BACHELOR

Hyperparameter tuning for Artificial Neural Networks applied to inverse mapping parameter updating

Janssen, Tom M.E.

Award date:
2022

Department of Mechanical Engineering

Hyperparameter tuning for Artificial Neural Networks

applied to inverse mapping parameter updating

Bachelor Final Project

Tom Janssen

Supervisors:
dr.ir. Rob H.B. Fey
ir. Bas M. Kessels

Report number: DC 2022.057

Eindhoven, June 23, 2022

Contents

List of symbols

Table 1: List of symbols.

Symbol	Variable
i	index of neuron in input layer
j	index of neuron in hidden layer
k	index of neuron in output layer
d	number of neurons in input layer
M	number of neurons in hidden layer
c	number of neurons in output layer
η	learning rate
g	activation function
z_j	output of neuron j
y_k	output of neuron k
a_j	linear combination of inputs and weights of neuron j
$w_{ij}^{(1)}$	weight between neuron j in hidden layer 1 and neuron i in the input layer
x_i	input value from neuron i in input layer
τ	batch index
n	training sample index
E	error
t_k	true value at output k

1 Introduction

1.1 General background information

The ASM Centre of Competency, located in Beuningen, the Netherlands, is responsible for identifying new high-tech innovation opportunities for the benefit of the full ASM Pacific Technology (ASM-PT) product portfolio, consisting of the complete back-end section of the semiconductor equipment market [1]. A type of machine belonging to this portfolio is the wire bonder, responsible for connecting interconnected circuits to their packaging using thin wires. Using first-principles, this system is modelled such that insight in its (dynamical) behavior is obtained. Due to incomplete modelling of the system, the accuracy of the dynamic behavior predicted by this reference model may, however, be limited compared to real behavior of the actual system. Furthermore, due to machine-to-machine variations, originating from, e.g., manufacturing tolerances, each physical machine would benefit from a dedicated and improved model derived from the reference model. In this context, autonomous updating of the model's parameters using measured response data yields a more accurate, dedicated model, allowing for increased performance due to, e.g., dedicated controller design. For now, it is assumed that the equations of motion are modelled and discretized correctly, i.e., it is assumed that there are no model structure or discretization errors.

1.2 State of the art and Objective

Current research on autonomous model parameter updating focuses on minimizing the mismatch between a physical (measured) system and the modelled system by employing model parameter updating using an inverse mapping model [2]. This inverse mapping model enables fast estimation of parameter values based on measured output signals and features derived therefrom, and is constituted by a (trained) Artificial Neural Network (ANN) [3]. This ANN is trained offline using simulation generated data existing of sets of model parameters and their corresponding output response features. Before training the ANN using these data, the structure of the ANN and its training settings should be defined (e.g., the number of neurons/layers, the activation function(s), and the number of epochs). These 'settings' are also referred to as hyperparameters and are traditionally tuned using the engineer's experience [4]. However, dedicated software exists that automates the (optimal) tuning of hyperparameters [5], [6]. The objective of this report is to employ, analyze and compare such software to tune the hyperparameters of an ANN serving as an inverse mapping model to update model parameter values using response data.

1.3 Report outline

This report is organized as follows. First, in Chapter 2, the concept of ANNs is introduced and an explanation on how these are trained and used is given. Since this report focuses on tuning the hyperparameters of an ANN, various types of hyperparameters are explicitly discussed in Chapter 2 as well. Then, in Chapter 3, a hyperparameter study is performed seperately for each hyperparameter that is introduced in Chapter 2. In Chapter 4, multiple automatic hyperparameter tuning methods are explained. In Chapter 5, a reference neural network is discussed which is trained by a relatively inexperienced engineer. In Chapter 6, results obtained using the automatic tuning methods are compared to this reference case. Finally, in Chapter 7, conclusions and recommendations for future work are given.

2 Neural networks

In this chapter, background information about the operation of a neural network is given. First, the way a neural network works is explained in Section 2.1. Subsequently, the way a neural network is build up is explained in Section 2.2. Finally, it is explained how this structure is used to train the network to achieve good performance, i.e., accurate inferring results, in Section 2.3. Throughout this chapter, hyperparameters will be underlined and highlighted in bold face writing.

2.1 Concept

A neural network tries to find a mathematical relation between a specified input and output. This mathematical relation is modelled in the form of neurons, in which the input values are multiplied by weights and added together. Input values can go through multiple neurons and the resulting value from a neuron can again be sent to another neuron. A neural network tries to update the weights that are used in these neurons in a way that this multiplication and summation accurately predicts the corresponding output. This updating is done using training data consisting out of many input data points and there corresponding output. This updating of the weights is called training. After a model has been trained, it can predict the output based on the input up to a certain accuracy. This predicting is called inferring. Training a neural network might take hours or days depending on its complexity and its training settings, however, inferring the output from some input may only take milliseconds.

2.2 Structure

There are two types of hyperparameters that determine the structure of a neural network, these are the **number of layers** in a network and the **number of neurons per layer**. A neural network always contains at least three layers:

- One input layer
- One hidden layer
- One output layer

An example of a neural network is shown in Figure 1. Here, the input layer is green, the hidden layer is blue, and the output layer is yellow. The number of hidden layers can range from 1 to any number, while there are always only one input and one output layer. When more hidden layers are added, more multiplications and summations are performed, which enables the network to model more complex relations between the input and the output.

In Figure 1, the input layer contains 3 neurons, the hidden layer contains 5 neurons and the output layer contains 2 neurons. The amount of neurons in the input layer is always equal to the amount of input parameters. The amount of neurons in the hidden layers can be arbitrarily chosen per layer. The amount of neurons in the output layer is always equal the the number of output parameters. When more neurons are added in a hidden layer, more multiplications and summations are performed, which enables the network to model more complex relations between the input and the output.

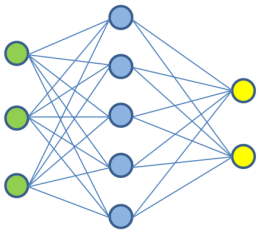


Figure 1: Basic fully connected feedforward neural network. Neurons in the input, hidden and output layer are represented in green, blue and yellow, respectively [7].

2.3 Training

Training of a neural network is done in two steps: the forward propagation step, in which the current network is used to estimate the output parameters, and the back propagation step, in which the performance is analysed and changes to the network are calculated and implemented. These two steps are explained in more detail below, after which it is explained how these steps work together to train a neural network.

2.3.1 Forward propagation

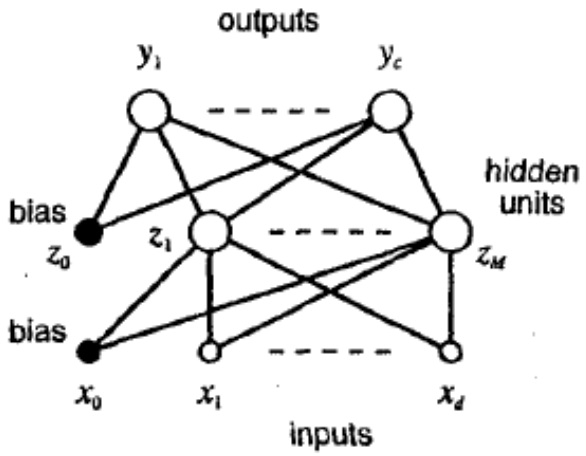


Figure 2: Simple neural network structure with d input neurons, M neurons in the hidden layer and c output neurons [8].

In forward propagation, the output z_j of the j-th neuron of the hidden layer, an example of which is seen in Figure 2, is calculated using

$$z_j = g(a_j), \tag{1}$$

where

$$a_j = \sum_{i=0}^{d} w_{ji}^{(1)} x_i \text{ with } x_0 = 1. \tag{2}$$

Here, $g()$ is the so called **activation function**. This activation function is added to enable nonlinear, in terms of inputs to the neuron, outputs of the network. Without an activation function, the output would be a linear combination of the inputs. $w_{ji}^{(1)}$ is the weight between neuron j in hidden layer 1 (in this case, there is only one hidden layer) and neuron i in the layer before it, thus one of the input neurons. x_i is the value coming from neuron i in the input layer. x_0 is always equal to 1, since this value is multiplied with $w_{j0}^{(1)}$, where $w_{j0}^{(1)}$ is also referred to as the bias. Note that the bias thus serves as an offset value. Similar to Equation (1), the value of output neuron y_k is obtained by

$$y_k = \widetilde{g}(\sum_{j=0}^{M} w_{kj}^{(2)} z_j), \text{ with } z_0 = 1, \tag{3}$$

where $\widetilde{g}()$ can be a different activation function than $g()$ (it can even differ per neuron in a layer, but this is not covered in this report for clarity) and z_0 is always equal to 1. In this way, the output y_k of a neuron in the output layer of a single hidden-layer neural network becomes

$$y_k = \widetilde{g}(\sum_{j=0}^{M} w_{kj}^{(2)} g(\sum_{i=0}^{d} w_{ji}^{(1)} x_i)). \tag{4}$$

2.3.2 Back propagation

The goal of back propagation [8] is to update the networks weights in such a way that the network best predicts the output values, or in other words, minimizes the error based on training data. This is done using an **optimizer**. In the explanation below, *mini-batch gradient descent* is used as the optimizer, but many optimizers exist that work in a slightly different way [9]. The goal of gradient descent is to update the weights in the direction that minimizes the error. This leads to

$$w_{ji,\tau+1} = w_{ji,\tau} - \eta \frac{\partial E}{\partial w_{ji}}|_{w_{ji,\tau}}, \tag{5}$$

in which $w_{ji,\tau+1}$ is the weight at neuron j in the hidden layer taking input from neuron i in the input layer at iteration $\tau + 1$, which iterates over all **batches** (a subset of the training data) in a specific **epoch** (a complete pass of the training data set through the training algorithm), η is the **learning rate** (a factor that determines how much the weights are updated), and E is the training **error**. The error can be defined as the sum of the error of all training samples n in a batch:

$$E = \sum_n E^{(n)}, \tag{6}$$

where n denotes the training sample in that batch. The partial derivative of the error can then be defined as:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^{(n)}}{\partial w_{ji}}. \tag{7}$$

The error is calculated using the **loss function**, which could, for example, be the *mean squared error* function. The mean squared error is

$$E^{(n)} = \frac{1}{2c} \sum_{k=1}^{c} (y_k^{(n)} - t_k^{(n)})^2, \tag{8}$$

where y_k is the predicted value of the network at neuron k in the output layer and where t_k is the true value of that neuron (as obtained from the training data).

