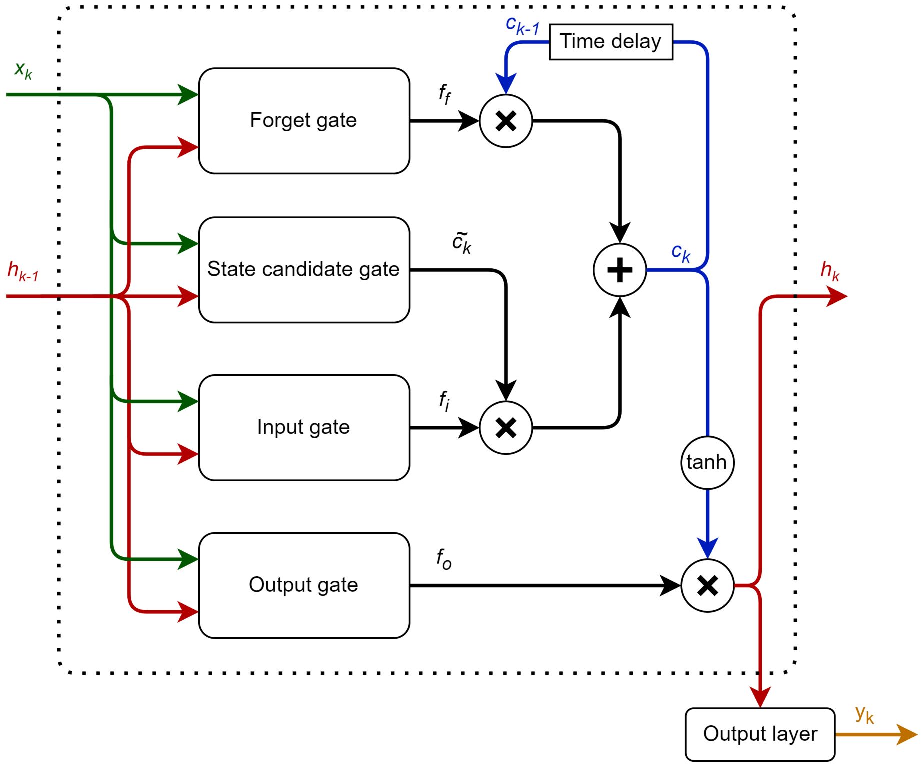


Figure 2.11: An illustration of the flow of information and series of computations occurring within a single cell of an LSTM.

$$\boldsymbol{\theta}_{LSTM} = \{\mathbf{W}_g, \mathbf{U}_g, \boldsymbol{b}_g\}$$

In addition to these twelve parameters - three parameters for each of the four gates - potential parameters of the output layer must lastly be considered.

Gated recurrent unit. The *gated recurrent unit* architecture (GRU), first introduced in [13], is a simplified version of the LSTM. With fewer operations, it seeks to accomplish much of the same as the LSTM, but computationally faster. It achieves this by removing the context variable $\boldsymbol{c}_k$ and using only the hidden state $\boldsymbol{h}_k$ and externally given input $\boldsymbol{x}_k$. Additionally, the forget, input and output gates of the LSTM are replaced by a *reset* and *update* gate, both of which use the sigmoid function as activation function. Similarly to the LSTM, all the gates take in $\boldsymbol{h}_{k-1}$ and $\boldsymbol{x}_k$, except for the state candidate gate, which replaces $\boldsymbol{h}_{k-1}$ with $f_r \boldsymbol{h}_{k-1}$; the factor f_r is produced by the reset gate as a means of controlling the hidden state's prevalence in the calculation of the hidden state candidate $\tilde{\boldsymbol{h}}_k$.

The update gate calculates the two factors f_u and $1 - f_u$. The contribution of $\tilde{\boldsymbol{h}}_k$ is scaled with f_u, and the contribution of the pre-existing $\boldsymbol{h}_{k-1}$ is scaled with $1 - f_u$. This way, less of previous hidden state information is included if the new information is deemed valuable and vice versa, i.e. $f_u > 0.5$ or $f_u < 0.5$, respectively.

For a short-hand visual introduction, refer to the full LSTM cell architecture illustration in Figure 2.12.

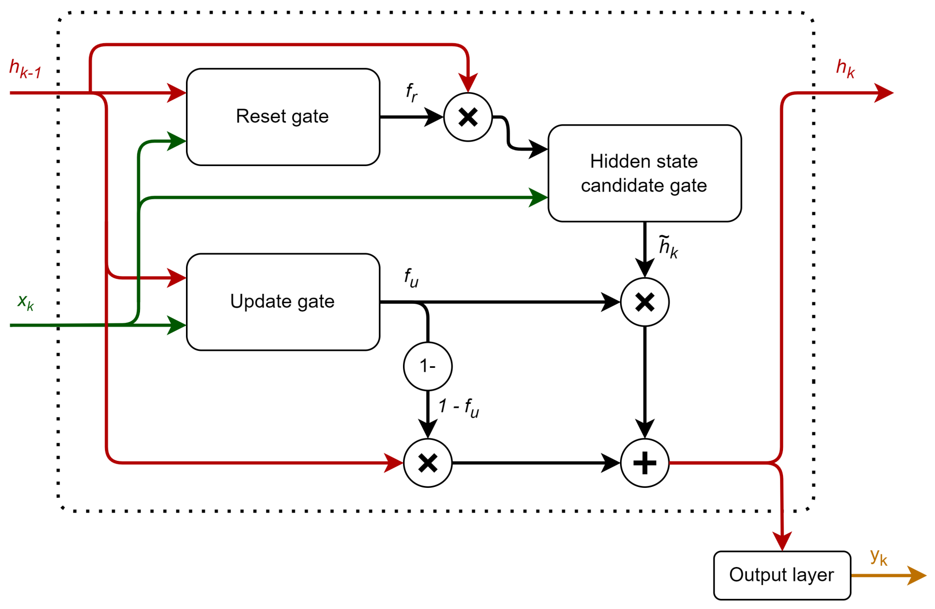


Figure 2.12: An illustration of the flow of information and series of computations occurring within a single cell of a GRU.

Training gated RNNs. Training of a gated RNN must be performed by means of the BPTT algorithm. Where the simpler RNNs, such as the RNNARX, may be trained without BPTT by means of e.g. teacher forcing, there exists no similar easy way of avoiding the use of BPTT for these models. Additionally, one would not want to simplify the training procedure in such a way, as training the LSTM or the GRU for several samples *in sequence* is integral in order to utilize its context-understanding capabilities and shape the parameters, such that they alter the importance of each information flow according to precisely sequential data. However, the use of BPTT and the LSTM's increased amounts of parameters with respect to e.g. the RNNARX does indeed make the training procedure considerably heavier. Though the GRU architecture does contain fewer parameters, it still contains more than a trivial RNN. Noteworthy, though, applications of LSTM and GRU prove to be the best-performing variants of the gated RNNs[17] (p. 406), which makes an argument that the added computational cost of training such models may be worth the investment.

2.4.4 The encoder-decoder structure

Common to the gated RNNs is their more extensive architecture, compared to what may be found in simpler RNNs such as the RNNARX. While the size does not *necessarily* cause the model to be inhibitively computationally heavy, [13] and [40] both proposed a way to work with the model's information more effectively. Instead of working on the system's high-dimensional state, they proposed to *encode* the input information into a lower-dimensional latent state formulation. A model trained on such encoded information may then be implemented more compactly than a model trained on the original system state information, and consequently perform calculations more effectively. Retrieving the predictions made by the lower-dimensional model is then done by *decoding* the latent state information to the desired output format. This output format may be of the same dimensions as the original system's state's format, but does not have to be; the *encoder-decoder structure* allows translating the input sequence's length to an output sequence of different length. Embedding a model within an encoder-decoder structure in this way is exemplified successfully in [12].

Note that the encoder and decoder themselves must themselves be implemented by procedures suited to process sequential data; they may for instance be implemented as RNNs[17].

2.5 A recurrent neural network-based MPC problem formulation

The main distinguishing element between recurrent neural network-based MPC (RNNMPC) and other MPC is its use of an RNN-based model as the system model encoded within the constraints. This section derives an NNMPC problem formulation suitable to control the system presented in Section 1.4 to any feasible target reference values. Specifically, the chosen NN-based is presented in Section 2.5.1, and integrated into a general MPC problem formulation in Section 2.5.2.

2.5.1 Recurrent neural network architecture for MPC

Many different modelling bases may be chosen on which to build a derivation of the mathematical formulation for the RNNMPC. Recall the goals presented in Section 1.3: the RNNMPC should be easily implementable, and the RNNMPC comparable to a linear MPC. Among the modelling options presented in Section 2.4, the RNNARX model presents the simplest RNN architecture.

The second part of the goal implies that the modelling basis for the RNNMPC should bear comparable similarities to the modelling basis for the LSRMPC, i.e. a linear step response model. Among the presented modelling methods, the philosophy of the NARX-model - as presented in (2.22) and (2.21) - is clearly reminiscent of the philosophy of the LSR-model - as presented in (2.10); both present a model of future output of a process as a static function of a history of previous inputs and outputs.

Based on the above considerations, the chosen RNN architecture for the RNNMPC in this thesis is the RNNARX model, as presented in (2.27). Differences between implementing the RNNMPC with the RNNARX model instead of e.g. an LSTM- or a GRU-based model are discussed in Section 5.3.2. Once designed for the oil and gas well presented in Section 1.4, the RNN becomes as illustrated in Figure 2.13. Following are the specific mathematical formulations of the resulting RNNMPC.

2.5.2 RNNMPC

The considerations with respect to the cost function and the constraints here presented are based on the ones presented in [19], as they share essential similarities, and the work done in [19] was intended as a basis for future work. The specific MPC problem formulation is somewhat expanded in order to include further considerations not made in [19].

Cost function. The goal of controlling the outputs to some given reference value motivates including a term penalizing deviations in the outputs from their given reference values. Note that, contrary to [19], the references are here assumed variable with time. The only requirement for the references is that they are known prior to optimization at

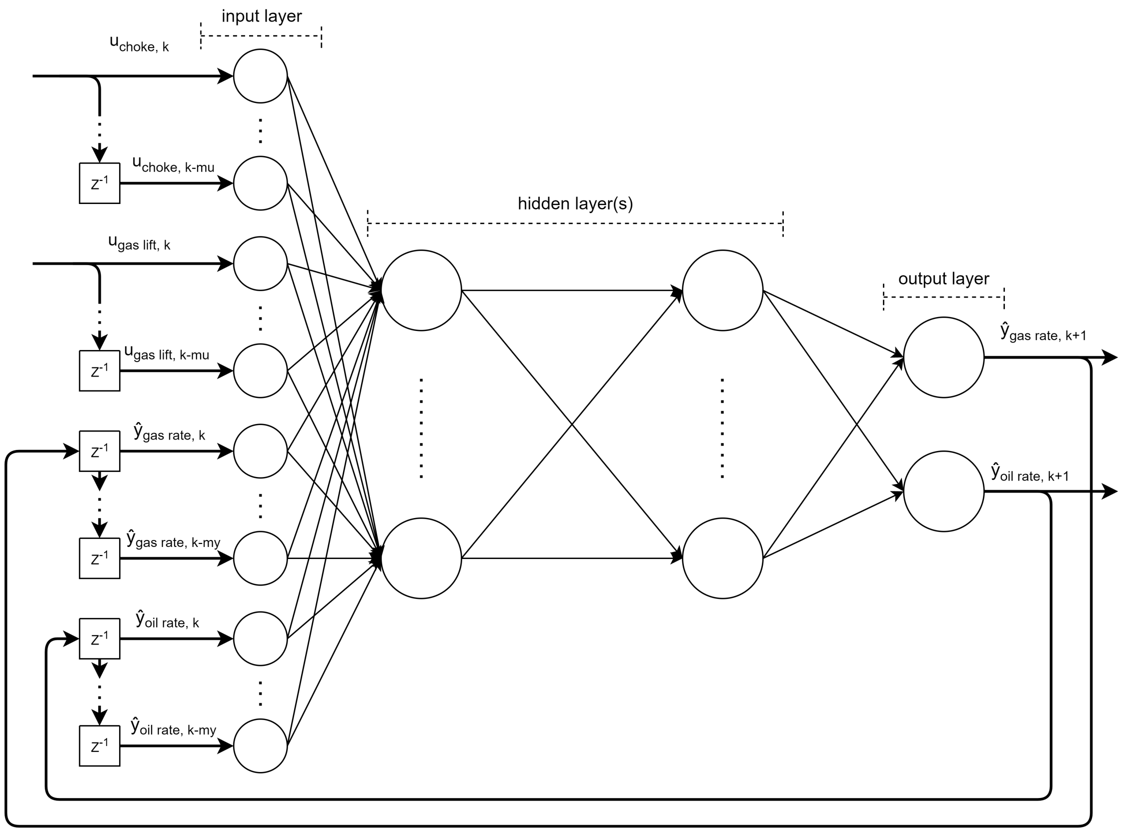


Figure 2.13: An illustration of the RNN resulting from chaining the single-step NNARX model of the single well process dynamics, implementing multi-step dynamics prediction as described in (2.27). The figure was first used in [19], there based - with permission - on an illustration from [16], page 3.

each time step, such that the cost function is well-defined. The goal of performing control that is economic with respect to wear and tear on actuators motivates a term penalizing too-high values in change in actuation. The prediction and control horizons are chosen to be equal for initial simplicity in implementation, $H_p = H_u = N$. The cost function is then formulated in the standard quadratic fashion:

$$l(\hat{\boldsymbol{y}}_{k+1:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref,k+1:k+N}) = \sum_{i=0}^{N-1} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^{\mathsf{T}} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) + \boldsymbol{\Delta u}_{k+i}^{\mathsf{T}} \mathbf{R} \boldsymbol{\Delta u}_{k+i} \tag{2.29}$$

where $i = 0, 1, 2, ..., N-1$ ensures all timesteps into the future are considered. $\mathbf{Q} \succeq 0 \in \mathbb{R}^{n_{out} \times n_{out}}$ is the diagonal matrix penalizing deviations in the outputs from their given reference, and $\mathbf{R} \succ 0 \in \mathbb{R}^{n_{in} \times n_{in}}$ the diagonal matrix penalizing too-high values in change in actuation.

Constraints. Constraints implemented in the MPC optimization problem formulation must ensure that optimization is always in accordance with the system model (2.27). This implies two things. Firstly, the initial prediction of our model must equal that of the system's current state, meaning that

$$\hat{\boldsymbol{y}}_k = \boldsymbol{y}_k$$

must hold. Secondly, the optimization is not free to alter the output predictions directly, but must adhere to the system model:

$$\hat{\boldsymbol{y}}_{k+1+i} = \hat{\boldsymbol{f}}_{MLP}(\hat{\boldsymbol{y}}_{k+i:k+i-m_y}, \boldsymbol{u}_{k+i-1:k+i-1-m_u}, \boldsymbol{u}_{k+i}),$$

Note that the system model becomes implicitly recurrent for increasing i.

In order to adhere to a defined feasible region for the outputs, we require:

$$\boldsymbol{y}_{lb} - \boldsymbol{\epsilon}_y \leq \hat{\boldsymbol{y}}_{k+1+i} \leq \boldsymbol{y}_{ub} + \boldsymbol{\epsilon}_y$$

Note that the addition of slack variables $\boldsymbol{\epsilon}_y$ implies an addition of a corresponding cost-term in the cost function, also introducing a third tuning variable, $\boldsymbol{\rho} > 0$. In addition to the constraints formulated above, any physical system's actuators have some saturation, as do their rates of change. This can be formulated as follows:

$$\boldsymbol{u}_{lb} \leq \boldsymbol{u}_{k+i} \leq \boldsymbol{u}_{ub}$$
$$\boldsymbol{\Delta u}_{lb} \leq \boldsymbol{\Delta u}_{k+i} \leq \boldsymbol{\Delta u}_{ub}$$

Since the optimization is with respect to changes in actuations, the actual input value $\boldsymbol{u}_{k+i}$ must be maintained consistent with the changes in actuation:

$$\boldsymbol{u}_{k+i} = \boldsymbol{u}_{k+i-1} + \boldsymbol{\Delta u}_{k+i}$$

In order to compensate for modelling errors, a bias assumed constant for all future time steps during each iteration of the MPC's optimization loop, is calculated as the difference between the last measured output and the predicted output for that same step:

$$\boldsymbol{v}_{k+i} = \boldsymbol{y}_k - \hat{\boldsymbol{y}}_{k|k-1}$$

The bias compensation is implemented by adding the bias to the model constraint. Lastly, the slack variables $\boldsymbol{\epsilon_y}$ are optimization variables and must be scaled by some weight defined as $\boldsymbol{\rho} > \boldsymbol{0}$, in which case also $\boldsymbol{\epsilon_y}$ must be positive semi-definite in order for the full cost function to remain positive semi-definite.

$$\boldsymbol{\epsilon_y} \geq \boldsymbol{0}$$

RNNMPC Problem Formulation. Based on the semi-general MPC problem formulation presented in (2.9), the considerations made above result in the following total RNNMPC problem formulation:

$$\begin{aligned}\min_{\boldsymbol{\Delta u}_{k:k+N-1}, \boldsymbol{\epsilon}_y} \sum_{i=0}^{N-1} & [(\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^{\mathsf{T}} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) \\ & + \boldsymbol{\Delta u}_{k+i}^{\mathsf{T}} \mathbf{R} \boldsymbol{\Delta u}_{k+i}] \\ & + \boldsymbol{\rho}^{\mathsf{T}} \boldsymbol{\epsilon}_y \end{aligned} \tag{2.30a}$$

$$s.t.$$

$$\begin{aligned}
\hat{\boldsymbol{y}}_k &= \boldsymbol{y}_k && && (2.30b)\\
\hat{\boldsymbol{y}}_{k+1+i} &= \hat{\boldsymbol{f}}_{MLP}(\hat{\boldsymbol{y}}_{k+i:k+i-m_y}, \boldsymbol{u}_{k+i-1:k+i-m_u}, \boldsymbol{u}_{k+i}) + \boldsymbol{v}_{k+i} && \forall\ i \in [0, N-1] && (2.30c)\\
\boldsymbol{y}_{lb} - \boldsymbol{\epsilon}_y \leq &\ \hat{\boldsymbol{y}}_{k+1+i} \leq \boldsymbol{y}_{ub} + \boldsymbol{\epsilon}_y && \forall\ i \in [0, N-1] && (2.30d)\\
\boldsymbol{u}_{lb} \leq &\ \ \boldsymbol{u}_{k+i} \ \ \leq \boldsymbol{u}_{ub} && \forall\ i \in [0, N-1] && (2.30e)\\
\boldsymbol{\Delta u}_{lb} \leq &\ \boldsymbol{\Delta u}_{k+i} \leq \boldsymbol{\Delta u}_{ub} && \forall\ i \in [0, N-1] && (2.30f)\\
\boldsymbol{u}_{k+i} &= \boldsymbol{u}_{k+i-1} + \boldsymbol{\Delta u}_{k+i} && \forall\ i \in [0, N-1] && (2.30g)\\
\boldsymbol{v}_{k+i} &= \boldsymbol{y}_k - \hat{\boldsymbol{y}}_{k|k-1} && \forall\ i \in [0, N-1] && (2.30h)\\
\boldsymbol{\epsilon}_{\boldsymbol{y}} &\geq \mathbf{0} && \forall\ i \in [0, N-1] && (2.30i)
\end{aligned}$$

Note that (2.30) is a nonlinear MPC problem formulation, and must be solved by a nonlinear solver, as explained in Section 2.1.4.

Chapter 3

Implementation

This section describes the relevant implementational details of both the LSRMPC and the RNNMPC, and how they both will be tested in accordance with the goals which specify the qualities of good control performance. This lays the foundation for performing the main overarching goal of this thesis: to compare an experimental nonlinear MPC based on neural network-modelling against an efficient, industry-relevant linear MPC - see Section 1.1. More specifically, Section 3.1 outlines the goals underlying the specific formulations of the desired qualities of the control performance results in this thesis, as well as the desired qualities of the tests to which the MPC-implementations will be subjected. Additionally, the system configuration and hardware specifications relevant for this thesis are presented. Section 3.2 and Section 3.3 present the implementations of the LSRMPC and RNNMPC, respectively. Both of the MPCs will be both tuned and tested on the same reference sequences, such that their control performances may be compared on equal bases.

Note that the implementation of the LSRMPC is a direct continuation of the work from [19]. The RNNMPC is instead fully a work of this thesis, while its underlying model is a continuation on, and significant improvement of, the work from [19].

3.1 Goals, specifications, tests and programmatic interfaces

3.1.1 Goal specifications

The measure of a target tracking control scheme is mainly its ability to minimize the deviation from the reference. Secondary to actually achieving the reference, but still of interest, is to minimize the deviation from the reference as *quickly* as possible. Further considerations based on which control performance may be deemed as good are based on safety, economic and environmental factors.

Performing proper control is integral for the safety production facility crew, as well as the long-term sustainability of surrounding areas. The Deepwater Horizon disaster[41] as an example, though not caused by poor flow rate control specifically, illustrates the importance of maintaining flow rates within a system's specifications. Though components of an oil rig should be specified to withstand extreme conditions, it is nevertheless desirable to avoid volatile flow rates. For the sake of this thesis, this is condensed into a control goal of *avoiding oscillations* in the flow rates, i.e. the system outputs.

Economically, though maximal production of oil and gas is ideal, unnecessary wear and tear on actuation equipment should be avoided. This supports the goal of avoiding oscillations also for the actuators, i.e. the system inputs. but also implies that maximal rates of change should be defined for the actuators. In dialogue with this thesis' partner, Equinor, I have decided to assume that the choke should not go from closed to fully open faster in less than 30 minutes. The gas lift rate is able to go from minimal to maximal actuation in 5 minutes. Then we have that the maximal rate of change in choke and gas lift are $\frac{100[\%]}{30[min]} \approx 0.55\,[\frac{\%}{10s}]$, and $\frac{10^4[m^3/h]}{5[min]} \approx 333.3\,[\frac{m^3/h}{10s}]$, respectively.

