TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

Modellierung und adaptive Regelung von industriellen Altholz- und Reststoffkesseln mit neuronalen Netzen

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Physikalische Energie- und Messtechnik

eingereicht von

Sebastian Voith
Matrikelnummer 11902027

ausgeführt am Institut für Festkörperphysik
der Fakultät für Physik der Technischen Universität Wien
in Zusammenarbeit mit CONENGA Group

Betreuerin: Associate Prof. Dr.in rer.nat. Sabine Andergassen
Externer Betreuer: Dipl.-Ing. Sebastian Sturm

Wien, 11.03.2025

[redacted]
(Unterschrift Verfasser)

[redacted]
(Unterschrift Betreuerin)

Abstract

Many real world processes are highly complex and not easily described by fundamental physical laws. This, along with the rising availability of data describing these processes, results in the modeling of these systems being increasingly performed using data-driven machine learning techniques. Additionally, the rise of computational processing power enables the use of larger datasets and more complex model architectures, leading to more powerful models in shorter training durations.

In the process industry, these system models can be used in many different ways in order to raise efficiency, reduce undesirable process dynamics, increase process outputs, and decrease pollution. Possible applications include advanced control techniques such as model-based or adaptive control, predictive maintenance, process optimization, and fault detection.

In this thesis, artificial neural networks are built to model industrial biomass combustion boilers using historical process data from two power plants. Specifically, two modeling approaches are tested, utilizing a static feedforward neural network on the one hand and a dynamic recurrent neural network on the other. Subsequently, the respective model performances are evaluated and compared to each other. After the evaluation of several model iterations the optimal model configuration is determined. Finally, different ways to further increase the modeling accuracy are discussed and implementation strategies into a conventional control setup are reviewed.

Contents

1 Introduction

In recent years, artificial intelligence in general and its subfield of machine learning in particular have experienced a substantial rise in popularity, caused by ever increasing advancements in computing power, machine learning techniques and algorithms, and software architecture [45]. This boom has also reached the process and control industry, where the integration of machine learning applications promises a higher level of efficiency and productivity. In this field, machine learning techniques can be used in many different ways, such as condition or predictive monitoring, quality prediction, fault detection, and process control and optimization [45, 48].
Typically, these applications require a system model of the respective industrial process. Conventionally, these models are devised as first principle, empirical, or mechanistic models, which require a deep knowledge of the process or a close analysis and identification of its dynamics. Unfortunately, these processes are often highly complex, due to their nonlinear dynamics and dependency on many, possibly unknown, state variables, which makes it difficult to explicitly formulate an accurate model. Even when a model is found, they are frequently too complex to be used online and have a high development cost [48].
Using machine learning, however, an explicit formulation of the model is not required, as the model implicitly learns the system dynamics from historical system data. This allows processes which are too complex or have insufficient information about the underlying phenomenon to be modeled nonetheless, given enough historical data is available. The rising trend of digitalization in the process industry, leading to a growing amount of data, and the increase of processing power further drive the growing use of machine learning applications in the industry [39].

In the field of machine learning, artificial neural networks have become the state-of-the-art algorithmic architecture in the past years. The earliest representations, so-called perceptrons, were introduced in 1958, but their development stagnated due to the low processing power of that time and inefficient training algorithms. Their rise in popularity started when, in 1986, the significantly improved backpropagation algorithm was developed, and accelerated when, in the early 2000s, computational hardware became sufficiently powerful [44]. Since then, artificial neural networks of various forms have been applied successfully to many different problems, ranging from image, speech, and natural language recognition to sequence prediction and forecasting [35].

In this thesis, the use of machine learning for the modeling of industrial biomass combustion plants is investigated. The aim of this thesis is to find well performing and accurate plant models, selected from different neural network architectures, which can be used for model-based process control in the future.
To this end, artificial neural network models are constructed and, using historical process data from two industrial waste wood boilers, trained to simulate the plant behavior. Specif-

ically, two different modeling approaches and corresponding neural network architectures are compared and evaluated. On the one hand, a static model is built using a feedforward neural network to predict the plant outputs. On the other hand, a dynamic model is constructed which incorporates past plant states using a recurrent neural network for the prediction. Subsequently, these models are contrasted and their performance and accuracy analyzed. Furthermore, the optimal model configuration is examined, featuring a low computational expense and a high accuracy. Moreover, the ability of recurrent neural networks to act as multi-step-ahead predictors is investigated. Finally, possible ways of integrating these models into the existing control setup and the corresponding challenges are reviewed.

This work was created in cooperation with CONENGA Group, who kindly provided the process data and computational hardware used for model training. Additionally, the team of CONENGA Group, first among them Dipl.-Ing. Sebastian Sturm who acted as my external supervisor, supported me with their extensive know-how about process engineering and process control.

This thesis is organized into four main chapters. In chapter 2, I introduce the theoretical background that the following work is based on. Here, the focus lies on the aspect of machine learning and neural networks (2.1), but fundamentals of the combustion process and plant control (2.2) and advanced model-based control schemes (2.3) are covered as well. Chapter 3 gives an overview of the software resources that were used during the work and reviews the methods that were employed to train the plant models. In chapter 4, I present and discuss the results, comparing the static neural network models to the dynamic models of both plants. Additionally, the use of the dynamic models for multi-step-ahead forecasts is discussed. Finally, chapter 5 concludes the work and presents an outlook for future work.

2 Theoretical Background

In this chapter, I give a short overview of the theory the subsequent work is based on. First, in section 2.1, I describe the fundamentals of machine learning and neural networks. Here, especially the different gradient descent variants and the backpropagation algorithm are examined in detail. In section 2.2, I discuss the process control fundamentals, and outline the combustion process in biomass furnaces and the corresponding control strategy. Finally, section 2.3 gives a short overview of model-based control and possible types of system modeling.

2.1 Machine Learning and Neural Networks

2.1.1 Machine Learning Basics

Machine learning is a subfield of artificial intelligence with the aim of extracting valuable information from data. This information can subsequently be used to design a model that makes predictions according to the underlying principles of the dataset. To this end, an algorithm needs to learn from the provided data, which means that the algorithm automatically finds patterns or structure in the dataset and tweaks the model to more closely follow this structure [38]. In other words: A model is understood to learn from data if its performance on a desired task improves after the data has been taken into account [3].
The machine learning model fundamentally is a data-driven or black box model, which means that the entire information is solely extracted from the dataset. No prior or external knowledge is needed or provided. This means that phenomena can be modeled without understanding the underlying physical or mathematical principles [38]. This is the defining difference to the extreme opposite of data-driven models: The so-called first principle models are process models that are constructed from the fundamental laws of physics, which of course requires total understanding of the process and its parameters [41].
Two factors drive the increasing use of machine learning models: First, many real world processes are highly complex and not easily described by fundamental physical laws. Second, large datasets are increasingly easy to obtain and, with advances in computational power, to process. Higher computational power also facilitates the use of increasingly complex and powerful models, such as deep neural networks, which, in the past, was a highly limiting factor.

Machine Learning Algorithm

The individual data points of a dataset are called examples and are denoted by a vector $\boldsymbol{x}_i$. Each example consists of one or more features, which describe separate measurements of the data point and are represented as the vector elements x_j. The full dataset of all examples can thus be expressed in the so-called design matrix $\boldsymbol{X} \in \mathbb{R}^{m \times n}$, where m is the number of examples and n is the number of features per example [44].

The full dataset is commonly split into a training set, whose data points are used for model training (i.e. parameter optimization), and a test set, which is used for the performance estimation of the model. In particular, the test set is used to evaluate how the model performs on unseen data, meaning data that has not been used for training. The exact percentage of the split can vary and depends on the type of model and on the size of the original data set. Commonly, the training set amounts to $80\,\%$ to $90\,\%$ of the data points, but can be set as low as $50\,\%$ [44].

The different types of machine learning algorithms can mostly be divided into supervised and unsupervised learning algorithms. Supervised learning is probably the most common type of learning and implies that each data point is labeled. When training a model to classify images of pets, for instance, each picture needs to be labeled beforehand ("cat", "dog", etc.). These labels are also called targets and are combined into the vector $\boldsymbol{y}$, where each entry y_i is the label of the i-th example or data point.
Unsupervised learning uses unlabeled data, and instead is used to find structure in the dataset. Such a model could, for example, be used for clustering, where the data points are divided into groups of closest similarities. Alternatively, unsupervised learning can be used to find the probability distribution that generated the dataset in question [29].
Besides these two big categories of machine learning algorithms, other variants exist. Semi-supervised learning, for example, is a mixture of the two types described above, where only a part of the dataset is labeled. In reinforcement learning, the algorithm interacts with the environment and learns to make certain decisions in order to maximize a reward. Due to the feedback loop between the learning system and its actions, the algorithm is primarily used for sequential decision-making tasks where it is required to reach long term goals [44]. The most famous example of reinforcement learning was performed by Google researchers, where a reinforcement algorithm learned the complex game Go by only playing against itself, ultimately resulting in a win against the European Go champion [33].

As seen from the examples for different machine learning algorithms provided above, a machine learning model can be optimized for one of many different tasks. Regression, for instance, is a supervised learning problem where the model has to predict a numerical value given some input. Essentially, the model is a function $f : \mathbb{R}^n \to \mathbb{R}$, and the learning algorithm has to adapt the function parameters to have the outputs match the labels y_i for the respective input $\boldsymbol{x}_i$. An example for a regression model could be a model that calculates the price of a house based on the number of rooms, its location, and its age. Depending on its architecture, the model can perform linear, polynomial or a different nonlinear regression, and can predict one or multiple outputs.
The second big task is classification, where the model sorts labeled inputs into one of k categories. For this task the mapping looks like $f : \mathbb{R}^n \to \{1, ..., k\}$. Alternatively, the classification model outputs a probability distribution over several classes, instead of one discrete class. Examples for classification include object recognition or image identification.

In spite of the name, logistic regression is a type of classification where a binary output is generated based on the logistic function [44].
Clustering is an unsupervised learning task, where examples are grouped according to shared similarities. The k-means algorithm is a frequent choice for clustering problems, where input data samples are sorted into k different clusters [44].
Other machine learning tasks include transcription, where an image of text is transcribed into digital text or human speech is transcribed, anomaly detection, where atypical events in an object stream are detected or flagged, and synthesis and sampling, where the model is used to generate new data that is similar to the input data [29].

Performance Measure

In order to make a statement about the capability of the model, a metric that quantitatively measures the model performance is needed. Depending on the task, this metric is calculated in different ways. For classification tasks, oftentimes the accuracy of the model is evaluated which, in this case, is the portion of the examples which are classified correctly. Alternatively, the error rate of the model can be used as a performance measure for classification. For other tasks it is harder to chose a fitting performance metric. When evaluating a transcription model, for example, that performs a speech-to-text task, quantifying its correctness or, alternatively, the numeric cost of one mistake in a sentence is difficult [29].
For regression models the standard way of evaluating performance is by comparing the predicted output $\hat{y}_i$ to the true label y_i of a data point $\boldsymbol{x}_i$. This comparison function is also called the loss function $\mathcal{L}(y_i, \hat{y}_i)$ and can be evaluated in different ways. Mostly, the loss function is set to be the mean squared error (MSE), but the mean absolute error (MAE) or the mean absolute percentage error (MAPE) are also often used:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \ , \tag{1}$$

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \ , \tag{2}$$

$$\text{MAPE} = \frac{100}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right| . \tag{3}$$

Here, N is the number of data points, $\hat{y}_i$ is the predicted label and y_i is the true label [38]. When designing a machine learning model, it has to be considered that the different metrics yield a different loss for the same data, which affects the way the model learns from the data. Compared to the MAE, for instance, the MSE results in higher loss for larger mistakes while small mistakes have less influence. Ideally, the chosen loss function fits the required machine learning task and the desired model capability well.

Optimization Techniques

With the machine learning task defined and the performance metric selected, the next step

is to let the model learn from the provided data. The fundamental goal of the machine learning process can be summarized as follows: The model characteristics are adjusted in such a way that model performance is maximized for the input data, which most of the time means minimizing a loss. This optimization task can be expressed as

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{X}; \boldsymbol{\theta})) \,, \tag{4}$$

with $f(\cdot\,; \boldsymbol{\theta})$ representing the model and its parameters $\boldsymbol{\theta}$, $\boldsymbol{X}$ being the input data with corresponding labels $\boldsymbol{y}$, and $\mathcal{L}(\cdot\,, \cdot)$ being the loss function. Essentially, during model training the machine learning algorithm finds the optimal model parameters $\boldsymbol{\theta}^*$ for the selected performance metric:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{X}; \boldsymbol{\theta})) \,. \tag{5}$$

After training, when the optimal parameters have been found, the model can be used in the real world with exactly this configuration [35].
Taking linear regression as an example, the model would look like the following:

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{w}^\top \boldsymbol{x} + b = \hat{y} \,, \tag{6}$$

where $\boldsymbol{w}$ is a vector of model weights and b is the model bias. Choosing the MSE as the loss function, the model performance can be written as

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = \frac{1}{N} \sum_{i=1}^{N} \Big(y_i - (\boldsymbol{w}^\top \boldsymbol{x}_i + b) \Big)^2 \,. \tag{7}$$

In order to minimize the loss with respect to the parameters, the gradient is set to zero and then needs to be solved for the parameters $\boldsymbol{w}$ and b:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L} = \begin{pmatrix} \dfrac{\partial \mathcal{L}}{\partial \boldsymbol{w}} \\ \dfrac{\partial \mathcal{L}}{\partial b} \end{pmatrix} = 0 \,. \tag{8}$$

This equation, however, is hardly ever analytically solvable, due to a highly nonlinear dependence on the weights and bias [8]. Instead, an iterative numerical algorithm is required to compute the values of $\boldsymbol{w}^*$ and b^*.
This algorithm is called gradient descent and works the following way: First, the gradient of the loss function with the current, not yet optimized parameters is calculated. Now, the fact that the gradient points to the steepest ascent of the function is used. The parameters

are corrected proportionally to the negative gradient:

$$\begin{aligned} \boldsymbol{w} &\leftarrow \boldsymbol{w} - \eta \frac{\boldsymbol{\partial \mathcal{L}}}{\partial \boldsymbol{w}} \,, \\ b &\leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b} \,. \end{aligned} \tag{9}$$

This way, the parameters take a step towards the minimum of the loss function. The size of the step can be adjusted by the learning rate η. With the value of the parameters updated, the next iteration step is performed, by calculating the new gradient [38, 44]. When the gradient is zero, the local minimum has been reached and the optimization is completed. Figure 1 depicts a gradient descent on a surface parametrized by x and y. Since the gradient descent only finds a local minimum, the final position depends on the initial configuration.

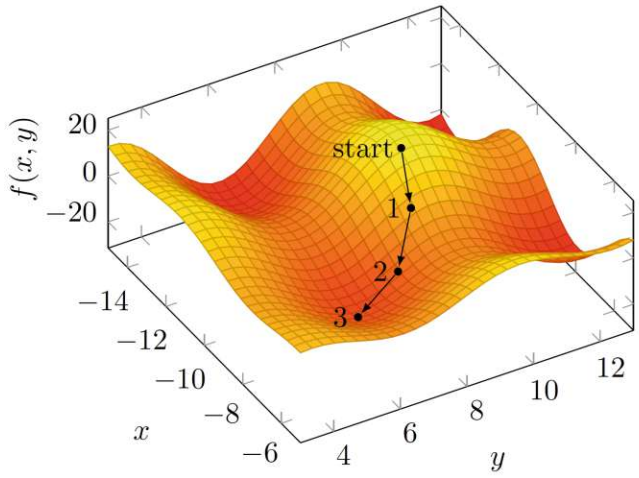


Figure 1: Gradient descent on a surface for the parameters x and y. For every iteration a step is taken toward the steepest descent until a local minimum of the function $f(x, y)$ is reached. Figure adapted from [44].

For a generic model with parameters $\boldsymbol{\theta}$ the gradient descent algorithm can be expressed as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) \,, \tag{10}$$

for the t-th iteration step. Importantly, the gradient descent is very sensitive to the value of the learning rate η. If η is too small the algorithm has to make many iterations, which increases the computation time and training duration, and if η is too large the local minimum could be overshot and the algorithm becomes unstable. Additionally, due to the fact that the loss is calculated as a sum over the whole dataset (see Eq. (1)), computing the gradient for datasets with many data points becomes computationally very expensive. These two problems can be improved by two modifications of the conventional gradient descent algorithm. First, adding a so-called momentum to the equation helps stabilizing the descent in high curvature regions while increasing the descent in shallower regions.

The momentum is implemented in Eq. (10) as follows:

$$\begin{aligned} \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) \,, \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t \,. \end{aligned} \tag{11}$$

Here, $\gamma \boldsymbol{v}_{t-1}$ is the momentum or inertia term that incorporates a moving average of previous gradients ($\boldsymbol{v}_{t-1}$) and scales them according to the momentum parameter $0 \leq \gamma \leq 1$ [35]. Second, in order to reduce the computation time for the gradient descent, the stochastic gradient descent (SGD) algorithm is used. Instead of computing the gradient for all N data points in the dataset, it is calculated for a subset of the data called a minibatch. Essentially, this means replacing the actual gradient over all N examples with an approximated gradient calculated from the minibatch. A minibatch B_k usually contains only ten to a few hundred examples, which are uniformly drawn from the training set. If the minibatch size is set to M, the entire dataset can be resampled into $n = \frac{N}{M}$ batches. Now, for each stochastic gradient descent step the algorithm only calculates the gradient of a minibatch (B_k), adjusts the parameters and then takes the next minibatch (B_{k+1}) for the next step:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}^{B_k}(\boldsymbol{\theta}_t) \,, \tag{12}$$

for $k \rightarrow k+1$. When all n minibatches have been cycled through, the training algorithm has processed one epoch. If the minimum has been found, the training stops, otherwise the next training epoch starts. SGD has the major advantage of being faster and computationally cheaper due to the smaller batch size [35]. Additionally, the stochastic gradient descent can be used in combination with a momentum term, which has made the algorithm one of the most widely used algorithms for machine learning. SGD is particularly common in deep learning where, because of the use of multilayered neural networks, a fast learning algorithm is a necessity [29].

In recent years, adaptive learning rate algorithms have been developed which demonstrated a successful increase in learning performance. These algorithms automatically adapt the learning rate during training and thus reduce the impact of the initial choice of the learning rate. Out of these, the most successful optimization methods, especially for deep learning, are RMSProp [21] and Adam [32], standing for "adaptive moments", which can be seen as a variant of RMSProp with added momentum [29]. Since all learning algorithms have advantages and weaknesses in certain tasks, no single method can be declared the best in all circumstances. However, in a comparison of many optimization algorithms by Schaul et al. [23] algorithms with adaptive learning such as RMSProp were robustly shown to perform very well.

Overfitting versus Underfitting

The goal of a machine learning model, once trained, is to perform well on new, unseen data. The generalization of a model describes its ability to provide good predictions for

previously unseen data points. When predictions for unseen data points are accurate, the model generalizes well. The error of the model for new input is called generalization or test error, and can be estimated with the test set which is separated from the training set before training. Generally, a low error during training also results in a low test error, due to the statistical learning theory assumptions of the two sets being independent from each other and identically distributed [29]. However, usually the test error is slightly higher than the training error, because of one of the two following reasons: Either the model is underfitting, which means that a model with low capacity, having too few features or parameters, is applied to a highly nonlinear problem. Underfitting occurs for instance, when a linear regression model is used to predict higher order polynomials. In this case, the model has too few parameters to make good predictions, which also results in a high training error. Or the model could be overfitting, where the training error is low, but the model generalizes poorly. This happens when the learning algorithm fits the training data perfectly, including the noise, which can occur when the model capacity is too high for the problem. In this case the model overestimates the importance of the noise of the data.
Figure 2 depicts the effects of underfitting (left) and overfitting (right) compared to an ideal fit (center). As can be seen by the ideal fit, the data seems to be distributed according to a quadratic function and some additional noise. The model on the left, however, is a linear model and thus is a bad fit for both training data and test data. In the overfitting case, the model is at least a fourth degree polynomial model, which fits the training data perfectly, but didn't learn the actual underlying data generating distribution (quadratic), and thus performs poorly with the unseen test data.

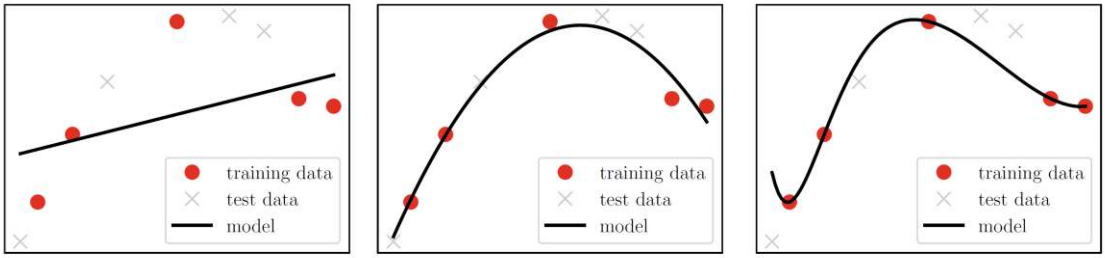


Figure 2: Depiction of the effects of underfitting (left), ideal fitting (center), and overfitting (right) for a one-dimensional regression. Figure adapted from [44].

Whether a model is underfitting or overfitting is primarily caused by the capacity of the model, which can be thought of as its ability to fit a wide variety of functions, and generally is expressed by its number of features and parameters [29]. This relationship is displayed in Fig. 3, where the test (generalization) error is lowest at optimal model capacity.

Because the optimal model capacity for a given problem is usually unknown, one way of reducing the risk of overfitting is using regularization strategies. Generally, regularization is any adjustment to a machine learning algorithm that intends to reduce the generalization

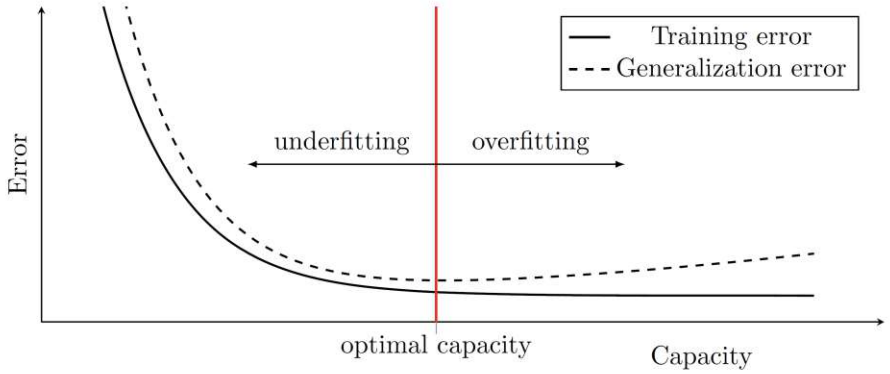


Figure 3: Typical trend for training error and generalization error for a model with increasing capacity. At optimal capacity the generalization error is lowest. Figure adapted from [44].

error but not the training error [29]. Using regularization, overfitting can be reduced by deploying a high capacity model but penalizing the modeling of certain functions. For example, linear regression can be regularized by using weight decay, where the training loss is extended by the squared L^2-norm of the weights:

$$\tilde{\mathcal{L}}(\boldsymbol{w}, b) = \mathcal{L}(\boldsymbol{w}, b) + \lambda \boldsymbol{w}^\top \boldsymbol{w} \,. \tag{13}$$

This forces the algorithm to choose smaller weights during training, which indirectly reduces the model capacity and thus overfitting [44].
Another way to regularize the machine learning algorithm is using the early stopping method. When training a model, the training error steadily decreases for every epoch, i.e. an iteration over the entire training set. However, after a certain number of epochs also depending on the capacity, the model starts to overfit. This effect can be seen by splitting off a third dataset from the training dataset before training. The so-called validation set is not used for model training per se, but instead is used to validate the model during training, usually by evaluating the loss after every epoch. As the training progresses, the validation loss is decreasing at first, concurrently to the training loss. However, while the training error decreases further, at one point the validation error starts to increase again, as can be seen in Fig. 4. This is precisely the moment where the model starts to overfit. Thus, the early stopping algorithm evaluates the validation error at every epoch and stops the training when the error is increasing again. Due to its effectiveness and simplicity, early stopping is probably the most widely used regularization strategy in deep learning [29].

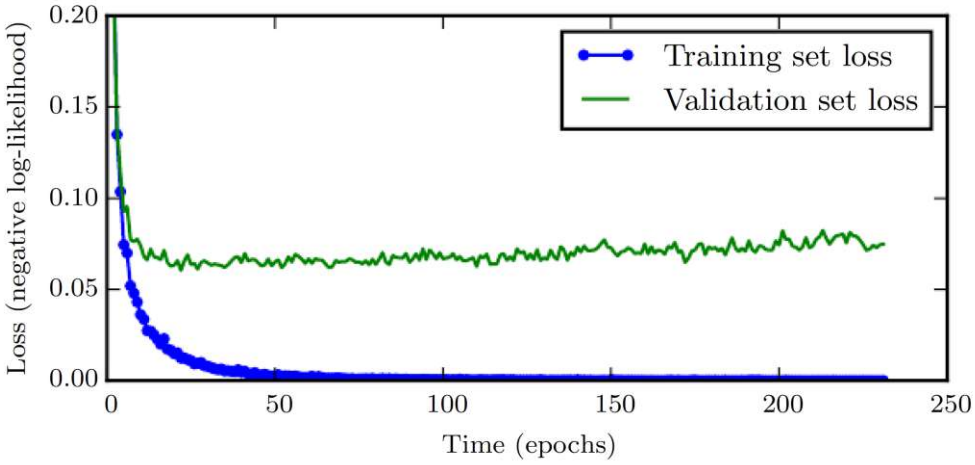


Figure 4: Typical training and validation error trend. While the training loss continuously decreases, the validation loss has a minimum at approximately epoch 25 and then slowly rises again. Figure adapted from [29].