Going back to the partial derivative of the error $E^{(n)}$ at the output with respect to some weight w_{ji}, part of the sum of Equation (7), we note that $E^{(n)}$ is only dependent on w_{ji} via $a_j^{(n)}$. Therefore, part of Equation 5 can be rewritten using the chain rule of partial derivatives to

$$\frac{\partial E^{(n)}}{\partial w_{ji}} = \frac{\partial E^{(n)}}{\partial a_j^{(n)}} \frac{\partial a_j^{(n)}}{\partial w_{ji}}, \tag{9}$$

where we introduce

$$\delta_j^{(n)} \equiv \frac{\partial E^{(n)}}{\partial a_j^{(n)}}. \tag{10}$$

Using Equation (2), we see that

$$\frac{\partial a_j^{(n)}}{\partial w_{ji}} = x_i^{(n)}. \tag{11}$$

Substituting Equation (10) and (11) into Equation (9) yields

$$\frac{\partial E^{(n)}}{\partial w_{ji}} = \delta_j^{(n)} x_i^{(n)}. \tag{12}$$

Thus, in order to compute the derivative we only need to calculate $\delta_j^{(n)}$ and multiply it with $x_i^{(n)}$. For the output layer, calculating $\delta_j^{(n)}$ is straightforward:

$$\delta_k^{(n)} = \frac{\partial E^{(n)}}{\partial a_k^{(n)}} = \frac{\partial E^{(n)}}{\partial y_k^{(n)}} \frac{\partial y_k^{(n)}}{\partial a_k^{(n)}} = g'(a_k^{(n)}) \frac{\partial E^{(n)}}{\partial y_k^{(n)}}, \tag{13}$$

where $g'()$ is the derivative of the activation function with respect to $a_k^{(n)}$. When Equation (8) is taken as the loss function, this would result in

$$\delta_k^{(n)} = g'(a_k^{(n)})(y_k^{(n)} - t_k^{(n)}). \tag{14}$$

To evaluate δ's in hidden layers, we use

$$\delta_j^{(n)} = \frac{\partial E^{(n)}}{\partial a_j^{(n)}} = \sum_k \frac{\partial E^{(n)}}{\partial a_k^{(n)}} \frac{\partial a_k^{(n)}}{\partial a_j^{(n)}}, \tag{15}$$

where the sum covers all neurons k in the output layer to which neuron j in the hidden layer sends values to in forward propagation. This leads to

$$\delta_j^{(n)} = g'(a_j^{(n)}) \sum_k w_{kj} \delta_k^{(n)}. \tag{16}$$

2.3.3 Iterative procedure

The iterative training procedure can now be summarized in 6 steps which should be applied to each weight:

1. Forward propagate through the network to find the output values based on the current weights.
2. Use Equation (14) to find the $\delta' s$ of all output neurons.
3. Back propagate the $\delta' s$ using Equation (16) to find δ_j of each hidden neuron (not relevant for weights between the last hidden layer and the output layer).
4. Use Equation (12) to find the required derivatives.
5. Find the derivative of the total error E of the batch by summing the errors of all individual training samples in the batch using Equation (7)
6. Use Equation (5) to update the weights of the neural network.

This updating of weights is done per batch until all training samples are covered, which is referred to as an epoch. After that, the next epoch is started and the next iteration of the procedure is carried out, until a satisfying error (defined by the user) is achieved.

3 Hyperparameter options

In this chapter, some hyperparameters and corresponding options will be further elaborated on. Common pitfalls and rules of thumb will be addressed.

3.1 Optimizer

Optimizers play a role in the back propagation step of a neural network. They determine how the weights and biases are updated. In Subsection 2.3.2, back propagation is explained using *Mini-Batch Gradient Descent* as the optimizer. There are many other optimizers, some very similar, some very different [9]. Commonly used optimizers are explained below.

- **Variations on Gradient Descent**

  Gradient Descent, see Equation (5), has three commonly used versions:

  - Regular Gradient Descent (RGD) - in which the gradient is calculated based on the sum of the error of every training sample in an epoch, i.e. the whole training dataset.
  - Stochastic Gradient Descent (SGD) - in which the gradient is calculated based on the error of a single training sample.
  - Mini-Batch Gradient Descent (MBGD) - in which the gradient is calculated based on the sum of the error of every training sample in a batch.

  If you want to use a version of Gradient Descent, it is best to use MBGD. RGD might take too long, as it has to store all losses before it can calculate the new weights. SGD on the other hand has the problem of fluctuating a lot which might cause it to shoot over minima. Mini-Batch Gradient Descent is a perfect balance of these methods [10].

- **Adam**

  Adam, adaptive moment estimate, estimates the first and second moments of the gradient of the loss, instead of only taking the gradient of the loss as done in Equation (5), for each weight of the neural network. The first moment m is the mean and the second moment v is the uncentered variance. To estimate the moments, Adam uses exponentially moving averages, computed on the gradient evaluated on the current batch:

  $$m^{(n)} = \beta_1 m^{(n-1)} + (1-\beta_1) g^{(n)}, \qquad (17) \qquad\qquad v^{(n)} = \beta_2 v^{(n-1)} + (1-\beta_2) g^{(n)^2}, \qquad (18)$$

  where n indicates the current training sample in the batch, $g^{(n)}$ indicates the gradient of the loss of the current training sample and where β_1 and β_2 are new hyper-parameters with default values equal to 0.9 and 0.999 respectively. Almost no one ever changes these values as they often are the best choice [11]. These moments still have a scaling bias, which is corrected for using

  $$\hat{m}^{(n)} = \frac{m^{(n)}}{1-\beta_1}, \qquad (19) \qquad\qquad \hat{v}^{(n)} = \frac{v^{(n)}}{1-\beta_2}. \qquad (20)$$

  .

  Next, the weights are updated after every training sample using

  $$w^{(n+1)} = w^{(n)} - \eta \frac{\hat{m}^{(n)}}{\sqrt{\hat{v}^{(n)}} + \epsilon}, \qquad (21)$$

  where ϵ is a small constant (usually equal to 10^{-7}) to avoid division by zero for the first training sample in a batch. Before a new batch is started, $m^{(n)}$and $v^{(n)}$ are set to 0. Advantages of Adam are that it converges fast and that it is able to find local minima that are very close to global minimum [11]. In addition, the step size of an update in Adam is invariant to the magnitude of the gradient, which helps a lot in going through areas with small gradients, as it is relatively fast in these areas compared to MBGD, and in going through areas with large gradients, as it is relatively slow in these areas compared to MBGD.

3.2 Loss function

Like optimizers, loss functions play a role in back propagation. They define the training error that the optimizer is trying to minimize. There are many loss functions available, of which a few will be discussed in more detail below.

- **Mean squared error**; the mean of the sum of the squares between the true and predicted values for all output neurons:

$$E = \frac{1}{2c} \sum_{k=1}^{c} (y_k - t_k)^2. \tag{22}$$

  This is the loss function that is used in Section 2.3.

- **Mean squared logarithmic error**; the mean of the sum of the squares between the logarithm of the true and predicted values for all output neurons:

$$E = \frac{1}{c} \sum_{k=1}^{c} (log(1 + y_k) - log(1 + t_k))^2. \tag{23}$$

  This loss function cares about the error of the logarithmic values, which is very close to a relative error [12]. In contrast, the mean squared error loss function only cares about the absolute error. This means that the mean squared logarithmic error is a good loss function when the training sample values lay in a large range. This is not the case for training data that is normalized. In addition, note that mean squared logarithmic error penalizes underestimates more than overestimates [12].

- **Huber**; the Huber error calculates the mean squared error when the error is below a threshold δ_E and the mean absolute error when the error is above δ_E. The mean absolute error is multiplied by δ_E and afterwards subtracted by $\frac{1}{2}\delta_E^2$ to make the loss function continuous.

$$E = \frac{1}{c} \sum_{k=1}^{c} \begin{cases} \frac{1}{2}(y_k - t_k)^2 & \text{when } |y_k - t_k| \leq \delta_E \\ \delta_E |y_k - t_k| - \frac{1}{2}\delta_E^2 & \text{otherwise} \end{cases} \tag{24}$$

  This can be regarded as a variation on the mean squared error loss, but it is less sensitive to outliers, because the error of the outliers is not squared.

The mean squared error is a good loss function for most problems, and often is preferred. If your data has a large range of values, the mean squared logarithmic error might be a better option, although it penalizes underestimates more than overestimates. When there are a lot of outliers, the Huber error might be a good choice.

3.3 Learning rate

The learning rate plays an important role in the training of a neural network. It is a hyperparameter used in back propagation that determines by how much weights are changed at each weight update. Choosing a too low learning rate will make training the network very slow, as it will take a long time before the network reaches a minimum of its loss function. Choosing a too high learning rate will make the network jump around at the minimum, which may prevent the network from ever reaching a minimum, i.e., it may lead to divergence. Finding a good balance is thus crucial for achieving good results in any neural network. This is illustrated in Figure 3 [13].

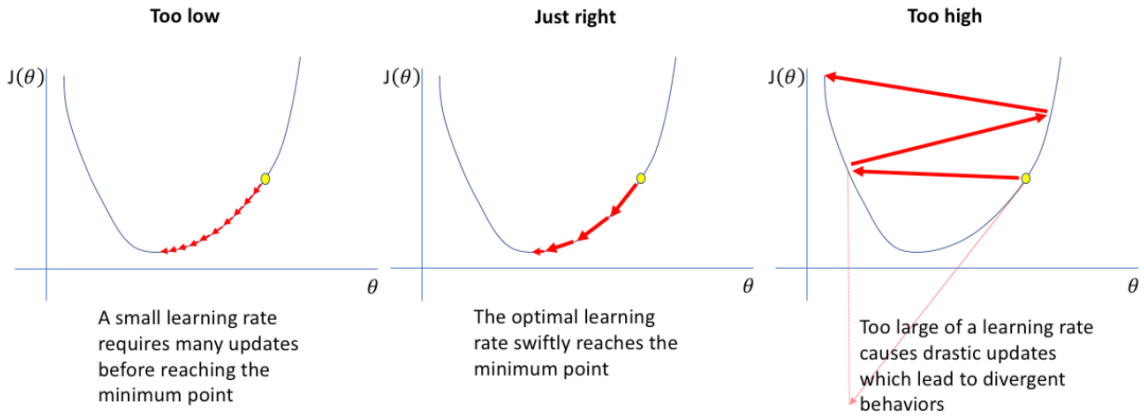


Figure 3: Effects of different learning rates. θ are the weights and $J(\theta)$ is the loss function [14].

