NTNU
Kunnskap for en bedre verden

Department of Cybernetics Engineering

TTK4550 - Engineering Cybernetics, Specialization Project

Towards Neural Network-based Model Predictive Control

Author:
Yngve Kippersund

December, 2022

Abstract

Model predictive control is a widely used method for process control in industry, and is often based on linear system models. The usage of a linear model, where the true dynamics may be nonlinear, indicates room for improvement in control performance if an appropriate nonlinear system model takes the linear model's place. A potential candidate for one such nonlinear system model may be implemented in the form of an artificial neural network, capable of highly nonlinear approximation, and free of first-principle derivations, requiring only relevant data for system identification. Derivations show that integrating an artificial neural network into the mathematical formulation of a model predictive controller is trivial, further indicating that neural network-based model predictive control has potential in improving control performance over traditional linear model-based model predictive control with only few alterations. By implementing both a model predictive controller based on a linear system model, and an artificial neural network approximating that same model. Both the linear model-based model predictive controller and the artificial neural network are tested in their performances to verify their function, and thus the foundation is laid for future integration of precisely the neural network-based model predictive controller.

Table of Contents

1 Introduction

1.1 Project background and its goal

Equinor operates multiple process industry locations, wherein proper process control naturally is essential. A widely used control method in process control is Model Predictive Control (MPC) [9]. A central issue in MPC is to implement a model for system dynamics that is accurate enough that the MPC performs at a desirable level. Performing control based on linear models of the system dynamics may in some cases be insufficient when the process in question deviates sufficiently from the point of linearization, or the process in reality is highly nonlinear. It follows that performance of process control may in such cases be improved by employing more precise models of system dynamics. Consequently, there is also interest for MPC based on nonlinear models for cases in which the true system dynamics are indeed nonlinear. In many cases nonlinear modelling from first principles is time-consuming and difficult. When developing models it is therefore in many cases desirable to do so in a data-driven manner, such that readily available test data can be applied, as an alternative to embarking on potentially heavy, time-consuming and resource-costly mathematical analyses. One such method of data-driven nonlinear system identification is modelling by means of artificial neural networks (ANN). Building on this, artificial Neural Network-based Model Predictive Control (NN-MPC) is of interest to the process industry, with many potential applications, depending on the system in question. The system handled in this project is a single subsea gas and oil well (from here-on referred to as the "single well"). This system is further described in Section 1.3.

The main goal of this project is to form a foundation for later implementing NN-MPC. Specifically, both a linear step response model predictive controller (LSR-MPC) as well as an artificial neural network (ANN) will be implemented. The LSR-MPC and the ANN are both implemented with an overarching goal of possible extension to the NN-MPC; the LSR-MPC presents a working MPC, which may be slightly altered to fit an ANN as a model, instead of the linear step response model. Through testing of the implemented LSR-MPC, a grounds for comparison with a future NN-MPC is also provided. The ANN presents a model for the input-output relations of the single well. The slight modifications presented, based on work done in [17], make the ANN compatible with a presented formulation of NN-MPC. Combining the implementations, results and experience from these two separate building blocks of the NN-MPC, an NN-MPC may thus be implemented in the future.

1.2 Problem description

Existing literature proposes several different ways of formulating NN-MPC - the interested reader is referred to [17][3][10]. This project is based on following the method applied in [17].

The underlying motivation of this project is integrating system dynamics modelling by means of ANN into MPC, thus implementing NN-MPC. However, due to time restrictions, this project will only implement LSR-MPC and an ANN, while not integrating them to form an NN-MPC. This task is divided into three main parts:

1. Firstly, a literature study forms the foundation for all implementational work.
2. Secondly, an LSR-MPC will be implemented, tested and results discussed.
3. Thirdly, an ANN will be implemented, tested and results discussed.

This project report consists of six sections that together form the content of the points presented above: This introduction is followed by Section 2, presenting the relevant theory prerequisite to both MPC and ANN, as well as how to combine their mathematical foundations in order to create an NN-MPC. Section 3 describes the implementations of an LSR-MPC and an ANN, as well as the procedures performed in order to evaluate their respective performances. Section 4 presents the

results obtained from testing both implementations. Section 5 discusses the obtained results, and draws attention to points that should and could be improved upon. Lastly, Section 6 concludes the project by summarizing the important points from implementation and testing of the LSR-MPC and the ANN, as well as what should be done of future work.

1.3 System description

This section presents the system in question. As mentioned in Section 1.1, the goal of this project is to approximate the system dynamics of a single well. The single well consists of a pipe connecting a reservoir of oil and gas from beneath the seabed, to a valve - the so-called *production choke* - 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. 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 production choke valve is to constrict the pipe's cross section where it is placed. In controlling its degree of constriction, the flow rate through the pipe may be controlled. Since the choke valve can be either fully open or fully constricted, its working range is between $[0, 100]$ [%]. The choke-opening is in reality a discrete variable, able to change ± 2 [%] at each step. This project makes the simplification that the choke-opening is a continuous variable. Controlling the choke-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 single well's pipe is through the second valve - the *gas lift choke*. This is done by injecting gas 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. In reality, physical constraints to the tools used in controlling the gas lift rate disallows a gas lift rate in the range $(0, 5000)$ $[\frac{m^3}{h}]$. This project considers a simplified gas lift rate, which is not subject to this constraint. The working range of the gas lift is then for the scope of this project assumed to be $[0, 10000]$ $[\frac{m^3}{h}]$. Note that this amount is from here-on assumed controlled externally with respect to the the scope of this project. Even though the gas lift is in reality controlled by the gas lift choke, gas lift will be handled as a flow rate for the rest of this project, measured in $[\frac{m^3}{h}]$. Conversely to the choke-opening, gas lift is an expensive means of control, as it involves compressing large quantities of gas, which is a power-demanding process.

The two outputs are the gas rate and the oil rate flowing from the reservoir and through the production choke, respectively.

The system's parameters may be summarized in a table:

System parameters	Value
n_{in}	2
n_{out}	2
$\bar{\boldsymbol{y}}$	$\begin{bmatrix}10^5 & 1000\end{bmatrix}^T$
$\underline{\boldsymbol{y}}$	$\begin{bmatrix}0 & 0\end{bmatrix}^T$
$\bar{\boldsymbol{u}}$	$\begin{bmatrix}100 & 10^4\end{bmatrix}^T$
$\underline{\boldsymbol{u}}$	$\begin{bmatrix}0 & 0\end{bmatrix}^T$
$\bar{\boldsymbol{z}}_{ub}$	$\begin{bmatrix}0.55 & 166.7 & 10^7\end{bmatrix}^T$
$\bar{\boldsymbol{z}}_{lb}$	$\begin{bmatrix}-0.55 & -166.7 & 0\end{bmatrix}^T$
$\boldsymbol{\rho}_h$	$\begin{bmatrix}1 & 1\end{bmatrix}^T$
$\boldsymbol{\rho}_l$	$\begin{bmatrix}1 & 1\end{bmatrix}^T$
Δt	$10[s]$
t_{final}	$7200[s]$

Table 1: The configuration of the MPC-scheme. The reference values were kept constant.

The parameters in this table define the system and its properties, the way they need be formulated in order to later implement an LSR-MPC in Section 3.2. n_{in} and n_{out} describe the amounts of inputs and outputs, respectively. $\bar{\boldsymbol{y}}$, $\underline{\boldsymbol{y}}$, $\bar{\boldsymbol{u}}$ and $\underline{\boldsymbol{u}}$ describe the upper and lower bounds for the outputs and inputs, respectively. The two first elements of both $\bar{\boldsymbol{z}}_{ub}$ and $\bar{\boldsymbol{z}}_{lb}$ describe upper and lower limits to rate of change in actuation for the single well. $\boldsymbol{\rho}_h$ and $\boldsymbol{\rho}_l$ describe that the limits on the outputs gas rate and oil rate are not absolute, and a solution may deviate from them if required in order to be feasible. The upper and lower limits on any such potential deviation from the boundaries defined by $\bar{\boldsymbol{y}}$ and $\underline{\boldsymbol{y}}$ are defined by the last elements of both $\bar{\boldsymbol{z}}_{ub}$ and $\bar{\boldsymbol{z}}_{lb}$, respectively. Finally, Δt and t_{final} describe the values for the sample time in the single well system, as well as the total simulation time, under which for example testing through simulation is interesting.

All specific values have been determined to be interesting values to use in the context of this project in dialogue with this project's industry partner, Equinor.

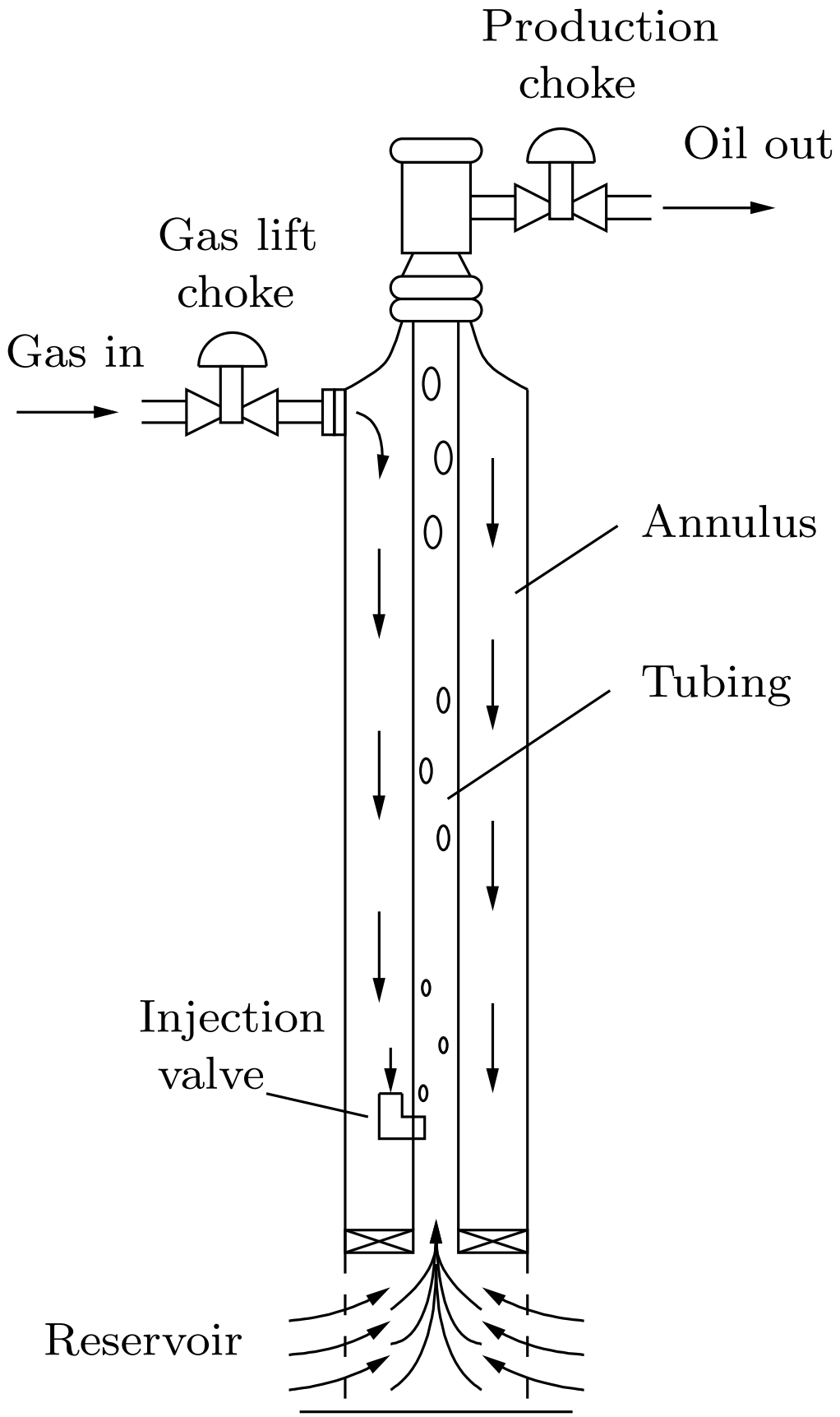


Figure 1: An illustration of the model to be discussed in this project: the single subsea oil- and gas well. Illustration is fetched from [5].

2 Theoretical background

This section will cover the theoretical background for Model Predictive Control (MPC) and Artificial Neural Networks (ANN) necessary to follow the developments and reasoning in further sections. This is done in five subsections. Section 2.1 gives a basic introduction to MPC as a method of process control, and further builds and explains the components of its basic mathematical structure. Furthermore, the linear step-response model MPC is developed, as this is intended as a "reference MPC" with which ANN-based MPC may be compared. Section 2.2 introduces the concept of ANN in its most basic form - the Multilayer Perceptron (MLP) - before presenting a different structure that is relevant for future work, the Recursive Neural Network (RNN). Section 2.3 goes through the relevant theoretical considerations made during the process of training an ANN. Section 2.4 presents the way in which the so-far presented MPC problem formulation may be modified in order to fit an ANN as the system model description. Lastly, Section 2.5 concludes by summarizing the full neural network-based MPC problem formulation.

2.1 Model Predictive Control (MPC)

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 [16]. MPC bears similarities to LQ-control, of which this project does not cover the details. The interested reader is referred to [12].

MPC is implemented by calculating the optimal control sequence for some finite prediction horizon, and applying only the first of these optimal control values in the process in question - repeating for each time step. This feedback structure makes 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 to MPC, as it allows the trajectory of optimal control values to be adjusted according to sources of inaccuracy, such as model imperfections (including dynamics, kinematics and disturbance) and noise. The Receding Horizon-principle is an important reason for the robustness qualities that MPC exhibits [8], 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 handling inequality and equality constraints as part of the optimization problem formulation. Elements of the MPC problem formulation are discussed in detail in sections 2.1.2, 2.1.3 and 2.1.4.

Even though the above-mentioned characteristics make MPC a widely used and state-of-the-art process control method, it has drawbacks. A significant drawback is the computational cost associated with solving an optimization problem at each time step, as this in many cases becomes non-trivial. For an MPC problem formulation that is convex in its cost function, and linear in its constraints, the drawback of computational cost is often negligible, due to precisely the convexity of the function to be optimized. We call this type of MPC problems *linear MPC*, examples of which are linear and quadratic programming optimization problems[14].

In the cases where the problem is inherently nonlinear, but a linear MPC problem formulation is

still desirable, one may opt to perform linearization of the system. One method of linearization is *linear step-response modelling*, used in linear step-response MPC (LSR-MPC). This method is explored further in Section 2.1.5.

Alternatively, one may use non-linear solvers for optimization problems with a non-convex cost function and/or non-linear constraints. We call such types of MPC problems *nonlinear MPC*. Non-convex optimization algorithms are more involved, and consequently computationally more demanding than the convex counterparts [14]. However, nonlinear MPC is shown to perform better than linear MPC in cases where the system dynamics are nonlinear [7]. Whether the added computational cost is a worthwhile investment or not is subject to consideration on a case-by-case basis.

2.1.2 The Cost Function

MPC determines the optimal control sequence by means of minimizing some objective function, called the *cost function*, over a prediction horizon with respect to the system's degrees of freedom [1]. In many cases, the system's degrees of freedom are the input variables or, equivalently, *changes* in input; a sequence of changes in inputs directly gives a sequence of inputs, given some initial state for the input - see (3g). One such minimization problem may look like the following:

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

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 variables $\boldsymbol{y}_{k+1:k+N}$ and $\boldsymbol{\Delta u}_{k:k+N-1}$ as well as parameters representing a given reference $\boldsymbol{y}_{ref,k+1:k+N}$. These variables usually represent measured state and change in actuation, respectively. Note that the formulation may be altered to better suit any specific MPC-scheme.

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 [16]. In (1) 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 actuation $\boldsymbol{\Delta u}_k$ is required in order to minimize this deviance - the less the better. Intuitively, the cost function's global minimum - which minimizes the *penalty* - will then provide the optimal solution.

Since the problem is one of minimization with respect to some set of variables, any values causing the function to go towards $-\infty$ would be preferred by the solver. This could cause the solver to find solutions with high absolute values in the optimization variables - typically input variables - and the variables with which they are linked - typically measured state variables. When the variables in question represent state and/or actuation values for physical systems, such minima would likely give values that are intractable for the actual physical system. Intuitively: close to infinitely high actuation as well as close-to-infinite state values are both highly undesirable in a physical system. This issue might be possible to avoid on a case-by-case basis through adequate cost function design, but is most often handled by simply designing the cost function to be positive semi-definite. When positive semi-definite, the cost is not minimized by increasing the optimization variables' absolute values towards infinity in order to obtain a minimum - the solver must instead seek the solution closest to zero.