Additionally, the use of heavily power-demanding actuation should be minimized for maximal economic gain. In this thesis' case, see Section 1.4, that means prioritizing the use of the choke (low power demand) over using the gas lift (high power demand). Lastly, in maximizing production capacity lies an implicit risk of *overshooting* the reference. If, for any reason, an oil and gas rig has a higher gas flow rate than its production capacity, the gas might not be sent to a refining facility, and must instead be burnt. Due to environmental considerations, the Norwegian government imposes fees on such excess burning of gas. Overshooting the gas rate reference thus implies three main consequences: loss of income from future unsold gas, environmental fees and excess CO_2-emissions. In light of recent international agreements on climate change, avoiding unnecessary and excessive CO_2-emissions should be prioritized alongside safety- and economic considerations.

Though overshooting the oil rate reference does not imply the same economic and environmental consequences, overshooting the reference still means that the system necessarily *will* have oscillatory tendencies to some degree while approaching its reference. This thesis then proposes to avoid overshoot also in oil rate as a goal.

In total, three main goals for the control performance in this thesis are proposed:

1. Track target references as closely (and quickly) as possible.
2. Avoid oscillations in both gas rate and oil rate, as well as both choke and gas lift rate.
3. Avoid overshoot in either output, both gas rate and oil rate.

Target tracking performance may be measured by means of e.g. MSE between reference output and measured output. Measuring the degree of oscillations and overshoot is more difficult, and judging the control performance of the LSRMPC and the RNNMPC based on these factors is done argumentatively, rather than numerically, in Section 4.1.1 and Section 4.3.1, respectively.

In agreement with Equinor, the oil rate has been determined to be the main control goal of this thesis, due to the expenses involved in using the gas lift. This only implies that later tuning will be performed with a prioritized emphasis on meeting the reference values for the oil rate over the reference values for the gas rate - if necessary.

3.1.2 Test specifications

The reference profiles used for testing the MPCs' performances should test their ability to accommodate the goals specified in Section 3.1.1. Specifically, the reference profiles for testing should in total cover:

1. testing high-valued references,
2. testing low-valued references,
3. changing the reference in both gas rate and oil rate at different times, and
4. changing the reference in both gas rate and oil rate simultaneously.

The first and second of the above test specifications test the MPC for its ability to operate in a broad working range, carrying implications of the model accuracy and general applicability or alternatively the MPC's robustness to the model's lack thereof. The third and fourth of the above test specifications test the MPC for its ability to control gas rate and oil rate separately. While this thesis does not investigate the MPC's ability to control the system's outputs to arbitrary combinations of steady-state values, it is of interest to assess whether the MPC is able to control the gas and oil rates separately.

Though the above-proposed test specifications will be applied when testing the two MPCs, I make some simplifications for the reference profiles to be used during tuning, in order to be able to isolate which changes to the tuning have which effects on the control performance. Specifically, they will contain fewer steps with longer waiting times between them, such that the dynamic response has more time to settle to steady-state for each step. Additionally, due to time restrictions on development, I use somewhat shorter reference profiles, such that iterating on tunings becomes faster. In spite of these simplifications, the final reference profiles used are still designed to uphold the first and third of the points presented above, such that the tuning results and test results are measured in light of the same requirements, indicating that good tuning results will correlate with good test results as well. The process of tuning the two MPCs are described in Section 3.2.3 and Section 3.3.4, and the above-described reference profiles are presented alongside the consequent tuning results in Section 4.1.1 and Section 4.3.1.

For the sake of guaranteed achievability of the references used during tuning, I mapped the steady-state values the system attains at different combinations of actuation. Since any one such steady-state value must be achievable by *at least* one combination of actuation when later performing control, they would serve as references the MPCs should be able to reach in *at least* one way. I performed the mapping by applying, in sequence, all combinations of $choke \in \{20, 22, 24, ..., 100\}$ $[\frac{\%}{10s}]$[1] and $gas\ lift\ rate \in \{0, 2000, 2200, 2400, ..., 10000\}$ $[\frac{m^3/h}{10s}]$. Interfacing with the system was done as described in Section 3.1.3. The system was allowed to reach steady-state between each new actuation, thus yielding each actuation combination's corresponding steady-states for gas and oil rates. These gas and oil rates are later used for inspiration when designing the reference profile used during tuning - see Section 4.1.1 and Section 4.3.1. Knowing these values also allows to "warm up" the system to meet the initial references for the tests of control performance later performed in Section 4.1.1 and Section 4.3.1.

The approach of investigating guaranteed achievable set-points prior to tuning does ensure that both the LSRMPC and the RNNMPC should be able to meet their references during tuning, given a proper system model. Knowing this, any failed target tracking during tuning becomes isolated to factors related to the specific MPC implementation, such as a poor system model or poor tuning. In actual real-world applications, regards external with respect to the system may decree what set-points should be attempted reached, and

[1] $choke \in \{0, 2, 4, ..., 18\}$ $[\frac{\%}{10s}]$ was not included in the set, as these are values below which regular operation does not happen due to the degree of restricted gas and oil flow.

thus the achievable set-points approach does not necessarily reflect real-world operations. Thus, I use this approach only for the tuning of the MPCs, whereas for the testing, reference profiles will also investigate whether the gas and oil rates seem to be separately controllable - as indicated in the enumerated list above. Concrete results are presented in Section 4.1.1 and Section 4.3.1.

The computation times of both tuning and testing both the LSRMPC and the RNNMPC are subject of interest as well. The computation times tend to increase with the mathematical complexity of the optimization problem and may become prohibitively long, depending on the hardware platform on which the optimization is performed. Once the problem is defined, however, the computation times are expected to remain approximately constant across iterations. The specific parameters causing variations in the computation times differ between the LSRMPC and the RNNMPC - see Section 3.2.4 and Section 3.3.5, respectively. The observed computation times are presented - after the corresponding test results - in Section 4.1.2 and Section 4.3.2.

The hardware platform used in the development of this thesis is described in Table 3.1. This platform offers no parallelization software compatibilities, and thus no parallelization has been employed throughout this thesis; all computations have been performed directly using the CPU. Faster computations than the ones later observed might be achieved if some parallelization techniques, like using CUDA[42] with a compatible GPU, are employed.

Processor	Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz 3.10GHz
Installed RAM	32.0 GB (31.7GB usable)
System type	64-bit operating system, x64-based processor
Edition	Windows 10 Education
OS build	19044.2965

Table 3.1: Specifications of the desktop PC used to run all code developed for this thesis.

3.1.3 Programming the system and its configuration

Here described are the programmatic interfaces used throughout this thesis, including programming language and the way in which the digital system model was interfaced with. All software used is additionally listed in Appendix A.

The programmatic work in this thesis, made available on GitHub[43], has been performed in Python (version 3.10.6)[44]. The reason is two-fold. Firstly, the foundation from [19], on which this thesis builds further, was implemented in Python. That includes both a functioning implementation of a MIMO LSRMPC, see Section 3.2, as well as an in-progress framework in which to develop neural networks utilized in Section 3.3.1 and Section 3.3.2. Secondly, Python enjoys vastly available open-source libraries for many functionalities, such as PyTorch[45] for developing neural networks. That the libraries' implementations are open-source proved helpful during development, as any uncertainties regarding functionality may be inspected at the core.

The work in this thesis has not been done with respect to the physical system described in Section 1.4, but instead with respect to a digital model of the physical system; for the sake of this thesis, the model has been made available in the form of a *Modelica*-model.

Modelica[46] is a programming language for encoding systems as *non-causal* models, where sets of equations must be simultaneously upheld. This is different from the more common *causal* structure implemented by e.g. Python, where a series of assignments to variables lead to sequential behaviour in those variables. For further details on Modelica, the interested reader is referred to [46]. Modelica-models may not be directly interfaced with using Python, but must first be exported as a *Functional Mock-up Unit* (FMU). The way in which the Modelica model was exported to an FMU during development for this thesis is described in Appendix C. Using the Python-library *pyfmi*[47], an FMU may be given inputs, for which it can simulate the physical system, after which the physical system's simulated outputs may be retrieved. Note that the FMU contains no noisy behaviour.

Embedded in the underlying Modelica-model of the physical system, and thus the FMU, are the predefined hard limits to actuation to which the system must adhere, as described in Section 1.4. Though not directly embedded into the FMU, these limits imply lower and upper limits to the steady-state output values, which have been derived empirically by applying minimal and maximal actuation until steady-state, respectively. The maximal rates of change in actuation, see Section 3.1.1, are not directly embedded in the FMU, but instead enforced manually within the implementation. Lastly, both the LSRMPC and the RNNMPC later implement slack on the limits on the outputs, here represented by $\boldsymbol{\epsilon_y}$. As explained in Section 2.1.2, $\boldsymbol{\epsilon_y} > 0$ is required as part of retaining the cost function as positive semi-definite. Though no upper limit is theoretically required, it is included in the system configuration for programmatic reasons, and tentatively set to 10^6 to serve as simply a very high value. In total, the above arguments motivate the limits listed in Table 3.2.

The sampling time of the system is defined to be $\Delta t = 10\,[s]$ for both the LSRMPC and the RNNMPC.

Variable	Bounds		Motivation
Choke	$[0,\ 100]$	$[\%]$	Given as specification
Gas lift	$[0,\ 10000]$	$[\frac{m^3}{h}]$	Given as specification
Gas rate	$[0,\ 18537]$	$[\frac{m^3}{h}]$	Empirically derived
Oil rate	$[0,\ 349]$	$[\frac{m^3}{h}]$	Empirically derived
Rate of change in choke opening	$[-0.55,\ 0.55]$	$[\frac{m^3}{h}]$	Given as specification
Rate of change in gas lift rate	$[-333.3,\ 333.3]$	$[\frac{m^3}{h}]$	Given as specification
Slack on gas rate limits	$[0,\ 10^6]$	$[\frac{m^3}{h}]$	Implementational necessity
Slack on oil rate limits	$[0,\ 10^6]$	$[\frac{m^3}{h}]$	Implementational necessity
Sampling time	10	$[s]$	Given as specification

Table 3.2: The static parameters of the oil and gas well system, their values and how they were derived. These provide a numerical basis for later development of the LSRMPC, Section 3.2.2, and the RNNMPC, Section 3.3.3.

3.2 Implementing the LSRMPC

3.2.1 LSRMPC problem formulation

The LSRMPC presented in (2.11) is a basic MPC problem formulation for the SISO case, whereas the system in question is MIMO. While extending the SISO linear step response model to the MIMO case is trivial, see (2.12), the integration of the MIMO system model into an MPC problem formulation depends on the chosen problem representation. The MIMO LSRMPC problem representation presented in [2] was developed in cooperation with Equinor, and thus presented a suitable choice for an LSRMPC problem formulation with industrial foundations, which may later be used as comparison grounds against the experimental, nonlinear RNNMPC. This LSRMPC problem representation implements an efficient MIMO LSRMPC. The matrices of the complete LSRMPC problem formulation and their derivations are extensive, and, for the sake of brevity, this thesis does not provide a comprehensive presentation of them. Instead, the interested reader is referred to the summary in [2] (pages 34-39 and 59-60). Though extensive, this MIMO LSRMPC problem representation still implements a *linear* MPC.

Importantly, by means of problem size reduction, [2] achieves to reduce the optimization variables in [2] to only:

$$\boldsymbol{z} \triangleq [\boldsymbol{\Delta u}_k \quad \boldsymbol{\epsilon_y}] \tag{3.1}$$

This is noteworthy, as it later determines which specific parameters must be assigned values during system configuration; see Section 3.2.3 and Table 3.3. Note also that the tunable weight matrices, which in [2] are denominated $\bar{\mathbf{Q}}$ and $\bar{\mathbf{P}}$, are instead referred to in this thesis as $\mathbf{Q}$ and $\mathbf{R}$, respectively, so as to ensure a consistent naming scheme throughout this thesis.

3.2.2 LSRMPC: implementational details

The implementation of the LSRMPC was a joint effort between Simen Bergsvik, Amalie Gjersdal and the author in association with project work leading up to our individual theses. This is well-documented in [19]. The implementation of the LSRMPC included in this thesis is the same, with some bug fixes and optimizations for code readability and running time improvements.

The main control loop of the LSRMPC requires four external elements: the FMU (see Section 3.1.3, the configuration of system parameters (static and tunable, see Section 3.2.3), a series of reference values which implement the desired target outputs to be tracked during control, as well as the MIMO linear step response model. While the FMU was provided for the sake of this thesis, the other three requirements must be supplied by us, the developers. The configuration of the system was encoded into a simple '.yaml'-file, and the reference values encoded into a simple '.csv'-file - the interested reader is referred to [19] for the specific details of the code.

The MIMO linear step response model was identified after the process described in Section 2.1.3: The system was driven to steady-state for an input of $[\text{choke}, \text{gas lift rate}] = [50\,[\%], 0\,[\frac{m^3/h}{10s}]]$, after which a step was made to $[\text{choke}, \text{gas lift rate}] = [52\,[\%], 0\,[\frac{m^3/h}{10s}]]$. The outputs were then retrieved from the FMU for each subsequent timestep. Once the input-output values were measured, the step response coefficients for each input-output relation from step to steady-state were derived, yielding the vectors $S_{gas\ rate,\ choke}$, $S_{oil\ rate,\ choke}$, $S_{gas\ rate,\ gas\ lift\ rate}$ and $S_{oil\ rate,\ gas\ lift\ rate}$. These vectors were stored separately as numpy arrays ('.npy'-files), and retrieved upon need during the main LSRMPC-script.

With the external requirements satisfied, implementing the LSRMPC-loop itself is a matter of choosing an appropriate framework with which to implement the required matrices of the LSRMPC problem formulation, as well as the solver required to solve the optimization problem at each iteration. The Python library *numpy*[48] was chosen as framework for the implementation of all matrices. The chosen solver is *gurobi*'s[49] quadratic programming-solver. Note that this solver is a commercial solver, and not openly available - an academic license was used for the sake of the LSRMPC development. This solver was chosen for its observed efficiency during the LSRMPC development process, but other, free-of-charge alternatives, such as *osqp*[50], exist.

Though *self-developed*, the specific implementational details of the code implementing the LSRMPC, and all its frameworks, is not included in the text of this thesis. Instead, an algorithmic presentation of the main LSRMPC-loop is given in algorithm 2. For further details, the interested reader is referred to the open-source code at [43].

3.2.3 Tuning the LSRMPC

The above-presented LSRMPC implementation requires both static and tunable parameters to be determined prior to running the control loop. Applying the system information provided in Table 3.2 to the LSRMPC-equations, the list of parameters as listed in Table 3.3 is derived. These values are parameters that configure the system and are *static*, as they reflect physical properties or desired behavioral limits, from which the system is not necessarily allowed to deviate. Recall the compact vector of optimization variables in (3.1). Accommodating this formulation, the upper and lower limits to both the rate of change in actuation $\boldsymbol{\Delta u}_{lb}$ and $\boldsymbol{\Delta u}_{ub}$, as well as to the slack variable $\boldsymbol{\epsilon_y}$ in this LSRMPC problem formulation are encapsulated within the variables $\boldsymbol{z}_{ub}$ and $\boldsymbol{z}_{lb}$, respectively.

In addition to the predefined parameters in Table 3.3, the system has *tunable* parameters - listed in Table 3.4 - whose values should be determined by the user based on what configurations provide the best control performance results. While not the core subject of this thesis, the tuning of the LSRMPC should be properly addressed for the sake of well-performing control.

Algorithm 2: Pseudo code for the program flow which implements the simulation of control by means of the LSRMPC. This is not an exhaustive guide to implementing an LSRMPC. For exhaustive details, the interested reader is referred to the open-source code at [43].

```
Input:
    1. digital system representation (FMU)
    2. initial actuation state $\boldsymbol{u}_0$
    3. system configuration (tuned parameters, time step size $\Delta t$, simulation length $t_{fin}$)
    4. reference value trajectory $\boldsymbol{y}_{ref,0:t_{fin}}$
    5. MIMO linear step response model $\mathbf{S}$
Output:
    1. optimal trajectory of actuation values
    2. corresponding trajectory of system outputs

Warm-start: apply $\boldsymbol{u}_0$ to FMU until steady-state is achieved;
$t \leftarrow 0$;
$k \leftarrow 0$;
Retrieve current output $\boldsymbol{y}_k$ from the FMU;
while $t < t_{fin}$ do
    Update constraints and costs according to $\boldsymbol{y}_k$ and $\boldsymbol{y}_{ref,t}$;
    Apply constraints and costs to optimization problem;
    Call solver on updated optimization problem;
    Retrieve optimal change in actuation $\boldsymbol{\Delta u}^*_k$;
    $\boldsymbol{u}^*_k \leftarrow \boldsymbol{u}^*_{k-1} + \boldsymbol{\Delta u}^*_k$;
    Apply optimal actuation to $\boldsymbol{u}^*_k$ to FMU;
    Retrieve current output $\boldsymbol{y}_k$ from the FMU;
    $t \leftarrow t + \Delta t$;
    $k \leftarrow k + 1$;
end
```

Static parameters	Value
$\boldsymbol{y}_{lb}$	$[0 \quad 0]$
$\boldsymbol{y}_{ub}$	$[18537 \quad 349]$
$\boldsymbol{u}_{lb}$	$[0 \quad 0]$
$\boldsymbol{u}_{ub}$	$[100 \quad 10^4]$
$\boldsymbol{z}_{lb}$	$[-0.55 \quad -166.7 \quad 0]$
$\boldsymbol{z}_{ub}$	$[0.55 \quad 166.7 \quad 10^6]$

Table 3.3: The static parameters of the system, encoded into the relevant variables of the LSRMPC problem formulation.

Tunable parameters
$\mathbf{Q} \in \mathbb{R}^{2\times 2}$
$\mathbf{R} \in \mathbb{R}^{2\times 2}$
$H_p \in \mathbb{R}$
$H_u \in \mathbb{R}$
$H_w \in \mathbb{R}$
$[\boldsymbol{\rho}_h^\mathsf{T}, \boldsymbol{\rho}_l^\mathsf{T}]^\mathsf{T} \in \mathbb{R}^{4\times 1}$

Table 3.4: The tunable parameters of the system, encoded into the relevant variables of the RNNMPC problem formulation.

The main method I used for tuning the tunable parameters of the LSRMPC was *grid search*. Grid search is a brute force method of searching for optimal parameter values: a set of candidate values is defined for each n parameter, such that an n-dimensional grid defines all possible configurations of all the parameter candidate values. The procedure, based on which the performance of each combination of parameters is judged, is then run for every single grid point in that grid, and the optimal choice of parameters is that which performs the best on the procedure. An example of performing one such procedure for a grid point could be to simulate a control sequence of an LSRMPC and measuring its resulting target tracking capabilities. An additional example is that of training a model on a specific hyperparameter candidate set, and then testing the model on a test data set. The measure of its performance is then typically its mean MSE over the full data set. Note that the amount of grid points grows in size as a product of the number of candidate values per parameter. As such, it is a naïve and computationally resource-demanding search-method, since every grid point is evaluated, regardless of considerations that could be made in order to effectivize around e.g. better-performing configuration values. This method was nevertheless chosen due to its implementational simplicity, as the tuning process itself is not the main focus of this thesis, and some time spent tuning was acceptable, as this may be done automatically in the background of other work. Other less naïve approaches, such as *genetic algorithms* and *random search* may yield improved performance. The interested reader is referred to [51] and [52](chapter 10) for introductory discussion of the two methods, respectively.

The variables $\boldsymbol{\rho}_h$ and $\boldsymbol{\rho}_l$ serve as weights on the slack variables ϵ_h and ϵ_l, respectively.

Since we want any given solution to allow as little slack as possible, $\boldsymbol{\rho}_h$ and $\boldsymbol{\rho}_l$ were chosen high-valued from the start and locked in place during further tuning procedure. This of course relies on the assumption that high-valued $\boldsymbol{\rho}_h$ and $\boldsymbol{\rho}_l$ are feasible, a condition which must be revisited if no good tuning is found during further tuning procedure.

Since the further parameters to be tuned involved all the horizons H_p, H_u and H_w as well as the weight matrices $\mathbf{Q}$ and $\mathbf{R}$, I performed the grid search in two rounds; one for the horizons, and one for the weight matrices. I assumed that by first performing a grid search with respect to the horizons, their values may later be locked in place without significantly affecting the later tuning process of $\mathbf{Q}$ and $\mathbf{R}$ adversely. This assumption is based on the argument that longer horizons tend to yield better performances in MPC control schemes - see Section 2.1.1. I thus assumed that tuning the horizons would result in the highest-valued horizons that I could justify with respect to computation times. The assumption that higher-valued horizons always perform better also justifies that $\mathbf{Q}$ and $\mathbf{R}$ may be locked into arbitrary values during this first grid search. The expectations I had for the tuning of $\mathbf{Q}$ and $\mathbf{R}$ differ significantly; I do not assume that strictly higher (or lower) values in $\mathbf{Q}$ and $\mathbf{R}$ will yield better performance - instead, the optimal values must be assumed to be the combination of $\mathbf{Q}$ and $\mathbf{R}$ which optimizes the trade-off between punishing deviations from the reference and punishing the use of actuation. For both rounds of grid search, I started with rough initial guesses of the parameters' values based on performance observed during test runs during the development of the LSRMPC.

For the tuning of the horizons, I first defined $H_w = 0$, as I know of no reason there should exist any time-delay in the system. I then grid searched for the optimal values of H_p and H_u, with the criterion for the best combination of values being a compromise between control performance and computation times. The resulting values for H_p and H_u are later described in Table 4.1 in Section 4.1.1, and considerations regarding the computation times of the LSRMPC are presented in Section 4.1.2.

After settling on values for the horizons, I performed a grid search over candidate values for $\mathbf{Q}$ and $\mathbf{R}$. Note that some level of effectivizing of the grid search procedure was implemented. Using the tuning of $\mathbf{Q}$ and $\mathbf{R}$ as an example; candidate ranges for $\mathbf{Q}$ and $\mathbf{R}$ were chosen, within which specific candidate values were chosen after a coarse resolution: 3 to 4 values per interval. The points were then logarithmically distributed, such that any given point would provide a significantly different magnitude of the corresponding penalty in the cost function than another. I performed the grid search iteratively; during each iteration, every grid point was investigated, but the next iteration's grid values were chosen in a neighbourhood surrounding the values yielding the best performance. In this way, the grid did not need to be high-resolution from the beginning. Instead, the procedure became more analogous to a tree search, where only the interesting branch is investigated. A hypothetical example, intended only for illustration of the concept, is provided in Figure 3.1.