Learning rate decay

Ideally, a network is trained with a large learning rate in the beginning and a small learning rate in the end. This will make sure that initially, the network quickly converges to weights near a good local minimum or preferably the global minimum. At this point, a smaller learning rate is preferred, to make small changes to the weights to achieve the best result, i.e., to end up in that local/global minimum. This can be achieved using learning rate decay. Learning rate decay lowers the learning rate as a function of the epoch as can be seen in Figure 4. In this case, a step function decrease is chosen as learning rate decay, however, this could also be an exponential function or any other function specified by the user.

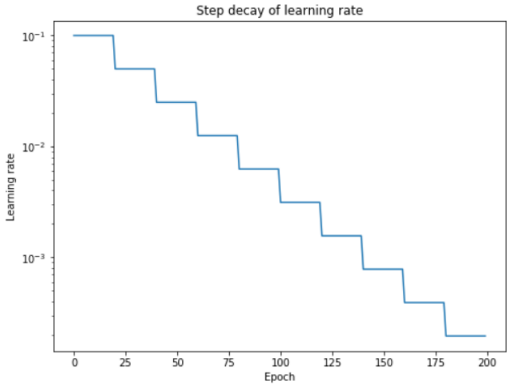


Figure 4: Learning rate per epoch using stepwise learning rate decay.

3.4 Activation function

There are two types of activation functions in a neural network. Most activation functions in hidden layers add nonlinearity to the network, allowing the network to estimate nonlinear relations. The activation function in the output layer also makes sure the output of the network has the desired format (this will be explained below). Three examples of activation functions are given by Figures 5, 6, and 7.

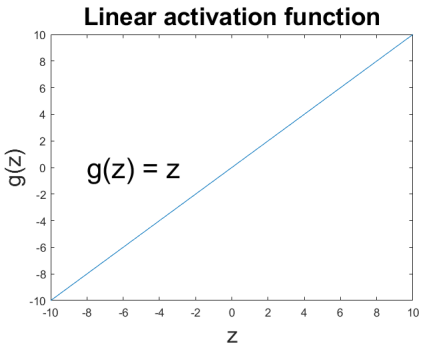


Figure 5: Linear activation function.

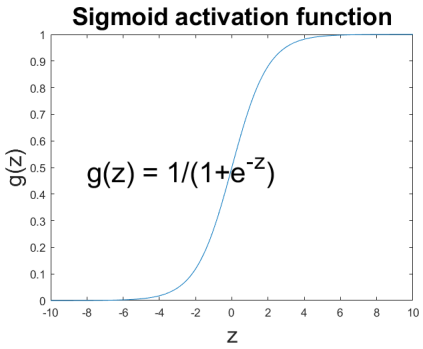


Figure 6: Sigmoid activation function.

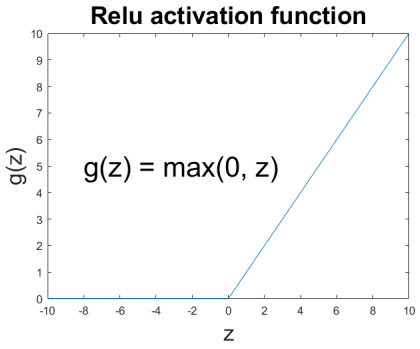


Figure 7: Relu activation function.

Hidden layer

As mentioned above, the purpose of the activation functions in the hidden layer is to add nonlinearity. This allows the neural network to estimate nonlinear nonlinear. In back propagation, the derivative of the activation function is used. It is therefore advantageous if the derivative of the activation function is easy to compute, as this is computationally effective. There are a lot of activation functions to choose from. Commonly used activation functions in hidden layers are Sigmoid and Relu [15]. Every layer and even every neuron can have a different activation function. In this report, only a single activation function per layer is used to avoid overcomplexity.

Output layer

The activation function of the output layer makes sure the output of the network has the desired format. For example, for classification problems, usually an activation function is chosen with values ranging from 0 to 1, such as the Sigmoid function, which makes it easy to compute the percentage likelihood of a prediction. For regression problems, usually a linear activation function is chosen, because values are unbounded, which is beneficial for extrapolating outside the bounds of the (normalized) training data.

3.5 Neural network structure

The structure of a neural network, meaning the number of hidden layers and number of neurons in each individual layer, influences how well the neural network is able to learn some (inverse) mapping. There are no strict rules on how to choose these parameters. However, some rules of thumbs for the number of neurons in the first hidden layer are provided by [4]. These rules may contradict each other, which illustrates how difficult it is to say something about good values:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

These rules only help to obtain a starting point for a network. A good structure is achieved by trial and error. One should start with a simple network and make it incrementally more complex if required to yield the desired (inferring) accuracy [16].

It is important not to make the network unnecessary large, as this can cause the network to overfit. Overfitting happens when the network is large enough to learn all training data (including noise) by heart, instead of learning features that occur in the data. Overfitting can be spotted when the performance of the network on the training data is good while the performance of the network on the validation data is relatively bad. This can be seen in Figure 8 on the right side.

Underfitting, on the other hand, happens when the features of the data are more complex than what the neural network is capable of learning due to a lack of trainable parameters, i.e., weights and biases. When underfitting, the performance of the neural network is bad on the training set as well as the validation set. This can be seen in Figure 8 on the left side.

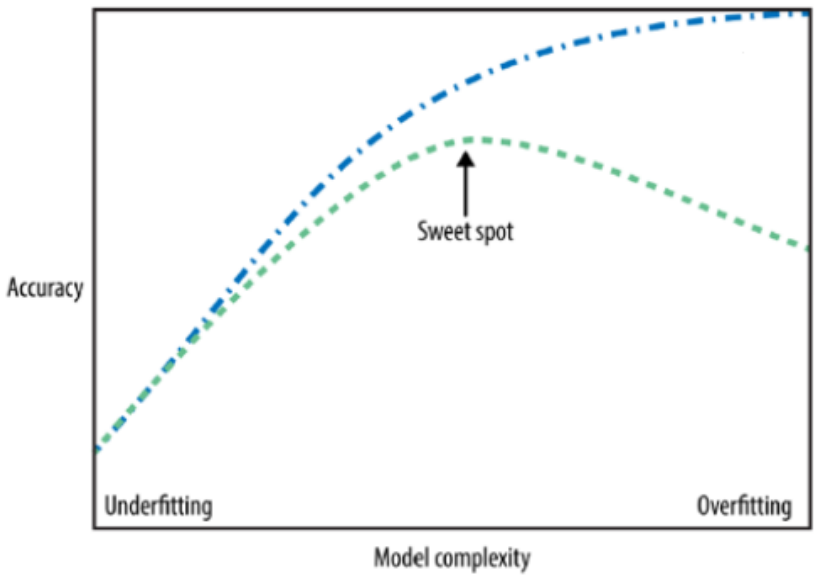


Figure 8: The blue line indicates the training accuracy, the green line indicates the validation accuracy [17].

3.6 Epochs & batch size

The epochs and the batch size manage how the data is used for training the network. The available data is split up in a training, a validation and a test set. The training data is used to tune the weights and biases of the neural network. This training data is split up in smaller batches, where, for most optimizers, the weights of the network are updated once an entire batch has passed through the network. Adam and Mini-batch gradient descent are optimizers that update the weights and biases after a batch has passed through the network. Stochastic or regular gradient descent update the weights and biases after a single training sample or all training samples have passed through the network respectively. For stochastic and regular gradient descent, the batch size thus does not matter.

Once all batches, and thus all training samples in the training set, have passed through the network (this event is called an epoch), the performance of the network is evaluated using the validation set. The samples in this set are passed through the network and the loss resulting from these samples is calculated. This loss is not used to update the weights of the network, however, it is an important step as this loss indicates how well the network performs on new data, and thus, how well the network generalizes. After the validation loss has been calculated, the network starts a new epoch. In this epoch, the network goes through the same training samples again to update the weights and biases, after which the network's performance is validated again on the validation set. This is repeated for any number of predefined epochs.

When tuning hyperparameters, the hyperparameter values/options are changed based on how well the network performs on the validation set. This may lead to the network becoming more biased towards the validation data. In order to give fair judgement about the network's performance, the final network is tested on a test set. This dataset has not been seen by the network before, and the loss resulting from this set thus gives a good, unbiased and general indication of how well this network performs.

Early stopping

It may happen that, once the network has gone through a number of epochs, the training loss continues to decrease, whereas the validation loss starts to increase. This is called overfitting, and is unwanted, as the network will not have a good general performance. This can be avoided using early stopping. Early stopping checks the validation loss of the network, and, if it has been detected that the networks validation loss does not decrease anymore for a set number of epochs, stops the training of the network. This is a useful tool, as it speeds up the training of the networks, as no unnecessary training is done, and in addition, it increases the performance of the network, because a network that is stopped using early stopping has a better general performance compared to a network that is not stopped and is able to overfit on the training set. Overfitting, as explained in Section 3.5, can thus be avoided using early stopping.

4 Automatic hyperparameter tuning methods

Tuning the hyperparameters of any neural network is a difficult task. There are no strict rules on how to set the hyperparameters, and some hyperparameter values/options might behave poorly with certain other hyperparameter values/options, whereas they might behave well in combination with other values/options. Manual hyperparameter tuning is thus usually done via a trial and error approach. The outcome of this approach is greatly affected by the skill and experience of the engineer, as well as some luck. Still, automation of hyperparameter tuning is of interest. In this chapter, three automatic hyperparameter tuning methods are explained: grid seach (Section 4.1), random search (Section 4.2), and Bayesian search (Section 4.3). In Section 4.4, the choice for which software is used to tune the hyperparameters is elaborated on.

4.1 Grid search

Grid search, as the name suggests, searches over a grid of hyperparameter values. The testable domain of every hyperparameter is split into a discrete grid by the user, and all possible combinations of hyperparameters are tested.

If the goal is to find the best combination of hyperparameter values/options, i.e., leading to the lowest validation loss, in a specific discrete domain, grid search will guarantee that you find this optimum. However, the drawback of grid search is that it is computationally very expensive. If there are 5 hyperparameters to tune, and each hyperparameter has 5 values in the domain to be tested, $5^5 = 3125$ neural networks have to be trained. If an additional hyperparameter is added with 3 options, the amount of networks that have to be trained is increased to $3125 \cdot 3 = 9375$. This is also referred to as the curse of dimensionality.

4.2 Random search

Random search trains neural networks where the hyperparameter values/options are chosen randomly. They can be either determined from a discrete or a continuous domain (if the hyperparameter permits this). Choosing the hyperparameter values/options is based on probability. If a hyperparameter is set to a discrete domain, the chance of choosing one of the hyperparameter options is equal for every option in the domain. In case of a continuous domain, the user can choose a distribution from which the hyperparameter value/option is sampled, e.g., a uniform distribution between a minimum and a maximum value or a normal distribution. This method might, at first glance, seem not to be very effective, but as will be discussed in the two points below, can actually be very effective.