2.1.2 Feedforward Neural Networks

An artificial neural network (ANN) is a special type of supervised machine learning model architecture loosely based on the interconnected neurons of the brain. Feedforward neural networks (FNNs) are a type of ANN where information flows uni-directionally from input to output, meaning there is no feedback flow (in contrast to recurrent neural networks, which are discussed in section 2.1.3). By far the most common FNN is the multilayer perceptron (MLP), which is why the terms FNN and MLP often are used interchangeably, although other types of FNN exist, such as radial basis function networks [8].
The building block of a MLP neural net is a basic neuron, which consists of an input $\boldsymbol{x}$, parameters subdivided into weights $\boldsymbol{w}$ and bias b, and a nonlinear function $\sigma(\cdot)$ which is also called activation function. Figure 5 depicts the structure of a single neuron for a three-dimensional input. The output $a(\boldsymbol{x})$ of a single neuron can be expressed as

$$a(\boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b) = \sigma(z)\,. \tag{14}$$

The nonlinear activation function is an integral part of the neuron, as it allows neural nets to model nonlinearities. Figure 6 shows a few common activation functions. Historically, mainly the step function (perceptron), sigmoid and hyperbolic tangent have been used as activation function. The perceptron or step function is the reason why feedforward neural networks were and still are called multilayer perceptron, despite the perceptron function seldom being used anymore. More recently, a shift towards rectified linear units (ReLUs) and its variants has occurred [35]. This is due to the derivative of the activation function being important for the model training based on gradient descent. When the derivatives of the functions in the top row of Fig. 6, which saturate for $z \gg 0$, are evaluated, they

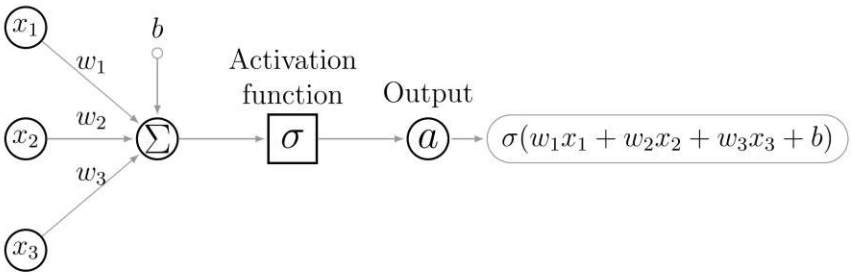


Figure 5: Structure of a single neuron with a three-dimensional input. Figure adapted from [44].

approach zero for large weights $\boldsymbol{w}$. This effect is called vanishing gradients and hinders model training, which is why non-saturating functions such as ReLU are preferred for deep learning [35].

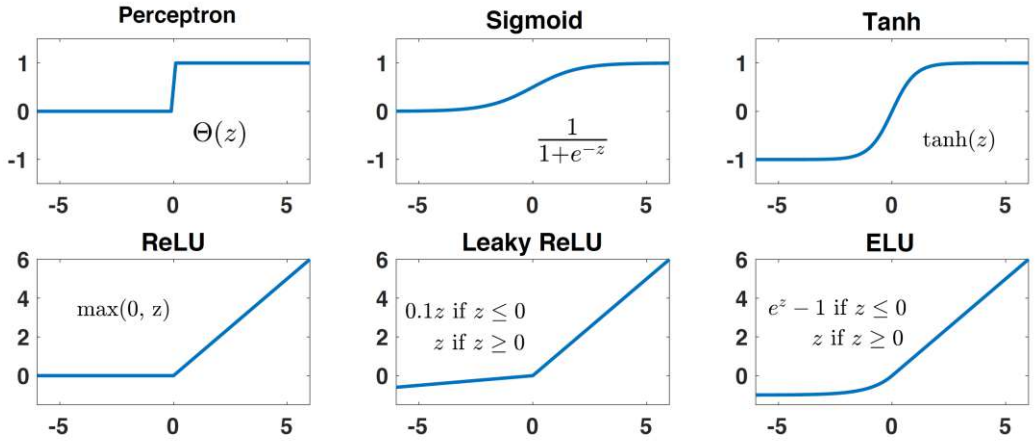


Figure 6: Different activation functions for neurons. Figure adapted from [35].

A neural network is made up of many neurons in a layered structure, where the outputs of one layer are connected to the next layer's inputs. Generally, a feedforward neural network has one input and output layer between which lies an arbitrary number of hidden layers. Additionally, each hidden layer can be arbitrarily wide. Usually, when a FNN has multiple hidden layers it is called a deep neural network. Figure 7 displays a typical FNN with two hidden layers. The numbers of neurons in the input and output layer correspond to the dimension of the input data point and the label, respectively.

The use of hidden layers gives the neural network a much greater ability to model nonlinear functions, when compared to only having an input and output layer. This is famously described by the universal approximation theorem, which states that a neural network with one hidden layer of sufficient width can approximate any continuous multi-input/multi-output function with arbitrary accuracy [35].

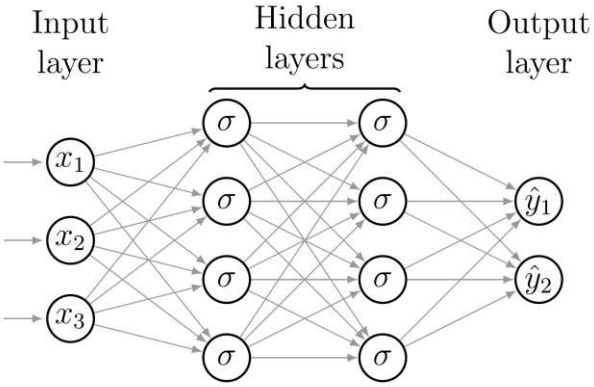


Figure 7: Typical fully connected feedforward neural network with two hidden layers. Figure adapted from [44].

Examining the entire neural net and expanding Eq. (14), the output of the j-th neuron in the l-th layer is calculated as

$$a_j^{(l)} = \sigma(z_j^{(l)}) = \sigma\left(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right) , \tag{15}$$

where $a_k^{(l-1)}$ is the output of the k-th neuron in the previous ($l-1$-th) layer, and $w_{jk}^{(l)}$ is the corresponding neuron weight. In vector notation, where $\boldsymbol{a}^{(l)}$ represents the vector of all outputs in the l-th layer, this equation reads as

$$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{z}^{(l)}) = \sigma\left(\boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\right) . \tag{16}$$

Here, the activation function $\sigma(\cdot)$ is evaluated element-wise, and $\boldsymbol{W}^{(l)}$ is the weight matrix for the l-th layer [44].

Backpropagation Algorithm

The training of neural networks is based on the same gradient descent methods discussed in section 2.1.1. However, due to the number of individual parameters of the model, the gradient cannot be calculated directly. Instead, a different, more efficient algorithm, called backpropagation, is used to exploit the layered structure of the neural nets. The aim of the backpropagation algorithm is to calculate the partial derivatives of the loss function with respect to the weights and biases $\left(\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}}, \frac{\partial \mathcal{L}}{\partial b_j^{(l)}}\right)$, which can then be used for gradient optimization, as in Eq. (10). Recalling Eq. (15), $\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}}$ can be written as

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} . \tag{17}$$

Similarly, $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}}$ is calculated as

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \,. \tag{18}$$

Both Eq. (17) and Eq. (18) use the variable $\delta_k^{(l)}$ which is defined as the sensitivity of the loss toward a change in the neuron's weighted input $z_j^{(l)}$. This means, in order to calculate the desired partial derivative for the weight, this sensitivity simply needs to be multiplied by the output $a_k^{(l-1)}$ of the previous layer's k-th neuron, which is calculated according to Eq. (15). The derivative for the bias directly computes as $\delta_j^{(l)}$. Thus, all that is needed is the value of $\delta_j^{(l)}$. Using the chain rule and Eq. (15) yields [44]

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \sigma'(z_j^{(l)}) \,. \tag{19}$$

Due to the sensitivity $\delta_j^{(l)}$ depending on $\delta_k^{(l+1)}$, which is the sensitivity in the subsequent layer, the algorithm needs to start the calculation at the last (output) layer and propagates backward to the first (input) layer [44]. At the final layer L the sensitivity can be calculated directly from the loss function, using $a_j^{(L)} = \hat{y}_j$:

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial z_j^{(L)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}) = \frac{\partial \mathcal{L}(y_j, \hat{y}_j)}{\partial \hat{y}_j} \sigma'(z_j^{(L)}) \,. \tag{20}$$

To summarize, the steps needed to perform the backpropagation algorithm are the following: First, a training data point $\boldsymbol{x}$ is used as an input for the neural net and is processed in forward direction through the network, storing the values of $z_j^{(l)}$ and $a_j^{(l)}$ for later. Then, the sensitivity at the final layer $\delta_j^{(L)}$ is calculated using the loss for prediction $\hat{y}_j$ and target y_j according to Eq. (20). Now, the algorithm propagates backward through the network and calculates the sensitivities in the previous layer $\delta_j^{(l)}$ using Eq. (19). Finally, with all sensitivities computed, Eq. (17) and Eq. (18) can be used to calculate the derivatives for every parameter. The derivatives can now be used to optimize the parameters with gradient descent as in Eq. (9) or Eq. (10) [35].

2.1.3 Recurrent Neural Networks

Feedforward neural networks evaluate the output for one input state, which means that the information or state of previous samples is not considered. This is suboptimal when modeling dynamic systems, where the current state often depends on previous states. In contrast, recurrent neural networks (RNNs) can process sequential data by incorporating loops that allow the flow of information from past states. Instead of having a single (multidimensional) data point as input, RNNs are presented a sequence or time series of data points $\boldsymbol{x}_t$, with the time step index t [29].
Thus, for every time step t the state of the system $\boldsymbol{h}_t$ which is usually represented by the hidden layer of the network can be described as

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t; \boldsymbol{\theta}) = \sigma(\boldsymbol{W}_h^h \boldsymbol{h}_{t-1} + \boldsymbol{W}_i^h \boldsymbol{x}_t + \boldsymbol{b}_h) \,. \tag{21}$$

Here, the current state depends on the past state $\boldsymbol{h}_{t-1}$ and the current input $\boldsymbol{x}_t$ according to the transition function $f(\cdot; \boldsymbol{\theta})$. Usually the inputs of the transition function are transformed by weight matrices and a bias before being evaluated by an activation function, see Eq. (21). Typically, the output of a RNN is not directly its state, but instead is transformed by an additional output layer. The full structure of a recurrent neural network is depicted in Fig. 8. This type of depiction is called a recurrent graph, as it shows the recurrence of the network as a dependence on the previous state, as in Eq. (21). The output $\boldsymbol{y}_t$ of the depicted network can be expressed by the following equation:

$$\hat{\boldsymbol{y}}_t = \boldsymbol{W}_h^o \boldsymbol{h}_t = \boldsymbol{W}_h^o \sigma(\boldsymbol{W}_h^h \boldsymbol{h}_{t-1} + \boldsymbol{W}_i^h \boldsymbol{x}_t) \,, \tag{22}$$

where $\sigma(\cdot)$ is a nonlinear activation function. Alternatively, the graph can be unfolded by creating a copy for each time step, explicitly showing the way the information flows through the RNN. The unfolded graph is displayed in Fig. 9 and visualizes the advantage of the recurrent function Eq. (21): for every step the parameters, in this case the weight matrices and the activation function, are the same. Without the recurrence, every step would have to be described by a different function [29].

Depending on the structure of the input and output layers of a RNN, they are designed to produce different predictions. For instance, many RNNs are set to produce one output every time step, which means that the input sequence and the output sequence have the same length. Alternatively, RNNs can read an entire input sequence and produce one single output [29].

The training of recurrent neural networks generally works the same way as with feedforward neural nets, using gradient descent and backpropagation. Due to the recurrent structure, however, the gradient descent algorithm needs to be adjusted slightly. Before starting the algorithm outlined in the previous section, the neural net is unfolded, resulting

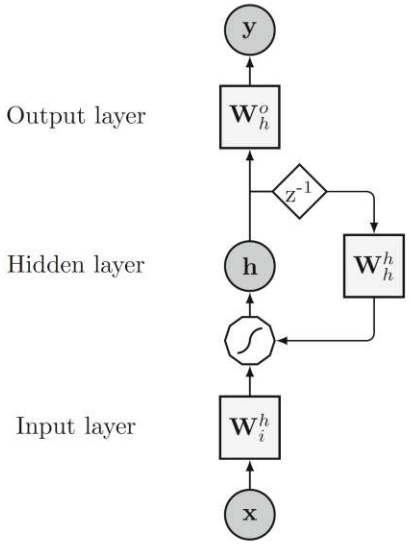


Figure 8: Structure of a simple recurrent neural net. $\boldsymbol{W}_i^h, \boldsymbol{W}_h^h$ and $\boldsymbol{W}_h^o$ are the weight matrices of the input, hidden, and output layer respectively. z^{-1} represents the delay of the hidden state by one step, which is added to the nonlinear activation function in the next step. Figure adapted from [31].

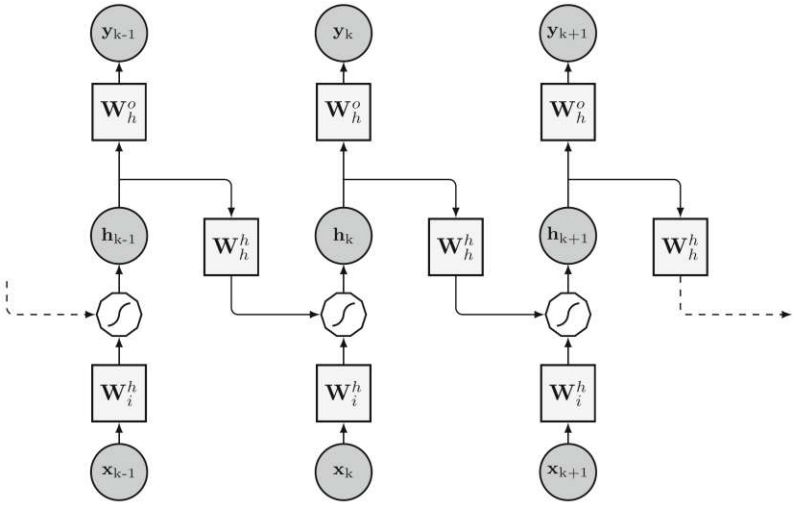


Figure 9: Unfolded structure of the recurrent neural net in Fig. 8, which makes the information flow visible. For every time step the weights are the same. Figure adapted from [31].

in an acyclic, one-directional network, comparable to a FNN. Now the backpropagation algorithm can be executed, with the added constraint that the weight matrices are required to have the same values for the different unfolded layers, since they have to be the same after refolding the graph. This procedure is called backpropagation through time, and is the most successful type of training algorithm for RNNs [31].

One challenge that arises when working with recurrent neural networks, especially deep

ones, is the problem of vanishing and exploding gradients, which refers to the exponential decrease or increase of the gradient during training [22]. In order to illustrate the problem, one can consider the gradient of a loss function $\mathcal{L}_t$ for the t-th time step with regard to the parameters $\boldsymbol{\theta}$:

$$\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{\theta}} = \sum_{1 \leq k \leq t} \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{\theta}} , \tag{23}$$

$$\text{with} \quad \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \prod_{t \geq i \geq k} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}} . \tag{24}$$

Due to the current state $\boldsymbol{h}_t$ recursively depending on the previous states the derivative has to take into account all past states, which is represented by the sum in Eq. (23). However, every summand in turn consists of a product of derivatives as stated by Eq. (24) [31, 22]. For longer sequences and, accordingly, higher values of of t, this product leads to the problem of vanishing and exploding gradients: If the terms in the product are smaller than one, the product exponentially converges to zero. For the sum in Eq. (23), this means that the summands with high k, corresponding to recent dependencies, dominate, while earlier dependencies with lower k are neglected. This is the effect of vanishing gradients. Exploding gradients occur, when the factors in Eq. (24) are bigger than one, which causes the value of the product to rise exponentially, making the less recent dependencies more impactful than more recent states [31].

As these effects limit the capacity of recurrent neural networks, different solutions were proposed to overcome the issue. Regularization, for instance, can be used to stop gradients from exploding. Alternatively, gradient clipping can be used, where gradients above a threshold are downscaled. The most popular solution, however, involves adapting the model structure by introducing gated architectures [22].

Long Short-Term Memory

The most widely used gated network is the Long Short-Term Memory (LSTM) which was first proposed by Hochreiter and Schmidhuber in 1997 [2]. The big advantage it has over other more conventional recurrent neural nets is its ability to accurately model both long- and short-term dependencies in data. A LSTM overcomes the issue of vanishing and exploding gradients by creating paths through time whose derivatives neither explode nor vanish. These so-called self loops act as a kind of dynamic memory for previous states, where information can be stored and which can be reset. Crucially, the actions of remembering and forgetting are performed automatically by the neural network, which is also learned during training [29].

A LSTM neural network consists of a hidden memory block, located between an input and

an output layer. The memory block itself contains one or more memory cells, which control the way information flows and is retained with gates. Data enters the cell via the input gate $\boldsymbol{i}_t$ and is read out with the output gate $\boldsymbol{o}_t$, while the forget gate $\boldsymbol{f}_t$ regulates which information is stored. Figure 10 depicts the general structure of a LSTM cell. For any input $\boldsymbol{x}_t$ these gates are calculated as follows:

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_x^i \boldsymbol{x}_t + \boldsymbol{W}_h^i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right) , \tag{25}$$

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_x^o \boldsymbol{x}_t + \boldsymbol{W}_h^o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right) , \tag{26}$$

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_x^f \boldsymbol{x}_t + \boldsymbol{W}_h^f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right) . \tag{27}$$

Here, $\boldsymbol{h}_{t-1}$ is the previous hidden state while $\boldsymbol{W}_k^j$ and $\boldsymbol{b}_k$ are the respective weight matrices and biases. The nonlinear activation function in this case is the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ which is evaluated element-wise and maps to $[0, 1]$ [36].
Apart from the hidden state $\boldsymbol{h}_k$ the LSTM cell has another internal state used for recurrence, the so-called memory $\boldsymbol{c}_k$. For the calculation of the memory, first the candidate memory $\tilde{\boldsymbol{c}}_t$ is calculated using the hyperbolic tangent function:

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_i^c \boldsymbol{x}_t + \boldsymbol{W}_h^c \boldsymbol{h}_{t-1} + \boldsymbol{b}_c) . \tag{28}$$

The candidate memory can be interpreted as a measure of how to update the internal memory, and depends on the current input and the previous hidden state. The memory is then computed as

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t , \tag{29}$$

with $\odot$ representing element-wise multiplication of the vectors. Thus, the new memory state $\boldsymbol{c}_t$ is the sum of the old memory modified by the forget gate $\boldsymbol{f}_t$ and the candidate memory $\tilde{\boldsymbol{c}}_t$ rescaled by the input gate $\boldsymbol{i}_t$ [36].

Finally, the new hidden state is calculated using the output gate and the new memory state

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) . \tag{30}$$

The new hidden state is then transferred to the output layer, where the actual model prediction $\hat{\boldsymbol{y}}_t$ is calculated, similar to Eq. (22) [36].

To summarize, the LSTM cell computes the hidden state $\boldsymbol{h}_t$, which is transmitted to the output layer, and the internal memory state $\boldsymbol{c}_t$ for every time step. The input, output, and forget gate, which control the flow of information and how the values for internal memory and hidden states are updated, are calculated from the previous hidden state $\boldsymbol{h}_{t-1}$ and data input $\boldsymbol{x}_t$. In the following time step, the new states of $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$ are used when the calculation starts once again.

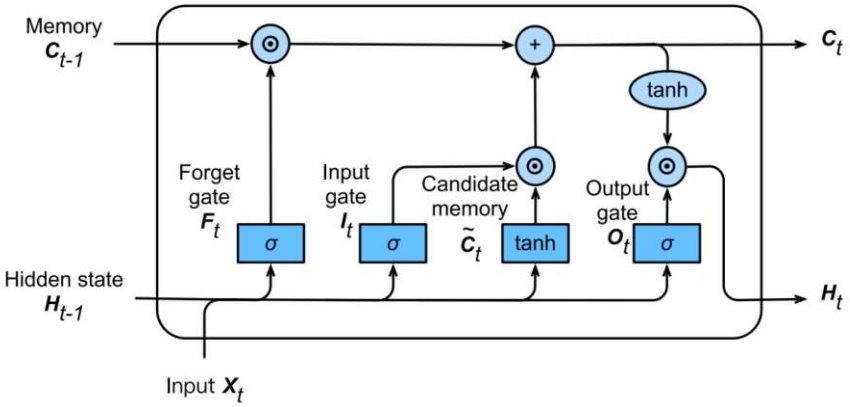


Figure 10: Structure of a LSTM Memory Cell. For a current time step t the cell uses the past hidden state $\boldsymbol{h}_{t-1}$, past memory $\boldsymbol{c}_{t-1}$ and the current input $\boldsymbol{x}_t$ to compute the current hidden state $\boldsymbol{h}_t$ and memory $\boldsymbol{c}_t$. These values are then used for the next time step. Figure adapted from [36].

The particular architecture of LSTM cells allows the network to decide when to store long-term information and when to discard it for new information automatically, without direct influence from outside. During training, the network learns to identify structures in the time series that causes it to prioritize more recent data or include long-term dependencies for the prediction [29]. Many studies have concluded that LSTM networks regularly outperform conventional RNNs. They have been particularly successful in the field of natural language processing, such as handwriting and speech recognition [29]. The capacity to model real-valued time series has also been investigated in different studies. For instance, Ma et al. [26] compared LSTMs with similar networks for a traffic speed prediction and found that in most cases the LSTM network was more accurate and more stable than its competitors. LSTMs have also been used in the process industry, for example for steam generation modeling [47], or for learning-based control [46].

2.2 Combustion Process and Control

2.2.1 Process Control Fundamentals

Process control is a technique used to efficiently and stably regulate dynamical systems and processes. Typically, a modern controller determines the current operation of a system, compares it against the desired behavior and calculates corrective measures based on a model of the system's response to external inputs. The controller output is transmitted to an actuator which performs the regulating action on the system. This type of control is called closed-loop or feedback control, as the controller uses feedback from the system to determine how to manipulate it. Closed-loop control is the most common kind of process control and forms the basis for many automatic control techniques.
Figure 11 shows a block diagram of a typical feedback control loop. The current value of the control variable or process variable $y(t)$ is compared to its desired value $r(t)$ (set point), and the difference $e(t)$ is used to calculate the manipulated variable or controller output $u(t)$ to adjust the system, which is constantly perturbed by external disturbances $d(t)$ [19]. A basic example of closed-loop control is the cruise control of an automobile. The control variable is the vehicle's current speed while the set point is the desired velocity. When disturbances such as the rising of the road's slope cause the speed to decrease, the control error grows and the controller adjusts the manipulated variable, in this case the accelerator position.

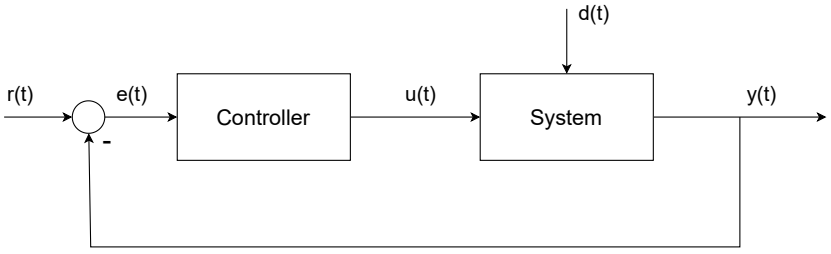


Figure 11: Block diagram of a feedback control system. The controlled variable $y(t)$ is compared to its set point $r(t)$ and the deviation, the so-called control error $e(t)$, is used as the controller input. The manipulated variable $u(t)$ is used to influence the system to minimize the control error. Noise and external disturbances $d(t)$ perturb the system.

There are several objectives for feedback control systems. Firstly, the control variable should always follow the set point behavior. Additionally, the response of the controller to a disturbance or rather a control error, which is the only input the controller has, should be fast with respect to the controlled process. Generally, the key objective for the dynamics of the closed-loop system is to be stable, which means that bounded disturbances give bounded errors, and that unwanted effects of external disturbances are reduced [9].

Among the advantages of feedback control are its simplicity and versatility, as almost any process can be regulated with feedback control. Furthermore, since the controller's input

is simply the control error, it always takes corrective action, no matter where the error originates from, whether it's an external disturbance, a change of the internal system behavior, or a change of the set point. This means that, if properly tuned, feedback control is very robust to changing process conditions.
On the other hand, the closed-loop can lead to feedback instability and oscillations, if the controller parameters are not set correctly. Since the controller only takes action when there is a control error, error is fundamental to feedback control and is to be expected. This also limits the speed of the response, as the controller can't act proactively [18].

Thus, in order to function properly the controller needs to be tuned to the system behavior and dynamics. The most effective way to determine how a process responds to changes of the manipulated variable is to analyze its step response. This response can then be described by three process characteristics: Firstly, the process gain G_P, that expresses how the system output changes for a change in its input:

$$G_P = \frac{\Delta y}{\Delta u} \,. \tag{31}$$

Secondly, the dead time t_D, which is the time delay between the step change at the input and the response of the process variable. Finally, the time constant τ describes how fast the process reaches its new equilibrium when the dead time has passed [18].