Lastly, after the coarse search for the ideal $\mathbf{Q}$ and $\mathbf{R}$ was performed by grid search, I applied the logic presented in Section 2.1.2 in order to perform some last fine-tuning of $\mathbf{Q}$ and $\mathbf{R}$.

Once the tuning process was complete, the LSRMPC was tested on reference profiles as described in Section 3.1.2. The results are presented in Section 4.1.1

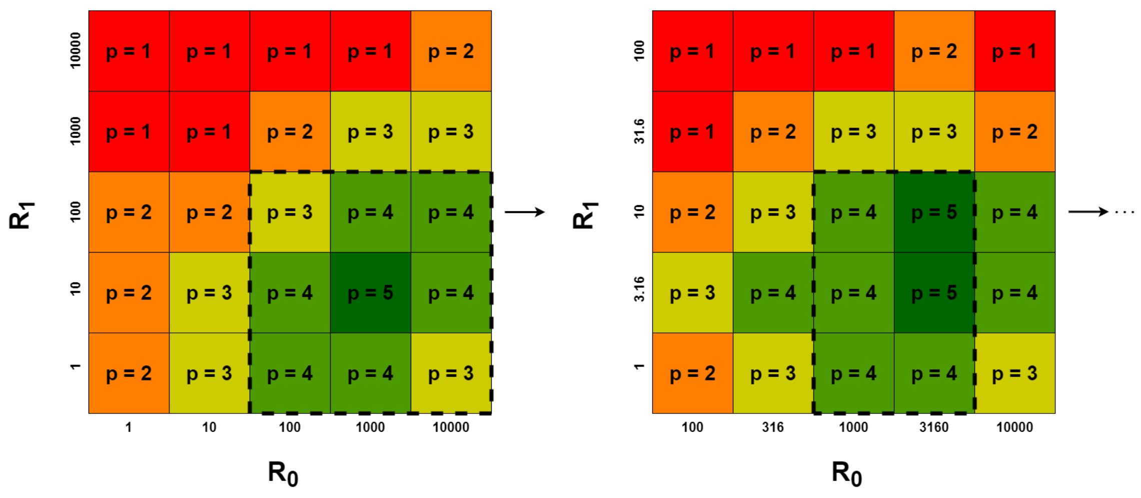


Figure 3.1: In this example intended *only* for illustration, the best-performing combination of two values R_0 and R_1 for some *arbitrary* procedure must be determined. The performance is measured on a scale $p \in [1, 5]$, where higher is better. The first iteration isolates $R_0 \in [100, 10000]$ and $R_1 \in [1, 100]$ as neighbourhood within the original search area as the best-performing region. The second iteration then explores that neighbourhood, while also *adjusting* the performance scale according to the worst- and best-performing grid points within the new neighbourhood, in order to accommodate the increased resolution. The procedure may be repeated for as long as desirable, but is here illustrated with only 2 iterations for simplicity.

3.2.4 Notes on computation times

The computation times associated with running the control loop of the chosen LSRMPC problem formulation vary depending on different factors. However, while factors such as hardware capabilities, programmatic efficiency in the implementation, and choice of solver of course matter greatly, these are factors external to the specific LSRMPC problem formulation, and may be addressed without regarding the specific parametric values of the LSRMPC. The factors which instead stem from the LSRMPC problem formulation itself are the values of the horizons H_p, H_u and H_w, as these directly determine the sizes of most matrices involved in optimization - see [2](Table 3.1, p. 41).

Observed computation times as a function of horizon sizes are presented in Section 4.1.2.

3.3 Implementing the RNNMPC

This section describes the implementation of the MLP-based RNNARX, as described in Section 2.4.2, underlying the later implementation of the RNNMPC, as described in Section 2.5.2. Specifically, Section 3.3.1 details how the data sets utilized during the training procedure of the model were gathered, Section 3.3.2 explains how the training procedure was applied on these data sets, resulting in a model applicable for a full RNNARX-based RNNMPC, whose implementation is presented in Section 3.3.3.

3.3.1 Data sets for training the model

This thesis has not had any data from real-life data-collection on which to train a model available. Instead all data sets on which to perform training, validation and testing of the model had to be synthesized by means of simulations of input-output responses using the FMU in the way described in Section 3.1.3. In order to synthesize data sets, an input profile must first be defined, then simulated through the FMU, which then produces outputs corresponding to the given inputs. Logging these outputs alongside the inputs yields synthetic data sets containing full sequences of input-output relations, which may later be tailored into samples usable during training. I created several such data sets based on different types of input sequences. Recall that the FMU is implemented without noisy behaviour.

In the case of synthesizing data through the FMU, all output values are direct consequences of the history of inputs - the degrees of freedom in the data are only the inputs. The question of how to best represent the system's dynamics by means of synthesized data is raised.

The end goal of the training procedure is for the model to be able to generalize. As covered in Section 2.3.1, this is better achieved if the distribution of the training data set matches that of real-world data, i.e. that the data set contains input-output relations mirroring the dynamics of the system. The *ideal* data set is entirely representing of the dynamics of the system from which it is sampled. This thesis proposes that representing the system's dynamics by means of synthesized data is best achieved by spanning the input domain as broadly as possible; the data set should contain as many combinations of values in choke and gas lift rate as possible. Furthermore, since the system is *dynamic*, the history of inputs - not only the current inputs - are relevant to the resulting outputs. As such, the data set should also contain as many sequences of combinations of values in choke and gas lift rate as possible.

This thesis proposes a way to create data set matching the above criteria by means of an FMU: Semi-randomly walking the input-space in both degrees of freedom - choke and gas lift rate - simultaneously, will tend towards covering all configurations within the possible input-domain, as well as all possible sequences, over time given that the two inputs are in asynchronous phase. Thus, given large enough simulation times, a close to fully representative data set is generated, in the sense that application of the model will encounter no new data which strongly deviates from any data contained within this data set. This idea is an extension of the qualities implemented by the amplitude-modulated pseudo-random binary signal (APRBS), acknowledged for its ability to emulate white noise and excite all frequencies of the system by applying step inputs[5]. Note that [5] argues that an APRBS signal is not necessarily an appropriate basis on which to identify nonlinear models, due to lacking coverage of highly nonlinear regions. This thesis nevertheless uses it as a basis for synthesizing a training data set, based on the argument that the data set may be made to cover broadly enough; this thesis hypothesizes that allowing a semi-randomly asynchronously walking APRBS-signal to semi-randomly walk across the inputs' domains *will* in fact grant the resulting data set broad enough coverage that the concerns raised by [5] are mitigated.

Since the system has two inputs, spanning the input-domain of any one of them is not sufficient; the data set should also span the domain of combinations of the two inputs. As long as the frequencies of the random choke walk and the random gas lift rate walk are proportional with an irrational number, allowing them to randomly walk makes the amount of combinations of their respective domains increase as time increases. However, since this thesis makes the assumption that the two inputs are continuous variables - their domains are infinitely resoluted - fully spanning all possible combinations of the two in reality is not possible; no data set may in practice be of infinite size. Furthermore, computation times during training increase with increasing size of the training set, due to the sheer amount of samples which need to be processed. This motivates restricting the data set. The size of the model also matters somewhat, due to the amount of computations needed to be made both during prediction and backpropagation. In general: when computational resources are a limiting factor, a compromise between model size and training data set size must be made against time considerations. Specific training times and reflections around these are omitted for brevity, as the main focus of this thesis is the RNNMPC built with the trained model.

Given the above reflections, I sought to create input profiles which spanned the input domain as broadly as computationally possible, while retaining computational feasibility during training of the model. I settled on the final sizes of the training and validation data sets to be 200.000 and 30.000 number of samples, respectively. The specific size of the training data set was chosen based on observations of computation times during training of the model; I wanted to be able to use as many samples as possible without the training over any one set of hyperparameters taking too long. The size of the validation data set was made relative to the training data set, and was based on the popular 70% / 15% / 15% ratio of training, validation and test sets; though amount of available data is not an issue in this thesis, this ratio justifies using a significantly larger data set for training than for validation.

The input profiles are designed such that each step happens in either positive or negative direction, with some uniform distribution over the values within the limits to the rate of change. Both inputs start low, with a skewed probability of incrementing, such that the semi-random walk will rise for some time. Once the maximal actuation value is reached for one input, the probability of its next steps are skewed towards decrementing, and vice versa once it reaches the minimal actuation value. In this way, both choke and gas lift rate are made to repeatedly span their domains. Since choke and gas lift rate have different limits to rate of change, see Table 3.2, their semi-random walks achieve asynchrony: gas lift rate seems to span its domain a little more than three times faster than choke. In order to also capture the transients, each input was forced to wait for some amount of time steps randomly sampled from 20 to 50. Simulating these input-profiles yielded the training data set shown in Figure 3.2. The validation data set was created exactly the same way. The test data sets were instead created differently in order to test for more specific predictive qualities of the trained models - see Figure 4.6 and Figure 4.7 in Section 4.2.2.

This training data set is intended for training of a model that may be chained into an RNN of the structure presented in (2.27), where both histories of the inputs' and outputs' values to the system are given as input values to the model. Since gas rate, oil rate, choke and gas lift rate all operate within different magnitudes, the data sets must be normalized prior to training. This means that the final model expects every input value to reside within $[0, 1]$. In order to ensure consistency, the validation and test data sets must also be normalized accordingly. The values used for normalization are the individual inputs' and outputs' upper and lower bounds, given in Table 3.2.

Due to being finite, the training data set can not be assumed to be fully representative of the system's dynamic behaviour for *all* configurations and working ranges of the system's inputs. Noteworthy: even if it had been fully representative, there would still exist no

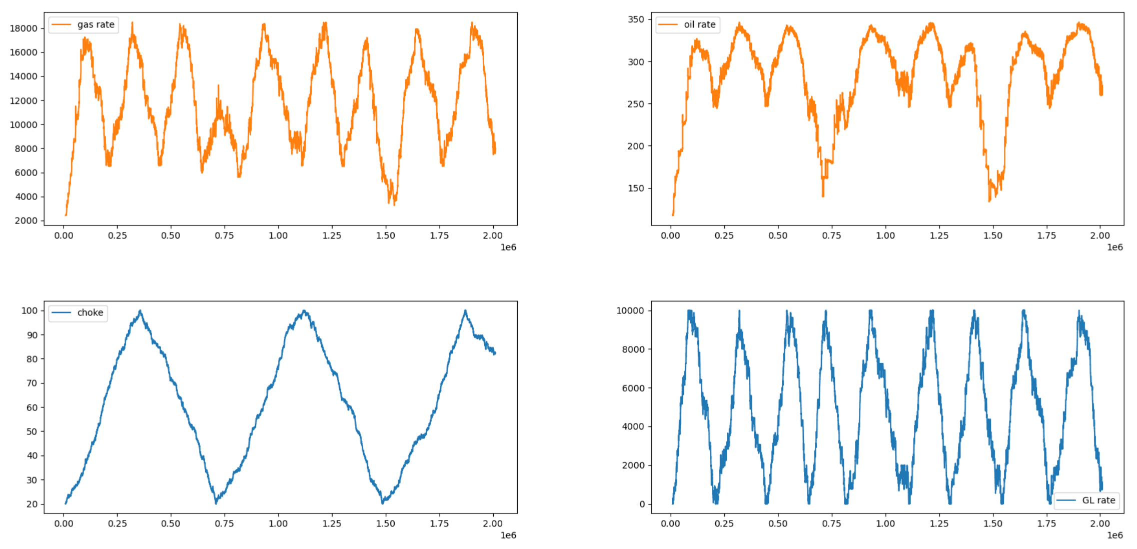


Figure 3.2: The synthesized data set later used for training models.

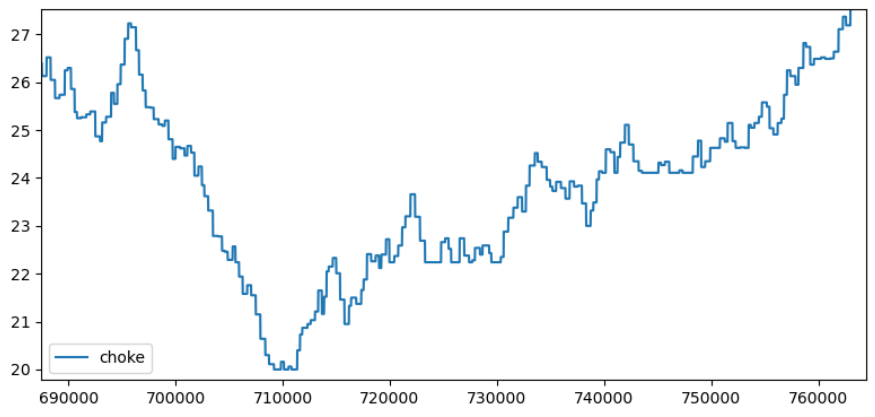


Figure 3.3: A magnified snippet of the input profile for choke, as presented in Figure 3.2. The snippet is intended to further clarify the structure of the excitation signal.

guarantee of the model perfectly learning the system's dynamics, as the cost function used by SGD during training, see Section 2.3.1, is highly nonlinear regardless of the training data set used, since even a very simple model has a nonlinear activation function at each neuron. Recall that SGD is inherently stochastic, and provides no guarantee of convergence to the *global* optimum. Hence, even though a very representative training data set is helpful in achieving an accurate model, it is no guarantee in and of itself. Whether the final performance is sufficiently accurate is a matter of application specifications, and judging whether the training data set is representative *enough* is then a matter of judging whether the specifications are met. This question is further addressed in the case of this thesis when the test results for the model and the consequent RNNMPC are presented, see Section 4.2.2 and Section 4.3.1, respectively, and consequently discussed in Section 5.2.1 and Section 5.3.1.

3.3.2 Training and testing of the model

I implemented a training-testing scheme for training models as MLPs using Python 3.10.6 and PyTorch 1.13.0. Code details are not covered in this thesis - see instead [43]. All the implementational work relevant to training and testing models is based off the theory found in Section 2.3.1 and Section 2.3.4.

Before training a model, structural considerations must first be made. Firstly, the chosen optimizer during training for this thesis is Adam, due to its proposed robustness in the tuning of the hyperparameters, Section 2.3.1. Furthermore, Adam is a variant of SGD, meaning all samples will be shuffled during training, removing all sequential correlation between the samples within Figure 3.2.

Secondly, the optimal set of hyperparameters must be identified. I attempted this by iterating over candidates for hyperparameters by means of grid search - completely similarly to how I tuned the LSRMPC - see Section 3.2.3. The hyperparameters which must be considered during such a process are outlined presently.

Adhering to the model's structure, see (2.27), the model itself required the hyperparameters m_u, m_y and $\eta_l \ \forall \ l$. The choice of activation function fell on *LReLU*, as it became evident that several neurons died during training during initial developments when using *ReLU*. Settling on *LReLU* seemed to remedy this issue, meaning that its leak rate α is a hyperparameter which must be defined. I chose to implement early stopping (see Section 2.3.4) as a regularization method to avoid overfitting, so the patience p must be determined. Furthermore, since I chose the optimization algorithm Adam, the hyperparameters batch size β and learning rate λ must be defined. Additionally, Adam allows the hyperparameter weight decay μ to be set, which implements L2-regularization - see Section 2.3.4. Lastly, as a safe-guard for termination of training, some amount of maximal epochs e must be decided. All the required hyperparameters are summed up in the Table 3.5.

Which figures should be varied over during the grid searched had to be addressed. The four figures m_u, m_y, the size(s) of the layer(s) η_l, and the number of layers l determine the nonlinear capacity of the model as well as its total size. The nonlinear capacity is important with respect to the models modelling capability, and the size is impactful with respect to the computation times associated with later solving the optimization problem at each iteration of the RNNMPC. These figures were thus deemed the most interesting to vary during the grid search.

Regarding the specific grid searchable values of m_u, m_y, η_l and l, the following considerations were made. Firstly, reducing the amount of neurons within the model is important to reduce the computational load on the optimization step of the RNNMPC control loop. Since the universal approximation theorem[7] states that universal approximation may be achieved with as little as one hidden layer, I decided to train all models in association with this thesis with precisely only one layer in order to reduce the computational load in the RNNMPC: $l \triangleq 1$ and thus $\eta_l = \eta$. Not knowing which hidden layer size would yield sufficient nonlinear capacity, I set $\eta \in \{20, 40, 200\}$ as candidate values for the grid search.

Secondly, the issue of exposure bias arises when the model takes in its own predicted values, while it is not trained to do so, see Section 2.4.1. Intuitively, exposure bias might then be pre-emptively mitigated by using a lower-valued m_y, such that fewer of the model's own predictions are fed back to the model's input. Instead, the model may then rely on the necessary information for system dynamics approximation being carried within the m_u

long history of inputs. Using lower-valued m_y also contributes to lowering computation times during RNNMPC optimization. Note that feedback of the model's own predictions may never be completely removed, as the model must have some knowledge of the system's outputs' state, from which to understand how future dynamics will behave. The ranges $m_u, m_y \in \{5, 20, 50\}$ were chosen for candidate values for the grid search, with the condition that $m_u \leq m_y$, such that no configuration would make predictions based on a longer history of output values than input values; while previous output values aid the predictions of a NARX-model, it is the input to the system which ultimately determine future output. Note also that as low as possible values for m_u and m_y are beneficial with respect to computation load.

Hyperparameter	Values	Description
m_u	$\in \{5, 20, 50\}$	the candidate sizes of the history of inputs considered
m_y	$\in \{5, 20, 50\}$	the candidate sizes of the history of outputs considered
η	$\in \{20, 40, 200\}$	the candidate sizes of the single hidden layer
α	0.2	the leak rate of *LReLU*
p	5	the number of epochs to wait for a better E_{val} before terminating
β	64	the amount of samples used in each step of the training
λ	0.001	the learning rate used in Adam
μ	0.1	the weight decay coefficient used in Adam.
e	200	the number of epochs for which the NN should train

Table 3.5: The hyperparameters and their chosen values or ranges of candidate values. Values are prior to grid search for best set of hyperparameters.

When using PyTorch to train regular models with linear units, like (2.17), the model's parameters - weights and biases - are initialized automatically according to the *Kaiming* method, unless otherwise specified[53]. As there was no apparent need to specify the initial parameters manually, I used this automatic initialization throughout all training.

Once all the above considerations were in place, the grid search could commence by training a model for each hyperparameter configuration in accordance with the procedure described in Section 2.3.1 and summed up in Figure 2.6.

Building on Section 1.1, the main goals of testing the model are to assess its ability to capture the system's nonlinear dynamics, as well as how broadly within the system's working range it is able to do so. The test data sets I used to assess the qualities of each trained model contained a single step in actuation in order to test for predictive accuracy around different working ranges for the system. The specific test data sets used are elaborated below.

The gas and oil rates of the well are in reality controlled by a step-choke with additional use of a semi-continuous gas lift supplementing *when needed* - see Section 1.4. This motivates to evaluate the model's ability to capture nonlinear dynamics by viewing its predictive accuracy on the outputs for steps in the choke, while the gas lift is kept constant. Since it is of interest that the model should perform well in all working ranges, steps are made in the choke for low-valued, medium-valued, and high-valued working ranges. Three different test data sets containing the three described steps were synthesized.

An additional test data set was synthesized in order to display the model's predictive accuracy more broadly: displaying a "staircase" input profile, this test data set is meant to emulate the ideal case where the choke is the preferred actuator, i.e. the one to be maxed out first. The result from testing the model on this data set is presented in Figure 4.7.

Note that all the above-described test data sets are synthesized independently of all training and validation data sets in accordance with the theory presented in Section 2.3.1. The results of testing the model on these test data sets, as well as the final choice of model to employ in the RNNMPC, are presented in Section 4.2.2.

3.3.3 RNNMPC: implementational details

The LSRMPC and the RNNMPC are both implementations of MPC, and thus share the same base structure; the high-level algorithmic representation of the control loop of the RNNMPC, see algorithm 3 remains practically identical to that of the LSRMPC, see algorithm 2. On a lower level, however, essential differences arise regarding their cost functions and constraints; the different problem formulations not only implement different cost functions and constraints, but the RNNMPC also requires symbolic expressions where the LSRMPC did not. Specifically: the RNNMPC's model constraint is implemented by a symbolic expression representing the RNNARX-model, (2.30c). Furthermore, that same symbolic expression is included in the cost function, (2.30a). The symbolic expression may not be evaluated prior to optimization, meaning that later solving the optimization problem requires differentiating symbolic expressions.

The above-described symbolic qualities of the RNNMPC mean the optimization problem during the control loop must be solved by a programmatic framework that supports symbolic optimization. *CasADi* is one such framework, tailored to solve optimization problems containing symbolic expressions *efficiently* - both linear and nonlinear[54]. CasADi achieves this by defining the symbolic expressions as computation graphs and finding the corresponding gradients by means of *automatic differentiation.* Note that automatic differentiation differs from symbolic differentiation, but associated details fall outside the scope of this thesis. The interested reader is instead referred to [54] for further details regarding automatic differentiation.

CasADi allows formulating tailored symbolic expressions and assigning these as constraints to any optimization problem. Enforcing the model as a constraint is then reduced to loading the trained model's parameters into the RNNMPC formulation, and assigning them to a function defined on the same form as the model itself. Though the trained model is defined only as an MLP, the open-loop multi-step prediction of (2.30c) is implicitly implemented when the constraint by defining the constraint for the desired amount of future time steps. Note that when using CasADi for one or more constraints, all other constraints, as well as the cost function, must be defined in the CasADi framework as well.

CasADi offers two different APIs for implementing optimal control problems: parametric form and via the *opti*-stack[55]. The latter is a high-level abstraction layer of the prior. I chose to use the opti-stack, as its simplified API made maintaining the code base for this thesis easier and more streamlined.