Firstly, when the possible hyperparameter values/options are set to the same discrete values in random search as in grid search, training 60 networks is usually enough to achieve a neural network that lays in the top 5% performing networks of the grid search, with 95% probability [18]. This can be explained as follows. If a combination of hyperparameter values/options is randomly sampled from the grid search space, that combination of hyperparameter values/options has a 5% chance of landing in the top 5% best performing combinations of that grid search. The chance of not landing in this top 5% is thus $(1 - 0.05)$. Then, if n points are randomly sampled, the chance of not hitting the top 5% combinations is $(1 - 0.05)^n$, and thus the change of hitting the top 5% best performing combinations is $1 - (1 - 0.05)^n$. If we want at least a 0.95 probability of success, i.e.,

$$1 - (1 - 0.05)^n > 0.95, \tag{25}$$

it should hold that $n \geq 60$. This is useful if the neural networks, parameterized with the top 5% best performing combinations of hyperparameter values/options of the grid search, still perform well enough. This can be true if the close to optimal region is large or if there are a lot of grid points in that region. The former is more likely since a neural network should not be overly sensitive to the hyperparameters, i.e., the close to optimal range is relatively large [18].

Secondly, when the possible hyperparameters values are set to a continuous range in random search as opposed to a set of discrete values in grid search, a larger set of hyperparameter values/options can be explored. This is possible for the same cost, as not all possible values have to be combined with each other. This is illustrated in Figure 9 for the case where two hyperparameters are tuned. Two hyperparameters that do not influence each other are tested together, where one hyperparameter affects the performance of the neural network by a lot, whereas the other barely has any influence on the networks performance. As can be seen, a lot more values for the important hyperparameters are tested in random search, and more importantly, values much closer to the maximum (indicating highest accuracy) have been found on the green peak, which was not the case in grid search. This makes it possible for random search to outperform grid search.

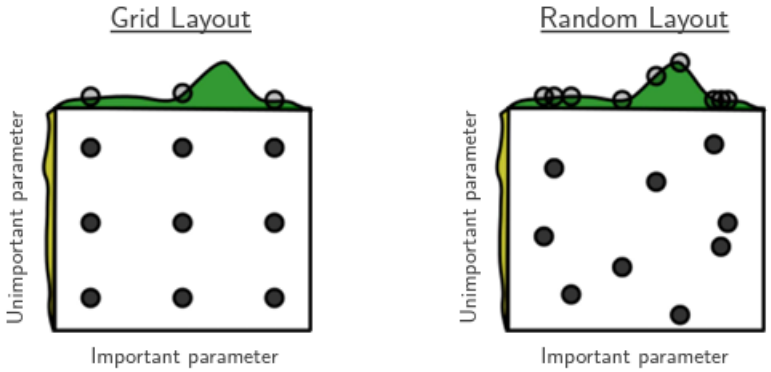


Figure 9: Analysed combinations of two hyperparameters in grid search and random search [19].

4.3 Bayesian search

Bayesian search uses Bayesian optimization to try to find the optimal solution in an efficient way. Bayesian optimization is an optimization method focused on finding the inputs x that lead to the global optimum of an unknown, continuous black box function $f(x)$. This method fits the tuning of hyperparameters of a neural network perfectly, where the performance of the network, i.e., the validation loss, can be considered as the (black box) function $f(x)$, where x consists of the hyperparameters of the neural network. The Bayesian optimization process works as follows:

1. Start off with a randomly chosen initial set of hyperparameter values/options x chosen from a pre-set range of hyperparameter values/options.
2. Evaluate the validation loss for the chosen set of hyperparameter values/options by training a network on those values/options x.
3. Estimate a model of the function $f(x)$ using a Guassian process [20] based on the already acquired data. This model consists of a mean $\mu(x)$ and a variance $\sigma(x)$ (note that these are functions of x) as illustrated in Figure 10. In this Figure, only one point (or vertical dotted line) is present the first time when going through these steps.

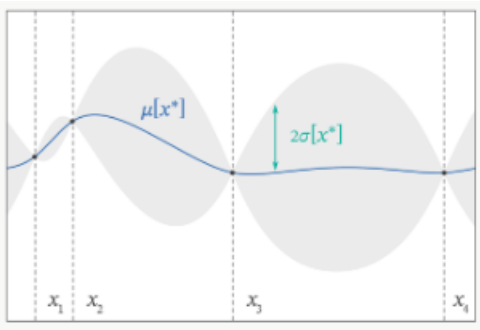


Figure 10: Mean (blue solid line) and variance (gray areas) of the prediction of function $f(x)$. The y-axis is the validation loss [21].

4. Calculate the acquisition function. This is a function that takes the mean and variance at each point and decides how desirable it is to explore that point. A function that is commonly used as the acquisition function is the Lower Confident Bound (LCB):

$$LCB(x) = \mu(x) - \kappa \cdot \sigma(x). \tag{26}$$

Here, κ is a hyperparameter (note that this hyperparameter is related to the optimization algorithm, not to the neural network itself) that determines the trade-off between exploration (exploring areas with few data points) and exploitation (exploiting already good data points to find even better data points). A high κ will lead to exploration as the uncertainty outweighs the mean, which is larger in unknown areas, and a low κ will lead to exploiting as the mean outweighs the uncertainty, which is larger in already successful areas. This can also be seen in Figure 11.

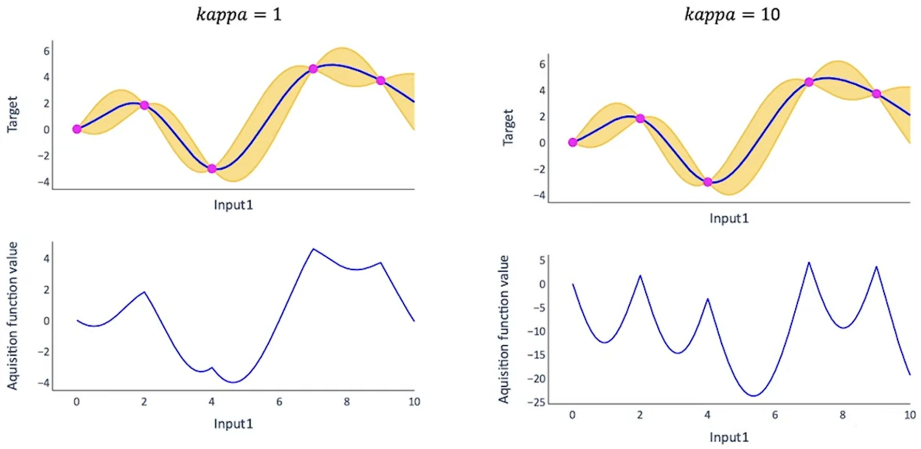


Figure 11: Lower Confidence Bound illustrating the effect of high and low *kappa*. Target, on the y-axis, is the validation loss [21].

5. Identify which set of hyperparameter values/options results in the minimum of the LCB.

6. Use the acquired set of hyperparameters for step 2 and repeat step 2 till 5 for any number of iterations until a satisfying performance is achieved.

4.4 Tuning software

Many tools exist that can tune the hyperparameters of a neural network. For this research, a tool is needed that meets the following requirements: supports the three hyperparameter tuning methods as explained above, has easy network management, i.e., allows for easy saving of and comparison between the performance of neural networks, and is free to use. Four popular neural network management tools have been looked into to check if they meet the requirements, these are: Weights and Biases (WandB) [22], Neptune.ai [23], Tensorboard [24], and Comet [25]. It was quickly found that WandB and Comet are the only 2 platforms, out of the 4 that were looked into, that support hyperparameter tuning. The documentation of WandB on hyperparameter tuning seemed way more elaborate and easy to find, and therefore, WandB was chosen as the tuning software.

WandB is an online tool that allows the user to easily specify any type of search (out of grid search, random search, and Bayesian search). The search can be started, after which no more interference with the software is required anymore. The trained networks are automatically saved, and the performance and other import parameters, e.g., the training time and the training settings, are clearly visualised for each network.

5 Reference network for a use case

Before discussing the results of the hyperparameter tuning methods as described in Chapter 4, the settings and result of a reference network will be given in the current chapter, to be able to compare this with the results of the automatic tuning methods in Chapter 6. However, the use case will be explained first.

An inverse mapping model is trained for a two-degrees-of-freedom nonlinear multibody system consisting of two connected rigid beams, as shown in Figure 12. There are 4 updating parameters: spring constants k_y and k_θ and damping constants d_y and d_θ, where it is assumed that their values may vary between the bounds specified in Table 2. The outputs of the system are the horizontal position y of beam 1 and the rotation θ of beam 2, and are simulated using the ode45 function (RelTol = 10^{-3}, AbsTol = 10^{-6}) in Matlab 2021b. These simulated output signals are contaminated with artificial additive zero-mean Gaussian output noise with standard deviations $\sigma_y = 5 \cdot 10^{-5}$ m and $\sigma_\theta = 0.015$ rad, respectively. For the output features, 255 equidistant time samples ($\Delta t = 0.0196$ s) are used for each output, amounting to a total set of 510 features per sample. To exemplify, the normalized feature values of y and θ for a single sample are shown in Figure 13. The initial conditions correspond to the static equilibrium position of a system parameterized with parameter values in the center of the admissible parameter space as listed in Table 2. The system is excited by force $F(t)$ and torque $T(t)$, respecitvely:

$$T(t) = \begin{cases} 5 \text{ N} & \text{if } 0.2 \leq t < 0.25, \\ 0 \text{ N} & \text{else,} \end{cases} \tag{27}$$

$$\begin{cases} 0.075 \text{ Nm} & \text{if } 0.2 \leq t < 0.25, \\ 0 \text{ Nm} & \text{else.} \end{cases} \tag{28}$$

.

This ensure that the output features are more sensitive to changes in the parameter values.

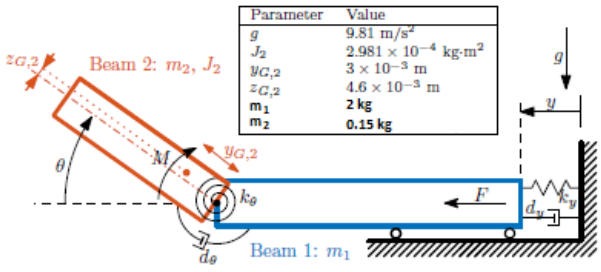


Figure 12: Use case model with the constant parameters and values [2].

