AE4320: System Identification of Aerospace Vehicles

System Identification Using Neural Networks

José Cunha - 5216087

Delft University of Technology

TU Delft

Contents

Nomenclature

EKF Extended Kalman Filter

FNN Feed-Forward Neural Network

$IEKF$ Iterated Extended Kalman Filter

LKF Linear Kalman Filter

LMA Levenberg-Marquardt Algorithm

MSE Mean Squared Error

OLS Ordinary Least Squares

RBF Radial Basis Function

RSS Residual Sum of Squares

Introduction

The aim of this report is to perform state and parameter estimation of an F-16 using flight data. The report is divided into three parts: state and parameter estimation using a Kalman filter and least squares estimator approaches respectively, creation of a radial basis function (RBF) neural network and creation of a feed-forward neural network both to reconstruct the F-16 dataset. Finally, the methods for parameter estimation are compared in terms of their validation metrics. Please refer to the `README.md` file or the *GitHub repository* for a description of the code files.

1 State & Parameter Estimation

1.1. Dataset treatment

For this assignment, an F-16 flight data dataset was provided, comprised of a series of inputs and outputs of the F-16. However, it has been reported that some of the measurements are biased (due to aerodynamic effects) and another measurement contains sensor glitches. For the biased measurement, a state estimator can be implemented to reconstruct the output, removing the bias (this shall be treated further into the section). Regarding the sensor glitch in the C_m measurements, the dataset can be treated to remove these corrupted data points.

Looking at the C_m plot in Figure 1.1, a series of outlier values can be identified. These values can simply be removed from the dataset. Even though this shortens the number of available training points, it improves the accuracy of the parameter estimation methods, discussed later in the report, as the outliers would affect the parameter estimation especially in their local region (as seen later in Figure 1.6).

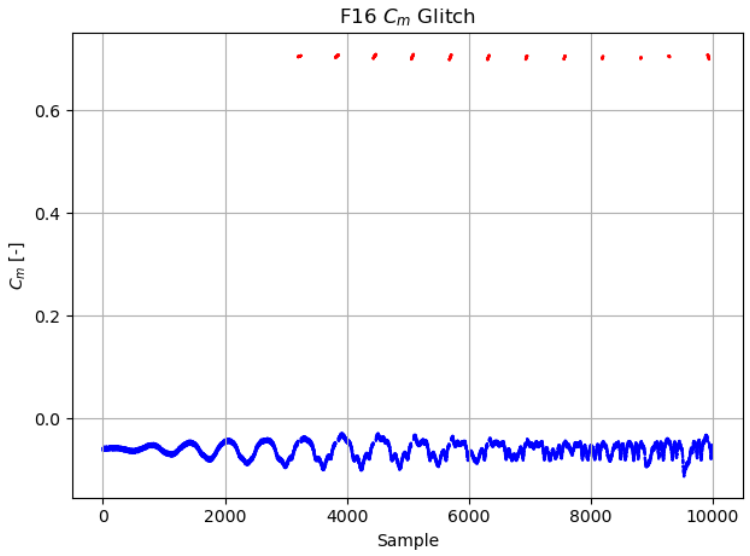

Figure 1.1: *Plot of the C_m time series data. Points in red are the outliers of the dataset, and were removed when treating the data.*

Considering the definition of identification and validation datasets, this was done for the parameter estimation methods - ordinary least-squares (OLS), radial basis function neural networks (RBF) and feed-forward neural networks (FNN) - in order to have available data to validate the model. Data was slit 50-50 into an identification and validation set, wherein the points were randomly sampled into either set (to ensure that the validation set contains sufficient dynamics).

1.2. Kalman filtering

Due to the unknown bias in the angle of attack measurements (α_m), a Kalman filter state estimator can be set up such that one of the states of the system can be estimated so that the angle of attack measurements can be reconstructed without any bias. To achieve this, one must first formulate the system and output equations to specify the relationship between the states and the outputs, which can be seen in Equation 1.2 and Equation 1.3 respectively.