CasADi has built-in compatibility with the open-source nonlinear interior point-based solver *IPOPT*[56]. The interested reader is referred to [56] for details regarding the underlying theory and implementation of IPOPT, as these details fall beyond the scope of this thesis. Following the above arguments of symbolic expression compatibility and efficiency as reasons to choose CasADi as the symbolic mathematical framework within which to im-

Algorithm 3: Pseudo code for the program flow which implements the simulation of control by means of the RNNMPC. This is not an exhaustive guide to implementing an RNNMPC. For exhaustive details, the interested reader is referred to the open-source code at [43].

```
Input:
    1. digital system representation (FMU)
    2. initial actuation state $\boldsymbol{u}_0$
    3. system configuration (tuned parameters, time step size $\Delta t$, simulation length $t_{fin}$)
    4. reference value trajectory $\boldsymbol{y}_{ref,0:t_{fin}}$
    5. input history length $m_y$
    6. output history length $m_u$
    7. NNARX-model $\hat{\boldsymbol{f}}_{MLP}(\hat{\boldsymbol{y}}_{k+i:k+i-m_y}, \boldsymbol{u}_{k+i-1:k+i-m_u}, \boldsymbol{u}_{k+i})$
Output:
    1. optimal trajectory of actuation values
    2. corresponding trajectory of system outputs
Warm-start: apply $\boldsymbol{u}_0$ to FMU until $\max(m_u, m_y)$ time steps after steady-state is
  achieved;
$t \leftarrow 0$;
$k \leftarrow 0$;
Retrieve current and past outputs $\boldsymbol{y}_{k:k-m_y}$ from the FMU;
while $t < t_{fin}$ do
    Update constraints and costs according to $\boldsymbol{y}_{ref,t}$, $\boldsymbol{y}_{k:k-m_y}$ and $\boldsymbol{u}_{k:k-m_u}$;
    Apply constraints and costs to optimization problem;
    Call solver on updated optimization problem;
    Retrieve optimal change in actuation $\boldsymbol{\Delta u}_k^*$;
    $\boldsymbol{u}_k^* \leftarrow \boldsymbol{u}_{k-1}^* + \boldsymbol{\Delta u}_k^*$ Apply optimal actuation $\boldsymbol{u}_k^*$ to FMU;
    Time-shift: $\boldsymbol{u}_{k-1:k-m_u} \leftarrow \boldsymbol{u}_{k:k-m_u+1}$;
    Time-shift: $\boldsymbol{y}_{k:k-m_y+1} \leftarrow \boldsymbol{y}_{k-1:k-m_y}$;
    Retrieve current output $\boldsymbol{y}_k$ from the FMU and add to $\boldsymbol{y}_{k-1:k-m_y}$;
    $\boldsymbol{y}_k \leftarrow \boldsymbol{u}_k^*$;
    $t \leftarrow t + \Delta t$;
    $k \leftarrow k + 1$;
end
```

plement the RNNMPC, choosing IPOPT as the solver for the RNNMPC implementation becomes the natural choice.

Note that the calculations used in the RNNMPC are not with respect to normalized data. Conversely, the training, validation and test data sets were all normalized prior to training the model. Hence, all data must be normalized within the RNNMPC to ensure consistency with the model, and the RNNMPC's output optimal change in actuation must be denormalized before being applied to the FMU. Elsewise, the model would handle data very unlike what it has previously seen, and must be expected to yield highly different performance than during training, validation and testing.

3.3.4 Tuning the RNNMPC

When implementing the RNNMPC, there are several figures, both static and tunable, whose values must be determined. Combining the parameters as required by the RNNMPC problem formulation in (2.30) with the system values as described in Section 3.1.3, yields the configuration of the RNNMPC's static parameters presented in Table 3.6.

In addition to the static parameters, the tunable parameters must be determined. Following (2.30), the RNNMPC's tunable parameters are $\mathbf{Q}$, $\mathbf{R}$, N and $\boldsymbol{\rho}$. Note that the prediction and control horizons were chosen to be equal, $H_p = H_u = N$. As with the tuning of the LSRMPC: even though the tuning of the RNNMPC is not the core subject of this thesis, it should be addressed for the sake of well-performing control. I used grid search also in the tuning of the RNNMPC. Both the justification of the choice of using grid search, as well as the way in which I employed grid search, were completely similar to the procedure described in Section 3.2.2; $\boldsymbol{\rho}$ was defined as a high value, then the horizon was chosen as a compromise between computation times and performance, before lastly a grid search was performed on $\mathbf{Q}$ and $\mathbf{R}$.

Static parameters	Value
$\boldsymbol{y}_{lb}$	$\begin{bmatrix}0 & 0\end{bmatrix}$
$\boldsymbol{y}_{ub}$	$\begin{bmatrix}18537 & 349\end{bmatrix}$
$\boldsymbol{u}_{lb}$	$\begin{bmatrix}0 & 0\end{bmatrix}$
$\boldsymbol{u}_{ub}$	$\begin{bmatrix}100 & 10^4\end{bmatrix}$
$\boldsymbol{\Delta u}_{lb}$	$\begin{bmatrix}-0.55 & -166.7\end{bmatrix}$
$\boldsymbol{\Delta u}_{ub}$	$\begin{bmatrix}0.55 & 166.7\end{bmatrix}$
$\boldsymbol{\epsilon}_{\boldsymbol{y},lb}$	$\begin{bmatrix}0 & 0\end{bmatrix}$
$\boldsymbol{\epsilon}_{\boldsymbol{y},ub}$	$\begin{bmatrix}10^6 & 10^6\end{bmatrix}$

Table 3.6: The static parameters of the system, encoded into the relevant variables of the RNNMPC problem formulation. The values for $\boldsymbol{y}_{lb}$, $\boldsymbol{y}_{ub}$, $\boldsymbol{u}_{ub}$ and $\boldsymbol{u}_{ub}$ are directly based off Table 3.2.

Tunable parameters
$\mathbf{Q} \in \mathbb{R}^{2\times 2}$
$\mathbf{R} \in \mathbb{R}^{2\times 2}$
$H_p \in \mathbb{R}$
$H_u \in \mathbb{R}$
$\boldsymbol{\rho} \in \mathbb{R}^{2\times 1}$

Table 3.7: The parameters of the RNNMPC problem formulation which must be tuned.

Once the tuning process was complete, the RNNMPC was tested on reference profiles as described in Section 3.1.2, completely similar to the testing of the LSRMPC.

3.3.5 Notes on computation times

The computation times associated with running the control loop of the RNNMPC vary depending on hardware capabilities, programmatic efficiency in the implementation as well as the choice of solver. Again, however - as with the LSRMPC - these are factors external to the specific RNNMPC problem formulation and may be addressed without regarding the specific parametric values of the RNNMPC. The factors which instead stem from the RNNMPC problem formulation itself are the value of the horizons N, as well as the values of the hyperparameters m_u, m_y and η which determine the size of the model; the complexity of the model is highly impactful on the complexity of the optimization problem to be solved at each time step. Thus, the choice of horizon and the final choice of model are interconnected and must be assessed in tandem. Note that, since the computation times associated with employing a trained model in an RNNMPC cannot be known prior to actually employing it, the choice of model in Section 4.2.2 is interconnected with the results observed in Section 4.3.1 and Section 4.3.2 in the sense that a model of lower complexity must be assumed to result in lower computation times. The final tuning of the RNNMPC is presented and assessed in Section 4.3.1.

Observed computation times as a function of horizon sizes and model hyperparameters are presented in Section 4.1.2.

Chapter 4

Results

This chapter presents the results retrieved from the implementational developments presented in chapter 3. Section 4.1 presents the results obtained during tuning of the LSRMPC, as well as the final control results obtained on a test reference profile to assess its final target tracking capabilities. Section 4.2 presents the results from the search for the set of optimal hyperparameters for the model, as well as the chosen model's predictive capabilities on a selection of test data sets. Section 4.3 presents the results obtained during tuning of the RNNMPC, as well as the final control results obtained on the same test reference profile as that of the LSRMPC, in order to assess its final target tracking capabilities.

4.1 Linear step response MPC

4.1.1 Tuning and testing of the LSRMPC

The tunable parameters of the LSRMPC were derived according to the procedure explained in Section 3.2.3. After determining H_p and H_u to be 120, a total of 5 rounds of grid search were required before finding a decent tuning for $\mathbf{Q}$ and $\mathbf{R}$, which in turn could be fine-tuned based on intuition. The resulting configuration turned out as summarized in Table 4.1.

Tunable parameters	Tuned values
$\mathbf{Q}$	$\begin{bmatrix} 0.0001 & 0 \\ 0 & 0.1 \end{bmatrix}$
$\mathbf{R}$	$\begin{bmatrix} 1000 & 0 \\ 0 & 0.02 \end{bmatrix}$
H_p	120
H_u	120
$\boldsymbol{\rho}$	$\begin{bmatrix} 10000 & 10000 & 10000 & 10000 \end{bmatrix}^\top$

Table 4.1: The final configuration of tunable parameters of the LSRMPC problem formulation.

The choice of this tuning was based on the specifications listed in Section 3.1.1; it provided

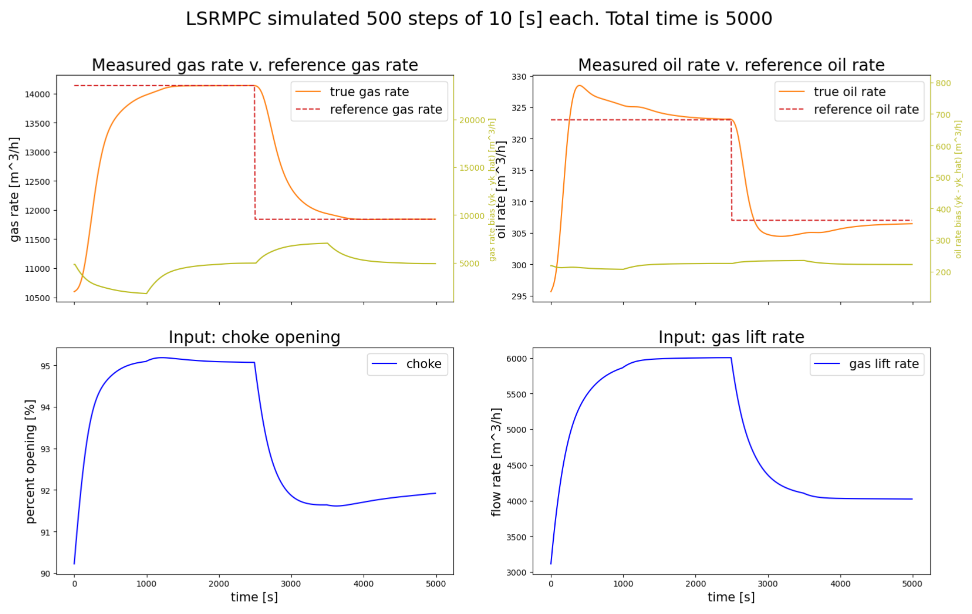


Figure 4.1: The LSRMPC's control performance over the reference profile used for tuning. Prediction error is given by a separate y-axis in green.

the best trade-off between least *overshoot*, most *accurate*, and quick control performance among the tested candidates. The control performance over the reference profile chosen for tuning is shown in Figure 4.1. The control behaviour looks well-performing for gas rate, both reaching and settling at the references relatively quickly, as well as achieving this without any overshoot. Differently, the control behaviour is worse with respect to oil rate; significant overshoot occurs, and actually reaching them happens slowly. Note the smooth actuation profiles for both actuators, however; this is highly desirable. The only sub-optimalities with respect to the control specifications are that the choke is not maxed out before the gas lift rate is utilized, and that the gas rate seems somewhat prioritized over the oil rate. Note also the very high model prediction error - shown by the green line and the corresponding y-axis on the right-hand side - revolving around 4000 for the gas rate and 200 for the oil rate.

After the tuning process, the LSRMPC was tested on more extensive reference profiles, which aim at testing the LSRMPC according to the goals specified in Section 3.1.2. Figure 4.2 and Figure 4.3 demonstrate the control performance results. Note that the system starts at steady-state values matching the initial references due to an initial warm-up phase (not depicted), as described in Section 3.1.2.

Figure 4.2a demonstrates the control performance for high-valued reference profiles and Figure 4.2b for low-valued reference profiles. Note that, even though the theoretically possible domains for the output gas and oil rates have a span according to the minimum and maximum values presented in Table 3.2, *high-valued* and *low-valued* reference profiles are considered, for the sake of control performance in this thesis, regions matching those of the reference profiles in Figure 4.2a and Figure 4.2b, respectively. That low-valued reference profiles do not lie around the minimum values of both gas and oil rates has its roots in the fact that a well usually operates around significantly higher values than the minimum ones during actual control, and occupies the regions around the minimum output values only during start-up.

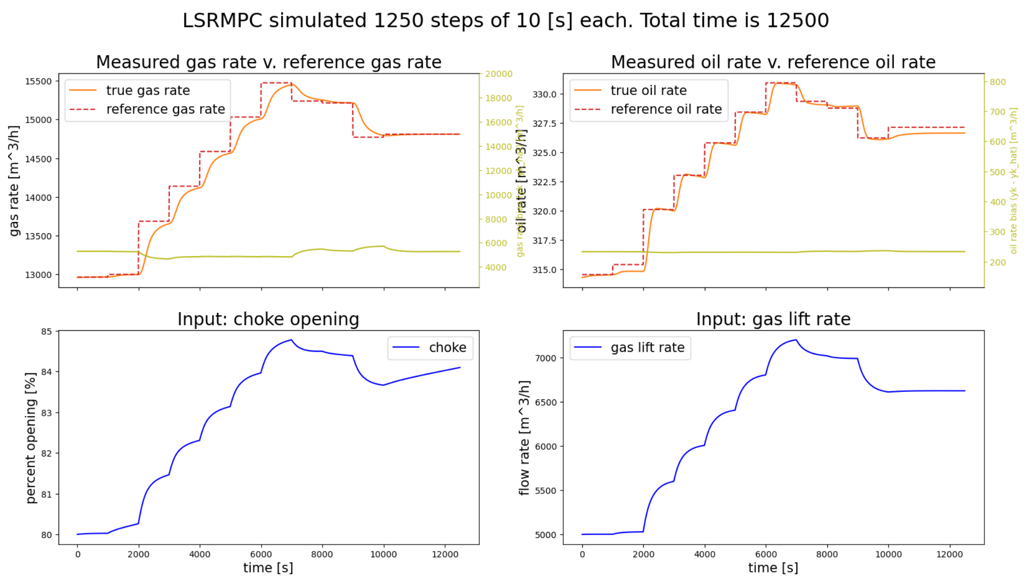
In Figure 4.2a, the reference values for gas rate are all reached, albeit somewhat slowly. The reference values are almost met for the gas rate, with exceptions being the first reference step made, as well as the last. In both of these cases, the gas lift rate can be observed to stay almost static, whereas the choke changes slowly. Completely similar behaviour is observed for the case of Figure 4.2b, though more exaggerated; oil rate references are consistently overshot, and the LSRMPC does not have time to reach the reference before a step in reference is made. This reminisces closely the behaviour observed during tuning, see Figure 4.1. Overall, the control performance is deemed satisfactory for the gas rate, and satisfactory for much of the oil rate, except for the cases in which the reference values are overshot strongly. Associated reasons for the behaviours observed in both of these cases are discussed in Section 5.1.

The reference profiles used in Figure 4.3a and Figure 4.3b are intended to demonstrate the LSRMPC's ability to control gas and oil rate individually, i.e. its ability to control one output to a different value, whilst attempting to maintain a stable value for the other. Initial experiments - not depicted for the sake of brevity - with reference profiles of similar qualities, but different values, showed that the LSRMPC struggled severely with separately controlling the outputs around higher-valued actuation; though the one which experienced a reference shift did not at all reach its reference, both of the outputs were nevertheless moved away from their references. This caused the highly undesirable situation that none of the outputs were able to track their references. Further experiments showed that lowering the working range for the actuators alleviated the issue. As can be seen in Figure 4.3, the new steady-state values are reached for both test cases, though it takes a very long time, and both outputs vary drastically as a consequence of the actuation which changes in response to the reference-change - not just the one to be altered in each individual case. Furthermore, the values for choke can be observed to almost not vary; in both cases, its maximal and minimal values for the simulation were less than 2 [%] different. The LSRMPC's varying ability to control the outputs individually is discussed in Section 5.1.

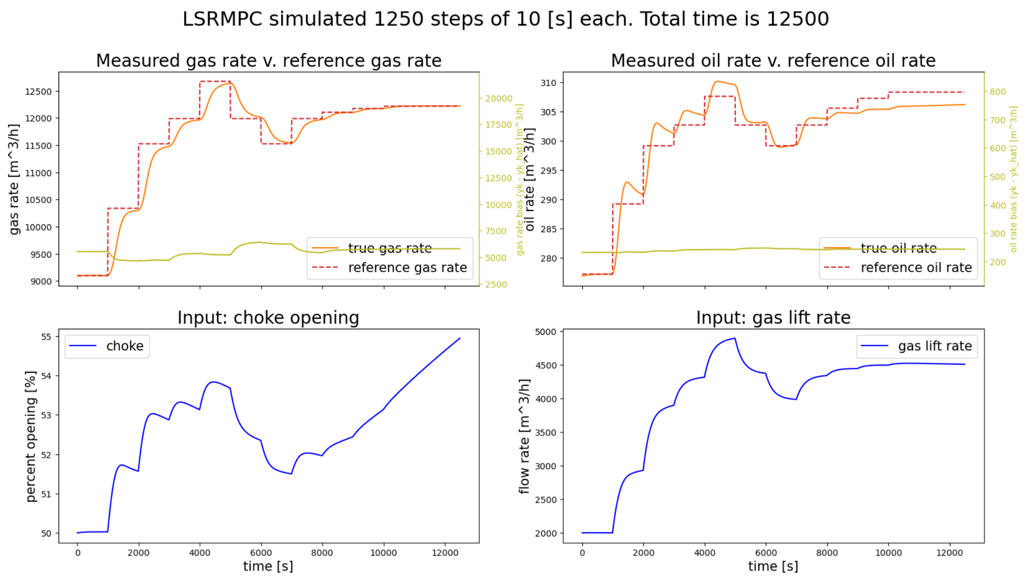
Interestingly, the trajectory of the output gas rate can be observed to almost precisely mirror that of the input gas lift rate in all tests.

4.1.2 Computation times of the LSRMPC

Figure 4.4 presents the computation times associated with running the simulation depicted in Figure 4.2a. The choice of test for which to show the associated computation times was arbitrary. However, it does illustrate a trend that was common for all LSRMPC simulations: the trend of the time spent both updating and solving the OCP at each iteration of the LSRMPC loop remained constant, thus causing a total simulation time that was linear with respect to the length of the simulation in number of iterations. This is as expected, desired, and required, as it means the LSRMPC may be run indefinitely without issues, as long as the time spent on each iteration is within the specified limits.

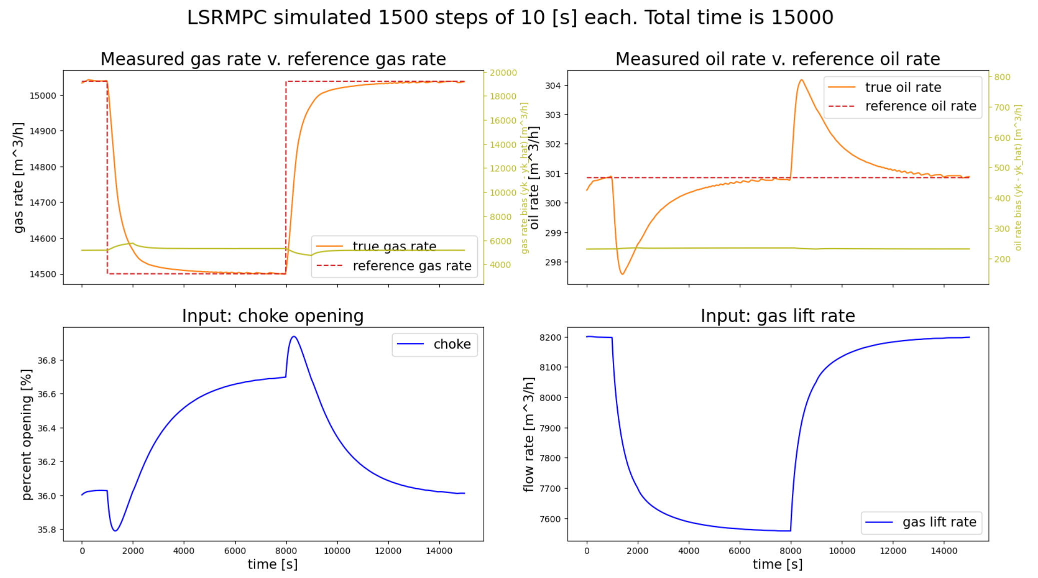


(a) The LSRMPC's control performance tested on high-valued references.

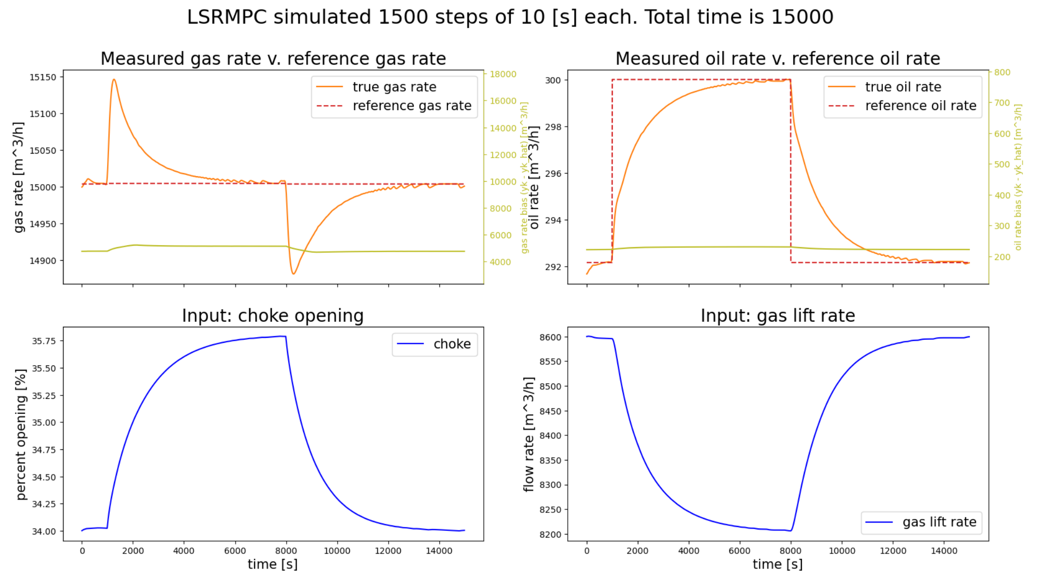


(b) The LSRMPC's control performance tested on low-valued references.