If the cost function is *convex* [14], any local minimum will also be a global minimum [1] meaning that non-convex optimization not necessarily finds the global optimum of the cost function. It follows that designing a convex cost function can be beneficial. One way of achieving convexity and positive semi-definiteness is to use a quadratic cost function like the following [14]:

$$l(\boldsymbol{y}_{k+1+i:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref,k+1+i:k+N}) = \sum_{i=0}^{N-1} (\boldsymbol{y}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^\mathsf{T} \mathbf{Q} (\boldsymbol{y}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) + \boldsymbol{\Delta u}_{k+i}^\mathsf{T} \mathbf{R} \boldsymbol{\Delta u}_{k+i}, \tag{2}$$

where $\mathbf{Q} \succeq 0$ and $\mathbf{R} \succ 0$. Note that, while $\mathbf{Q}$ may be positive semi-definite, $\mathbf{R}$ has to be positive definite. This is in order to avoid potentially cancelling the cost function's actuation-term in (2), which would imply the actuation could assume any arbitrary value without impacting the solution. In reality, the actuation determines the system's state. Intuitively, arbitrary values in actuation are then not acceptable as a solution, as this would cause arbitrary system behaviour. This same requirement is not made for $\mathbf{Q}$, since the acceptable solution space, described in the MPC problem's constraints - see Section 2.1.3, should be formulated such that any value within this space is acceptable. The state values are implicitly determined from the determined actuation values, not the other way around, and so state values resulting from non-arbitrary actuation values do not represent arbitrary system behaviour. Furthermore, if any actuation values would lead to infeasible state values, is is taken care of by the above-mentioned acceptable solution space.

The solution to (1) will vary depending on the values of $\mathbf{Q}$ and $\mathbf{R}$. Usually, they are implemented as diagonal matrices, with each diagonal element corresponding to some scaling of the corresponding state or actuation. Due to this, they are called weighting matrices, because their diagonal elements add or remove weight from a corresponding state's or input's impact in determining the total cost calculated by the cost function. Consequently, tuning $\mathbf{Q}$ and $\mathbf{R}$ changes the behaviour of minima found during minimization, and can be used to alter the effectiveness in the MPC-scheme with respect to different aspects. Higher-valued elements of $\mathbf{Q}$ will make deviations in $\boldsymbol{y}_{k+1+i}$ from $\boldsymbol{y}_{ref,k+1+i}$ contribute more towards a higher-valued cost function, so the solver will find solutions that "prioritize" these deviations to be small-valued. Similarly, higher-valued elements of $\mathbf{R}$ will make high-valued $\boldsymbol{\Delta u}_{k+i}$, i.e. rapid changes in actuation, contribute more towards a higher-valued cost function, making the solver "prioritize" the changes in actuation to be slow.

Another important aspect of tuning an MPC-scheme is defining the size of the finite prediction horizon, as its size will define how many time steps the solver will allow in its optimal control sequence. If the prediction horizon is small, the reference values must be reached in fewer time steps; the solution will have 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 will be found. Indeed, a larger prediction horizon *does* result in a better control performance, as indicated in [1]. 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.

Thus far, the *control horizon* has been implicitly equal to the prediction horizon. However the two may be different, in which case the sequence of optimal control values is limited to the size of the control horizon, whereas the amount of steps into the future that the MPC predicts for the system model is limited to the size of the prediction horizon. An example of this is handled in Section 3.2.

2.1.3 Constraints

As mentioned in Section 2.1.1, the true strength of MPC as a method of process control comes from its ability to take into account constraints. This allows physical regards to be facilitated. Take for instance the minimization problem in (1) and the cost function in (2). By subjecting this minimization problem to constraints, feasible regions of values are encoded into the problem formulation:

$$\min_{\boldsymbol{\Delta u}_{k:k+N-1}} \sum_{i=0}^{N-1} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^\mathsf{T} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) + \boldsymbol{\Delta u}_{k+i}^\mathsf{T} \mathbf{R} \boldsymbol{\Delta u}_{k+i} \tag{3a}$$

$$s.t. \forall i \in [0, N-1]:$$

$$\hat{\boldsymbol{x}}_{k+1+i} = \boldsymbol{f}(\hat{\boldsymbol{x}}_{k+i}, \boldsymbol{u}_{k+i}) \tag{3b}$$
$$\hat{\boldsymbol{y}}_{k+1+i} = \hat{\boldsymbol{x}}_{k+1+i} \tag{3c}$$
$$\boldsymbol{y}_{lb} \leq \hat{\boldsymbol{y}}_{k+1+i} \leq \boldsymbol{y}_{ub} \tag{3d}$$
$$\boldsymbol{u}_{lb} \leq \boldsymbol{u}_{k+i} \leq \boldsymbol{u}_{ub} \tag{3e}$$
$$\boldsymbol{\Delta u}_{lb} \leq \boldsymbol{\Delta u}_{k+i} \leq \boldsymbol{\Delta u}_{hb} \tag{3f}$$
$$\boldsymbol{u}_{k+i} = \boldsymbol{u}_{k-1+i} + \boldsymbol{\Delta u}_{k+i} \tag{3g}$$

Given (3b) and (3c), the minimization problem (3a) is forced to comply with the system dynamics, here assumed to be described by some state space formulation. Furthermore, (3d) 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 (3e), (3f) and (3g), respectively.

Note that by adhering to the requirement in (3b) and (3c), the optimizer is performing open loop predictions of future state. Consistency with current state at step k is ensured by the constraint in (3b) when $i = 0$. As is convention, this project will denote any prediction $\hat{(\cdot)}$, exemplified with $\hat{\boldsymbol{y}}_{k+1+i}$.

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* [1]. In the case of (3), 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 p *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:

$$\min_{\boldsymbol{\Delta u}_{k:k+N-1}} \sum_{i=0}^{N-1} [(\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^\mathsf{T} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) + \boldsymbol{\Delta u}_{k+i}^\mathsf{T} \mathbf{R} \boldsymbol{\Delta u}_{k+i}] + \boldsymbol{\rho}^\mathsf{T} \epsilon_{\boldsymbol{y}} \tag{4a}$$

$$s.t. \forall i \in [0, N-1]:$$

$$\hat{\boldsymbol{x}}_{k+1+i} = \boldsymbol{f}(\hat{\boldsymbol{x}}_{k+i}, \boldsymbol{u}_{k+i}) \tag{4b}$$
$$\hat{\boldsymbol{y}}_{k+1+i} = \hat{\boldsymbol{x}}_{k+1+i} \tag{4c}$$
$$\boldsymbol{y}_{lb} - \epsilon_{\boldsymbol{y}} \leq \hat{\boldsymbol{y}}_{k+1+i} \leq \boldsymbol{y}_{ub} + \epsilon_{\boldsymbol{y}} \tag{4d}$$
$$\boldsymbol{u}_{lb} \leq \boldsymbol{u}_{k+i} \leq \boldsymbol{u}_{ub} \tag{4e}$$
$$\boldsymbol{\Delta u}_{lb} \leq \boldsymbol{\Delta u}_{k+i} \leq \boldsymbol{\Delta u}_{hb} \tag{4f}$$
$$\boldsymbol{u}_{k+i} = \boldsymbol{u}_{k-1+i} + \boldsymbol{\Delta u}_{k+i} \tag{4g}$$

where the dimensionalities of $\boldsymbol{\epsilon_y}$ and $\boldsymbol{\rho}$ match that of $\mathbf{y}$. Subject to linear constraints, (4) is still convex [1]. 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 contribution the slack-term can give. This is true 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, and would defeat the purpose of the constraints. In the example of (4), the constraints are only made soft for $\boldsymbol{y}_{k+1+i}$ in (4d), since allowing slack in actuations would mean potentially allowing violation of physical constraints, such as saturations.

2.1.4 Choosing a solver

The solver is the algorithm that finds the solution to the MPC problem formulation as it is presented in sections 2.1.2 and 2.1.3. Delving into the theoretical foundations of available solvers is beyond the scope of this project, but considerations to be made in the solver choices relevant to this project will be briefly covered. This project handles two cases where the MPC problem formulations are linear and highly nonlinear, respectively. Consequently two different solvers must be chosen. Brief descriptions and explanations of their properties are presented.

The first MPC problem formulation is introduced in Section 2.1.5. Its minimization problem has a quadratic cost function, and is entirely linear in its constraints. Thus the MPC problem formulation as a whole is convex, and a convex solver may be used.

The second MPC problem formulation is introduced in Section 2.5. Its minimization problem is highly nonlinear in the model's system dynamics. Regardless of the form of the rest of the constraints and the cost function, this is enough to make the problem, as a whole, highly non-convex. A non-convex solver must then be used.

2.1.5 Linear Step Response MPC

The distinguishing factor between LSR-MPC and the generic MPC derived in (4) 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 that changes in an input have on a corresponding output in a linear fashion [16]. The first of these coefficients represents the effect a change in input has from the moment it occurs to the next timestep. 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 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 [16] and [12].

The series of coefficients is the full step response model for the SISO relationship between an input and an output [16], and is denoted S, $S \in \mathbb{R}^N$, where N is the amount 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 [16] (equation (20-10)) for a SISO system:

$$\hat{y}_{k+j} = \sum_{i=1}^{j} [S_i \Delta u_{k+j-i}] + \sum_{i=j+1}^{N-1} [S_i \Delta u_{k+j-i}] + S_N u_{k+j-N}, \tag{5}$$

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 (5) represent the contribution of future changes in input and the contribution of past changes in input, respectively. The last term represents the lasting contribution to the output after it has settled from some past input.

Altering (4) to instead comply with the model of system dynamics as described in (5), we arrive at a new MPC problem formulation:

$$\min_{\Delta u_{k:k+N-1}} \sum_{j=0}^{N-1} [(\hat{y}_{k+1+j} - y_{ref,k+1+j})^2 Q + \Delta u_{k+j}^2 R] + \rho \epsilon_y \tag{6a}$$

$$s.t. \forall j \in [0, N-1] :$$

$$\hat{y}_k = y_k \tag{6b}$$

$$\hat{y}_{k+1+j} = \sum_{i=1}^{1+j} [S_i \Delta u_{k+1+j-i}] + \sum_{i=j+2}^{N-1} [S_i \Delta u_{k+1+j-i}] + S_N u_{k+j-N} \tag{6c}$$

$$y_{lb} - \epsilon_y \leq \hat{y}_{k+1+j} \leq y_{lb} + \epsilon_y \tag{6d}$$

$$u_{lb} \leq u_{k+j} \leq u_{ub} \tag{6e}$$

$$\Delta u_{lb} \leq \Delta u_{k+j} \leq \Delta u_{ub} \tag{6f}$$

$$u_{k+j} = u_{k-1+j} + \Delta u_{k+j} \tag{6g}$$

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 $\mathbf{S} \in \mathbb{R}^{n_{out}N \times n_{in}}$:

$$\boldsymbol{S} = \begin{bmatrix} S_{1,1} & S_{1,2} & \cdots & S_{1,n_{in}} \\ S_{2,1} & S_{2,2} & \cdots & S_{2,n_{in}} \\ \vdots & \vdots & \ddots & \vdots \\ S_{n_{out},1} & S_{n_{out},2} & \cdots & S_{n_{out},n_{in}} \end{bmatrix}, \tag{7}$$

where each row represents the SISO relationships between all inpus 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 Section 3.2.1.

2.2 Artificial Neural Network (ANN)

ANN is the name of a class of machine learning models that can be trained in order to learn certain behaviours. Its structure is inspired by the brain's - hence the name - mimicking signals firing between interconnected neurons in order to produce some output [2]. Exactly what is meant by an ANN learning certain behaviours is further explained in Section 2.2.1.

2.2.1 Machine learning and deep learning

A high-level explanation of Machine learning is presented in [13]:

> 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.

An example is the *k nearest neighbours*-algorithm. This algorithm can take in a set of data-points and their associated classes, and determine an unclassified data-point based on to which classes its k nearest neighbours belong. The class which is most represented among these k nearest neighbours is then determined to be the class to which the unclassified data-point should belong. The *learning* can in this case be seen as the algorithm better learning how to classify new data-points if given more data; the more densely populated the dataset on which it bases its decisions, the better the algorithm will "understand" how to classify. Thus the algorithm *learns* with an increasing dataset [6]. Note that the learning happens statically during classification; the algorithm has no inherent understanding of the behaviour of the data-points it classifies. This type of learning is called *lazy learning* [6].

The complement to lazy learning is *eager learning* [6], a type of machine learning in which the algorithm uses given datasets to *train* some model of the behaviour of the dataset's data-points. Deep learning represents a vast class of eager machine learning algorithms, all of which are implemented as ANNs. Deep learning can be employed in a vast array of tasks, such as classification, regression, language translation and anomaly detection, to name a few [2]. The high-level goal of this project is to be able to predict some priorly unknown outputs given some inputs, $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}_{in})$, for a system by means of training an ANN, such that the ANN implements the input-output relation $\hat{\boldsymbol{y}} = \hat{\boldsymbol{f}}(\boldsymbol{x}_{in})$. Our ideal result is thus to achieve:

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}_{in}) \approx \hat{\boldsymbol{f}}(\boldsymbol{x}_{in}) = \hat{\boldsymbol{y}}, \tag{8}$$

and consequently our task is one of *regression*, as it is presented in [2] (quote is only partial):

> [...] 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 $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. [...]

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 output 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 $\boldsymbol{x}_{in} \in \mathbb{R}^n$.

Specifically how the regression problem is tackled in this project is further explored in sections 2.4 and 3.3. Firstly, specifics of the basic functionalities of an ANN are presented in the following section.

2.2.2 The feed forward structure

The ANN 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 (8) - and the output layer delivers the final function value - $\hat{\boldsymbol{f}}(\boldsymbol{x}_{in})$ in (8). In the case of regression, the ANN's goal is to drive this output to match the ground truth - $\boldsymbol{f}(\mathbf{x}_{in})$ in (8) - as closely as possible. The hidden layers are implemented as intermediate steps between the input and the output, in order to facilitate the approximation. Together with the output layer, the hidden layers implement a set of parameters that shape the function $\hat{\boldsymbol{f}}(\mathbf{x}_{in})$ 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 ANN here described is called a *Multilayer Perceptron* (MLP), an example of which is illustrated in Figure 2.

Each neuron represents a function applied on 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 sequential a convolution of every layer. In the case of the MLP illustrated in Figure 2:

$$\boldsymbol{f} = \boldsymbol{f}_3(\boldsymbol{f}_2(\boldsymbol{f}_1(\boldsymbol{x}_{in}))), \tag{9}$$

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. 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

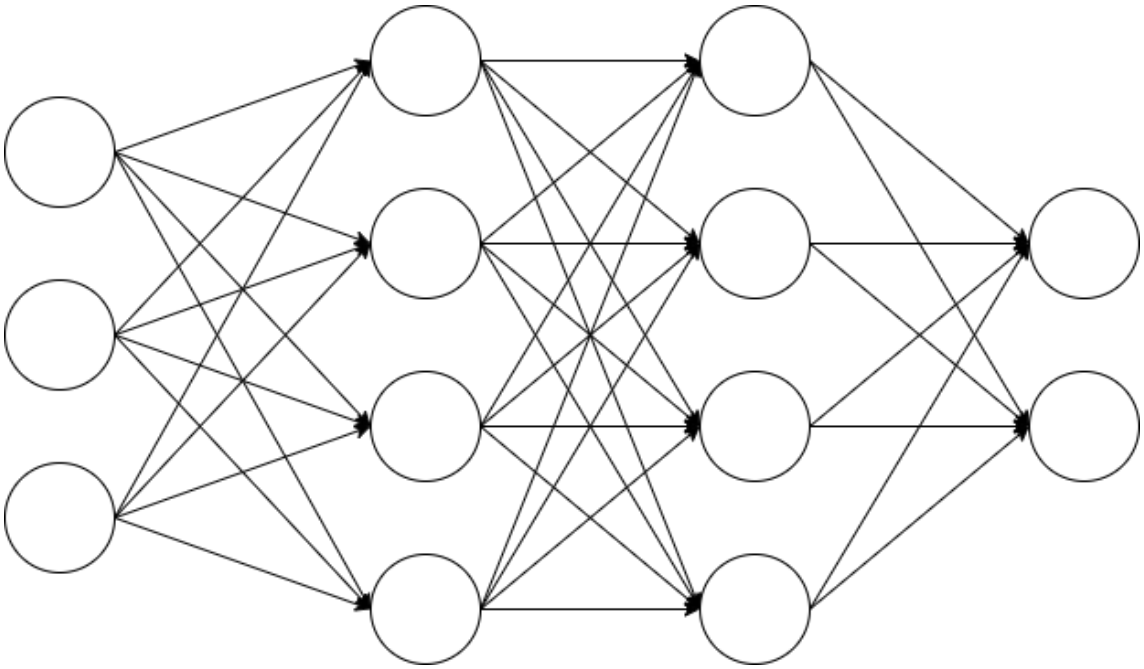


Figure 2: An illustration of a simple MLP. The input layer has 3 neurons, the two hidden layers have 4 neurons each, and the output layer has 2 neurons.

this project. 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:

$$\boldsymbol{f}_l(\boldsymbol{x}) = \mathbf{W}_l \boldsymbol{x}, \tag{10}$$

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:

$$\boldsymbol{f}_l(\boldsymbol{x}) = \mathbf{W}_l \boldsymbol{x} + \boldsymbol{b}_l, \tag{11}$$

where of course $\boldsymbol{b}_l \in \mathbb{R}^{\eta_l}$. Note that this function approximation is only linear, and is not able to approximate nonlinear dynamics. The third and last component, the *activation function*, remedies this by passing the calculation at each node through a nonlinear function. An important such function, the *rectified linear unit* (ReLU)[2], is defined as:

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

applied element-wise if the input is vector-valued. There exist many different activation functions, and each is relevant to their specific use cases, but this project will only handle the ReLU for simplicity, as this is often the default choice [2].