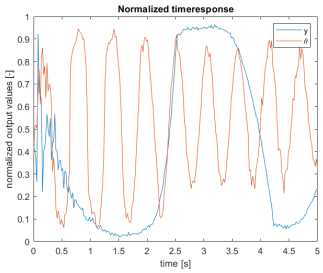


Figure 13: Normalized output features for single sample.

Table 2: Updating parameters and their corresponding lower and upper bound.

Parameter	Lower bound	Upper bound
k_y	5 N/m	15 N/m
k_θ	0.027 Nm/rad	0.045 Nm/rad
d_y	0.9 Ns/m	1.1 Ns/m
d_θ	0.000175 Nsm/rad	0.000225 Nsm/rad

A network has been trained on 1000 training samples, generated for as many distinct combinations of uniformly distributed updating parameter values. All parameter values and output features are normalized between 0 and 1. The utilized neural network is made by an engineer, inexperienced in the field of machine learning, whose choices are based on some basic research and trial and error. The reference network has been obtained after approximately 5 iterations, and is a fully connected

feedforward neural network with 2 hidden layers, where the consecutive layers have 200 and 100 neurons, respectively, both using Relu activation functions. Adam is used as the optimizer with a learning rate of 0.001. The network is trained in batches of 50 training samples over 200 epochs using the Mean Squared Error as the loss function. The network has been iteratively improved based on the validation loss consisting out of 100 validation samples.

The test loss of this network, based on $o = 2000$ test samples, which is analyzed only after the final network structure is found, is equal to 0.00261. This value is used to judge the performance of the network and is the value that in the end should be improved, i.e., decreased, using the techniques as explained in Chapter 4.

Error metrics, that express how well the neural network estimates the true model parameter values, are explained below. These are all based on the relative estimation error for test sample i:

$$\epsilon_{rel,k}(i) = \frac{y_k(i) - t_k(i)}{t_k(i)}, \tag{29}$$

where k indicates an output neuron. The bias, $\mu_{\epsilon,k}$, for output neuron k is calculated using

$$\mu_{\epsilon,k} = \frac{1}{o}\sum_{i=1}^{o} \epsilon_{rel,k}(i). \tag{30}$$

The standard deviation, $\mu_{\epsilon,k}$, for output neuron k is calculated using

$$\sigma_{\epsilon,k} = \sqrt{\frac{1}{o-1}\sum_{i=1}^{o}(\epsilon_{rel,k}(i) - \mu_{\epsilon,k})^2} \tag{31}$$

The mean absolute relative error, $\mu_{|\epsilon,k|}$ is calculated using

$$\mu_{|\epsilon|,k} = \frac{1}{o}\sum_{i=1}^{o} |\epsilon_{rel,k}(i)|. \tag{32}$$

Using the above defined error metrics, the performance of the reference network with respect to these metrics is calculated and listed in Table 3. Note that in this table the values of Equation (30), (31) and (32) have been multiplied with 100%.

Table 3: Error metrics based on test data results from reference network.

	μ_ϵ [%]	σ_ϵ [%]	$\mu_{\lvert\epsilon\rvert}$ [%]
k_y	-0.26	1.74	1.45
k_θ	0.15	0.58	0.48
d_y	-0.96	1.07	1.19
d_θ	-0.13	1.70	1.31

6 Network hyperparameter tuning results

The results and conclusions drawn from the application of each of the three hyperparameter tuning methods from Chapter 4 to the case study of Chapter 5 will be discussed in Section 6.1 (grid search), Section 6.2 (random search) and Section 6.3 (Bayesian search), respectively. The performance of each neural network is judged based on the validation loss, which is the loss calculated on the validation output data, i.e., the normalized inferred model parameter values. The loss function is taken identical for all methods, because this allows for a fair comparison in performance between models, and is set to the mean squared error, see Equation 22, as this is a good choice for most networks as explained in Section 3.2. In addition, early stopping has been applied to all search methods to avoid overfitting. In Section 6.4, general conclusions are given and the best performing network taken from the best performing hyperparameter tuning method is analysed on the test set and compared to the performance of the reference network.

6.1 Grid search

A grid search has been performed on the set of hyperparameter values/options as listed in Table 4.

Table 4: Hyperparameter options for grid search. The architecture column contains the number of neurons in every hidden layer, where "/" indicates the addition of a hidden layer.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer
510	10	100	0.1	Sigmoid	Adam
510/250	40	200	0.05	Relu	MBGD
510/350/200	100	400	0.01	Linear	
510/375/250/125	200		0.001		
375/250/125			0.0001		
1000					
10000					

From Table 4, 2520 combinations are possible. The total training time of all these networks is approximately 64 hours. This concludes to an average training time of 1.5 minutes per network. However, note that, the training time per network configuration is actually variable and depends mainly on the number of epochs. The top 10 performing networks based on the validation loss are listed in Table 5.

Table 5: Top 10 performing networks from grid search.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer	Validation loss
510/375/250/125	200	400	0.001	Sigmoid	Adam	0.000817
510/250	100	400	0.001	Sigmoid	Adam	0.000827
510/375/250/125	100	400	0.001	Sigmoid	Adam	0.000894
375/250/125	200	400	0.001	Sigmoid	Adam	0.000901
510/250	40	400	0.001	Sigmoid	Adam	0.000970
510/350/200	100	400	0.001	Sigmoid	Adam	0.000991
510/250	10	400	0.001	Sigmoid	Adam	0.001002
510/250	200	400	0.001	Sigmoid	Adam	0.001111
510/350/200	40	400	0.001	Sigmoid	Adam	0.001156
510/350/200	200	200	0.001	Sigmoid	Adam	0.001162

Although not fully shown here for brevity, the best 114 networks trained using this grid search perform better than the reference network (which has a test loss of 0.00261) as explained in Chapter 5. Furthermore, the best performing network from the grid search performs 3.2 times better than the reference network based on the validation loss.

Figure 14 illustrates how the networks perform compared to the reference network. It can be seen that for the best 350 networks, the validation loss increases relatively fast compared to the worse performing networks. The best network performs around 10 times better than the network in position 350. After this, the line becomes less steep and the performance does not decrease that quickly anymore compared to the first 350 networks. This shows that at this level of accuracy, there are a lot of possible combinations of hyperparameter values/options that can achieve this accuracy. Unfortunately, this is the case for networks performing more than 10 times worse than the reference network, which makes this a useless feature to exploit. Only a small, specific set of hyperparameter values/options performs better than the reference network.

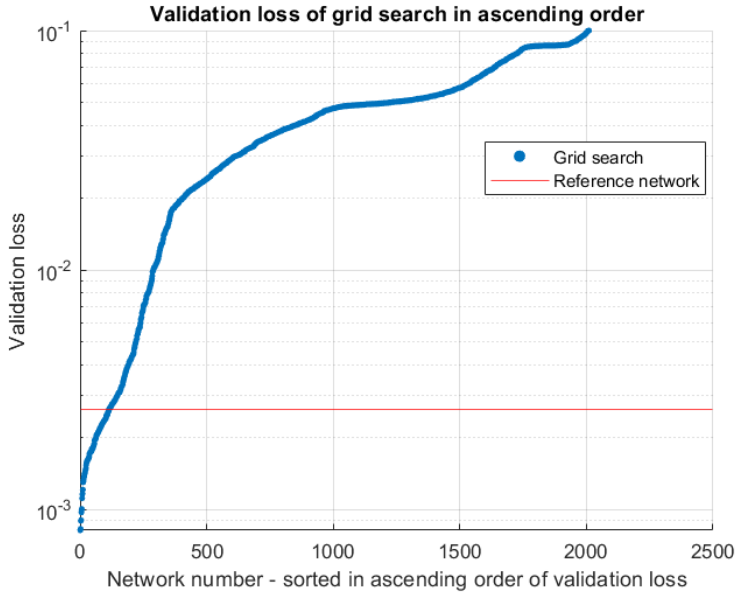


Figure 14: Validation loss over neural network sorted from most accurate (left) to least accurate (right) for grid search.

A few general observations can be made about the combinations and values of hyperparameters that perform best based on the grid search:

- Adam is the best performing optimizer on this dataset. The first network that uses MBGD is the 202nd best performing network based on the validation loss.
- Looking at the top 10 networks, Sigmoid seems to be the best activation function. However, out of the top 100 neural networks trained with this grid search, 58 networks were using Relu.
- Successful networks using Relu as activation function look very different from successful networks using Sigmoid. Successful networks using Sigmoid tend to use a learning rate of 0.001 and an architecture containing multiple layers. Whereas successful networks using Relu tend to use a learning rate of 0.0001 and an architecture consisting out of 1 layer with 1000 or 10000 neurons. This shows that all hyperparameters are very dependent on eachother, illustrating how difficult it is to find a global optimum.
- The batch size does not really seem to matter. All types of batch sizes randomly appear in the top 10.
- A large number of epochs seems to be preferable. A larger number of epochs of course comes with the risk of overfitting, so setting the number of epochs to a large number does not always guarantee a successful network. The problem of overfitting can be avoided by using early stopping, as has been done in this grid search.

6.2 Random search

Multiple random searches over different values have been performed. The results of each run will be discussed below.

6.2.1 Random search 1

A random search has been performed on the hyperparameter values/options as set in Table 6. The values are similar to the possible values in the grid search as listed in Table 4 (except for the minimum batch size and number of epochs, those are both equal to 1 instead of 10 and 100 respectively), however, the continuous variables are set to a continuous range. The batch size and number of epochs are uniformly distributed, whereas the distribution for the learning rate is log uniform [26]. This means that it is equally likely for the random search to train a network with a learning rate of 0.01 as it is to train a network with a learning rate of 0.001.

Table 6: Hyperparameter options for random search 1. The architecture column contains the number of neurons in every hidden layer, where "/" indicates the addition of a hidden layer.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer
510	Uniform	Uniform	Log uniform values	Sigmoid	Adam
510/250	min: 1	min: 1	min: 0.0001	Relu	MBGD
510/350/200	max: 200	max: 400	max: 0.1	Linear	
510/375/250/125					
375/250/125					
1000					
10000					

Sixty networks have been trained using random search. The sum of the training times of these networks is approximately 1.6 hours. This concludes to an average training time of 1.6 minutes per network. This is approximately the same training time as in the grid search, which makes sense as networks having epochs in the same range as the grid search should take about as long as an average network in the grid search. The top 10 performing networks based on the validation loss are listed in Table 7.

