Norwegian University of Science and Technology

Specialization Project

Optimal operation of a heat exchanger network with stream splits using artifical neural networks

Espen Karlsen

December 17, 2020

Abstract

Maximisation of energy recovery for a heat exchanger network with three stream splits is considered. Its objective is defined as the maximisation of temperature in the cold outlet stream, T. Two split ratios of the cold feed stream are considered to be the degrees of freedom available for this system. Optimal operation of the heat exchanger network is characterized as the state in which the gradient of T with respect to the degrees of freedom are zero. This is achieved when the *Jäschke temperatures* of the stream splits are of equal value.

Artificial neural networks' abilities to aid in achieving optimal operation of the heat exchanger network is investigated. The neural networks are constructed using AutoKeras in Python. Bayesian optimization is used to tune the neural networks' hyperparameters in order to obtain networks with good performance. The neural networks are used in three different methods: Predicting the optimal stream splits of the heat exchanger network, predicting the outlet temperature, and predicting the difference between Jäschke temperatures. The methods are implemented in a closed loop analysis. The first method gives information directly applicable for achieving optimal operation. The second and third method drives the heat exchanger network from a sub-optimal state towards an optimal state in an iterative process.

The results show that all methods are able to achieve operation within one degree of the optimal outlet temperature. The best case managed to achieve a maximum temperature difference of 0.02 degrees between the achieved and optimal temperature for all test data.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Heat exchanger networks

Heat exchanger networks are used in process industries to recover energy. This is done by allowing hot streams to transfer their heat to cold streams. This is important from an economical standpoint, as process plants can decrease costs by increasing energy recovery. It is also important on a more global scale, as climate change and a growing focus on ”green” businesses forces industries to better their energy management. However, determining the optimal operation of a heat exchanger network is not necessarily straightforward and can be limited by which measurements of the network are available. Jäschke and Skogestad(2014) reports that the *Jäschke temperatures* can be used to determine optimal operation of a heat exchanger network with stream splits under a set of specific assumptions. This project will use the concept of Jäschke temperatures to define optimality for the heat exchanger network.[16]

1.2 Artificial neural networks

Artificial neural networks are inspired by the human brain. The brain consist of billions of interconnected neurons giving man the capabilities of complex tasks. Similarily, artifical neural networks consists of artificial neurons that each recieve, process and pass on information. Specht(1991) reported that a general regression neural network can be used for control or prediction of a plant model and that it converges to the underlying regression surface.[22] Zhang(2007) showed that in a batch polymerisation process a stacked neural network model was able to improve performance in the presence of unknown disturbances.[24] To expand upon on these findings this report will investigate the use of artificial neural networks to aid in achieving optimal operation of a heat exchanger network with stream splits.

1.3 Scope of work

This report will investigate artificial neural networks' ability to achieve optimal operation for a heat exchanger network with stream splits. The effect of available measurements is investigated by changing the inputs of the artifical neural networks. Three different

methods are investigated: Predicting the heat exchanger network's optimal inputs, objective function, and the Jäschke tempearture differences.

1.4 Outline of project

This project will first present the reader with necessary background information in order to be able to understand the methods that have been used and interpret the presented results.

Subsequently, the methods for achieving optimal operation of the heat exchanger network with stream splits is presented.

Lastly, the case studies are presented and the results are discussed.

2 Theory

This chapter's function is to present the necessary background information to the reader. The first section provides theoretical insight of optimal operation of heat exchanger networks with stream splits. The section is mainly based on the works of Jäschke and Skogestad(2014). The second section gives a brief introduction of artificial neural networks, with a focus on multilayer perceptrons and its architecture. Subsequently, the third section provides insight into how artificial neural networks can be tuned to give the best performance, particularly with focus on Bayesian optimization.

2.1 Heat exchanger networks with stream splits

Figure 1 shows a primitive overview of a heat exchanger network with N stream splits. The network's objective is to transfer heat from the hot streams to the cold stream, F_0. This is done by splitting F_0 into F_1 , ..., F_i, ..., F_N, and passing them through heat exchangers. The splits can also be referred to as branches.

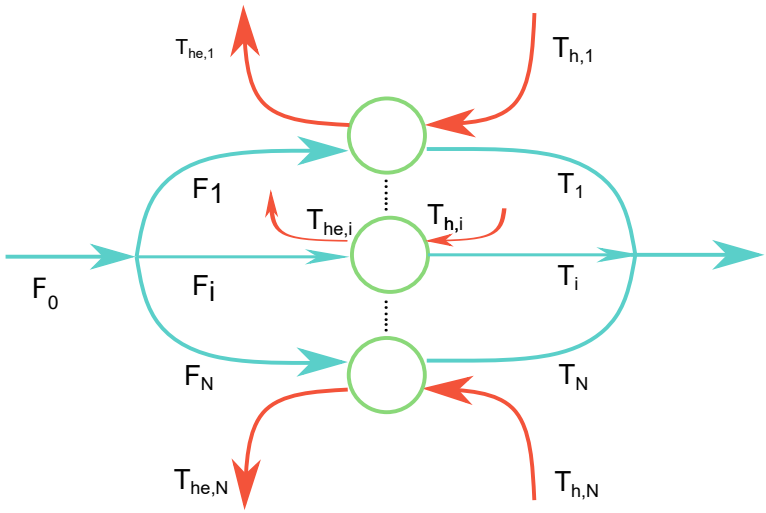

Figure 1: Overview of a heat exchanger network with stream splits.

The blue lines represent the cold stream, and the red lines represent the hot streams. The green circles represent the heat exchangers. F_i, T_i, $T_{h,1}$ and $T_{he,i}$ represent a branch's inlet flow, cold outlet temperature, hot inlet temperature and hot outlet temperature for

branch $i = 1, ..., N$.

Consider the following objective function in order to achieve maximum energy recovery

$$\max_{u} T(u) \tag{1}$$

where $T(u)$ is the outlet temperature of the heat exchanger network and u indicates the available degrees of freedom. As all branches originate from the feed stream F_0, the coupling constraint originating from the mass balance of F_0 implies that only $N - 1$ branches can be changed individually. Consequently, *u* can be defined as

$$\mathbf{u} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_{N-1} \end{pmatrix} \tag{2}$$

Regulating one flow will impact the other flows. By reducing the size of one flow, another stream will correspondingly become larger to satisfy the mass balance. Branches that are more effective at recovering energy should be utilized as much as possible, while ineffective branches should be used as little as possible. The optimal allocation of the mass flow for each branch must be determined. In order to achieve optimal operation it is necessary to control the marginal cost for each branch, $\frac{\partial T_i}{\partial F_i}$, to be equal:[14]

$$\frac{\partial T_1}{\partial F_1} = \frac{\partial T_2}{\partial F_2} = \cdots = \frac{\partial T_{N-1}}{\partial F_{N-1}} = \frac{\partial T_N}{\partial F_N} \tag{3}$$

Jäschke and Skogestad introduces the "Jäschke Temperature", T_J , which for a branch with one heat exchanger is defined as

$$T_J = \frac{(T - T_0)^2}{T_h - T_0} \tag{4}$$

where T is the outlet temperature of the branch, T_0 is the inlet temperature of the branch and T_h is the inlet temperature of the branch's heat exchanger. This proposal assumes that arithmetic mean temperature difference can be used to describe the heat transfer driving

force, that there are no phase changes and that heat capacities can be considered constant. It also assumes that the benefit of recovering energy is independent of which branch it is recovered from. Under these assumptions the Jäschke temperature will be equal to the marginal cost for each branch.

This implies that in order to achieve maximum heat transfer, i.e maximise T in Figure 1, all Jäschke temperatures must be equal. This suggests that optimal operation is achieved when

$$\frac{(T_i - T_0)^2}{T_{h,i} - T_0} = \frac{(T_N - T_0)^2}{T_{h,N} - T_0} \implies \frac{(T_i - T_0)^2}{T_{h,i} - T_0} - \frac{(T_N - T_0)^2}{T_{h,N} - T_0} = 0 \tag{5}$$

for all branches $i = 1, ..., N - 1$.

From this it is possible to create a control variable, c, based on the Jäschke temperatures. c can be defined as the generalization of Equation 5 to matrix form:

$$\mathbf{c} = \begin{pmatrix} T_{J,1} - T_{J,N} \\ T_{J,2} - T_{J,N} \\ \vdots \\ \vdots \\ T_{J,N-1} - T_{J,N} \end{pmatrix} \tag{6}$$

where $T_{J,1}$,$T_{J,2}$, $T_{J,N-1}$ and $T_{J,N}$ are the Jäschke temperatures of branch one, two, $N - 1$ and N, respectively. Optimal operation is achieved when the marginal costs of each branch are of equal value. Therefore, it follows that optimal operation is achieved when $c = 0$.[16]

2.2 Artificial Neural Networks

The human brain is capable of performing demanding and complex computational tasks. This includes pattern recognition, speech, control of bodily functions and more. The brain utilizes a highly parallell computing structure consisting of biological neurons to achieve this.[1] In total, the brain consists of 86 billion interconnected neurons that each are able to receive, process and pass on information. Each neuron consists of three parts in general: The dendrites that recieve signals from surrounding neurons, the neuron cell body, and the axon that passes on signals to other neurons. Each neuron can be inhibited or excited

through recieving signals from other neurons. This signal is in the form of complex time series of electrical signals which allow humans to make complex decisions.[11]

Artificial neural networks(ANN) are motivated to replicate this behaviour, and can be described as computing systems derived from the biological neural networks. Analogous to the biological neural network, ANNs consist of artifical interconnected neurons that can recieve signals from other neurons.[3] This signal is then processed before it is sent to other neurons until the output neurons are reached. This can be seen in Figure 2, which shows an example of an ANN. Each circle represents a neuron and the arrows represent the flow of information and connections between the neurons.

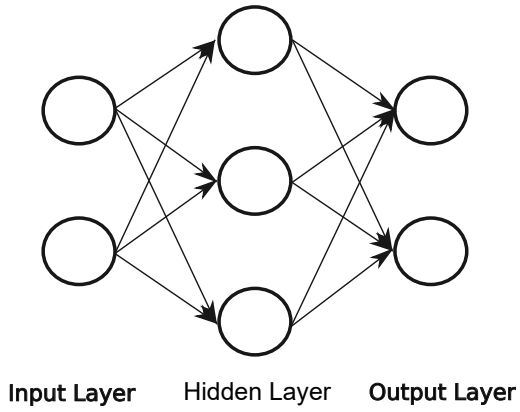

Figure 2: An example of an ANN.

The network in Figure 2 is a special classification of ANN's. It is called Multilayer perceptron(MLP) and is a fully connected feedforward ANN. Fully connected implies that the output of a neuron, referred to as a perceptron, is calculated using every output in the previous layer. Feedforward implies that the network is not cyclical, meaning that the output of a perceptron is only dependant on perceptrons from previous layers.

An MLP is composed of minimum three layers: an input layer, a hidden layer, and an output layer. By using nonlinear activation functions in the hidden layers an MLP can be used to solve nonlinear regression problems. In order to achieve this property, a suitable learning method which is capable of converging to a local minimum when the training data is not linearly seperable must be chosen.[17]

The signal of a neuron is computed from the sum of its input connections in addition to some bias, β. Each connection is also associated with a weight, noted ω. The output μ_j of the j-th neuron is calculated by the following equation:

$$\mu_j = a\left(\sum_{i=1}^{n} \omega_i x_i + \beta_j\right) = a(\vec{\omega_j}^T \vec{x_j} + \beta_j) \tag{7}$$

where x_i is the input from the i-th neuron and ω_i is its respective weight. Equivelantly, $\vec{\omega_j}^T$ is the transposed weight vector and $\vec{x_j}$ is the input vector for the j-th neuron. a is the neuron's activation function which transforms the output, often to the interval $(-1, 1)$ or $(0, 1)$. In a primitive example, μ_j could be computed by the following activation function

$$\mu_j = \begin{cases} 1 & \text{if} \quad \vec{\omega_j}^T \vec{x_j} + \beta_j \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

which is known as the binary step function with a threshold of zero. The calculation of the perceptron's signal is showcased in Figure 3.[1]

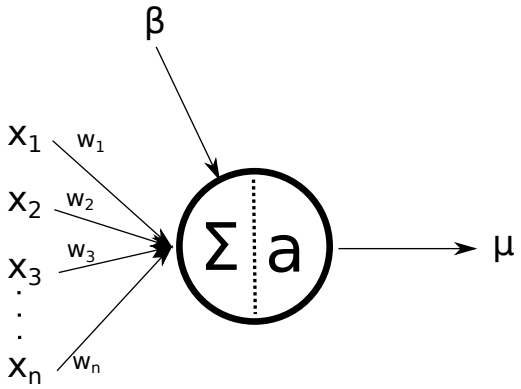

Figure 3: Perceptron calculation.

Each input with its associated weight is summed together in addition to a bias. The sum is then passed through the activation function to produce the neuron's signal.

In order for a neural network to produce meaningful outputs it needs to be trained. This is commonly done by processing data with a given input vector and an ideal output vector. An

objective function is assigned to the neural network. The objective function indicates the error between the current predicted outputs compared to the true outputs. Mean squared error is typically used as the ANN's objective function:

$$\min_{\vec{\omega},\vec{\beta}} \frac{1}{n} \sum_{i=1}^{m} \sum_{j=1}^{n} (\mu_{ij} - \hat{\mu_{ij}})^2 \tag{9}$$

where $\vec{\omega}$ and $\vec{\beta}$ is the neural networks weight and bias vectors, m is the number of outputs for the neural network, n is the number of training samples, and $(\mu_{ij} - \hat{\mu_{ij}})^2$ is the squared residual between the real value and predicted value of the output, μ_{ij}. The objective function can be minimized by applying a learning method to the ANN. This is typical for problems where there are measurements available for the output to be predicted, known as supervised learning. Supervised learning is an iterative process, where for each iteration the objective function is evaluated and subsequently the weights and biases are updated in the direction that mimizes it. This allows the ANN to produce increasingly accurate outputs. The process is repeated until the objective function is minimized to an acceptable level.[10]

2.2.1 Activation Functions

An activation function is in ANNs a method of transforming the output of a perceptron. Activation functions differ in several important properties which make them suitable for different tasks: Linearity, range, differentiability, monotonicity and others. The choice of activation function is important for the performance of the ANN, but is not necessarily straight forward. As previously stated, in order for an ANN to solve nonlinear problems it must have nonlinear activation functions. Intuitively this must be true, because a linear combination of linear combinations is still linear. The simplest activation function, the *identity function*, is a linear function that keeps the output untransformed:[12]

$$f(x) = x \tag{10}$$

Consider an ANN whose objective is to predict the probability of some arbitrary classification: The weighted sum of an output perceptron is given as $\sum_{i=1}^{n} \omega_i x_i + \beta$, which can

take on any value. The output range of each output perceptron should be transformed to be in the range of (0, 1) in order for the network to give logical probability predictions. This can be achieved by using the *Sigmoid* activation function given in Equation 11.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{11}$$
$$f'(x) = f(x)\,(1 - f(x))$$

The Sigmoid function maps very small values to 0 and very large values to 1. This gives the ANN the capability to predict the probability of all given classifications and output the classification with the highest probability. It is also continuously differentiable with an easy-to-calculate derivative, and monotonic. The property of being continuously differentiable is important for performance when using gradient based learning methods, and the property of being monotonic guaranties a convex error surface for a single-layer model. Figure 4 shows the mapping of the Sigmoid function.[9]

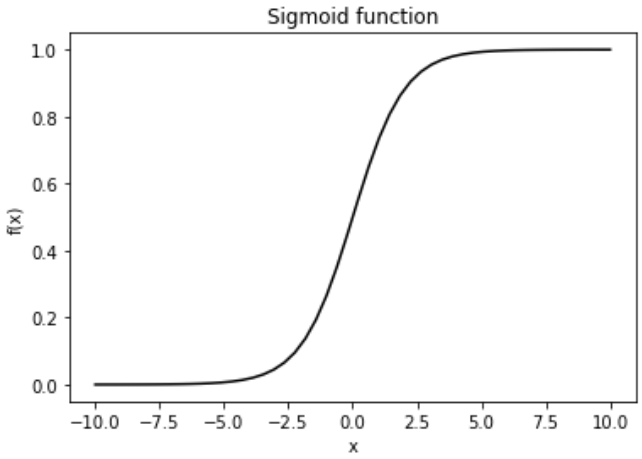

Figure 4: The S-shaped Sigmoid function.

Another commonly used activation function is the rectified linear unit (ReLU). ReLU has shown successful results in achieving good performance compared with other long-established activation functions. [20] The ReLU activation function is defined as

$$f(x) = \max(0, x) \tag{12}$$

and has been shown to perform better than the Sigmoid function, despite being non-differentiable at zero. Figure 5 shows the mapping of the ReLU function.[23]

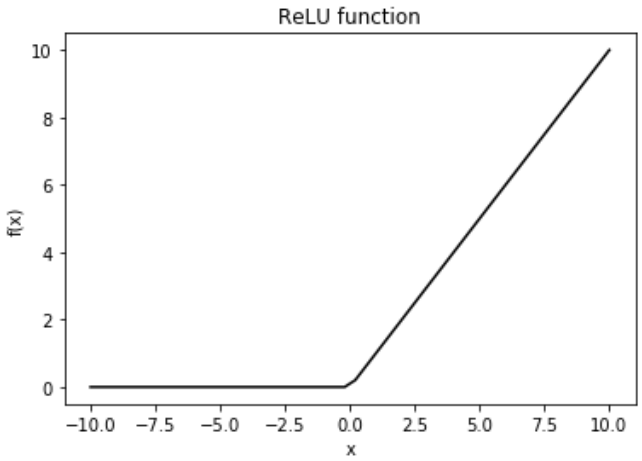

Figure 5: The rectified linear unit.