PID Controller

The proportional-integral-derivative (PID) controller is by far the most used controller for feedback control systems. About $90\,\%$ to $95\,\%$ of all control problems are handled by the PID controller and its variants [9]. As the name suggests, the PID controller consists of a proportional, integral and derivative part, where the total controller output is the sum of the three individual terms:

$$u(t) = u_P(t) + u_I(t) + u_D(t) \,. \tag{32}$$

The proportional part is simply calculated as

$$u_P(t) = K_C e(t) \,, \tag{33}$$

with K_C being the proportional controller gain. Using a proportional-only controller for a process with gain G_P, the closed-loop response which is the relationship between the controlled variable and set point evaluates to

$$y(t) = \frac{K_C G_P}{1 + K_C G_P} r(t) \,. \tag{34}$$

Since the fraction never equals one, this means that a proportional controller always results in a nonzero steady-state error [9]. In order to remove this error, an integral part is

introduced, whose output is calculated as

$$u_I(t) = K_I \int_0^t e(s)ds = \frac{K_C}{T_I} \int_0^t e(s)ds \,. \tag{35}$$

Here, the integral gain K_I is expressed as a fraction of the controller gain and the integration time T_I, which parametrizes the speed of the integration action. The integral part of the controller modifies the manipulated variable based on the accumulated error, and thus helps to remove the steady-state error caused by the proportional part. Finally, the derivative term is computed as

$$u_D(t) = K_D \frac{de(t)}{dt} = K_C T_D \frac{de(t)}{dt} \,, \tag{36}$$

with the derivative time parameter T_D. Intuitively, the derivative term can be thought as a linear extrapolation of the error for T_D time steps into the future:

$$e(t + T_D) \approx e(t) + T_D \frac{de(t)}{dt} \,.$$

This means that the derivative part of the controller is used to anticipate the future control error [19].
To summarize, the total control output of a PID controller is described by

$$u(t) = K_C \left(e(t) + \frac{1}{T_I} \int_0^t e(s)ds + T_D \frac{de(t)}{dt} \right) \,, \tag{37}$$

with the control parameters K_C, T_I and T_D.
The derivative mode is useful for very fast control responses, however its primary disadvantage is its sensitivity to noise. Usually, noise is made up of high frequencies which lead to amplification when differentiating, resulting in cyclic or unstable control output. Thus, many applications only use proportional-integral (PI) controllers, which are more stable although a bit slower [18].

2.2.2 Combustion Process and Combustion Plants

The combustion of solid biomass, for instance in the form of wood logs, pellets or chips, still is the most common way of generating energy from biomass. Typically, modern biomass combustion plants, the largest of which can reach a power output in the range of hundreds of megawatt, consist of grate or circulating fluidized bed boilers which produce hot water or steam. The steam can subsequently be transformed into electricity by way of a steam turbine, or the heat of the steam or water is used directly, for example for residential heating [28]. Generally, when operating biomass combustion plants the aim is to reach high efficiencies while minimizing emission of unwanted pollutants such as NO_x, CO or tars. Unfortunately, compared to other fuels, solid biomass consists of a relatively high amount of volatile substances like nitrogen, sulfur or potassium. These can lead to high pollutions in the flue gas when the combustion's operating conditions are suboptimal.

Additionally, during the biomass combustion process many complex chemical reactions and phase transitions occur, partly in parallel, partly in sequence, which can result in fast changing process conditions. This means that in order to reach the goal of an efficient, well performing combustion process with minimal pollutants, the boiler requires optimal control [30].

Biomass Combustion Process

The basic combustion process of solid biomass can be outlined in a sequence of four steps: Initially, the fuel is heated by radiation from the fire and hot interior of the furnace. This causes the water in the biomass to evaporate and dries the moist fuel [30].
Then, as the dry biomass reaches a temperature of approximately $150\,^{\circ}\mathrm{C}$ the process of pyrolysis starts. At that temperature the cellulose molecules which can be represented chemically by $CH_xO_yN_z$ are torn apart, leaving behind solid carbon in the form of char, while the other components escape as gases and leak into the air. These pyrolysis gases immediately combust in the form of different exothermic oxidation reactions, resulting in a variety of partially incomplete combustion gases (CO, CO_2, H_2O, N_2, NO_x, etc.).
The remaining char reacts in a gasification process at an even higher temperature, which mainly produces CO [30].
Finally, the incomplete combustion gas CO is fully oxidized to CO_2 in the upper part of the furnace. The resulting heat is transferred to the flue gas to heat the water in the boiler and is used to continue the combustion cycle. Approximately $2\,\%$ of the dry biomass consist of nonburnable minerals which remain as ash on the grate and need to be disposed of[30].

Figure 12 depicts the thermal decomposition of fuel in a grate boiler for co-current combustion, meaning the ignition starts at the bottom of the fuel bed and follows the primary air flow upwards [15]. On the grate the fuel is transported from left to right, where initially the moist fuel is heated in the "dead" zone until the thermal decomposition starts with the steps outlined above.

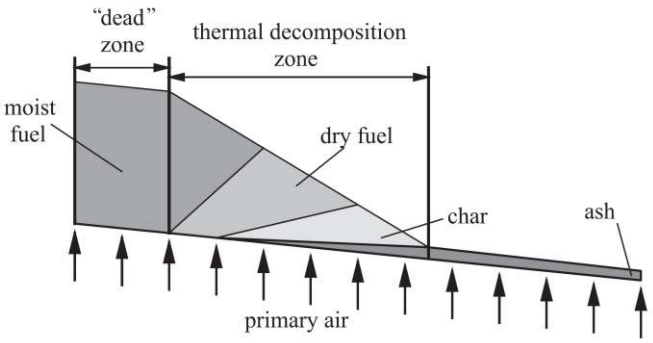


Figure 12: Thermal decomposition of fuel in a grate boiler for co-current combustion. Figure adapted from [15].

Biomass Combustion Plants

The full structure of a typical biomass furnace is shown in Fig. 13. Typically, such a moving grate furnace can be divided into a primary and secondary combustion zone, behind which lies the hot water boiler, where the heat exchanger is located. In the primary combustion zone the primary air flows form below the grate and mainly partial oxidation occurs. In the secondary combustion zone additional air is supplied via the secondary air fan in order to achieve complete combustion. Here, the flue gas has the highest temperature (up to 1000 °C) before most of its heat is transferred to the hot water which in turn is heated up to 100 °C. After the heat exchanger the flue gas has a temperature of around 120 °C to 180 °C. Part of the flue gas is recirculated back into the furnace in order to regulate the temperature of the combustion zone.

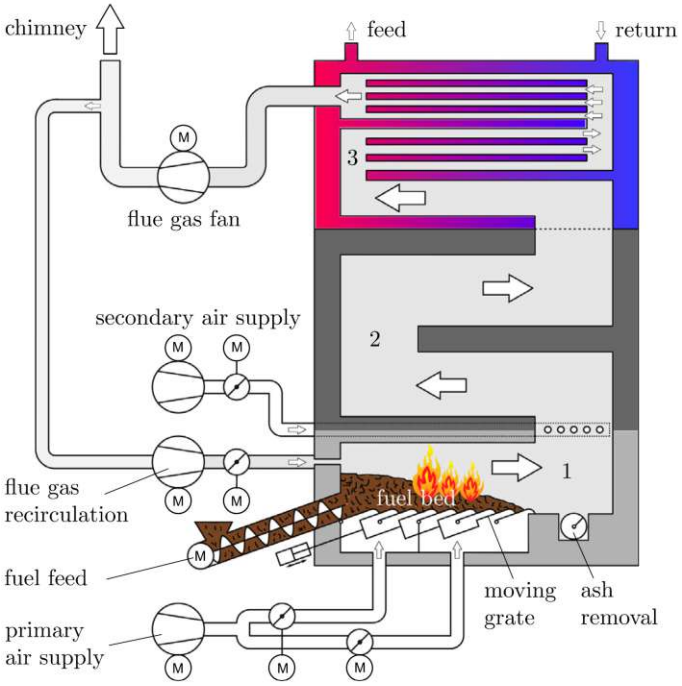


Figure 13: Schematic structure of a typical medium scale biomass furnace. The furnace can be divided into three segments: (1) primary combustion zone, where mainly incomplete combustion occurs; (2) secondary combustion zone, where combustion gases are fully oxidized; (3) hot water boiler, where heat from the flue gas is transferred to the water via heat exchanger. A part of the flue gas is recirculated to the combustion zones for temperature regulation. Figure adapted from [42].

The standard control scheme for biomass combustion plants like the one depicted in Fig. 13 is described in the following: Typically, the furnace control consists of four decoupled control loops each regulating one specific aspect of the combustion process. These loops normally are controlled by a PID or PI controller.

The first control loop is the power control. Depending on the type of plant, the controlled variable could either be the feed temperature or mass flow rate of the hot water or steam.

The manipulated variables are the primary air flow rate or, more commonly, simply the fan speed, and the fuel mass flow of the input. The task of the power control loop is to ensure a power output according to the desired set point even during fluctuations of the load or fuel properties [30].
The next control loop handles the temperature in the combustion chamber. The temperature is controlled with the flow rate of the recirculated flue gas. Due to the low oxygen content in the flue gas and its comparably low temperature, a higher recirculation flow leads to lower temperatures in the furnace. The temperature control is needed because too high temperatures cause compounds in the ash to melt which leads to slagging or fouling of the furnace. This means that the furnace temperature must not exceed a certain value [30].
The O_2 content of the flue gas is also regulated by its own control loop. Here, the secondary air flow rate is the manipulated variable, where a higher flow rate leads to a higher oxygen content. On the one hand, the oxygen content shouldn't be too high, as this results in a lower efficiency of the furnace. On the other hand, a very low oxygen content in the flue gas is an indication for incomplete combustion and thus for high CO values. The oxygen control is thus used to operate at the optimal point with high plant efficiency and low CO content [30].
Finally, the last control loop concerns the pressure control in the combustion chamber. This pressure is manipulated by the flue gas fan and should always be slightly below atmospheric pressure. This is a security measure to prevent leakage of combustion gases from the furnace [30].

2.3 Model-Based Control

The aim of model-based control is to achieve a higher control performance, especially for highly dynamical systems. As the name suggests, model-based control loops consist of a controller and a model of the system which provides additional information for the controller. A prominent example of model-based control is model predictive control (MPC). Here, at certain intervals the model is used to simulate the system behavior for different future control scenarios over a limited time horizon. Out of the different simulated control sequences, one sequence is selected that performs optimally according to some metric like smallest control error or least actuator manipulation. This optimal control sequence is then used for the control of the actual plant [30]. Another common model-based control technique is internal model control (IMC), where the model is placed in parallel to the plant and the real and modeling outputs are compared. Their difference, caused by the modeling mismatch and additional disturbances, is fed back through a robustness filter to the controller [4]. Generally, different types of models can be used for model-based control.

One possibility is to simulate the furnace using a mathematical model based on physical considerations. Due to the complex, nonlinear behavior of the furnace and boiler these models are hard to formulate. However, they are also very robust under different operating conditions. Gölles et al. [17] developed such a model as a nonlinear fourth order model and described its implementation in the process control setup, which was expanded upon, among others, by Schörghuber et al. [27, 42] and Seeber et al. [24]. The model is represented by four to five first order ordinary differential equations each describing one part of the furnace system, such as the fuel bed or the heat exchanger. For the control unit design the model is used for input-output-linearization, which is a strategy to transform a nonlinear control system into a linear control system by adjusting the control inputs and variables, see [17, 27].
Unfortunately, however, the model and corresponding control strategy require knowledge of many process variables such as disturbances or internal plant states which are seldom ever measured if they can be measured at all [30]. Additionally, many parameters in the model equations are dependent on specific plant properties which can only be determined during test runs of the plant. In summary, mathematical models, if properly configured, provide very precise and robust plant simulations, however they require extended knowledge of the plant and its process variables.

An alternative to mathematically construct explicit model equations is to implicitly learn the plant behavior from process data. Using machine learning, neural network models can be built by just using the process data without needing additional information about the plant. Typically, the model inputs are the controlled variables of the standard control scheme, such as power output and temperature, and the model outputs are the corresponding

manipulated variables, such as air flow rates and fuel input. Examples where artificial neural network models were used for the modeling or control of combustion processes have been reviewed by Kalogirou [6]. Further, both De et al. [11] and Smrekar et al. [14] used feedforward neural networks for combustion plant modeling, while He et al. [47] compared different types of recurrent neural networks for modeling purposes. In their work, Bonassi et al. [46] show how a recurrent neural network can be used as a system model for a MPC controller, as is depicted schematically in Fig. 14. Gregorčič and Lightbody [20] devised an IMC setup that uses a Gaussian process model to predict a nonlinear system, as can be seen in Fig. 15. Blonbou et al. [5] also present an IMC approach using ANNs, while in their review Hagan and Demuth [4] present different model-based control architectures.
Of course, neural network modeling requires a large amount of data for the model training. Accordingly, the accuracy of the model is limited by the quality of the training data. Additionally, the model can only properly predict in the operation regions that were represented in the training data, as the highly nonlinear nature of ANNs makes extrapolations very unstable [29].

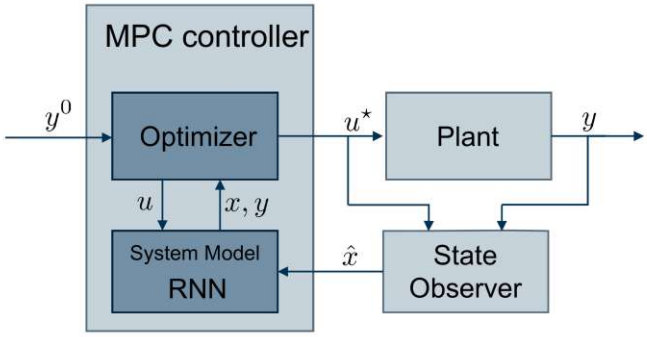


Figure 14: Schematic of a model predictive control setup using a RNN model. The MPC controller consists of the model and an optimizing algorithm that finds the ideal control sequence for the plant. The controller receives the set point y^0 and the current system state $\hat{x}$, and returns the optimal control output u^*. Figure adapted from [46].

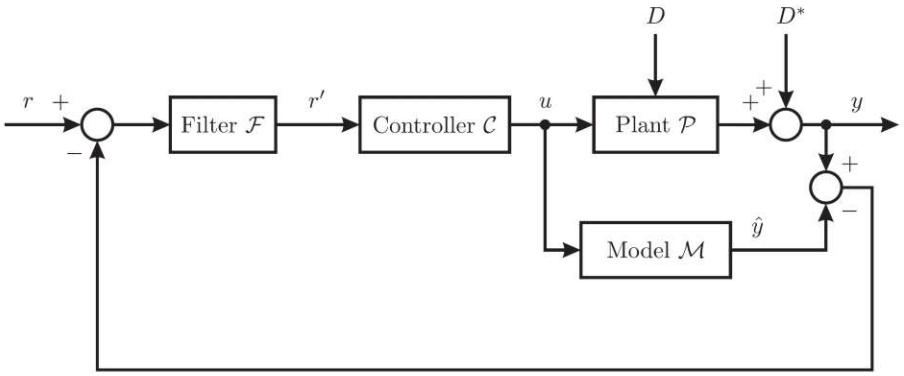


Figure 15: Schematic of an internal model control setup. The model output $\hat{y}$ is compared against the plant output y, which is perturbed by the disturbances D, D^*. The difference is compared to the set point r, filtered, and fed back to the controller. Figure adapted from [20].

3 Materials and Methods

For this work, I used different software tools for data preparation, neural network design, and model training. Initially, the raw data was preprocessed using *MATLAB* [49]. For the modeling of the biomass combustion plants I decided to create both a static FNN (MLP) model and a dynamic RNN (LSTM) model. This allows for the comparison between the different model types, contrasting their performance. The models were created in *Python* using the deep learning library *Keras* with the *TensorFlow* backend [25], which enables neural network design, model training and model evaluation. Plots were generated either in *MATLAB* or using the *Matplotlib* library for *Python*. In the appendix (5), the *MATLAB* and *Python* scripts for the data preprocessing and model training are listed.

3.1 Data Preprocessing

The raw data for this work originates from two different plants and consisted of several years of historical process data. The data was provided by CONENGA Group [51] and, on request, can be made available to the interested reader. The first plant subsequently called plant A is a small scale industrial furnace and boiler that provides warm water for district heating. The plant has a nominal power output of $1200\,\mathrm{kW}_{th}$ and supplies heating to a few dozen customers. The process variables that are used for plant modeling are listed in the following table:

	Measurement	Unit	Type
1	primary air fan 1 RPM	%	model input
2	primary air fan 2 RPM	%	model input
3	recirculated air fan RPM	%	model input
4	flue gas fan RPM	%	model input
5	secondary air fan 1 RPM	%	model input
6	secondary air fan 2 RPM	%	model input
7	fuel insertion pause time	s	model input
8	oxygen content in flue gas	%	model output
9	combustion chamber temperature	°C	model output
10	boiler power	kW	model output

Table 1: List of process variables used for the modeling of plant A. The variable type is determined by their role in the standard control scheme (see 2.2.2): manipulated variables are model inputs and controlled variables are model outputs.

In plant A, the respective fan speeds are given as a percentage of their maximal value. Both the primary air and the secondary air is supplied by two fans. Signal 7, the fuel insertion

pause time, describes the time between fuel insertion cycles, which means that a higher pause time results in less biomass fuel being inserted into the furnace.

Plant B consists of a medium scale furnace which generates steam for electricity generation by a steam turbine. The turbine has a nominal power of $20\,\mathrm{MW}_{el}$ and the expanded steam's residual heat is used as a heat source for an industrial wood pellets production. The process variables of the furnace are listed below in Table 2:

	Measurement	**Unit**	**Type**
1	fuel mass flow rate	$\mathrm{t/h}$	model input
2	primary air flow rate	$\mathrm{m^3/h}$	model input
3	recirculated air flow rate	$\mathrm{m^3/h}$	model input
4	secondary air flow rate	$\mathrm{m^3/h}$	model input
5	steam pressure	bar	model input
6	oxygen content in flue gas	$\%$	model output
7	combustion chamber temperature	$\mathrm{°C}$	model output
8	steam mass flow rate	$\mathrm{t/h}$	model output

Table 2: List of process variables used for modeling of plant B. The variable type is determined by their role in the standard control scheme (see 2.2.2): manipulated variables are model inputs and controlled variables are model outputs.

The volumetric air flow rates are measured in standard cubic meters. Because the steam pressure is controlled externally from the turbine, the steam mass flow rate is a measure of the boiler power.

Generally, these measurements are recorded every one to two seconds. In order to get a dataset with regular intervals the first data preprocessing step is to resample the dataset to a consistent frequency. This is needed for the subsequent data preprocessing procedures as well as the RNN model inputs. Thus, the signals were resampled to a sample time of one second using linear interpolation.

The next crucial step in the data preprocessing sequence is filtering. Generally, signals are afflicted by noise which is made up of high frequency variations in the signal. These variations can be caused by random thermal motion of electrons in the measurement electronics, for example. The goal of filtering is to reduce or entirely remove these high frequency perturbations which carry no inherent information and would otherwise lead to a worse model performance. Importantly, the noise filter should not impact the true signal as that would reduce the information of the signal. This means the filter needs to be configured in such a way that it attenuates the high frequency noise without affecting the

variation of the base signal.
For this work, the process data was filtered according to a first order transfer function with a time constant of $\tau = 30\,\mathrm{s}$. This removes much of the high frequency noise while preserving the slower process behavior of the combustion process. Figure 16 depicts the raw (blue) and filtered (orange) signal of the combustion chamber temperature of plant B. While the noise can be seen clearly in the blue graph, the filtered signal shows practically no noise while still following the trend of the raw signal.

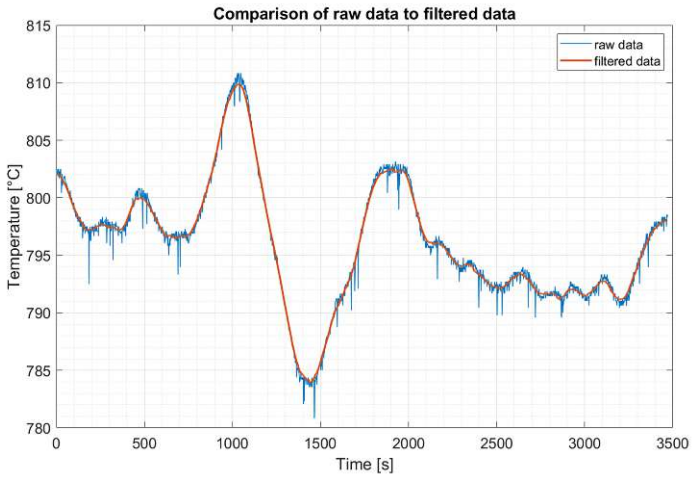


Figure 16: Plot of the combustion chamber temperature measurement of plant B. The blue graph shows the noisy raw data compared to the filtered signal depicted by the orange graph.

After a few preliminary tests, the modeling dataset for the two plants was chosen to encompass roughly one full year of process data. This is important in order to capture the entire range of process states which vary widely with different seasonal conditions, especially for heating plants. Additionally, when selecting the data for modeling more recent data is preferred, as many plants are modified or upgraded every few years which also changes the operating conditions.
Another thing that has to be considered is the fact that plants usually have a revision period once a year where the plant is powered down to clean the furnace and make small repairs or fixes. During these times, naturally, no process data is recorded. Furthermore, unplanned (partial) plant shutdowns, data outages, off-nominal operations, faulty sensor measurements or errors in the data recording system lead to erroneous data points and outliers in the dataset. Before using the data for model training, these outliers need to be removed, as they would otherwise hinder model training due to their abnormal process conditions.
The data is best sorted through by plotting the process variables and visually identifying periods of shutdowns and faulty data which have to be removed. Figure 17 shows the power and temperature trend of plant A for the period that was selected for training,

covering one full year. Outliers and shutdown phases have been removed from the data, as can be seen from the gaps in the trend. Especially the power graph (blue) shows the seasonally changing operating conditions, but also the daily variations with spikes usually representing higher power usage in the morning and evening.

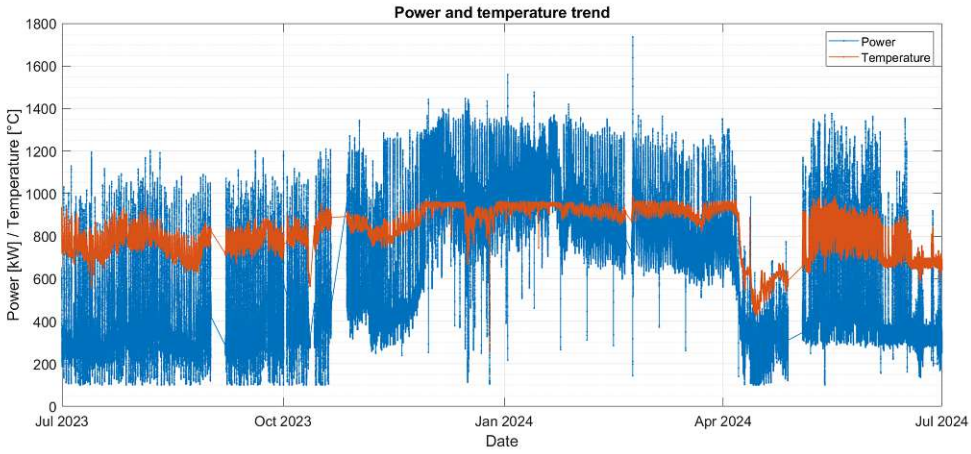


Figure 17: Plot of the power (blue) and temperature (orange) trend of plant A over one whole year.

Generally, due to selecting an entire year of measurements, this should result in the data evenly covering the different operation points of the plant, represented by state of power, temperature, and oxygen content in the flue gas. This roughly even or consistent distribution of the process states is visualized in Fig. 18. Here, in order to show their frequency, the measurements are sorted by their value, with a shallow slope meaning a high sample density in that operating region and a steep slope meaning a lower sample density. In this case, the sample density of the three plant outputs is low only at the extreme ends of the operating spectrum.
The next data preprocessing step is to downsample the dataset to a lower sampling frequency. This is done because of the high number of data points for year-round one second data logging ($\approx 3.1 \times 10^7$). The individual samples only provide very little new information because of the comparatively slow combustion process, while slowing down the training procedure and corresponding data handling due to their large number. The new sample time should be chosen to optimally capture the desired process dynamics. In this work, the data was resampled to two datasets with sampling times of $5\,\mathrm{s}$ and $30\,\mathrm{s}$, which allowed comparisons between the datasets.

One issue that arises when working with artificial neural networks is that model training is slowed down when the training data values differ by order of magnitudes. This is because high data values can lead to vanishing or exploding gradients during backpropagation [35]. To counteract this issue, the data needs to be standardized or rescaled to a lower range. In this case, the data was rescaled to the range of $[0, 1]$ by using the minimum and maximum

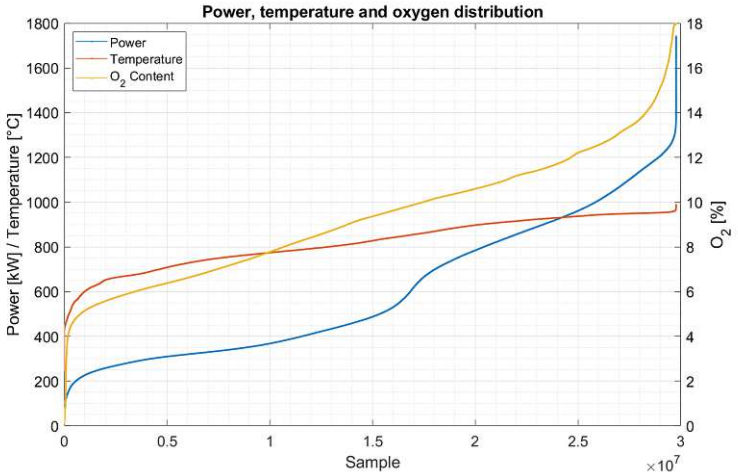


Figure 18: Plot of the power (blue), temperature (orange), and O_2 content (yellow) distribution of plant A over one whole year. Measurements are sorted by their values to determine their frequency of occurrence at that operating point.

values of the individual measurements:

$$\tilde{x}_j = \frac{x_j - x_{min}}{x_{max} - x_{min}}\,, \tag{38}$$

with $\tilde{x}_j$ being the rescaled value of the j-th feature, x_j the original value, and x_{max} and x_{min} the maximal and minimal value.