$$x_k = \begin{bmatrix} u & v & w & C_{\alpha_{up}} \end{bmatrix}^T \quad \text{and} \quad u_k = \begin{bmatrix} \dot{u} & \dot{v} & \dot{w} \end{bmatrix}^T \tag{1.1}$$

$$\dot{x}_k = f(x_k, u_k, t) = \begin{bmatrix} \dot{u} & \dot{v} & \dot{w} & 0 \end{bmatrix}^T = \begin{bmatrix} u_{k1} & u_{k2} & u_{k3} & 0 \end{bmatrix}^T \tag{1.2}$$

$$z = h(x_k, u_k, t) = \begin{bmatrix} \alpha_{true}(1 + C_{\alpha_{up}}) \\ \beta_{true} \\ V_{true} \end{bmatrix} = \begin{bmatrix} \arctan\left(\frac{w}{u}\right) \\ \arctan\left(\frac{v}{\sqrt{u^2+w^2}}\right) \\ \sqrt{u^2+v^2+w^2} \end{bmatrix} = \begin{bmatrix} \arctan\left(\frac{x_{k3}}{x_{k1}}\right) \\ \arctan\left(\frac{x_{k2}}{\sqrt{x_{k1}^2+x_{k3}^2}}\right) \\ \sqrt{x_{k1}^2+x_{k2}^2+x_{k3}^2} \end{bmatrix} \tag{1.3}$$

With the system and output equations found, one must now consider which type of Kalman filter to use for state estimation, namely the linear Kalman filter (LKF), the extended Kalman filter (EKF) or the iterated extended Kalman filter (IEKF). Due to the nonlinear nature of the system, an LKF would be a poor choice due to its limitation to linear systems. Of the EKF and IEKF, the latter would be the better choice, as the EKF does not provide a guarantee of global convergence, which is improved on by the IEKF as it provides more accurate linearization through local iterations of the updated state. As such, the IEKF shall be used as a state estimator.

A further step to show the convergence of the IEKF for the F-16 system, is to analyze the observability matrix of the system, which can be calculated from the Lie derivatives of output equations with respect to the system equation (where ∂_x denotes the Jacobian, and $L_f^n h$, the n$^{\text{th}}$ Lie derivative of h w.r.t f). The system is said to be observable if it satisfies Equation 1.4. If the system is in fact observable, then the IEKF will converge. For the system described in Equation 1.2 and 1.3, its observability matrix has full rank (rank(O) = 4).

$$\text{rank}(O) = n \qquad O = \begin{bmatrix} \partial_x h \\ \partial_x (L_f^1 h) \\ \partial_x (L_f^2 h) \end{bmatrix} \tag{1.4}$$

With this in mind, an IEKF can be constructed to estimate the $C_{\alpha_{up}}$ state, the result of which can be seen in Figure 1.2. From this figure, the convergence of the IEKF can also be seen, as the $C_{\alpha_{up}}$ state converges to a final value of $C_{\alpha_{up}} = 0.3091$.

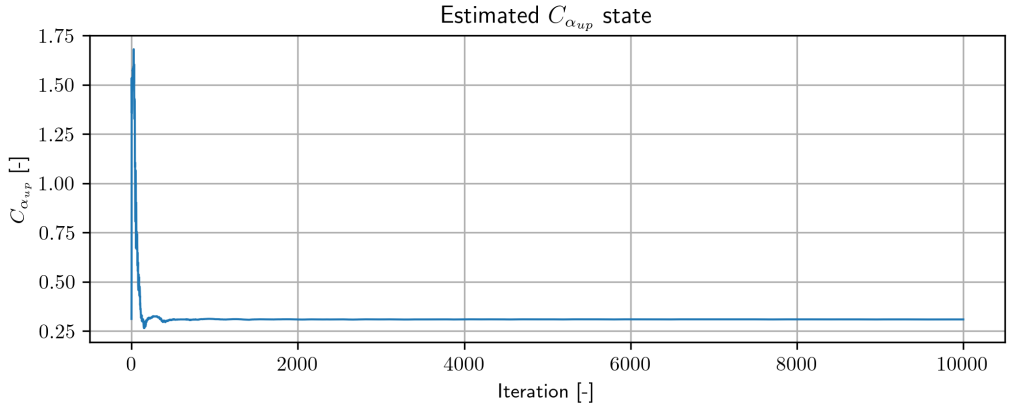

Figure 1.2: *$C\alpha_{up}$ state estimation using IEKF.*

With this value, the unbiased estimate of the angle of attack can be calculated, according to Equation 1.5. With this, the estimated outputs from the IEKF can be plotted, seen in Figure 1.3 and Figure 1.4