Table 7: Top 10 performing networks from random search 1.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer	Validation loss
510/375/250/125	26	220	0.000299	Relu	Adam	0.001962
510	45	236	0.000421	Relu	Adam	0.001977
375/250/125	76	244	0.000277	Relu	Adam	0.002551
510/350/200	163	182	0.002125	Sigmoid	Adam	0.002760
10000	195	390	0.000641	Sigmoid	Adam	0.010454
510/375/250/125	184	28	0.001159	Relu	Adam	0.013537
10000	56	395	0.000314	Sigmoid	Adam	0.019516
510	170	336	0.000391	Linear	Adam	0.020706
375/250/125	79	264	0.000264	Linear	Adam	0.020846
10000	196	341	0.000156	Linear	Adam	0.023157

The best three networks from this random search perform better than the reference network, but only slightly. The worst network in the top 5% of the grid search has a validation loss of 0.00277. According to Equation (25), it is 95% likely that a network from a random search over the same grid as a grid search performs better than 0.00277 when 60 networks are trained. The conditions required to use this equation are not exactly matched, as the continues variables are set to a continuous range, so the same grid is not used. However, the essence of this statement still holds as values over the same range are explored and it is therefore not unexpected that a few networks perform better than the worst network out of the top 5% from the grid search.

Figure 15 shows how all networks perform compared to the reference network based on validation loss. It can be seen that 4 networks are comparable to the reference network. The remaining networks perform between 4 and 40 times worse than the reference network. It can be seen that no networks seem to perform between 1 and 4 times worse than the reference network. This gap is clearly visible in Figure 15. This could mean that the local minima are located between steep hills but have a relatively wide bottom, making it very difficult to find a network that lays on the sides of this minimum and thus has a validation loss between the good and the bad networks.

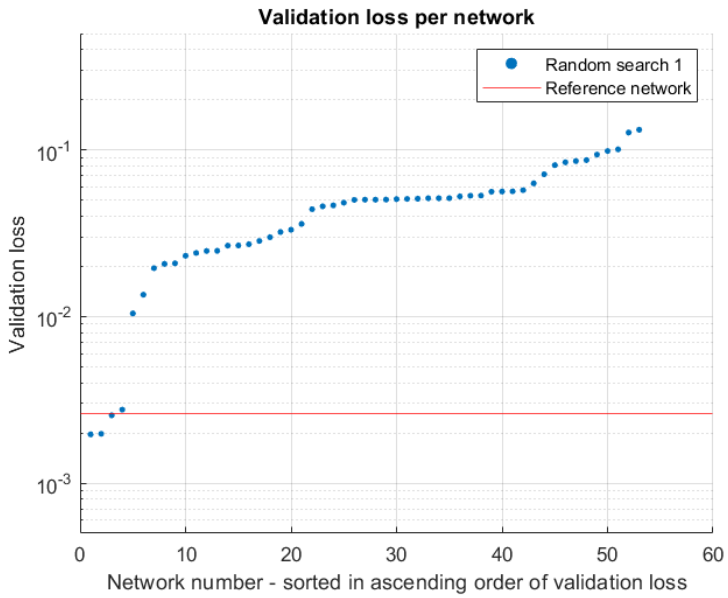


Figure 15: Validation loss over neural network sorted from most accurate (left) to least accurate (right) for random search 1

6.2.2 Random search 2

A second random search has been performed on the hyperparameter options as listed in Table 8. The options for architecture, activation function, and optimizer have remained similar to the grid search and first random search, but the other hyperparameter values have been extended. The learning rate values have been set to a larger, continuous range using a log uniform distribution. The batch size and epoch values have been set to a larger, discrete uniform distribution of integers. This has been done as this is a computationally inexpensive way to explore a large range of networks compared to grid search.

Table 8: Hyperparameter options for random search 2. The architecture column contains the number of neurons in every hidden layer, where "/" indicates the addition of a hidden layer.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer
510	Uniform	Uniform	Log uniform values	Sigmoid	Adam
510/250	min: 1	min: 1	min: 0.00001	Relu	MBGD
510/350/200	max: 1000	max: 1000	max: 0.1	Linear	
510/375/250/125					
375/250/125					
1000					
10000					

60 Networks have been trained in a random search using these hyperparameters. The total training time of all these networks is approximately 3.5 hours. This concludes to an average training time of 3.5 minutes per network. This is longer than the first random search, because the number of epochs,

the most impacting hyperparameter when it comes to training time, has a maximal possible value of 1000 in this random search compared to 400 in the previous random search. The top 10 performing networks based on the validation loss are listed in Table 9.

Table 9: Top 10 performing networks from random search 2.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer	Validation loss
1000	936	834	0.001933	Sigmoid	Adam	0.000839
510/250	484	902	0.001614	Sigmoid	Adam	0.001173
510	656	948	0.000087	Relu	Adam	0.001633
375/250/125	262	143	0.001011	Sigmoid	Adam	0.001744
375/250/125	610	447	0.000240	Sigmoid	Adam	0.002187
510/250	313	829	0.004309	Sigmoid	Adam	0.002246
1000	405	326	0.001137	Sigmoid	Adam	0.002720
510	725	607	0.000040	Relu	Adam	0.002826
510	901	221	0.006523	Sigmoid	Adam	0.002958
10000	115	140	0.001065	Relu	Adam	0.003622

The best network from this random search performs almost 2.5 times better than the best network from the first random search based on the validation loss. In addition, this search performs almost as good as the grid search while being almost 20 times faster. The reason why this search performs so well, is that the epochs were allowed to have higher values. The reason for this choice was namely that, since, in the grid search, the top performing networks had the number of epochs equal to the highest possible value (400), this upper bound apparently was not chosen well. Therefore, it could be that a larger number of epochs would have performed better, but, this was not explored in the grid search. It can be seen that in this second random search, the top performing networks typically have a number of epochs greater than 400. Note that if a second grid search, with the broader range for the number of epochs, was performed, even better results than those obtained from this random search may be obtained. This analysis has however not been performed due to the huge computational cost of such a grid search.

Figure 16 shows how all networks trained for the second random search perform compared to the reference network based on the validation loss. More networks perform comparable to the reference network compared to the first random search. The clear gap that was present in the first random search between the networks performing with relatively high and low validation losses is present here too.

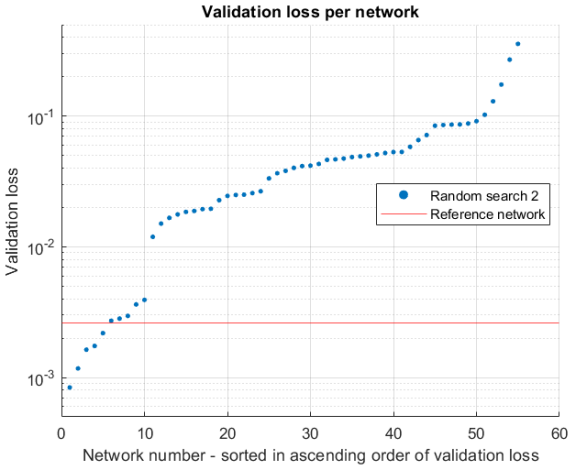


Figure 16: Validation loss over neural network sorted from most accurate (left) to least accurate (right) for random search 2.

6.2.3 Random search 3

A last random search has been performed where only the learning rate is sampled from a log uniform values distribution between 0.1 and 0.00001. The remaining hyperparameters are not varied and set equal to the values/options from the best performing network in the grid search as listed in Table 5. Sixty Networks have been trained taking 2.5 hours. The top 10 performing networks based on validation loss are listed in Table 10.

Table 10: Top 10 performing networks from random search 3.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer	Validation loss
510/375/250/125	200	400	0.001357	Sigmoid	Adam	0.000646
510/375/250/125	200	400	0.001278	Sigmoid	Adam	0.000783
510/375/250/125	200	400	0.000783	Sigmoid	Adam	0.000855
510/375/250/125	200	400	0.001127	Sigmoid	Adam	0.000871
510/375/250/125	200	400	0.001911	Sigmoid	Adam	0.000932
510/375/250/125	200	400	0.001674	Sigmoid	Adam	0.000937
510/375/250/125	200	400	0.002155	Sigmoid	Adam	0.000969
510/375/250/125	200	400	0.000976	Sigmoid	Adam	0.001016
510/375/250/125	200	400	0.001830	Sigmoid	Adam	0.001076
510/375/250/125	200	400	0.000896	Sigmoid	Adam	0.001134

As can be seen, this random search is able to improve the performance of the best network from the grid search. This is logical, as for the best performing network, more options are explored. This makes it more likely to find a better performing network, as more networks around a minimum are explored. This method could be applied to any hyperparameter in this network, likely also leading to improved performance, once enough networks are trained.

Figure 17 shows how all networks, trained in the third random search, perform compared to the reference network based on the validation loss. The gap between networks with high and low validation loss that is present in the other two random searches is still present here. This means that the learning rate is (at least one of) the hyperparameter(s) that is causing this gap. Furthermore, a lot of networks are able to outperform the reference network in this random search.

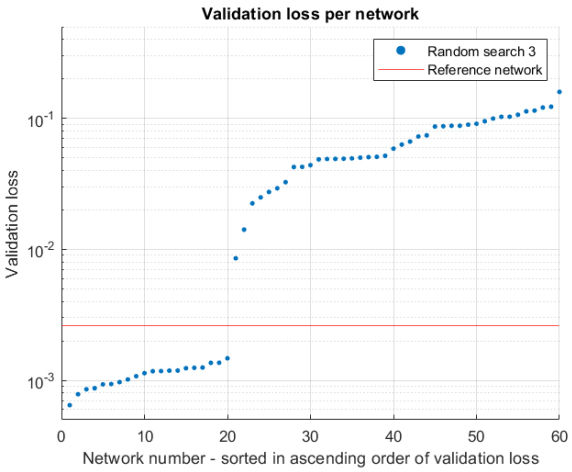


Figure 17: Validation loss over neural network sorted from most accurate (left) to least accurate (right) for random search 3.

In Figure 18, the validation losses which resulted from the training of the 60 networks are plotted as a function of the learning rate. Because the learning rate is sampled randomly, and thus not all

points are distributed evenly, a logarithmically evenly distributed grid search has been performed on the same range of learning rate values to confirm this graph. The results of this grid search can be seen in Figure 18 as well. This graph illustrates the difficulty of tuning hyperparameters and networks in general very well. One might think that at a learning rate of 10^{-4} a minimum validation loss is achieved as there is a (local) minimum of the validation loss near this learning rate (this can be observed when looking at the blue line but even more profound when the points of the blue and red line would be combined). However, near a learning rate of 10^{-3}, the loss is much smaller. This shows the problem of local and global minima, and that you are never sure at which type of minimum you are as the function is unknown. It should also be noted that Figure 18 clearly shows a wide valley with low losses in between two steep hills, which supports the statement made based on Figure 15.