The linear combination (11) is passed through the activation function at each neuron, meaning the full vector-valued formulation of a layer becomes:

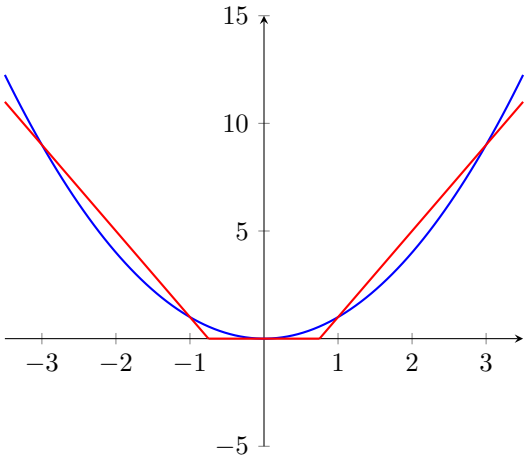


Figure 3: A second order polynomial and three linear functions approximating it.

$$\boldsymbol{f}_l(\boldsymbol{x}) = ReLU(\mathbf{W}_l\boldsymbol{x} + \boldsymbol{b}_l) \tag{12}$$

Where the sum of all the neurons' linear functions previously resulted in a linear function (11), simply performing linear regression, now having introduced nonlinearity at every neuron makes the resulting function (12) 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 a nonlinear function. An example is provided in Figure 3. It has been shown that the MLP can learn its parameters such that an arbitrary nonlinear function may be approximated [4], given a sufficient amount of hidden neurons. Specifically how the learning of parameters takes place is further explored in Section 2.3.

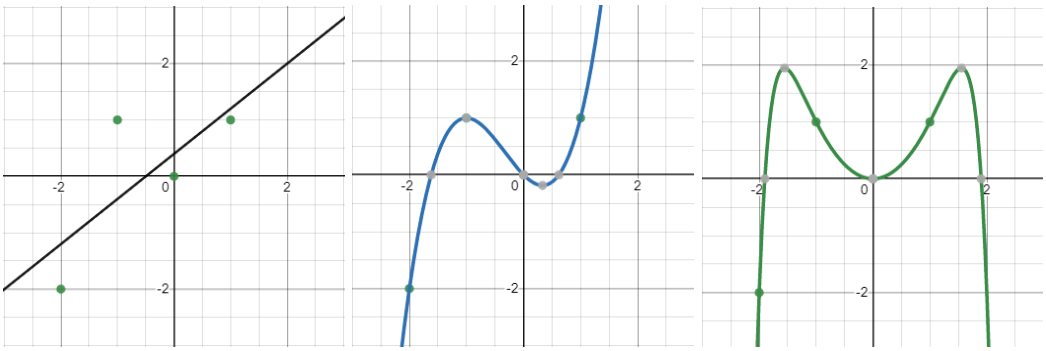
2.2.3 Recurrent Neural Network (RNN)

The Recurrent neural network (RNN) is a variant of the feed forward architecture presented in Section 2.2.2, where several instances of the MLP are chained, such that the output of one becomes the input of the next. Importantly, these instances share parameters (weights and biases), granting them equal function approximation qualities. The RNN can then be viewed as a single network, where its own output is fed back into its own input. An example is illustrated in Figure 7. The RNN's parameter sharing is important, because it enables the RNN to display some sense of "memory", making it more suited for modelling sequences of data than the standard MLP [2] (chapter 10). In Section 2.4, we shall see that this architectural property makes the RNN a natural candidate for system dynamics approximation in the MPC problem.

2.3 Training of an Artificial Neural Network

This section is based on material found in [2], and will present theory relevant to training ANNs for the purpose of regression.

Training of an ANN can on a high level be described as the process of defining its hyperparameters and parameters, such that it approximates some function with higher accuracy than before training. This is further covered in Section 2.3.2. First, the next section covers some basics underlying the top-level goal of generalization.



(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 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 created using Desmos Online Graphing Calculator.

2.3.1 Capacity of an ANN

Function dynamics seen in the relations between data-points in datasets may in general terms be described by some function. An ANN's ability to learn a broad spectrum of such functions is called its *capacity*. Informally, one can say that an ANN with a low capacity will struggle to approximate very nonlinear functions. An example could be trying to approximate a third-order polynomial with a linear function - see Figure 4a. In such cases, we say that the ANN is *underfitted*. In the opposite case, an ANN with a high capacity has the ability to approximate highly nonlinear functions, but will as a consequence tend to approximate any function as a highly nonlinear function. When attempting to approximate only somewhat nonlinear functions, the highly nonlinear approximation may yield great results for a set of specific data-points, but will in reality predict the wrong behaviour for data-points not part of that specific set. In such cases, we say that the ANN is *overfitted*. An example could be trying to approximate a third-order polynomial as a tenth-order polynomial - see Figure 4c. Both the underfitted and overfitted case deviate substantially from the ground truth in Figure 4b. Thus, the ideal goal is to achieve the optimal capacity; neither too high, nor too low.

As mentioned in Section 2.2.2, a simple MLP is able to approximate any arbitrary function, given enough hidden neurons, meaning that an ANN's *potential* capacity is determined by its structure. The structure is determined by the so-called *hyperparameters*, for example the amount of hidden layers and their sizes.

How much of the ANN's potential capacity is obtained depends on how the training is executed.

2.3.2 Training, validation and testing

The metric used to measure the predictive accuracy of the ANN is the training error E_{train}, roughly describing how correct the ANN is in its predictions - the outputs in the output layer. Qualitatively, if E_{train} is low, then the ANN performs well on the dataset on which it is trained.

The training error E_{train} can be defined in many ways. For problems of regression, it is often defined as the mean square error between the predicted output and the true output:

$$E_{\text{train}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=0}^{m}(\hat{\boldsymbol{y}}_i(\boldsymbol{\theta}) - \boldsymbol{y}_i)^2 \tag{13}$$

It follows from (13) that lowering E_{train} as much as possible indicates that the ANN is performing as well as possible. Since every output depends on the same parameters that define the ANN -

the weights and biases - minimizing E_{train} with respect to these parameters will yield the parameters that makes the ANN predict the outputs as precisely as possible. To this end, *stochastic gradient descent* (SGD) is used, with E_{train} as the cost function. Note also that the parameters have to be initialized, however specific the details regarding parameter initialization and relevant considerations thereof are not discussed in this project.

SGD is only covered in some of its superficial qualities. The interested reader is referred to [2] for further reading. SGD is a method of optimization used for ANNs, employing the gradient of the cost function with respect to the parameters to seek out a minimum of the cost function. The minimum is found iteratively; the minimization performs steps in the direction of the negative gradient of the cost function. The negative gradient is found by means of an algorithm called *backpropagation*, deriving the resulting output's gradients with respect to the ANN's parameters over some amount of the data-points within the training dataset. How large steps are made is decided by a parameter called the *learning rate*. After a minimum is found, the parameters of the ANN are updated accordingly, and the process may be repeated, yielding parameters that perform better with respect to the cost function for each iteration. One such iteration is called an *epoch*.

While considering the full dataset simultaneously for each epoch would yield good results with regards to optimization, it could also become extremely time-consuming. As such, dividing the training dataset into so-called *batches* is a much-used practice that allows SGD to perform minimization based on a smaller subset of the training dataset. This is where the stochasticity in SGD stems from - samples are drawn randomly from the full training dataset, with a uniformly distributed probability of any single sample being drawn. Even though this reduces the efficiency of the minimization per epoch, much computational time is saved by utilizing only subsets of the full training dataset. The amount of samples drawn per epoch is called the *batch size*. It follows that batch size is a compromise between performance in the optimization at each optimization step and computation-time. As the amount of epochs increases, E_{train} will tend to decrease.

The true success of training an ANN, however, can be described as its generalization error E_{gen}, being how correct the ANN is in its predictions when tested on a dataset of previously unseen data. In reality, E_{gen} is not directly available, as the error over every theoretically possible data-point may not be measured, and thus we approximate it with another metric, E_{test}, calculated as the mean square error over the test dataset. Note that this is an *approximation* of E_{gen}, precisely because the test dataset is not exhaustive of all possible data-points. However, given that both the training datasets and the test datasets are i.i.d. and separately collected, it is a viable assumption [2].

Given prolonged periods of training, the parameters will converge to an optimum - local or global - and the training error E_{train} will decrease steadily. As previously mentioned, the hyperparameters define what complexity is achievable for an ANN. With hyperparameters that allow high degrees of complexity, the fact that the parameters eventually converge indicates that the ANN becomes perfectly tailored to predict the data-points as they behave in the specific dataset on which the ANN is trained. This would resemble the case in Figure 4c, where the data-points available are perfectly predicted by the ANN. However, previously unseen data-points would be drawn from the true data-generating process, in the example case being Figure 4b, all for which the model in Figure 4c would predict erroneously. Note that, if in theory the training dataset had been completely exhaustive of all theoretically possible data-points, perfectly tailoring the ANN to this dataset would mean perfectly tailoring the ANN to all possible data-points from which to predict an output. This is unrealistic, but still indicates that a larger, more general training dataset will in general yield increased generalization.

Even though the model decreases both E_{train} and E_{test} for some time during the beginning of training, this is illustrative of how a decrease in E_{train} not necessarily implies a decrease in E_{test} as time further increases indefinitely - instead the ANN at some point becomes overfitted. A decrease in E_{test} is the top-level goal, as that is the indicator of actually achieving generalization. It follows that training an ANN for indefinite times is not desirable. The issue of overfitting due to training a model for too long is illustrated in Figure 5.

The issue of overfitting may be somewhat mitigated if methods for reducing E_{test} are employed. This is briefly looked into in Section 2.3.4.

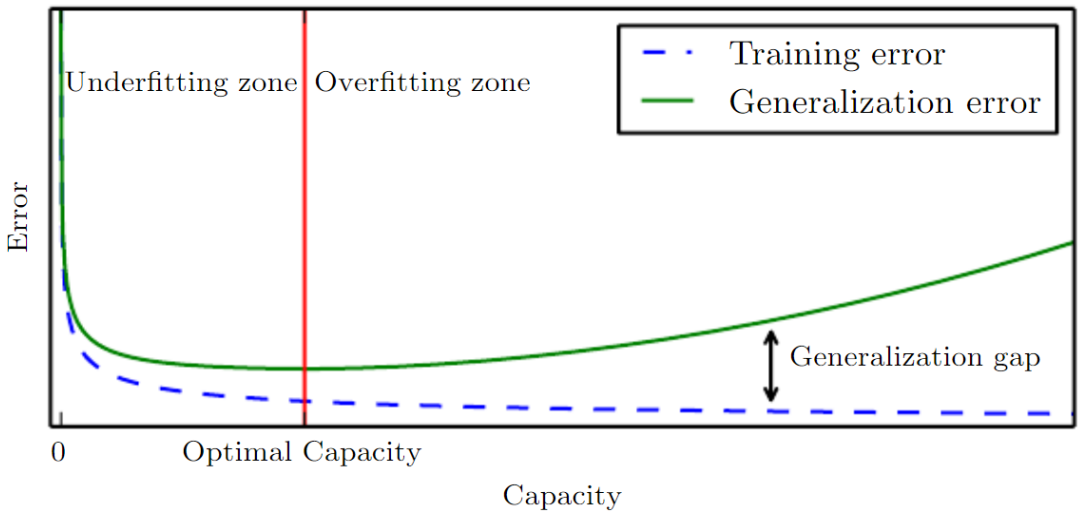


Figure 5: An illustration of how generalization error and training error develops over time. Illustration is fetched from [2], page 113.

The question of determining the hyperparameters remains. Finding the optimal hyperparameters is done by training an ANN iteratively, using a candidate set of hyperparameters from a set of candidate sets of hyperparameters for each iteration of training. By evaluating the predictive accuracy after each such iteration, the validation error, denoted as E_{val}, it is possible to compare the performance each set of hyperparameters gives rise to. The validation error is found in the same way as the test error. When all candidates are evaluated, the best set of hyperparameters is deemed to be the one which minimizes E_{val}. Note that the training dataset used in this iterative scheme is usually quite small, as a means of reducing the computational time of finding the best set of hyperparameters. If this is the case, it will be necessary with a final step of training with the full training dataset, using the best set of hyperparameters, after which the ANN may be tested, in order to evalute E_{test}. The case is most often:

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

The final flow of developing a fully trained ANN may be summarized by Figure 6.

2.3.3 Requirements on the datasets

An ANN is trained using datasets with information regarding the behaviour it is supposed to learn to approximate. The data-points within such a dataset represent some pattern, for example a process' system dynamics. There are a few requirements that must be met by the datasets used, in order to achieve better results.

Firstly, in order to teach the ANN to approximate a specific process, the data-points should all represent the same specific process; all data-points should be independent and identically distribued (i.i.d.), stemming from the same data-generating process. This has to apply for datasets used in both training and testing. The data-generating process may for example be measurements from a system, or values from a simulation of one.