Finally, in the preparation for the training procedure the data needs to be shuffled and split into a training, validation and test set. For this purpose, what must first be discussed is the structure of the individual training samples for the FNN and LSTM models. Since the FNN model simply predicts the plant output at time t from the matching plant inputs, one sample is easily constructed: The model gets the individual inputs listed in Table 1 or 2 for one point in time and has to predict the corresponding outputs. In the case of plant A, one training sample for the FNN has seven inputs and three outputs.
For the LSTM network, the samples are constructed as a sequence of measurements. Here, one sample consists of k data points or time steps for the input and the $k+1$-th time step for the output which the model has to predict (one-step-ahead prediction). While the output again simply consists of the three plant outputs, the inputs for every time step now are made up of all plant measurements at that time. This means the RNN model gets both the manipulated variables and the controlled variables of the plant over a number of time steps and predicts the controlled variables for the next time step. In the case of plant A, one training sample for the LSTM network is comprised of k time steps for the input, where each time step contains ten data points, and one time step for the output which consists of the three plant outputs. Note that this structure of the sample allows the network to be used for multi-step-ahead prediction, where the model can iteratively predict multiple time

steps into the future.
This means when preparing the training data the samples must be constructed from the measurement data accordingly. While the samples for the FNN model just need to be sorted into inputs and outputs, the LSTM samples are constructed using the *Keras*-native `timeseries_dataset_from_array` function which separates the raw data array and arranges it into time series with the desired shape.

Now, before splitting the data into a training, validation and testing set the samples are shuffled to guarantee a uniform distribution of all plant operations in the different datasets, which is a prerequisite for stochastic gradient descent. After performing a few tests with different percentages it was found that, due to the large amount of data, the split ratio does not have a noticeable effect on the model performance. Thus, the split percentages were chosen to be $60\,\%$ for the training set, $15\,\%$ for the validation set and $25\,\%$ for the test set, which are typical values recommended in the literature [29].

3.2 Model Hyperparameter Selection

Typically, a machine learning model is characterized by several so-called hyperparameters that determine its architecture, learning procedure, and other model properties. In the case of a neural net typical hyperparameters are the number of hidden layers, the width of these layers, or the learning rate, for instance. Depending on the type, the values of these hyperparameters have a large influence on the model performance and the overall training efficiency. The number of hidden units, for example, which can be adjusted by the number of hidden layers or the width of these layers generally affects the capacity of the model, with more hidden units resulting in a higher representational capacity. However, more hidden units also lead to more time expensive training, eventually yielding diminishing returns, and, additionally, might lead to model overfitting. Similarly, the learning rate should be tuned optimally, as too high or too low values lead to poor model optimization during training [29].

A common way to select the values of model hyperparameters is to perform grid search [29]. For this algorithm, first every parameter is assigned a set of possible values. Then, the grid search procedure starts by training a model for every possible combination of values and compares the resulting validation errors. The model with the lowest error has the best configuration. One problem with grid search is that the number of necessary model trainings rises exponentially with increasing number of tested hyperparameters and larger sets of possible values. This means that it is faster to limit the number of investigated hyperparameters per grid search and perform the grid search algorithm several times with different configurations. Another way to reduce the duration of the grid search procedure is by decreasing the size of the training dataset.

With the help of the grid search algorithm, different hyperparameters and neural network characteristics such as the optimizing function or the learning rate were explored. For the optimizer, which primarily affects the training speed and resulting validation error, three different gradient descent functions (SGD, RMSProp and Adam) were compared against each other. In all tests, both RMSProp and Adam yielded significantly better results than SGD, while they performed equally well. Thus, Adam which is a common recommendation in literature was chosen as the optimizing function for both the FNN and RNN model.

Next, the optimal learning rate was investigated, where possible values ranged from $\eta = 0.0001$ to $\eta = 0.1$ on a logarithmic scale. Here, one correlation that was found during training was the fact that, generally, a smaller learning rate resulted in a longer training duration. At the same time, however, lower learning rates yielded a lower validation error. These relations are depicted in Fig. 19. For most cases, it was found that a learning rate of $\eta = 0.001$ showed the best performance, which was settled on.

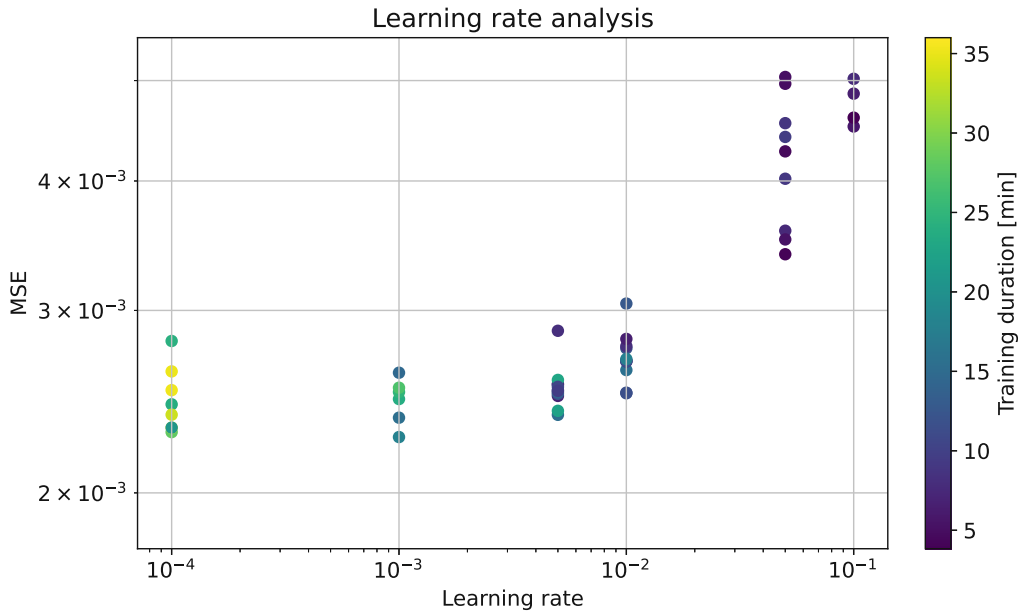


Figure 19: Scatter plot of the validation error for FNN models with different learning rates. The individual data points are colored to indicate the training duration of the model. While lower learning rates lead to a smaller validation error, they also result in a higher training duration.

The next characteristics to investigate were the neural network dimensions, which are the depth and width of the hidden layers in the case of the FNN, and the number of memory cells in the case of the LSTM.

For the FNN, the hidden layer depth was varied from one to five, and the width from 4 to 128 on a logarithmic scale. More hidden units mean a higher representational capacity for the model, and indeed it was found that, in general, networks with a higher number of hidden units showed a better performance. At a certain number of hidden units, this effect is reduced to a point where no performance improvement is generated from increasing the number further. Moreover, a higher number of hidden units makes the network computationally more expensive to train, resulting in longer training durations per epoch, for no additional gain. Both these findings are shown in Fig. 20, where the validation error decreases for models with an increasing hidden unit count of up to approximately 200, and then stays roughly constant. Thus, the dimensions for the multilayer perceptron FNN were chosen to be 4 hidden layers with a width of 64 neurons.

The LSTM dimensions were tested in the range from 8 to 64 memory cells. Again, higher dimensions improved the performance, as the network gained the capacity to fully model the process dynamic. However, models with 32 and 64 memory cells showed similar validation performances, indicating that the process can already be modeled sufficiently well with 32 memory cells. Therefore, the number of memory cells in the LSTM network was chosen to be 32 as it showed equal performance at a lower computational expense (i.e. lower training duration).

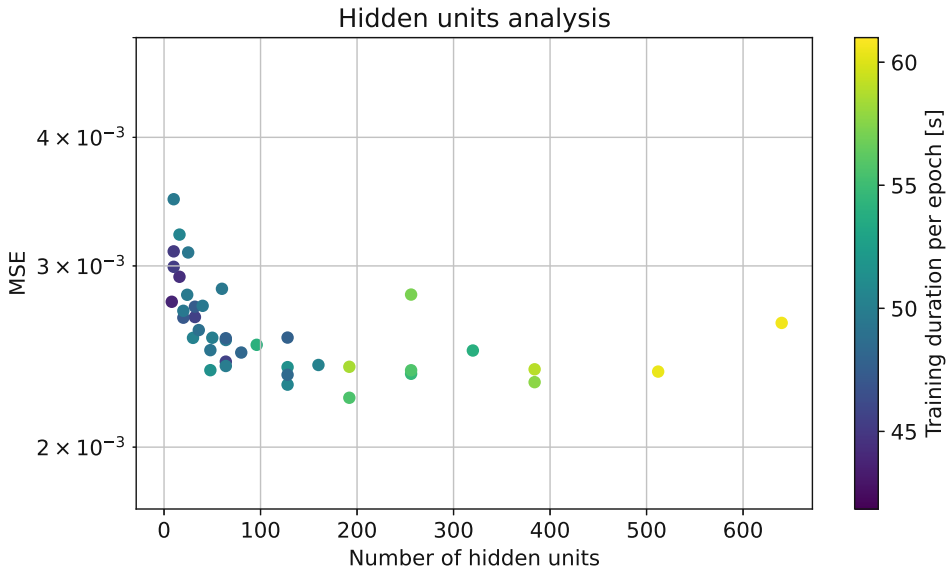


Figure 20: Validation error for FNN models with different hidden layer dimensions. The plot shows the total number of hidden units (i.e. hidden layer depth $\times$ hidden layer width) versus the validation error. Additionally, the data points are colored to show the model training duration per training epoch. While the total number of hidden units positively impacts the model error up to a point, increasing the number of hidden units further offers no additional benefit, instead making only the model more expensive.

The selection of the activation function was not based on a grid search but simply on the default recommendation for deep neural feedforward nets. It has been shown that deep FNNs with ReLU activations are easier and faster to optimize than comparable networks using saturating functions such as perceptrons [29]. Thus, the activation function for the FNNs was chosen to be the ReLU.
For the LSTM network, the activation functions used in the default configuration (as described in section 2.1.3) were selected. Thus, the hyperbolic tangent and the sigmoid function were used as activations.

Next, the loss metric needed to be decided upon. Typically for regression tasks, the loss is either calculated as a mean square or a mean absolute of the error. However, comparisons of different models trained with MSE and MAE loss metrics showed no discernible difference in their performance. Models which were trained with the MAE also had a low MSE loss and showed a similar performance to models trained with the MSE. The loss function for the models was decided to be the MSE (see Eq. (1)).

The final training characteristic that needed to be set was the batch size. The (mini-)batch size defines how many samples are regarded for one stochastic gradient descent or Adam step. A large batch size more accurately portrays the gradient of the full dataset. However, calculating the gradient of a larger batch becomes computationally expensive very quickly,

which is why recommended values range from tens to a few hundred samples at maximum [29]. For this work, the batch size was chosen to be 64 for the FNN models and 128 for the LSTM networks.

To summarize, the neural network hyperparameters for both the FNN and the LSTM are listed below, in Table 3. The training and testing procedure is discussed in the following section.

Hyperparameter	**FNN**	**LSTM**
Optimizing function	Adam	Adam
Learning rate	$\eta = 0.001$	$\eta = 0.001$
Network dimensions	4×128 hidden units	32 memory cells
Activation function	ReLU	tanh, sigmoid
Loss metric	MSE	MSE
Batch size	64	128

Table 3: List of hyperparameter values used for the FNN and LSTM model.

3.3 Training Procedure and Testing

With the hyperparameter values selected, the model is designed according to these specifications. Using the `fit` function from *Keras* the neural network is trained, using the provided training and validation datasets. During training, the function records training and validation metrics for every epoch, which help to evaluate the training progress of the model. Additionally, the early stopping method is used during model training. This acts as a regularizer and prevents overfitting, as training is stopped when the validation error increases again during training. In this case, the early stopping algorithm was used with a patience of a few epochs, so as to only stop training when the validation error hasn't improved for a number of epochs. For the training of the LSTM networks, the early stopping patience was set to 5, while the FNN stopping patience was set to 20 due to an overall higher number of training epochs. When the training is stopped, the model weights are restored to their optimal values. With this, the model training is concluded, the model and its weights can be saved, and the next step is to evaluate the model performance.

To this end, *Keras* provides the `evaluate` function which returns the loss metric, in this case the MSE, for the provided test data. This is a good first indication of the model performance and can serve as a quick sanity check, as the test loss should be roughly equal to the validation loss. In order to get a deeper insight into the model's performance, for example the model accuracy in different operating conditions, a more thorough analysis needs to be carried out. This analysis is done using the `predict` method which generates output predictions for the inserted samples. These predictions then are compared to the actual outputs, which enables investigating under what circumstances the model performs well and where it makes larger errors. The performance analysis for the different models is discussed in chapter 4.

The LSTM models can be evaluated further by utilizing their ability to iteratively make predictions several steps into the future. Here, the one-step-ahead prediction is calculated, which is fed back to the model as the latest entry in the input sequence for the next prediction step [16]. This way, the model which was trained as a one-step-ahead predictor can be used to forecast multiple steps into the future. These multi-step-ahead predictions can then be compared to the actual plant behavior, evaluating the model accuracy over a longer time horizon.

4 Results

In the following section I present and discuss the results of the plant modeling. First, I review the static FNN models for the two plants are and analyze their performance. Then, I evaluate the LSTM models and compare them to the static models. I contrast different LSTM model configurations and review the use of these one-step-ahead predictors for multi-step predictions.

4.1 FNN Models

4.1.1 Plant A

The first prototype models were trained with a reduced set of process data. Initially, for the modeling of plant A, three operating regions (low, medium, and high power operations) were selected from the raw data. This was done following the training procedures described by De et al. [11] and Smrekar et al. [14] who in their work selected distinct data sections from different operating categories. This initial dataset is depicted in Fig. 21, where every operating region is represented by roughly five days worth of data points.

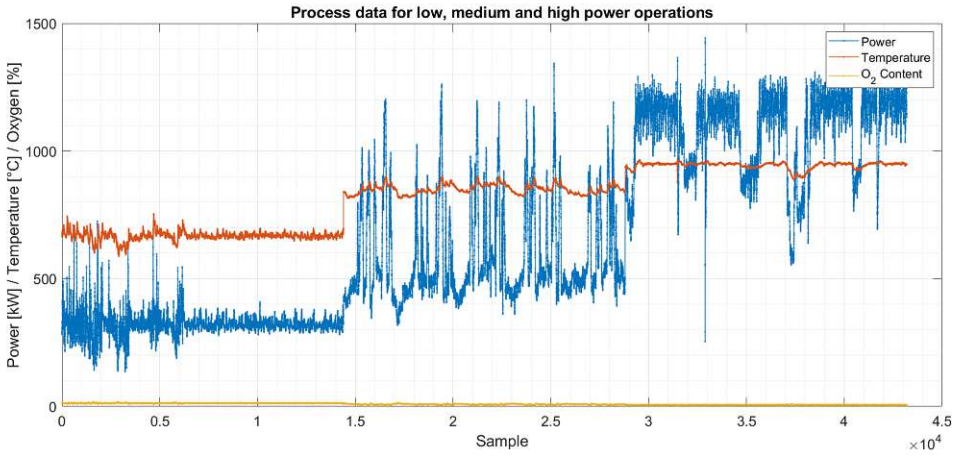


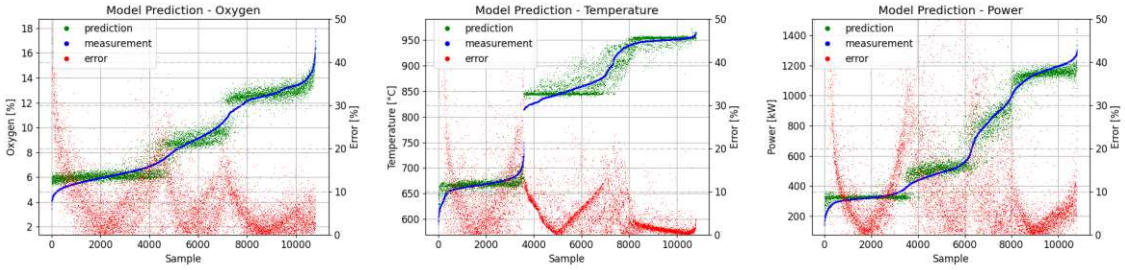
Figure 21: Initial training dataset where data is selected from low, medium and high power operation of the plant. The figure displays the plant outputs power (blue), combustion chamber temperature (orange), and flue gas oxygen content (yellow).

Quite fast, however, it became clear that this approach of only using data from a few operating regions without transitions between those regions resulted in an unsatisfactory model performance, as is shown in the following figure.

Figure 22 depicts the performance analysis of the model using the test data, showing the oxygen prediction on the left, the temperature prediction in the center, and the power prediction on the right. In all three cases the model predictions are compared to the true values for the whole range of the test data samples. The predictions are shown as a green dot versus the blue dot of the true measurement. The relative error, computed as

$e_i = |\frac{y_i - \hat{y}_i}{y_i}|$, is plotted in red, with its values being marked by the axis on the right-hand side. For better readability the samples are sorted by the value of the true measurement. Not only does this visualize in which process variable ranges the model performs badly, but it also shows how densely which part of the operational range is covered by the dataset.

What becomes apparent is that the model predictions hardly reflect the true process behavior. Instead, especially in the case of Fig. 22(a) and Fig. 22(b), the model makes roughly constant predictions for the different operational regions. This leads to high errors at the ends of the individual sections. Additionally, because of the way the data was selected, it does not evenly cover the entire operation region, particularly in the case of the combustion chamber temperature measurement depicted in Fig. 22(b), where there are no samples from $750\,°\mathrm{C}$ to $800\,°\mathrm{C}$ in the dataset. Due to the highly nonlinear nature of neural networks the model would act unpredictably if it encountered samples that lie in untrained operating regions.



(a) Oxygen predictions. (b) Temperature predictions. (c) Power predictions.

Figure 22: Test data predictions from the prototype model whose training is based on the categorized data. For every sample the model prediction (green), the true value (blue), and the relative error of the prediction (red) is depicted. Additionally, the samples are ordered according to their true value. This enables assessing the distribution of samples in the dataset.

All in all, these findings show that a proper model requires a bigger, more diverse training dataset that covers the entire operating region of the plant. Thus, a more comprehensive dataset was created following the steps outlined in section 3.1, using one whole year of process data (as in Fig. 17), which was used for the training of a new FNN model that is analyzed in the following.

The new model for plant A used the hyperparameter configuration listed in Table 3. In this case, the training took place over 127 epochs with a total training duration close to 32 minutes. Figure 23 depicts the decline and eventual leveling out of the validation error as the training progresses alongside the training error. The minimal validation error is reached at the 107th training iteration with $\mathrm{MSE} = 2.142 \times 10^{-3}$, after which the loss doesn't improve for the next 20 epochs. Note that the MSE is calculated from the training data which is normalized to the range of $[0, 1]$.

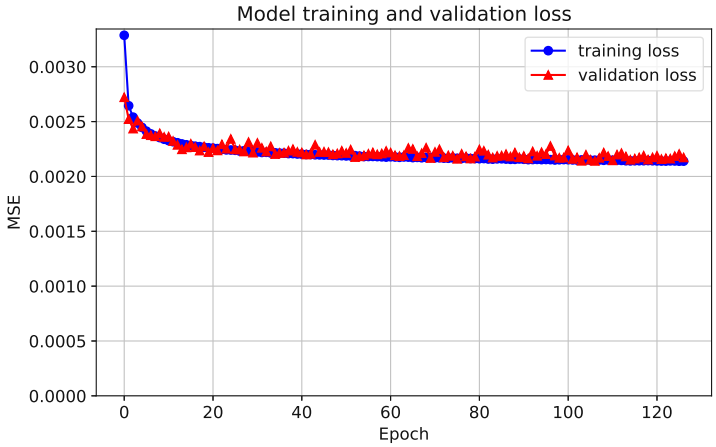
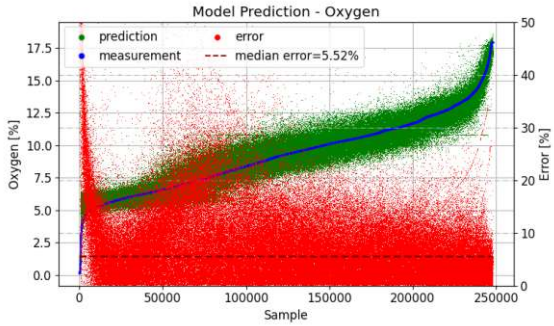


Figure 23: Trend of the training error (blue) and validation error (red) for the FNN model of plant A. The lowest validation error was reached at the 107th epoch with a loss of $\mathrm{MSE} = 0.002142$.

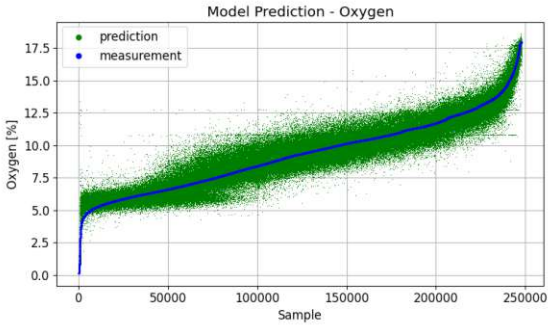
Figure 24 shows the model predictions compared to the true values for the three model outputs O_2 content, temperature, and boiler power. In the plots on the left side the relative error between prediction and measurement is plotted for every data point, and the median error is marked. The median error was chosen as it is less susceptible to outliers. Since the error markers partially cover the measurement and prediction data points, the same plots are depicted on the right side without error markers.

Due to the changes in the modeling dataset the biggest issues of the prototype model have been corrected. The operational range is now represented evenly by the training and test set, as is evident by the blue graphs in the plots. Only at the extreme ends of the process spectrum there are fewer samples, but most of the process behavior is covered consistently. Furthermore, for a large extend the predictions now follow the trend of the true values more correctly, where the predictions are scattered around the measurements with a roughly constant spread, and, importantly, with fewer sections of constant predictions.

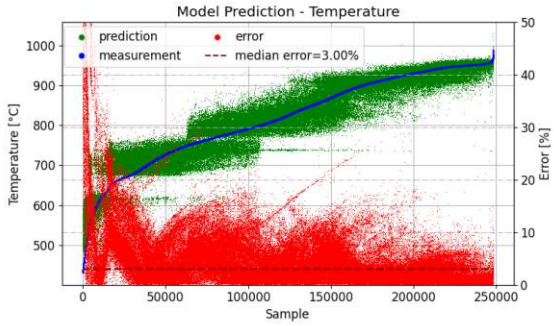
However, the relative errors of the predictions are still quite large, especially for the power predictions with a median error of $8.48\,\%$, and there are many cases where predictions deviate by over $20\,\%$. For the oxygen content and temperature predictions the errors are lower, with a median of $5.52\,\%$ and $3.00\,\%$ respectively, and most errors below $20\,\%$, but they still are considerable.
Taking a closer look at the oxygen content predictions in Fig. 24(a-b), the error distribution shows a peak at the low end of the sample spectrum, where the sample density is very low. The model also generates slightly worse predictions at a measurement value of approximately $7.5\,\%$ O_2. Apart from that, the oxygen predictions are distributed relatively uniformly around the true values.



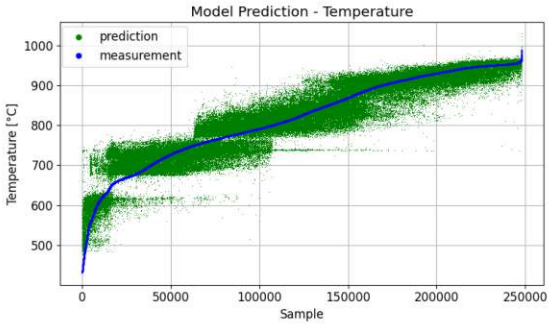
(a) Oxygen predictions.



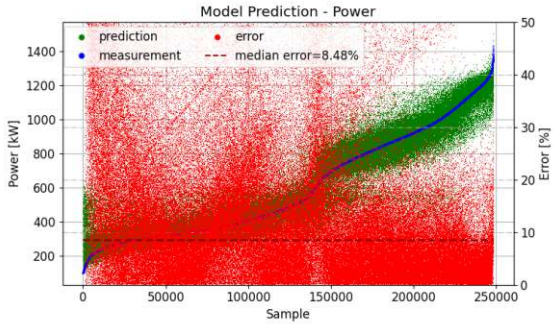
(b) Oxygen predictions without errors.



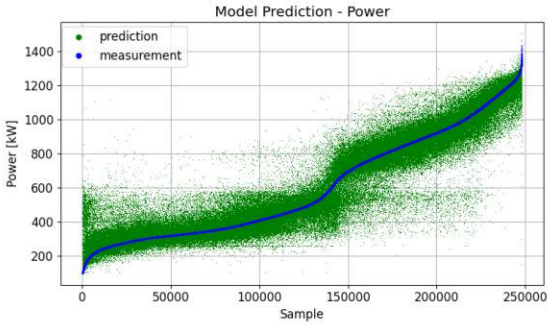
(c) Temperature predictions.



(d) Temperature predictions without errors.



(e) Power predictions.



(f) Power predictions without errors.