Figure 4.2: The LSRMPC's control performance tested on references in high-valued and low-valued regions, according to the specifications in Section 3.1.2. Prediction error is given by a separate y-axis in green.



(a) The LSRMPC's control performance tested on a reference profile wherein only the gas rate reference is varied.



(b) The LSRMPC's control performance tested on a reference profile wherein only the oil rate reference is varied.

Figure 4.3: The LSRMPC's control performance tested on references in performing a step isolated to gas rate and oil rate, respectively, testing for the ability to control the rates separately. Prediction error is given by a separate y-axis in green.

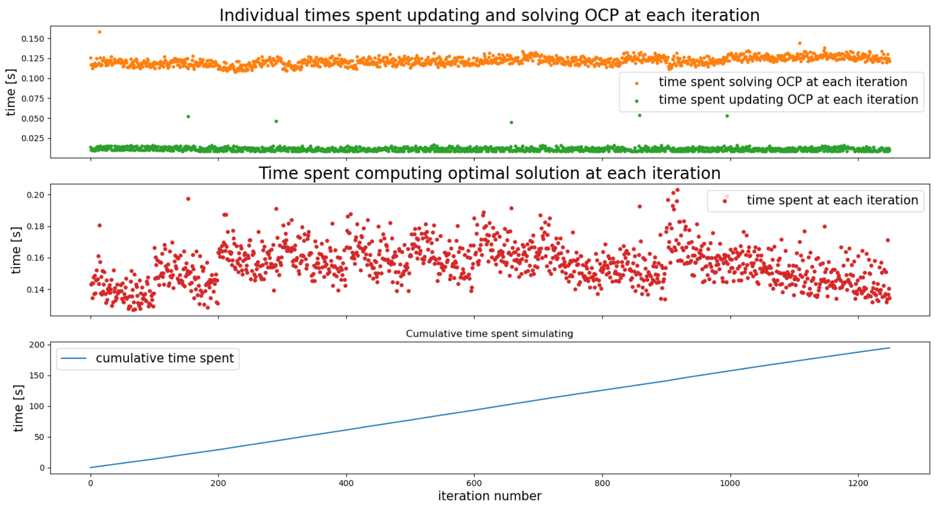


Figure 4.4: The figures illustrate the computation times associated with the LSRMPC simulation depicted in Figure 4.2a. Top: the times spent updating and solving the optimal solution at each iteration. Mid: Total time spent at each iteration. Bottom: The cumulative time spent on the full simulation.

4.2 Results from implementing the model

4.2.1 Hyperparameter grid search results

Here presented are the results from performing the grid search outlined in Section 3.3.2 by training a model on each configuration of hyperparameters from Table 3.5. The hyperparameter configurations are repeated and numbered in Table 4.2 for convenience. In total, since m_u, m_y and η all had 3 different candidate values when accounting for the condition of $m_u \leq m_y$, that meant testing 18 different configurations, all presented in Table 4.2.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
m_u	5			20						50								
m_y	5			5			20			5			20			50		
η	20	40	200	20	40	200	20	40	200	20	40	200	20	40	200	20	40	200
α	0.2																	
p	5																	
β	64																	
λ	0.001																	
μ	0.1																	
e	200																	

Table 4.2: The hyperparameter configurations, numbered according to their specific combinations of hyperparameter values.

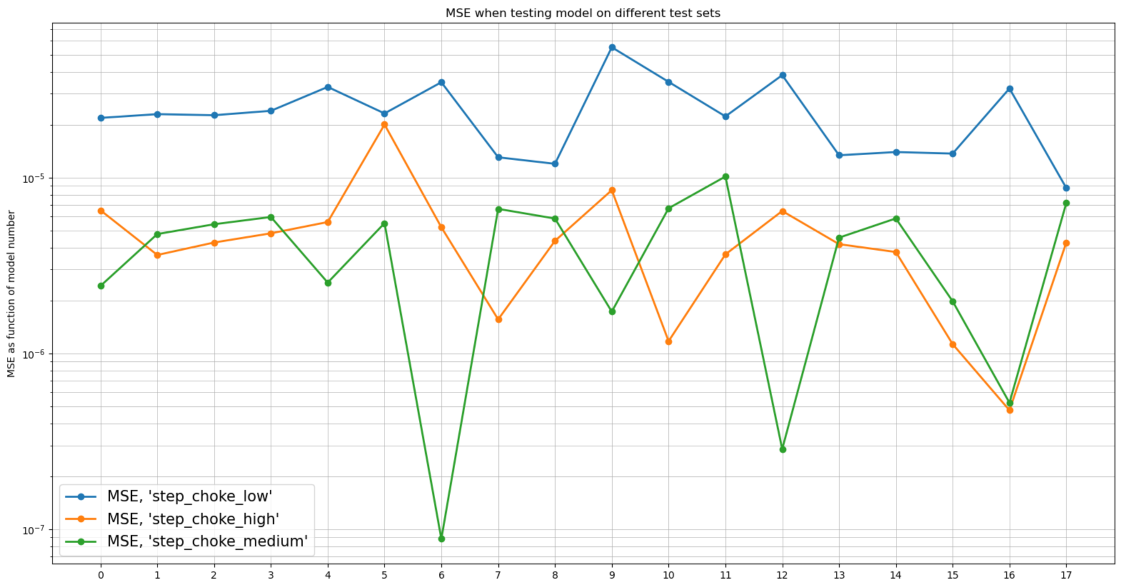


Figure 4.5: The MSE of all the tested models over the step test data sets. The graphs show MSE as a function of hyperparameter number

In order to determine the best-suited model among them after training, I tested them over the three test data sets containing a step in the choke, as described in Section 3.3.2. Specifically, the test data sets all excite the system with a positive 2 [%] step in choke within a low-valued, medium-valued, and high-valued working range. An *indicator* of the models' abilities to handle the system's span of nonlinearity across its different working ranges is thus given. Figure 4.5 displays the MSEs for each of these tests as a function of hyperparameter set.

4.2.2 Choice of hyperparameter set

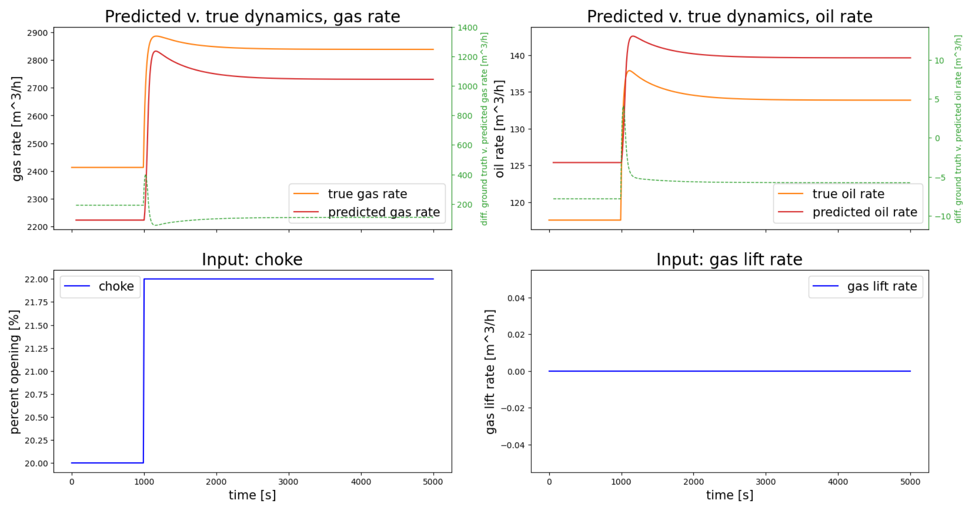
Figure 4.5 only *indicates* which model is the best predictor, as it is simply a measure of total prediction error; it does not say anything regarding the *qualities* of the ways in which the respective models predicted correctly and/or incorrectly. Furthermore, seeking to find a well-performing *light* model - in order to later reduce computational costs resulting from high-valued m_u, m_y and η, see Section 3.3.5 - the choice fell on the model with hyperparameter set #0 in Table 3.5. Its performance on the described test data sets is presented in Figure 4.6, and its performance on the ”staircase” test data set is presented in Figure 4.7, illustrating the predictive accuracy over the system's full working range. Note that, as it is the NNARX model which is tested, the results display a concatenated series of single-step predictions; they do not illustrate the model's ability of multi-step predictions. Though performing seemingly averagely in Figure 4.5, this model exhibits relatively *qualitatively* accurate transient response predictions; its transients experience approximately the same amplitude-shift as the ground truth and at approximately the same speed. The higher MSE thus resulted from the constant prediction error produced during steady-state.

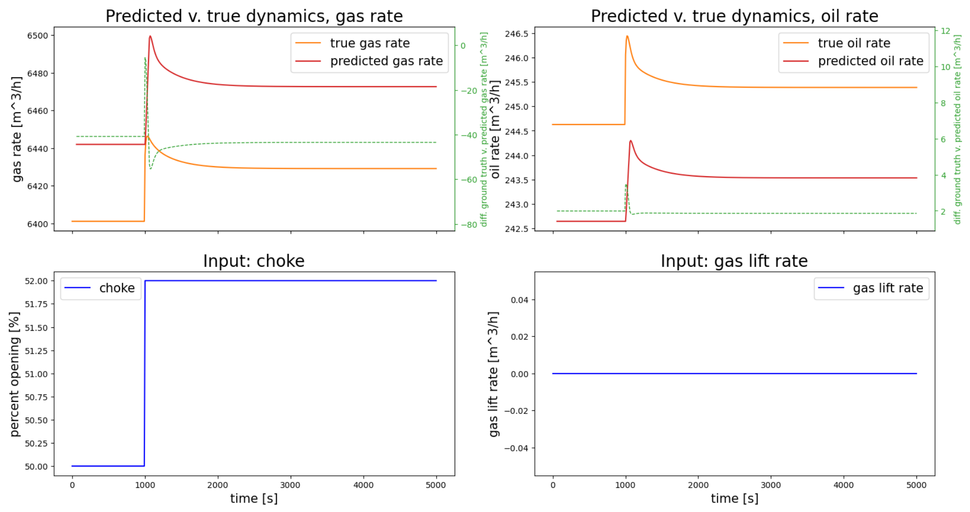
Many of the other models, though producing a lower MSE over all the samples of the test data sets due to their constant prediction error being lower, exhibited transient responses which qualitatively did not match that of the ground truth as closely. As an example, models with a larger hidden layer performed worse due to the shape of their predicted transient trajectories. This caused a spiking behaviour in the prediction error around the step which was more volatile and unpredictable than that of the chosen model. Though this behaviour is present also in the chosen model, it is so to a lower degree; that the prediction matches the truth more closely makes the prediction error spike more short-lived.

The choice of model was, therefore, a tradeoff between less volatile prediction error around the step and higher steady-state prediction error. Recall from Section 2.1.2 ((2.8) and (2.9)) that an MPC problem formulation may correct for bias in closed-loop control as long as it contains some model of model prediction error. In the cases presented in Section 2.1.2, the model prediction error is assumed constant. The prediction errors exhibited by the chosen model indicate that this may be a fair assumption in both low-valued, medium-valued, and high-valued working ranges during steady-state. If the spike in prediction error does not present a too-large issue, this indicates that the chosen model may prove to perform well across all working ranges once later integrated into the RNNMPC, results of which are later presented in Section 4.3.1.

A note on the training of the chosen model. Figure 4.8 depicts the development of the training and validation errors from the training of the chosen model. Note the discrepancy between the training and validation errors; the training error E_{train} starts high and then plummets for every subsequent epoch, while the validation error E_{val} starts and stays low. This indicates that, while the random initialization of the model's parameters causes the initial E_{train} during training to be high, the parameters are learned very quickly, such that all predictions subsequent to the first epoch have much better accuracy, and all consequent E_{train} are much lower. Since validation is performed after the first training epoch and onwards, it never suffers poor predictive accuracy resulting from the parameters being untrained, and thus E_{val} starts and stays low.

Note also that both E_{train} and E_{val} achieve their lowest values almost immediately - after only the second epoch - with all subsequent errors being only *almost* as low. Reasons for this are revisited and discussed in Section 5.2.


(a)


(b)

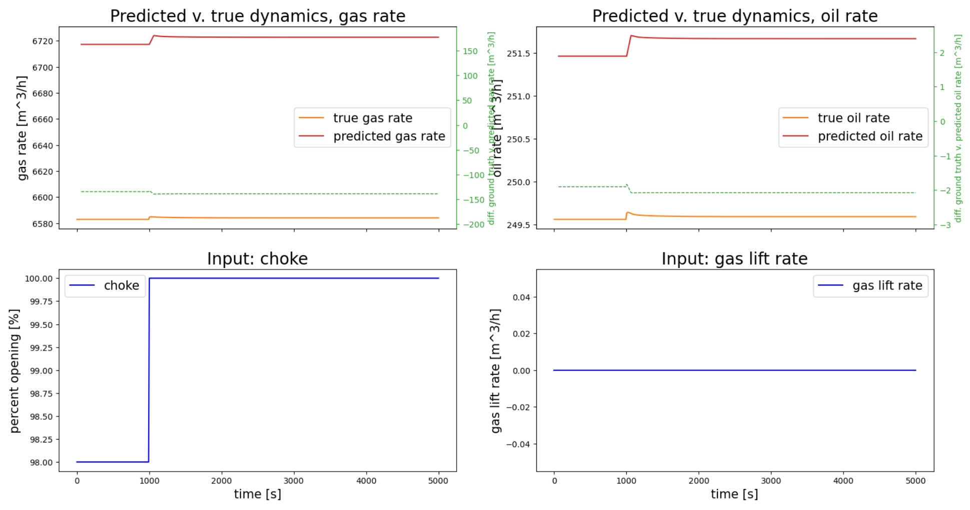

(c)

Figure 4.6: The chosen model's predictions for steps in the choke in low-, medium- and high-valued working ranges, respectively, with gas lift kept constant at 0. The orange, red, blue, and green lines are true outputs, predicted outputs, actuated inputs, and prediction errors, respectively.

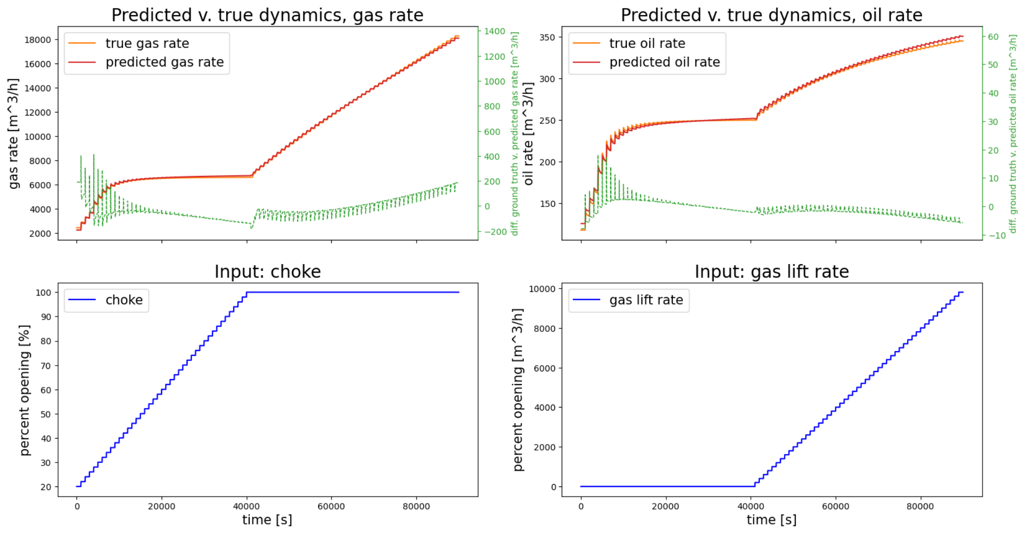


Figure 4.7: An input profile resembling a "staircase" in both choke and gas lift rate, designed to drive the system to maximum gas and oil rates by prioritizing choke.

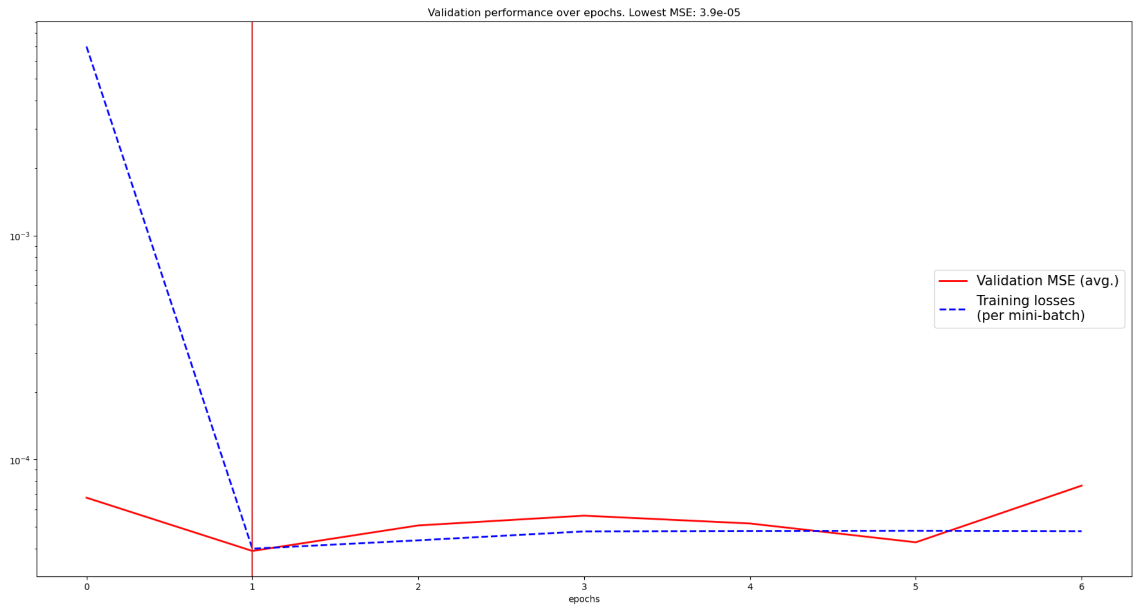


Figure 4.8: The training and validation errors during training of the chosen model. As expected, the training error decreases asymptotically towards zero, while the validation oscillates somewhat more.

Also important to note is the implications actually carried within the choice of hyperparameters. Among the chosen model's hyperparameters are $m_u = m_y = 5$ - see Table 4.2. This carries the implication that only $5\Delta t = 50[s]$ of historic actuation and corresponding dynamic response is passed as information to the model. As is evident from all sub-figures in Figure 4.6, the transient times of step responses in a nonlinear system vary depending on its current working range; Figure 4.6c shows a very brief transient, whereas a same-sized step cause longer-lasting transients for Figure 4.6a and Figure 4.6b. This has the implication that only $50[s]$ of information about the system's past necessarily becomes insufficient to describe the lasting effects of a step in the input for most of the system's working ranges. Development and testing showed that such a lack of information caused the system to experience a larger amplitude in the prediction error. On the other hand, however, it seemed to cause the system to predict *faster* transient responses.

4.3 Recurrent neural network MPC

4.3.1 Tuning and testing of the RNNMPC

The tuning of the RNNMPC was performed according to the procedure outlined in Section 3.3.4. After determining the highest feasible horizon to be $H_p = H_u = N = 100$, the grid search spanned only 4 iterations, before settling on a best result for **Q** and **R**. Some further intuition-based refinement led to the final tuning configuration as summarized in Table 4.3.

Tunable parameters	Tuned values
$\mathbf{Q}$	$\begin{bmatrix} 1000 & 0 \\ 0 & 4000 \end{bmatrix}$
$\mathbf{R}$	$\begin{bmatrix} 125200 & 0 \\ 0 & 62600 \end{bmatrix}$
H_p	100
H_u	100
$\boldsymbol{\rho}$	$\begin{bmatrix} 10000 & 10000 \end{bmatrix}$

Table 4.3: The tunable parameters of the RNNMPC problem formulation and their final tuned values.

As with the choice of tuning for the LSRMPC, the chosen tuning was, among all tested ones, the best-performing based on the specifications listed in Section 3.1.1. The grid search for a good tuning was performed over the same reference profile as was done with the LSRMPC for comparability, and the results are depicted in Figure 4.9. Though both clear overshoot and some oscillatory behaviour are present - mostly in the oil rate - the chosen tuning still represents the best found candidate with respect to minimizing these aspects while still controlling towards a reference trajectory. Note the seeming stationary error in the oil rate for the first half of the sequence, and the similar error observed in the gas rate for the second half of the sequence - in addition to the control being somewhat slow. Though the actuators experience some vague oscillations, this is not deemed critical, as they vary relatively slowly over time.

In total, the presented results from tuning are not very positive; none of the control goals of Section 3.1.1 are met to a satisfactory degree. Nevertheless, they indicate that the implemented RNNMPC indeed might be able to *fully* track the given reference trajectories for both outputs simultaneously. This indication is strengthened by the following paragraphs, which present more extensive testing of the RNNMPC. Note that the RNNMPC also was initiated by a warm-up phase (not depicted), in which the steady-state values were driven to match the initial references.