Secondly, the different datasets for training and testing should be completely disjoint. This is important, because the function of the test dataset is to test the ANN on previously *unseen* data. An ANN shapes its parameters during training based on the data it is given. However, the ANN may never know *for certain* the behaviour of previously *unseen* data, and testing for the gap between predictive accuracies on previously seen and previously unseen data is what gives a measure of the ANN's ability to generalize after training. If the ANN has previously seen the test data due to overlap between training and test data, then the gaps is not in reality what is being

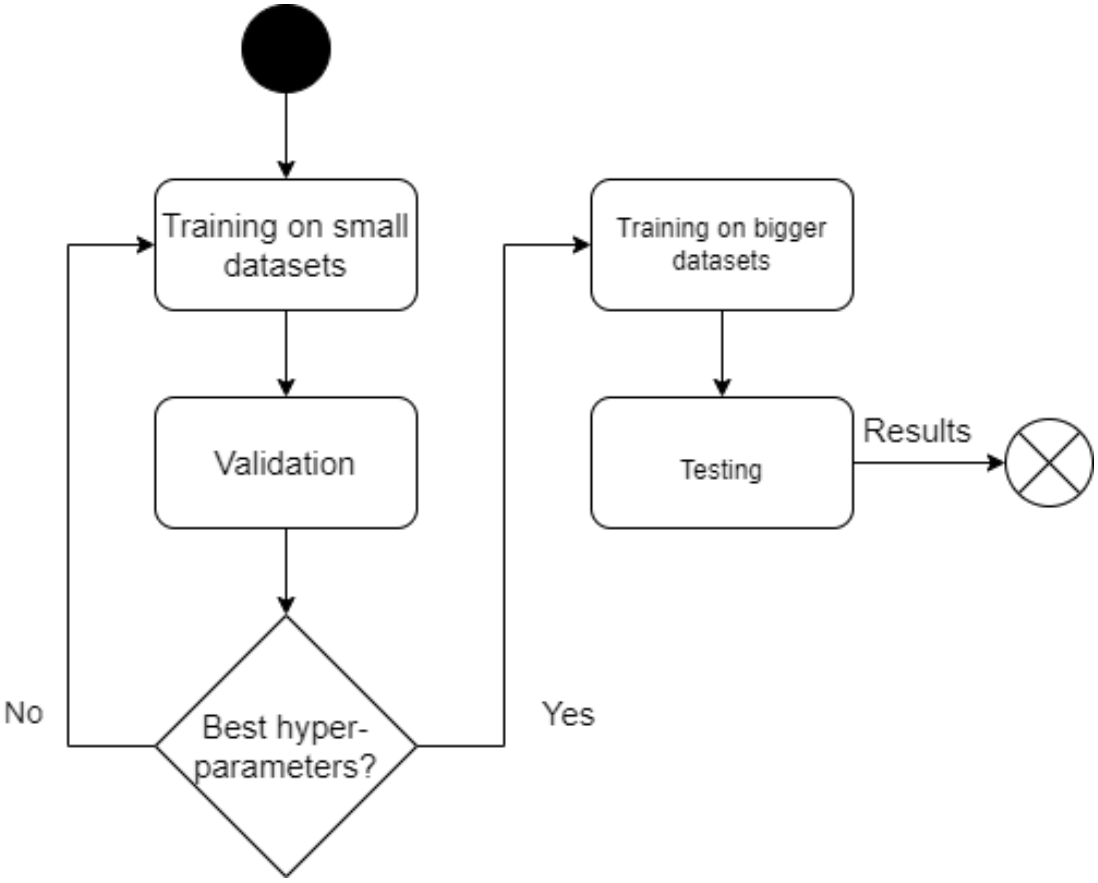


Figure 6: The procedure of training an ANN.

tested for. Instead, one is simply testing whether the ANN "remembers" the training data. If this is the case, the ANN is overfitted; the ANN has become tailored to the specific dataset on which it has trained, and is not able to generalize.

The argument that testing the ANN should be done with previously unseen data must be adhered to even when only a small dataset is available for all of training, validation and testing. When the available data is limited, splitting it into strictly disjoint portions for training, validation and testing is useful. The portions' ratios may be subject to tuning. Much used values are 70% for training, 15% for validation and 15% for testing - see for example [17]. Intuitively, it makes sense that training requires more data, as it is the process that shapes the ANN, while validation and testing only serve verification purposes. Considerations around splitting available data into ratios need not be made when unlimited data is available, except for considerations of computational time.

Lastly, the datasets may require *normalization*. If different outputs of the ANN are of inherently different magnitudes - an example for which is discussed in Section 3.3 - then E_{train} may become dominated by errors in one output, even though errors in other outputs may be as large or larger, simply due to different magnitudes of relevant ranges for the respective values. When normalizing all data, any error instead becomes relative.

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 [2]. As there exist many methods of regularization, this project will focus only on the one utilized in Section 3.3.4: *early stopping*.

Early stopping is meant to specifically remedy the issue with the increasing gap between E_{train} and E_{test} for increasing amounts of epochs trained. This is done, as the name suggests, simply by terminating the training, even if the designated amount of epochs have been reached. Specifically, early stopping terminates the training at the point which minimizes the difference between E_{train} and E_{test} - the point of optimal capacity. Early stopping may also be used during validation, following the exact same logic, simply replacing E_{test} with E_{val}.

In practice, E_{train} and E_{test} are 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 11, and so the difference $E_{\text{train}} - E_{\text{test}}$ may have several local minima. 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".

Early stopping may be done with respect to either $E_{\text{test}} - E_{\text{train}}$ or simply E_{train}. Since E_{train} and E_{test} may develop at different rates, the difference $E_{\text{test}} - E_{\text{train}}$ may then increase, even though both E_{train} and E_{test} decrease. However, since E_{train} is expected to strictly decrease for increasing epochs, implementing early stopping with respect to only E_{test} will ensure that training ceases around the minimum of E_{test}. Since, the minimum in E_{test} is not guaranteed to correlate with the minimum in $E_{\text{train}} - E_{\text{test}}$, stopping early only with respect to E_{test} does not guarantee optimal capacity.

During testing for the optimal set of hyperparameters, E_{val} simply replaces E_{test} in the early stopping criterion.

2.4 Recurrent Neural Network-based Model Predictive Control (RNN-MPC)

Even though this project does not present an implementation of the Recurrent Neural Network-based Model Predictive Control (RNN-MPC), this section will still present theoretical background

required for the RNN-MPC as a basis for future work. Importantly, this section will propose that an RNN is a natural extension to the regular MLP upon integration into an MPC-scheme.

The following derivations are heavily based on [17].

2.4.1 State formulation

MPC requires open loop prediction of future state values, and it is natural to assume that future values of the outputs will be affected by previous values of the outputs as well as previous values of the inputs. Additionally, accounting for potential noise, a *general* formulation for any future value of the outputs can be formulated as follows:

$$\boldsymbol{y}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k) + \boldsymbol{w}_k, \tag{14}$$

where $\boldsymbol{y}_k \in \mathbb{R}^m$, and $\boldsymbol{x}_k \in \mathbb{R}^n$ holds all the previous outputs and inputs necessary for prediction. More specifically, a *nonlinear autoregressive model with exogenous inputs* (NARX model) is proposed to match this specific case, since utilizing the NARX model implies that system identification can be performed by considering the *history* of previous outputs and inputs. We formulate the length of said history by means of m_y and m_u, and $\boldsymbol{x}_k$ becomes:

$$\begin{aligned}
\boldsymbol{x}_k \triangleq [&y_{0,k}, y_{0,k-1}, \cdots, y_{0,k-m_y}, \\
&\vdots \qquad\qquad\qquad , \\
&y_{n_{out},k}, y_{n_{out},k-1}, \cdots, y_{n_{out},k-m_y}, \\
&u_{0,k-1}, u_{0,k-2}, \cdots, u_{0,k-m_u}, \\
&\vdots \qquad\qquad\qquad , \\
&u_{n_{in},k}, u_{n_{in},k-1}, \cdots, u_{n_{in},k-m_u}],
\end{aligned} \tag{15}$$

where n_{out} is the amount of outputs and n_{in} the amount of inputs. While m_y and m_u could be chosen individually for each specific output and input, this project will not.

As is discussed in Section 3.3, this project handles a digital system model by means of a Functional Mock-up Unit (FMU), in which noise is not modelled. As such, we omit the noise term $\boldsymbol{w}_k$ also from (14) and consider the model

$$\boldsymbol{y}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{16}$$

as a basis in every further derivation.

Interestingly, $m_y = 0$ and $m_u = 0$ would indicate that the following holds:

$$\boldsymbol{y}_{k+1} = \boldsymbol{f}([y_{1,k}, y_{2,k}], \boldsymbol{u}_k),$$

meaning that the Markov assumption holds, and that the system can be modelled as a Markov chain - completely independent of previous history, except for the current value of the output and the input to the system between each timestep. Whether this assumption holds or not depends on the specific system in question. Sections 4.2 and 5.2 further discuss whether the relevance of the Markov assumption holds in developing the ANN for model approximation in this project.

2.4.2 Predicting future state by means of artificial neural networks

Building on the material presented in sections 2.2.2 and 2.4.1, this project makes the naïve assumption that an MLP will suffice in system model approximation:

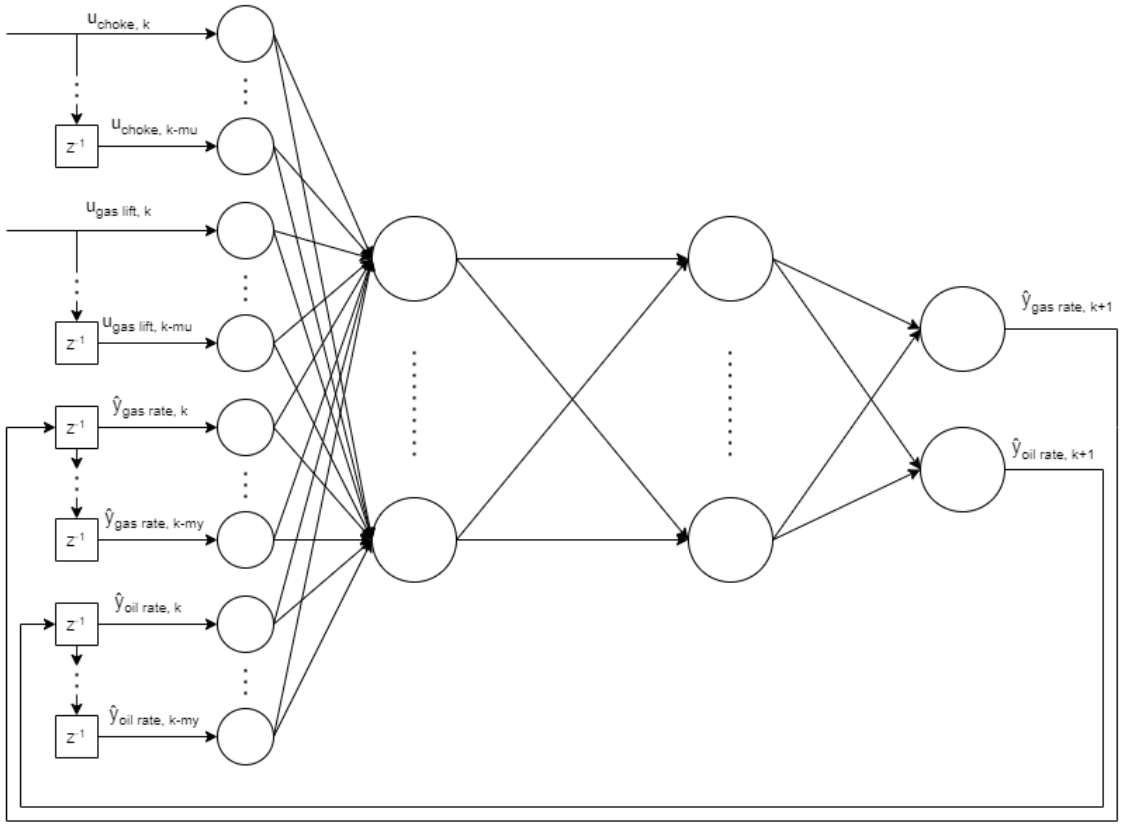


Figure 7: An illustration of the RNN resulting from integrating an MLP as the system model into the MPC formulation, as is done in (20). Illustration is based - with permission - on an illustration from [17], page 3.

$$\hat{\boldsymbol{y}}_{k+1} = \hat{\boldsymbol{f}}_{MLP}(\mathbf{x}_k, \mathbf{u}_k) \tag{17}$$

The only requirement the MLP must meet, is that of input and output dimensionality. Specifically, the input layer must match the dimensionality of the state vector in (15), in addition to the input at time k, and the output layer must match the dimensionality of the number of outputs in the system:

$$\hat{\boldsymbol{f}} : \mathbb{R}^{n_{out} \cdot (m_y+1) \times n_{in} \cdot (m_u+1)} \longrightarrow \mathbb{R}^{n_{out}},$$

Note that the MLP that will be integrated into the MPC problem formulation necessarily has to predict future values in an open loop-manner. More specifically, within a prediction horizon H_p we have that $\hat{\mathbf{y}}_{k+i}\ \forall\ i = 1, 2, ..., H_p$ must be predicted, of which the last $\hat{\mathbf{y}}_{k+i}\ \forall\ i = 2, 3, ..., H_p$ depend on future values of the outputs that necessarily have not yet occured in the physical system. Thus the network must rely on its own predictions, implying that predictions are fed back to the input layer uncorrected, resulting in a structure as illustrated in Figure 7.

Upon feeding back from the output layer to the input layer, the network has now taken the form of an RNN; basing our choice of ANN architecture on the basic MLP, the emergence of the RNN architecture is an immediate consequence of the open-loop prediction of sequential data, rather than a design choice. Intuitively, this is unsurprising, given its memory-like qualities, as mentioned in Section 2.2.3. Note that this open-loop prediction is conceptually identical to the open-loop prediction taking place during optimization of the MPC problem in other MPC problem formulations. As such, the RNN may directly take the place of the system model in the relevant MPC problem formulation's constraints.

Interestingly, the need for propagating predictions falls away during training of the network, as full datasets should in this case be available, and no input needs ever be a previous prediction. Thus we have the possibility of treating each prediction in a standalone-manner, reducing the architecture back again to the standard MLP we initially chose as our architecture. This will likely improve performance during training [17], as using real data - the ground truth - instead of predicted data necessarily provides a cost function E_{train} that is more accurate with respect to the true input-output relations, giving more accurate grounds from which to predict some value. Furthermore training the network as an RNN would complicate optimization of parameters during training, as pure backpropagation by means of stochastic gradient descent will no longer apply [17]. Additionally, since RNN implements a means of parameter sharing, which improves sequential data modelling performance [2], training these parameters to give the network an as accurate approximation at each single timestep as possible must be assumed to improve performance in predictions of sequential data as well. Thus, the network should be trained as a regular MLP, and the feedback connection should be made only during application, where future data is *not* available.

2.5 Formulation of the full RNN-MPC optimization problem

This section presents a full RNN-MPC formulation, even though it is not implemented. This is done in order to lay the grounds for future work.

The RNN-MPC formulation here presented is heavily inspired by work done in [17], but is tailored to this project's specific case, the single well model.

2.5.1 The cost function

For this project's relevant control problem, controlling the outputs to some given reference value, it is also desired to do this in a manner that is economic with respect to wear and tear on actuators. Consequently, the cost function should contain a term penalizing deviations in the outputs from their given reference, as well as a term penalizing too-high values in change in actuation. The cost function $l(\hat{\boldsymbol{y}}_{k+1:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref})$ can then be formulated in the standard quadratic fashion:

$$l(\hat{\boldsymbol{y}}_{k+1:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref}) = \sum_{i=0}^{N-1} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref})^{\mathsf{T}} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref}) + \boldsymbol{\Delta u}_{k+i}^{\mathsf{T}} \mathbf{R} \boldsymbol{\Delta u}_{k+i}, \quad (18)$$

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.

Should the reference be time-varying, a slight change in formulation is required:

$$\begin{aligned} l(\hat{\boldsymbol{y}}_{k+1:k+N}, \boldsymbol{\Delta u}_{k:k+N-1}; \boldsymbol{y}_{ref,k+1:k+N}) = \sum_{i=0}^{N-1} & (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i})^{\mathsf{T}} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref,k+1+i}) \\ & + \boldsymbol{\Delta u}_{k+i}^{\mathsf{T}} \mathbf{R} \boldsymbol{\Delta u}_{k+i} \end{aligned} \quad (19)$$

2.5.2 Constraints

Constraints implemented in the MPC optimization problem formulation must ensure that optimization is always in accordance with the system model, as approximated by the model (17). 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. Furthermore, the optimization is not free to alter the output predictions directly, but must adhere to the system model. Consequently, we must have:

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

where the contents of (15) have been directly inserted. 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. 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:

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

Lastly, as we seek to control with respect to change in actuations, the actual input value $\boldsymbol{u}_k$ does not represent a degree of freedom in and of itself, but instead comes as a direct consequence from the aforementioned change in actuations:

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

All the above constraints must of course apply for all $i = 0, 1, ..., N-1$

2.5.3 The problem formulation

Based on theory and arguments presented in prior sections, using a constant reference $\boldsymbol{y}_{ref}$, the full RNN-MPC formulation is summarized as follows:

$$\min_{\boldsymbol{\Delta u}_{k:k+N-1}} \sum_{i=0}^{N-1} [(\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref})^\mathsf{T} \mathbf{Q} (\hat{\boldsymbol{y}}_{k+1+i} - \boldsymbol{y}_{ref}) + \boldsymbol{\Delta u}_{k+i}^\mathsf{T} \mathbf{R} \boldsymbol{\Delta u}_{k+i}] + \boldsymbol{\rho}^\mathsf{T} \boldsymbol{\epsilon}_y \tag{20a}$$

$$s.t. \forall i \in [0, N-1]:$$

$$\hat{\boldsymbol{y}}_k = \boldsymbol{y}_k \tag{20b}$$
$$\hat{\boldsymbol{y}}_{k+1+i} = \hat{\boldsymbol{f}}_{MLP}(\hat{\boldsymbol{y}}_{k+i:k+i-m_y}, \boldsymbol{u}_{k+i-1:k+i-1-m_u}, \boldsymbol{u}_{k+i}) \tag{20c}$$
$$\boldsymbol{y}_{lb} - \boldsymbol{\epsilon}_y \leq \hat{\boldsymbol{y}}_{k+1+i} \leq \boldsymbol{y}_{ub} + \boldsymbol{\epsilon}_y \tag{20d}$$
$$\boldsymbol{\Delta u}_{lb} \leq \boldsymbol{\Delta u}_{k+i} \leq \boldsymbol{\Delta u}_{ub} \tag{20e}$$
$$\boldsymbol{u}_{lb} \leq \boldsymbol{u}_{k+i} \leq \boldsymbol{u}_{ub} \tag{20f}$$
$$\boldsymbol{u}_k = \boldsymbol{u}_{k-1} + \boldsymbol{\Delta u}_k \tag{20g}$$

3 Implementation and testing

This section describes the implementation of the MIMO linear step response MPC for the single well system and the MLP developed in relation to this project. Pseudo-code and high-level descriptions will be given, such that the process should be reproducable.