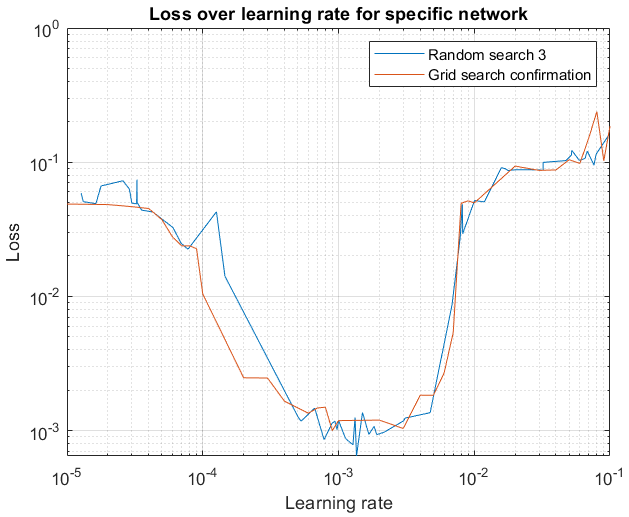


Figure 18: Validation loss over learning rate for specific network as obtained from the third random search, confirmed with a grid search.

6.3 Bayesian search

Three Bayesian searches have been performed. The results of each run are discussed below.

6.3.1 Bayesian search 1

A Bayesian search has been performed on the same possible ranges for the hyperparameter values/options as the second random search as listed in Table 8. Larger ranges have been explored compared to the grid search of Section 6.1 as this illustrates ranges chosen by an inexperienced engineer. The first network that is trained is chosen at random, given the distributions as specified in Table 8. In total, 167 networks are trained using this Bayesian search. This took 5.5 hours, or 2 minutes per network on average. The top 10 performing networks based on validation loss are listed in Table 11.

Table 11: Top 10 performing networks from Bayesian search 1.

Run	Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer	Validation loss
167	510/250	956	475	0.001887	Sigmoid	Adam	0.000887
95	510/250	950	410	0.000881	Sigmoid	Adam	0.001081
119	510/250	884	492	0.000651	Sigmoid	Adam	0.001123
165	510/250	853	591	0.002529	Relu	Adam	0.001308
107	510/375/250/125	641	269	0.001912	Sigmoid	Adam	0.001360
96	10000	920	274	0.000495	Relu	Adam	0.001400
131	510/375/200	857	479	0.000794	Relu	Adam	0.001448
161	510/375/200	632	252	0.004307	Sigmoid	Adam	0.001556
85	510/250	806	435	0.000234	Relu	Adam	0.001574
62	510/375/200	731	368	0.000435	Sigmoid	Adam	0.001577

The performance of the best network is comparable to the best network from the grid search. Here, it should be noted, however, that this value was achieved 12 times faster using the Bayesian search. It can be seen that networks that have been trained towards the end of the Bayesian search perform best as may be expected since information about mean and standard deviation becomes more accurate as the number of processed networks increases. Note, for example, that network 167, 165, and 161 are all in the top 10 performing networks. This can also be seen from Figure 19, where the validation loss tends to decrease as more networks are trained.

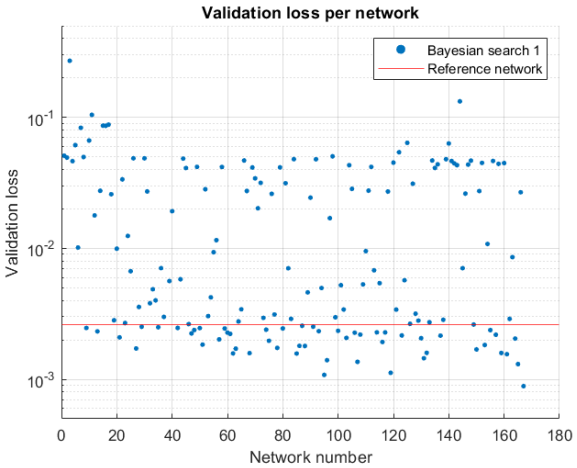


Figure 19: Validation loss over neural network as obtained from the first Bayesian search.

6.3.2 Bayesian search 2

A second Bayesian search has been performed on the same settings as the first run, as, e.g., the choice of prior in this search is made randomly, and thus it is interesting to see if a Bayesian search is (approximately) reproducible. Now, 117 Networks have been trained over 4.5 hours, or 2.4 minutes per network. The top 10 performing networks based on validation loss of this run can be seen in Table 12.

Table 12: Top 10 performing networks from Bayesian search 2.

Run	Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer	Validation loss
100	510/375/250/125	7	354	0.001178	Sigmoid	Adam	0.000959
2	510/350/200	148	399	0.003167	Sigmoid	Adam	0.000991
24	10000	4	619	0.000321	Relu	Adam	0.001031
107	10000	48	627	0.000129	Relu	Adam	0.001041
5	510/350/200	106	419	0.001390	Sigmoid	Adam	0.001275
27	10000	77	488	0.000126	Relu	Adam	0.001394
21	1000	102	689	0.000167	Relu	Adam	0.001587
116	1000	109	599	0.000537	Relu	Adam	0.001589
53	510/375/250/125	614	971	0.000730	Relu	Adam	0.001616
111	10000	73	284	0.000964	Relu	Adam	0.001786

As can be seen, this second Bayesian search is able to converge to comparably successful networks as in the first Bayesian search, but with less networks. This happens because this search is lucky by already hitting a well performing network in the second and fifth run. This makes it possible for the search to use information on what hyperparameter values/options apparently work well earlier on in the search.

Figure 20 shows the validation loss as a function of the neural network number for the second Bayesian search. As can be seen, a lot of networks perform better than the reference network. Note that the gap between the high and the low validation loss that has been present in the random searches can also be seen here.

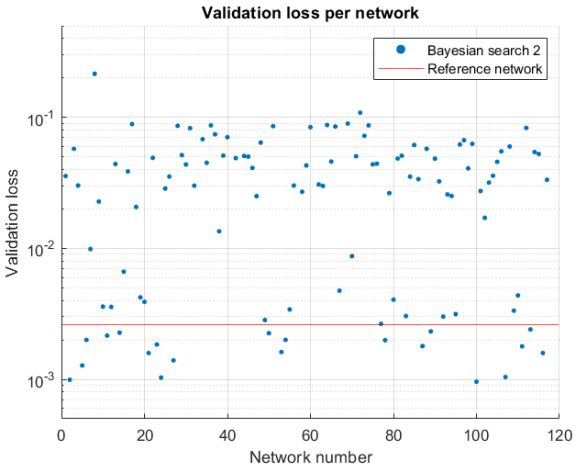


Figure 20: Validation loss over neural network as obtained from the second Bayesian search.

6.3.3 Bayesian search 3

The third and last Bayesian search is performed on a limited set of hyperparameter values/options based on the first random search. This has been done to test if limiting the ranges of hyperparameter values/options to those of the best performing networks found in an exploratory random search, results in better performing networks explored in a subsequent Bayesian search. The chosen hyperparameter values/options for the Bayesian search are listed in Table 13. These hyperparameters values/options correspond to the hyperparameter values/options present in the top 4 performing networks from Table 7. The hyperparameter values/options from the top 4 networks were chosen as in Table 7 there is a clear jump in the validation loss between the 4th and 5th best performing network.

Table 13: Hyperparameter options for Bayesian search 3. The architecture column contains the number of neurons in every hidden layer, where "/" indicates the addition of a hidden layer.

Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer
510	Uniform	Uniform	Log uniform values	Sigmoid	Adam
510/350/200	min: 1	min: 100	min: 0.00001	Relu	
510/375/250/125	max: 200	max: 300	max: 0.01		
375/250/125					

In this case, 107 networks have been trained taking approximately 2.7 hours in total or 1.5 minutes per network on average. The top 10 performing networks, based on validation loss, can be seen in Table 14. Here, the run number indicates which consecutive network it is including the 60 networks that have been previously trained on the random search, leading to a total of 167 evaluated networks such that a fair comparison with the results of Bayesian search 1 can be made. This means that the first network trained using Bayesian search 3 has run number 61.

Table 14: Top 10 performing networks from Bayesian search 3.

Run	Architecture	Batch size	Epochs	Learning rate	Activation function	Optimizer	Validation loss
103	510	15	291	0.003844	Sigmoid	Adam	0.001052
126	510	40	291	0.002603	Sigmoid	Adam	0.001182
141	375/250/125	21	220	0.001002	Sigmoid	Adam	0.001210
143	375/250/125	135	288	0.001223	Sigmoid	Adam	0.001330
150	510/375/250/125	76	133	0.002785	Sigmoid	Adam	0.001482
136	510	193	189	0.003619	Sigmoid	Adam	0.001636
132	510/350/200	182	300	0.000980	Relu	Adam	0.001679
133	510	178	170	0.003483	Sigmoid	Adam	0.001710
121	510/375/250/125	21	201	0.000721	Relu	Adam	0.001735
122	510/350/200	169	267	0.000469	Relu	Adam	0.001772

Figure 21 shows the validation loss over the neural networks trained during the Bayesian search. As can be seen, a lot of networks are concentrated around the reference network. This makes sense, as they are trained based on the 4 best performing networks from the random search, which also lay near to the reference network. Some networks still have a relatively high validation loss. These networks are most likely created during an exploration attempt of the Bayesian search.