The fully tuned RNNMPC was tested on the same reference profiles as the LSRMPC for comparability. The results are mixed. While Figure 4.10a proves that the RNNMPC

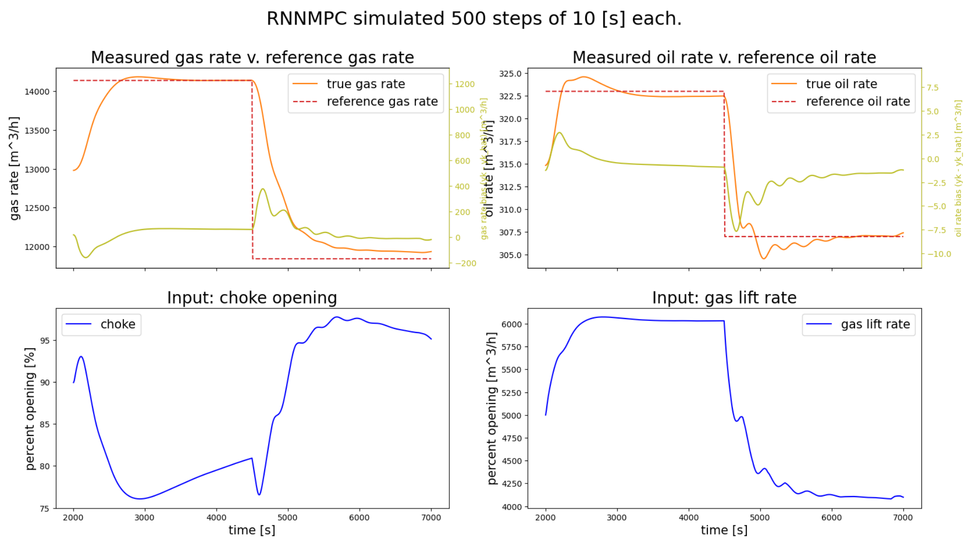


Figure 4.9: The RNNMPC's control performance over the same reference profile used for tuning the LSRMPC. Prediction error is given by a separate y-axis in green.

is able to track the reference in gas rate relatively accurately, and even quicker than the LSRMPC, this positive control performance comes at the cost of an oil rate which both over- and undershoots its reference - almost without ever successfully tracking it. Furthermore, this behaviour also clearly prioritizes the gas lift rate over the choke, thus not adhering to the goals in Section 3.1.1, while also causing some mild spikes for the choke's behaviour.

The control behaviours observed for Figure 4.10b are the same as in Figure 4.10a, only further exaggerated. Here, rapid oscillations can be observed for both the choke and the gas lift rate when the choke is below 50 [%], further causing rapid oscillations also in the gas and oil rates. Not does this make the RNNMPC barely reach the references for gas rate, but it also causes the already relatively poor oil rate control to become worse. Furthermore, the oscillatory input profiles would in a physical application be deemed completely detrimental, as they would quickly wear out and ruin the actuators. This observed control behaviour is not at all in line with the goals outlined in Section 3.1.1. Also symptomatic of severe issues was that some experiments even showed that the optimization problem became *infeasible* when applying reference profiles causing the optimal solutions to lead the choke sufficiently below 50 [%], *in spite* of the slack embedded within the MPC problem formulation.

When testing the RNNMPC for the ability to control the outputs separately, some of the results are more positive. Figure 4.11a shows that the RNNMPC is *almost* able to accommodate the step in reference, as the gas rate is controlled to steady-state *just* below the reference value. When prompted to return to the original gas rate reference, the RNNMPC achieves this efficiently, though the oil rate takes long to settle. As with the LSRMPC, control of the gas rate is not achieved completely separate from the oil rate; the oil rate is significantly perturbed by the sudden shift in actuation.

Note also that the steady-state error in gas rate occurs not only after the step in reference, but also at the very beginning of the simulation; the gas rate is controlled to lie *just* above the reference, in spite of no external motivation to do so.

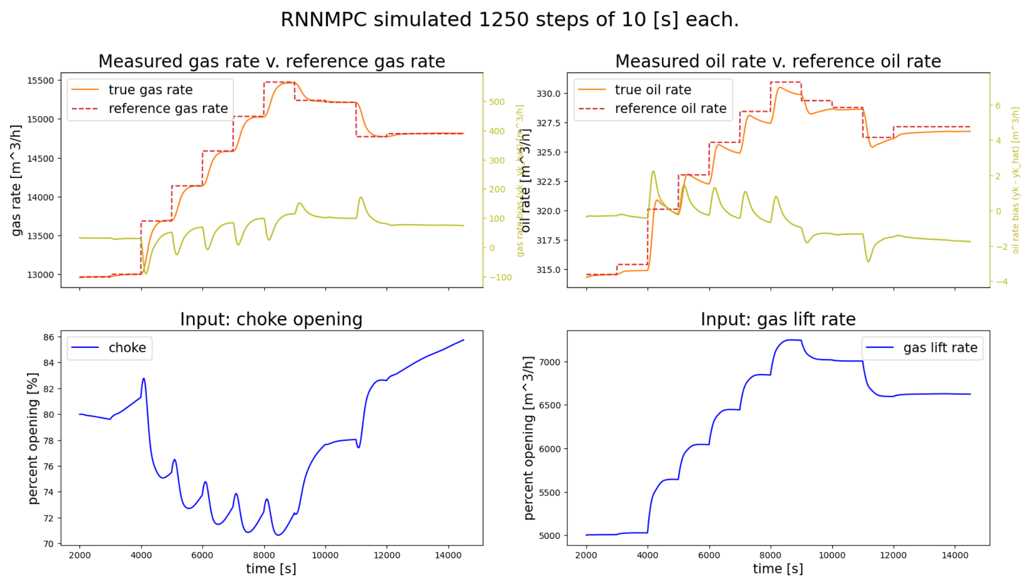
Lastly, controlling the oil rate separately was also tested, and Figure 4.11b depict the results. This case is especially interesting, as it is evident the choke is almost maxed out, and the gas lift rate nevertheless remains steady. As with the LSRMPC, all the tests of the RNNMPC indicate that the trajectory of the output gas rate follows the trend of the gas lift rate almost identically, suggesting a tightly linked connection. Raising the gas lift rate higher would thus increase also the gas rate further. That the gas lift rate then is *not* raised higher in order to help the oil rate reach its reference when the choke is almost saturated, clearly shows that the chosen tuning of the RNNMPC prioritizes gas rate more heavily than the oil rate.

In summary, though the RNNMPC *is* able to control the outputs to some given references, substantial amounts of its observed control behaviour are very poor, and significant priority seems to be given to the gas rate. Potential reasons include the tuning values, but even more likely the model prediction error. This, along with potential remedies, is further discussed in Section 5.3.1.

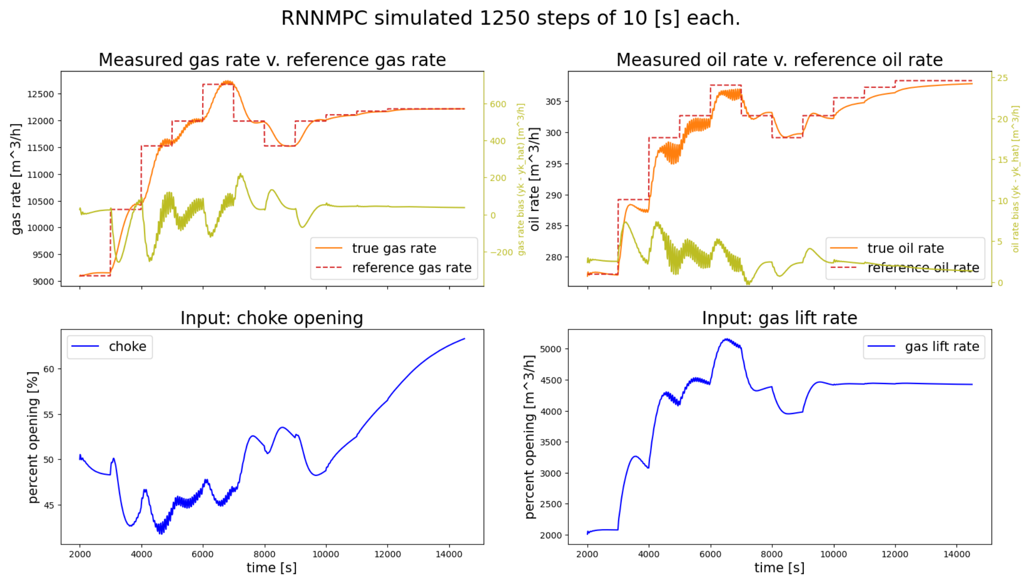
Noteworthy is that that model prediction error - shown by the green line and the corresponding y-axis on the right-hand side - is lower than the LSRMPC's with orders of magnitude; while the LSRMPC's model prediction error revolved around 4000 for gas rate and 200 for oil rate, it is centered almost around 0 for both the gas and oil rates. The important difference, however, is that the RNNMPC's model prediction error behaves much more volatile than that of the LSRMPC; especially so locally around the steps in actuation as well as for intervals where the choke is below 50 [%]. This is likely an extremely important factor contributing to the overall poor performance. More in-depth discussions, covering its potential sources and remedies, are provided in Section 5.2.1.

4.3.2 Computation times of the RNNMPC

Figure 4.12 depicts the computation times associated with simulating the RNNMPC, both cumulative and at each iteration. The ideal case would be that in which the solution requires constant computation time across iterations - with some expected stochastic variation - meaning the cumulative computation time would increase linearly, as is the case for the LSRMPC, see Section 4.1.2. Instead, Figure 4.12 shows that simulating the RNNMPC over any given reference profile consumes approximately linearly increasing time across iterations. The linear growth is an estimate, and the exact order of growth of the time spent per iteration is not known. Nevertheless, this has the immediate implication that the cumulative computation time of the full RNNMPC simulation becomes quadratic with respect to the number of iterations run. While the specific times observed will vary depending on the hardware system on which the optimization is performed, this nevertheless quickly causes the control loop to consume prohibitive amounts of time - especially for extensive periods of control. Indeed, in an actual application, where control is run continuously, an increase in time spent for each iteration would accumulate indefinitely, forcing the application to occasionally have to restart, such that the computation times associated with a single time step do not supersede the size of one such time step - e.g. $\Delta t = 10[s]$, as in this thesis. This is clearly undesirable, and there is no inherent reason why any MPC implementation should behave this way. As such, the issue is likely one of implementation, rather than connected to the RNNMPC problem formulation itself.

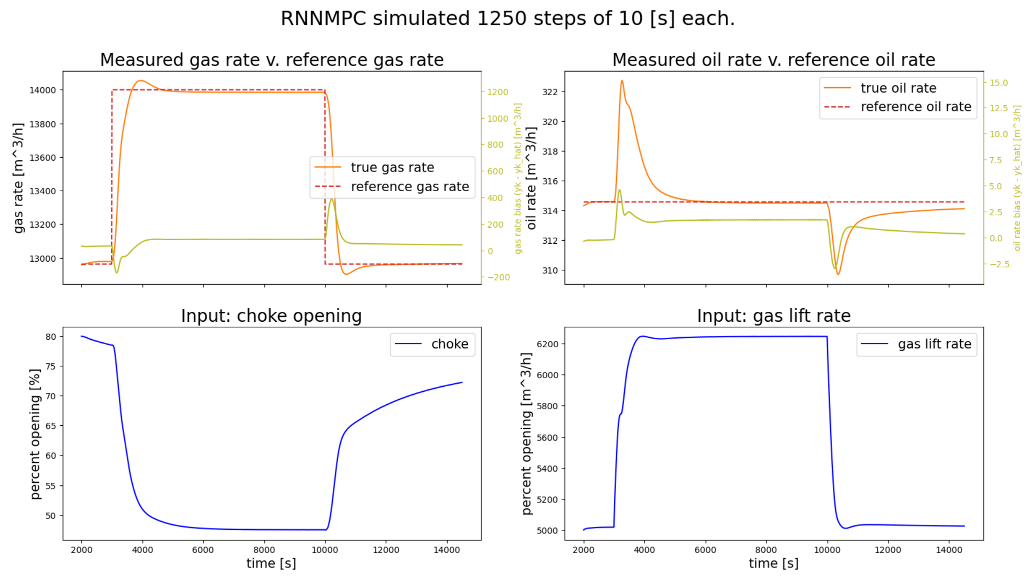


(a) The RNNMPC's control performance tested on high-valued references.

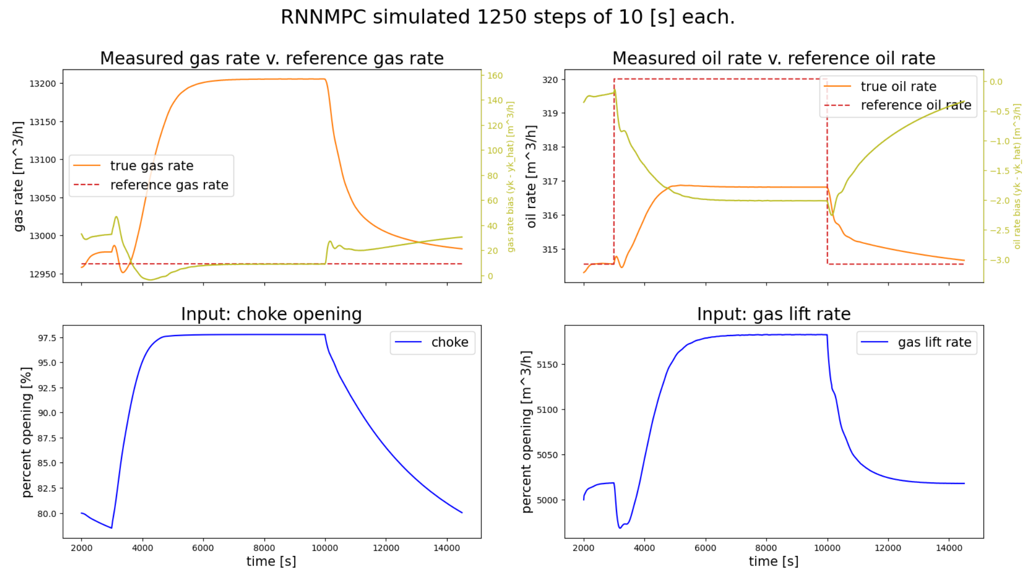


(b) The RNNMPC's control performance tested on low-valued references.

Figure 4.10: The RNNMPC's control performance tested on references in high-valued and low-valued regions, according to the specifications in Section 3.1.2. The reference profiles are identical to the ones to which the LSRMPC was subjected in Figure 4.2 for comparison purposes. Prediction error is given by a separate y-axis in green.



(a) The RNNMPC's control performance tested on a reference profile wherein only the gas rate reference is varied.



(b) The RNNMPC's control performance tested on a reference profile wherein only the oil rate reference is varied.

Figure 4.11: The RNNMPC's control performance tested on references in performing a step isolated to gas rate and oil rate, respectively, testing for the ability to control the rates separately. The reference profiles are identical to the ones to which the LSRMPC was subjected in Figure 4.3 for comparison purposes. Prediction error is given by a separate y-axis in green.

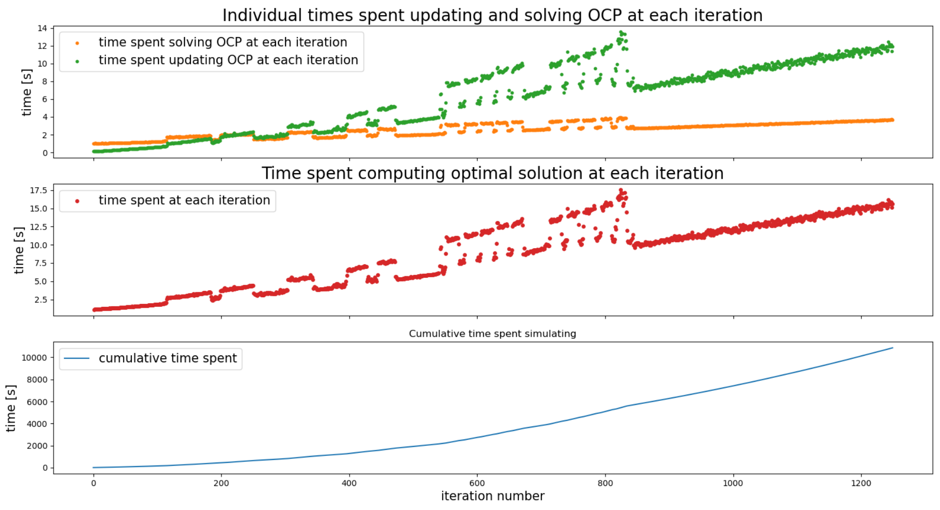


Figure 4.12: The figures illustrate the computation times associated with the RNNMPC simulation depicted in Figure 4.10a. Top: the times spent updating and solving the optimal solution at each iteration. Mid: Total time spent at each iteration. Bottom: The cumulative time spent on the full simulation.

Indeed, the problem was isolated to occur around the step where the constraints of the optimization problem were updated. Updating the full set of constraints used in optimization problems implemented with CasADi's opti-stack is a two-step process: first, all previously set constraints must be cleared, before the full new set of constraints is declared[55] (section 9.1). The step of declaring the full new set of constraints was the singular step within the RNNMPC implementation which consumed increasing amounts of time for each iteration. Suggested improvement regarding this issue is discussed in Section 5.3.3. Additionally to this issue, the step of simply *calling* the solver on the declared optimization problem via the opti-stack also consumed significant amounts of time *before* performing the optimization itself; the larger the optimization problem, the longer this pre-optimization spent. The optimization itself slowed down only slightly with respect to increasing problem size, but does show a slight increase over time in Figure 4.12. This is likely connected to the issue of redeclaring the constraints.

A clear correlation between the model's size and the RNNMPC's running time was observed during grid search. Table 4.4 presents the computation time associated with the first iteration of the RNNMPC simulated for the reference profile in Figure 4.10a for a selection of models tested during the grid search. The time is measured only for the first iteration in order to exclude the issue of the ever-increasing time consumption of re-declaring the optimization problem. Notably, model #17 (the largest) takes $\frac{531.9274[s]}{0.9747[s]} \approx 546$ times longer than model #0 (the smallest). Evidently, the complexity of the model is essential with respect to the computation time associated with solving the optimization problem at each iteration. This is a strong argument in favour of choosing smaller, and thus computationally lighter, models for applications - as previously indicated in Section 3.3.2.

As a further argument for choosing light models, it is of interest to view the relation between model size and computation times in the current implementation more systematically. The model's total amount of parameters, though not directly providing the number of operations performed in the compiled optimization problem, are directly indicative of the number of operations contained within a computation graph representing it within an optimization problem, thus providing an indicator for its computational demand. The gas and oil well system has $n_{in} = 2$ inputs, each of which yields m_u inputs to the model. Additionally, it has $n_{out} = 2$ outputs, each of which yields m_y inputs to the model. Since all models here discussed contain only a single layer, the amount of parameters contained within each of them is the sum of the elements contained within the 2 weight matrices $\mathbf{W}_1 \in \mathbb{R}^{\eta \times \eta_{in}}$ and $\mathbf{W}_{out} \in \mathbb{R}^{\eta_{out} \times \eta}$, as well as the 2 bias vectors $\boldsymbol{b}_1 \in \mathbb{R}^{\eta}$ and $\boldsymbol{b}_{out} \in \mathbb{R}^{\eta_{out}}$. The figures $\eta_{in} = n_{in}m_u + n_{out}m_y$, η and $\eta_{out} = n_{out}$ are the sizes of the input, hidden and output layer, respectively. A general expression for the amount of the parameters within any model trained in this thesis may be derived:

$$\begin{aligned}
\xi &= \underbrace{\eta\eta_{in} + \eta}_{\text{hidden layer}} + \underbrace{\eta_{out}\eta + \eta_{out}}_{\text{output layer}} \\
&= (\eta_{in} + 1)\eta + \eta + 1\eta_{out} \\
&= ([n_{in}m_u + n_{out}m_y] + 1)\eta + (\eta + 1)n_{out} \\
&= ([2(m_u + m_y)] + 1)\eta + 2(\eta + 1)
\end{aligned} \tag{4.1}$$

#	0	3	6	7	17
m_u	5	20	20	20	50
m_y	5	5	20	20	50
η	20	20	20	40	200
ξ	462	1062	1662	3322	40602
Time	0.9747[s]	3.4344[s]	10.3405[s]	38.4889[s]	531.9274[s]

Table 4.4: Computation times of the initial iteration of the RNNMPC for a selection of models from the grid search for optimal hyperparameters. The models are directly fetched from Table 4.2, and the non-variable hyperparameters are omitted for brevity.

Noteworthy is also the impactful contribution by the prediction and control horizons $H_p = H_u = N$, set to $N = 100$ during the experiments which lead to the numbers in Table 4.4; a higher-valued N quickly increased the observed computation times, compounding on the effects described above. This has the intuitive explanation that, on top of the model implementing a potentially quite large function, depending on m_u, m_y and η, it is numerically differentiated as many times as the size of N for each iteration of the solver at each iteration of the RNNMPC.

Chapter 5

Discussion

5.1 Performance of the LSRMPC

This section briefly recaps the control performance of the LSRMPC, before detailing underlying reasons and assessing the employed method. Though interesting to assess the LSRMPC's control performance and underlying reasons, it is mainly intended as a later grounds for comparison for the RNNMPC. As such, the discussion of the LSRMPC is kept brief.

The results presented in Section 4.1 show that the LSRMPC is relatively able to control both gas rate and oil rate to their respective outputs, though oil rate experiences somewhat slow control and overshoot, reasons for which are presently discussed.

Recall from Section 3.2.2 that the model was created as a linearization of the system dynamics using a step from choke and gas lift rate of $[50\,[\%], 0\,[\frac{m^3/h}{10s}]]$ to $[52\,[\%], 0\,[\frac{m^3/h}{10s}]]$. As reasoned in Section 2.1.3, a linearized model of a system describes the system best in a neighbourhood around the point around which it was linearized. All the test results shown in Section 4.1.1 include non-zero values for gas lift rate, and almost all include values for choke *not* around $50\,[\%]$. This immediately implies that the model prediction errors should become quite high, which is indeed observed across all test results. Though a more accurate model would likely yield significantly lower model prediction error, its magnitude does not seem detrimental to the control performance. Noteworthy, the model prediction error is very slow-moving. So much so, that it may be approximated constant - which in the implementation it is. Even still, it is imaginable that e.g. low-pass filtering the model prediction error could improve control performance, as it would make the optimizer not react strongly in response to what is in reality model inaccuracy.