Implementing the full linear step response MPC-scheme in code, see Section 3.2, consumed more time than first anticipated that it would - practically all time available for code development was sunk into this part of the project specifically. This was largely due to the learning curve involved in learning about LSR-MPC, but also due to the process of code development itself. Every piece of code and functionality discussed in this chapter demanded fair amounts of iterations on both debugging and testing. Still, *some* time was left for code development of the ANN, see Section 3.3. Unfortunately, no time was left to actually implement the RNN-MPC, as described in Section 2.5.3. Nevertheless, the implementations done in sections 3.2 and 3.3 provided much preparatory learning, and should also provide a useful fundament of code on which to base a future implementation of the RNN-MPC.

3.1 Choice of language and framework

This project involved programming both a basic MPC implementation, as well as a system model in the form of an ANN. In developing these code projects, I used Python (version 3.9) as programming language. MATLAB was considered as an alternative. However I am more familiar with Python, and Python has a vastly available open-source community of both resources - especially the library PyTorch - and examples within the deep learning field. This was also the programming language taught in the subject TTK28 Modeling with Neural Networks, which I took as a complementary course to this project, and so I chose Python.

3.2 The linear step response MPC

I, Simen Bergsvik and Amalie Gjersdal developed an implementation of a MIMO linear step response MPC scheme. We did this cooperatively, as our different theses all had use of an implementation close to today's industry, with which we could compare results from our specific works.

The implementation of the LSR-MPC is from here-on referred to as the "MPC-scheme".

3.2.1 Choice of problem representation

The first step was to choose an LSR-MPC formulation suitable for the single well system. The full step response for the single well system can be described as follows:

$$\boldsymbol{S} = \begin{bmatrix} S_{gas\ rate,\ choke} & S_{gas\ rate,\ gas\ lift} \\ S_{oil\ rate,\ choke} & S_{oil\ rate,\ gas\ lift} \end{bmatrix}, \tag{21}$$

where $\boldsymbol{S} \in \mathbb{R}^{2N \times 2}$.

Since our system is MIMO, we had to find an MPC problem representation that generalized the MPC problem formulation in (6) to the MIMO case. The representation presented in [9] proved a suitable choice. The matrices involved in the full MPC problem formulation and their derivations are extensive, and will consequently not be presented in detail in this project. Instead, the interested reader is referred to [9] (pages 59 and 60) for precise and exhaustive derivations of the matrices involved.

3.2.2 Handling the digital system model

In order to implement the above-mentioned MPC-scheme, we first needed to identify the step response model. This required data that described the input-output relation in our system. The single well system from which we needed data, was represented as a digital model, developed in the programming language *Modelica*.

To interface with the digital model in Python, we first had to export the model as a Functional Mock-up Unit (FMU). With an FMU, we could simulate the system in Python by means of the library *pyfmi*. Exporting the FMU involved a process of much debugging. The final solution involved installing jModelica 2.1.4 and using a 64-bit Python-terminal from among the installation-files as a Python environment, wherein the appropriate commands could be used in order to generate the FMU. Details regarding the specific commands are given in appendix A.

Once the FMU was compiled, we could simulate the single well by loading the FMU into a Python environment, initializing it and then simulating it, step-by-step. The FMU in the context of a Python environment will from here-on be referred to as *the model*. Pseudo-code is provided, but note that arguments to functions are omitted for simplicity:

```
# simulate_fmu.py

from pyfmi import load_fmu

def init_model(...):
    # -- Initialization -- #
    # use load_fmu() to load the model-object
    # use model.initialize() to initialize the model-object

    # Warm start: run simulation loop for sufficient amount of steps

def simulate_singlewell_step(...):
    # -- SIMULATION LOOP -- #
    # use model.set_real() to set the input values
    # use model.do_step() to simulate one step with the given input values
    # use model.get() to retrieve output values
```

3.2.3 Generating the step response model

To generate the step response model $\boldsymbol{S}$, we first initialized the model as indicated in Section 3.2.2. We then applied a step on one input, and recorded the resulting outputs until they settled at some value. Repeating for the second input, we now had all the data between inputs and outputs needed to perform the system identification by calculating the ratio between the output and input [16]. After finding the full step response model, it was stored as a file, such that it could be loaded into the MPC-scheme which would later utilize it. Note that finding the step response model was done separately from running the MPC-scheme.

3.2.4 Code development of the MPC-scheme

This section describes the process of implementing an MPC-scheme capable of calculating optimal control values for the single well system, and how the required step response model was generated. We implemented the full MPC-scheme from scratch - discussion provides reasons as to why. Pseudo-code is provided:

```
# MPC.py

# -- SETUP -- #
```

```
# Setting system configuration parameters
# Generating output reference values
# Loading the step response model
# Construction of matrices

# Initialize FMU, warm start

# -- CONTROL LOOP -- #
# Bias estimation
# Solving the optimization problem
# Simulating the model with optimal input
```

The further paragraphs aim to explain each step of the pseudo-code above in greater detail. The tree describing the file hierarchy of the implementation of the MPC-scheme is presented in appendix B for convenience.

Setting system configuration parameters The system configuration parameters, such as constraint values, reference values and tuning parameters, were stored in a separate file. Reading and setting the values in the main program was done in a general manner, such that testing with different values in the configuration parameters could then be performed without altering the code directly in the main MPC-scheme. This was done with consideration to writing more failsafe code.

Generating output reference values We worked with a constant reference signal for this project, in order to achieve a baseline MPC-scheme. An extension of the MPC-scheme to account for a time-varying reference signal may be implemented, but due to time restrictions, we did not for this project. As such, when having read the desired reference values for gas rate and oil rate from the configuration file, the reference matrix - as described in [9] (page 34) - could be constructed.

The step response model The step response model was already built prior to running the MPC-scheme, see Section 3.2.3, and preparing it for the MPC-scheme became a matter of loading it into variables from data-files.

Construction of matrices The matrices used to formulate the optimization problem itself depended on system configuration parameters, output reference values and the step response model, and could thus at this point be constructed. Construction of each matrix was done in separate functions, which were stored in a separate file: *matrix_generation.py*. This was a design choice during development, made in order to reduce the amount of code in the main program flow, as well as facilitating modularity and unit-testing of each specific function. Unit-testing each of these functions proved essential, as the full code of the MPC-scheme became somewhat involved, and was consequently prone to errors. Importantly, the whole MPC-scheme would fall apart if only some part of some matrix construction function was not working as per intention, since the correctness of the solution to the optimization relied on the correct representation of our MPC formulation. Any errors in any matrix would make the MPC formulation in practice different from what we aimed for, and so the structure in keeping these important functions separated into their own file proved extremely useful.

Together, these matrices formed the total formulation of the optimization problem, system dynamics, state penalty weighting and constraints.

Initialize FMU The FMU had to be initialized before running the optimization loop, as one step of the model was simulated for each iteration of the loop. Note that we used warm start for the model, meaning that we simulated the model with the inputs set to the same values as the point of linearization. This simulation was set to run for a sufficient amount of steps *within* the initialization routine, such that the system dynamics settled around the point of linearization

before simulation proceeded. This was done as an attempt to avoid undefined behaviour that might occur if the initial values of the model's internal parameters and variables - external to the scope of this project - are not known.

Bias estimation We implemented bias estimation as a means of improving the MPC-scheme's accuracy. The bias estimation was performed based on the difference between prediction and measurement of the outputs at each step k, and thus had to be part of the optimization loop before the call of the solver itself.

Solving the optimization problem As indicated, the main goal of all stages presented above was to build the full optimization problem formulation. Solving the optimization problem simply meant calling the solver of choice, setting its cost function and constraints to correspond to the newly constructed problem formulation matrices. The resulting output contained the optimal control sequence given the desired reference and system formulation, and could then passed to the the model for simulation of the model. See Section 3.2.5 for choice of solver.

Simulating the model with optimal input Lastly, we simulated the model for one single timestep by feeding it the first optimal control value from the control value sequence granted by the optimization solver. The simulation returned the output values corresponding to the given input. Logging these, and iterating through the optimization loop for some predefined amount of steps, we constructed a timeseries of input-output dynamics, which could be plotted in order to observe the performance of the MPC-scheme.

Note that the simulation functionalities were implemented as functions in a separate file, *simulate_fmu.py*, also in order to facilitate modularity and unit-testing. This was even more useful than the case of *matrix_generation.py*, as these functionalities were necessary when generating the step response models as well as when running the MPC-scheme.

3.2.5 Choice of solver

We initially intended to solve each iteration's optimization problem using the open-source solver *osqp*. However, issues arose with the semantics in our implementation of the optimization problem's matrices and the semantics of osqp. Due to time restrictions, choosing a different, more easily compatible solver seemed like a more viable path. We thus opted to use the commercial variant *gurobi* instead. Note that the gurobi solver is a commercial alternative. Whether it is a worthwhile investment in a hypothetical application of this project's results boils down to performance differences. Measuring and evaluating such performance differences fall beyond the scope of this project.

3.2.6 Performance testing of MPC-scheme

This section only describes the manner in which I tested and tuned the performance of the MPC-scheme - results will be presented in Section 4.1.

Section 2.1.2 presents some guidelines for tuning an MPC-scheme. Given the chosen problem formulation in Section 3.2.1, the weighting matrices $\bar{\boldsymbol{P}}$ and $\bar{\boldsymbol{Q}}$, as well as the control and prediction horizons H_u and H_p, are the parameters available for tuning.

It follows from the problem formulation chosen in Section 3.2.1 that $\bar{\boldsymbol{Q}}$ is the diagonal weighting matrix for the measured outputs. Thus, making its elements high-valued would mean that deviance in the corresponding outputs from the desired references should be prioritized. Conversely, making its elements low-valued would make the MPC-scheme prioritize the deviance between outputs and references less.

$\bar{\boldsymbol{P}}$ is the diagonal weighting matrix for the changes in actuations. Thus, making its elements high-valued would make the MPC-scheme prioritize smaller changes in the inputs, and vice-versa.

The goal of the MPC-scheme is to track the reference outputs as closely as possible, all the while utilizing as little expensive actuation as possible. Furthermore, we base the tuning of the MPC-scheme on a desire to track both the reference in gas rate and oil rate equally well. Section 1.3 describes gas-lift rate as an expensive actuation, whereas utilizing choke-opening as an actuation is virtually free. Note that, even though choke-opening as an actuation is inexpensive, too aggressive control may lead to high degrees of wear on the actuator, and thus its weight should not be simply 0.

Given the above regards, the initial guess of values for $\bar{\boldsymbol{Q}}$ and $\bar{\boldsymbol{P}}$ is:

$$\bar{\boldsymbol{Q}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \bar{\boldsymbol{P}} = \begin{bmatrix} 10^{-2} & 0 \\ 0 & 10^{2} \end{bmatrix}$$

The goal in tuning the horizons H_u and H_p was to make the MPC-scheme predict far enough ahead in order to calculate an optimal control sequence that would not yield aggressive changes in the actuators. Note that high values for H_u and H_p would be expected to yield longer computational times, thus there is a trade-off between the MPC-scheme's performance and its computational time when increasing H_u and H_p. Though no formal requirement with respect to computational time has been made in this project, the tuning will take computational times into account by attempting to maintain the values of H_u and H_p as low as possible.

Given the above regards, the initial guess of values for H_u and H_p is:

$$H_u = 50, \qquad H_p = 80$$

The chosen MPC formulation has a term H_w representing time-delay in the input-output relations. I assume $H_w = 0$ throughout the testing procedure, unless indications imply otherwise.

Though the here presented arguments will work as guidelines in tuning the MPC-scheme, they are ultimately only guidelines, and the tuning itself is a trial-and-error process, following these guidelines as closely as possible. Note that the final tuning parameters' values may deviate significantly from the above-presented values, in which case underlying reasons as to why will be discussed in Section 5.1. Consistent with the system description in Table 1, the trial-and-error process of tuning would consist in simulating the system for 7200 simulation seconds, and evaluating how closely the MPC was able to track the reference. Further system parameters were chosen in accordance with values presented in the system description, see Section 1.3.

Although it would be interesting to investigate other possible set-points in the process of testing the MPC-scheme's performance, this project considers only one constant set of reference values throughout all testing. This is done due to time restrictions, and in order to maintain consistency between tests.

The specific values chosen to be tracked are chosen upon advice from this project's industrial partner on what may be interesting values to track. The MPC-scheme's potential ability to track these reference values, or lack thereof, is discussed in Section 5.1.

Set-points	Value
gas rate reference value	11000 $[\frac{\mathrm{m}^3}{\mathrm{h}}]$
oil rate reference value	305 $[\frac{\mathrm{m}^3}{\mathrm{h}}]$

Table 2: The chosen reference values for testing the performance of the MPC-scheme.

3.2.7 Testing input-output relations on the ground truth model

As a way of supplementing the process of testing and tuning the MPC-scheme, I also performed two experiments to investigate the effect of each single input on the outputs. In each experiment, one input was kept constant, while the other was varied within some range - see Table 3. The range within which each individual input varied reflects realistic values for that respective actuator, consistent with Section 1.3. Due to time restrictions, I used the same type of input-profiles as are explained in Section 3.3.2.

Experiment nr.	Choke-opening	Gas-lift rate
#1	$[0, 100]$	0
#2	50	$[5000, 10000]$

Table 3: The experiment values on the inputs for investigating each specific input's effect on the output.

Results from this investigation are presented in Section 4.1.

3.3 Approximating system dynamics by means of an ANN

As presented in Section 1, the goal of this project is to approximate the input-output relations of a single well by means of an ANN. The following sections argue for this project's choice of ANN architecture, as well as present the process of implementing the ANN.

3.3.1 Choice of neural network architecture

Even though Section 2.4 argues that an ANN integrated into an MPC problem formulation should be of the type RNN, this section describes the implementation of an MLP instead. This is because this project only implements an ANN to approximate the single well's input-output relations, and does not *integrate* it into an MPC formulation. Since the MLP organically extends to an RNN when predicting sequences of data, as shown in Section 2.4, the MLP developed in this project may simply be feedback-connected to itself in order to create an adequate RNN for future work on implementing the full RNN-MPC.

I chose to implement the ANN with an MLP architecture with only a single hidden layer, as this was done with success in [17], and has been shown to be theoretically sufficient to achieve potentially universal approximation - see Section 2.3.1. This choice would also prove beneficial in simplifying the search for the optimal set of hyperparameters, as is discussed in Section 3.3.5. Lastly, an additional argument may be made for using only a single hidden layer, as this simplifies the search for the optimal set of hyperparameters, and such an ANN is theoretically still capable of approximating any nonlinear function - see Section 2.2.2.

3.3.2 Generating datasets

As mentioned in Section 3.2, the system I have worked on is a digital model of the real system. In order to acquire datasets to use with the MLP, I performed the same procedure as in Section 3.2.3 to log the output for some given input.

The input profiles I generated for this purpose were designed to somewhat resemble true input profiles the real physical single well system could face. To this end, two factors were considered: In a physical system, an increase or decrease in control values would be *discrete*, and *input-blocking* is also common [16]. In order to ensure an i.i.d. distribution of steps in control values around some initial values of the inputs, I generated the input profiles as sequences of steps, where the sign -

positive or negative - was drawn from a uniform distribution; an increment was equally likely as a decrement. For each increment or decrement, the input value was kept static for two timesteps to emulate input-blocking. This data-generating process created input profiles resembling a random stairwalk - see Figure 12. I created several such randomly generated input-profiles, normalized them to the range $[0, 1]$, and logged their corresponding output values after simulation, such that all input-output relations could be grouped in .csv-files. Then, simply using different .csv-files for training and test datasets would then ensure them being disjoint, but still i.i.d.