2.2.2 Backpropagation

Backpropagation is an algorithm commonly used in MLPs for training. The algorithm uses gradient descent to determine how the weights and biases should be updated, and has the ability to approximate nonlinear functions given nonlinear activation functions. [5]

For the network to learn, it must have an objective function to minimize. As previously stated, mean squared error is commonly used. Mean squared error has smooth derivative properties and a penalty for large errors. The algorithm must search the hypothesis space defined by all possible weight and bias combinations for every perceptron to minimize the objective function. Gradient descent is a commonly used method to traverse the hypothesis space in an attempt to find the combination of weights and biases that minimizes the objective function.[19]

Gradient descent is the method of taking the partial derivative of the objective function $MSE(\vec{\omega}, \vec{\beta}) = E$ with respect to each weight and bias. This gives information about

which direction will minimize the objective function. Consider the derivative of the objective function with respect to a given weight k for perceptron p, ω_{kp}, for a single output network as motivation:

$$\frac{\partial E}{\partial \omega_{kp}} \tag{13}$$

If the derivative is positive, then decreasing the weight will result in a smaller error. Likewise, if the derivative is negative, then increasing the weight results in a smaller error. This implies that the change applied to the weight should be a function of the negative derivative. We define the change as follows:

$$\Delta\omega_{kp} = -\eta \frac{\partial E}{\partial \omega_{kp}} \tag{14}$$

where η is defined as the method's *learning rate*. The learning rate is a specified number between 0 and 1, and describes how much the weights should be updated between each learning iteration. Determining a good learning rate is important for performance of the network. If the learning rate is small the network requires more training iterations and might get stuck before reaching an optimal solution. If the learning rate is large the network does not require as many training iterations, but might learn sub-optimal weights and the training process could be unstable. The change in weights can also be written on vector form

$$\Delta\vec{\omega} = -\eta \nabla E(\vec{\omega}) = -\eta \left[\frac{\partial E}{\partial \omega_{11}}, \frac{\partial E}{\partial \omega_{21}}, ..., \frac{\partial E}{\partial \omega_{n-1l}}, \frac{\partial E}{\partial \omega_{nl}} \right] \tag{15}$$

where $\Delta\vec{\omega}$ is a vector of weight changes, $\nabla E(\vec{\omega})$ is the gradient of E with respect to each individual weight, l is the number of layers and n is the number of perceptrons in the last layer.[1]

In order to speed up the training of the ANN, a *momentum term* can also be added to Equation 14.

$$\Delta\omega_{kp} = -\eta \frac{\partial E}{\partial \omega_{kp}} + \alpha \Delta\omega_{kp}^{-1} \tag{16}$$

where the momentum term, α, is a chosen value between 0 and 1, and $\Delta\omega_{kp}^{-1}$ is the weight change from the previous weight update. The momentum term allows previous movements in the weight hypothesis space to be taken into account when updating the weights in the current training iteration. Momentum can also have the added benefit of being able to

pass through local minima and flat regions in the hypothesis space. Similarily to learning rate, the value of the momentum term is chosen by the implementer of the MLP, and is referred to as a *hyperparameter*.[4]

There exists several variations of the gradient descent method. *Stochastic gradient descent* is a method that updates the weights based on the mean squared error of one training data sample. *Batch gradient descent* updates the weights based on the mean squared error of all training data samples, n. *Mini-batch gradient descent* updates the weights after b training data samples where $1 < b < n$. Another variation is *Adaptive Moment Estimation* (Adam). Adam differs from the other methods in that it calculates individual learning rates for each weight and bias. The weight update equation for Adam is given as

$$\Delta\omega_{kp} = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{17}$$

where $\hat{m}_t$ is the estimate of an exponential moving average of the gradient, $\sqrt{\hat{v}_t}$ is the estimate the squared gradient, and η and ϵ are user specified constants. Adam has shown good results for large networks with many tuneable parameters.[6]

In order to make use of Equation 16, the partial derivative of the error with respect to ω_{kp} must be expressed analytically. The method of stochastic gradient descent does this for output perceptrons by calculating the partial differential of the error for each training iteration, $E_i = \frac{1}{2}\sum_{j=1}^{m}(\mu_j - \hat{\mu_j})^2$:

$$\frac{\partial E_i}{\partial \omega_{kp}} = \frac{\partial}{\partial \omega_{kp}}\left(\frac{1}{2}\sum_{j=1}^{m}(\mu_j - \hat{\mu_j})^2\right) \tag{18}$$

where m is the number of ouput perceptrons. The derivative of $(\mu_j - \hat{\mu_j})^2$ is zero for all output perceptrons except p. The summation can be removed and j set equal to p.

$$\frac{\partial E_i}{\partial \omega_{kp}} = \frac{1}{2}\frac{\partial}{\partial \omega_{kp}}(\mu_p - \hat{\mu_p})^2$$

$$= \frac{1}{2}2(\mu_p - \hat{\mu_p})\frac{\partial}{\partial \omega_{kp}}(\mu_p - \hat{\mu_p})$$

$$= (\mu_p - \hat{\mu_p}) \frac{\partial}{\partial \omega_{kp}} (\mu_p - a(\vec{\omega_p} \cdot \vec{x_p})) \tag{19}$$

Where a is the activation function of the output perceptron associated with weight ω_{kp}. The derivative of the activation function is non-zero only for ω_{kp} in the weight vector $\vec{\omega_p}$. Using the Sigmoid function as activation function it follows:

$$\frac{\partial E_i}{\partial \omega_{kp}} = -(\mu_p - \hat{\mu_p}) a(\omega_{kp} \cdot x_{kp})(1 - a(\omega_{kp} \cdot x_{kp})) \cdot x_{kp} \tag{20}$$

Combining Equation 16 and 20 to get the final weight training equation for output perceptrons:

$$\Delta\omega_{kp} = \eta(\mu_p - \hat{\mu_p}) a(\omega_{kp} \cdot x_{kp})(1 - a(\omega_{kp} \cdot x_{kp})) \cdot x_{kp} + \alpha \Delta\omega_{kp}^{-1} \tag{21}$$

The weight training rule changes based on the chosen activation function, but the procedure for obtaining Equation 21 is the same. The weight training rule for hidden perceptrons can be derived in a similar manner, but the rule must also consider ω_{kp}'s effect on the output for perceptrons in layers closer to the output layer, $DeeperLayers$. The training rule using the Sigmoid function is given as:

$$\Delta\omega_{kp} = \eta a(\omega_{kp} x_{kp})(1 - a(\omega_{kp} x_{kp})) x_{kp} \sum_{j \epsilon DeeperLayers} \vec{\omega_j} \phi_j + \alpha \Delta\omega_{kp}^{-1} \tag{22}$$

Where ϕ_j is calculated from:

$$\phi_j = \begin{cases} a(\vec{\omega_j}\vec{x_j})(1 - a(\vec{\omega_j}\vec{x_j}))(\mu_j - \hat{\mu_j}) & \text{if} \quad j \text{ is an output perceptron} \\ a(\vec{\omega_j}\vec{x_j})(1 - a(\vec{\omega_j}\vec{x_j})) \sum_{h \epsilon DeeperLayers} \vec{\omega_h}\phi_h & \text{if} \quad j \text{ is a hidden perceptron} \end{cases} \tag{23}$$

2.3 Tuning hyperparameters

The perfomance of ANNs is heavily dependant on the choice of hyperparameters. The hyperparameters include learning rate, momentum, activation functions, batch size and number of epochs, but the architectural structure itself could also be considered as hyperparameters. The number of hidden layers, the number of perceptrons in each layer and choice of learning method should also be explored when searching for the ANN with best performance.

The most primitive way of tuning hyperparameters is hand-tuning. This is both time consuming and often relies on insight and knowledge that is difficult to quantify.

A more common method is grid search. Grid search explores all given combinations of hyperparameters in an attempt to find the best combination. This is obviously extremely computationally expensive and time consuming, as the number of combinations increases exponentially with the number of hyperparameters. In addition, grid search does not take into account the magnitude of performance impact for each hyperparameter. Some hyperparameters might be irrelevant for performance compared to others. Random grid search, a similar method which randomly choses hyperparameter combinations, has been shown to be able to find equally good or better hyperparameters compared with grid search while using a modicum of computational cost. It is however, mathematically unsatisfactory and not easily replicable.[13]

A better method is that of Bayesian optimization. Bayesian optimization has proven to be a powerful tool capable of outperforming human experts and search methods, while using a fraction of the computational cost.[2]

Bayesian optimization works by treating the neural network as a black box, $f(\theta)$, whose inputs are the hyperparameters, θ. The black box is assigned a Gaussian process prior which describes the probabilistic belief about the neural network $p(f) = \text{GP}(f; \mu, \sigma^2)$. Each time a new set of hyperparameters is evaluated, knowledge about the neural network is realized. This produces a Bayesian posterior probability distribution, $p(f|D) = GP(f|D; \mu, \sigma^2)$, which follows another GP. The posterior probability gives information

about the likely values of $f(\theta)$ for a given θ. The explicit objective of Bayesian optimization is to find the best set of hyperparameters, θ^*, that globally minimizes $f(\theta)$.[8]

$$\theta^* = \arg\min_{\theta} f(\theta) \tag{24}$$

To find θ^*, the combination of hyperparameters to test is chosen based on previous knowledge about the behaviour of the neural network. The method of chosing a new hyperparameter combination is called an acquistion function. The most common acquisition function is expected improvement(EI)

$$\text{EI}_n(\theta) = E_n[0, f' - f(\theta)] \tag{25}$$

where f' is the best value of $f(\theta)$ observed so far and E_n is the expectation taken under the Bayesian posterior distribution of $f(\theta)$ given observations $f(\theta_1)...f(\theta_n)$. EI evaluates $f(\theta)$ at the point θ which improves upon f' the most. If $f(\theta)$ is closer to minimum, then the expected improvement is $f' - f(\theta)$, otherwise it is zero as f' is still the best point. Equation 25 can be analytically solved as

$$\text{EI}_n(\theta) = \left(f' - \mu(\theta)\right) \Phi\left(f'; \mu(\theta), \sigma^2(\theta,\theta)\right) + \sigma^2(\theta,\theta) N\left(f'; \mu(\theta), \sigma^2(\theta,\theta)\right) \tag{26}$$

where $\mu(\theta)$ is the mean function derived from the Gaussian process prior and $\sigma^2(\theta,\theta)$ is the normally distributed covariance function of the prior. The next evaluation point, θ_{n+1} is chosen as the argument that maximizes Equation 26

$$\theta_{n+1} = \arg\max_{\theta} \text{EI}_n(\theta) \tag{27}$$

Equation 26 can be maximised by looking at the two terms. The first term can be increased by evaluating $f(\theta)$ at points where $\mu(\theta)$ is small. The second term can be increased by evaluating $f(\theta)$ at points where the covariance is large. This can be understood as a compromise between exploration and exploitation. The acquistion function explores the areas

where there is large uncertainty, and it exploites the areas where the mean is small. Using this algorithm the hyperparameter space is explored in a way that reduces the number of unnecessary evaluations, i.e where the mean is large and the variance is small.[21]

Upper confidence bound(UCB) is another commonly used acquisition function. UCB is similar to expected improvement as it compromises between exploration and exploitation in the same manner. UCB is defined as

$$UCB(\theta, \kappa) = \mu(\theta) + \kappa\sigma(\theta) \tag{28}$$

where $\mu(\theta)$ is the mean function of the prior, κ is a user specified constant and $\sigma(\theta)$ is standard deviation of $f(\theta)$. κ determince the balance between exploitation and exploration. The larger κ is, the more explorative the acquisition function is. [7]

3 Heat exchanger application

This section presents the reader with the framework for which the case studies are based on. In the first section the model and data is presented. The second section gives an overview of the different methods of using artificial neural networks to operate the heat exchanger network optimally. The third section presents the different case studies that were conducted.

3.1 Heat exchanger network with three stream splits

The process to be approximated is a heat exchanger network with stream splits. The network has three branches, each with one heat exchanger. The objective of the heat exchanger network is to maximize the transfered heat between the cold stream and the hot streams. In simpler terms, the goal is to maximize T. Figure 6 illustrates the heat exchanger network.

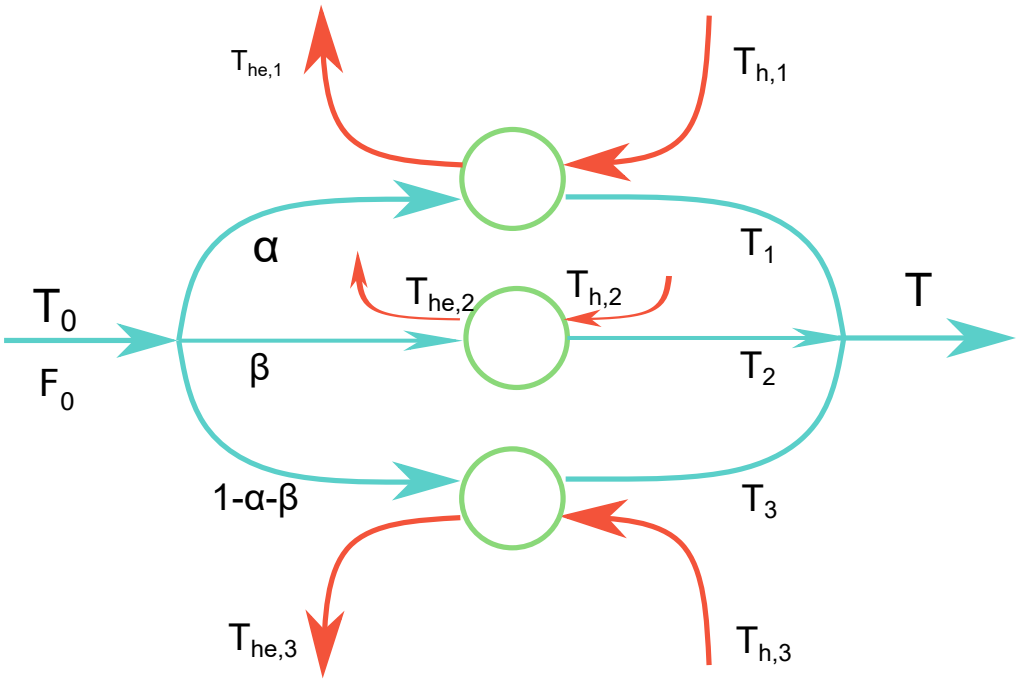

Figure 6: Overview of the heat exchanger network with three branches.

The network has several variables. The outlet temperature, inlet temperature and flow of

the cold stream, T, T_0 and F_0. For heat exchanger $i = 1, 2, 3$: the cold outlet temperature, hot inlet temperature, hot outlet temperature, overall heat transfer coefficient and hot mass flow, T_i, $T_{h,i}$, $T_{he,i}$, UA_i and $w_{h,i}$. Lastly, the branch split fractions is given as α, β and $1 - \alpha - \beta$, where $\alpha = \frac{F}{F_1}$ and $\beta = \frac{F}{F_2}$.

It is necessary to determine what constitutes optimal operation for this heat exchanger network in order to be able to maximize the temperature. Under the assumptions that arithmetic mean temperature difference can be used to describe the heat transfer driving force, no phase changes and constant heat capacities, Jäschke temperatures can be used to describe optimality. This also assumes that the benefit of recovering energy is independant of which branch it is recovered from. From the theory in section 2 it follows that optimal operation is achieved when control variable c is equal to zero. c for this heat exchanger network is given as:

$$\mathbf{c} = \begin{pmatrix} T_{J,1} - T_{J,3} \\ T_{J,2} - T_{J,3} \end{pmatrix} \tag{29}$$

where $T_{J,1}$, $T_{J,2}$ and $T_{J,3}$ are the Jäschke temperatures of branch one, two and three, respectively. Furthermore, we define c_1 and c_2 as the *Jäschke temperature differences*:

$$\begin{aligned} c_1 &= T_{J,1} - T_{J,3} \\ c_2 &= T_{J,2} - T_{J,3} \end{aligned} \tag{30}$$

Data is required for an artifical neural network to be able to train, validate and test. A first principle model with noise is used to generate data. Data is provided to the neural network in the form of measurement sets. Each measurement set represents a scenario where only a select number of variables are measured in the heat exchanger network. Several training data sets is therefore generated by the first principle model of the heat exchanger network, where operation is either optimal or sub-optimal. This allows for several methods when using neural networks in order to operate the heat exchanger network optimally. The different measurement sets is given in Equation 31. Each column represents a measurement set.

$$\textbf{Measurement sets} = \begin{pmatrix} T_0 & T_0 & T_0 & T_0 \\ T_1 & T_{h,1} & T & T \\ T_{h,1} & T_{h,2} & T_{he,1} & T_{h,1} \\ T_2 & T_{h,3} & T_{he,2} & T_{h,2} \\ T_{h,2} & T_{he,1} & T_{he,3} & T_{h,3} \\ T_3 & T_{he,2} & w_0 & \alpha \\ T_{h,3} & T_{he,3} & w_{h,1} & \beta \\ - & - & w_{h,2} & - \\ - & - & w_{h,3} & - \end{pmatrix}^{\mathrm{T}} \tag{31}$$

Sixteen data sets are generated in total. One data set for each measurement set at optimal operating conditions and at sub-optimal operating conditions. This is done for both training and test data. Each data sample also contains the values of α, β, T, and c. In addition to this, disturbances on the plant is available in order to make it possible to simulate the heat exchanger network in a closed loop. The given disturbances, d, are shown in Equation 32:

$$\mathbf{d} = \begin{pmatrix} T_0 \\ w_0 \\ w_{h,1} \\ w_{h,2} \\ w_{h,3} \\ T_{h,1} \\ T_{h,2} \\ T_{h,3} \\ UA_1 \\ UA_2 \\ UA_3 \end{pmatrix}^{T} \tag{32}$$

3.2 Artificial Neural Network Methods

Using the generated data sets makes it possible to use supervised learning to build ANNs. The objective of the neural network is to be able to predict a given variable as close to the real value as possible. This is done by minimizing the mean squared error objective function

$$\min_{\vec{\omega},\vec{\beta}} \frac{1}{n} \sum_{i=1}^{m} \sum_{j=1}^{n} (\mu_{ij} - \hat{\mu_{ij}})^2 \tag{33}$$

where $\vec{\omega}$ and $\vec{\beta}$ is the neural networks weight and bias vectors, m is the number of outputs for the neural network, n is the number of training samples, and $(\mu_{ij} - \hat{\mu_{ij}})^2$ is the mean squared error between the real value and predicted value of the output, μ_{ij}. The different neural network methods described in this section are based on which output the neural networks are trying to predict. The output of the neural networks should be a parameter which can give information with respect to operating the heat exchanger network optimally.

It is difficult to decide which architecture of a given neural network will give the best performance, as alluded to in subsection 2.2. In order to minimize the objective function in Equation 33, the neural network's hyperparameters must be given. This is a challenge, as the choice of hyperparameters is not obvious. Bayesian optimization is therefore used to overcome this challenge. The implementation of Bayesian optimization is done with AutoKeras in Python, an efficient neural network search system, and uses UCB as its aquisition function.[15] AutoKeras is built on the machine learning software TensorFlow.[18] The general procedure for creating the neural networks can be seen in Algorithm 1.

Algorithm 1 ANN General Procedure

1: **procedure** CREATE ANN
2: **Required:** $Dataset$
3: $x \leftarrow get\_measurements(Dataset)$ ▷ Get inputs for ANN
4: $\mu \leftarrow get\_optimal\_outputs(Dataset)$ ▷ Get real outputs
5: $ANN \leftarrow AutoKeras(x, \mu)$ ▷ Get best model from Bay. opt.

AutoKeras will in step 5 of the algoritm determine the suitable choice of hyperparameters. Our methods consists of training the generated neural network on one thousand data samples. AutoKeras can however generate very large neural networks with hundreds of thousands of tuneable parameters. Using neural networks with such a difference in number of parameters and data samples raises concern for overfitting. Luckily, AutoKeras automatically deals with this by using *dropout*, a regularization technique.

3.2.1 Predicting optimal inputs

The first method is constructing an ANN that predicts the optimal splits α_{opt} and β_{opt}. This gives information which can be directly used to operate the heat exchanger network at optimal conditions. A closed loop analysis can be done for this method by applying the predicted optimal inputs to the first principle model. This will generate new measurements. The new measurements can be used by the neural network to predict a new pair of optimal inputs. The cycle is repeated until the change in optimal inputs is below a specified tolerance. The last generated optimal input pair is then applied to the first principle model to determine the achieved temperature. An overview of the closed loop analyis can be seen in Algorithm 2.

Algorithm 2 Closed loop analysis: α_{opt}&β_{opt}

procedure
  Required: $ANN, \alpha_{k-1}, \beta_{k-1}, tol$
  while $(\alpha_k - \alpha_{k-1}) >$ tol or $(\beta_k - \beta_{k-1}) >$ tol **do**
    $\alpha_{k-1} \leftarrow \alpha_k$
    $\beta_{k-1} \leftarrow \beta_k$
    $measurements \leftarrow$ First principle model
    $\alpha_k, \beta_k \leftarrow ANN(measurements)$

An alternative strategy for applying this method is to use two ANNs for α_{opt} and β_{opt}, instead of one ANN with two outputs. The closed loop analysis procedure is the same as with one model.

3.2.2 Predicting the objective function

The second method constructs an ANN that predicts the heat exchanger network's temperature based on the available measurements. This does not provide much useful information in an open loop analysis. However, finite elements can be used in a closed loop analysis to predict the gradients of T, $\frac{\partial T}{\partial \alpha}$ & $\frac{\partial T}{\partial \beta}$. This is done by starting at a random, but feasable operating point α_0, β_0 and the corresponding T_0. Finite elements is then used to estimate the gradients of T. Inside the loop a small change, Δ, is applied to α and β in the direction that maximises T. New measurements are then generated from the first principle model. A new value of T can then be predicted by the neural network. Subsequently, a new estimation of the gradient can be done. This continues until the estimated gradients are below a set tolerance value. An overview of the closed loop analysis can be seen in Algorithm 3.