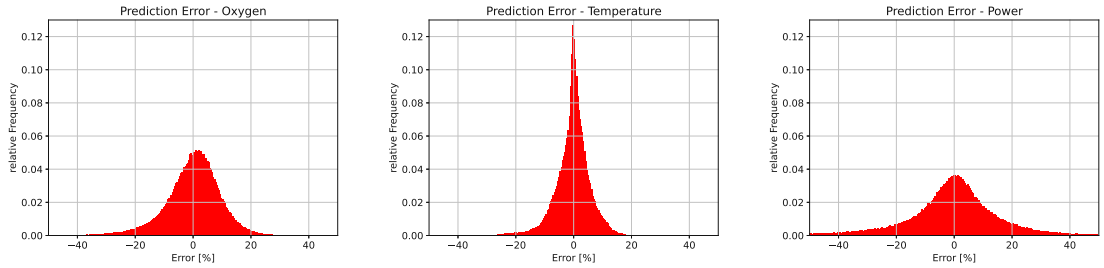
Figure 24: Test data predictions from the model trained with the comprehensive dataset. For every sample the model prediction (green), the true value (blue), and the relative error of the prediction (red) is depicted. Additionally, the samples are arranged according to their true value. For every output the plot is shown with and without error marker.

Curiously, the temperature predictions in Fig. 24(c-d) still show gaps or discontinuities, and sections with close to constant values. Since the neural network model is a black box model, the reason for these inconsistencies cannot be determined explicitly. Apparently, given the provided inputs, it is not possible for the model to make more precise predictions. Still, due to the high absolute value of the temperature measurements the median relative error is the lowest out of the three outputs. For high temperatures (above $900\,°\mathrm{C}$) the prediction errors are practically all below $10\,\%$.

The power predictions in Fig. 24(e-f) exhibit the highest errors out of the three plant outputs. Especially for low and medium power operations the model struggles to make

accurate predictions. This might also be partly due to the comparably wide spectrum of power operations ranging from $100\,\mathrm{kW}$ to $1400\,\mathrm{kW}$. Still, it means that the current boiler power cannot be easily or accurately modeled just by considering the momentary plant inputs.

Figure 25 depicts histograms for the errors of the individual model outputs. As can be seen, due to higher accuracy of the temperature predictions the histogram in Fig. 25(b) is narrower and higher with a sharper peak than the histograms for oxygen and power predictions. The histogram for the power prediction error in Fig. 25(c) on the other hand is lower and wider, with a substantial amount of errors outside the $[-20\,\%, 20\,\%]$ interval. Interestingly, the histogram in Fig. 25(a) leans somewhat to the right indicating that the model slightly underestimates the oxygen content.



(a) Oxygen prediction errors.

(b) Temperature prediction errors.

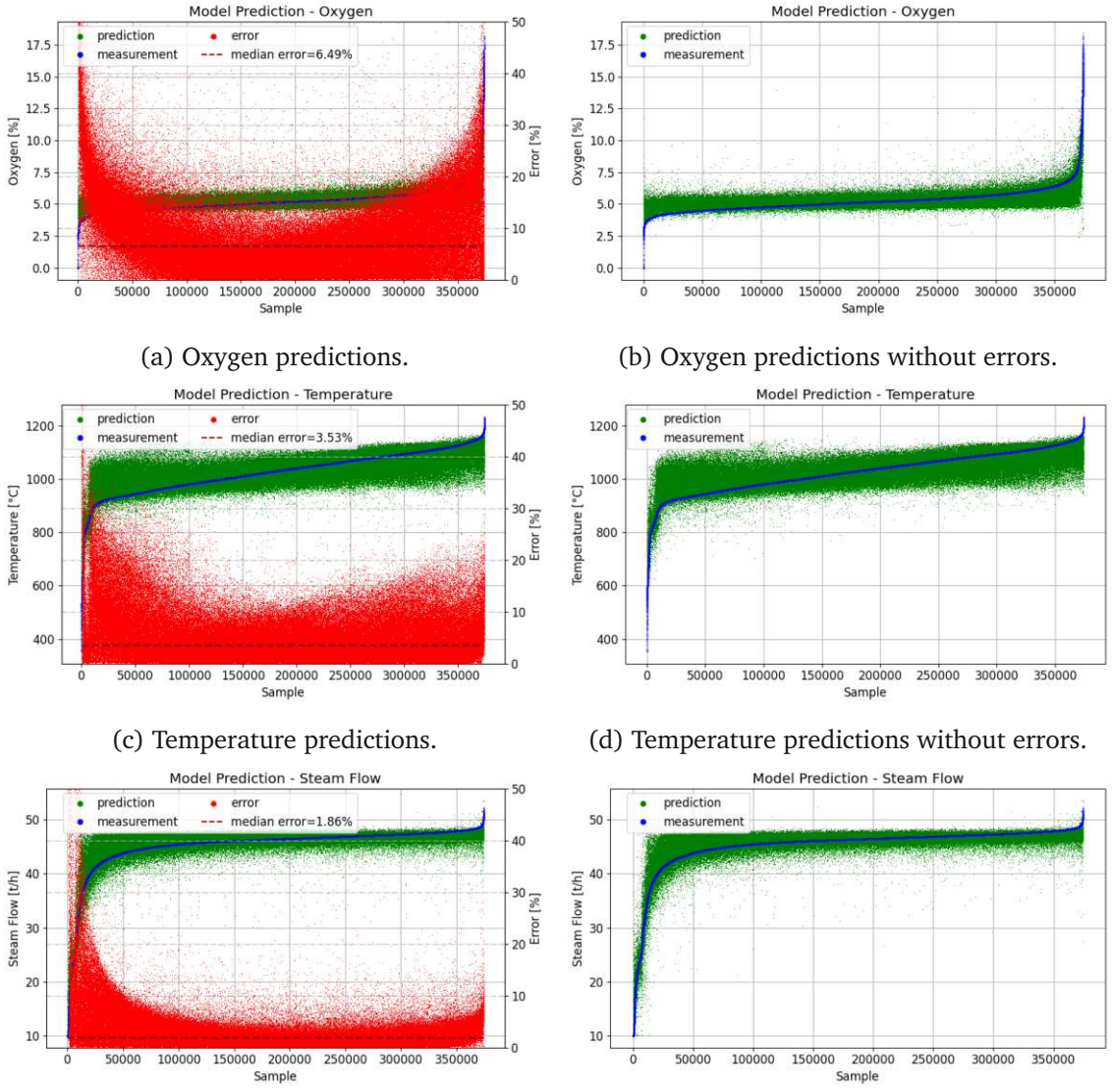
(c) Power prediction errors.

Figure 25: Histograms showing the relative frequency of errors for the oxygen, temperature, and power predictions.

In general, it can be said that the large error of the FNN model is still unsatisfactory, with many cases where predictions are deviating from the true value by more than ten to twenty percent, even when using the expanded set of process data covering one year of operation. This error is too large to use the model for any kind of integration into the plant control setup.

4.1.2 Plant B

Plant B consists of a medium scale furnace that produces steam for electricity generation in a steam turbine. The FNN model was trained for 102 epochs (approximately $36\,\mathrm{min}$) with the process data from the furnace of plant B, now encompassing roughly one full year from the beginning. As early stopping with a patience of 20 epochs was used, the lowest validation loss was reached after the 82nd iteration, with $\mathrm{MSE} = 1.484 \times 10^{-3}$.



(a) Oxygen predictions.

(b) Oxygen predictions without errors.

(c) Temperature predictions.

(d) Temperature predictions without errors.

(e) Steam flow predictions.

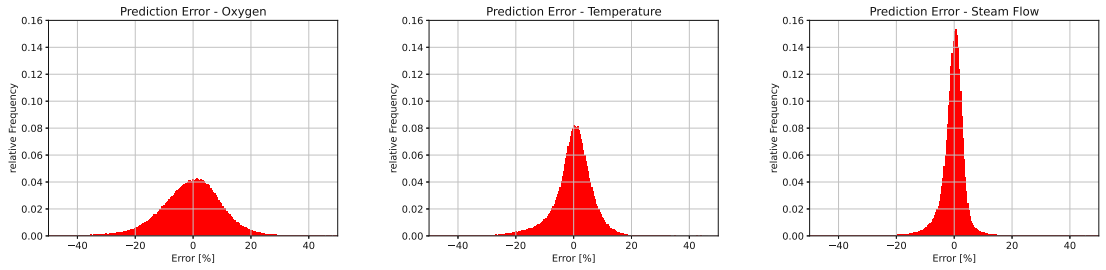
(f) Steam flow predictions without errors.

Figure 26: Plant B test data predictions from the FNN model. For every sample the model prediction (green), the true value (blue), and the relative error of the prediction (red) is depicted. Additionally, the samples are arranged according to their true value. For every output the plot is shown with and without error marker.

Figure 26 shows the model predictions for the plant outputs oxygen content, temperature,

and steam flow. What stands out is that, compared to the plots for plant A in Fig. 24, plant B has a more static operation in the sense that for most of the year the furnace operates in approximately the same process range, as can be seen from the blue graphs depicting the true measurement range. The steam flow for instance lies between $40\,\mathrm{t/h}$ to $50\,\mathrm{t/h}$ for roughly $90\,\%$ of the time, and the temperature is mostly between $900\,°\mathrm{C}$ and $1100\,°\mathrm{C}$. This is because, unlike plant A which provides residential heating with strong variations in demand (seasonal and daily), plant B provides steam for electricity generation with a more or less constant demand.

However, the errors which are also depicted as histograms in Fig. 27 are similarly large. In this case, the oxygen predictions display the biggest relative deviations with a median error of $6.49\,\%$. The temperature predictions amount to a median error of $3.53\,\%$. The most accurate output is the steam flow prediction which reaches a median error of $1.86\,\%$ with most samples having an error well below $10\,\%$. Accordingly, the histogram in Fig. 27(c) is the narrowest. For all three outputs, the errors are largest at the ends of the process variable spectrum where the sample density is lower.



(a) Oxygen prediction errors.

(b) Temperature prediction errors.

(c) Steam flow prediction errors.

Figure 27: Histograms showing the relative frequency of errors for the oxygen, temperature, and steam flow predictions.

In conclusion, the static FNN models have difficulties with accurately predicting the combustion process for the investigated plants. In many cases the relative prediction error is bigger than $10\,\%$, which is a significant uncertainty for any productive use on-site. Since the model capacity is sufficient and larger models do not perform better, as tested in section 3.2, it seems like the models' performance is limited by their structure and inputs. Apparently, the dynamical combustion process cannot be predicted with a higher accuracy by a static model. This is plausible as, depending on the prior plant states, the same combination of plant inputs can lead to different plant outputs. According to this, a dynamical model which incorporates the information of previous process states should be more precise when predicting the next state. Dynamical models in the form of LSTM networks are evaluated in the following section.

4.2 LSTM Models

4.2.1 Plant A

For the dynamic plant models the LSTM networks use the hyperparameters specified in Table 3. For the first iteration of LSTM models the training samples, constructed the way described in section 3.1, consisted of 120 time steps with a sampling time of 30 seconds. Thus, the model input in this case amounted to 1 hour of past states, plant inputs and plant outputs, to predict the subsequent plant outputs for the next sample. Again, the full dataset was used, covering one year of process data.

The corresponding model training took place over 52 epochs with a training time of 157 minutes and resulted in a minimal validation loss of $\mathrm{MSE} = 6.756 \times 10^{-5}$, as can be seen in Fig. 28. Note that the MSE is two orders of magnitude lower than the MSE of the corresponding FNN model. Additionally, it becomes apparent that the LSTM network training is computationally more expensive, in this case being roughly five times longer than FNN training, presumably because of the more complicated backpropagation through time algorithm. Interestingly, the FNN models presented above consist of far more trainable parameters, with one FNN having $\approx 5 \times 10^4$ tunable parameters compared to $\approx 5 \times 10^3$ for the LSTM network.

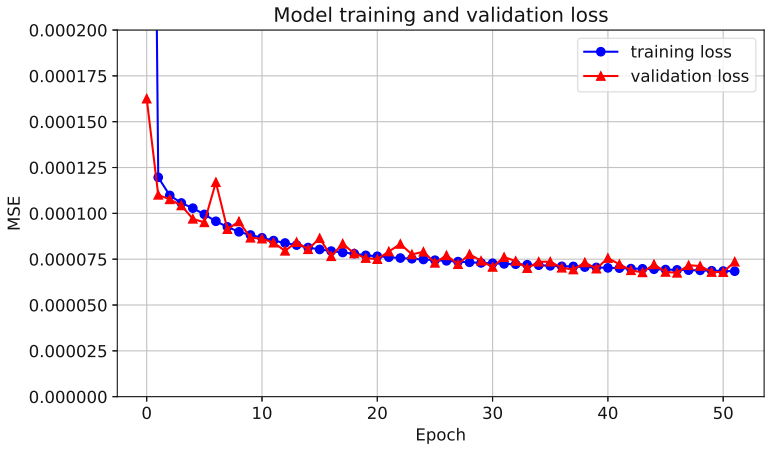
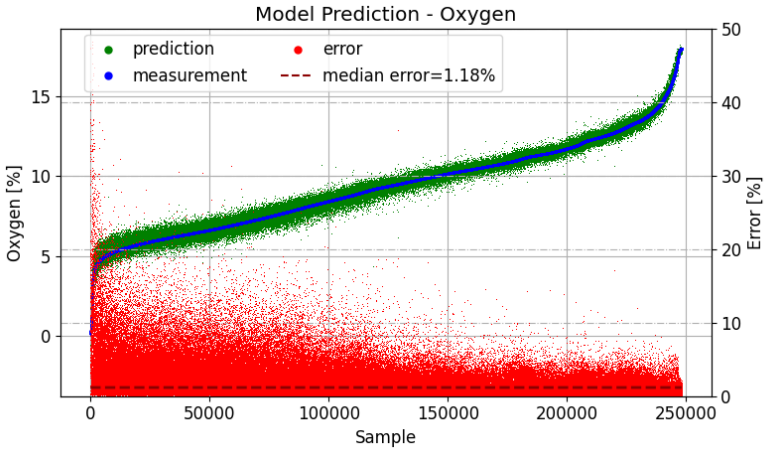


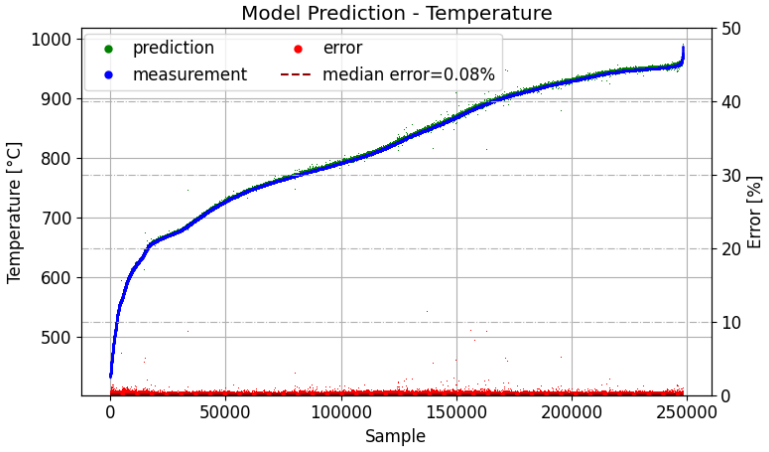
Figure 28: Trend of the training error (blue) and validation error (red) for the FNN model of plant A. The lowest validation error was reached at the 47th epoch with a loss of $\mathrm{MSE} = 6.756 \times 10^{-5}$.

Figure 29 depicts the prediction plots for the LSTM model. Due to the errors being considerably lower and not covering the other data points only the plots with errors are shown. In this figure two features stand out: First, as indicated by the validation MSE, the relative prediction errors are significantly lower than the ones from the FNN model.

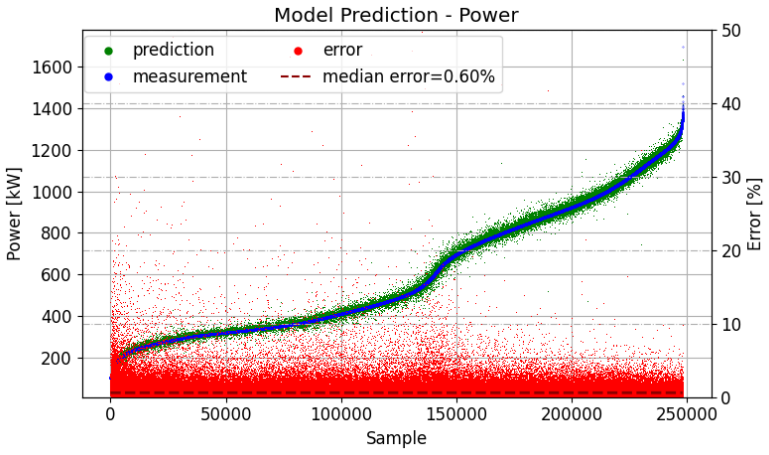
Second, there is a substantial accuracy difference between the individual model outputs,



(a) Oxygen predictions.



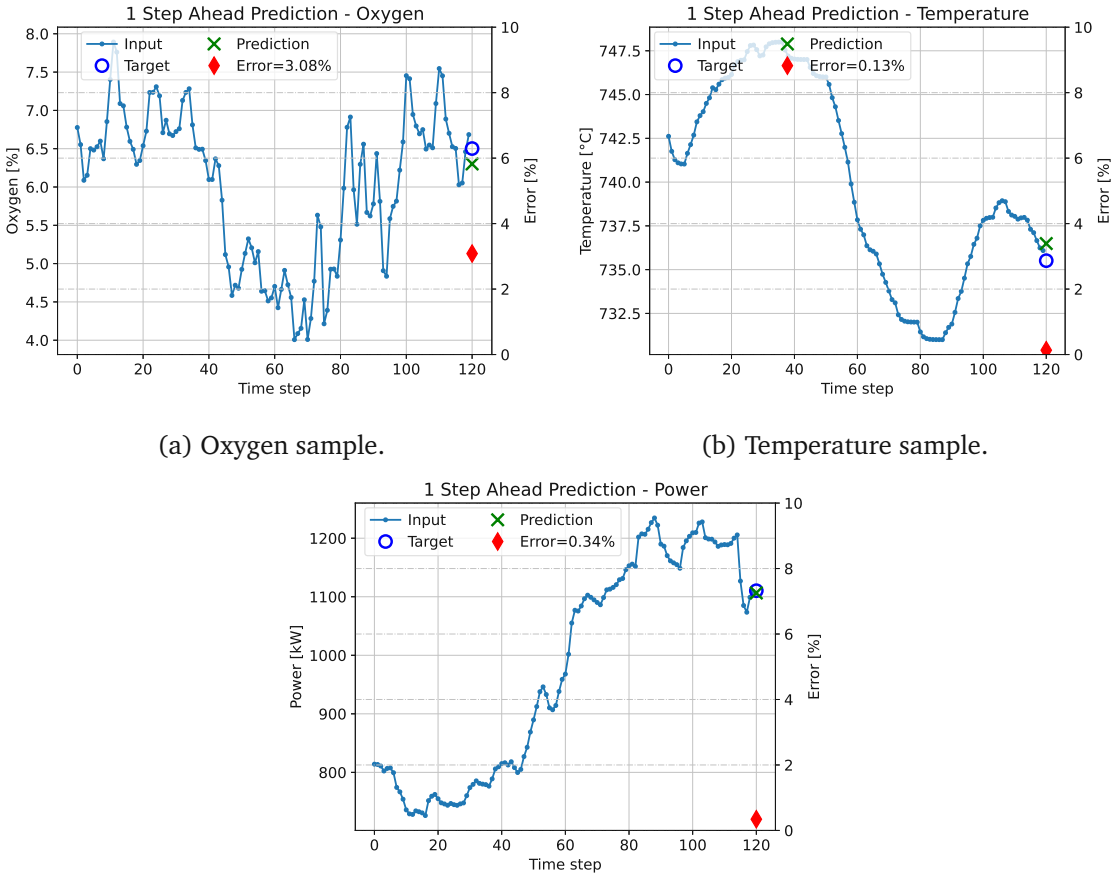
(b) Temperature predictions.



(c) Power predictions.

Figure 29: Plant A test data predictions from the LSTM model. For every sample the model prediction (green), the true value (blue), and the relative error of the prediction (red) is depicted. Additionally, the samples are arranged according to their true value.

with the oxygen content predictions deviating the most and the temperature predictions being the most precise. The reason for this observation can be found when taking a look at a few exemplary samples depicted in Fig. 30. The plots show the one-step-ahead prediction of the model compared to the true target and the measurement sequence of the previous 120 time steps (sampled at $30\,\mathrm{s}$) for every model output. As can be seen, the measurement variation from one time step to the next is vastly different for the three outputs, especially when considering their scale. In the case of the oxygen measurements the data points vary by up to half a percentage point from one time step to the next, resulting in the jagged graph in Fig. 30(a). The temperature graph in Fig. 30(b) on the other hand is much smoother with very little variation between the time steps and a slower temporal behavior. Naturally this makes the subsequent temperature value easier to predict. The power graph is less smooth than the temperature graph but still has smaller variations than the oxygen values.



(a) Oxygen sample.

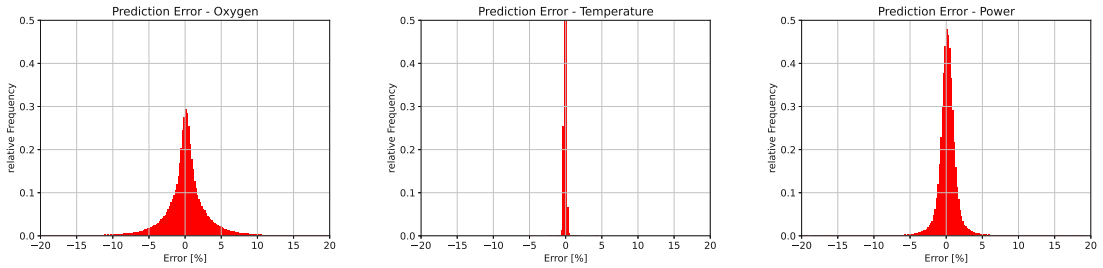
(b) Temperature sample.

(c) Power sample.

Figure 30: Plant A test samples for the oxygen content (a), temperature (b), and power predictions (c). Every plot shows the past measurement sequence the model gets as an input (120 time steps covering one hour), the true value of the subsequent time step (target), the model prediction, and the relative model error (right axis).

Figure 31 displays the error histograms corresponding to Fig. 29. The oxygen content

predictions are the least accurate, with a median error of $1.18\,\%$ and a noticeable drop in accuracy for lower process values. Still, most errors are below $5\,\%$, which is a significant improvement to the predictions of the FNN model, which, as a reminder, has a median error of $5.52\,\%$.
Due to the reasons described above, the LSTM model predicts the temperature values very well, with a median error of only $0.08\,\%$, compared to $3.00\,\%$ for the FNN model. The histogram in Fig. 31(b) is practically a single spike.
The power prediction saw the biggest increase in accuracy with a median error of $0.60\,\%$ compared to $8.48\,\%$ for the MLP model. Almost all of the errors are smaller than $5\,\%$ as can be seen in the histogram of Fig. 31(c).



(a) Oxygen prediction errors.

(b) Temperature prediction errors.

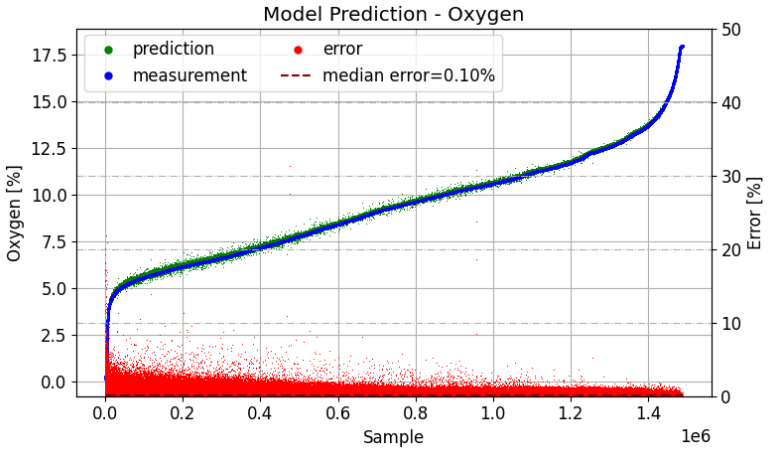
(c) Power prediction errors.

Figure 31: Histograms showing the relative frequency of errors for the oxygen, temperature, and power predictions for the LSTM model of plant A.

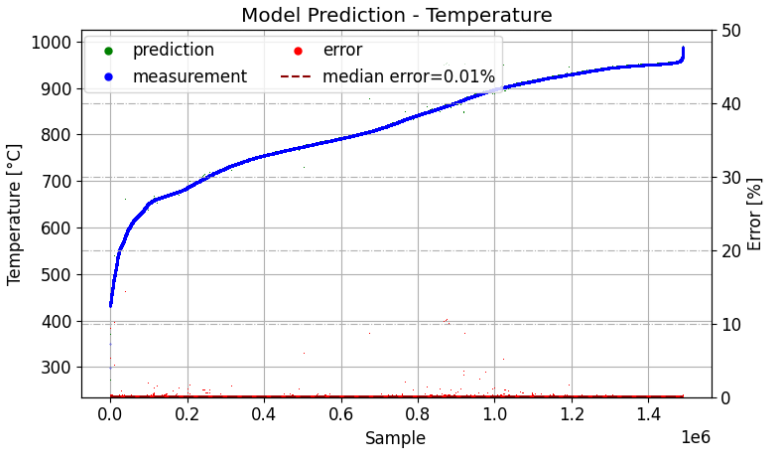
After this successful first iteration of the LSTM model the next step was to investigate how the model performance is affected by different sample configurations. The first change to the configuration consisted of reducing the sampling time of the data. This should improve the model accuracy as the variation between time steps decreases.