Tuning-related issues. The most important reason for the LSRMPC's observed control performance is thus hypothesized to be its tuning. Recall especially the values for $\mathbf{Q}$ and $\mathbf{R}$ presented in Table 4.1. The values in $\mathbf{Q}$ indicate that the deviation from the reference in the oil rate should be penalized more strongly than that in the gas rate. Nevertheless, the results indicate the opposite occurring; the target tracking for gas rate against the neglect of target tracking for oil rate in Figure 4.2b provides a suitable example. This is likely due to the values in $\mathbf{R}$ penalizing changes in choke *much* more strongly than those of gas lift rate. As briefly mentioned in Section 4.1.1, all the performed tests indicate that the gas rate trajectory seems to closely mirror that of the gas lift rate. Intuitively, it stands to reason that increasing the rate of gas injected into the fluid also necessarily increases the resulting output gas rate, strengthening the claim that gas rate and gas lift rate necessarily are tightly linked. Thus, the LSRMPC is able to alter the gas rate more directly and controlledly by means of using the gas lift rate, and the total penalty of achieving the gas rate references more efficiently by means of the gas lift rate does not become high. Conversely, altering both the choke and the gas lift rate in order to achieve the oil rate reference efficiently would likely cause a high penalty from altering the choke too quickly. Another clear example is how the LSRMPC achieves target tracking in both Figure 4.3a and Figure 4.3b by altering mostly the gas lift rate and the choke only very little. The result is an LSRMPC which clearly prioritizes the gas rate over the oil rate.

A high output gas rate then does not necessarily indicate high production of gas, as significant parts of that output gas rate necessarily is a contribution from the gas lift rate, supplied from the system, i.e. an expense. Furthermore, though the final tuning of the LSRMPC yielded relatively adequate control performances, it took into account neither the goal of prioritizing oil rate over gas rate as output, nor prioritizing the use of choke over gas lift rate as input - both of which were underlying goals from Section 3.1.1. This illustrates the weakness of the tuning method: grid search.

Notes on grid search. Grid search is an extremely naïve method of tuning, and yields little exploration of the controller's performance across various combinations of tuning configurations, due to the quickly rising computational complexity associated with it. Since the grid search here employed was based on *coarsely* spanning the possible configurations of tuning values, and then limiting further searching to the most promising sub-region of values based on some criterion as a means of reducing computational load, there exists no guarantee that other potentially better-performing candidate sub-regions - other local optima - are not overlooked. Alternatively, both performing a sufficiently exhaustive grid search, as well as processing all the resulting simulation data, is prohibitively resource-demanding. Not only would it consume vast amounts of computation time, but in order to ensure that the final tuning does not suffer discrepancies with the top-level goals, each individual result must be judged accordingly - not only on target tracking abilities. Judging each individual result based on higher-level qualities, such as the degree of volatile behaviour in the inputs, or the degree to which one output or input is prioritized over the other, would likely require involving human judgement, unless rigorous mathematical descriptions of said higher-level qualities are derived. The grid search was ratified by the rationale that it could be performed as a background process, searching in parallel to other work. Requiring a human decision-maker to judge every grid search result defeats this rationale, and does not necessarily even result in sufficiently good search results, due to the issue of overlooking sub-regions.

All the above-made arguments are amplified with increasing dimensionality of the search; the more variables are included, the more resource-costly performing such a search becomes. Grid search is thus not a recommended strategy. Instead, genetic algorithm-based or random searches could be expected to perform better, as previously mentioned in Section 3.2.3.

5.2 The NNARX-model

Section 4.2.2 shows that the NNARX-model used in this thesis is able to approximately predict the general trends of the system's dynamic response to step inputs. This proves, as a proof-of-concept demonstration, that the NNARX-modelling paradigm is able to capture much of the nonlinear behaviour of the system apparent in a sampled data set. However not without fault: the predicted transient response is consistently slower than the real transient response, and the model consistently produces steady-state prediction error, which stays relatively constant *within* working ranges, but varies *across* working ranges. Potential causes of and remedies for these issues are the subjects of discussion in the following sections. Specifically, Section 5.2.1 discuss how the way the data set is synthesized may cause issues upon training a model, Section 5.2.2 discusses the role of implications arising from architectural choices, and Section 5.2.3 discusses how the thus far presented reasons for loss of predictive accuracy compound to cause issues in multi-step predictions when the model is connected recurrently.

5.2.1 Qualities of the training data set

Recall from Section 4.2.2 the development of the training and validation errors for the chosen NNARX-model (Figure 4.8). That E_{train} and E_{val} both achieved their minima after only two epochs of training, indicates that the training data set is quite large, relative to the number of parameters to be shaped - depicted in Figure 3.2, it contains 200.000 samples. If the training data set does not contain samples *fully* representative of the data-generating process from which the model will receive samples during application, training on a too-large data set can easily cause the model to become overfitted. When vast amounts of potentially misinformative data are exploited to shape the model's parameters, they will be fit very well to the wrong process. After reaching its optimum, E_{train} stays around similar values for further epochs, and even increases somewhat. That it does not monotonically decrease, as expected from Section 2.3.1, is likely due to the noisy convergence properties observed by SGD-based algorithms, due to the stochasticity involved in the random sampling of each mini-batch. Building on the argument that a large and potentially misinformative training data set might cause overfitting, this thesis hypothesizes that the reason the tested models of Table 4.2 with a larger hidden layer performed worse than models with a smaller hidden layer, might be due to their increased capacity facilitating the already somewhat present overfitting.

Thus the question of the training data set's applicability is raised: *does* the training data set accurately represent the full spectrum of dynamic behaviour which is desired that the NNARX-model should be able to predict accurately? Recall that the training data set used in this thesis was not sampled from the true process, but synthesized from a pre-existing digital model of the system - the FMU. The generality of the synthesis was based on the *assumption* that a semi-randomly asynchronously walking APRBS would excite the system across all frequencies, thus yielding transient responses fully representative of the system's broad spectrum of dynamic responses. However, the assumption was likely somewhat lacking. The following arguments justify how.

First recall that [5] argues that, while an APRBS-based excitation signal is well-suited for identifying linear systems, it is not necessarily suitable for detecting the dynamics of a nonlinear system, since the resulting input-output relations will have "holes" in the nonlinearities which they span. As generally holds: any data set containing samples of continuous processes can not be assumed to be exhaustively descriptive of the process in question, due to the infinite resolution. This thesis argues that the significance of such "holes" depends on the system's degree of nonlinearity in the region in question; if the system is highly nonlinear, much dynamic behaviour is overlooked by assuming linearity, and vice versa. In other words, the success of modelling nonlinear dynamic behaviour based on an APRBS-based excitation signal depends on whether the signal's *resolution* in a region matches that region's degree of nonlinearity. Whether the data set utilized in training the NNARX model in this thesis is a sound basis on which to identify the system's nonlinearities accurately, depends on whether it contains a higher rate of samples around highly nonlinear regions than linear regions, such that the stronger nonlinearities are represented with a higher rate, thus affecting the learning procedure more. Note that this does not imply that a high rate of samples around more linear regions causes worse predictions, it simply implies that sufficient attention must be paid towards the more nonlinear regions, lest they be approximated too coarsely. Investigating the predicted nonlinear behaviour for different working ranges of the system might yield answers to whether this has been the case.

The transient responses were indeed not modelled very accurately; recall how the prediction error spikes in all test data sets of Figure 4.6, resulting from the system's transients being consistently predicted too slow-moving. This seems to be *somewhat* more the case for the step made in a low-valued working range - see Figure 4.6a; the spike in prediction error seems to take somewhat longer to settle to being constant than in the other cases. This implies that the training data set does not describe the nonlinearities in the lower-valued working ranges *quite* as well as in the higher-valued working ranges, as the prediction is wronger for a longer time. Note also that the steady-state prediction errors are the largest in this case. In total, the results imply that the NNARX model "understands" the lower-valued working ranges worse than the other working ranges. By looking at Figure 3.2, there are very few samples present covering the same region as the step in Figure 4.6a. When comparing all tests of Figure 4.6, the profile in Figure 4.6a looks no less nonlinear than the others. The fact that values around that region are then underrepresented in the training data set, substantiates that the model has been trained on a data set which is *unbalanced*; the training set is simply too low-resolution around the lower-valued working ranges.

Thus, the initial assumption that the semi-randomly walking APRBS excitation signal might be suitable likely lacked the distinction of semi-randomly walking with an even frequency across all working ranges of the system, as the lower-valued ranges became neglected. Furthermore, this has the implication that such a data synthesis should be based on a mapping of the system's degrees of nonlinearity across its working ranges, such that the semi-random walk might be somewhat skewed towards the more nonlinear regions, thus sufficiently representing them in the final data set. Not performing such a mapping effectively leaves the data synthesis blind to the "holes" that [5] describe might occur from using an APRBS signal to excite a nonlinear system.

In summary, the training data set seems to be unbalanced: it represents the system's nonlinear dynamic behaviour insufficiently around the system's lower-valued working ranges, which likely causes worse predictive accuracy of the trained model for lower-valued working ranges. Also, the sheer size of the data set has likely caused some overfitting to the training data. There has been developed no guarantee that the training data is representative of the data which the model encounters during application. Section 5.2.3 outlines how this may be a major contributory factor for the poor control behaviours observed from the RNNMPC in Section 4.3.

5.2.2 Architectural implications

Recall from Section 4.2.2 that the lengths of transient times vary within a system depending on the current working range. One major weakness with the employed architecture is then that the values for m_u and m_y must be fixed after training. Since the model should be applicable for the full range of samples for which it is trained, this is obviously problematic, as the model should ideally take in only what information is relevant to its prediction problem, not more, not less.

If too much information is consistently present, the model may learn correlations which are not there - as an example, recall the wolf/husky-classification case of [32], briefly presented in Section 2.3.1. Instead of a causal relationship between outdated outputs and future output, the model should learn the causal relationship between the change in actuation and the consequent outputs. However, if too long sequences of data are present, that might mean that the model learns that there exists some correlation between the output before a change in actuation and the output after the effect from the change in actuation has passed.

Looking at the step responses observed in Figure 4.6, it is likely not the case in this thesis that $m_u = m_y = 5$ were chosen too large, as most transients seem longer than $5\Delta t = 50[s]$. A different explanation may then stem from the model not receiving *sufficient* information, with m_u and m_y being chosen too low. If the model receives too little information, parts of its predictions must necessarily be wild guesses. It then stands to reason that the model will necessarily miss its target. This would cause errors in predictions of both transients and steady-states.

While the unbalanced data set's insufficient representations of certain working ranges caused a loss of predictive accuracy in the trained model, so did likely the choice of m_u and m_y. The faults in the choice of m_u and m_y are two-fold. Firstly, better combinations of values which might have existed were not identified, due to the coarseness of the employed grid search; the grid search used to find the optimal set of hyperparameters suffered the exact same disadvantages as the grid search used to find the optimal tuning of the LSRMPC - as discussed in Section 5.1. Secondly, since the model is built on the NARX assumption, the values for m_u and m_y could likely never be precisely correctly chosen, as they would ideally be required to vary across the system's working ranges, which is an impossibility for the employed architecture. Based on the results from [16], however, there is reason to believe that the grid search provided the largest fault and that better-performing values for m_u and m_y could have been identified, as the NARX-modelling paradigm has been proven in the context of RNNMPC before.

5.2.3 Implications of implicit RNNARX multi-step modelling

The model in this thesis performs in reality only single-step predictions, and extending such single-step predictions to the multi-step case by means of recurrence is based on the assumption that this is at all viable for the model in question. Section 2.4.1 presents exposure bias as a notable issue related to implementing an RNN as has been done in this thesis: by implicit recurrence as a model constraint of an RNNMPC over its prediction horizon. Exposure bias arises due to the discrepancy observed between true and predicted values. Since the model does not predict perfectly, see Figure 4.6, the recurrence during closed-loop application causes the future input to follow a different distribution than that of the data on which it was trained. Exposure bias must then be assumed to occur as soon as on the first recurrence, and further be assumed to propagate for each further recurrence.

Additionally, though the excitation signal described in Section 3.3.1 and Figure 3.2 spans many types of inputs, the commanded actuations from the optimization during RNNMPC application are still not guaranteed to follow the same distribution as that of the training data set, to which the model might be overfitted. This causes a further potential for discrepancy between input during closed-loop application and the training data set, implying further prediction error, adding to the effect that inputs to the model do not follow the same distribution as the training data during recurrence. Both of these effects then add to the total effect of exposure bias in the model once it is applied in the RNNMPC, and cause potentially very poor dynamics prediction in closed-loop application. However, this thesis hypothesizes that the sum of the issues the model faces during deployment in the RNNMPC is likely the very cause of the detrimental control performance challenges experienced by the RNNMPC. Note that a method of investigating the degree to which this is a problem is to assess the model's multi-step predictions outside of application and compare it against the model's multi-step prediction which underlies the optimal solution of the RNNMPC at every iteration.

Section 2.4.1 proposes scheduled sampling as a means to avoid the exposure bias the model suffers after being trained by means of teacher forcing. This thesis strongly suggests that scheduled sampling be employed in order to remedy the compounded effects of exposure bias.

Alternatively, the training method BPTT provides a suitable alternative to teacher forcing. While BPTT might imply a more complex both training procedure and model implementation, and cause training to consume larger amounts of time, it eradicates the issue of exposure bias directly, as the model is trained to tackle sequential prediction from the start. However, if using BPTT, the argument is raised that not much is gained by selecting a simple architecture. The question is then raised, whether the model should be replaced by a different model which is empirically proven more capable, such as the LSTM or GRU. This topic is further discussed in Section 5.3.2.

5.3 Recurrent neural network MPC

This section discusses the reasons for the RNNMPC's observed control performance, related to the underlying model, the RNNMPC's tuning, as well as the grid search employed to determine the tuning. Some model architectural alternatives are then presented, as well as how they might improve the modelling capabilities, and consequently the RNNMPC's control performance.

5.3.1 Performance of the RNNMPC

The results presented in Figure 4.10 and Figure 4.11 show that the RNNMPC is partly able to control both the gas rate and the oil rate to desired values. Compared to the LSRMPC, the RNNMPC actually controls the gas rate to its reference value faster in the case of a higher-valued reference profile. However the RNNMPC suffers several issues, such as significant over- or undershoot in the oil rate, as well as severe oscillatory behaviour in both inputs and outputs for the working range of choke below 50 [%]. Especially the latter is control-wise an unacceptable issue.

Model-related issues. The issue likely arises mostly due to modelling inaccuracies. Recall that the RNNMPC suffers model prediction errors much lower than that of the LSRMPC. Nevertheless, the RNNMPC observes significantly worse control performance. The ways in which inaccuracies affect the performance are two-fold.

Firstly, the spikes observed in prediction error upon steps in actuation are very significant - see Figure 4.6 for examples. This is different from the LSRMPC where, in spite of the prediction errors being large, they seem to vary according to a more constant profile. The RNNMPC calculates the optimal actuation sequence at each iteration with "blind trust" in the model, correcting only for an assumed constant bias, see (2.30). Due to the spikes in prediction error, this assumption does *not* hold as well as it does in the case of the LSRMPC. As with the LSRMPC, this thesis hypothesizes that the associated issues might be somewhat mitigated by adding a low-pass filter to the RNNMPC's formulation of the bias model. This would allow the optimization within the RNNMPC to calculate solutions that do not account for precisely these prediction error spikes, and would likely yield a less volatile control profile.

Secondly, the exposure bias resulting from even minor prediction errors likely causes the model to predict very wrongly as early as the first recurrence. Compounded with spiky prediction error, this effect is likely to only become worse. Figure 4.6 and Figure 4.7 demonstrate the model's performance on data which matches the behaviour of its training data set fairly well, though depict only a concatenated collection of single-step predictions - completely analogous to predictions made during training, and not representative of the recurrent predictions made during application. Conversely, the RNNMPC bases the optimal actuation at each time step on a multi-step prediction, effectively trusting blindly the model to be descriptive of the system - see (2.30c) - and solves for the optimal actuation thereafter. In reality, the model is only somewhat descriptive of the system for single-step predictions and likely performs significantly worse for multi-step predictions due to exposure bias. This is likely the reason for the little "bump" observed in gas rate the beginning of the simulation depicted in Figure 4.10b; the exposure bias effect occurs even if no change in control is performed, which provides the RNNMPC reason to believe that the system will deviate, even at steady-state, thus prompting action and causing a steady-state offset. A similar effect can be observed in Figure 4.9.

Though the predictive capabilities of the model are quite similar across all working ranges, the issues observed are amplified for low-valued working ranges for the inputs. While the MPC paradigm is known for its robustness to poor models, it is expected that the control performance will degrade along with any degradation of the underlying model. With the model being sufficiently misinformative of the true system dynamics, the optimization problem eventually becomes infeasible. Note that the reference trajectories of Figure 4.10a and Figure 4.10b are both followed (at least broadly speaking), though much better in the case of the prior, as the case of the latter suffers many undesired qualities, strong oscillations being the most prominent. It is reasonable to assume that the model is simply *sufficiently* poor for the working ranges in Figure 4.10b compared to Figure 4.10a, that the RNNMPC experiences the two above-explained issues strongly enough, such that the issues observed arise. This hypothesis is strengthened by the fact that some reference trajectories led to an infeasible optimization problem, as briefly noted in Section 4.3.1. A slightly better model would likely alleviate the issues somewhat, and a much better model would likely mitigate the issues completely.

This thesis therefore hypothesizes that the key to solving the observed control issues is tightly linked with simply improving the model. Improvements might be achieved by reducing the model's predictive errors and eradicating its exposure bias as much as possible, as discussed in Section 5.2, or by simply changing the chosen model architecture, as later discussed in Section 5.3.2.

Tuning-related issues. The contribution of the tuning of the RNNMPC must also be considered. It is unsurprising that the gas rate seems to be the prioritized output in both Figure 4.10 and Figure 4.11. Recall the values presented in Table 4.3. The weights in $\mathbf{Q}$ penalize deviations in oil rate only *somewhat* more than they do deviations in the gas rate, while the gas rate operates around significantly higher values than the oil rate. This causes the penalty on deviations from the oil rate reference to become effectively smaller than the penalty on deviations from the gas rate reference.

Noting the values in $\mathbf{R}$, the weight on the choke is double the value of the weight on the gas lift rate. This implies that the RNNMPC should change the choke as little as possible, such that using the gas lift rate as actuation becomes prioritized. This behaviour is further strengthened by the same tight link between gas rate dynamics and the gas lift rate as observed in the LSRMPC - see Section 5.1 - which makes the RNNMPC more able to control by using gas lift rate. Consequently, the RNNMPC will prioritize tracking the gas rate reference, and secondly, it will prioritize using the gas lift rate to make changes to the output. None of these behaviours are in line with the goals presented in Section 3.1.1. Also, the matter of separate controllability almost mirrors that of the LSRMPC. The step in reference is met in Figure 4.11a, but not without the oil rate responding strongly to the actuation. In the case of Figure 4.11b, the new reference is not even met, as the preference for gas rate is strong enough that the RNNMPC does not make the required step in oil rate.

Note additionally that it is inherently difficult to tune an MPC such as the RNNMPC here implemented to prioritize one actuator over the other when weighing the *change* in an actuator, and not its direct actuation value. Weighing the change in a variable does not carry any direct consequence for that variable's starting or final value. Hence, weighing the choke heavily implies in this case that the choke should be reluctant to change, meaning it evidently becomes the least prioritized actuator. Conversely, if the penalty on the use of choke is very low, the choke will fluctuate much, and not have any incentive to maximize before using *some* gas lift rate - as is the desired behaviour. Hence, if the RNNMPC should prioritize choke by only using gas lift rate after the choke is saturated, as outlined in Section 3.1.1, the RNNMPC problem should be reformulated. This thesis provides two suggestions as to how.

A reference value for the actuator may be added to the cost function. That way, deviation from that reference may be punished in exactly the same way as is done for the outputs, in turn providing the RNNMPC incentive to achieve the output references, while also doing this as closely to the desired actuator values as possible. However, this method requires references for the actuators provided from a higher level; for instance, if the output references may be achieved with for instance $[\text{choke}, \text{gas lift rate}] = [90\,[\%], 0\,[\frac{m^3/h}{10s}]]$, then using a static reference of choke $= 100\,[\%]$ - as an incentive for the RNNMPC to saturate choke before using gas lift rate - would instead incentivize the RNNMPC to *allow* deviations from the output rates in order to saturate the choke. As the system outputs must be assumed prioritized higher than the exact actuation values, it follows naturally that the reference actuation values must be determined on a higher level, based on the desired output references. Such methods exceed the scope of this thesis and are not discussed. However, if any such method is present in later applications, this method is recommended.

Alternatively, soft constraints bounding the actuators to the desired regions may be added additionally to their hard constraints. By using a small weight on the corresponding slack variables, the RNNMPC is incentivized, though not required if it provides sub-optimality, to adhere to the desired soft bounds. Note that these must be added additionally to the actuators' hard constraints, as the hard constraints describe regions outside which the physical equipment is not allowed to operate.

While this thesis hypothesized in Section 3.3.4 that $\mathbf{Q}$ and $\mathbf{R}$ would be the main factors determining the RNNMPC's control behaviour, potential gain may have been overlooked in not investigating further the values of H_p and H_u. As noted in (2.7) in Section 2.5.2, the horizons could have been chosen separately, though this was not done for this thesis. In the case where the size of H_u matters less than the size of H_p, lowering H_u could allow increasing H_p while retaining comparable computation times; recalling from Section 2.1.1 that an MPC's robustness stems from the receding horizon principle, increasing H_p would likely yield improved performance. Note that, due to the structure of (2.30c), this would require a definition of $\boldsymbol{\Delta u}_{k+H_u+1:k+H_p-1}$, such that $\boldsymbol{u}_{k+H_u+1:k+H_p-1}$ would be defined as well. Adding a solution such as the one implemented by (2.9h) is suggested as a trivial way of ensuring the RNNMPC problem formulation is well defined for all $i \in [0, H_p - 1]$.

Notes on grid search. The chosen tuning stems from a grid search procedure entirely similar to that of the LSRMPC, meaning the exact same issues apply: Dimensionality makes the search prohibitively resource-demanding unless the search grid is divided very coarsely - as was done in this thesis. Then, however, many local optima within the grid might be overlooked, and the resulting tuning might be sub-optimal, even though it was deemed best. Consequently, the best result from the grid search did not represent a tuning in line with the specified goals.

Due to the issues related to computation times, described in Section 4.3.2, and for comparability with the LSRMPC, the grid search was performed for the RNNMPC over the same reference trajectory as was done for the LSRMPC. Notably, these references stay quite high-valued, and thus none of the issues related to low-valued working ranges, as described above, were detected during the search for a good tuning, which meant the issues were not accounted for during development.

While both of the above issues could have been corrected by performing more extensive grid searches and for larger working ranges of the system, there was not enough allocated time to do new and extensive searches to find optima more in line with the goals. The initial hypothesis was that the search would yield satisfactory results by simply running in the background. However, grid search again proved to spend much time yielding not great results, which additionally broke with the specified goals. Grid search is not recommended.