Algorithm 3 Closed loop analysis: T

procedure
  Required: $ANN, \alpha_0, \beta_0, T_0, \alpha, \beta, tol, \Delta$
  $T \leftarrow ANN(openloop)$
  $\frac{\partial T}{\partial \alpha} \leftarrow$ Finite elements
  $\frac{\partial T}{\partial \beta} \leftarrow$ Finite elements
  while $(\frac{\partial T}{\partial \alpha}) >$ tol or $(\frac{\partial T}{\partial \beta}) >$ tol **do**
    $\alpha_0 \leftarrow \alpha$
    $\alpha \leftarrow \alpha + \Delta \cdot \text{sign}(\frac{\partial T}{\partial \alpha})$ ▷ Add Δ to α in the direction of max T
    $T_0 \leftarrow T$
    $measurements \leftarrow$ First principle model
    $T \leftarrow ANN(measurements)$
    $\frac{\partial T}{\partial \alpha} \leftarrow$ Finite elements

    $\beta_0 \leftarrow \beta$
    $\beta \leftarrow \beta + \Delta \cdot \text{sign}(\frac{\partial T}{\partial \beta})$
    $T_0 \leftarrow T$
    $measurements \leftarrow$ First principle model
    $T \leftarrow ANN(measurements)$
    $\frac{\partial T}{\partial \beta} \leftarrow$ Finite elements

3.2.3 Predicting the Jäschke temperature differences

The third method constructs an ANN that predicts the Jäschke temperature differences, c_1 and c_2 from Equation 30. This does not give much information in an open loop analysis, but optimal operation can be achieved in a closed loop analysis similar to method two. The Jäschke temperature differences give information regarding which direction α and β should be changed in order to drive c_1 and c_2 to zero. This method can also be done with two ANNs that predict one Jäschke temperature difference each. The algorithm for this method is the same as Algorithm 3, except the finite elements steps are not needed.

3.3 Case Studies

The case studies are divided into five groups based on the neural network method that was used. Each case study group has four case studies, based on which measurement set was used to train the neural network model. An overview of the case studies is shown in Table 1.

Case study group one uses one neural network model to predict the optimal split ratios, α_{opt} and β_{opt}. The data is taken from a heat exchanger network which is operating optimally.

Case study group two uses two neural network models to predict α_{opt} and β_{opt}, respectively. The data is the same as used in case study group one.

Case study group three uses one neural network model to predict the current temperature based on plant measurements. The case studies also address the neural networks performance in closed loop analysis, where changes in α and β are used to estimate the gradient of T with respect to α and β based on finite elements. This is done to drive the temperature towards its optimal value.

Case study group four uses one neural network model to predict the Jäschke temperature differences. Similar to the previous case study group, these case studies also investigates the networks ability to use the predictions in a closed loop analysis. The closed loop analysis' goal is to drive the Jäschke temperature differences to zero.

Case study group five uses two neural network models to predict the Jäschke temperature differences. The same closed loop analysis from case study group four is used.

Table 1: Summary of cases studies.

Case	Inputs									Output		No. of models
1.1	T_0	T_1	T_2	T_3	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$			α_{opt}	β_{opt}	1
1.2	T_0	$T_{h,1}$	$T_{he,2}$	$T_{he,3}$	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$			α_{opt}	β_{opt}	1
1.3	T_0	T	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$	F_0	$F_{h,1}$	$F_{h,2}$	$F_{h,3}$	α_{opt}	β_{opt}	1
1.4	T_0	T	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$	α	β			α_{opt}	β_{opt}	1
2.1	T_0	T_1	T_2	T_3	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$			α_{opt}	β_{opt}	2
2.2	T_0	$T_{h,1}$	$T_{he,2}$	$T_{he,3}$	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$			α_{opt}	β_{opt}	2
2.3	T_0	T	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$	F_0	$F_{h,1}$	$F_{h,2}$	$F_{h,3}$	α_{opt}	β_{opt}	2
2.4	T_0	T	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$	α	β			α_{opt}	β_{opt}	2
3.1	T_0	T_1	T_2	T_3	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$			T		1
3.2	T_0	$T_{h,1}$	$T_{he,2}$	$T_{he,3}$	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$			T		1
3.3	T_0	T	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$	F_0	$F_{h,1}$	$F_{h,2}$	$F_{h,3}$	T		1
3.4	T_0	T	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$	α	β			T		1
4.1	T_0	T_1	T_2	T_3	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$			c_1	c_2	1
4.2	T_0	$T_{h,1}$	$T_{he,2}$	$T_{he,3}$	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$			c_1	c_2	1
4.3	T_0	T	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$	F_0	$F_{h,1}$	$F_{h,2}$	$F_{h,3}$	c_1	c_2	1
4.4	T_0	T	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$	α	β			c_1	c_2	1
5.1	T_0	T_1	T_2	T_3	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$			c_1	c_2	2
5.2	T_0	$T_{h,1}$	$T_{he,2}$	$T_{he,3}$	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$			c_1	c_2	2
5.3	T_0	T	$T_{he,1}$	$T_{he,2}$	$T_{he,3}$	F_0	$F_{h,1}$	$F_{h,2}$	$F_{h,3}$	c_1	c_2	2
5.4	T_0	T	$T_{h,1}$	$T_{h,2}$	$T_{h,3}$	α	β			c_1	c_2	2

4 Results

This section presents results of each case study. The neural network architecture generated from Bayesian optimization is presented. Furthermore, the network's open loop predictions and closed loop performance is shown. Both open loop mean squared error(OLMSE) and closed loop mean squared error(CLMSE) is reported. OLMSE is the mean squared error of the difference between achieved temperature in open loop and optimal temperature. Similarily for CLMSE, except for the difference between achieved temperature in closed loop and optimal temperature. Only one illustrative example of the results is shown for each case study group, as the figures are very similar. The figures of the remaining case studies can be seen in Appendix B. Some figures include a bisecting line. This represents the values of true predictions for the neural network. In layman's terms, good predictions are close to the bisecting line.

4.1 Case Study Group 1

Case study group one utilizes measurements from the heat exchanger model under optimal operating conditions. The output is the optimal split ratios.

Table 2 shows the generated neural network architectures from AutoKeras using Bayesian optimization.

Table 2: Generated neural network architectures.

Case	Hidden Layers	Perceptrons	Activation Functions	Tuneable Parameters
1.1	3	16, 16, 16	ReLU, ReLU, ReLU	913
1.2	1	16	ReLU	177
1.3	1	16	ReLU	213
1.4	1	16	ReLU	241

Figure 7 and 8 shows the prediction of the optimal inputs in open loop for case study 1.1. The figures show that the neural network is able to encapsulate the nature of the optimal inputs. This is especially true near the center of the bisecting line, where the majority of the datapoints lie. The predictions are less accurate near the edges where the data is

sparse.

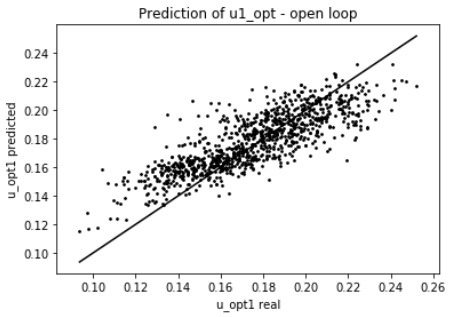

Figure 7: Prediction of α_{opt} in open loop.

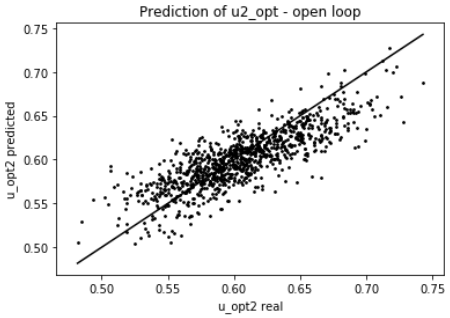

Figure 8: Prediction of β_{opt} in open loop.

Figure 9 and 10 shows the prediction of the optimal inputs in closed loop for case study 1.1. The predictions show a similar pattern to the open loop predictions, as the predictions are more accurate near the center. However, the closed loop analysis has more outliers.

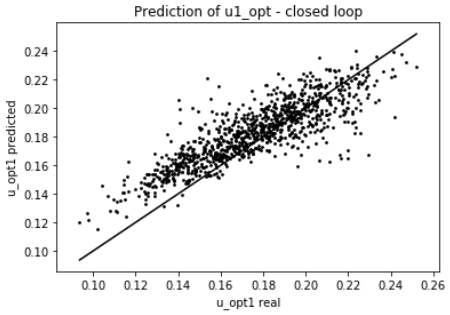

Figure 9: Prediction of α_{opt} in closed loop.

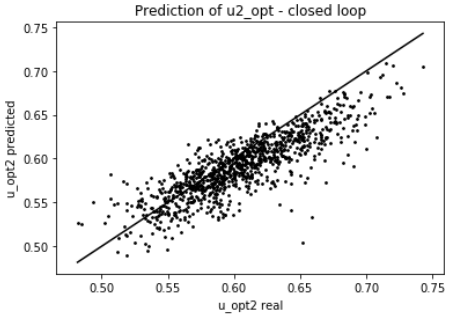

Figure 10: Prediction of β_{opt} in closed loop.

Figure 11 and 12 shows histograms of the residual between the optimal temperature and the achieved temperature by the predicted stream splits for case study 1.1. It is evident that the residual is quite similiar for both the open and closed loop. One can however observe a higher frequency of residuals above 0.2 for the closed loop.

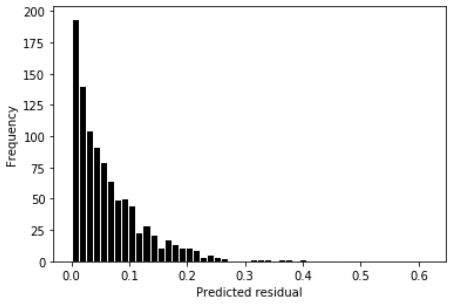

Figure 11: Residual between optimal temperature and predicted temperature in open loop.

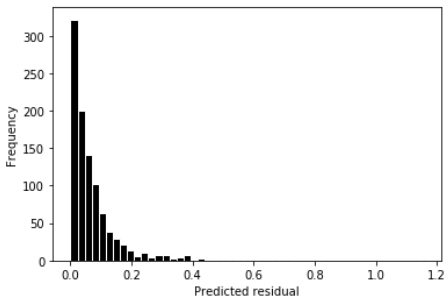

Figure 12: Residual between optimal temperature and predicted temperature in closed loop.

Table 3 shows the results of case study group one. The results show a similar performance between case 1.1, 1.2 and 1.4. Case study 1.3 has a significantly higher mean squared error, especially in the closed loop.

Table 3: The results of case study group one.

Case	OLMSE	CLMSE
1.1	0.00956	0.0153
1.2	0.00268	0.0126
1.3	0.01433	0.0551
1.4	0.00641	0.00921

4.2 Case Study Group 2

Case study group two is structured similarily to case study group one. The goal for both is to predict α_{opt} and β_{opt}. However, case study group two generates two neural networks to attempt this.

Table 4 shows the generated neural network architectures from AutoKeras using Bayesian optimization.

Table 4: Generated neural network architectures.

Case	Hidden Layers	Perceptrons	Activation Functions	Tuneable Parameters
2.1	2	32, 16	ReLU, ReLU	801
	2	16, 256	ReLU, ReLU	4752
2.2	3	32, 32, 32	ReLU, ReLU, ReLU	2401
	2	32, 64	ReLU, ReLU	2433
2.3	2	32, 32	ReLU, ReLU	1428
	2	32, 64	ReLU, ReLU	2516
2.4	2	32, 32	ReLU, ReLU	1345
	3	32, 32, 32	ReLU, ReLU, ReLU	2401

Figure 13 and 14 shows the prediction of the optimal inputs in open loop for case study 2.1. Similarily to case study group one, the predictions are close to optimal near the center.

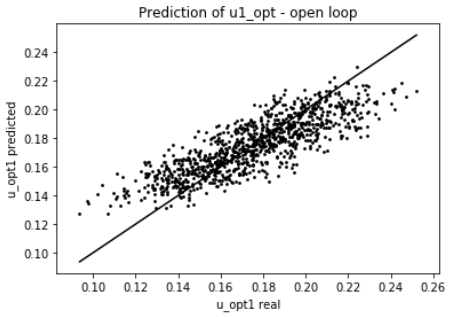

Figure 13: Prediction of α_{opt} in open loop.

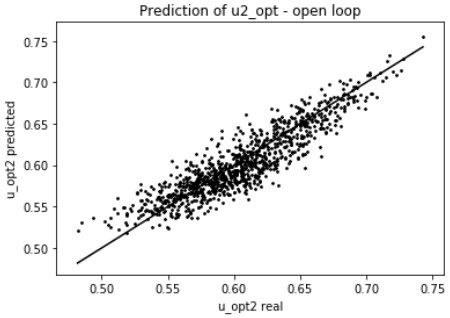

Figure 14: Prediction of β_{opt} in open loop.

Figure 15 and 16 shows the prediction of the optimal inputs in closed loop. The prediction of α_{opt} looks almost identical to the open loop figure. The prediction of β_{opt} however has several outliers generated in the closed loop analysis.

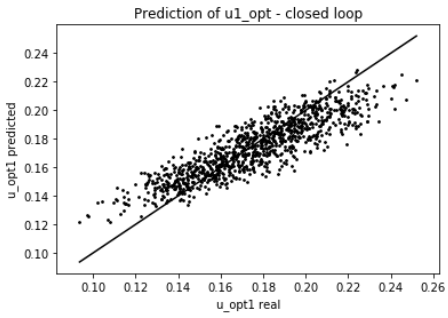

Figure 15: Prediction of α_{opt} in closed loop.

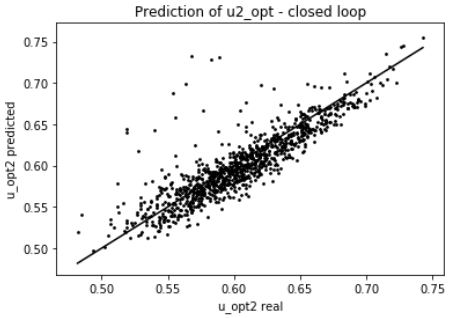

Figure 16: Prediction of β_{opt} in closed loop.

Figure 17 and 18 shows histograms of the residual between the optimal temperature and the achieved temperature by the predicted stream splits. It is evident that the residual is quite similiar for both the open and closed loop.

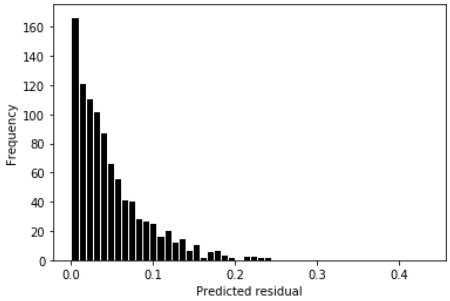

Figure 17: Residual between optimal temperature and predicted temperature in open loop.

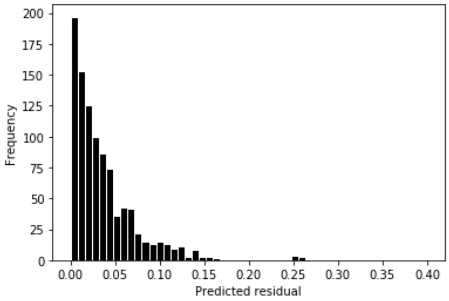

Figure 18: Residual between optimal temperature and predicted temperature in closed loop.

Table 5 shows the results of case study group two.

Table 5: The results of case study group two.

Case	OLMSE	CLMSE
2.1	0.00487	0.0222
2.2	0.00822	0.00828
2.3	0.00885	0.0328
2.4	0.00690	0.0108

There is little variation in the results, as all cases report similar mean squared errors for both open and closed loop. However, measurement set two provided the best CLMSE.

4.3 Case Study Group 3

Case study group three predicts the outlet temperature of the heat exchanger network.

Table 6 shows the generated neural network architectures from AutoKeras using Bayesian optimization.

Table 6: Generated neural network architectures.

Case	Hidden Layers	Perceptrons	Activation Functions	Tuneable Parameters
3.1	1	512	ReLU	4624
3.2	2	16, 1024	ReLU, ReLU	18576
3.3	2	16, 32	ReLU, ReLU	756
3.4	2	128, 512	ReLU, ReLU	70145

Figure 19 shows the open loop prediction of T. The figure is very similar to the open loop predictions of α_{opt} and β_{opt} in case study group one and two. However, the magnitude of the error between the prediction and true value is much larger.

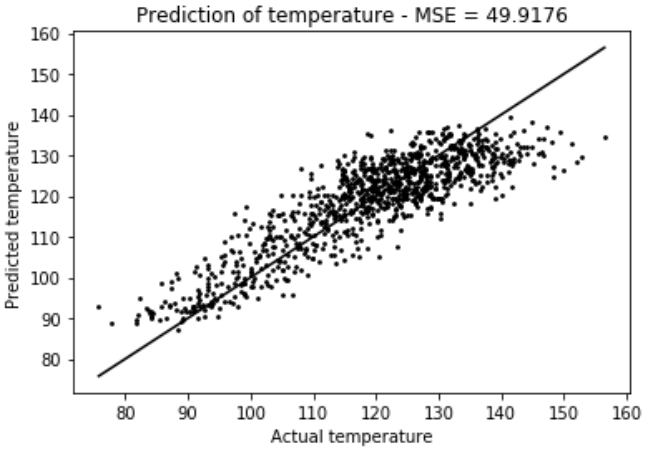

Figure 19: Prediction of T in open loop.

The open loop predictions are used in Algorithm 3 in order to obtain the closed loop predictions of the temperature. Figure 20 shows the closed loop results of case study 3.1. The histogram shows the residual between the optimal temperature and the achieved temperature in the closed loop.

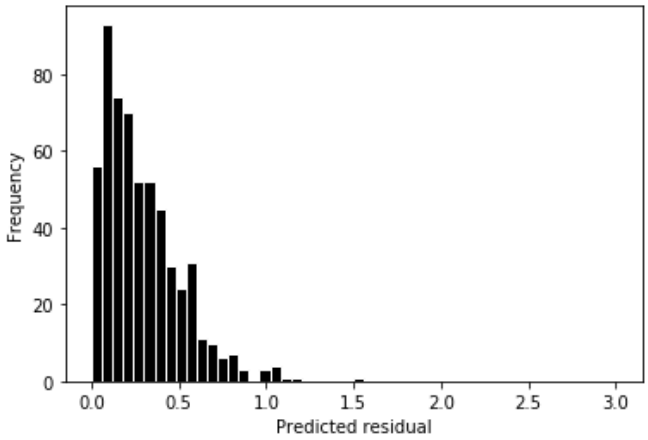

Figure 20: Residual between optimal temperature and achieved temperature in closed loop.

Table 7 shows the results of case study group three. The open loop results can not be use directly to calculate an error between optimal temperature and achieved temperature, therefore only CLMSE is presented. Case study 3.4 stands out from the results, as it is

three orders of magnitude smaller compared with the second best case. This could be due to the fact that the generated neural network for case 3.4 was much larger compared with the other cases.

Table 7: The results of case study group three.

Case	CLMSE
3.1	0.1320
3.2	1.75
3.3	1.02
3.4	0.0001

4.4 Case Study Group 4

Case study group four predicts the Jäschke temperature differences.

Table 8 shows the generated neural network architectures from AutoKeras using Bayesian optimization.

Table 8: Generated neural network architectures.

Case	Hidden Layers	Perceptrons	Activation Functions	Tuneable Parameters
4.1	2	1024, 32	ReLU, ReLU	41073
4.2	2	256, 256	ReLU, ReLU	68369
4.3	3	512, 16, 1024	ReLU, ReLU, ReLU	39013
4.4	1	512	ReLU	7185

Figure 21 and 22 shows the prediction of the Jäschke temperature differences in open loop for case 4.2. The data is from a sub-optimal first principle model, therefore we want to use Algorithm 3 to drive the predicted Jäschke temperature differences towards zero.

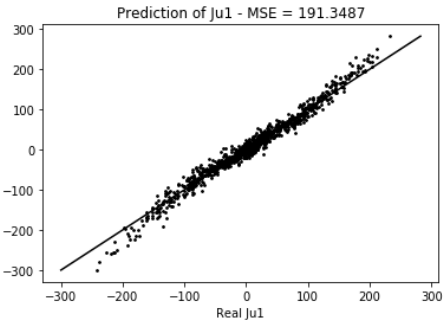

Figure 21: Prediction of c_1 in open loop.