$$\alpha_{true} = \frac{\alpha_m}{1 + C_{\alpha_{up}}} \tag{1.5}$$

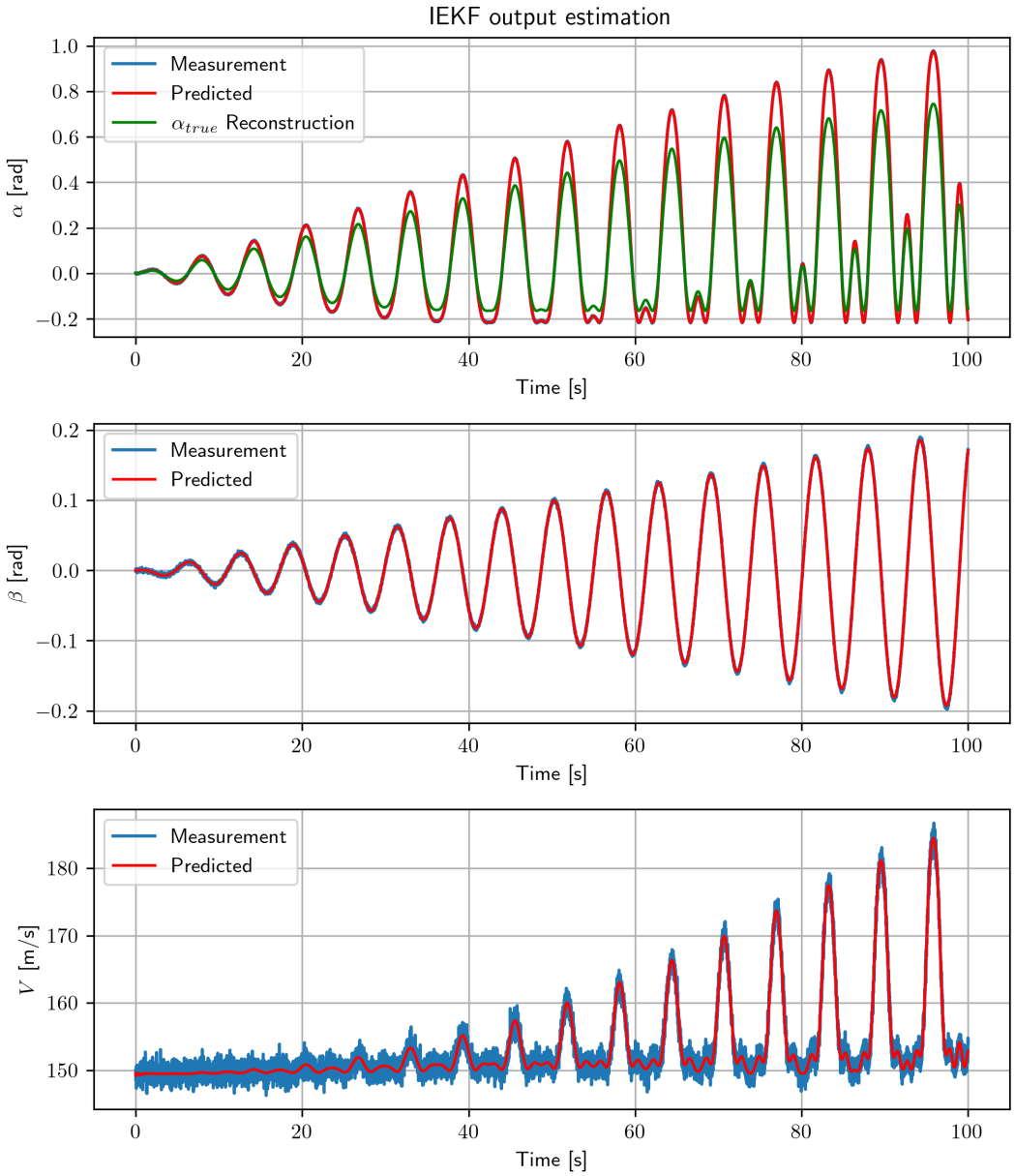

Figure 1.3: *Estimation of outputs (α, β, V) from IEKF state estimation.*

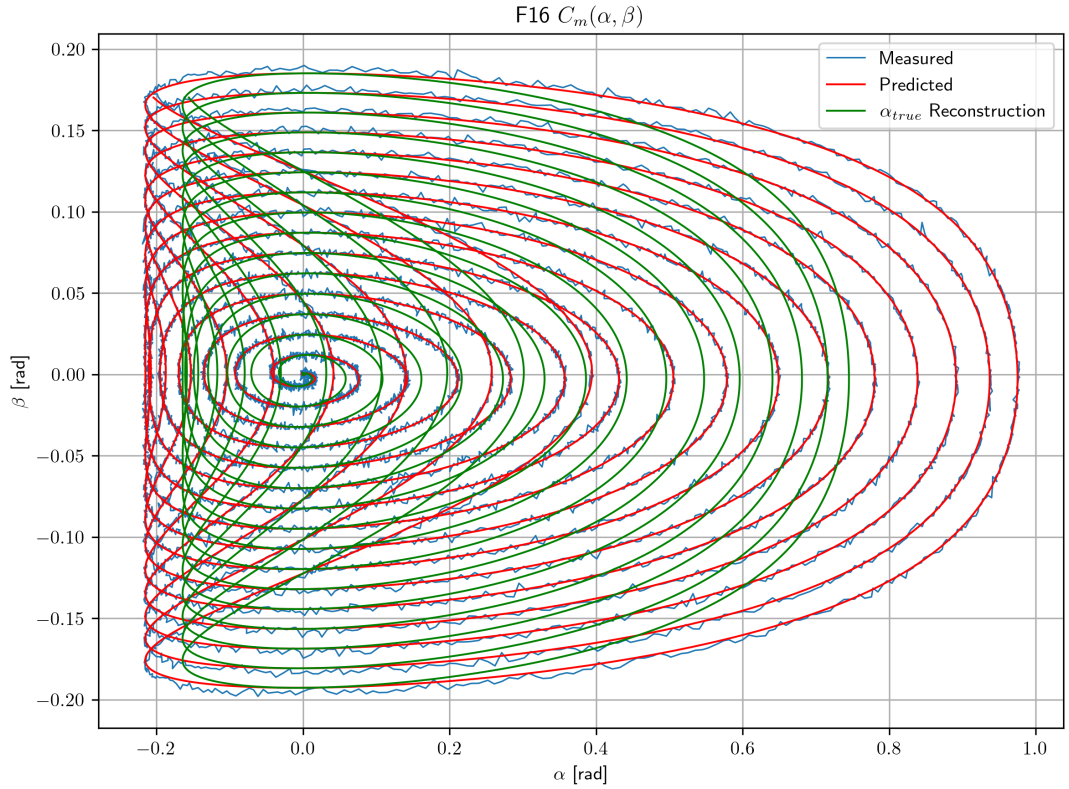

Figure 1.4: *Comparison of α and β raw measurements, IEKF prediction and unbiased α_{true} reconstruction.*

The differences between the measured α_m and the reconstructed unbiased α_{true} can be seen in both Figure 1.3 and Figure 1.4. The range of the true angle of attack is somewhat smaller than that of the biased value due to upwash, and the variations due to noise are mostly eliminated.

1.3. Least squares parameter estimation

With the IEKF, an unbiased and noise-free estimate for the system outputs was calculated. Now, these measurements can be used to create a model between the system outputs and the C_m measurements, namely to find the function $C_m(\alpha, \beta)$. This is a parameter estimation problem, which can be tackled using ordinary least squares (OLS) parameter estimation.

An OLS problem can be stated as finding a model (i.e. optimal parameters) that best fist a sequence of data, such as in Equation 1.6, where $y = C_m$ (the measurement), $x = [\alpha \quad \beta \quad V]^T$ (the states), $A(x)$ is the regression matrix, θ are the parameters to be optimized and ε the residuals (to be minimized).

$$y = A(x) \cdot \theta + \varepsilon \tag{1.6}$$

With this model structure, an OLS estimator can be constructed using Equation 1.7 (note that $A^+(x)$ denotes the pseudo-inverse), where the complexity of matrix $A(x)$ can be used to modulate the accuracy of the estimator.

$$\begin{aligned} \hat{\theta}_{OLS} &= A^+(x) \cdot y = \left[A^T(x)A(x)\right]^{-1} A^T(x)y \\ \hat{y} &= A(x) \cdot \hat{\theta}_{OLS} \end{aligned} \tag{1.7}$$

An OLS estimator was constructed using a regression matrix of order six (i.e. all polynomial combinations of the states up to order six). For training the OLS estimator, solely the identification/training data partition was used, to reduce issues with overfitting. The results of this OLS parameter estimation ($\hat{y}$) can be seen in Figure 1.5.