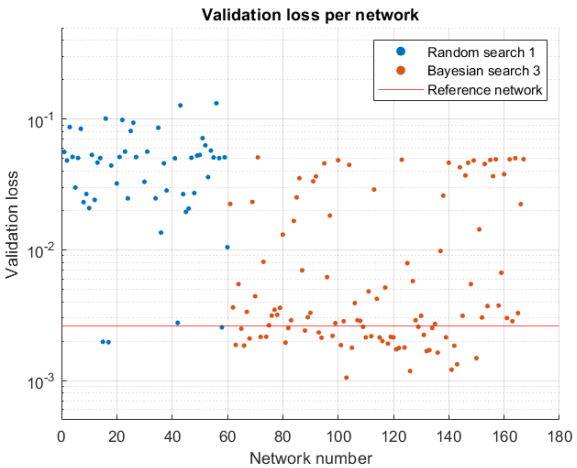
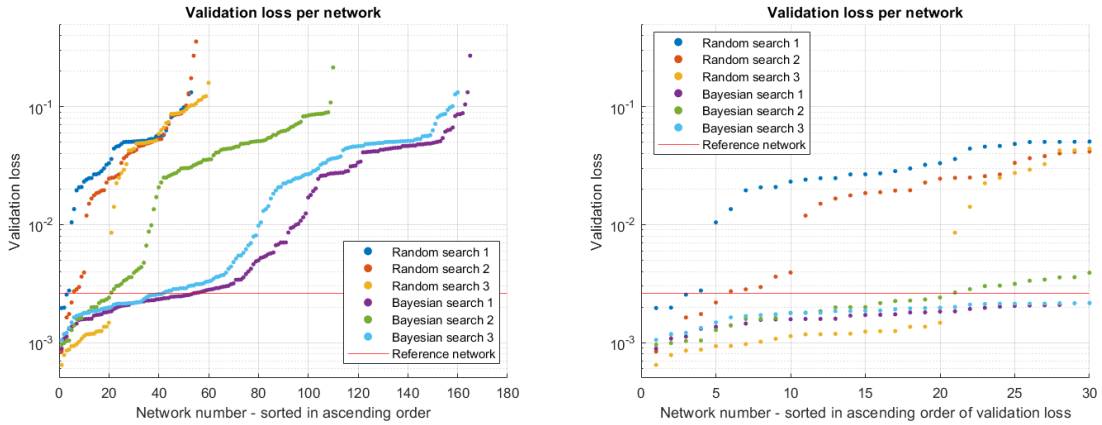


Figure 21: Validation loss as a function of the neural network number obtained from the first random search (blue) and the consecutive third Bayesian search (orange).

6.4 Comparison between automatic hyperparameter tuning methods

Grid search is a good (brute force) method to get an idea of what performance can be reached with a good choice of hyperparameter values/options. It is, however, computationally very expensive and therefore not an advisable method for hyperparameter tuning.

Random search and Bayesian search are both methods that are able to find networks that perform better than a network designed and trained by an inexperienced engineer. All random search runs and Bayesian search runs have been plotted together in Figure 22. From this Figure, it may be concluded with some caution that Bayesian search 1 and 3 are the most promising and efficient approaches, as they have a lot of networks that perform better than the reference network and the amount of required time for these approaches is reasonable. It is important that a method produces many networks that perform better than the reference network, because in this comparison only one identification problem on a specific set of data is investigated. It could therefore mean that a good performance in this comparison is partially based on luck, rather than that it follows from applying a robust method. The larger the set of networks that perform better than the reference network, the more likely it is that on a different training attempt, i.e., on a different data set for a different problem, also a network with a relatively low validation loss will be achieved. This is what also happened with Bayesian search 2. This search was done on the same hyperparameter values/options as Bayesian search 1, but it did not perform as well as Bayesian search 1. It should, however, be noted that Bayesian search 2 only has 117 runs, whereas Bayesian search 1 has 167 runs. This can be part of the reason for the difference in performance. Because a Bayesian search finds a lot of networks that perform well in a relatively successful search, e.g., Bayesian search 1 (which has around a third of the networks performing better than the reference network), it is still able to find enough successful network in a relatively less successful search, e.g. Bayesian search 2 (which only has less than a fifth of the networks performing better than the reference network).



(a) All networks.

(b) Top 30 networks per method.

Figure 22: Validation loss over neural network in ascending order.

Comparing the approaches applied in Bayesian search 1 and 3, it seems to be best to go for the approach that is applied in Bayesian search 1, i.e., choosing a large range of hyperparameter values/options and starting with a Bayesian search immediately opposed to doing a random search first as is carried out before Bayesian search 3. This is conjectured because of the following two reasons.

First of all, it might not be very clear which hyperparameter values/options should be kept and which values/options should be dropped after the random search, as there might not be a very clear preference based on the validation loss. For example, when looking at the top 10 results of random search 2, it would be very difficult to determine which hyperparameter values/options to keep.

Secondly, limiting Bayesian search (by only allowing a limited set of randomly picked exploratory hyperparameter values/options) might deteriorate the Bayesian because it may not find the global minimum anymore. To illustrate this: if the values of the random search end up in a (good) local minimum which is far away from the global minimum, it is difficult or even impossible for the Bayesian search to find hyperparameter values/options close to the global minimum. When starting off with a Bayesian search from the beginning, however, this limitation does not hold.

To make a fair comparison in performance between the best performing network from Bayesian search run 1 and the reference network, this best performing network is analysed on the test set. This results in a test loss of 0.000937, which is slightly worse than the performance on the validation set. The error metrics as described in Equations (30), (31) and (32) result in the values listed in Table 15. Compared to Table 3, most values are improved by a factor in between 2 and 5, while some values (only related to μ_ϵ) perform worse. Overall, the network created with Bayesian search 1 performs better than the reference network.

Table 15: Error metrics based on test data results from best performing network from Bayesian search 1.

	μ_ϵ [%]	σ_ϵ [%]	$\mu_{\lvert\epsilon\rvert}$ [%]
k_y	0.63	0.86	0.88
k_θ	0.08	0.90	0.70
d_y	-0.42	0.44	0.47
d_θ	-0.30	0.31	0.36

7 Conclusions & recommendations

This report focuses on finding an automatic hyperparameter tuning method that can find a neural network that outperforms a manually trained neural network for the inverse mapping parameter updating method. Grid search, random search and Bayesian search have been explored, and all three methods are able to outperform a manually trained neural network. Taking both computational cost and chance of success, i.e., producing many networks that outperform the manually trained neural network based on the validation loss, into account, Bayesian search seems to be the best performing method. This method is able to outperform a manually trained neural networks, even when the hyperparameter values/options are set to large ranges, i.e., set by an inexperienced engineer. It should be noted that the qualitative as well as quantitative results as reported in this report have been tested on a specific use case and a specific data set, and thus results may differ when investigating other use cases and/or data sets. However, based on the large chance of success Bayesian search has, it is believed that this method also works on different, more complex use cases, and on different data sets. This should be investigated in the future.

Apart from testing these tuning methods on different use cases and different data sets, other future work should include a Bayesian search where the hyperparameter values/options are based on a random search, in which the hyperparameter values/options in the random search should be set equal to the hyperparameter values/options of Bayesian search 1 and 2. This will lead to a fairer comparison between Bayesian search 3 and Bayesian searches 1 and 2. Furthermore, a larger grid should be explored using grid search, especially for the epochs, as this might give interesting information about what performance is achievable with a larger number of epochs. In addition, it should be attempted to set the network architecture as a variable, meaning that the structure of the network is set to a hyperparameter indicating the number of layers, and that for each layer a hyperparameter exists that indicates the number of neurons present in that layer. This may help in finding a better structure, especially when using Bayesian search. Lastly, the training time of a neural network should be compared with its performance, to see how these affect each other.

References

[1] ASM-PT. *ASM Pacific Technology*. 2021. URL: https://www.asmpacific.com/en/.

[2] B.M. Kessels, R.H.B. Fey, M.H. Abbasi, and N. van de Wouw. "Model updating for nonlinear dynamic digital twins using data-based inverse mapping models". In: IMAC-XL 2022. Orlando, USA, Feb. 2022.

[3] S. Rahan. *An Introduction to Artificial Neural Networks*. 2020. URL: https://towardsdatascience.com/an-introduction-to-artificial-neural-networks-5d2e108ff2c3.

[4] R. Dagli. *The Art of Hyperparameter Tuning in Deep Neural Nets by Example*. 2021. URL: https://towardsdatascience.com/the-art-of-hyperparameter-tuning-in-deep-neural-nets-by-example-685cb5429a38.

[5] S. Es. *Hyperparameter tuning in Python: A complete guide 2021*. 2021. URL: https://neptune.ai/blog/hyperparameter-tuning-in-python-a-complete-guide-2020.

[6] L. Shukla. *Hyperparameter tuning for Keras and Pytorch models*. 2020. URL: https://wandb.ai/site/articles/hyperparameter-tuning-as-easy-as-1-2-3.

[7] D. Mikhail and K. Arkadi. "A classification of meteor radio echoes based on artificial neural network". In: *Open Astronomy* 27 (Dec. 2018), pp. 318–325. DOI: 10.1515/astro-2018-0037.

[8] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. USA: Oxford University Press, Inc., 1995.

[9] Tensorflow. *Built-in optimizer classes*. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers.

[10] S. Doshi. *Various Optimization Algorithms For Training Neural Network*. 2019. URL: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6.

[11] V. Bushaev. *Adam — latest trends in deep learning optimization*. 2018. URL: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.

[12] Peltarion. *Mean squared logarithmic error (MSLE)*. URL: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-logarithmic-error-(msle).

[13] J. Jordan. *Setting the learning rate of your neural network*. 2018. URL: https://www.jeremyjordan.me/nn-learning-rate/#:~:text=If%5C%20your%5C%20learning%5C%20rate%5C%20is,behavior%5C%20in%5C%20your%5C%20loss%5C%20function..

[14] J. Konar, P. Khandelwal, and R. Tripathi. "Comparison of Various Learning Rate Scheduling Techniques on Convolutional Neural Network". In: *2020 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)* (2020), pp. 1–5.

[15] J. Brownlee. *How to Choose an Activation Function for Deep Learning*. 2021. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

[16] G. De Luca. *Neural Network Architecture: Criteria for Choosing the Number and Size of Hidden Layers*. 2020. URL: https://www.baeldung.com/cs/neural-networks-hidden-layers-criteria.

[17] Towards AI. *Underfitting Overfitting — The Thwarts of Machine Learning Models' Accuracy*. 2020. URL: https://towardsai.net/p/machine-learning/underfitting-overfitting%5C%E2%5C%80%5C%8A-%5C%E2%5C%80%5C%8Athe-thwarts-of-machine-learning-models%5C%E2%5C%80%5C%8Baccuracy.

[18] A. Zheng. "Chapter 4: Hyperparameter tuning". In: *Evaluating Machine Learning Models*. USA: O'Reilly Media, Inc., 2015.

[19] J. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305.

[20] Scikit learn. *Gaussian Processes*. URL: https://scikit-learn.org/stable/modules/gaussian_process.html.

[21] Paretos. *Bayesian Optimization (Bayes Opt): Easy explanation of popular hyperparameter tuning method*. 2021. URL: https://www.youtube.com/watch?v=M-NTkxfd7-8&t=170s.

[22] WandB. URL: https://wandb.ai/site.

[23] Neptune. URL: https://neptune.ai/.

[24] Tensorflow. URL: https://www.tensorflow.org/tensorboard.

[25] Comet. URL: https://www.comet.ml/site/.

[26] WandB. *Sweep Configuration*. 2022. URL: https://docs.wandb.ai/guides/sweeps/configuration.