Thus, the sampling time was reduced from $30\,\mathrm{s}$ to $5\,\mathrm{s}$ resulting in six times the number samples, since the dataset still covers one year of plant operations. For this new configuration the model training required 12 epochs and a duration of 158 minutes. Since the dataset is six times larger, one training iteration was more time expensive, but, as fewer iterations are needed, the full model training required roughly the same amount of time as the previous model. During training the model achieved a minimal validation loss of $\mathrm{MSE} = 1.0034 \times 10^{-6}$ which is one order of magnitude lower than the previous model's loss. Thus, a shorter sample time indeed improves the modeling accuracy without increasing the computational expense of the model training.

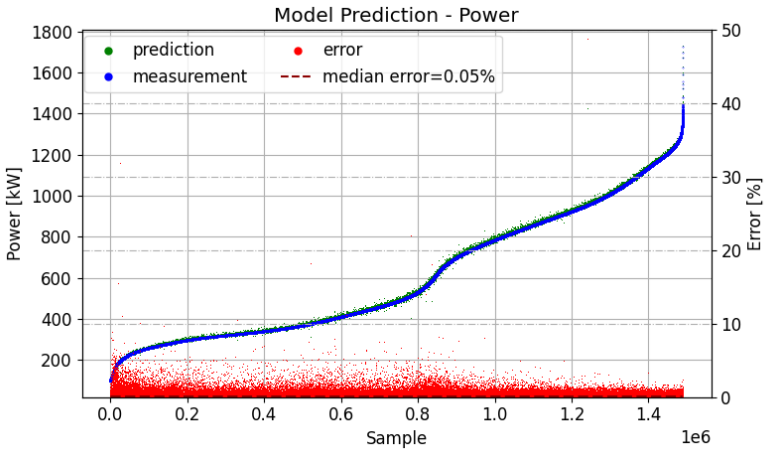
Figure 32 shows the test data predictions for this model. For example, the median error for the oxygen predictions has decreased to $0.10\,\%$, the oxygen predictions now being almost as accurate as the temperature predictions of the previous model. The accuracy of the temperature and power predictions has increased as well, with a median error of



(a) Oxygen predictions.



(b) Temperature predictions.



(c) Power predictions.

Figure 32: Plant A test data predictions from the LSTM model using a sample time of $5\,\mathrm{s}$. For every sample the model prediction (green), the true value (blue), and the relative error of the prediction (red) is depicted. Additionally, the samples are arranged according to their true value.

$0.01\,\%$ and $0.05\,\%$, respectively. The temperature predictions in particular have such a high accuracy that the predictions in Fig. 32 are practically indistinguishable from the true value.

Another possible sample reconfiguration is to reduce the number of time steps in the input sample, shortening the time horizon the model has access to. This should improve training speed as the backpropagation through time algorithm needs to make fewer steps. However, making the input sample too short negatively impacts the model performance as less information of the past dynamics of the system is provided. The ideal sample configuration thus allows for an increase of training speed while not reducing the prediction accuracy.

For dynamic models the number of time steps (i.e. past states) in the input sample is called model order m_m and plays an important role when describing a dynamic system [13]. Importantly, the optimal value for the model order depends on the order of the original system m_o. In order to achieve optimal modeling of a dynamical system, it has been shown that the model order needs to be sufficiently high. For linear systems the model order should be selected to be equal to the system order $m_m = m_o$. Nonlinear systems require a higher model order, resulting in $m_m > m_o$. The Takens' embedding theorem for forced systems [7] states that highly nonlinear systems require a model order of $m_m \geq 2m_o + 1$. Thus, for any system that is not highly nonlinear the optimal model order lies in the interval

$$m_o \leq m_m \leq 2m_o + 1\,, \tag{39}$$

with m_m being the model order and m_o being the order of the original system [13]. Choosing a model order higher than the optimal model order does not yield significant improvements for the modeling accuracy.
Unfortunately, the order of the original system is seldom known, which means that the optimal model order has to be determined by training several models with different orders and evaluating their performance. When comparing the loss of models with increasing model order, the optimal value for m_m should be marked by a significant drop of the loss after which it stays roughly constant for even higher order models [13].

This procedure was performed for the LSTM model of plant A to determine the optimal model order for the combustion process. In this case, the model order (i.e. number of time steps in the input sample) was varied from 1 to 50 with increasing step sizes, and a sampling time of $5\,\mathrm{s}$. The resulting validation loss was evaluated for every model and compared to the others. Additionally, the training duration per epoch was assessed for every model in order to provide a measure of the computational cost of the model.
Figure 33 depicts this evaluation for the different models. Interestingly, the loss already drops off for the second order model indicating that the nonlinear combustion process is a first order system. For higher order models the MSE stays approximately at a value of $\approx 1 \times 10^{-6}$, yielding no further modeling improvement. The training duration, however,

steadily increases for higher order models since they are computationally more expensive.

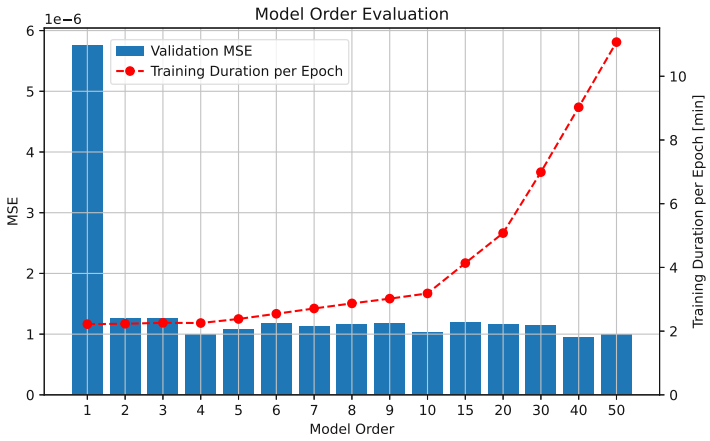


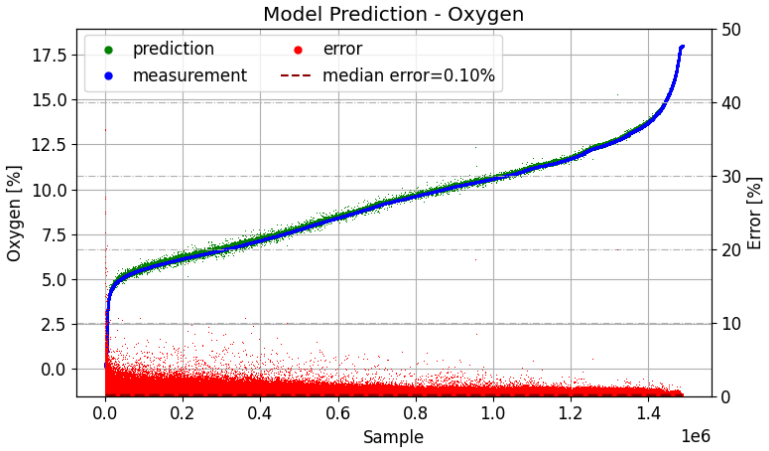
Figure 33: Model order evaluation for plant A.

Out of the tested models the fourth order model was selected as the optimal model order for plant A, as it has the lowest validation loss but still has the same computational expense as the second order model. Thus, the optimal sample configuration was determined to have a sampling time of $5\,\mathrm{s}$ and consist of four time steps of previous data. Figure 34 shows the test predictions for the fourth order model which has practically the same accuracy as the previous model in Fig. 32, despite its training being significantly faster.

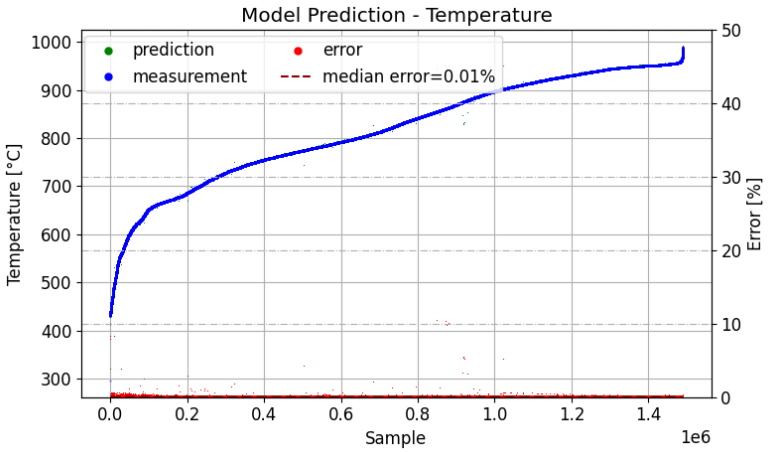
Figure 35 depicts a short section of testing data where the true measurement trend is compared to the one-step-ahead predictions for the respective plant outputs. In all three cases, the predictions (dashed green graph) almost perfectly reflect the real plant behavior with only minimal deviations. Thus, for the one-step-ahead prediction this model configuration works very well.

Finally, the plant model can be evaluated even further. Due to the structure of the one-step-ahead predictor the model can be used to make predictions several steps into the future by using the predicted outputs as the newest time step of the input sample for the subsequent prediction. Due to its simplicity, and the fact that the one-step-ahead prediction model can be reused and no new model needs to be trained this so-called iterated method is the oldest and most common technique for multi-step-ahead predictions [16].

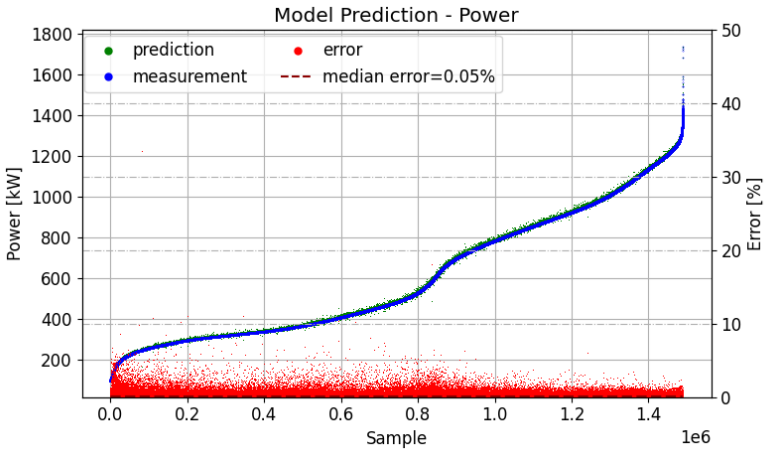
However, using predictions as new inputs causes the prediction error to propagate with every step, resulting in increasingly large errors especially for long horizon multi-step-ahead predictions. Furthermore, the multi-step-ahead predictor might struggle to represent long term dependencies of the system because it was trained to merely make (short term) one-step-ahead predictions [16].



(a) Oxygen predictions.

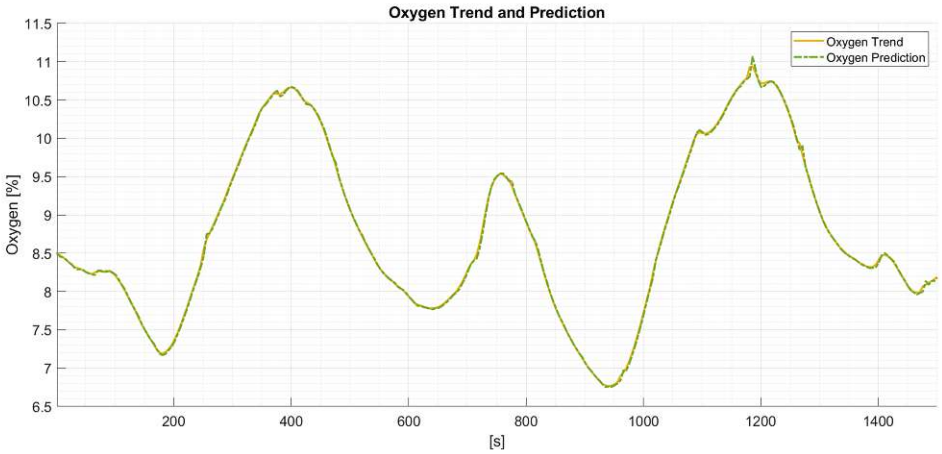


(b) Temperature predictions.

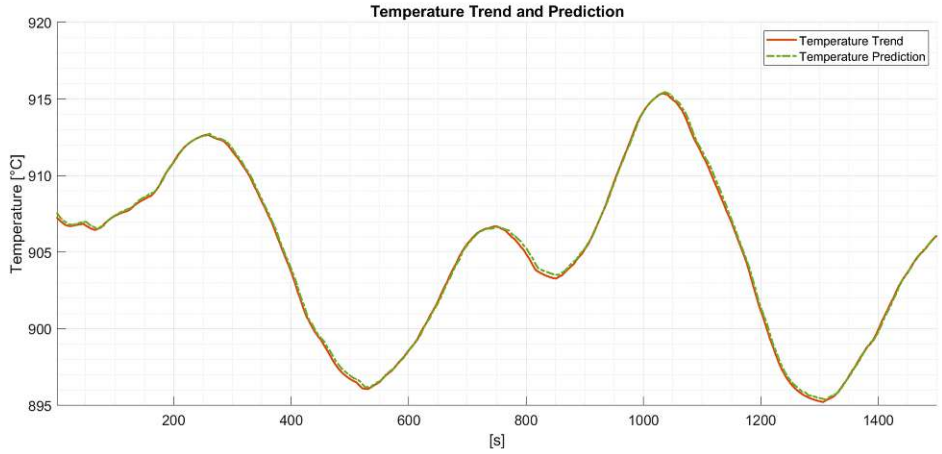


(c) Power predictions.

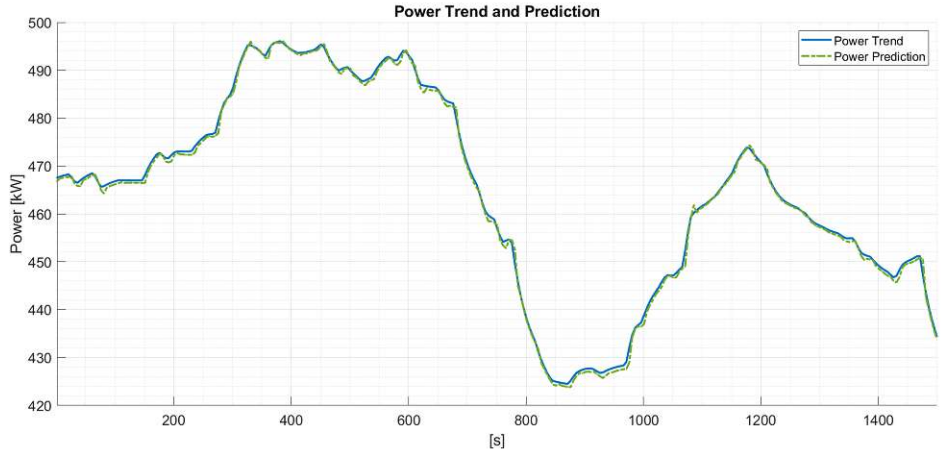
Figure 34: Plant A test data predictions from the fourth order LSTM model. For every sample the model prediction (green), the true value (blue), and the relative error of the prediction (red) is depicted. Additionally, the samples are arranged according to their true value.



(a) Oxygen trend and one-step-ahead predictions.



(b) Temperature trend and one-step-ahead predictions.



(c) Power trend and one-step-ahead predictions.

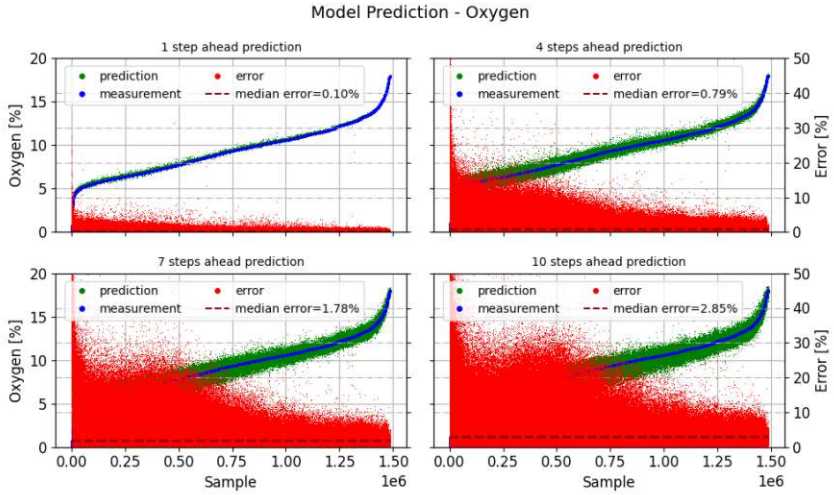
Figure 35: True measurements and prediction trends for plant A.

Still, due to its advantages the iterated method for multi-step-ahead predictions was used in combination with the already trained model of plant A. For the evaluation of the multi-step predictions the model was used to predict several steps into the future, using the previous prediction as newest input step. To be more precise, since the model only produces predictions for the plant outputs (oxygen, temperature, and power) and an input sample consists both of plant inputs and plant outputs, the new sample was constructed from the measured plant inputs and the predicted plant outputs. This is the way such an iterated multi-step-ahead prediction model would be integrated into a productive control setup, see for example [20]. Every prediction step is then compared to the true measurement value at that time.

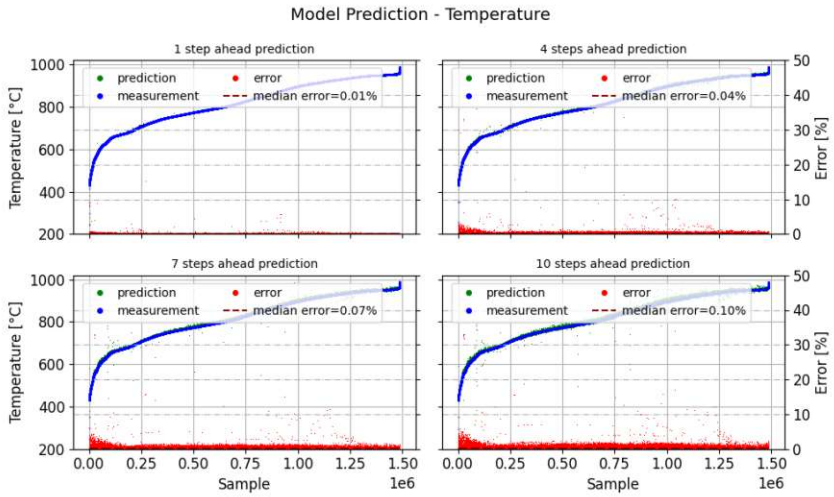
Figure 36 depicts these predictions for the three outputs for the first, fourth, seventh, and tenth prediction step, respectively. As expected, the prediction accuracy decreases for predictions that are more distant in the future. The oxygen prediction in Fig. 36(a), for instance, has a median error of $0.10\,\%$ for the first prediction step. However, the error steadily increases for further prediction steps. At the tenth prediction iteration the median error for all test samples is $2.85\,\%$, with high deviations from the true value especially for process conditions with low oxygen content.
The temperature predictions in Fig. 36(b) again show the highest accuracy with a median error of just $0.10\,\%$ for the tenth prediction into the future.
The iterated power predictions show the same behavior as the other outputs, with rising errors for subsequent predictions. For the tenth iteration step, which is a prediction $50\,\mathrm{s}$ into the future, the median error stands at $1.51\,\%$, with several cases where the errors are above $10\,\%$.

Figure 37 visualizes the difference between iterated predictions and the true measurement even better. The plots show a short section of true signals and the corresponding multi-step predictions (green). In this case, the one-step-ahead model was iterated for 60 steps ($5\,\mathrm{min}$), after which the next iteration series was started, resetting the prediction by using true plant data for the first iteration step. This is the reason for the discontinuities in the prediction graph, which occur between the (mostly far off) 60th prediction step and the (still very accurate) first prediction step of the new iteration series.

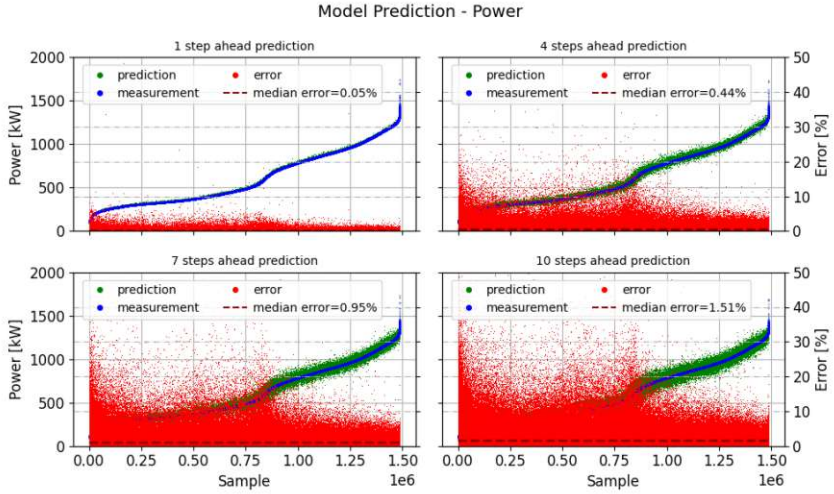
The figure illustrates the effect of error propagation very well, which is one of the main problems of the iterated method. This can be seen by the fact that for most iteration series the predictions drift off after a certain number of steps and do nöt recover, leading to a large deviation from the true signal after the 60th step.
In some cases the qualitative plant behavior is modeled quite well, even though the exact values are not entirely correct. For example in Fig. 37(b) in the case of the prediction series starting at $2400\,\mathrm{s}$, where the temperature minimum at $2550\,\mathrm{s}$ and the subsequent rise in



(a) Oxygen multi-step-ahead predictions.

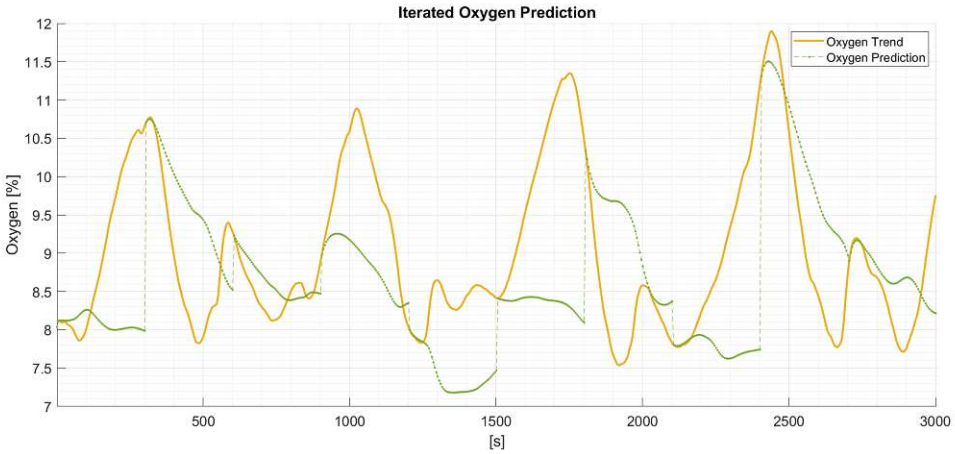


(b) Temperature multi-step-ahead predictions.

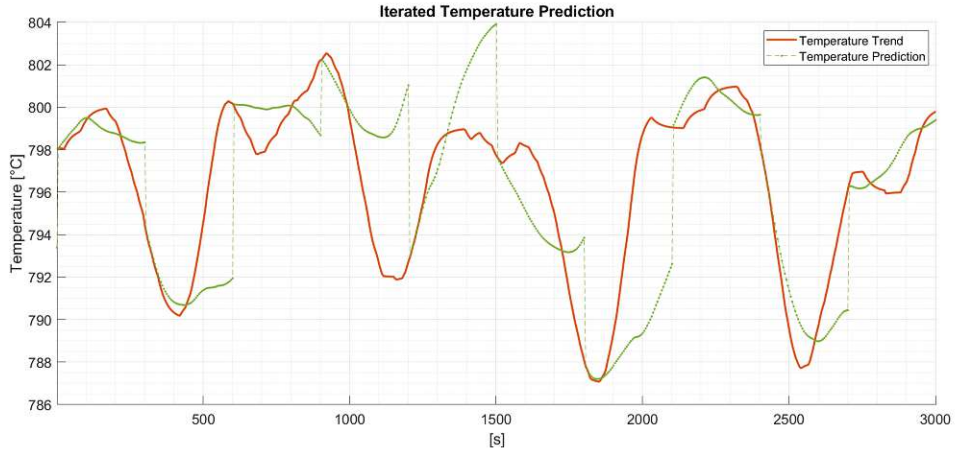


(c) Power multi-step-ahead predictions.

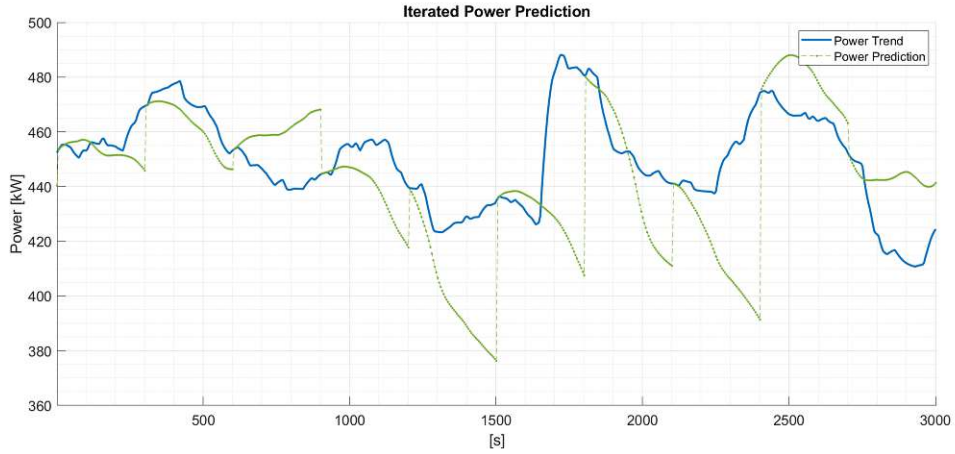
Figure 36: Multi-step-ahead predictions using the iterated one-step-ahead model for plant A.