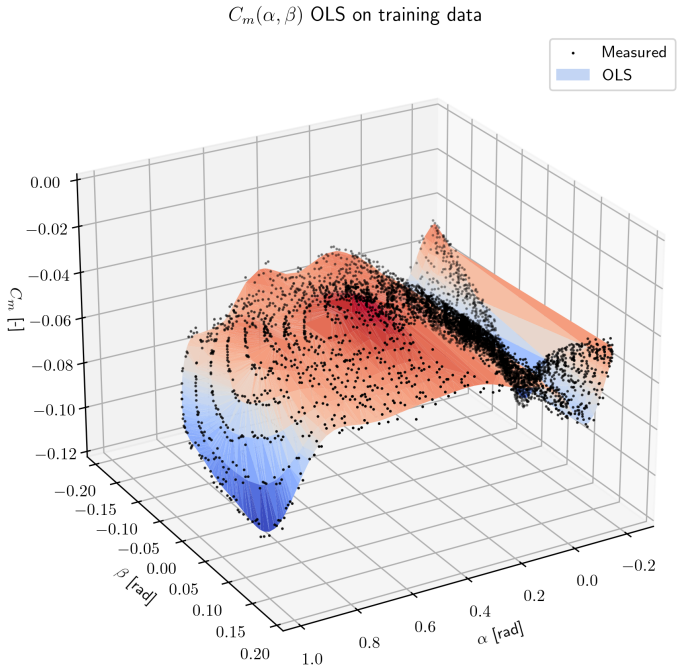

Figure 1.5: $C_m(\alpha, \beta)$ *model estimated using OLS, with treated training data.*

Figure 1.6 shows the importance of performing data preprocessing. As can be seen, the sensor glitch in the C_m measurement has completely skewed the OLS model, rendering it completely inaccurate.

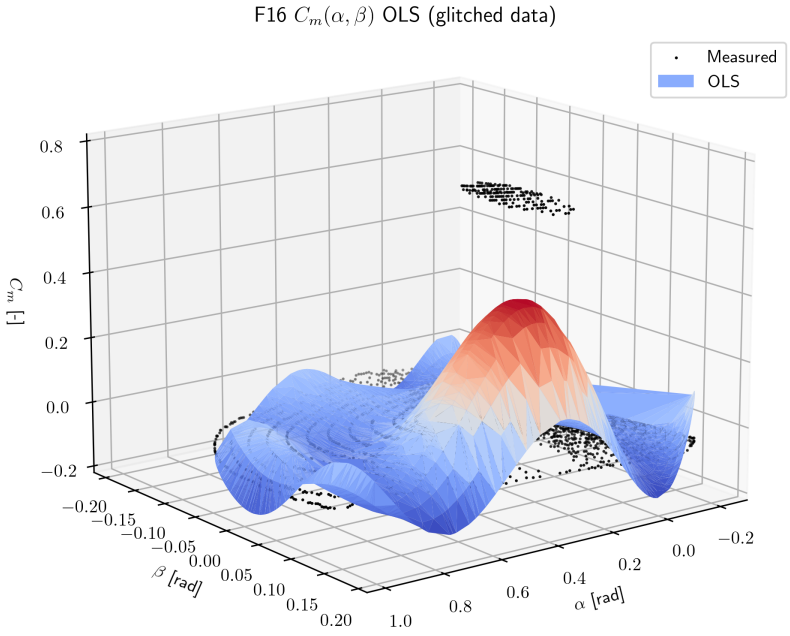

Figure 1.6: $C_m(\alpha, \beta)$ *model estimated using OLS, using glitched* C_m *data.*

In constructing this model, one important hyperparameter had to be considered: the polynomial model order. In Figure 1.5, an order of six was chosen, though this number can have an effect on accuracy of fit. To measure the accuracy, the mean squared error (MSE) was calculated (using Equation 1.8). A series of OLS estimators were constructed using a range of orders, and the accuracy of each can be seen in Figure 1.7.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1.8}$$