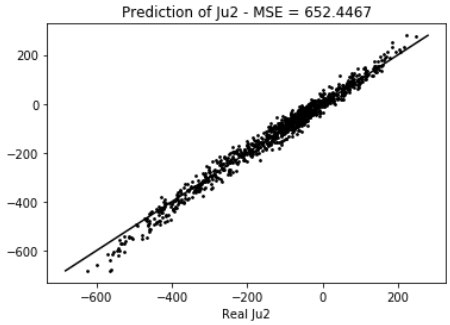

Figure 22: Prediction of c_2 in open loop.

The results of the closed loop analysis for case 4.2 is given in Figure 23. The histogram shows that the neural network was able to achieve a temperature within one degree of the true optimal temperature by using Algorithm 3.

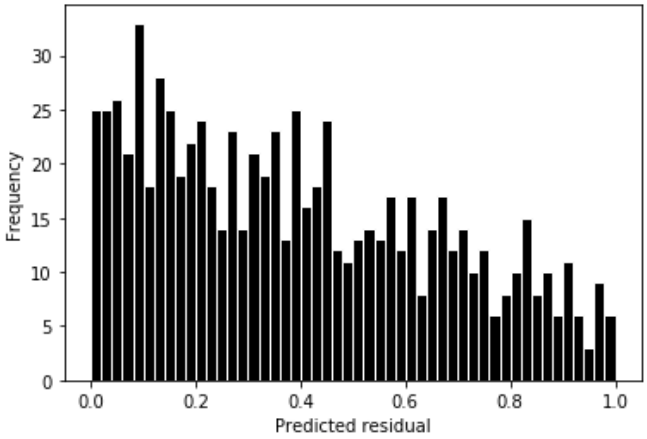

Figure 23: Closed loop analysis.

The closed loop results of case study group four is given in Table 9. Similarily to case study group three, only the closed loop results are relevant for determining the achieved temperature compared to the optimal temperature. Case study 4.1 achieved the best result while case study 4.2 achieved the worst.

Table 9: The results of case study group four.

Case	CLMSE
4.1	0.0510
4.2	0.741
4.3	0.526
4.4	0.173

4.5 Case Study Group 5

Case study group four predicts the Jäschke temperature differences using two neural networks.

Table 10 shows the generated neural network architectures from AutoKeras using Bayesian optimization.

Table 10: Generated neural network architectures.

Case	Hidden Layers	Perceptrons	Activation Functions	Tuneable Parameters
5.1	2	256, 1024	ReLU, ReLU	271376
	3	512, 32, 32	ReLU, ReLU, ReLU	21616
5.2	1	128	ReLU	1168
	2	16, 512	ReLU, ReLU	11472
5.3	3	32, 256, 32	ReLU, ReLU, ReLU	18324
	3	512, 32, 128	ReLU, ReLU, ReLU	25908
5.4	1	128	ReLU	1680
	1	1024	ReLU	9232

Figure 24 and 25 shows the prediction of the Jäschke temperature differences in open loop for case 5.1. Figure 24 shows better predictions compared to Figure 25, especially near the ends.

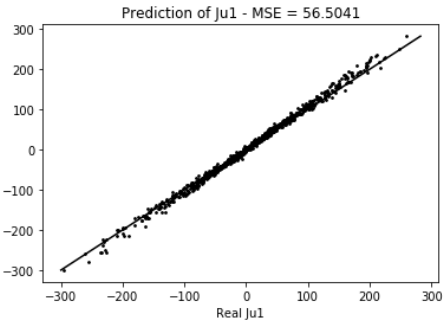

Figure 24: Prediction of c_1 in open loop.

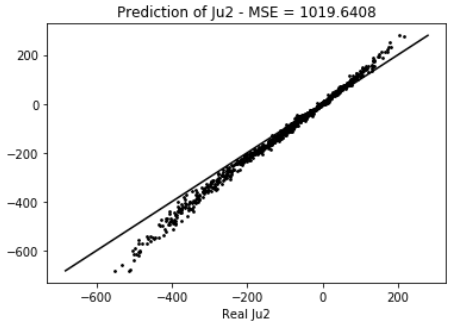

Figure 25: Prediction of c_2 in open loop.

The results of the closed loop analysis is given in Figure 26. The histogram shows that the neural network was able to achieve a temperature within one degree of the true optimal temperature for all data samples by using Algorithm 3. Almost all data samples achieved an error of at most 0.4 from the optimal temperature.

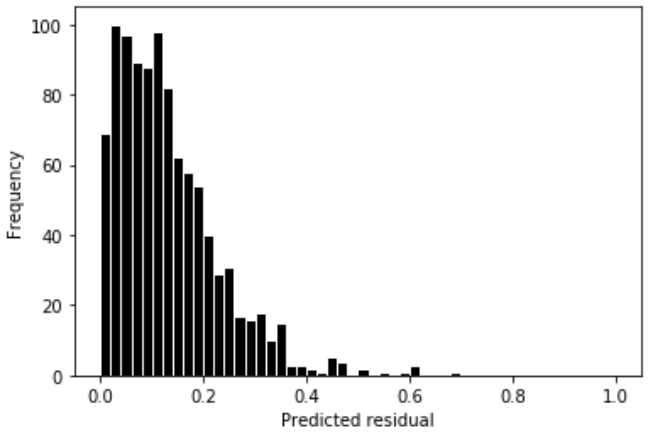

Figure 26: Closed loop analysis.

The closed loop results of case study group five is given in Table 11. Case study 5.1 outperformed the other case studies by as much as two orders of magnitude. This could be due to measurement set one containing the measurements needed to calculate the Jäschke temperature differences. Another cause could be the size of the neural network generated, as it has more than ten times the number of tuneable parameters compared to the second largest network in the case study group. The other cases had poor performance with the

worst closed loop result of all cases. Case 5.4 was unable to converge in the closed loop, due to calculating split fractions that summed to more than 1. This resulted in illogical achieved temperatures whose value were larger than the optimal value.

Table 11: The results of case study group four.

Case	CLMSE
5.1	0.0276
5.2	2.11
5.3	6.06
5.4	Unconverged

The CLMSE results showcases that case 5.1 was able to get a good performance in the closed loop despite its open loop prediction of c_2 being poor. On the other hand, the performance of the case study group was very volatile.

5 Discussion

The closed loop results show that all methods were able to achieve close-to-optimal operation for the heat exchanger network. Every case group managed to get a CLMSE of less than 0.06 for atleast one case. Case 3.4, where T was estimated and optimal operation was achieved with finite elements, had the lowest CLMSE. Figure 27 shows a histogram of the residuals for this case.

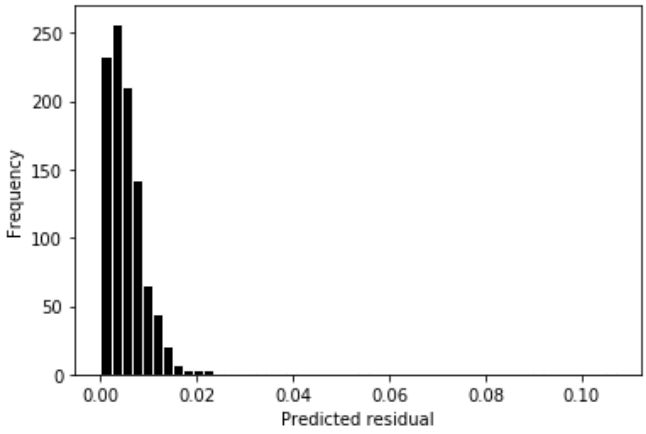

Figure 27: Histogram of residuals for case 3.4

Figure 27 shows that case 3.4 had a maximum residual of just over 0.02 with the majority of residuals being below 0.01. This implies that the heat exchanger network was operated within at most 0.02 degrees from the optimal outlet temperature for all data samples by using the generated neural network and Algorithm 3.

Cases 1.2, 1.4, 2.2 and 2.4 had the best results in group 1 and 2. Measurement set four was expected to perform well, as α_{opt} and β_{opt}, which the neural network is trying to predict for this case, is part of the measurement set. The performance of the other measurement sets did not deviate much in performance. This could indicate that the prediction of the optimal split ratios is independant of which measurements are available. Alternatively, it could indicate that the outputs of the neural network is not accurate enough to be non-trivial. This is supported by the fact that the hypothesys space of T with respect to α and β is relatively flat near the optimum, so predictions can be considered good in a large region.

The measurement sets seemingly had a bigger impact on performance in case group three, as there were four orders of magnitude between the best and worst case.

Case group four and five showed similar results. Both had the best CLMSE when using measurement set one. This was expected, as the measurement set contains the variables needed to calculate the Jäschke temperature differences. The performance between using one and two models was also illustrated well by the results. Two models were able to achieve a better CLMSE compared to one model when using measurement set one, while one model performed better with the other three measurements sets. In addition to this, case 5.4 with two models were unable to converge in a closed loop, as they were outputting illogical values for α and β in the closed loop algorithm. This indicates that using one model was better than two models.

The first method of predicting α_{opt} and β_{opt} is seemingly more robust compared with the two other methods. The closed loop results for case group one and two had less variation across the measurement sets compared with group three, four and five. The CLMSE for group one and two ranged between 0.00828 and 0.0551, while for case three it ranged between 0.0001 and 1.75. Furthermore, group four's CLMSE ranged between 0.0510 and 0.741, while group five had an unconverged case. This could indicate that the expected result of method one is on average better than the other two methods.

The neural networks generated from AutoKeras varied greatly with respect to architecture. The largest network from case 5.1 had 271 376 tuneable paremeters, while the smallest network from case 1.2 had only 177 tuneable parameters. AutoKeras choses the architectures based on the best open loop performance. As a result of this, the dependency of closed loop performance and network size is difficult to determine. The results does however indicate that the largest neural networks in general performed better than small networks. The smallest network within a study group never achieved the smallest CLMSE. However, the largest network in study group three and five achieved the smallest error. There does not seem to be a clear cause for why the variation between network architectures is so large. Part of the variation can however be attributed to the complexity of each case group method, as case group 1 and 2 generated smaller networks compared

with case group 3, 4 and 5. Furthermore, the data used in case group 1 and 2 is not the same as for case group 3, 4 and 5. As mentioned in section 3.2, the size of the networks could indicate overfitting. For instance, case 5.1 had over 270 000 tuneable parameters in its model for predicting c_1. One would normally expect overfitting when the number of parameters is much larger than the number of training samples, which is 1000 for this case. However, AutoKeras uses dropout to counteract overfitting, so it should not be of concern. ReLU and the Adam learning method was chosen for all networks, which is expected as both have shown good results compared to their respective alternatives.

6 Conclusion

The objective of this project was to investigate neural networks' abilities to aid in achieving optimal operation of a heat exchanger network with three stream splits. Optimal operation was defined as the maximisation of the cold outlet temperature, T. Under the assumptions proposed by Jäschke and Skogestad (2014), this could be achieved by having the Jäschke temperature of each branch be of equal value. Three main methods were used: Predicting the optimal stream splits of the heat exchanger network, predicting the outlet temperature, and predicting the Jäschke temperature differences. Each method carried out a closed loop analysis using Algorithm 2 and 3 to determine each method's ability to achieve optimal operation.

Data was gathered from a first principle model and used as input for the neural networks. Both optimal data and sub-optimal data was used. The data was split into four different groups based on which measurements were available to the neural networks. AutoKeras with integrated Bayesian optimization was used to generate the best neural network achitectures.

The results of the closed loop analysis showed that all methods were able to achieve close-to-optimal operation for the heat exchanger network with three stream splits. Every group had at least one case that managed to get a closed loop mean squared error of 0.06 or less. The method of using the neural network to predict T with measurement set 4 had the best performance. It had a mean squared error of 0.0001, and almost all of its achieved temperatures had a residual of less than 0.02 degrees from optimal operation. The variations within the individual case study groups indicated that the method of predicting α_{opt} and β_{opt} gave the most consistent performance in closed loop.

6.1 Future work

The work done in this project can be extended by investigating how the proposed neural network methods compare to a standard PI or PID controller for achieving optimal operation. Reactions to changes in disturbances and setpoints should be investigated. Furthermore, the oscillatory behaviour can be compared for a wide range of operating conditions

to test stability of both neural network methods and PI or PID controllers.

The dependancy of training samples could also be further investigated. This project has used 1000 training samples to train each neural network model. The performance gained by increasing the number of training samples, or performance lost by removing training samples would be of interest to quantify.

References

[1] Ajith Abraham. “Artificial Neural Networks.” In: *Handbook of Measuring System Design* (2005).

[2] Simon Bartels et Al. “Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets.” In: *International Conference on Artificial Intelligence and Statistics* (2017).

[3] Yung-Yao Chen et Al. “Design and Implementation of Cloud Analytics-Assisted Smart Power Meters Considering Advanced Artificial Intelligence as Edge Analytics in Demand-Side Management for Smart Homes.” In: *Sensors* (2019).

[4] Nii O. Attoh-Okine. “Analysis of learning rate and momentum term in backpropagation neural network algorithm trained to predict pavement performancen.” In: *Advances in Engineering Software 30 (1999) 291–302* (1999).

[5] Fu-Chuang Chen. “Back-Propagation Neural Networks for Nonlinear Self-Tuning Adaptive Control.” In: *IEEE Control Systems Magazine* (1990).

[6] Jimmy Lei Ba Diederik P. Kingma. *SADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. 2017.

[7] Nando de Freitas Eric Brochu Vlad M. Cora. *Optimal operation of heat exchanger networks with stream split: Only temperature measurements are required*. 2010.

[8] Peter Frazier. “A Tutorial on Bayesian Optimization.” In: (2018).

[9] Moraga C. Han J. “The influence of the sigmoid function parameters on the speed of backpropagation learning.” In: *From Natural to Artificial Neural Computation* (1995).

[10] Jeff Heaton. *Artificial Intelligence for Humans Volume 3: Deep Learning and Neural Networks*. Heaton Research, Inc, 2015.

[11] Suzana Herculano-Houzel. “The Human Brain in Numbers: A Linearly Scaled-up Primate Brain.” In: *Frontiers in Human Neuroscience* (2009).

[12] Knut Hinkelmann. *Neural Networks p.7*. University of Applied Sciences Northwestern Switzerland. URL: https://www.cse.wustl.edu/~garnett/cse515t/spring_2015/files/lecture_notes/12.pdf.

[13] Yoshua Bengio James Bergstra. “Random Search for Hyper-Parameter Optimization.” In: *Journal of Machine Learning Research 13 (2012) 281-305* (2012).

[14] Sigurd Skogestad James J. Downs. *An Industrial and Academic Perspective on Plantwide Control*. 2011.

[15] Haifeng Jin, Qingquan Song, and Xia Hu. “Auto-Keras: An Efficient Neural Architecture Search System.” In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2019, pp. 1946–1956.

[16] Sigurd Skogestad Johannes Jäschke. “A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning.” In: *Computers and Chemical Engineering* (2014).

[17] Navin Kumar Manaswi. *Deep Learning with Applications Using Python*. Apress, Berkeley CA, 2018.

[18] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[19] Tom Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

[20] Quoc V. Le Prajit Ramachandran Barret Zoph. *Searching for activation functions*. 2017.

[21] Roman Garnett. *Bayesian Optimization*. Department of Computer Science and Engineering Washington University in St. Louis. 2015. URL: https://www.cse.wustl.edu/~garnett/cse515t/spring_2015/files/lecture_notes/12.pdf.

[22] JDonald F. Specht. “A General Regression Neural Network.” In: *IEEE TRANSACTIONS ON NEURAL NETWORKS* (1991).

[23] Yoshua Bengio Xavier Glorot Antoine Bordes. *Deep Sparse Rectifier Neural Networks*. 2011.

[24] Jie Zhang. “Batch-to-batch optimal control of a batch polymerisation process based on stacked neural network models.” In: *Chemical Engineering Science* (2007).

A Appendix - Python Scripts

A.1 Python Code 1

Listing 1: Prediction of optimal inputs

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 26 11:02:48 2020

@author: Espen Karlsen
"""

import numpy as np
import pandas as pd
import tensorflow as tf
import os as os
from autokeras import StructuredDataRegressor
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
from hex3_output import hex3_output

# %% Meta
# def neural_network_model(csv_data, csv_test_data, msn,
    meas_set):
csv_data = 'meas_3_opt.xlsx'
csv_test_data = 'meas_3_test_opt.xlsx'
msn = 3
# meas_set = ['T0', 'T1', 'Th1', 'T2', 'Th2', 'T3', 'Th3
    ']
# meas_set = ['T0', 'Th1', 'Th2', 'Th3', 'Th1e', 'Th2e',
    'Th3e']
meas_set = ['T0', 'T', 'Th1e', 'Th2e', 'Th3e', 'w0', 'wh1
```

```
    ', 'wh2', 'wh3']
# meas_set = ['T0', 'T', 'Th1', 'Th2', 'Th3', 'alpha', '
    beta']

# %% Load the dataset
data = pd.read_excel(csv_data)
data_test = pd.read_excel(csv_test_data)

dataset = np.array(data)
data_test = np.array(data_test)
np.random.shuffle(dataset) # randomise the data
np.random.shuffle(data_test) # randomise the data


# %% Split into input (X) and output (Y) variables
n_measurements = len(meas_set)        #number of
    measurements vary between 7 and 9
n_parameters = 11
n_outputs = 2
X = dataset[:, 0:n_measurements]
Y = dataset[:, n_measurements+n_parameters:n_measurements
    +n_parameters+n_outputs]

X = tf.keras.utils.normalize(X)

# %%
# tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir='logs/OneModel{}'.format(int(time.time())))
# earlystopping_callback = tf.keras.callbacks.
    EarlyStopping(monitor='val_loss', patience=200)
```

```
reg = StructuredDataRegressor(max_trials=15, directory=os
    .path.normpath('C:/'), overwrite=True, loss='
    mean_squared_error', tuner='bayesian')

reg.fit(x=X, y=Y, verbose=2, batch_size=40, epochs=25)

model = reg.export_model()



## model.predict(x_train)

# %% Test Data
test_rows = 999
random_test_data = np.random.randint(0, len(data)-
    test_rows)
# random_test_data = 1

test_x  = data_test[random_test_data:random_test_data +
    test_rows, 0:n_measurements]
test_p  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements:n_measurements+n_parameters]
test_y  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters:n_measurements+
    n_parameters+3]
test_ju = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+3:
    n_measurements+n_parameters+5]
test_b  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+5:]
```

```
test_u1 = test_y[:, 0]
test_u2 = test_y[:, 1]
test_u  = test_y[:, 0:2]
test_j  = test_y[:, 2]

test_x = tf.keras.utils.normalize(test_x)



# %% Prediction - open loop
prediction_open_loop = model.predict(test_x)
J_open_loop = []

for i in range(test_rows):
    plant_open_loop = hex3_output(prediction_open_loop[i
       ], test_p[i])
    J_open_loop.append(plant_open_loop[0])
# %% Prediction - closed loop
tol = 1e-7
meas_dict = {'T0'   :  0, 'T1'   :  1, 'T2'    :  2, 'T3'
      :  3,
             'Th1'  :  4, 'Th2'  :  5, 'Th3'   :  6, '
                Th1e' :  7,
             'Th2e' :  8, 'Th3e' :  9, 'T'     : 10, 'wh1
                '  : 11,
             'wh2'  : 12, 'wh3'  : 13, 'alpha' : 14, '
                beta' : 15,
             'w0'   : 16}

iteration_limit = 100
J_closed_loop = []
Ju = [[], []]
```

```
prediction_closed_loop_u1 = []
prediction_closed_loop_u2 = []
alpha = 0.5
cl_fail = 0

for i in range(test_rows):
    it = 0 # iteration checker
    measurement_list = test_x[i:i+1] # to get 7 by 1
       instead of 7 by 0
    u_opt0 = [1, 1]
    u_opt = model.predict(measurement_list)[0]
    u_imp = u_opt
    while (abs(u_imp[0] - u_opt0[0]) > tol or abs(u_imp
       [1] - u_opt0[1]) > tol) and it < iteration_limit:
        u_opt0 = u_imp