(a) Oxygen trend and iterated predictions.



(b) Temperature trend and iterated predictions.



(c) Power trend and iterated predictions.

Figure 37: Multi-step-ahead predictions (green) compared to the true trend. Starting from true measurement data, the single-step predictor is iterated for 60 steps, using previous predictions for the input. After 60 iteration steps the prediction is reset and the original data is used for the next step.

temperature is predicted. Another case can be found in Fig. 37(c) starting at $300\,\mathrm{s}$, where the model predicts the initial rise in power until $400\,\mathrm{s}$ after which the power decreases again. Here, the final prediction value is quite close to the true value.
However, these cases generally are an exception to the majority of predictions which drift off sooner or later.

In general, the initial trajectory of the predictions strongly depends on the true signal's slope or tangent at the starting point. This can be seen, for instance, in Fig. 37(a) at $1500\,\mathrm{s}$, where the iteration series starts right in a local minimum and thus stays roughly horizontal, although the true oxygen content rises by more than two percentage points. In cases where the slope of the signal stays similar to the slope at the starting point the predictions mostly stay closer to the true trend for longer.

4.2.2 Plant B

The optimizations and findings obtained during the different iterations of plant A models were used for the creation of the LSTM model of plant B. Thus, the optimal model order was determined right from the beginning, using a training dataset with a sampling time of $5\,\mathrm{s}$.

Figure 38 shows the model order analysis for plant B. Again, the validation error drops off for the second order model, confirming the assumption that the combustion process is a nonlinear first order system. After the second order model, the validation loss roughly stays in the range of $[0.2 \times 10^{-6}, 0.4 \times 10^{-6}]$ which is less than half the loss of the best model for plant A. Once more, the relation between higher model orders and longer training time is apparent.

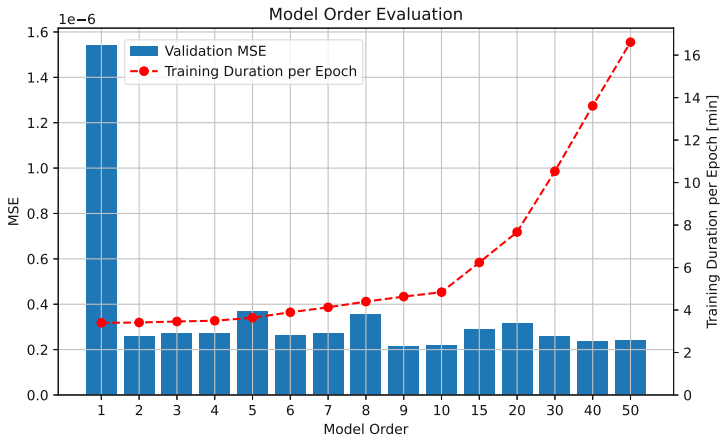
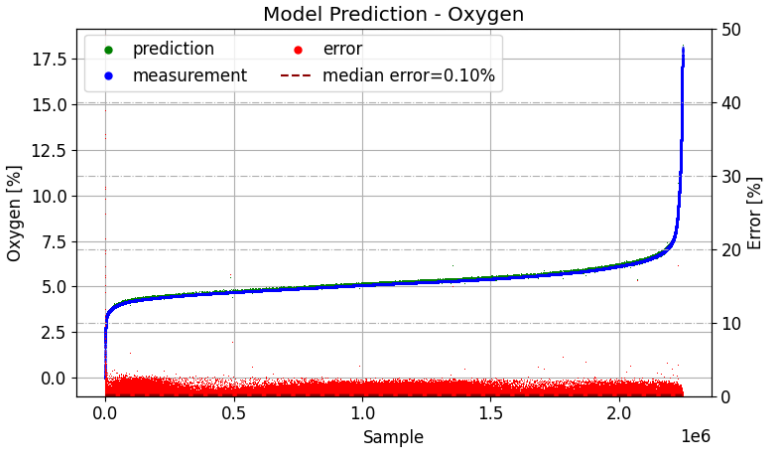


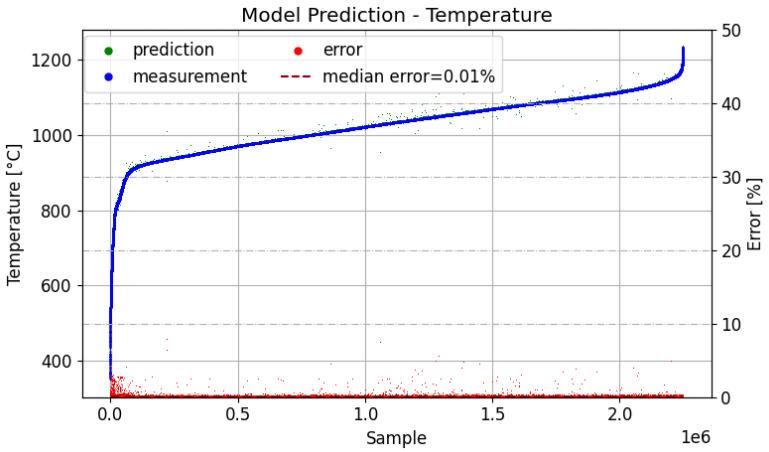
Figure 38: Model order evaluation for plant B.

In this case, the second order model was used for the further model analysis as it had the best combination of low validation loss and fast training time. Training for this model required 20 epochs, resulting in a total training duration of just over $68\,\mathrm{min}$ and reaching a minimal validation error of $\mathrm{MSE} = 2.722 \times 10^{-7}$.

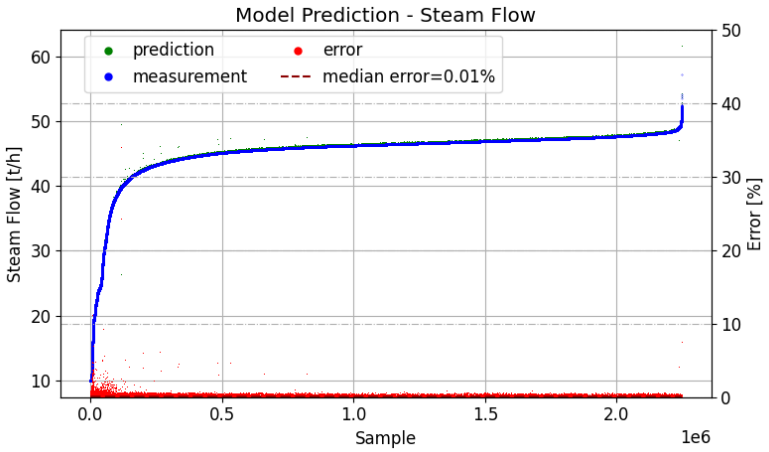
Figure 39 shows this model's predictions for the test dataset from plant B. As in the case of plant A, the LSTM model has a much higher accuracy than the static FNN model of plant B. Most output predictions have such an accuracy that they are indistinguishable from the true measurement graphs in the plots. Out of the three outputs the oxygen content prediction has the worst performance, with at median error of $0.1\,\%$. However, that still lies far below the corresponding FNN error of $6.49\,\%$.
Both the temperature predictions and the steam flow predictions have a median error of $0.01\,\%$, which again is far below their FNN counterparts with an median error of $3.53\,\%$ and $1.86\,\%$, respectively. Both plots have slightly higher errors at the bottom end of the process variable spectrum where the sample density is very low.



(a) Oxygen predictions.

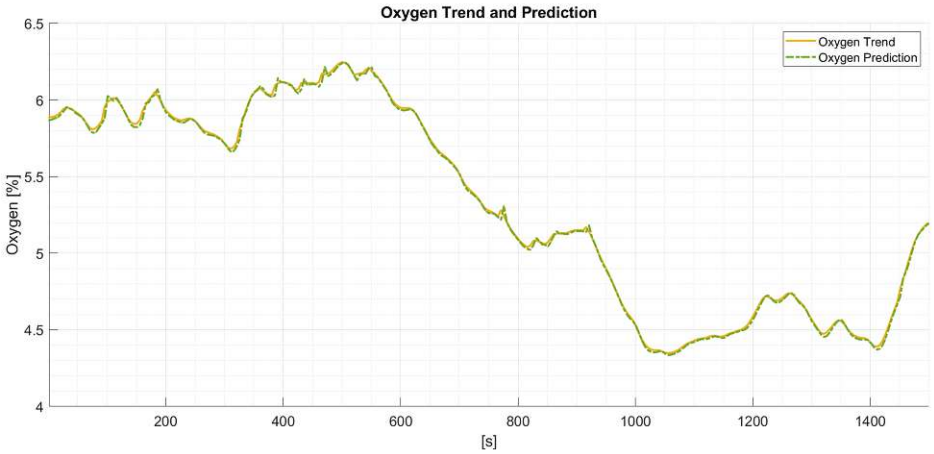


(b) Temperature predictions.

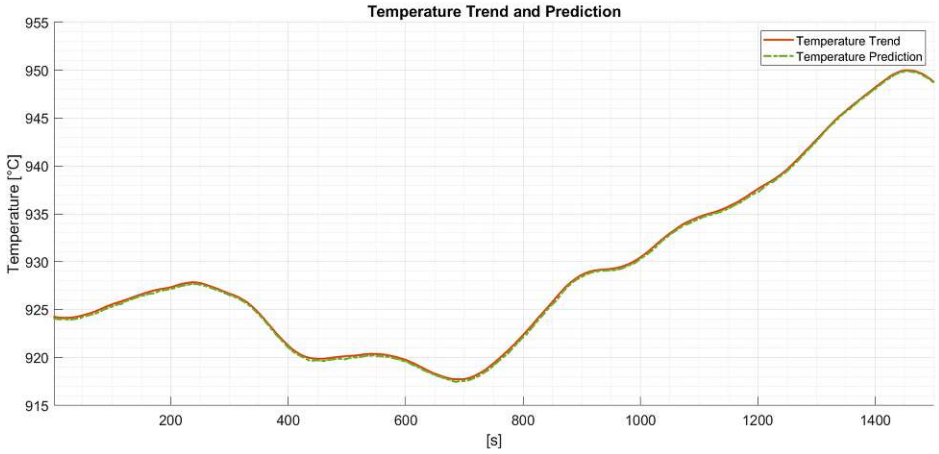


(c) Steam flow predictions.

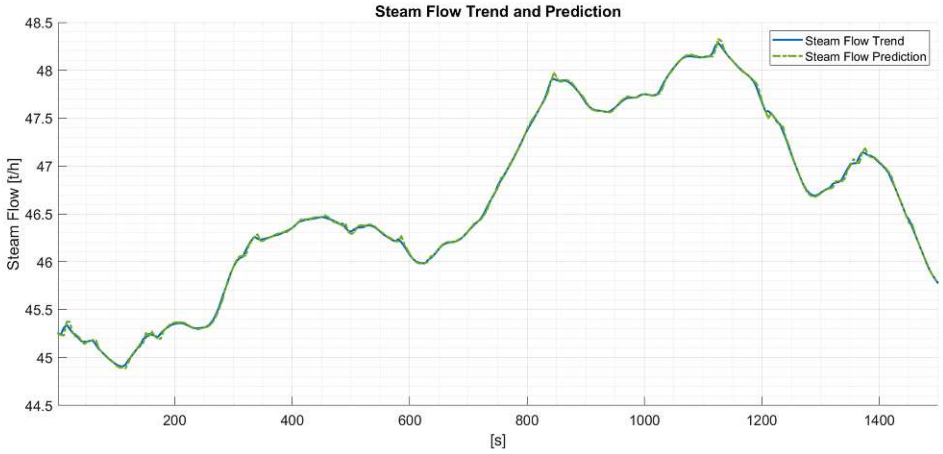
Figure 39: Test data predictions from the second order LSTM model for plant B. For every sample the model prediction (green), the true value (blue), and the relative error of the prediction (red) is depicted. Additionally, the samples are arranged according to their true value.



(a) Oxygen trend and one-step-ahead predictions.



(b) Temperature trend and one-step-ahead predictions.



(c) Steam flow trend and one-step-ahead predictions.

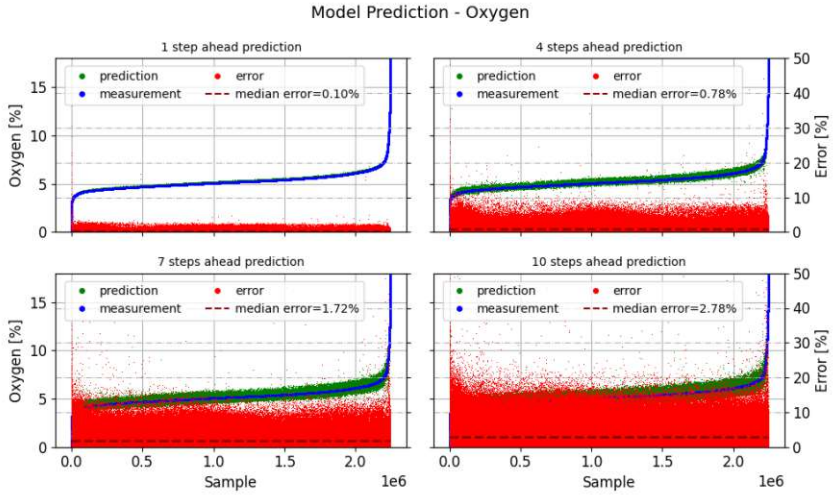
Figure 40: True measurements and prediction trends for plant B.

To better illustrate of the model's accuracy, Fig. 40 depicts one exemplary section of the combustion process of plant B and the corresponding one-step-ahead model prediction (green). Again, the one-step-ahead predictions very closely match the true signal.

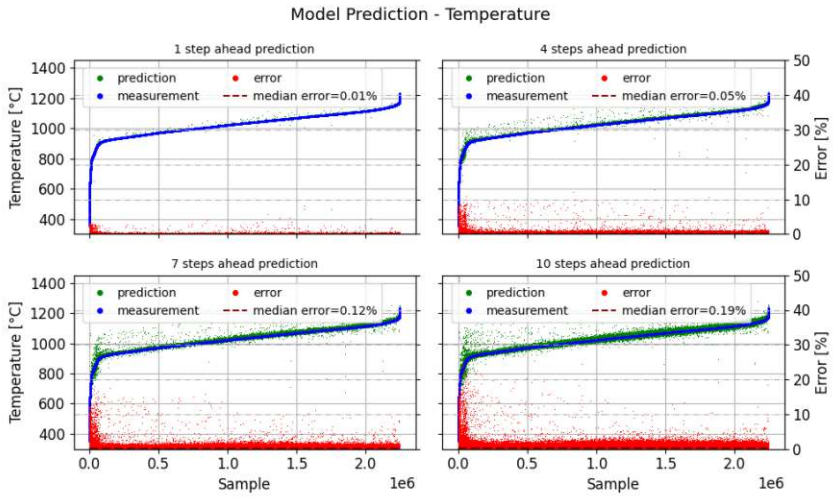
Next, the one-step-ahead model was used to create multi-step-ahead predictions via the iterated method. Figure 41 shows the test set predictions for the first, fourth, seventh and tenth iteration step. Again, the error increases for higher iterations due to error propagation. Naturally, due to the single-step predictions for the oxygen content being least accurate, the oxygen forecast in Fig. 41(a) exhibits the biggest error. After the tenth iteration the median error amounts to $2.78\,\%$ with many errors being larger than $10\,\%$.
The iterated temperature predictions stay surprisingly accurate even for higher iterations. At the tenth prediction step the median error lies at $0.19\,\%$. However, at the lower end of the temperature range, where, as already observed, the single-step model is less accurate, the prediction errors are larger. Additionally, a few outliers are scattered across the spectrum but almost all errors are smaller than $5\,\%$ even at the tenth prediction step.
Despite being similarly accurate in the single-step prediction, the iterated steam flow predictions in Fig. 41(c) perform slightly worse than the temperature predictions. After the tenth prediction step the median error is $0.42\,\%$. Presumably, this is due to the trend of the steam flow being less smooth, its value changing more quickly. As in the case of the temperature predictions, for low steam flow process conditions the predictions are less accurate.

Figure 42 displays a section of the combustion process to illustrate the multi-step-ahead predictions for the three plant outputs. Again, the single-step model is iterated for one prediction series of 60 steps before it is reset and a new series starts.
Generally, the plots show the same phenomena which were identified for the multi-step prediction of plant A: The prediction series mostly drift off after a few steps leading to large deviations at the end of the series. The trajectory of the predictions depends strongly on the slope at the start of the prediction. Still, in some cases the true trend is predicted quite well.

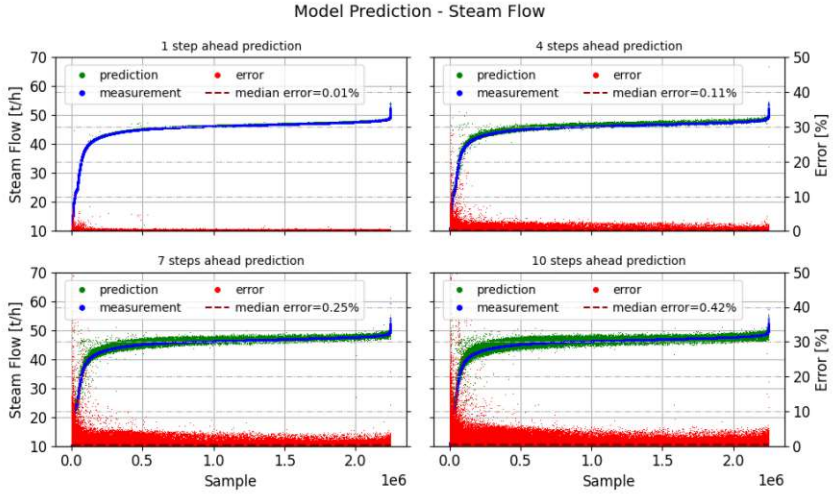
What stands out is that in Fig. 42(a) the oxygen trend is very jagged or uneven. While the model predicts these small variations quite accurately in the one-step-ahead case (see Fig. 40(a)) here the model can't reproduce the true system behavior as accurately and the predictions are smoother.
Interestingly, the iterated temperature predictions in this particular section mostly tend downwards. At the same time, the steam mass flow predictions predominately tend upwards, which is a bit contradictory as usually the temperature positively correlates with the steam flow. One explanation for this phenomenon is that the plant was experiencing, as far as the model is concerned, slightly unusual or abnormal conditions as the data was recorded. Presumably, during these conditions the recirculated air flow was increased,



(a) Oxygen multi-step-ahead predictions.

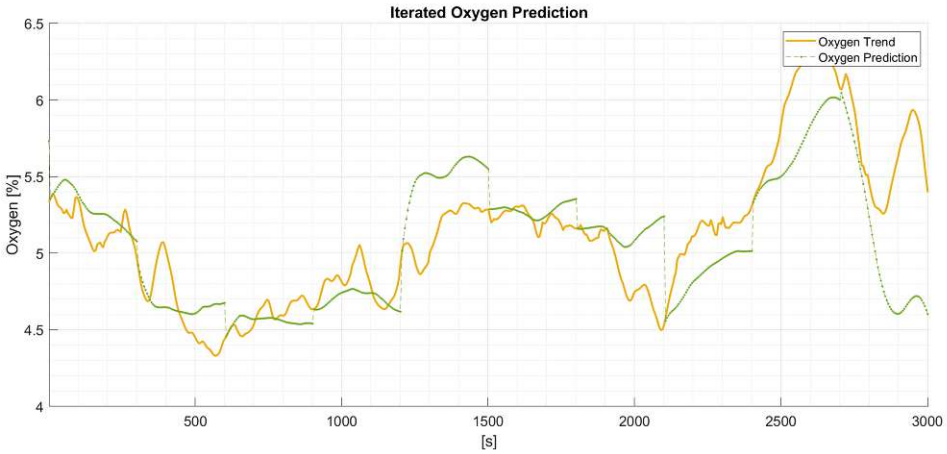


(b) Temperature multi-step-ahead predictions.

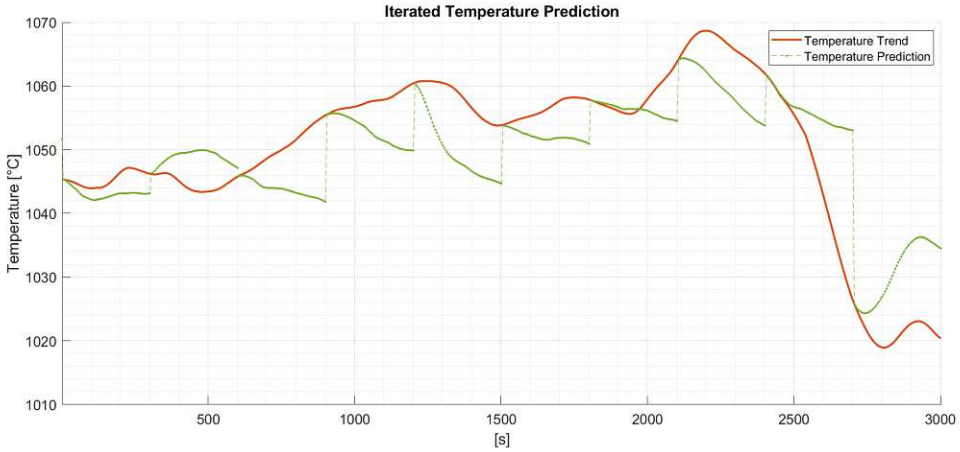


(c) Steam flow multi-step-ahead predictions.

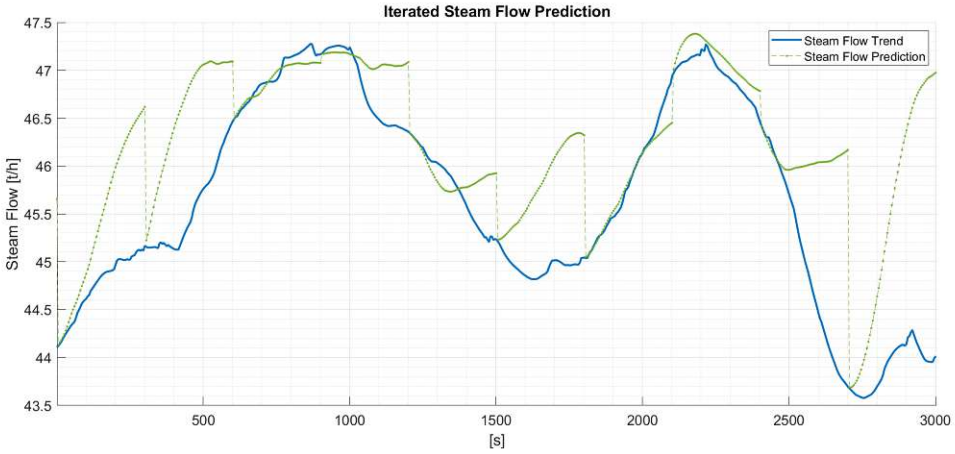
Figure 41: Multi-step-ahead predictions using the iterated one-step-ahead model for plant B.



(a) Oxygen trend and iterated predictions.



(b) Temperature trend and iterated predictions.



(c) Steam flow trend and iterated predictions.

Figure 42: Multi-step-ahead predictions (green) compared to the true trend for plant B. Starting from true measurement data, the single-step predictor is iterated for 60 steps, using the previous predictions for the input. After 60 iteration steps the prediction is reset and the original data is used for the next step.

which under normal circumstances reduces the combustion chamber temperature, and the primary air flow was increased, which normally causes the power, in this case the steam mass flow, to rise. Thus, the model, expecting the process behavior it learned during training and not the abnormal conditions the plant apparently experienced, falsely predicts a reduction in temperature and rise in power. The reason for abnormal process conditions could, for instance, be changing fuel properties such as its moisture or composition. In that case, adding these properties as input signals should help to improve the prediction accuracy of the model.

All in all, the dynamical LSTM models far outperform their static FNN counterparts. Even the first model iteration showed a large increase in accuracy compared to the best FNN models. By tweaking the input sample configuration the performance could be improved further.
Decreasing the sampling time to $5\,\mathrm{s}$ improved the model accuracy without increasing the computational expense. Further, by reducing the length of the input samples and thus the model order to a certain point, the training duration could be decreased without losing modeling accuracy. Due to the combustion process being a nonlinear first order system, it can be modeled very accurately by a predictor with as few as two input time steps.
However, while the trained models work very well for one-step-ahead prediction, for multi-step-ahead predictions the models struggle to stay accurate for more than a few steps before the predictions mostly drift off.

5 Conclusion

In this thesis, real world process data was used to train neural network models of industrial biomass combustion plants. Specifically, for two plants both a static feedforward neural network (FNN) in the form of a multilayer perceptron and a dynamic recurrent neural network in the form of a Long Short-Term Memory (LSTM) was developed and tested.
For the FNN models it was found that optimal results were achieved when using a full year of process data for the modeling dataset, in order to cover the entire process range of the plant. However, even with this configuration the static model still produced unsatisfactorily large prediction errors, with many cases where the predictions were deviating by more than $10\,\%$. This large error derives from the fact that the dynamical combustion process cannot be predicted more accurately by a static model.
Thus, for more accurate predictions a dynamic model is required which in this thesis consisted of LSTM networks. Indeed, it could be shown that the prediction accuracy of the dynamic LSTM models was far higher. In order to obtain the optimal model order, optimizing the models both in terms of computational expense and accuracy, different input sample configurations were tested. As a result it was shown that the combustion process which is a nonlinear first order system can be accurately modeled using a second order model. These models achieved very accurate one-step-ahead predictions at a comparatively low training time.
To further evaluate their performance the use of these models as iterated multi-step-ahead predictors was analyzed. Here, it was shown that, while in some cases plant dynamics are forecast quite well, mostly the predictions drift off after a few iteration steps. Therefore, in their present state these models cannot be used in any kind of model-based control integration operating over a longer time horizon.