3.3.3 Preparing and dividing the datasets

The next step was to prepare the logged datasets for training. To this end, I wrote a class in Python, implementing a dataset consisting of all the samples contained within some given .csv-file. Instances of this class could then be created, given the path to some appropriate .csv-file, after which this instance can be made into a *Dataloader*-object, as used to train and test ANNs in the PyTorch-framework.

A final consideration to be made before using the datasets, was how to divide them between training, validation and testing. As argued in Section 2.3.1, the ANN's ability to generalize is improved if the training, validation and test datasets are fully disjoint. In order to ensure this, simply generating different datasets from different, input profiles would suffice, as long as the different input profiles followed the same probability distributions - as emphasized in Section 2.3.3. Note that this requirement should already be met, as explained in Section 3.3.2. Furthermore, splitting the available data into ratios, as explained in Section 2.3.2 is not necessary in this case, as the amount of data is limited only to what I generate myself. Using three separate datasets for training, validation and testing, respectively, is thus be feasible. Different datasets were then created and made into Dataloader-objects as explained.

3.3.4 Code development of the ANN

In order to have a good work-flow in finding optimal hyperparameters and performing the training of the ANN, I first developed the code implementing the ANN and the required overhead.

The code was structured as illustrated in Appendix C, and files referred to may be located there. In order to train the ANN, *main.py* was be run. An overview of this file is presented below as pseudo-code:

```
# main.py

class GroundTruth(): ...

def main(): ...
    for hyperparameters in sets_of_hyperparameters:
        # Load hyperparameters
        # Load data from .csv-files
        # Train ANN - pass hyperparameters and .csv-files as arguments

        # Plot ANN's performance against ground truth

        # Save model to file
        # Save figures to file
```

As indicated, *main.py* would load one set of hyperparameters and some datasets, currently in the form of .yaml- and .csv-files, respectively. One full training cycle would then be run on that specific configuration of hyperparameters and datasets, before the performance would be evaluated. After saving the results to file, the whole process would be repeated, now for a different set of hyperparameters, but the *same* datasets.

Note that the work done in this project utilizes the automatic initialization offered by the PyTorch-framework, being the *Kaiming* method. The interested reader is referred to the open-source implementation [15].

Using the same set of datasets was done in order to create a consistent training, validation and testing environment for different sets of hyperparameters. In theory it should not be necessary, as properly designed datasets should be representative enough of the dynamics that the ANN shall learn; it should not matter which datasets are used, if they all represent the same input-output relations equally well. Nevertheless, I chose to implement it like this to create consistency.

Each iteration of the main-loop trained an ANN by means of the functionalities implemented in *neuralnetwork.py*:

```
# neuralnetwork.py

class NeuralNetwork(): ...
class EarlyStopping(): ...

def train_loop(): ...
def validation_loop(): ...
def test_loop(): ...

def train_ANN(): ...
def predict(): ...
def load_ANN(): ...
```

The *NeuralNetwork*-class implemented the ANN itself, defining hyperparameters and the MLP-architecture. The *EarlyStopping*-class implemented the regularization technique early stopping, introduced in Section 2.3.4. The different loop-functions defined were used to train, validate and test, respectively, while *train_ANN()* served as the entry-point for *main.py*, which converted the data in the .csv-files to Datasets, and actually ran the training, validation and testing. *predict()* differed from *test_loop()* in that it predicted input-output relations given some dataset for an *already trained* ANN, loaded by *load_ANN()*.

3.3.5 Training and evaluation of the ANN

As described in Section 2.3.2, the first step in training the ANN is to find the best set of hyperparameters, for which some range of candidate-values should be tentatively determined. The candidate values are presented in Table 4. For explanations of the hyperparameters, the reader is referred back to sections 2.3 and 2.4.

Hyperparameter	Description	Values
learning rate	the learning rate of the SGD-method used in training	10^{-3}
batch size	the amount of samples used in each step of the training	100
epochs	the number of epochs for which the ANN should train	500
patience	the number of epochs to wait for a better E_{val}	5
m_u	the size of the history of inputs considered	$[0, 2]$
m_y	the size of the history of outputs considered	$[1, 3]$
η	the size of the single hidden layer	$2^i, i \in [4, 6]$

Table 4: The candidate values for hyperparameters.

In order to retain the amount of sets of hyperparameters at a feasible level, I chose to vary the sets of hyperparameters only in some specific values. The learning rate, batch size, epochs and

patience were kept constant throughout, while values of m_u, m_y and η were varied within the interval specified in Table 4. Note that the amount of epochs varied from each training-iteration, as early stopping was used as a method of regularization, see Section 3.3.4.

I chose to vary only m_u, m_y and η because they alter the ANN's structure, and thus capacity - see Section 2.3.1. Since I made an argument that the ANN's capacity needs not be very high, these variables were the ones for which I could say something about a reasonable interval within which to search for the best hyperparameters. More specifically, I assumed the hyperparameters of this project to resemble those obtained in [17], and created intervals within which to search thereafter.

The intervals presented in Table 4 represent 27 different sets of hyperparameters, for all of which the process described in Section 3.3.4 was performed in order to determine the optimal set. Note that the set of hyperparameters deemed best would be the one minimizing the validation error, consistent with Section 2.3.2. Results are presented in Section 4.2.

I wanted to find the best set of hyperparameters with some certainty. Thus, I first performed a "filtering" of sets of hyperparamters, such that the best subset of the full list of candidates could be trained on larger datasets. The sets of hyperparameters that yielded an ANN with better capabilities than the rest, should also perform better when trained on a larger dataset, due to the increased amount of data, from which to learn to generalize, see Section 2.3. I chose to train only a subset of the sets of hyperparameters on larger datasets in order to save computational time. The procedure was as follows: I first trained the model on each set of hyperparameters, using the same two disjoint datasets as training and validation datasets across each training in order to ensure maximal comparability. Among the trained ANNs, the best performing 33% were selected for further training with a larger dataset. After training this subset of set of hyperparameters on the larger dataset, the set of hyperparameters finally deemed best was the one minimizing the validation error. Finally, after choosing a set of hyperparameters, the generalization error could be approximated by the test error E_{test}, calculated as the mean square error of the ANN's predictive accuracy on a different, fully disjoint test dataset.

4 Results

This section presents the results seen from the implementations as presented in sections 3.2 and 3.3 as a prerequisite to the discussion that will follow in Section 5.

4.1 Linear step response MPC results

The trial-and-error process of tuning the parameters' values outlined in Section 3.2.6 resulted in the values presented in Table 5. The corresponding results of a simulation of two hours of control towards the desired reference values are presented in Figure 8.

Tuning parameters	Value
$\bar{\boldsymbol{Q}}$	$\begin{bmatrix} 10^{-7} & 0 \\ 0 & 100 \end{bmatrix}$
$\bar{\boldsymbol{P}}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
H_p	300
H_u	200
H_w	0

Table 5: The configuration of the MPC-scheme. The reference values were kept constant.

The MPC-scheme controls the oil rate fairly well to the desired reference value, while the gas rate is controlled to a constant offset from the reference value. Both output-profiles have oscillations, though they are most prominent in oil rate, which overshoots significantly before being able to track its reference.

When tuning $\bar{\boldsymbol{Q}}$, an increase in its values caused the control values to change faster, leading to an oscillatory response in the outputs. Conversely, a decrease in its values caused the changes in control values to be slow-changing, and the response in the outputs to look highly dampened.

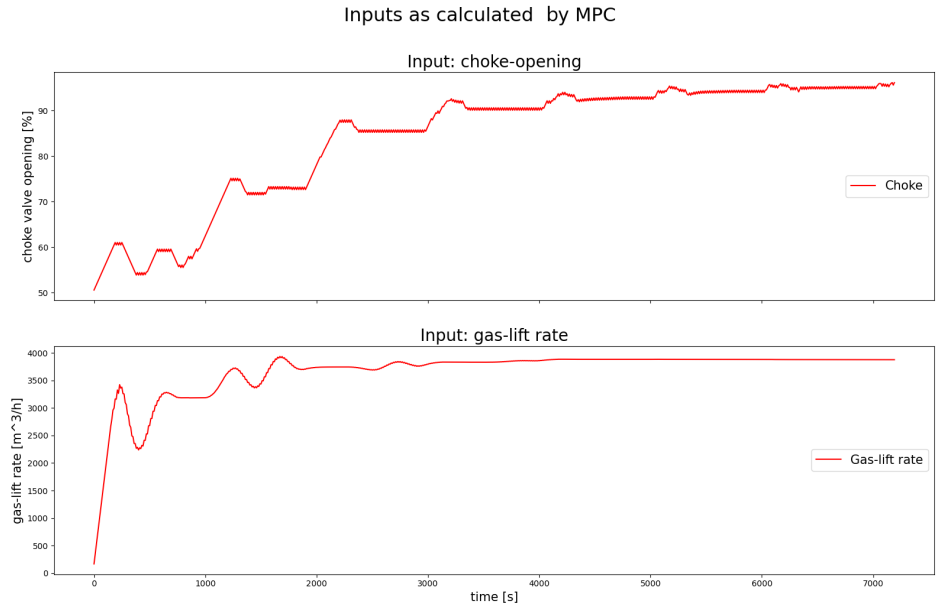
When tuning $\bar{\boldsymbol{P}}$, an increase in its values caused usage of the corresponding control variables to be less slow-changing. Conversely, a decrease in its values caused the changes in control values to be faster, leading to an oscillatory response in the outputs.

Ultimately, the chosen set of tuning parameters were obtained as an attempt to make the MPC-scheme focus on ensuring that at least one output was led to track its reference, with some emphasis on using choke-opening more actively than the gas-lift rate in so doing.

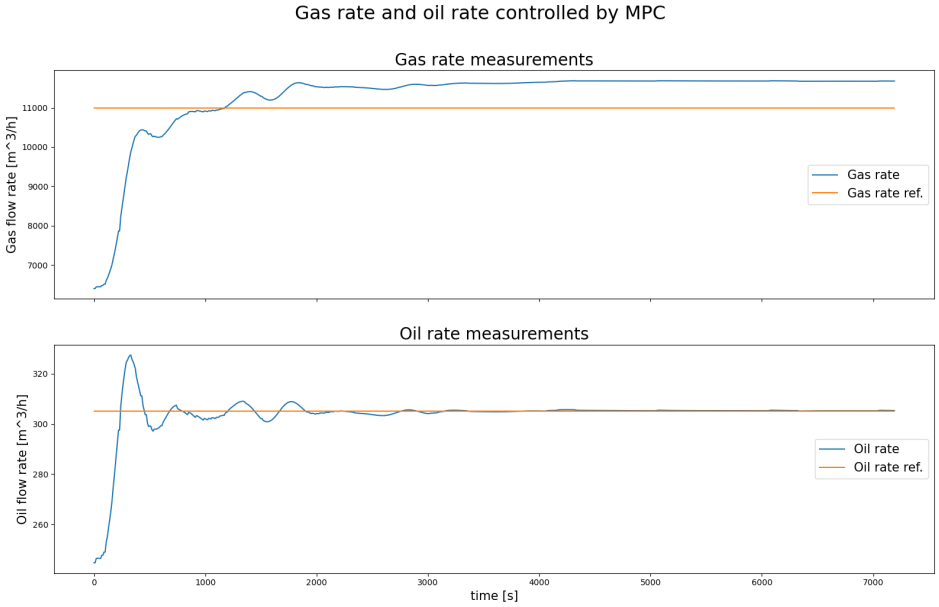
Following the argumentation in Section 2.1.2, the chosen values for $\bar{\boldsymbol{P}}$ imply that changes in actuation are *not* weighed equally, even though the weights are the same values. Since values of gas-lift rate, and thus changes in gas-lift rate, are of a much higher order of magnitude than those of choke-opening, changes in gas-lift rate are in reality weighed more heavily than changes in choke-opening. Thus, the MPC-scheme was set to prioritize choke-opening as a means of actuation.

During tuning, trying to find a balance between the values of $\bar{\boldsymbol{Q}}$ such that both outputs tracked the reference resulted in oscillations in the outputs, and no output was tracked for increasing time. Instead choosing to prioritize one output to be tracked indeed led that output to be tracked, at the expense of the other output obtaining a steady state error for increasing time, as seen in Figure 8b. The final values of $\bar{\boldsymbol{Q}}$ imply that the MPC-scheme should prioritize reaching the reference value in oil rate over gas rate, which is reflected in the results.

Note that the argument of different orders of magnitude may also be made between gas rate and oil rate. Since the difference in magnitudes between gas rate and oil rate is much smaller than the difference in magnitudes between their respective weights, this consideration is already taken into



(a)



(b)

Figure 8: An input profile calculated by MPC leading to an output profile.

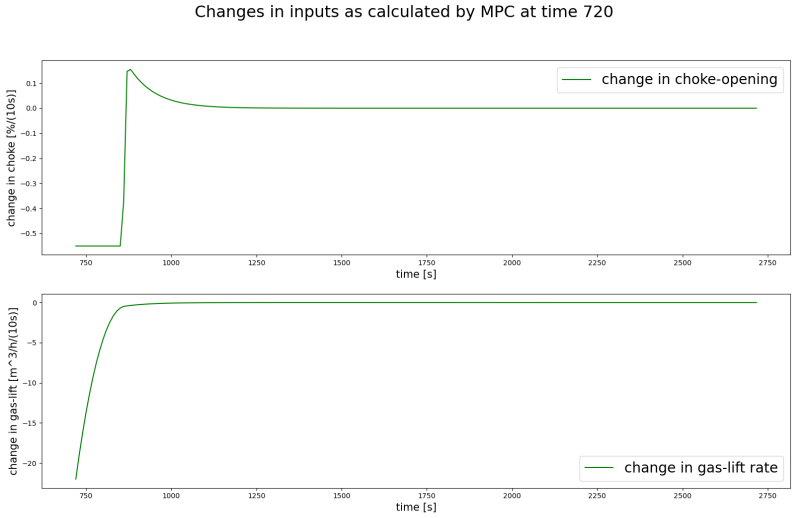
account in the final tuning values.

Tuning the prediction and control horizons H_p and H_u showed some effect on the final value of the two outputs, but little effect on the qualitative trajectories towards the references. Any increase or decrease in gas rate always seemed to correlate with an increase or decrease in oil rate - and vice-versa, but only when the system was at steady state. For example, when increasing H_p, both outputs would have some decrease in the final value, though the gap between them would not reduce. Increasing H_p thus resulted in the gas rate settling closer to its reference, while the oil rate undershot its reference. While an increase in H_u did reduce the steady state error, thus improving performance, it also shifted both outputs away from the reference when made very high. Furthermore, any increase in either H_p or H_u would give noticeably longer computational times.

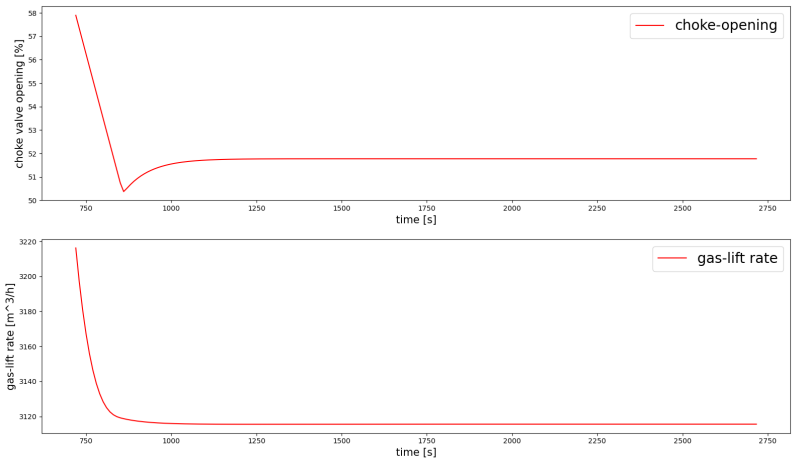
Thus, the final values for H_p and H_u were chosen as the best compromise between computational times and performance with respect to tracking the reference values for the outputs.

Neither the final results, nor the results seen during tuning, indicated that the process responded with any particular time-delay. As such, H_w was kept constant at 0 throughout the testing, as assumed in Section 3.2.6.