        # get measurements from plant based on prev u_opt
        # this will give us new measurements
        plant = hex3_output(u_imp, test_p[i])
        plantJ = plant[0]
        plant_measurements = plant[1] # jascke herel23,

        # collect the measurements relevant to our case
        # delete previous measurements
        measurement_list = []
        for measurement in meas_set:
            measurement_list.append( plant_measurements[
               meas_dict[measurement]] )

        # add bias to our plant measurements and
           normalize
```

```
        for j in range(len(measurement_list)):
            measurement_list[j] += test_b[i][j]
        measurement_list = tf.keras.utils.normalize(
            measurement_list)

        # calculate new u_opt based on measurements
        u_opt = model.predict(measurement_list)[0]
        u_imp = u_opt0 + (u_opt - u_opt0)*alpha

        # iteration control
        it += 1
        print(i, it)
        if it == iteration_limit:
            print('Closed-loop analysis did not converge
                for datarow: {0:d} ({1:d} + 1)'.format(i +
                 1, i))
            cl_fail += 1
    prediction_closed_loop_u1.append(u_imp[0])
    prediction_closed_loop_u2.append(u_imp[1])
    J = hex3_output(u_imp, test_p[i])[0]
    J_closed_loop.append(J)

    # Gradiant approximation
    # J is the newest calculated temperature, plantJ is
        the previous one
    # obtained in the loop
    # Ju_meas =  hex3_output(u_imp, test_p[i])[1]
    T0 = plant_measurements[meas_dict['T0']]
    J_u1 = ((plant_measurements[meas_dict['T1']] - T0)**2
         / (plant_measurements[meas_dict['Th1']] - T0)) -
        ((plant_measurements[meas_dict['T3']] - T0)**2 / (
```

```
        plant_measurements[meas_dict['Th3']]- T0))
    J_u2 = (plant_measurements[meas_dict['T2']] - T0)**2
        / (plant_measurements[meas_dict['Th2']] - T0) - (
        plant_measurements[meas_dict['T3']] - T0)**2 / (
        plant_measurements[meas_dict['Th3']]- T0)
    Ju[0].append(J_u1)
    Ju[1].append(J_u2)

# %% Compute loss
Open_loop_loss = 0
Closed_loop_loss = 0
for i in range(len(test_x)):
    Open_loop_loss += -(J_open_loop[i] - test_j[i])
    Closed_loop_loss += -(J_closed_loop[i] - test_j[i])

# Mean squared error
MSE_J_ol = mean_squared_error(test_j, J_open_loop)
MSE_J_cl = mean_squared_error(test_j, J_closed_loop)



# %% Plotting

# Plot prediction of u1_opt : open loop - bisecting line
pyplot.title('Prediction of u1_opt - open loop')
pyplot.scatter(test_u1, prediction_open_loop[:,0], c='k',
    s=3)
bisecting_line = np.linspace(min(test_u1), max(test_u1))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt1 real')
pyplot.ylabel('u_opt1 predicted')
pyplot.show()
```

```
# Plot prediction of u1_opt : closed loop - bisecting
    line
pyplot.title('Prediction of u1_opt - closed loop')
pyplot.scatter(test_u1, prediction_closed_loop_u1, c='k',
    s=3)
bisecting_line = np.linspace(min(test_u1), max(test_u1))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt1 real')
pyplot.ylabel('u_opt1 predicted')
pyplot.show()

# Plot prediction of u2_opt : open loop - bisecting line
pyplot.title('Prediction of u2_opt - open loop')
pyplot.scatter(test_u2, prediction_open_loop[:,1], c='k',
    s=3)
bisecting_line = np.linspace(min(test_u2), max(test_u2))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt2 real')
pyplot.ylabel('u_opt2 predicted')
pyplot.show()

# Plot prediction of u1_opt : closed loop - bisecting
    line
pyplot.title('Prediction of u2_opt - closed loop')
pyplot.scatter(test_u2, prediction_closed_loop_u2, c='k',
    s=3)
bisecting_line = np.linspace(min(test_u2), max(test_u2))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt2 real')
pyplot.ylabel('u_opt2 predicted')
```

```
pyplot.show()


#Plot prediction of J_opt
pyplot.title('Prediction of J_opt closed-loop - [MS{0:d}]
    '.format(msn))
pyplot.scatter(test_j,J_closed_loop, c='k', s=3)
bisecting_line = np.linspace(min(test_j), max(test_j))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Data Sample Index')
pyplot.ylabel('J_opt')
pyplot.legend()
pyplot.show()

pyplot.hist([a-b for (a, b) in zip(test_j, J_open_loop)],
     bins=50, color='k', ec='white')
pyplot.xlabel('Predicted residual')
pyplot.ylabel('Frequency')
pyplot.show()

pyplot.hist([a-b for (a, b) in zip(test_j, J_closed_loop)
   ], bins=50, color='k', ec='white')
pyplot.xlabel('Predicted residual')
pyplot.ylabel('Frequency')
pyplot.show()

model.summary()
print(MSE_J_ol, MSE_J_cl, "\n", cl_fail)
```

A.2 Python Code 2

Listing 2: Prediction of optimal inputs - two models

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov  5 18:16:39 2020

@author: Espen

DOUBLE AI MODEL
"""



import numpy as np
import pandas as pd
import tensorflow as tf
import os as os

from autokeras import StructuredDataRegressor
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
from hex3_output import hex3_output



# %% Meta
# def neural_network_model(csv_data, csv_test_data, msn,
    meas_set):
csv_data = 'meas_4_opt.xlsx'
csv_test_data = 'meas_4_test_opt.xlsx'
msn = 4
# meas_set = ['T0', 'T1', 'Th1', 'T2', 'Th2', 'T3', 'Th3
```

```
    'T']
# meas_set = ['T0', 'Th1', 'Th2', 'Th3', 'Th1e', 'Th2e',
    'Th3e']
# meas_set = ['T0', 'T', 'Th1e', 'Th2e', 'Th3e', 'w0', '
    wh1', 'wh2', 'wh3']
meas_set = ['T0', 'T', 'Th1', 'Th2', 'Th3', 'alpha', '
    beta']

# %% Load the dataset
data = pd.read_excel(csv_data)
data_test = pd.read_excel(csv_test_data)
dataset = np.array(data)
data_test = np.array(data_test)
np.random.shuffle(dataset) # randomise the data
np.random.shuffle(data_test) # randomise the data


# %% Split into input (X) and output (Y) variables
n_measurements = len(meas_set)        #number of
    measurements vary between 7 and 9
n_parameters = 11
n_outputs = 2
X = dataset[:, 0:n_measurements]
Y_1 = dataset[:, n_measurements+n_parameters:
    n_measurements+n_parameters+n_outputs-1]
Y_2 = dataset[:, n_measurements+n_parameters+1:
    n_measurements+n_parameters+n_outputs]

X = tf.keras.utils.normalize(X)
```

```
# %%  Autokeras
reg1 = StructuredDataRegressor(max_trials=15, directory=
    os.path.normpath('C:/ai1'), overwrite=True, loss='
    mean_squared_error', tuner='greedy')
reg1.fit(x=X, y=Y_1, verbose=2, batch_size=40, epochs=25)
model1 = reg1.export_model()
model1.summary()

reg2 = StructuredDataRegressor(max_trials=15, directory=
    os.path.normpath('C:/ai2'), overwrite=True, loss='
    mean_squared_error', tuner='greedy')
reg2.fit(x=X, y=Y_2, verbose=2, batch_size=40, epochs=25)
model2 = reg2.export_model()
model2.summary()

# %% Test Data
test_rows = 999
random_test_data = np.random.randint(0, len(data)-
    test_rows)
# random_test_data = 1

test_x  = data_test[random_test_data:random_test_data +
    test_rows, 0:n_measurements]
test_p  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements:n_measurements+n_parameters]
test_y  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters:n_measurements+
    n_parameters+3]
test_ju = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+3:
    n_measurements+n_parameters+5]
```

```
test_b  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+5:]

test_u1 = test_y[:, 0]
test_u2 = test_y[:, 1]
test_u  = test_y[:, 0:2]
test_j  = test_y[:, 2]

test_x = tf.keras.utils.normalize(test_x)
# %% Prediction - open loop
prediction_open_loop_1 = model1.predict(test_x)
prediction_open_loop_2 = model2.predict(test_x)
J_open_loop = []

for i in range(test_rows):
    plant_open_loop = hex3_output([prediction_open_loop_1
        [i], prediction_open_loop_2[i]], test_p[i])
    J_open_loop.append(plant_open_loop[0][0])

# %% Prediction - closed loop
tol = 1e-7
meas_dict = {'T0'   :  0, 'T1'   :  1, 'T2'    :  2, 'T3'
      :  3,
             'Th1'  :  4, 'Th2'  :  5, 'Th3'   :  6, '
                Th1e' :  7,
             'Th2e' :  8, 'Th3e' :  9, 'T'     : 10, 'wh1
                '  : 11,
             'wh2'  : 12, 'wh3'  : 13, 'alpha' : 14, '
                beta' : 15,
             'w0'   : 16}
```

```
iteration_limit = 100
J_closed_loop = []
Ju = [[], []]
prediction_closed_loop_u1 = []
prediction_closed_loop_u2 = []
alpha = 0.5
cl_fail = 0

for i in range(test_rows):
    it = 0 # iteration checker
    measurement_list = test_x[i:i+1] # to get 7 by 1
       instead of 7 by 0
    # measurement_list = np.asarray(measurement_list)
    u_opt0 = [1, 1]
    u_opt1 = model1.predict(measurement_list)[0][0]
    u_opt2 = model2.predict(measurement_list)[0][0]
    u_imp = [u_opt1, u_opt2]
    while (abs(u_imp[0] - u_opt0[0]) > tol or abs(u_imp
       [1] - u_opt0[1]) > tol) and it < iteration_limit:
        u_opt0 = u_imp

        # get measurements from plant based on prev u_opt
        # this will give us new measurements
        plant = hex3_output(u_imp, test_p[i])
        plantJ = plant[0]
        plant_measurements = plant[1] # jascke herel23,

        # collect the measurements relevant to our case
        # delete previous measurements
        measurement_list = []
        for measurement in meas_set:
```

```
            measurement_list.append( plant_measurements[
                meas_dict[measurement]] )

        # add bias to our plant measurements and
            normalize
        for j in range(len(measurement_list)):
            measurement_list[j] += test_b[i][j]
        measurement_list = tf.keras.utils.normalize(
            measurement_list)



        # calculate new u_opt based on measurements
        u_opt1 = model1.predict(measurement_list)[0][0]
        u_opt2 = model2.predict(measurement_list)[0][0]
        u_imp1 = u_opt0[0] + (u_opt1 - u_opt0[0])*alpha
        u_imp2 = u_opt0[1] + (u_opt2 - u_opt0[1])*alpha
        u_imp = [u_imp1, u_imp2]

        # iteration control
        it += 1
        print(i, it)
        if it == iteration_limit:
            cl_fail += 1
            print('Closed-loop analysis did not converge
                for datarow: {0:d} ({1:d} + 1)'.format(i +
                 1, i))

    prediction_closed_loop_u1.append(u_imp[0])
    prediction_closed_loop_u2.append(u_imp[1])
    J = hex3_output(u_imp, test_p[i])[0]
    J_closed_loop.append(J)
```

```
    # Gradiant approximation
    # J is the newest calculated temperature, plantJ is
       the previous one
    # obtained in the loop
    # Ju_meas =  hex3_output(u_imp, test_p[i])[1]
    T0 = plant_measurements[meas_dict['T0']]
    J_u1 = ((plant_measurements[meas_dict['T1']] - T0)**2
        / (plant_measurements[meas_dict['Th1']] - T0)) -
       ((plant_measurements[meas_dict['T3']] - T0)**2 / (
       plant_measurements[meas_dict['Th3']]- T0))
    J_u2 = (plant_measurements[meas_dict['T2']] - T0)**2
       / (plant_measurements[meas_dict['Th2']] - T0) - (
       plant_measurements[meas_dict['T3']] - T0)**2 / (
       plant_measurements[meas_dict['Th3']]- T0)
    Ju[0].append(J_u1)
    Ju[1].append(J_u2)

# %% Compute loss
Open_loop_loss = 0
Closed_loop_loss = 0
for i in range(len(test_x)):
    Open_loop_loss += -(J_open_loop[i] - test_j[i])
    Closed_loop_loss += -(J_closed_loop[i] - test_j[i])

# Mean squared error
MSE_J_ol = mean_squared_error(test_j, J_open_loop)
MSE_J_cl = mean_squared_error(test_j, J_closed_loop)

# %% Plotting
```

```
# Plot prediction of u1_opt : open loop - bisecting line
pyplot.title('Prediction of u1_opt - open loop')
pyplot.scatter(test_u1, prediction_open_loop_1, c='k', s
    =3)
bisecting_line = np.linspace(min(test_u1), max(test_u1))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt1 real')
pyplot.ylabel('u_opt1 predicted')
pyplot.show()

# Plot prediction of u1_opt : closed loop - bisecting
    line
pyplot.title('Prediction of u1_opt - closed loop')
pyplot.scatter(test_u1, prediction_closed_loop_u1, c='k',
     s=3)
bisecting_line = np.linspace(min(test_u1), max(test_u1))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt1 real')
pyplot.ylabel('u_opt1 predicted')
pyplot.show()

# Plot prediction of u2_opt : open loop - bisecting line
pyplot.title('Prediction of u2_opt - open loop')
pyplot.scatter(test_u2, prediction_open_loop_2, c='k', s
    =3)
bisecting_line = np.linspace(min(test_u2), max(test_u2))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt2 real')
pyplot.ylabel('u_opt2 predicted')
pyplot.show()
```

```
# Plot prediction of u1_opt : closed loop - bisecting
    line
pyplot.title('Prediction of u2_opt - closed loop')
pyplot.scatter(test_u2, prediction_closed_loop_u2, c='k',
    s=3)
bisecting_line = np.linspace(min(test_u2), max(test_u2))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('u_opt2 real')
pyplot.ylabel('u_opt2 predicted')
pyplot.show()


#Plot prediction of J_opt
pyplot.title('Prediction of J_opt closed-loop - [MS{0:d}]
    '.format(msn))
pyplot.scatter(test_j,J_closed_loop, c='k', s=3)
bisecting_line = np.linspace(min(test_j), max(test_j))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Data Sample Index')
pyplot.ylabel('J_opt')
pyplot.legend()
pyplot.show()


pyplot.hist([a-b for (a, b) in zip(test_j, J_open_loop)],
    bins=50, color='k', ec='white')
pyplot.xlabel('Predicted residual')
pyplot.ylabel('Frequency')
pyplot.show()

pyplot.hist([a-b for (a, b) in zip(test_j, J_closed_loop)
```

```
    ], bins=50, range=(0, 0.4), color='k', ec='white')
pyplot.xlabel('Predicted residual')
pyplot.ylabel('Frequency')
pyplot.show()


model1.summary()
model2.summary()
print(MSE_J_ol, MSE_J_cl, "\n", cl_fail)
```

A.3 Python Code 3

Listing 3: Prediction of temperature

```
  # -*- coding: utf-8 -*-
"""
Created on Sat Nov 28 10:13:34 2020

@author: Sommerjobb
"""

import numpy as np
import pandas as pd
import tensorflow as tf
import os as os


from autokeras import StructuredDataRegressor
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
from hex3_output import hex3_output

# %% Meta
# def neural_network_model(csv_data, csv_test_data, msn,
    meas_set):
csv_data = 'grad_meas_3.xlsx'
csv_test_data = 'grad_meas_3_test.xlsx'
csv_opt_split = 'grad_meas_3_opt_split.xlsx'
msn = 3
# meas_set = ['T0', 'T1', 'Th1', 'T2', 'Th2', 'T3', 'Th3
    ']
# meas_set = ['T0', 'Th1', 'Th2', 'Th3', 'Th1e', 'Th2e',
```

```
    'Th3e']
meas_set = ['T0', 'T', 'Th1e', 'Th2e', 'Th3e', 'w0', 'wh1
    ', 'wh2', 'wh3']
# meas_set = ['T0', 'T', 'Th1', 'Th2', 'Th3', 'alpha', '
    beta']


# %% Load the dataset
data = pd.read_excel(csv_data)
data_test = pd.read_excel(csv_test_data)
data_opt_split = pd.read_excel(csv_opt_split, header=None
    )
dataset = np.array(data, dtype=np.float32)
data_test = np.array(data_test, dtype=np.float32)
data_opt_split = np.array(data_opt_split, dtype=np.
    float32)
np.random.shuffle(dataset) # randomise the data
# np.random.shuffle(data_test) # randomise the data


# %% Split into input (X) and output (Y) variables
n_measurements = len(meas_set)        #number of
    measurements vary between 7 and 9
n_parameters = 11
n_outputs = 1
X = dataset[:, 0:n_measurements] # float32 is the
    standard for tensorflow
Y = dataset[:, n_measurements+n_parameters+2:
    n_measurements+n_parameters+2+n_outputs]

X = tf.keras.utils.normalize(X)
```

```
#


XY = X
XY[:,:-1] = Y
XY=XY[XY[:,0].argsort()]
# %%  Autokeras
reg = StructuredDataRegressor(max_trials=3,seed=1,
    directory=os.path.normpath('C:/Temperature'),
    overwrite=True, loss='mean_squared_error', tuner='
    bayesian')
reg.fit(x=XY[0:400, 0:8], y=XY[0:400, 8], validation_data
    =(XY[400:600, 0:8], XY[400:600, 8]), verbose=2,
    batch_size=40, epochs=25)
model = reg.export_model()
model.summary()

# %% Test Data
test_rows = 15
random_test_data = np.random.randint(0, len(data)-
    test_rows)
# random_test_data = 1

test_x  = data_test[random_test_data:random_test_data +
    test_rows, 0:n_measurements]
test_p  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements:n_measurements+n_parameters]
test_y  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters:n_measurements+
    n_parameters+3]
test_ju = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+3:
```

```
    n_measurements+n_parameters+5]
test_b  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+5:]

test_opt_split = data_opt_split[:, random_test_data:
    random_test_data + test_rows]

test_u1 = test_y[:, 0]
test_u2 = test_y[:, 1]
test_u  = test_y[:, 0:2]
test_j  = test_y[:, 2]

test_x = tf.keras.utils.normalize(test_x)



# %% Prediction - open loop
prediction_open_loop = reg.predict(test_x)
# pyplot.scatter(prediction_open_loop, test_j)
# bisecting_line = np.linspace(min(test_j), max(test_j))
# pyplot.plot(bisecting_line, bisecting_line)
# pyplot.show()

# pyplot.title('Prediction of u1_opt - open loop')
mse_ol = mean_squared_error(test_j, prediction_open_loop)
pyplot.title('Prediction␣of␣temperature␣-␣MSE␣=␣{0:.4f}'.
    format(mse_ol))
pyplot.scatter(test_j, prediction_open_loop,  c='k', s=3)
bisecting_line = np.linspace(min(test_j), max(test_j))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Actual␣temperature')
pyplot.ylabel('Predicted␣temperature')
```

```
pyplot.show()

# %% Closed loop analysis

prediction_closed_loop = []
J_opt_list = []
invalid_tot = 0

tol = 0.5
iteration_limit = 500
delta = 0.005
meas_dict = {'T0'   :  0, 'T1'   :  1, 'T2'    :  2, 'T3'
      :  3,
             'Th1'  :  4, 'Th2'  :  5, 'Th3'   :  6, '
                Th1e' :  7,
             'Th2e' :  8, 'Th3e' :  9, 'T'     : 10, 'wh1
                '  : 11,
             'wh2'  : 12, 'wh3'  : 13, 'alpha' : 14, '
                beta' : 15,
             'w0'   : 16}

for i in range(test_rows):
    it = 0

    plant_opt = hex3_output([test_opt_split[0, i],
       test_opt_split[1, i]], test_p[i])
    J_opt = plant_opt[0]
    J_opt_list.append(J_opt)

    u1_0 = 0.401851312      # these values are taken from
    u2_0 = 0.505778624      # grad_meas_1_opt.xlxs
```

```
J_0 = 132.3978169

u1 = 0.10
u2 = 0.60
J = prediction_open_loop[i]

Ju1 = (J - J_0)/(u1 - u1_0)
Ju2 = (J - J_0)/(u2 - u2_0)

while ((abs(Ju1) > tol or abs(Ju2) > tol) and it <
    iteration_limit):
    ############################## Update based on u1
        ##########################
    # update 0-data
    u1_0 = u1
    J_0 = J

    # update inputs
    u1 += delta*np.sign(Ju1[0])
    if u1 > 1 or u1 < 0:
        print("Invalid u1: ", u1)
        invalid_tot += 1
        break

    # get new measurements from plant
    plant = hex3_output([u1, u2], test_p[i])
    plantJ = plant[0]
    plant_measurements = plant[1]

    # collect the measurements relevant to our case
    # delete previous measurements
```

```
measurement_list = []
for measurement in meas_set:
    measurement_list.append( plant_measurements[
        meas_dict[measurement]] )

# add bias to our plant measurements and
    normalize
for j in range(len(measurement_list)):
    measurement_list[j] += test_b[i][j]
measurement_list = tf.keras.utils.normalize(
    measurement_list)
# measurement_list = np.array(measurement_list,
    dtype=np.float32)
J = model.predict(measurement_list)[0]

if (u1 - u1_0) == 0: # avoid divison by zero
    Ju1 = np.array([0])
else:
    Ju1 = (J-J_0)/(u1 - u1_0)

print(J, J_0, abs(J - J_0), Ju1, Ju2, u1, u2)
# iteration control
it += 1
print(i, it, 'u1')
if it == iteration_limit:
    print('Closed-loop analysis did not converge
        for datarow: {0:d} ({1:d} + 1)'.format(i +
         1, i))


############################## Update based on u2
```

```
    ##########################


u2_0 = u2
J_0 = J
u2 += delta*np.sign(Ju2[0])

if u2 > 1 or u2 < 0:
    print("Invalid u2: ", u2, i)
    invalid_tot += 1
    break

# get new measurements from plant
plant = hex3_output([u1, u2], test_p[i])
plantJ = plant[0]
plant_measurements = plant[1]

# collect the measurements relevant to our case
# delete previous measurements
measurement_list = []
for measurement in meas_set:
    measurement_list.append( plant_measurements[
        meas_dict[measurement]] )

# add bias to our plant measurements and
    normalize
for j in range(len(measurement_list)):
    measurement_list[j] += test_b[i][j]
measurement_list = tf.keras.utils.normalize(
    measurement_list)
# measurement_list = np.array(measurement_list,
```

```
            dtype=np.float32)
        J = model.predict(measurement_list)[0]

        if (u2 - u2_0) == 0: # avoid divison by zero
            Ju2 = np.array([0])
        else:
            Ju2 = (J-J_0)/(u2 - u2_0)

        print(J, J_0, abs(J - J_0), Ju1, Ju2, u1, u2)
        # iteration control
        it += 1
        print(i, it, 'u2')
        if it == iteration_limit:
            print('Closed-loop analysis did not converge
                for datarow: {0:d} ({1:d} + 1)'.format(i +
                 1, i))

    prediction_closed_loop.append(plantJ)
print("Invalid: ", invalid_tot)

#remove nan
it_nan = 0
while it_nan < len(prediction_closed_loop):
    if np.isnan(prediction_closed_loop[it_nan]):
        del prediction_closed_loop[it_nan]
        del J_opt_list[it_nan]
        it_nan -= 1
    it_nan += 1


#plot
```

```
mse = mean_squared_error(J_opt_list,
    prediction_closed_loop)
pyplot.title('Temperature achieved in closed loop - MSE =
    {0:.4f}'.format(mse))
pyplot.scatter(J_opt_list, prediction_closed_loop,  c='k'
    , s=3)
bisecting_line = np.linspace(min(J_opt_list), max(
    J_opt_list))
pyplot.plot(bisecting_line, bisecting_line,  c='k')
pyplot.xlabel('Optimal temperature')
pyplot.ylabel('Achieved temperature')

pyplot.show()

res = [a-b for (a, b) in zip(J_opt_list,
    prediction_closed_loop)]

pyplot.hist(res, bins=50, color='k', ec='white', range
    =(0,1))
pyplot.xlabel('Predicted residual')
pyplot.ylabel('Frequency')
pyplot.show()
# %% Compute loss
Open_loop_loss = 0
for i in range(len(test_x)):
    Open_loop_loss += -(prediction_open_loop[i] - test_j)

# Mean squared error
MSE_J_ol = mean_squared_error(test_j,
    prediction_open_loop)
```

```
# %% Print model summary
model.summary()
```