In the future, several steps can be made to improve the long horizon accuracy of the plant models in order to effectively use them in control setups. First, a quick and simple measure would be to provide the model with additional inputs describing external plant conditions, such as the fuel moisture for example. These variables would provide additional information for the model which should improve prediction accuracy because of the plant state being more thoroughly described.
Second, other multi-step-ahead prediction techniques might provide more accurate predictions than the iterated method which is hindered by the propagation of the prediction error. Alternatives could be the direct method, where several models make predictions in parallel, each for their own time step, or the multiple-input multiple-input (MIMO) approach, where the model predicts a series of future time steps in the form of a multidimensional output [16]. Taieb et al. also propose a intermediate approach (MISMO) which adapts the MIMO method to have more adaptive prediction horizon and to incorporate dependencies on past predictions [16].

Furthermore, the model architecture itself should be reviewed. It has to be tested whether other forms of dynamical neural networks are better suited for long term ahead plant predictions and their implementation into control setups. Gaussian process (GP) models, for instance, which can be regarded as a form of radial basis function network, are commonly used to model nonlinear dynamic systems, see for example [12, 13, 40]. The advantage of GP models is that, in addition to the prediction output, they provide an estimate of the output's variance, which can be interpreted as the confidence or uncertainty of the model [10]. However, the drawback of using GP models is that they have a high computational load for large datasets, because of the training algorithm scaling with $\mathcal{O}(n^3)$, with n being the number of training samples [10]. Recently, to circumvent this problem it has been proposed to use GP models performing online learning by continuously having them update their model with information from current samples, which dramatically reduces the number of training samples needed [48].
Another recent alternative to RNNs is the use of so-called neural ordinary differential equations (ODE) [34], where, instead of specifying a discrete hidden layer, only the derivative of the hidden state is parametrized. This allows the models to adapt their evaluation strategy to each input, making them more versatile than conventional RNNs, for example when predicting a time series with irregular time points [34].
Additionally, ensemble learning methods which combine multiple learning algorithms and models can be used to obtain better performances [37]. Examples for such ensemble methods include Bagging, Boosting, or Stacking, which improve the final generalization of the model by averaging the outputs of the different submodels [29]. In their work Wu et al. employed ensemble regression to RNN models to increase the prediction accuracy of nonlinear process dynamics [37]. In all these cases, the eventual model accuracy should be set in relation to the computational cost of training the model in order to have a transparent comparison.

Regardless of the exact realization of the model, in order to use the model in any kind of model-based control setup, such as predictive control or internal model control, a few requirements have to be met from the perspective of the control system. Most importantly, the stability and robustness of the model must be guaranteed at all times. Stability or robustness in this case means that small perturbations of the inputs do not result in catastrophic output changes, and that, for any possible input sequence, the outputs always stay bounded and in the range of physically-meaningful values [46]. To test the model's robustness ideally the model output is evaluated for a sufficiently large amount of perturbed input samples. Alternatively, the robustness is guaranteed if the model can be shown to have Incremental Input-to-State Stability (δISS) which can under certain conditions be enforced by adding a regularization-like term to the model's loss function, see [43, 46].
However, even when the model satisfies these requirements it is not necessarily guaranteed to be usable for model-based control purposes. In the case of neural network model predictive control, for instance, which additionally requires nominal closed-loop stability,

proving this for a robust model may still be challenging, as even a very accurate plant model cannot predict output disturbances which cause problems for the system's stability [46].
For internal model control (IMC) setups, different steps can be taken to improve the closed-loop performance. Gregorčič and Lightbody, using a GP model for their IMC approach, improved the closed-loop stability by introducing a constraint on the controller based on the GP model's uncertainty [20]. Blonbou et al., on the other hand, using a regular FNN model, deployed an adaptive control algorithm that continuously adjusts the controller parameters during plant operation [5]. This form of adaptive control helps to stabilize the closed-loop performance because of the constant tuning of the controller parameters. A common use for adaptive control strategies is to apply them to neural network controllers which are generally simply inverted plant models, and to continuously tune the controller with some kind of backpropagation algorithm [1].

Finally, it has to be investigated how to integrate the neural network model into an existing control setup on a technical level. In the vast majority of cases, industrial process and plants are monitored and controlled by programmable logic controllers (PLCs) which are industrial computers that are specialized to work reliably and in harsh environments. However, the task of implementing ANNs into PLCs is hindered by the fact that PLCs are limited by both their hardware, consisting of a comparably low power CPU and no additional GPU, and their software, which is standardized by IEC 61131-3 [52].
Thus, the easiest way to incorporate the neural network model is using external computing devices for training and during online prediction. This allows external devices such as an industrial PC on-site or a remote server to perform the computationally expensive calculations on their optimized hardware and to simply transmit the resulting outputs to the PLC which subsequently uses them for plant control. For example, Körösi et al. used a remote cloud service to perform an image detection and classification task, and had the results transmitted back to the PLC [50]. This allows the use of advanced deep neural nets such as convolutional neural networks, which would be hard to implement directly on PLCs. However, using external devices carries the risk of communication outages and, for remote servers, internet connection failures, in which case some kind of fallback procedure would have to fill in, in order to guarantee an uninterrupted process control.
Another approach is to deploy the neural network models directly on the PLC, as is proposed by Körösi and Kajan in [52]. Modern PLC processing units are powerful enough to perform regression using larger ANN structures such as deep neural nets. However, although the PLC software supports basic mathematical operations, it has limitations for more advanced operations like vector or matrix operations. Thus, Körösi and Kajan advise to perform the network training on an external machine before transferring the fully trained ANN structure and parameters to the PLC. This can be done manually, configuring the network according to the trained settings, hyperparameters, and weights, which is time consuming and might lead to errors. Alternatively, third-party software can be used to export the ANN

structure to the PLC, for example in the XML format [52]. Deploying the ANN models directly on the PLC makes them less susceptible to communication outages and removes the need for additional hardware such as on-site PCs or remote servers. However, such models are limited by the reduced processing power and the more basic software implementation.

All in all, it is apparent that there still remain challenges before the models can be utilized in any kind of industrial model-based control setup, such as model predictive control or adaptive control. Nonetheless, the findings in this thesis demonstrate that neural network models can be powerful tools for modeling in the process industry. Because of the high potentials, there is a clear trend to make the integration of data-driven models into the process, manufacturing, and control industry a reality in the medium term. To this end, efforts in research and development are done on multiple fronts, both in academia and in the industry, resulting in an active and fast-evolving field of research.

References

[1] Kumpati S. Narendra and Kannan Parthasarathy. “Identification and control of dynamical systems using neural networks”. In: *IEEE Transactions on Neural Networks* 1.1 (1990), pp. 4–27. DOI: 10.1109/72.80202.

[2] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[3] Tom M. Mitchell. *Machine learning*. McGraw-hill New York, 1997.

[4] Martin T. Hagan and Howard B. Demuth. “Neural networks for control”. In: *Proceedings of the 1999 American Control Conference (Cat. No. 99CH36251)*. Vol. 3. 1999, 1642–1656 vol.3. DOI: 10.1109/ACC.1999.786109.

[5] R. Blonbou et al. “Active Adaptive Combustion Control Using Neural Networks”. In: *Combustion Science and Technology* 156.1 (2000), pp. 25–47. eprint: https://doi.org/10.1080/00102200008947295. URL: https://doi.org/10.1080/00102200008947295.

[6] Soteris A. Kalogirou. “Artificial intelligence for the modeling and control of combustion processes: a review”. In: *Progress in Energy and Combustion Science* 29.6 (2003), pp. 515–566. ISSN: 0360-1285. DOI: https://doi.org/10.1016/S0360-1285(03)00058-3. URL: https://www.sciencedirect.com/science/article/pii/S0360128503000583.

[7] J. Stark et al. “Delay embeddings for forced systems. II. stochastic forcing”. In: *Journal of Nonlinear Science* 13.6 (Nov. 2003), pp. 519–577. ISSN: 0938-8974. DOI: 10.1007/s00332-003-0534-4.

[8] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC, 2006. ISBN: 0-387-31073-8.

[9] Bela G. Liptàk. *Process Control and Optimization*. 4th ed. Vol. 2. Instrument Engineers’ Handbook. CRC Press, 2006. ISBN: 978-0-8493-1081-2.

[10] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. ISBN: 0-262-18253-X.

[11] Sudipta De et al. “Development of an artificial neural network model for the steam process of a coal biomass cofired combined heat and power (CHP) plant in Sweden”. In: *Energy* 32.11 (2007), pp. 2099–2109. ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2007.04.008. URL: https://www.sciencedirect.com/science/article/pii/S0360544207000722.

[12] Gregor Gregorčič and Gordon Lightbody. “Local Model Network Identification With Gaussian Processes”. In: *IEEE Transactions on Neural Networks* 18.5 (2007), pp. 1404–1423. DOI: 10.1109/TNN.2007.895825.

[13] Gregor Gregorčič and Gordon Lightbody. “Gaussian process approach for modelling of nonlinear systems”. In: *Engineering Applications of Artificial Intelligence* 22.4 (2009), pp. 522–533. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2009.01.005. URL: https://www.sciencedirect.com/science/article/pii/S0952197609000268.

[14] Jure Smrekar et al. “Development of artificial neural network model for a coal-fired boiler using real plant data”. In: *Energy* 34.2 (2009), pp. 144–152. ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2008.10.010. URL: https://www.sciencedirect.com/science/article/pii/S0360544208002880.

[15] Robert Bauer et al. “Modelling of grate combustion in a medium scale biomass furnace for control purposes”. In: *Biomass and Bioenergy* 34.4 (2010), pp. 417–427. ISSN: 0961-9534. DOI: https://doi.org/10.1016/j.biombioe.2009.12.005. URL: https://www.sciencedirect.com/science/article/pii/S096195340900244X.

[16] Souhaib Ben Taieb, Antti Sorjamaa, and Gianluca Bontempi. “Multiple-output modeling for multi-step-ahead time series forecasting”. In: *Neurocomputing* 73.10 (2010), pp. 1950–1957. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2009.11.030. URL: https://www.sciencedirect.com/science/article/pii/S0925231210001013.

[17] Markus Gölles et al. “Model based control of a biomass grate furnace”. In: *Proceedings of the 9th European Conference on Industrial Furnaces and Boilers*. 2011. ISBN: 978-972-99309-6-6.

[18] Jacques F. Smuts. *Process Controll for Practitioners. How to Tune PID Controllers and Optimize Control Loops*. OptiControls Inc., 2011. ISBN: 978-0-9838438-1-8.

[19] Karl Johan Aström and Richard M. Murray. *Feedback Systems. An Introduction for Scientists and Engineers*. Princton University Press, 2012. ISBN: 978-0-691-13576-2.

[20] Gregor Gregorčič and Gordon Lightbody. “Gaussian process internal model control”. In: *International Journal of Systems Science* 43.11 (2012), pp. 2079–2094. DOI: 10.1080/00207721.2011.564326. eprint: https://doi.org/10.1080/00207721.2011.564326. URL: https://doi.org/10.1080/00207721.2011.564326.

[21] Tijmen Tieleman and Geoffrey Hinton. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[22] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG]. URL: https://arxiv.org/abs/1211.5063.

[23] Tom Schaul, Ioannis Antonoglou, and David Silver. *Unit Tests for Stochastic Optimization*. 2014. arXiv: 1312.6055 [cs.LG]. URL: https://arxiv.org/abs/1312.6055.

[24] Richard Seeber et al. “Erweiterung eines modellbasierten Regelungskonzepts für Biomassefeuerungsanlagen”. In: *at - Automatisierungstechnik* 62.12 (2014), pp. 891–902. DOI: doi:10.1515/auto-2013-1034. URL: https://doi.org/10.1515/auto-2013-1034.

[25] François Chollet et al. *Keras*. https://keras.io. 2015.

[26] Xiaolei Ma et al. “Long short-term memory neural network for traffic speed prediction using remote microwave sensor data”. In: *Transportation Research Part C: Emerging Technologies* 54 (2015), pp. 187–197. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2015.03.014. URL: https://www.sciencedirect.com/science/article/pii/S0968090X15000935.

[27] Christoph Schörghuber et al. “Control of a biomass-furnace based on input-output-linearization”. In: *2015 European Control Conference (ECC)*. 2015, pp. 3508–3513. DOI: 10.1109/ECC.2015.7331077.

[28] Wim van Swaaij, Sascha Kersten, and Wolfgang Palz. *Biomass Power for the World. Transformations to Effective Use*. Vienna Edition. Pan Stanford Publishing Pte. Ltd., 2015. ISBN: 978-981-4669-24-5. DOI: 10.1007/978-3-662-47438-9.

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

[30] Martin Kaltschmitt, Hans Hartmann, and Hermann Hofbauer. *Energie aus Biomasse. Grundlagen, Techniken und Verfahren*. Springer-Verlag Berlin Heidelberg, 2016. ISBN: 978-3-662-47437-2. DOI: 10.1007/978-3-662-47438-9.

[31] Filippo Maria Bianchi et al. *Recurrent Neural Networks for Short-Term Load Forecasting. An Overview and Comparative Analysis*. SpringerBriefs in Computer Science. Springer Cham, 2017. DOI: https://doi.org/10.1007/978-3-319-70338-1.

[32] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[33] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (2017), pp. 354–359. DOI: https://doi.org/10.1038/nature24270.

[34] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG]. URL: https://arxiv.org/abs/1806.07366.

[35] Pankaj Mehta et al. “A high-bias, low-variance introduction to Machine Learning for physicists”. In: *Physics Reports* 810 (May 2019), pp. 1–124. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2019.03.001. URL: http://dx.doi.org/10.1016/j.physrep.2019.03.001.

[36] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: 1912.05911 [cs.LG]. URL: https://arxiv.org/abs/1912.05911.

[37] Zhe Wu et al. “Machine learning-based predictive control of nonlinear processes. Part I: Theory”. In: *AIChE Journal* 65.11 (2019), e16729. DOI: https://doi.org/10.1002/aic.16729. URL: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.16729.

[38] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020. ISBN: 9781108455145.

[39] Simon Fahle, Christopher Prinz, and Bernd Kuhlenkötter. “Systematic review on machine learning (ML) methods for manufacturing processes – Identifying artificial intelligence (AI) methods for field application”. In: *Procedia CIRP* 93 (2020). 53rd CIRP Conference on Manufacturing Systems 2020, pp. 413–418. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2020.04.109. URL: https://www.sciencedirect.com/science/article/pii/S2212827120307435.

[40] Tianhong Liu et al. “Industrial time series forecasting based on improved Gaussian process regression”. In: *Soft Computing* 24.20 (2020), pp. 15853–15869. ISSN: 1433-7479. DOI: 10.1007/s00500-020-04916-6. URL: https://doi.org/10.1007/s00500-020-04916-6.

[41] Gerrit Muller. *Modeling and Analysis; Performance Modeling*. 2020.

[42] Christoph Schörghuber et al. “Control of biomass grate boilers using internal model control”. In: *Control Engineering Practice* 96 (2020), p. 104274. ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2019.104274. URL: https://www.sciencedirect.com/science/article/pii/S0967066119302205.

[43] Enrico Terzi et al. *Model predictive control design for dynamical systems learned by Long Short-Term Memory Networks*. 2020. arXiv: 1910.04024 [eess.SY]. URL: https://arxiv.org/abs/1910.04024.

[44] Stefan Kollmannsberger et al. *Deep Learning in Computational Mechanics. An Introductory Course*. Vol. 977. Studies in Computational Intelligence. Springer Nature Switzerland AG, 2021. ISBN: 978-3-030-76586-6. DOI: https://doi.org/10.1007/978-3-030-76587-3.

[45] Christoph Thon et al. “Artificial Intelligence in Process Engineering”. In: *Advanced Intelligent Systems* 3.6 (2021), p. 2000261. DOI: https://doi.org/10.1002/aisy.202000261. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/aisy.202000261.

[46] Fabio Bonassi et al. “On Recurrent Neural Networks for learning-based control: Recent results and ideas for future developments”. In: *Journal of Process Control* 114 (June 2022), pp. 92–104. ISSN: 0959-1524. DOI: 10.1016/j.jprocont.2022.04.011. URL: http://dx.doi.org/10.1016/j.jprocont.2022.04.011.

[47] *UTSG Water Level Prediction Model Based on LSTM and GRU Algorithms*. Vol. 3: I&C, Digital Control, and Influence of Human Factors. International Conference on Nuclear Engineering. Aug. 2022. DOI: 10.1115/ICONE29-91798. URL: https://doi.org/10.1115/ICONE29-91798.

[48] Max Mowbray et al. "Industrial data science – a review of machine learning applications for chemical and process industries". In: *React. Chem. Eng.* 7 (7 2022), pp. 1471–1509. DOI: 10.1039/D1RE00541C. URL: http://dx.doi.org/10.1039/D1RE00541C.

[49] The MathWorks Inc. *MATLAB version: 24.1.0 (R2024a)*. Natick, Massachusetts, United States, 2024. URL: https://www.mathworks.com.

[50] Körösi L. et al. "From Detection to Action: Implementing Deep Learning Inference in PLC Systems via Docker Services". In: *IFAC-PapersOnLine* 58.9 (2024). 18th IFAC Conference on Programmable Devices and Embedded Systems PDES 2024, pp. 19–24. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2024.07.365. URL: https://www.sciencedirect.com/science/article/pii/S2405896324004543.

[51] *CONENGA Group*. Kontakt: office@conengagroup.com, Telefon: +43 1 23 500 32. Auhofstraße 1/10, 1130 Wien, Österreich, 2025. URL: https://conengagroup.com.

[52] Ladislav Körösi and Slavomír Kajan. "Overview of implementation principles of artificial intelligence methods in industrial control systems". In: *Journal of Electrical Engineering* 76.1 (2025), pp. 99–105. DOI: 10.2478/jee-2025-0010. URL: https://doi.org/10.2478/jee-2025-0010.

List of Figures

List of Tables

Appendix

MATLAB script for data preprocessing

```
%% resampling
TT = table2timetable(T_,"RowTimes","DateTime");
TT = retime(TT,"secondly","linear");
T = timetable2table(TT,"ConvertRowTimes",true);

%% filtering
A = T{:,2:47};
range = [0:1:length(A)-1].';
B = A;
tau = 30;
filter = tf(1,[tau 1]);
for i = 1:45
    B(:,i) = lsim(filter, A(:,i), range(:,1));
end
T{:,2:47} = B;

%% remove outliers and plant shutdown phases
TT_ = table2timetable(T);
S1 = timerange('2022-11-01 00:00:00','2022-12-26 12:00:00');
S2 = timerange('2023-01-01 00:00:00','2023-04-06 19:00:00');
S3 = timerange('2023-04-11 11:00:00','2023-05-24 06:00:00');
S4 = timerange('2023-05-24 14:00:00','2023-06-17 13:00:00');
S5 = timerange('2023-06-18 10:00:00','2023-07-17 10:00:00');
S6 = timerange('2023-07-18 14:00:00','2023-07-26 12:00:00');
S7 = timerange('2023-08-25 05:00:00','2023-10-21 15:00:00');
S8 = timerange('2023-10-21 20:00:00','2023-11-08 15:00:00');
S9 = timerange('2023-11-16 09:00:00','2023-12-28 06:00:00');
S10 = timerange('2023-12-28 18:00:00','2023-12-30 17:00:00');
S11 = timerange('2024-01-03 15:00:00','2024-04-02 00:00:00');
S12 = timerange('2024-04-05 07:00:00','2024-06-01 00:00:00');

TT = [TT_(S1,:);TT_(S2,:);TT_(S3,:);TT_(S4,:);TT_(S5,:);TT_(S6,:);TT_(S7,:)
    ;TT_(S8,:);TT_(S9,:);TT_(S10,:);TT_(S11,:);TT_(S12,:)];
T_new = timetable2table(TT,"ConvertRowTimes",true);

% downsampling to 30s sampling time
Data_new = T_new(1:30:height(T_new),:);
```

Python script for FNN modeling

```
import keras
import tensorflow as tf
import numpy as np
import pandas as pd
from time import time
import os

# Hyperparameters
test_data_split = 0.25
validation_data_split = 0.15
dim_hidden = 128
n_hidden = 4
learn_rate = 0.001
es_patience = 20
loss_metric = 'mse'

# Data preparation
df = pd.read_csv("data.csv")
data = df.to_numpy()
rng = np.random.default_rng()
rng.shuffle(data)
# split into features and labels
X_raw = data.take([2,3,4,5,11],axis=1)
Y_raw = data.take([6,10,12],axis=1)
# rescale to [0,1]
X_max = X_raw.max(axis = 0)
X_min = X_raw.min(axis = 0)
X_mean = X_raw.mean(axis = 0)
X = (X_raw - X_min)/(X_max-X_min)
Y_max = Y_raw.max(axis = 0)
Y_min = Y_raw.min(axis = 0)
Y_mean = Y_raw.mean(axis = 0)
Y = (Y_raw - Y_min)/(Y_max-Y_min)
X = np.asarray(X).astype('float32')
Y = np.asarray(Y).astype('float32')
# split into training und test set
N = X.shape[0]
X_test = X[0:int(test_data_split*N)]
X_train = X[int(test_data_split*N):N]
Y_test = Y[0:int(test_data_split*N)]
Y_train = Y[int(test_data_split*N):N]

# Define model architecture
dim_input = X.shape[1]
dim_output = Y.shape[1]

inputs = keras.Input(shape=(dim_input,))
hid1 = keras.layers.Dense(dim_hidden, activation="relu")(inputs)
hid2 = keras.layers.Dense(dim_hidden, activation="relu")(hid1)
```

```
hid3 = keras.layers.Dense(dim_hidden, activation="relu")(hid2)
hid4 = keras.layers.Dense(dim_hidden, activation="relu")(hid3)
outputs = keras.layers.Dense(dim_output, activation="relu")(hid4)
model = keras.Model(inputs=inputs, outputs=outputs, name= "model")

# compile model
model.compile(optimizer=keras.optimizers.Adam(learning_rate=learn_rate),
    loss=keras.losses.MeanSquaredError(),
    metrics = ["mse","mae"],
)
callback1 = keras.callbacks.EarlyStopping(patience=es_patience,
    restore_best_weights = True)

# Train model
start = time()
hist = model.fit(
    x = X_train,
    y = Y_train,
    batch_size = 64,
    epochs = 3000,
    validation_split = validation_data_split/(1-test_data_split),
    callbacks=[callback1],
    verbose = 1
)
training_duration = time()-start

# Evaluate model
eval = model.evaluate(
    x = X_test,
    y = Y_test,
    verbose=1
)
model.save(model.name+".keras")
```

Python script for LSTM modeling

```
import keras
import tensorflow as tf
import numpy as np
import pandas as pd
from time import time

# hyperparameters
sequence_length = 120
validation_split = 0.15
learn_rate = 0.001
prediction_step = 1
batch_size = 128
dim_lstm = 32

# data preparation
df = pd.read_csv("data.csv")
data = df.to_numpy()
x_data = data.take([2,3,4,5,11,6,10,12],axis=1)
y_data = data.take([6,10,12],axis=1)
# rescale to [0,1]
X_max = x_data.max(axis = 0)
X_min = x_data.min(axis = 0)
X_mean = x_data.mean(axis = 0)
X = (x_data - X_min)/(X_max-X_min)
x_data = X[0:-prediction_step]
Y_max = y_data.max(axis = 0)
Y_min = y_data.min(axis = 0)
Y_mean = y_data.mean(axis = 0)
Y = (y_data - Y_min)/(Y_max-Y_min)
y_data = Y[sequence_length:]

dataset = keras.utils.timeseries_dataset_from_array(
    data = np.asarray(x_data).astype('float32'),
    targets = np.asarray(y_data).astype('float32'),
    sequence_length=sequence_length,
    sequence_stride=1,
    sampling_rate=1,
    batch_size=batch_size,
    shuffle=True,
    seed=None,
    start_index=None,
    end_index=None,
)
dataset_train = dataset.skip(int(dataset.cardinality().numpy() *
    validation_split))
dataset_val = dataset.take(int(dataset.cardinality().numpy() *
    validation_split))

# define model architecture
```

```
for batch in dataset:
  labels, targets = batch
  break
inputs = keras.layers.Input(shape=(labels.shape[1], labels.shape[2]))
lstm_out = keras.layers.LSTM(
    units = dim_lstm,
    activation = "tanh",
    recurrent_activation="sigmoid")(inputs)
outputs = keras.layers.Dense(units = targets.shape[1])(lstm_out)
model = keras.Model(inputs=inputs, outputs=outputs, name= "lstm")

# compile model
model.compile(optimizer=keras.optimizers.Adam(learning_rate=learn_rate),
    loss=keras.losses.MeanSquaredError(),
    metrics = ["mse","mae"],
)
callback_es = keras.callbacks.EarlyStopping(patience=5,restore_best_weights
    = True)

# train model
start = time()
hist = model.fit(
    x = dataset_train,
    epochs = 5000,
    validation_data= dataset_val,
    callbacks=[callback_es],
    verbose = 1
)
training_duration = time()-start

# evaluate model
eval = model.evaluate(
    x = dataset_val,
    verbose=1
)
model.save(model.name+".keras")
```