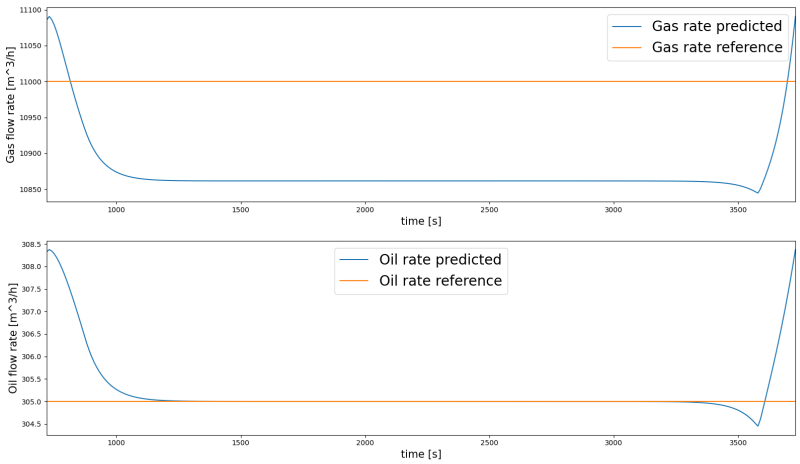
Figure 9 shows that the tuning strategy of prioritizing reference tracking in oil rate is further supported by the sequence of optimal changes in control, the consequent sequence of optimal control values and the corresponding predicted outputs. Although Figure 9 is the result from the MPC-scheme's calculations at $t = 720$ $[s]$, the plots looked quite similar at almost any other given timestep.



(a)



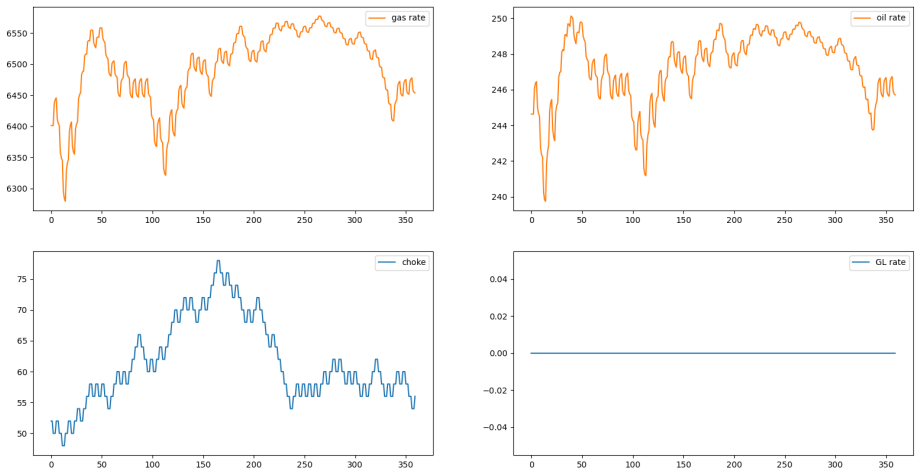
(b)



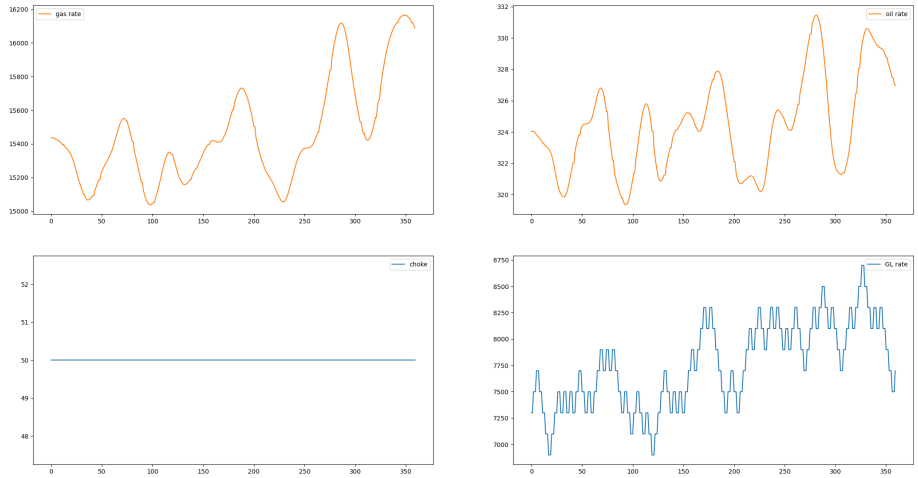
(c)

Figure 9: An input profile calculated by MPC leading to an output profile.

The results from running simulations with the input-profiles described in Section 3.2.7 are presented in Figure 10.



(a) Experiment #1: varying choke-opening while keeping gas-lift rate static.



(b) Experiment #2: keeping choke-opening static while varying gas-lift rate.

Figure 10: An input profile calculated by MPC leading to an output profile.

The results in Figure 10a show that a decrease in choke-opening leads to lower mass-flow rates - both in gas rate and oil rate - and vice-versa. Note that small changes in choke-opening sometimes lead to large changes in the amplitude of the outputs, see for example $time \in [0, 50]$. At other times, large changes in choke-opening lead to relatively small changes in the amplitude of the outputs, see for example $time \in [150, 250]$. Also noteworthy is that each step made in the choke-opening seems to have a corresponding impulse-response in both outputs. Whether the amplitudes in the outputs are large or small, the outputs change fast in response to changes in choke-opening.

The results in Figure 10b indicate that an increase in gas lift rate leads to an increase in both mass-flow rates, and vice versa. Comparing with the effects from changing the choke-opening, the output responses to changes in gas lift rate have comparable gradients, but seem less acute. The ranges within which the outputs change as response to the inputs are relatively similar, but the

Hyperparameter	Description	Values
learning rate	the learning rate of the SGD-method used in training	10^{-3}
batch size	the amount of samples used in each step of the training	100
epochs	the number of epochs for which the ANN should train	500
patience	the number of epochs to wait for a better E_{val}	5
m_u	the size of the history of inputs considered	1
m_y	the size of the history of outputs considered	2
η	the size of the, in this case, single hidden layer	2^5

Table 6: The final values for the hyperparameters of the ANN.

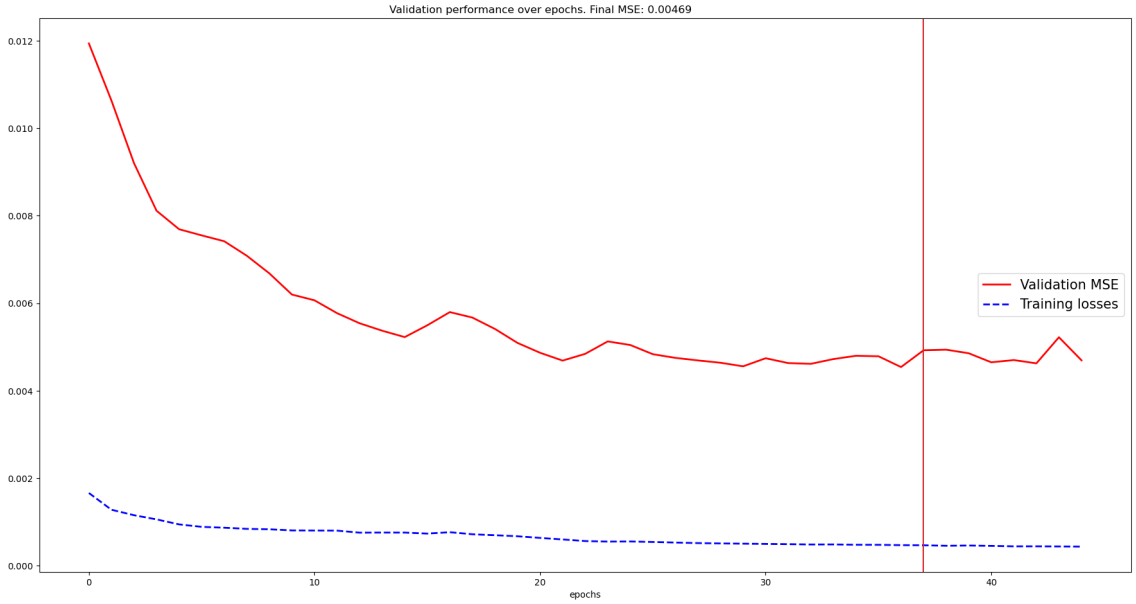


Figure 11: Validation and training errors E_{val} and E_{train} for the training of the ANN.

dynamics in Figure 10b seem smoother than the ones in Figure 10a.

Interestingly, in both Figure 10a and Figure 10b, both of the mass-flow rates change in very similar manners when subject to changes in inputs, no matter which input is changed.

4.2 Neural network training results

In finding the best set of hyperparameters, I applied the iterative procedure described in Section 3.3.5, training and evaluating the ANN for every candidate set of hyperparameters, as listed in Table 4. The set of hyperparameters described in Table 6 yielded best performance with regards to E_{val} among the sets tested, and was chosen as the set of hyperparameters for further training.

The final training of the ANN yielded the curves for training and validation errors as shown in Figure 11. The vertical red line indicates the point after which early stopping detected that overfitting occurred. The results from testing the fully trained ANN are shown in Figure 12.

Though the ANN predicts general trends of the input-output relations accurately over the whole test dataset, the predictions have large offsets specifically around the regions with less fast dynamics - for example $time \in [2200, 2800]$ in Figure 12. Notably, the faster dynamics are approximated with almost no visible error for gas rate, and only some error for oil rate. Reasons why this behaviour

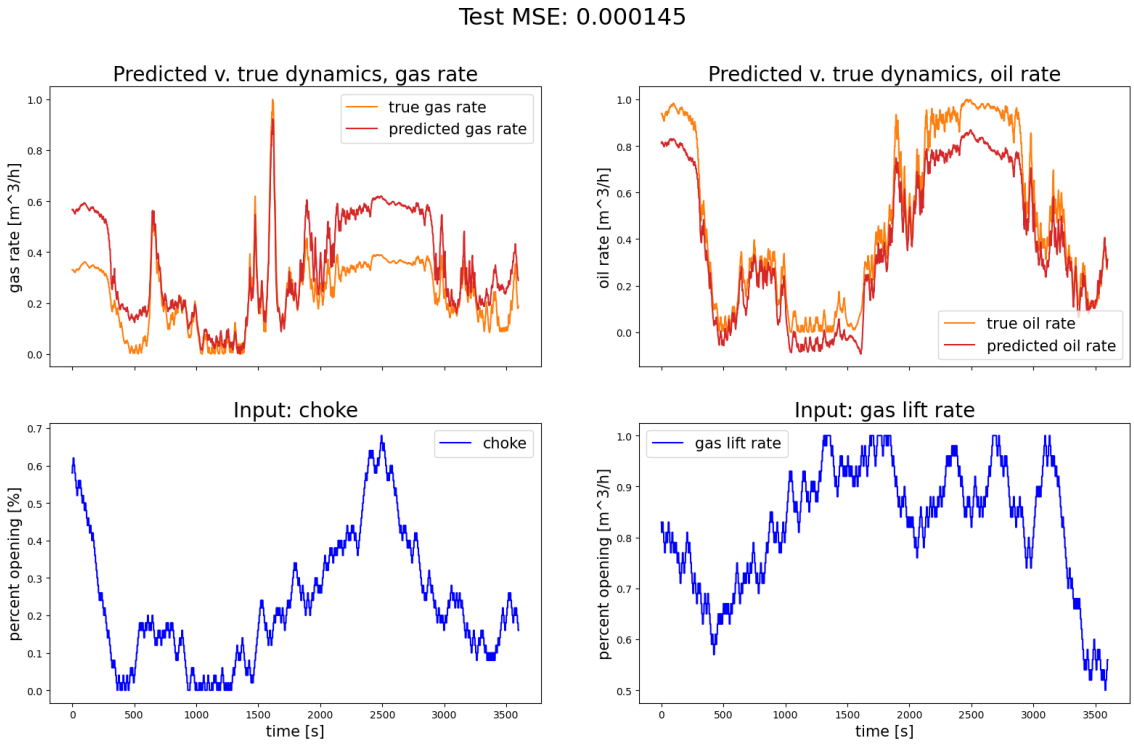


Figure 12: The illustration shows the resulting output (orange) for gas rate (top left) and oil rate (top right) given the inputs (blue) choke-opening (bottom left) and gas lift rate (bottom right). The ANN's predicted output (red) is added for comparison, for which the total MSE is given in the title.

occurs are discussed in Section 5.2. The total test error for both outputs over the whole test dataset is the MSE, and is calculated to be $E_{\text{test}} = 0.000145$, indicating a low error on average. Note that E_{test} is valid only for the *normalized* data shown in Figure 12, and thus describes the errors relative to their magnitudes.

Interestingly, during the above-described procedure, many sets of hyperparameters terminated after only around 5 epochs, as E_{val} increased from the start - illustrated in Figure 13. Reasons as to why are discussed in Section 5.2.

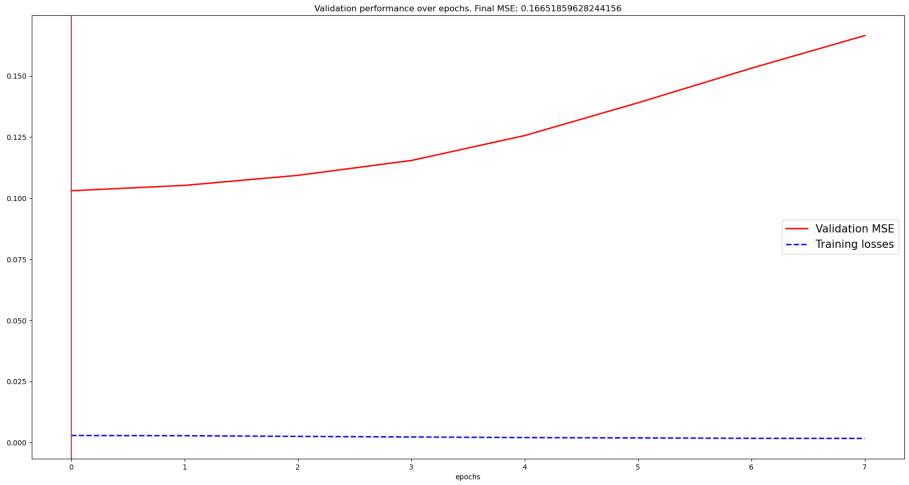


Figure 13: Example of early stopping terminating the training early due to validation increasing from the beginning.

5 Discussion

This section discusses the implications of and reasons behind the results presented in Section 4.

5.1 Linear step response MPC results

The fact that the MPC-scheme is able to control one of the outputs to the reference indicates that there may be errors in the implementation of MPC-scheme's bias estimation. However, were this true, then the MPC-scheme would have been given wrong measurements throughout the full simulation time. Section 4.1 presents observations that, even though tracking of oil rate reference was emphasized in the tuning, the MPC-scheme was also able to track gas rate, given appropriate tuning values. Since both outputs may be made to track their reference, but only individually, and not simultaneously, I argue in this work that the bias estimation works as intended. Had it not been fully functional, I would have expected to see no reference tracking at all, since the information fed from the bias estimation to the solver would convey simply *wrong* information about the current system state. Had the solver had erroneous information in this manner, I would expect it to track what it would wrongly "believe" was the reference, resulting in some offset, with outputs never properly reaching their references. Verifying or falsifying the correctness of the bias estimation implementation would still be interesting in future work, in order to ensure the correctness of the implementation of the MPC-scheme as a whole.

A different potential reason for the steady state error observed in gas rate is here presented. The linear step response model was made around a working point of choke-opening $= 50$ [%] and gas lift rate $= 0$ $[\frac{\mathrm{m}^3}{\mathrm{h}}]$. Since the references are set so high that both high values in choke-opening and gas lift is required by the controller in order for the references to be reached, the inputs quickly move away from the point of linearization, see Figure 8a. As commented in Section 2.1.5, any linearization is a more accurate approximation around the point of linearization, than further away. When the references given the system in this case make the MPC-scheme calculate optimal control far from the point of linearization, it thus follows that the system output predictions performed by the MPC-scheme become less accurate. Remedies could be either operating with references causing the MPC-scheme to calculate optimal control sequences closer to the point of linearization, or employing different step response models for different ranges of control inputs. While this would likely improve control, it is important to consider that any predictions made just before such a step response model switch could be very different from the predictions made right after the switch. Whether this would be an issue, or simply an improvement would be interesting to investigate in future work.