A.4 Python Code 4

Listing 4: Prediction of c

```
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 17 21:00:05 2020

@author: Espen Karlsen
"""

import numpy as np
import pandas as pd
import tensorflow as tf
import itertools as itt
import os as os
import time as time
from autokeras import StructuredDataRegressor
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras import metrics
from matplotlib import pyplot
from sklearn.preprocessing import normalize
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from hex3_output import hex3_output

# %% Meta
# def neural_network_model(csv_data, csv_test_data, msn,
    meas_set):
csv_data = 'grad_meas_4.xlsx'
```

```
csv_test_data = 'grad_meas_4_test.xlsx'
csv_opt_split = 'grad_meas_4_opt_split.xlsx'
msn = 4
# meas_set = ['T0', 'T1', 'Th1', 'T2', 'Th2', 'T3', 'Th3
    ']
# meas_set = ['T0', 'Th1', 'Th2', 'Th3', 'Th1e', 'Th2e',
    'Th3e']
# meas_set = ['T0', 'T', 'Th1e', 'Th2e', 'Th3e', 'w0', '
    wh1', 'wh2', 'wh3']
meas_set = ['T0', 'T', 'Th1', 'Th2', 'Th3', 'alpha', '
    beta']

# %% Load the dataset
data = pd.read_excel(csv_data)
data_test = pd.read_excel(csv_test_data)
data_opt_split = pd.read_excel(csv_opt_split, header=None
    )
# data = data.drop(data[(data['u_opt_2'] > 0.59) & (data
    ['u_opt_2'] < 0.61)].index)
#Dinesh proposal

dataset = np.array(data)
data_test = np.array(data_test)
data_opt_split = np.array(data_opt_split, dtype=np.
    float32)
np.random.shuffle(dataset) # randomise the data
# np.random.shuffle(data_test) # randomise the data


# %% Split into input (X) and output (Y) variables
n_measurements = len(meas_set)        #number of
```

```
    measurements vary between 7 and 9
n_parameters = 11
n_outputs = 2
X = dataset[:, 0:n_measurements]
Y = dataset[:, n_measurements+n_parameters+3:
    n_measurements+n_parameters+3+n_outputs]

X = tf.keras.utils.normalize(X)

# %%
# tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir='logs/OneModel{}'.format(int(time.time())))
# earlystopping_callback = tf.keras.callbacks.
    EarlyStopping(monitor='val_loss', patience=200)

reg = StructuredDataRegressor(max_trials=15, directory=os
    .path.normpath('C:/'), overwrite=True, loss='
    mean_squared_error', tuner='bayesian')

reg.fit(x=X, y=Y, verbose=2, batch_size=40, epochs=25)

model = reg.export_model()

model.summary()

## model.predict(x_train)

# %% Test Data
test_rows = 999
random_test_data = np.random.randint(0, len(data)-
    test_rows)
```

```
# random_test_data = 1

test_x  = data_test[random_test_data:random_test_data +
    test_rows, 0:n_measurements]
test_p  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements:n_measurements+n_parameters]
test_y  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters:n_measurements+
    n_parameters+3]
test_ju = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+3:
    n_measurements+n_parameters+5]
test_b  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+5:]

test_opt_split = data_opt_split[:, random_test_data:
    random_test_data + test_rows]

test_u1 = test_y[:, 0]
test_u2 = test_y[:, 1]
test_u  = test_y[:, 0:2]
test_j  = test_y[:, 2]

test_x = tf.keras.utils.normalize(test_x)


# %% Prediction - open loop
prediction_open_loop = model.predict(test_x)
mse_ol_1 = mean_squared_error(test_ju[:,0],
    prediction_open_loop[:,0])
mse_ol_2 = mean_squared_error(test_ju[:,1],
```

```
    prediction_open_loop[:,1])

pyplot.scatter(prediction_open_loop[:,0], test_ju[:,0],
    c='k', s=3)
pyplot.title('Prediction of Ju1 - MSE = {0:.4f}'.format(
    mse_ol_1))
bisecting_line = np.linspace(min(test_ju[:,0]), max(
    test_ju[:,0]))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Predicted Ju1')
pyplot.xlabel('Real Ju1')
pyplot.show()
pyplot.scatter(prediction_open_loop[:,1], test_ju[:,1],
    c='k', s=3)
pyplot.title('Prediction of Ju2 - MSE = {0:.4f}'.format(
    mse_ol_2))
bisecting_line = np.linspace(min(test_ju[:,1]), max(
    test_ju[:,1]))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Predicted Ju2')
pyplot.xlabel('Real Ju2')
pyplot.show()
# %% Prediction - closed loop
invalid_tot = 0

iteration_limit = 200
tol = 0.5
delta = 0.001
meas_dict = {'T0' : 0, 'T1' : 1, 'T2' : 2, 'T3'
      : 3,
             'Th1' : 4, 'Th2' : 5, 'Th3' : 6, '
```

```
                Th1e' :  7,
            'Th2e' :  8, 'Th3e' :  9, 'T'     : 10, 'wh1
                '  : 11,
            'wh2'  : 12, 'wh3'  : 13, 'alpha' : 14, '
                beta' : 15,
            'w0'   : 16}


J_closed_loop = []
J_opt_list = []
prediction_closed_loop_u1 = []
prediction_closed_loop_u2 = []
alpha = 0.5

for i in range(test_rows):
    it = 0

    plant_opt = hex3_output([test_opt_split[0, i],
        test_opt_split[1, i]], test_p[i])
    J_opt = plant_opt[0]
    J_opt_list.append(J_opt)

    u1_0 = 0.401851312      # these values are taken from
    u2_0 = 0.505778624      # grad_meas_1_opt.xlxs
    J_0 = 132.3978169

    u1 = 0.10
    u2 = 0.60

    Ju1 = prediction_open_loop[i, 0]
    Ju2 = prediction_open_loop[i, 1]
```

```
while ((abs(Ju1) > tol or abs(Ju2) > tol) and it <
    iteration_limit):

    u1 -= delta*np.sign(Ju1)
    if u1 > 1 or u1 < 0:
        print("Invalid u1: ", u1)
        invalid_tot += 1
        break

    u2 -= delta*np.sign(Ju2)

    if u2 > 1 or u2 < 0:
        print("Invalid u2: ", u2, i)
        invalid_tot += 1
        break
    # get measurements from plant based on prev u_opt
    # this will give us new measurements
    plant = hex3_output([u1, u2], test_p[i])
    plantJ = plant[0]
    plant_measurements = plant[1] # jascke herel23,

    # collect the measurements relevant to our case
    # delete previous measurements
    measurement_list = []
    for measurement in meas_set:
        measurement_list.append( plant_measurements[
            meas_dict[measurement]] )

    # add bias to our plant measurements and
```

```
            normalize
        for j in range(len(measurement_list)):
            measurement_list[j] += test_b[i][j]
        measurement_list = tf.keras.utils.normalize(
            measurement_list)

        Ju = model.predict(measurement_list)[0]
        Ju1 = Ju[0]
        Ju2 = Ju[1]
        # iteration control
        it += 1
        print(i, it, u1, u2, Ju1, Ju2)
        if it == iteration_limit:
            print('Closed-loop analysis did not converge
                for datarow: {0:d} ({1:d} + 1)'.format(i +
                1, i))

    prediction_closed_loop_u1.append(u1)
    prediction_closed_loop_u2.append(u2)
    J = hex3_output([u1, u2], test_p[i])[0]
    J_closed_loop.append(J)
print("Invalid: ", invalid_tot)


mse = mean_squared_error(J_opt_list, J_closed_loop)
pyplot.title('Temperature achieved in closed loop - MSE =
    {0:.4f}'.format(mse))
pyplot.scatter(J_opt_list, J_closed_loop, c='k', s=3)
bisecting_line = np.linspace(min(J_opt_list), max(
    J_opt_list))
pyplot.plot(bisecting_line, bisecting_line, c='k')
```

```
pyplot.xlabel('Optimal temperature')
pyplot.ylabel('Achieved temperature')
pyplot.show()


res = [a-b for (a, b) in zip(J_opt_list, J_closed_loop)]


pyplot.hist(res, bins=50, color='k', ec='white', range
    =(0,1))
pyplot.xlabel('Predicted residual')
pyplot.ylabel('Frequency')
pyplot.show()


model.summary()
```

A.5 Python Code 5

Listing 5: Prediction of c - two models

```
# -*- coding: utf-8 -*-
"""
Created on Sat Dec 12 11:12:57 2020

@author: Sommerjobb
"""



import numpy as np
import pandas as pd
import tensorflow as tf
import itertools as itt
import os as os
import time as time
from autokeras import StructuredDataRegressor
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras import metrics
from matplotlib import pyplot
from sklearn.preprocessing import normalize
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from hex3_output import hex3_output

# %% Meta
# def neural_network_model(csv_data, csv_test_data, msn,
    meas_set):
```

```
csv_data = 'grad_meas_4.xlsx'
csv_test_data = 'grad_meas_4_test.xlsx'
csv_opt_split = 'grad_meas_4_opt_split.xlsx'
msn = 4
# meas_set = ['T0', 'T1', 'Th1', 'T2', 'Th2', 'T3', 'Th3
    ']
# meas_set = ['T0', 'Th1', 'Th2', 'Th3', 'Th1e', 'Th2e',
    'Th3e']
# meas_set = ['T0', 'T', 'Th1e', 'Th2e', 'Th3e', 'w0', '
    wh1', 'wh2', 'wh3']
meas_set = ['T0', 'T', 'Th1', 'Th2', 'Th3', 'alpha', '
    beta']

# %% Load the dataset
data = pd.read_excel(csv_data)
data_test = pd.read_excel(csv_test_data)
data_opt_split = pd.read_excel(csv_opt_split, header=None
    )
# data = data.drop(data[(data['u_opt_2'] > 0.59) & (data
    ['u_opt_2'] < 0.61)].index)
#Dinesh proposal

dataset = np.array(data)
data_test = np.array(data_test)
data_opt_split = np.array(data_opt_split, dtype=np.
    float32)
np.random.shuffle(dataset) # randomise the data
# np.random.shuffle(data_test) # randomise the data



# %% Split into input (X) and output (Y) variables
```

```
n_measurements = len(meas_set)          #number of
    measurements vary between 7 and 9
n_parameters = 11
n_outputs = 2
X = dataset[:, 0:n_measurements]
Y1 = dataset[:, -2]
Y2 = dataset[:, -1]



X = tf.keras.utils.normalize(X)

# %%
# tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir='logs/OneModel{}'.format(int(time.time())))
# earlystopping_callback = tf.keras.callbacks.
    EarlyStopping(monitor='val_loss', patience=200)

reg1 = StructuredDataRegressor(max_trials=15, directory=
    os.path.normpath('C:/GradOne'), overwrite=True, loss='
    mean_squared_error', tuner='bayesian')
reg1.fit(x=X, y=Y1, verbose=2, batch_size=2, epochs=35)
model1 = reg1.export_model()
model1.summary()

reg2 = StructuredDataRegressor(max_trials=15, directory=
    os.path.normpath('C:/GradTwo'), overwrite=True, loss='
    mean_squared_error', tuner='bayesian')
reg2.fit(x=X, y=Y2, verbose=2, batch_size=2, epochs=35)
model2 = reg2.export_model()
model2.summary()
## model.predict(x_train)
```

```
# %% Test Data
test_rows = 15
random_test_data = np.random.randint(0, len(data)-
    test_rows)
# random_test_data = 1

test_x  = data_test[random_test_data:random_test_data +
    test_rows, 0:n_measurements]
test_p  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements:n_measurements+n_parameters]
test_y  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters:n_measurements+
    n_parameters+3]
test_ju = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+3:
    n_measurements+n_parameters+5]
test_b  = data_test[random_test_data:random_test_data +
    test_rows, n_measurements+n_parameters+5:]

test_opt_split = data_opt_split[:, random_test_data:
    random_test_data + test_rows]

test_u1 = test_y[:, 0]
test_u2 = test_y[:, 1]
test_u  = test_y[:, 0:2]
test_j  = test_y[:, 2]

test_x = tf.keras.utils.normalize(test_x)
```

```
# %% Prediction - open loop
prediction_open_loop1 = model1.predict(test_x)
prediction_open_loop2 = model2.predict(test_x)
mse_ol_1 = mean_squared_error(test_ju[:,0],
    prediction_open_loop1)
mse_ol_2 = mean_squared_error(test_ju[:,1],
    prediction_open_loop2)

pyplot.scatter(prediction_open_loop1, test_ju[:,0], c='k
    ', s=3)
pyplot.title('Prediction of Ju1 - MSE = {0:.4f}'.format(
    mse_ol_1))
bisecting_line = np.linspace(min(test_ju[:,0]), max(
    test_ju[:,0]))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Predicted Ju1')
pyplot.xlabel('Real Ju1')
pyplot.show()
pyplot.scatter(prediction_open_loop2, test_ju[:,1], c='k
    ', s=3)
pyplot.title('Prediction of Ju2 - MSE = {0:.4f}'.format(
    mse_ol_2))
bisecting_line = np.linspace(min(test_ju[:,1]), max(
    test_ju[:,1]))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Predicted Ju2')
pyplot.xlabel('Real Ju2')
pyplot.show()
# %% Prediction - closed loop
invalid_tot = 0
```

```
iteration_limit = 200
tol = 0.5
delta = 0.001
meas_dict = {'T0'   :  0, 'T1'   :  1, 'T2'    :  2, 'T3'
      :  3,
             'Th1'  :  4, 'Th2'  :  5, 'Th3'   :  6, '
                Th1e' :  7,
             'Th2e' :  8, 'Th3e' :  9, 'T'     : 10, 'wh1
                '  : 11,
             'wh2'  : 12, 'wh3'  : 13, 'alpha' : 14, '
                beta' : 15,
             'w0'   : 16}


J_closed_loop = []
J_opt_list = []
prediction_closed_loop_u1 = []
prediction_closed_loop_u2 = []
alpha = 0.5

for i in range(test_rows):
    it = 0

    plant_opt = hex3_output([test_opt_split[0, i],
       test_opt_split[1, i]], test_p[i])
    J_opt = plant_opt[0]
    J_opt_list.append(J_opt)

    u1_0 = 0.401851312       # these values are taken from
    u2_0 = 0.505778624       # grad_meas_1_opt.xlxs
    J_0 = 132.3978169
```

```
u1 = 0.10
u2 = 0.60

Ju1 = prediction_open_loop1[i]
Ju2 = prediction_open_loop2[i]


while ((abs(Ju1) > tol or abs(Ju2) > tol) and it <
    iteration_limit):

    u1 -= delta*np.sign(Ju1)
    if u1 > 1 or u1 < 0:
        print("Invalid u1: ", u1)
        invalid_tot += 1
        break

    u2 -= delta*np.sign(Ju2)

    if u2 > 1 or u2 < 0:
        print("Invalid u2: ", u2, i)
        invalid_tot += 1
        break
    # get measurements from plant based on prev u_opt
    # this will give us new measurements
    plant = hex3_output([u1, u2], test_p[i])
    plantJ = plant[0]
    plant_measurements = plant[1] # jascke herel23,

    # collect the measurements relevant to our case
    # delete previous measurements
```

```
        measurement_list = []
        check = 0
        for measurement in meas_set:
            if check == 1 or check == 3 or check == 5:
                measurement_list.append(
                    plant_measurements[meas_dict[
                    measurement]][0] )
            else:
                measurement_list.append(
                    plant_measurements[meas_dict[
                    measurement]] )
        # add bias to our plant measurements and
            normalize
        for j in range(len(measurement_list)):
            measurement_list[j] += test_b[i][j]
        measurement_list = tf.keras.utils.normalize(
            measurement_list)
        measurement_list = np.asarray(measurement_list).
            astype('float32')
        Ju1 = model1.predict(measurement_list)[0]
        Ju2 = model2.predict(measurement_list)[0]
        # iteration control
        it += 1
        print(i, it, u1, u2, Ju1, Ju2)
        if it == iteration_limit:
            print('Closed-loop analysis did not converge
                for datarow: {0:d} ({1:d} + 1)'.format(i +
                1, i))

    prediction_closed_loop_u1.append(u1)
    prediction_closed_loop_u2.append(u2)
```

```
        J = hex3_output([u1, u2], test_p[i])[0]
        J_closed_loop.append(J)
print("Invalid: ", invalid_tot)


mse = mean_squared_error(J_opt_list, J_closed_loop)
pyplot.title('Temperature achieved in closed loop - MSE =
    {0:.4f}'.format(mse))
pyplot.scatter(J_opt_list, J_closed_loop, c='k', s=3)
bisecting_line = np.linspace(min(J_opt_list), max(
    J_opt_list))
pyplot.plot(bisecting_line, bisecting_line, c='k')
pyplot.xlabel('Optimal temperature')
pyplot.ylabel('Achieved temperature')
pyplot.show()


res = [(a-b)[0] for (a, b) in zip(J_opt_list,
    J_closed_loop)]

pyplot.hist(res, bins=50, color='k', ec='white', range
    =(0,5))
pyplot.xlabel('Predicted residual')
pyplot.ylabel('Frequency')
pyplot.show()


model1.summary()
model2.summary()
```

B Appendix - Case Study Figures

B.1 Case 1.2

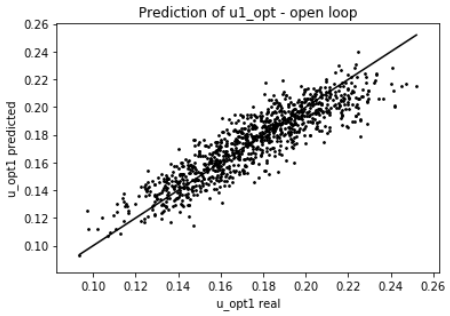

Figure 28: Prediction of α_{opt} in open loop.