A summary of the RNNMPC's control performance. The initial tests of the RNNMPC's control performance were promising. As the tests of the underlying model proved somewhat satisfactory, and a decent-looking control profile was found during tuning, the RNNARX-based MPC proved its conceptual feasibility. Though it proved worse than the LSRMPC on almost every measure, it *did* provide better gas rate control in Figure 4.10a. Noteworthy, that example shows vast usage of gas lift rate instead of choke, which is undesirable. Furthermore, Figure 4.10b uncovered significant issues. These likely stem from an imbalance in the training data set, architectural flaws of the chosen model, flaws in the chosen training method of the model, as well as the employed tuning method of the RNNMPC itself. Though these issues are not without probable solutions, their totality, as is, nevertheless cause the RNNMPC to be, in spite of its proven conceptual functionality, infeasible.

Presently, alternative approaches are briefly explored.

5.3.2 Alternative modelling approaches

This section discusses alternative modelling approaches, and how they would remedy weaknesses in the chosen modelling approach.

Though the weaknesses in the training data set mentioned in Section 5.2.2 likely cause some of the issues related to the model's predictive performance, they might also be related to the model's architecture. The values of m_u and m_y are determined through a grid search, and not based on observations of the durations of the transient responses of the system. There may then be room for improvement by tailoring the values of m_u and m_y more targeted to the system, even specifically to each input and output. Even with such an effort made to determine the ideal values for m_u and m_y, however, recall from Section 5.2.1, that m_u and m_y cannot be decided perfect for the full working range of the model. Even if the perfect m_u and m_y are determined for specific working ranges of the system, they will not apply for *all* working ranges.

To remedy this issue, recall the observation made in Section 1.2: the NARX-modelling paradigm predicts future values based on evaluating all relevant data simultaneously, while the GRU architecture, applied in [12], instead accumulates transient information over time, thus facilitating a more adaptive dynamics integration. The latter is also common to the LSTM architecture.

On the basis of [16], it is not given that a NARX-based model is inherently unfit for the purpose of implementing an RNNMPC such as in this thesis. Many issues unrelated to the architecture specifically have been uncovered, and mitigating their sources might improve the RNNMPC's control performance significantly. It was chosen for its simplicity, and based on NARX being a long-known method for data-driven approximation[5]. Nevertheless, the LSTM or GRU architectures are acknowledged for their generally better results than the more basic RNN-based structure[17] and represent promising options.

Recall from Section 2.4.3 that such recurrent architectures are more complex, and also require the BPTT algorithm for training. Implementing and training them might thus prove somewhat more difficult. Furthermore, when considering the results presented in Section 4.3.2, there is reason to believe that a platform intended to run RNNMPC based on an LSTM- or GRU-model would require more computationally capable hardware, due to the increased complexity, than the one on which the work for this thesis has been developed - see Table 3.1.

Lastly, the tested and discussed modelling methods are all based on black-box modelling. Alternatively, on the basis of the results obtained in [18], a grey-box approach might prove feasible also in the case of the subsea gas and oil well for this thesis. This is especially interesting, as it would reduce the role of the NN-based model to fewer aspects of the modelling, effectively making the model smaller and computationally lighter, without necessarily losing predictive accuracy.

5.3.3 Computational regards and potential for improvement

The issue of the computation time spent at each iteration of the RNNMPC should be made constant, instead of approximately linearly increasing. Since the RNNMPC problem formulation requires re-declaring all constraints at each iteration, this issue is unavoidable when using CasADi's opti-stack, as the implementation follows the documentation. The remedy then involves porting the implementation to a framework which does not cause the same issue. As CasADi offers a lower-level API besides the opti-stack, the issue might be promptly resolved by porting the implementation to that lower-level API. This should indeed be done in order for the RNNMPC implementation to be scalable to longer simulations or even continuous application.

Further computational improvements may be made to the current implementation by the use of parallelization techniques. Recall from Table 3.1, that the desktop computer on which the work in this thesis has been developed offered no such parallelization compatibilities. However, if instead deploying the application on hardware supporting this, e.g. through CUDA as suggested in Section 3.1.3, the computation times associated with the underlying operations performed may likely be significantly sped up. Looking into how parallelization techniques might speed up computations is recommended, as this would allow feasible usage of a wider array of models without inhibitions due to their computational demand.

Chapter 6

Conclusion

This thesis has investigated the feasibility of a model predictive control (MPC) application that uses a nonlinear autoregressive model with exogenous inputs (NARX) implemented as a recurrent neural network (RNN), dubbed *RNNMPC*. The RNNMPC was intended for the flow control of a subsea oil and gas well system, and the results were compared with a linear model-based MPC, dubbed *LSRMPC*. The RNNMPC was implemented successfully, and proved capable of some target tracking, thus serving as a proof-of-concept application, showing that MPC built on a neural network-based model is indeed feasible for such flow rate control. However, significant challenges arose. The desired level of control performance was only observed for high-valued references, in which case the performance was better than the LSRMPC's only in gas rate control, and not oil rate control. In all other cases, the RNNMPC proved far worse than the LSRMPC. Additionally, detrimental qualities such as significant under- and overshoot, as well as strong oscillations in both actuators and output flow rates, were observed. Though the implementation stands as a proof-of-concept, the total feasibility of the specific RNNMPC implemented was thus rejected due to severe such shortcomings.

The data acquisition was somewhat, but not sufficiently, satisfactory. An excitation signal was designed, but without consideration of the differing degrees of nonlinearity across the system's working ranges. The resulting data did not describe the system's nonlinearities fully, and the consequently trained model's predictive performance, in spite of promising initial tests, proved insufficient.

Teacher forcing was used for training the model, on the assumption that the nature of the model allowed organically scaling from single-step to multi-step predictions by means of recurrence. While the tests of single-step predictions provided optimistic results, no countermeasure was implemented against exposure bias, which arose upon multi-step prediction and proved a significant hindrance.

Lastly, the RNNMPC was implemented using CasADi's opti-stack for symbolic mathematics in optimization problems. Though the implementation worked, the constraints needed to be re-declared at every iteration, causing increasing time consumption across the iterations of the RNNMPC. The source for this error is unknown to the author and, to the best of the author's knowledge, to the CasADi community as well. The error implies no conceptual fault with the RNNMPC, but nevertheless causes infeasibility for longer simulations or continuous application.

Chapter 7

Future Work

In order to implement a successful RNNMPC application based on an NARX model, some key considerations must be made. Firstly, the data acquisition must be performed to match the problem at hand; the data sets should be sufficiently descriptive of the system dynamics to be learnt by the model. The author suggests mapping the system's degrees of nonlinearities across working ranges as a way of understanding which working ranges the data sets should be more descriptive of. Secondly, the neural network-based model must be trained appropriately. When implementing the NARX architecture as the modelling basis, *scheduled sampling* is suggested as a modification to the random sampling of the training data set that is usually employed during the training of the model. This might strengthen the model's ability to perform decent predictions, also in light of poor input data. Additionally, in the face of prediction errors which develop non-constantly, a low-pass filter is suggested implemented, in order to facilitate a smoother development of the perceived bias, as this might yield less volatile control in return. Furthermore, if using a similar architecture, future similar works should port the RNNMPC implementation from CasADi's opti-stack to the lower-level API, such that the re-declaration of constraints does not cause computation times to increase across iterations.

Alternatively, future similar works might consider utilizing different neural network architectures as a basis for the RNNMPC implementation. The gated recurrent unit is among the RNN candidates which are also proven in other works considering RNNMPC. The long short-term memory architecture is also popular due to its proven sequence modelling capabilities. By instead utilizing such architectures, the challenges faced during training and implementation in this thesis might be instantly mitigated: Training of such architectures is done via the backpropagation through time-algorithm, which, though computationally heavier than teacher forcing, gives rise to no exposure bias, as sequences are used for training directly. Furthermore, RNNMPC problem formulations using such architectures would likely yield no need to re-declare the constraints of the RNNMPC at every iteration, effectively eradicating any associated issues.

If, however, such architectures prove too computationally demanding for the platform on which the RNNMPC is run, the author suggests investigating the performance of a grey-box RNN. This might prove as efficient as a black-box RNN in predictive accuracy and might prove computationally lighter.

Limited computational resources might additionally be overcome by employing parallelization techniques. The author suggests that parallelization platforms, such as e.g. CUDA, may in such cases be employed to provide a substantial increase in computational capacity.

Bibliography

[1] S Joe Qin and Thomas A Badgwell. 'An overview of industrial model predictive control technology'. In: *AIche symposium series*. Vol. 93. 316. New York, NY: American Institute of Chemical Engineers, 1971-c2002. 1997, pp. 232–256.

[2] D.K.M. Kufoalor, Lars Imsland and Tor Johansen. 'High-performance Embedded Model Predictive Control using Step Response Models**This work is funded by the Research Council of Norway (NFR) and Statoil through the PETROMAKS project 215684, and also by NFR, Statoil, and DNV through the AMOS project 223254.' In: *IFAC-PapersOnLine* 48 (Dec. 2015), pp. 1, 31–60. DOI: 10.1016/j.ifacol.2015.11.073.

[3] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer New York, NY, 2006. ISBN: 978-0-387-40065-5.

[4] Wilson J. Rugh and Jeff S. Shamma. 'Research on gain scheduling'. In: *Automatica* 36.10 (2000), pp. 1401–1425. ISSN: 0005-1098. DOI: https://doi.org/10.1016/S0005-1098(00)00058-3. URL: https://www.sciencedirect.com/science/article/pii/S0005109800000583.

[5] Oliver Nelles. 'Nonlinear Dynamic System Identification'. In: *Nonlinear System Identification: From Classical Approaches to Neural Networks and Fuzzy Models*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 547–577.

[6] Fabio Bonassi et al. 'On Recurrent Neural Networks for learning-based control: Recent results and ideas for future developments'. In: *Journal of Process Control* 114 (2022), pp. 92–104. ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2022.04.011. URL: https://www.sciencedirect.com/science/article/pii/S0959152422000610.

[7] Kurt Hornik, Maxwell Stinchcombe and Halbert White. 'Multilayer feedforward networks are universal approximators'. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[8] Katrine Seel et al. 'Convex Neural Network-Based Cost Modifications for Learning Model Predictive Control'. In: *IEEE Open Journal of Control Systems* 1 (2022), pp. 366–379. DOI: 10.1109/OJCSYS.2022.3221063.

[9] Yu Cao and Jian Huang. 'Neural-network-based nonlinear model predictive tracking control of a pneumatic muscle actuator-driven exoskeleton'. In: *IEEE/CAA Journal of Automatica Sinica* 7.6 (2020), pp. 1478–1488. DOI: 10.1109/JAS.2020.1003351.

[10] Felix Bünning et al. 'Input convex neural networks for building MPC'. In: *Learning for Dynamics and Control*. PMLR. 2021, pp. 251–262.

[11] Tim Salzmann et al. *Real-time Neural-MPC: Deep Learning Model Predictive Control for Quadrotors and Agile Robotic Platforms*. 2022. DOI: 10.48550/ARXIV.2203.07747. URL: https://arxiv.org/abs/2203.07747.

[12] Nicolas Lanzetti et al. 'Recurrent Neural Network based MPC for Process Industries'. In: *2019 18th European Control Conference (ECC)*. 2019, pp. 1005–1010. DOI: 10.23919/ECC.2019.8795809.

[13] Kyunghyun Cho et al. 'On the properties of neural machine translation: Encoder-decoder approaches'. In: *arXiv preprint arXiv:1409.1259* (2014).

[14] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-Term Memory'. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[15] Paisan Kittisupakorn et al. 'Neural network based model predictive control for a steel pickling process'. In: *Journal of Process Control* 19.4 (2009), pp. 579–590. ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2008.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0959152408001388.

[16] Katrine Seel et al. 'Neural Network-Based Model Predictive Control with Input-to-State Stability'. In: *2021 American Control Conference (ACC)*. 2021, pp. 3556–3563. DOI: 10.23919/ACC50511.2021.9483190.

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

[18] Zhe Wu, David Rincon and Panagiotis D. Christofides. 'Process structure-based recurrent neural network modeling for model predictive control of nonlinear processes'. In: *Journal of Process Control* 89 (2020), pp. 74–84. ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2020.03.013. URL: https://www.sciencedirect.com/science/article/pii/S095915241930825X.

[19] Yngve Kippersund. *Towards Neural Network-Based Model Predictive Control*. TTK4550 Specialization Project Report. 2022.

[20] Lars S. Imsland. 'Topics in Nonlinear Control: output feedback stabilization and control of positive systems'. PhD thesis. Norwegian University of Science and Technology, 2002.

[21] Dale E. Seborg et al. *Process Dynamics and Control, 3rd Edition*. Wiley, 2011, pp. 414–427. ISBN: 978-1-118-50671-4.

[22] L. Magni and R. Scattolini. *Advanced and multivariable control*. Pitagora, 2014. ISBN: 9788837119058. URL: https://books.google.no/books?id=d1d4rgEACAAJ.

[23] Mina Kamel et al. 'Model Predictive Control for Trajectory Tracking of Unmanned Aerial Vehicles Using Robot Operating System'. In: *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2017, pp. 3–39. ISBN: 978-3-319-54927-9. DOI: 10.1007/978-3-319-54927-9_1. URL: https://doi.org/10.1007/978-3-319-54927-9_1.

[24] Bjarne Foss and Tor Aksel N. Heirung. *Merging Optimization and Control*. Mar. 2016. ISBN: 978-82-7842-201-4.

[25] Mina Kamel, Michael Burri and Roland Siegwart. 'Linear vs Nonlinear MPC for Trajectory Tracking Applied to Rotary Wing Micro Aerial Vehicles'. In: *IFAC-PapersOnLine* 50.1 (2017). 20th IFAC World Congress, pp. 3463–3469. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2017.08.849. URL: https://www.sciencedirect.com/science/article/pii/S2405896317313083.

[26] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997, p. 2. ISBN: 0070428077.

[27] Liangxiao Jiang et al. 'Survey of Improving K-Nearest-Neighbor for Classification'. In: *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*. Vol. 1. 2007, pp. 679–683. DOI: 10.1109/FSKD.2007.552.

[28] David C. Lay, Steven R. Lay and Judi J. McDonald. *Linear Algebra and its Applications*. Pearson, 2016. ISBN: 978-0-321-98238-4. URL: https://home.cs.colorado.edu/~alko5368/lecturesCSCI2820/mathbook.pdf.

[29] Andrew L Maas, Awni Y Hannun, Andrew Y Ng et al. 'Rectifier nonlinearities improve neural network acoustic models'. In: *Proc. icml*. Vol. 30. 1. Atlanta, Georgia, USA. 2013, p. 3.

[30] A. Helbig, W. Marquardt and F. Allgöwer. 'Nonlinearity measures: definition, computation and applications'. In: *Journal of Process Control* 10.2 (2000), pp. 113–123. ISSN: 0959-1524. DOI: https://doi.org/10.1016/S0959-1524(99)00033-5. URL: https://www.sciencedirect.com/science/article/pii/S0959152499000335.

[31] Diederik P. Kingma and Jimmy Ba. 'Adam: A Method for Stochastic Optimization'. In: (2017). arXiv: 1412.6980 [cs.LG].

[32] Marco Tulio Ribeiro, Sameer Singh and Carlos Guestrin. '"Why Should I Trust You?": Explaining the Predictions of Any Classifier'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1135–1144. ISBN: 9781450342322. DOI: 10.1145/2939672.2939778. URL: https://doi.org/10.1145/2939672.2939778.

[33] David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. 'Learning representations by back-propagating errors'. In: *nature* 323.6088 (1986), pp. 533–536.

[34] *A GENTLE INTRODUCTION TO TORCH.AUTOGRAD*. URL: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html. (accessed: 05.05.2023).

[35] Adam Paszke et al. 'Automatic differentiation in pytorch'. In: (2017).

[36] Michael I. Jordan. 'Chapter 25 - Serial Order: A Parallel Distributed Processing Approach'. In: *Neural-Network Models of Cognition*. Ed. by John W. Donahoe and Vivian Packard Dorsel. Vol. 121. Advances in Psychology. North-Holland, 1997, pp. 471–495. DOI: https://doi.org/10.1016/S0166-4115(97)80111-2. URL: https://www.sciencedirect.com/science/article/pii/S0166411597801112.

[37] Jeffrey L Elman. 'Finding structure in time'. In: *Cognitive science* 14.2 (1990), pp. 179–211.

[38] Paul J. Werbos. 'Generalization of backpropagation with application to a recurrent gas market model'. In: *Neural Networks* 1.4 (1988), pp. 339–356. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(88)90007-X. URL: https://www.sciencedirect.com/science/article/pii/089360808890007X.

[39] Samy Bengio et al. 'Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks'. In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/file/e995f98d56967d946471af29d7bf99f1-Paper.pdf.

[40] Ilya Sutskever, Oriol Vinyals and Quoc V Le. 'Sequence to Sequence Learning with Neural Networks'. In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf.

[41] *Deepwater Horizon oil spill.* URL: https://www.britannica.com/event/Deepwater-Horizon-oil-spill/Environmental-costs. (accessed: 12.05.2023).

[42] *CUDA Toolkit.* URL: https://developer.nvidia.com/cuda-toolkit. (accessed 04.06.2023).

[43] *TTK4900_NNMPC.* URL: https://github.com/yngveki/TTK4900_NNMPC/. (accessed: 12.05.2023).

[44] *Python 3.10.6.* URL: https://www.python.org/downloads/release/python-3106/. (accessed 04.06.2023).

[45] *PyTorch.* URL: https://pytorch.org. (accessed: 12.05.2023).

[46] *The Modelica Association.* URL: https://modelica.org. (accessed 18.05.2023).

[47] *Welcome.* URL: https://jmodelica.org/pyfmi/. (accessed 18.05.2023).

[48] *NumPy.* URL: https://numpy.org. (accessed 18.05.2023).

[49] *Gurobi Optimization, Python.* URL: https://www.gurobi.com/documentation/9.5/quickstart_mac/cs_python.html. (accessed 18.05.2023).

[50] *osqp.* URL: https://osqp.org. (accessed 18.05.2023).

[51] Sourabh Katoch, Sumit Singh Chauhan and Vijay Kumar. 'A review on genetic algorithm: past, present, and future'. In: *Multimedia Tools and Applications* 80 (2021), pp. 8091–8126.

[52] Sigrún Andradóttir. 'A Review of Random Search Methods'. In: *Handbook of Simulation Optimization.* Ed. by Michael C Fu. New York, NY: Springer New York, 2015, pp. 277–292.

[53] PyTorch. *pytorch/torch/nn/modules/linear.py.* https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/linear.py. 2022.

[54] Joel A E Andersson et al. 'CasADi – A software framework for nonlinear optimization and optimal control'. In: *Mathematical Programming Computation* 11.1 (2019), pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

[55] *Welcome to CasADi's documentation!* URL: https://web.casadi.org/docs/. (accessed 28.05.2023).

[56] *IPOPT Documentation.* URL: https://coin-or.github.io/Ipopt/. (accessed 18.05.2023).

Appendix

A Resources

Resources made available.

Resource	Description of resource	Provider
FMU	Digital system model compiled to be interfacable with Python	Equinor

Other resources used.

The documentation cells contain hyperlinks to their respective documentations. Should hyperlinks not be available, the documentations are also listed in the bibliography for convenience.

Resource	Description	Documentation	Date accessed
Python 3.10.6	Open-source programming language with vast support for, among others, artificial intelligence development tools	Official documentation	04.06.2023
PyTorch	Python library for the development of neural networks	Official website	12.05.2023
Modelica	Programming language for acausal systems modelling	Official website	18.05.2023
pyfmi	Python library for interfacing with FMUs	Official website	18.05.2023
numpy	Python library for multi-dimensional computations	Official website	18.05.2023
gurobi	Commercial Python library containing effective optimization problem solvers	Documentation for Python	18.05.2023
CasADi	A library for implementing symbolic mathematics-based optimization problems[54]	Official documentation	28.05.2023
IPOPT	An open-source nonlinear interior point-based optimal problem solver available within the CasADi library	Official documentation	18.05.2023

B List of acronyms

Acronym	Meaning
MPC	Model Predictive Control / Model Predictive Controller
LSRMPC	Linear Step Response-based Model Predictive Control/ Linear Step Response-based Model Predictive Controller
NN	(artificial) Neural Network
MLP	Multilayer Perceptron
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
NNMPC	(artificial) Neural Network-based Model Predictive Control/ (artificial) Neural Network-based Model Predictive Controller
RNNMPC	Recurrent Neural Network-based Model Predictive Control/ Recurrent Neural Network-based Model Predictive Controller
MIMO	Multiple Input Multiple Output
SISO	Single Input Single Output
PRBS	Pseudo Random Binary Sequence
APRBS	Amplitude-modulated Pseudo Random Binary Sequence
GD	Gradient Descent
SGD	Stochastic Gradient Descent
NARX(-model)	Nonlinear Autoregressive model with Exogenous inputs
NNARX(-model)	Neural NARX-model
RNNARX	Recurrent Neural NARX-model
FMU	Functional Mock-up Unit

C Compiling a 64-bit FMU for Python

The following description of exporting the FMU is directly fetched from [19], as the process was the exact same during this thesis. Minor typographical alterations are made to the formulations, but not to the content.

In order to compile Modelica-code to an FMU that will interface with a 64-bit Python environment on Windows, some Python terminal with access to a library capable of compiling Modelica-code must be used. One solution, as found and utilized in relation to this project, specific to 64-bit Windows 10, is as follows:

1. Install jModelica 2.1.4
2. Access the installation folder. This will typically be located at `C:\Users\User\AppData\Roaming\JModelica.org-2.14`, but may vary between installations.
3. open the file `Python64.bat`. This will launch a 64-bit Python terminal. Within this terminal, write the following commands

```python
# Importing the necessary library function
from pymodelica import compile_fmu

# Compiling the FMU
fmu = compile_fmu('PathToModel',\
                  {'PathToModelDependency0', ..., 'PathToModelDependencyN'},\
                  target ='cs',\
                  version='2.0',\
                  compiler_log_level='error',\
                  compiler_options={"variability_propagation":False})
```