The fact that the MPC-scheme tracks only one output during steady state, even though its tunings imply a prioritization, indicates an inability in the given system model to accurately represent the nonlinearities stemming from the *interconnectedness* of the two outputs, and the effects each input has on each output, both individually and connectedly. This issue is also likely to stem from modelling inaccuracy, and may explain why, at steady state, making a change to one output always gives a change in the other output as well. If the model does not accurately convey the MIMO dynamics of the system, then the controls calculated by the MPC will be naïve and inaccurate; any attempt to remedy a steady state error in one output may lead to corresponding changes in the other output, without the MPC-scheme "understanding" that precisely this will happen - due to modelling inaccuracies. Even though the MPC-scheme is supposed to account for offset errors with bias estimation, it is unable to calculate a sequence of changes in control that lead the erroneous output to its reference value without moving the other away from *its* reference value.

Another aspect of the modelling inaccuracies, is that they might add up if they are biased in either positive or negative direction, with respect to the true dynamics. This would be especially impactful during intervals of large actuation, such as $time \in [0, 1000]$ in Figure 8, where small errors may lead to cumulatively large errors, due to steep gradients. In such case, if the MPC-scheme calculates optimal control sequences with the goal of tracking one of the two outputs, these control sequences are not calculated to compensate for this modelling inaccuracy during transition. When the system no longer is in transition, and cumulative errors becomes a resulting steady state

error, the MPC-scheme might still not be able to control each mass flow rate individually, since they change very much in parallel, as illustrated in Figure 10. If this is the case, this indicates that a tuning that prioritizes of the MPC-scheme that prioritizes one output over another is not feasible if both outputs are to be tracked.

In light of the arguments made for modelling inaccuracy in the implementation presented in this project, it is noteworthy that the MPC-scheme is able to track any reference at all. That it is able to at least track the reference of one output, is testament to the robustness of MPC as a control method, as hinted to in Section 2.1.1.

As displayed in Figure 9c, one can observe that the MPC calculates an optimal control sequence with the aim of tracking the reference for oil rate, but not gas rate. This corresponds to expectations after tuning, as more weight was put on tracking the oil rate than the gas rate. Note that the calculated sequence of changes in control and consequent sequence of control values, Figure 9a and Figure 9b, are somewhat aggressive, indicating that the MPC-scheme puts much more weight on tracking the reference, than on being "careful" with actuations. This is also consistent with expectations from the chosen tuning values, Table 5, and arguments made above. However in a theoretical application on a physical system, this would result in potentially excessive strain on the actuators, and could thus be considered a suboptimal solution. Solving the issue with modelling inaccuracy would likely make the MPC-scheme calculate control sequences that would be able to more precisely control the system to track its references, potentially allowing more "careful" usage of the actuators. Notably, the MPC-scheme calculates control that creates a "sawtooth"-profile for the choke-opening, creating rapid oscillations in the choke-valve. This should be avoided in order to prolong the actuator's lifetime. Choosing an MPC formulation that calculates the optimal control sequence directly, and not the optimal changes in control, might also be interesting to investigate, as this somewhat alter the tuning process. Also note that, as formulated now, the weights in $\bar{\boldsymbol{P}}$ imply only that *changes* in gas lift are expensive and that *changes* in choke-opening are inexpensive. In reality, the value of the control itself, not its change, determines whether it should be considered an expensive means of control or not. Encoding such economic considerations into the MPC formulation could be done by instead calculating optimal control based on direct control values, not their rates of change.

As pointed to in Section 4.1, increasing H_p and H_u at some point introduced a slight steady state error in *both* outputs, even though the system previously was able to track at least *one* output. This seems counterintuitive, since the Receding Horizon-principle increases robustness in the MPC-scheme for increasing horizons, see Section 2.1.1. An imaginable cause might be the modelling inaccuracies; future work could investigate whether sufficient modelling inaccuracies may induce steady state errors, since the MPC-scheme likely calculates optimal control based on sufficiently inaccurate predictions of future system state.

Tuning the MPC-scheme only by measure of eye proved a long and tedious process, with little rigorous evidence that the chosen tuning is indeed the best possible. Future work should investigate different more rigorous measures of determining the performance of an MPC-scheme, and implement a test-routine to iterate over many configurations of tuning parameters and storing the results in a more automated fashion.

As indicated earlier, the gas rate and oil rate are tightly interconnected, and consequently control of the individual mass flow rate fails during steady state. A way to understand how to better control each of them individually, might be to include one or several further states as outputs of the system. Adding for example the *well head pressure* as an output could give direct information as to the effects on pressure arising from altering choke-opening and/or gas lift rate, possibly facilitating improved performance by the MPC-scheme. Interesting future work might thus include expanding the system model. It could also include a controllability analysis of the expanded system, to investigate whether actually controlling the gas rate and oil rate separately is even feasible.

Lastly, the question of why we did not utilize an existing toolbox for MPC in python, an example being the *do-mpc* toolbox ([11]), may arise. First, a step back: the high-level goal of the MPC-scheme is of course finding the optimal control sequence for every timestep in order to track some given reference values for the outputs. In order to achieve this, a solver must be called. However, the solver required that we implemented generated, stored and loaded the step response model,

implemented system configuration parameters, as well as generated output reference values, the required matrices and performed bias estimation, such that the minimization problem to be input to the solver may actually be formulated. All these pieces of overhead were specific to our MPC formulation, and would need to be implemented regardless of what MPC toolbox we could have chosen to aid in specifically the solving of the optimization-part of the MPC-scheme itself. The case would have been different had the MPC formulation been a more typical one, as *do-mpc* has many convenient facilities in such a case. Ultimately, even though utilizing a python MPC toolbox would be a way of accessing a solver, it provided no further facilities for our specific case. Thus, we opted to not utilize a python MPC toolbox, and instead find and implement use of an optimization problem solver ourselves.

5.2 Neural network training results

That the ANN predicts fast dynamics with high accuracy, usually displaying a constant offset, though still predicting the correct trends during slow dynamics, indicates that there are additive effects in the input-output relations that the ANN is unable to capture. An assumption made in this project was that the final values for m_u and m_y would resemble those in [17], ultimately leading to the final values chosen in Table 6. The direct consequence of this, is that the information available to the ANN of the history of inputs and outputs is greatly limited. This quickly becomes a significant hindrance for the prediction if the system is subject precisely to such additive effects; that long-lasting effects of previous inputs add to any effects of current inputs. Given the qualitative behaviour of the results observed in Figure 12, I argue that this is the case. Consequently, the system may *not* be modelled as a Markov chain as hinted to in Section 2.4.1, and interesting future work should investigate greatly increasing the values of m_u and m_y, such that long-lasting input-output relations may be accounted for.

The experiments performed in Figure 10 indicate that the outputs respond faster to changes in the choke-opening, than to changes in the gas lift rate, suggesting that altering the pipe's physical properties has more impact in affecting the flow rate than what altering the total fluid's density has. Though this work will not delve into the details of the gas and oil fluid mechanics, these results are still interesting with respect to future investigation into each input's individual contribution. As hinted to in Section 2.4.1, performance may differ if m_u and m_y are chosen specific to each input and output, respectively. Investigating possible effects of this is interesting for future work.

A central question to consider when evaluating the performance of the ANN is whether the dataset used during training provides general enough information of the input-output relations, such that the ANN is able to generalize after training. Though the input-profiles I have used to generate datasets in this project are random and do not resemble realistic data, they do perform steps on the inputs, and steps are from digital signal processing known to contain all frequencies. This forms an argument that they indeed should create general input-output relations.

An interesting aspect of generating training data only randomly, as I have in this case, is that the resulting data does *not* precisely replicate an actual use-case. Even though considerations such as discrete steps and input-blocking in the input signals are both accurate, for them to vary *randomly* is not. As such, the input-profile generated from an MPC-scheme is likely to differ greatly from a truly random input-profile. In reality, an actual control signal would instead steer the system in some intended direction - toward the reference value. This forms a valid argument that the datasets employed in training the ANN in relation to this project are non-exhaustive with respect to actual use-cases. Furthermore, if the datasets are not representative of actual use-cases, then any performance seen in the system approximation by the ANN may not translate to actual use-cases either.

Had the training dataset's data-point been non-general - not drawn from a data-generating process in an i.i.d. manner - then the optimization during training would lower the training error, but validation error would increase, as the input-output relations the ANN would learn would not apply for new, necessarily differently distributed. However, given the argumentation in Section 3.3.2, I have reason to believe that, even though not realistic, the training dataset should follow the same probability distributions as the validation dataset, in which case the training dataset should be

general. This is further supported by the fact that some - though not all, see below - sets of hyperparameters indeed allow the ANN to generalize after training on the very same training dataset. This should not be the case for any ANN based on any hyperparameter set, had the very same training dataset indeed been non-general.

Investigating other input-profiles and observing whether the same issues occur and what effect is seen with respect to performance, would nevertheless be interesting as future work. Other input-profiles may be for example a PRBS [18], or a variety of the input-profiles in Figure 10, where one input may perform slow steps, while the other varies around some given values. After showing that some input-profiles excite the system in such a way that the input-output relations obtained are descriptive of the system dynamics in a general manner, additional interesting future work would be to create extremely large datasets, and observe the effects on system performance. Of course this also requires the set of hyperparameters to provide the ANN with an adequate capacity, capable of better generalizing to the true input-output relations of the system than the ANN developed in this project is.

As mentioned in Section 4.1, for some sets of hyperparameters the validation error rose from the start, even though the training error decreased for every iteration. This indicates that the ANN is *not* able to generalize for those specific sets of hyperparameters. Since the datasets are already argued to be general, a possible explanation may be that the ANN may fail to generalize for specific sets of hyperparameters if they define a structure of the ANN that makes its capabilities of regression not able to utilize the information available. As illustrated in Section 2.2.2, how a machine learning model performs regression will depend on the capacity resulting from a specific set of hyperparameters. Furthermore, as argued above, the structure may limit crucial pieces of information about previous inputs and outputs, giving the ANN insufficient means of accurately predicting future outputs.

6 Conclusion

This project has laid the foundation to build a full RNN-MPC-scheme by implementing an MPC-scheme and an ANN separately. In order to further the work from this project to build a full RNN-MPC-scheme, some elements must be considered.

The MPC-scheme models the system dynamics by means of a linear step response model. Consequently, the MPC problem formulation presented in this project must be altered in order to allow the system dynamics prediction to be replaced by an ANN predicting the input-output relations of the system. One such MPC formulation is derived and presented.

Furthermore, the ANN implemented in this project is of the MLP-architecture, and not in itself suited to predict sequences of input-output relations for any system. However, expanding on the MLP-architecture by feeding the output back to the inputs, thus implementing an RNN, is a simple way in which to make the ANN implemented in this project compatible with a suitable MPC formulation.

Some concluding notes on the MPC- and ANN-implementations in this project are now presented.

The LSR-MPC has some performance issues, but successfully controls at least oil rate to the reference. This is in spite of potentially large modelling inaccuracies resulting in the system's input values existing within a range that is far from the point around which the system was linearized. The system is not able to control gas rate and oil rate separately once steady state has been reached. This project proposes that this is caused by the interconnected dynamics of the two rates, stemming from them together forming the same fluid. Alternatively, there might be implementational errors at the core, which must be investigated in future work.

A proposed way of improving control is to further investigate the specific input-output relations of the single well system, and potentially expand the system to include some measure of pressure, for example the well head pressure. This is proposed as a way of providing more informtion to the MPC-scheme, such that control actions may be more directed. Switching between linear step response models appropriate for different working ranges of the inputs might also be viable, and should be investigated.

A suggested piece of future work is also to implement an automatic test procedure, testing several configurations of tuning parameters for the MPC-scheme and determining their performance by means of an adequate metric. This could provide increased clarity as to the specific contributions of each tuning variable.

In total, there is much future work that may be done on the implementation of the MIMO LSR-MPC in this project. Such work is suggested as future work, due to time restrictions.

The ANN is able to approximate the trends in both fast and slow dynamics, but obtains a constant error after periods of high-valued actuation. This implies that additive effects from long-lasting input-output relations are not taken into consideration. This further implies that the Markov property does not hold for the single well system, and that future work should investigate the effect on performance in increasing the values of m_u and m_y to provide the ANN with information relevant to predicting the effects of long-lasting input-output relations. It would also be interesting to further investigate potential performance benefits from choosing specific values of m_u and m_y to each input and output, respectively.

This project proposes that the goal of the input-profiles, exciting the system in a general fashion, is met by the "staircase" input-profiles generated for this project. However, though considering some realistic factors such as discrete steps in control and input-blocking, the way in which they change is unsystematic. Future work could investigate using different input-profiles, such as ones more inspired by input-profiles seen in application in industry, or for example the PRBS. The effect of increasing the amounts of training data should also be looked into; the potential data from which to train is in theory unlimited, since it stems from simulations.

Bibliography

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

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

[3] Hesam Hassanpour, Brandon Corbett and Prashant Mhaskar. 'Artificial Neural Network-Based Model Predictive Control Using Correlated Data'. In: *Industrial & Engineering Chemistry Research* 61.8 (2022), pp. 3075–3090. DOI: 10.1021/acs.iecr.1c04339. eprint: https://doi.org/10.1021/acs.iecr.1c04339. URL: https://doi.org/10.1021/acs.iecr.1c04339.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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. 138–143. DOI: 10.1016/j.ifacol.2015.11.073.

[10] 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.

[11] Sergio Lucia and Felix Fiedler. *Model predictive control python toolbox*. URL: https://www.do-mpc.com/en/latest/. (accessed: 04.12.2022).

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

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

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

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

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

[17] 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.

[18] Alev Topuzoğlu and Arne Winterhof. 'PSEUDORANDOM SEQUENCES'. In: *Topics in Geometry, Coding Theory and Cryptography*. Ed. by Arnaldo Garcia and Henning Stichtenoth. Dordrecht: Springer Netherlands, 2007, pp. 135–166. ISBN: 978-1-4020-5334-4. DOI: 10.1007/1-4020-5334-4_4. URL: https://doi.org/10.1007/1-4020-5334-4_4.

Appendix

A Compiling a 64-bit FMU for Python

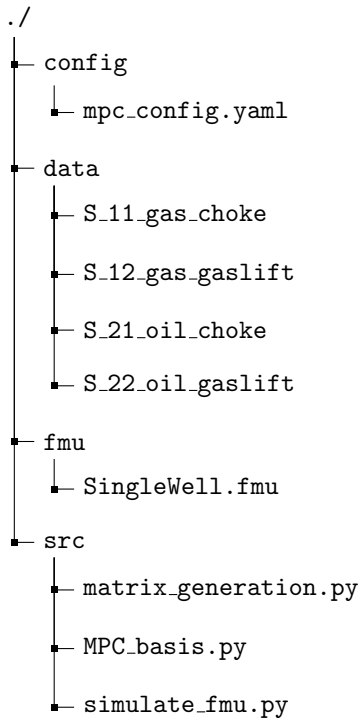
In order to compile Modelica-code to an FMU that will interface with a 64-bit Python environment, some Python terminal with access to a library capable of compiling Modelica-code must be used. One solution, as found and utilized in relation to this project, specific to 64-bit Windows 10, is as follows:

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

```
# 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})
```

B File hierarchy for the MPC-scheme



C File hierarchy tree for the ANN-code

```
./
├── main.py
├── config
│   ├── generate_data.yaml
│   └── hyperparameters.yaml
├── fmu
│   └── SingleWell.fmu
├── generate_data
│   ├── __init__.py
│   ├── generate_data.py
│   ├── load_stepresponse_data.py
│   └── data
│       ├── input_profile_0.csv
│       ├── input_profile_1.csv
│       ├── ...
│       └── input_profile_n.csv
├── models
│   └── model.pt
└── src
    ├── __init__.py
    └── neuralnetwork.py
```

D List of acronyms

Acronym	Meaning
MPC	Model Predictive Control / Model Predictive Controller
LSR-MPC	Linear Step Response-based Model Predictive Control/ Linear Step Response-based Model Predictive Controller
ANN	Artificial Neural Network
MLP	Multilayer Perceptron
RNN	Recurrent Neural Network
NN-MPC	artificial Neural Network-based Model Predictive Control/ artificial Neural Network-based Model Predictive Controller
RNN-MPC	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
SGD	Stochastic Gradient Descent
NARX	Nonlinear Autoregressive model with Exogenous inputs
FMU	Functional Mock-up Unit