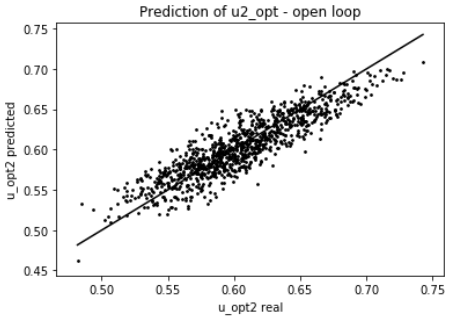

Figure 29: Prediction of β_{opt} in open loop.

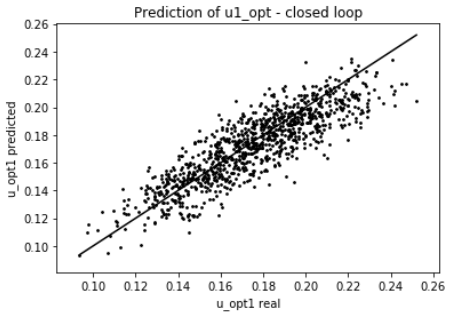

Figure 30: Prediction of α_{opt} in closed loop.

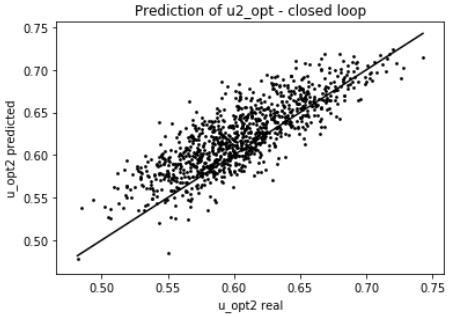

Figure 31: Prediction of β_{opt} in closed loop.

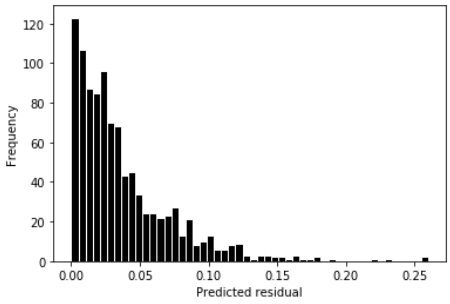

Figure 32: Open loop residual

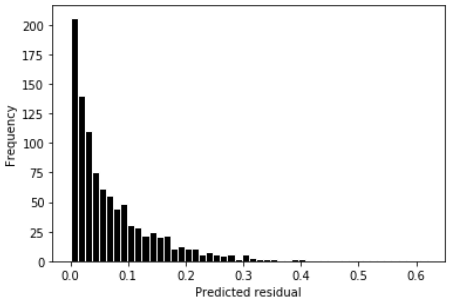

Figure 33: Closed loop residual

B.2 Case 1.3

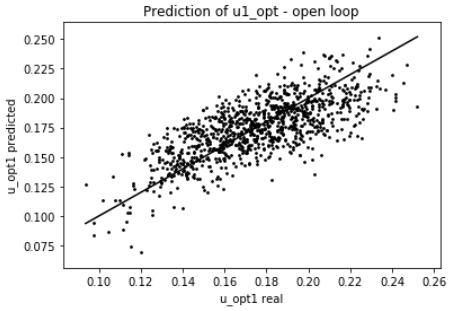

Figure 34: Prediction of α_{opt} in open loop.

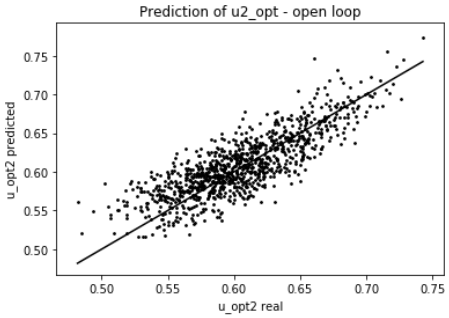

Figure 35: Prediction of β_{opt} in open loop.

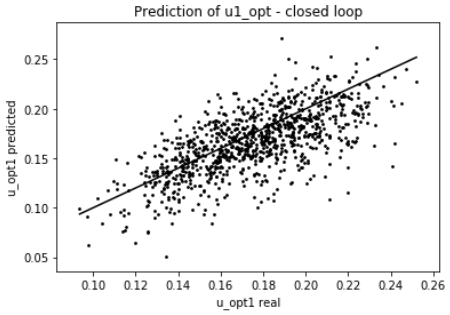

Figure 36: Prediction of α_{opt} in closed loop.

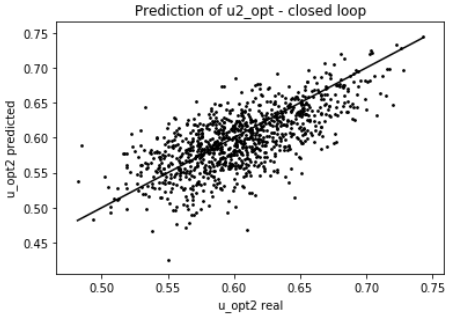

Figure 37: Prediction of β_{opt} in closed loop.

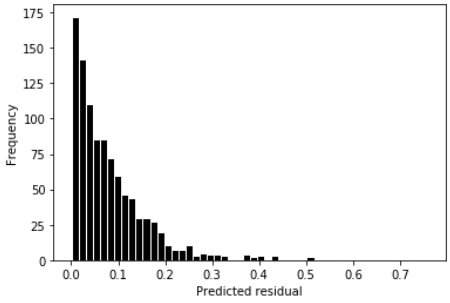

Figure 38: Open loop residual

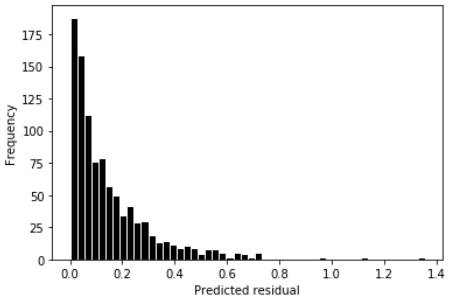

Figure 39: Closed loop residual

B.3 Case 1.4

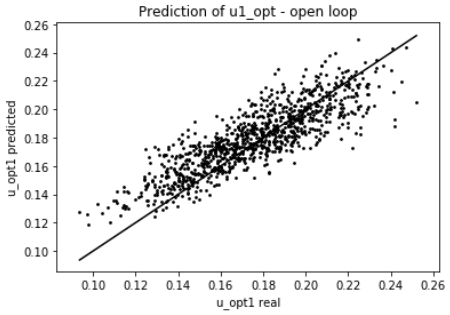

Figure 40: Prediction of α_{opt} in open loop.

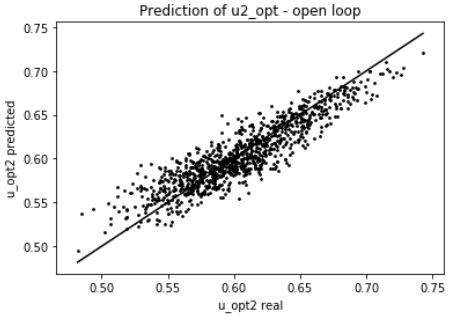

Figure 41: Prediction of β_{opt} in open loop.

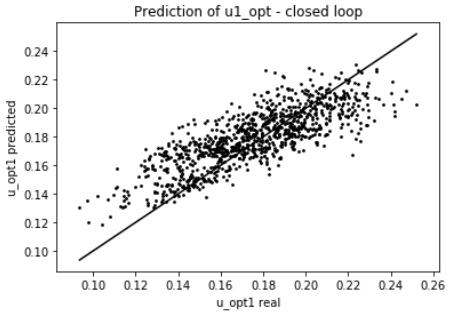

Figure 42: Prediction of α_{opt} in closed loop.

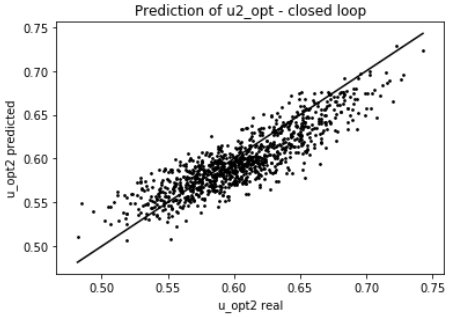

Figure 43: Prediction of β_{opt} in closed loop.

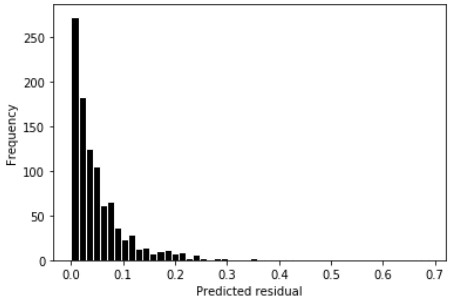

Figure 44: Open loop residual

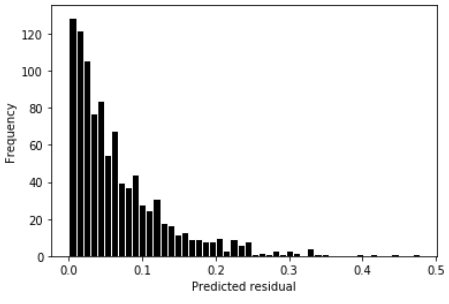

Figure 45: Closed loop residual

B.4 Case 2.2

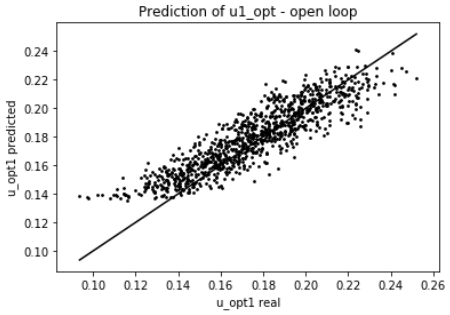

Figure 46: Prediction of α_{opt} in open loop.

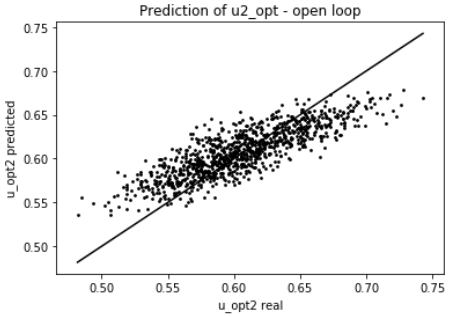

Figure 47: Prediction of β_{opt} in open loop.

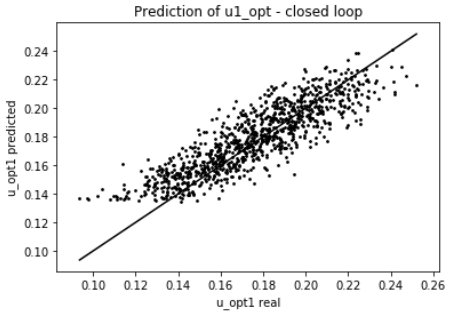

Figure 48: Prediction of α_{opt} in closed loop.

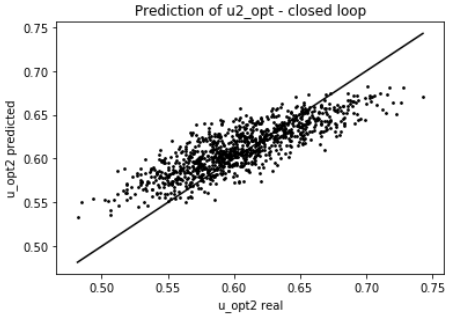

Figure 49: Prediction of β_{opt} in closed loop.

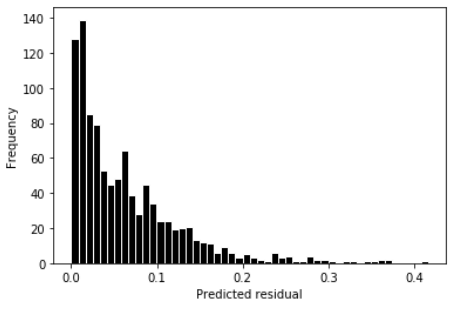

Figure 50: Open loop residual

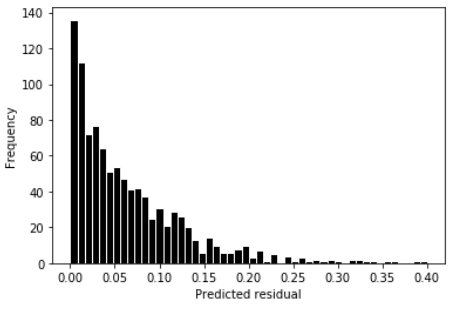

Figure 51: Closed loop residual

B.5 Case 2.3

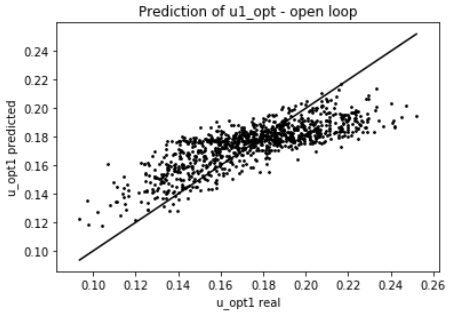

Figure 52: Prediction of α_{opt} in open loop.

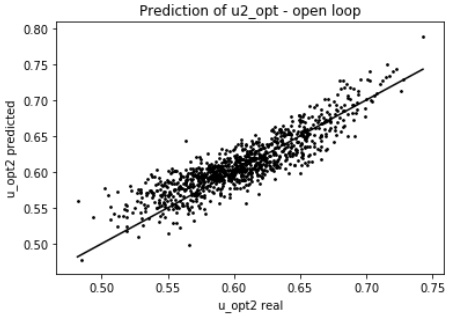

Figure 53: Prediction of β_{opt} in open loop.

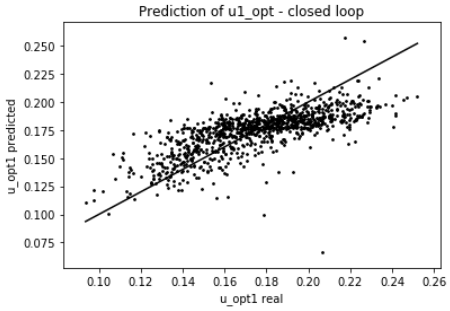

Figure 54: Prediction of α_{opt} in closed loop.

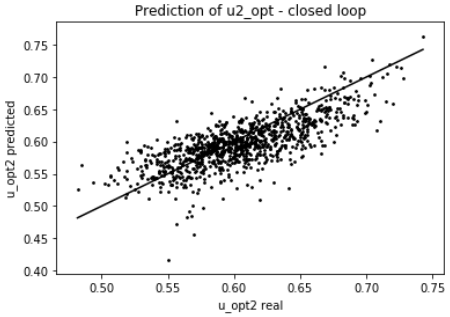

Figure 55: Prediction of β_{opt} in closed loop.

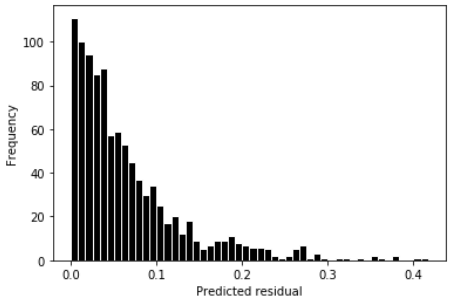

Figure 56: Open loop residual

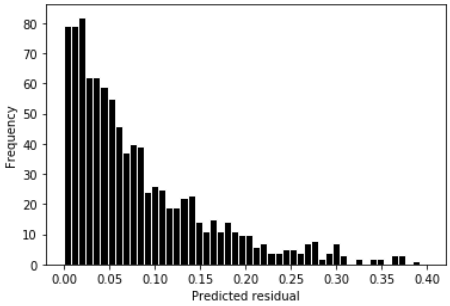

Figure 57: Closed loop residual

B.6 Case 2.4

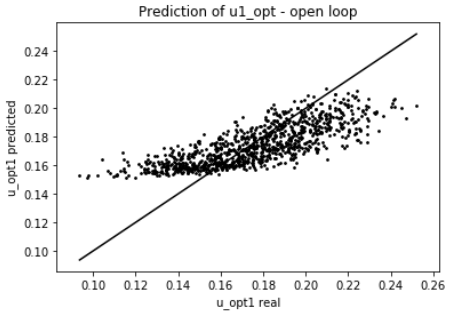

Figure 58: Prediction of α_{opt} in open loop.

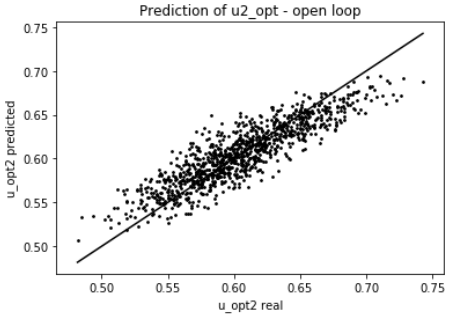

Figure 59: Prediction of β_{opt} in open loop.

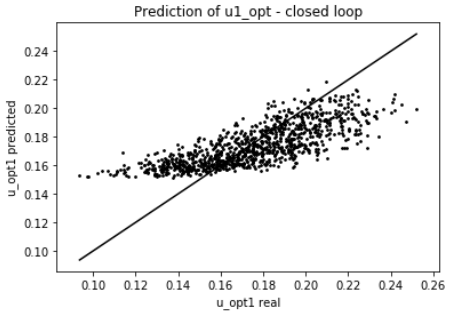

Figure 60: Prediction of α_{opt} in closed loop.

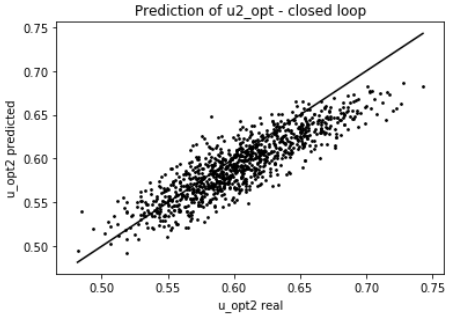

Figure 61: Prediction of β_{opt} in closed loop.

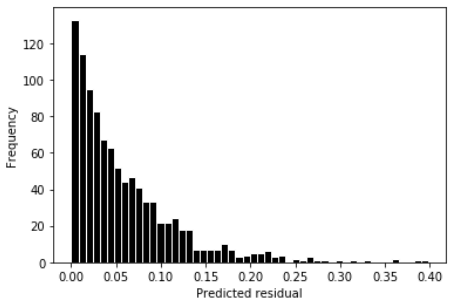

Figure 62: Open loop residual

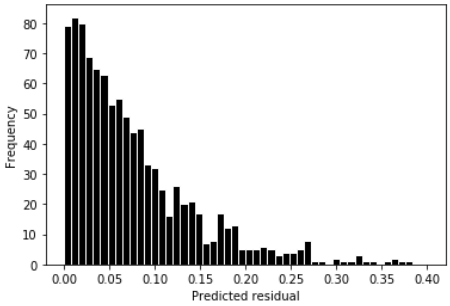

Figure 63: Closed loop residual

B.7 Case 3.2

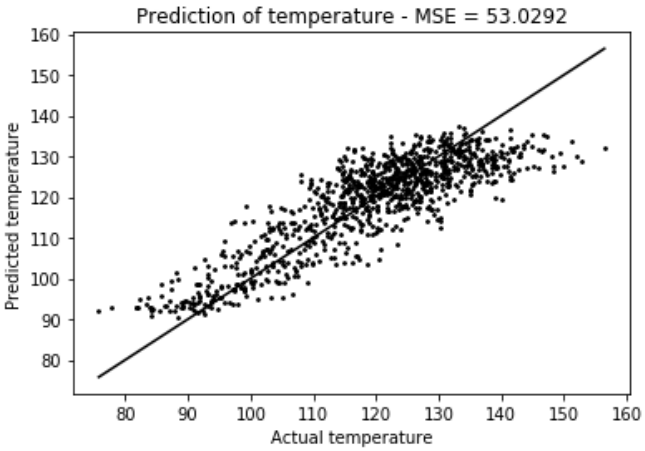

Figure 64: Prediction of T in open loop.

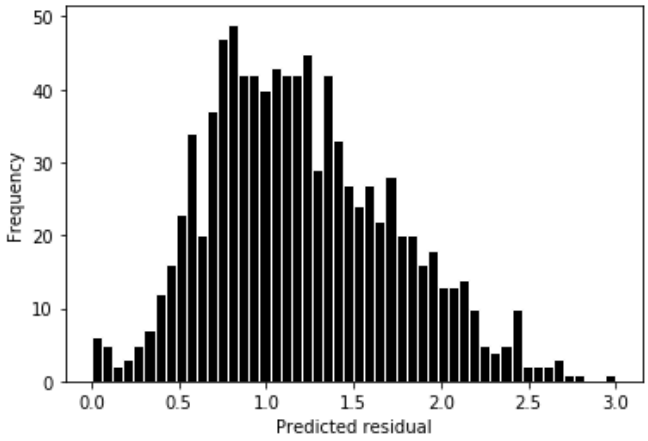

Figure 65: Residual between optimal temperature and achieved temperature in closed loop.

B.8 Case 3.3

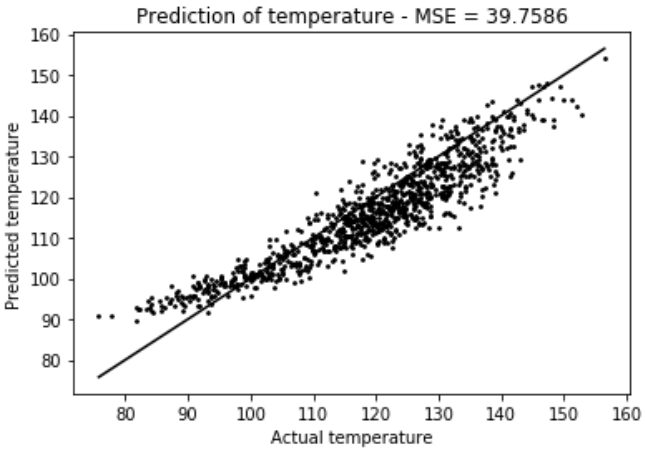

Figure 66: Prediction of T in open loop.

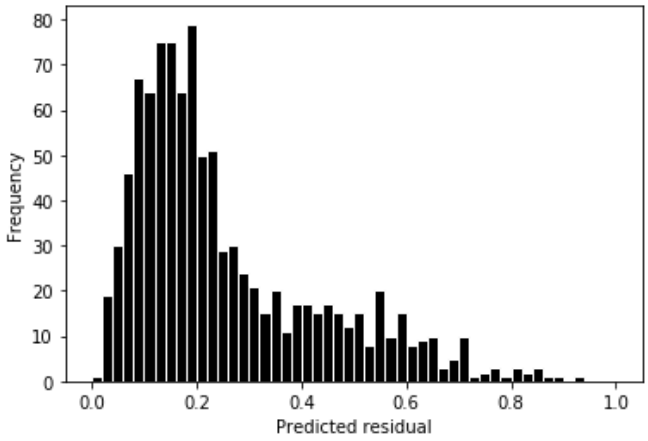

Figure 67: Residual between optimal temperature and achieved temperature in closed loop.

B.9 Case 3.4

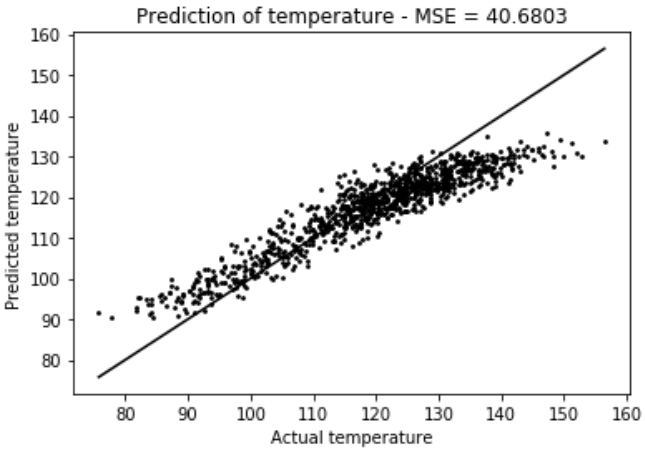

Figure 68: Prediction of T in open loop.

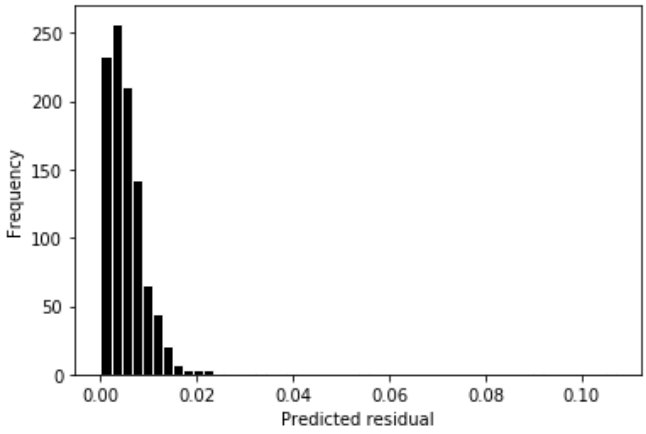

Figure 69: Residual between optimal temperature and achieved temperature in closed loop.

B.10 Case 4.1

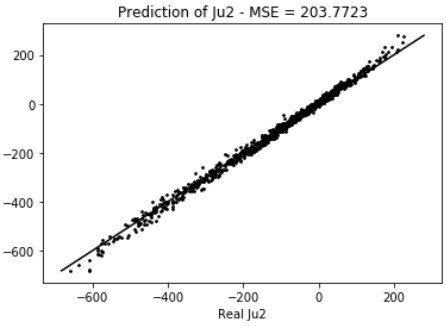

Figure 70: Prediction of c_1 in open loop.

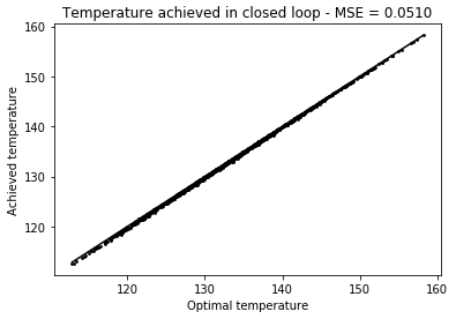

Figure 71: Prediction of c_2 in open loop.

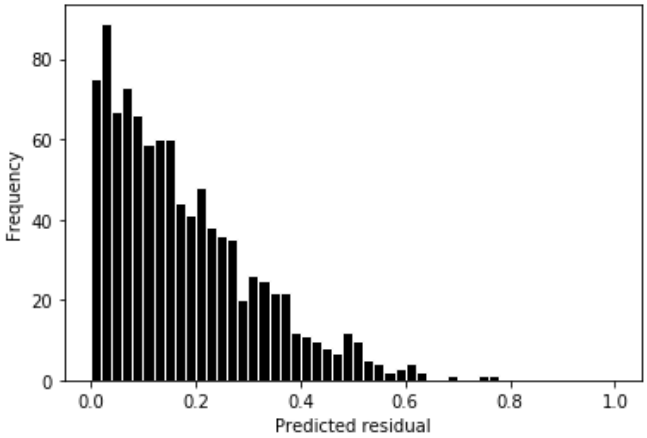

Figure 72: Closed loop analysis.

B.11 Case 4.3

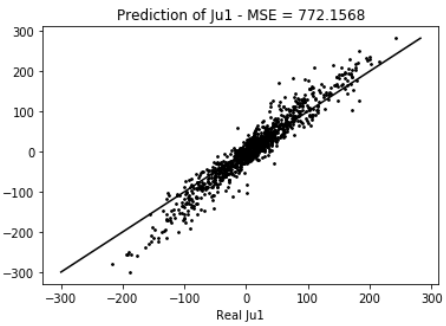

Figure 73: Prediction of c_1 in open loop.

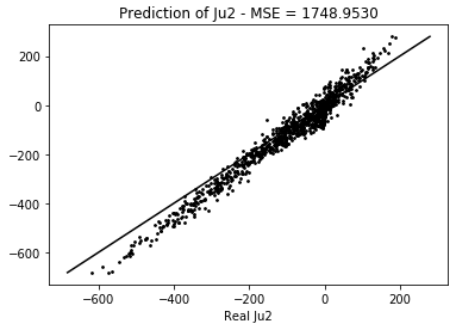

Figure 74: Prediction of c_2 in open loop.

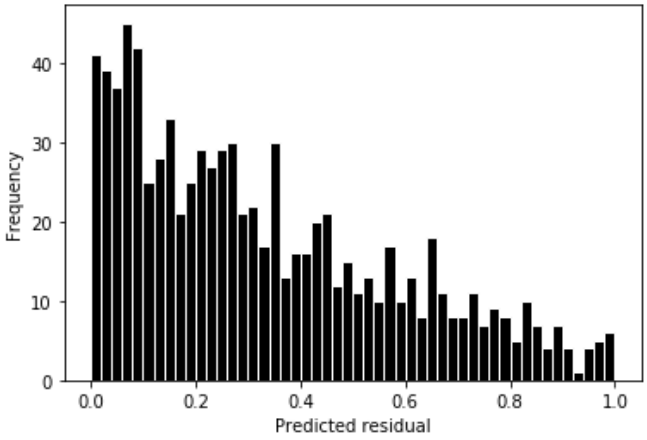

Figure 75: Closed loop analysis.

B.12 Case 4.4

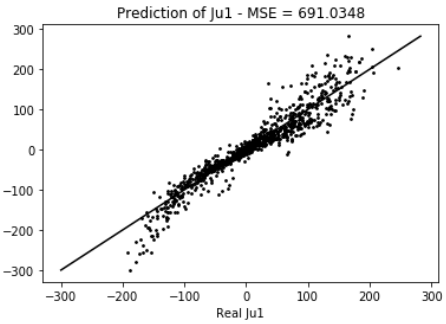

Figure 76: Prediction of c_1 in open loop.

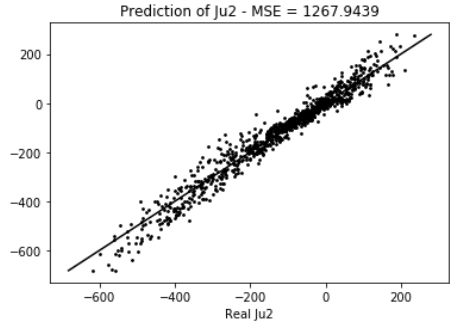

Figure 77: Prediction of c_1 in open loop.

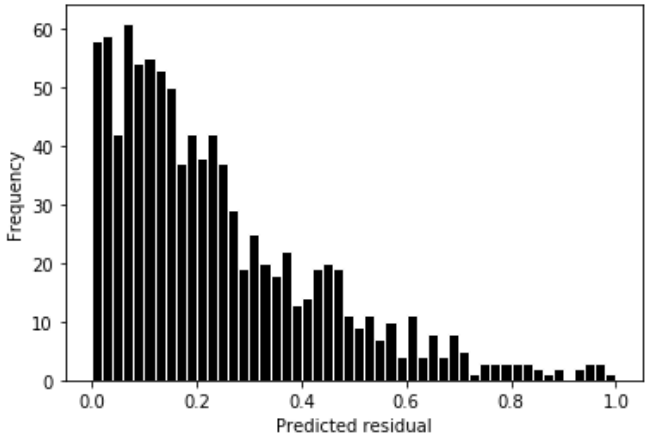

Figure 78: Closed loop analysis.

B.13 Case 5.2

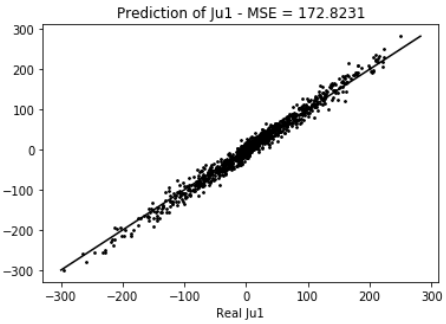

Figure 79: Prediction of c_1 in open loop.

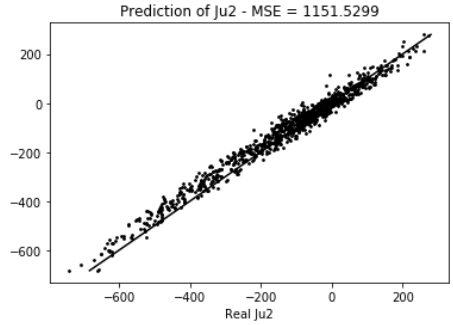

Figure 80: Prediction of c_1 in open loop.

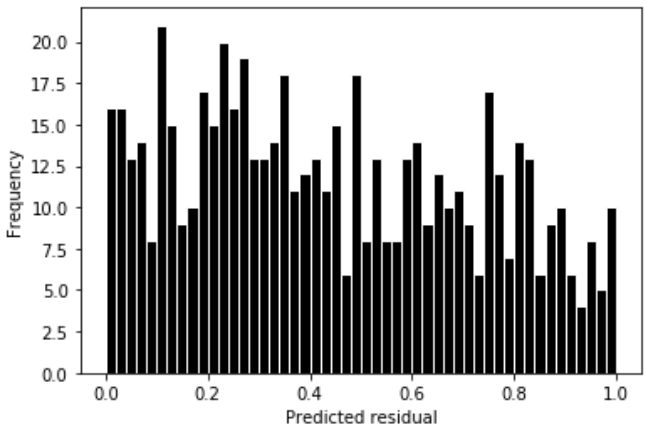

Figure 81: Closed loop analysis.

B.14 Case 5.3

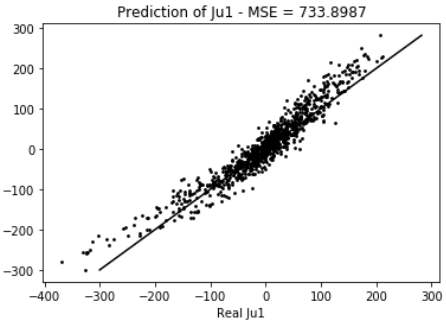

Figure 82: Prediction of c_1 in open loop.

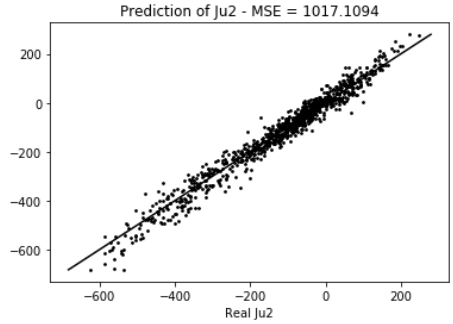

Figure 83: Prediction of c_1 in open loop.

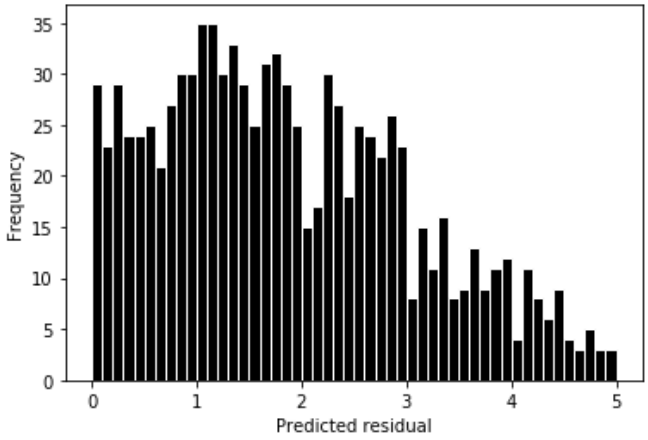

Figure 84: Closed loop analysis.

B.15 Case 5.4

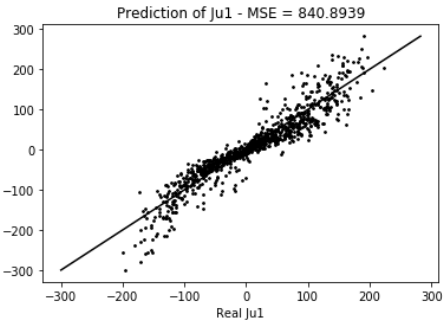

Figure 85: Prediction of c_1 in open loop.

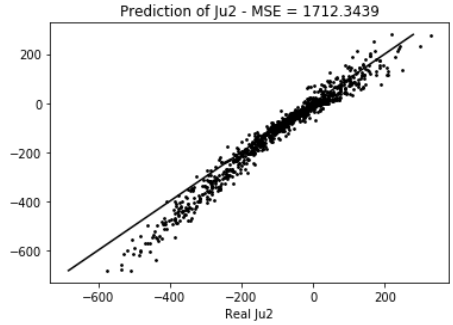

Figure 86: Prediction of c_1 in open loop.

Figure 87 shows how 5.4 created illogical inputs which achieved temperatures of larger value than the optimal value.

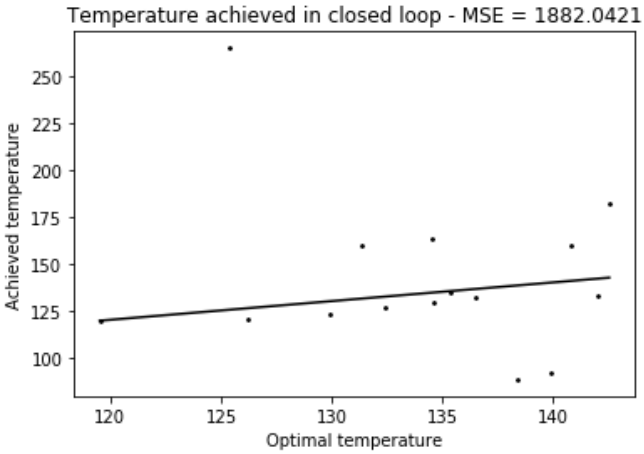

Figure 87: Closed loop analysis.