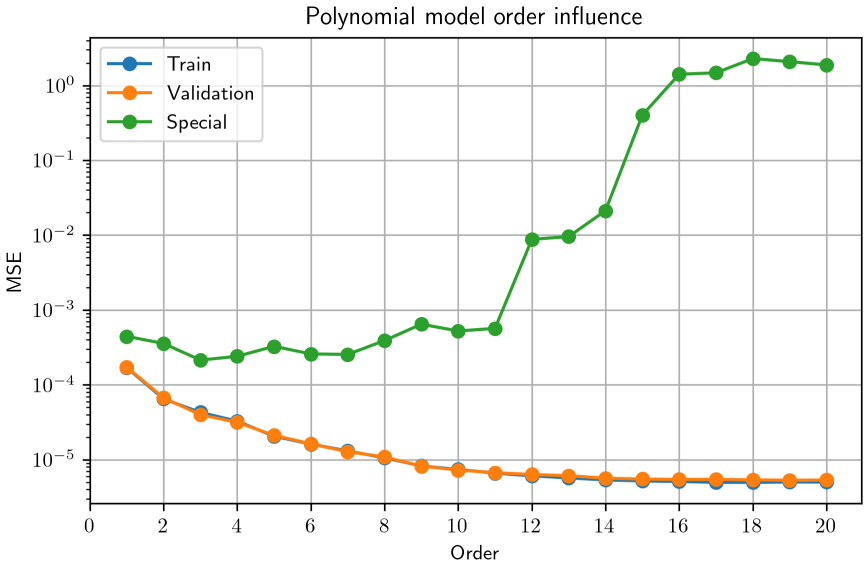

Figure 1.7: *Influence of polynomial model order on the accuracy of an OLS estimation fit, for the three different datasets*

It is evident that for very low orders (linear and quadratic polynomials), the model is not complex enough to capture the nonlinearities of the underlying relationship in the data. However, for orders greater than three, the MSE is much lower, showing a better accuracy of fit. Interestingly, for higher orders, the error does not further decrease, leading to diminishing returns of greater complexity on fit accuracy for orders greater than 6. This same procedure can be applied to the provided 'special validation dataset', which contains much fewer points and whose data points do not necessarily match with that of the training set. Therefore, the error is always higher than for the training/validation sets. Due to overfitting to the training set, for high orders, the fitting accuracy with the special validation set increases considerably, therefore the order of the polynomial OLS model should remain below seven (to reduce the effect of overfitting). The results of the model estimation using the special validation set can be seen in Figure 1.8, which yields a model RMS of 0.0155.

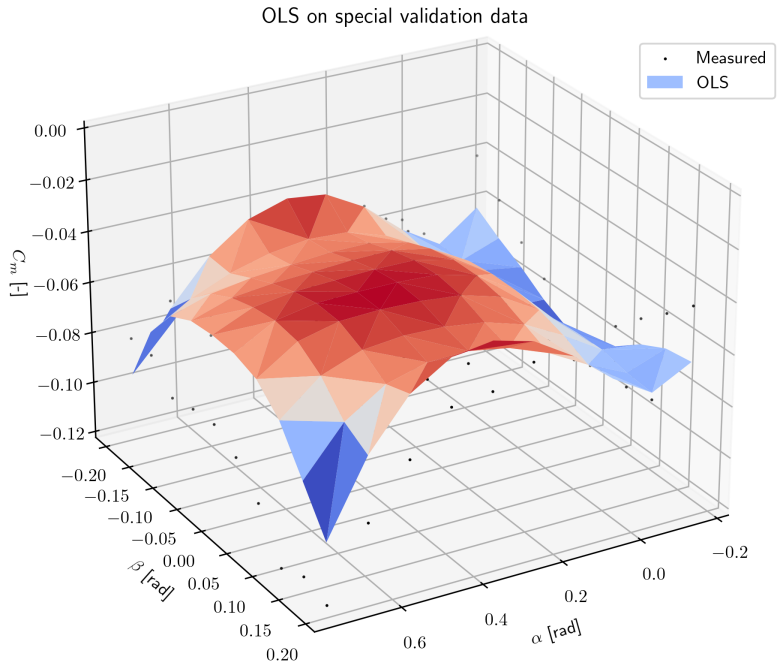

Figure 1.8: $C_m(\alpha, \beta)$ *model estimated using OLS of order six, using special validation dataset.*

Furthermore, to ensure that the model is also applicable to other large datasets, model validation should be performed. This shall be done using two approaches: statistical (analyzing estimator variance) and model-error-based (analyzing model residuals) validation. The statistical approach pertains to finding the covariance matrix of the OLS parameter estimates, and analyzing the parameter variances from it. The covariance matrix can be calculated from the regression matrix and residuals according to Equation 1.9, where the variances of the OLS parameters are the diagonal elements of the covariance matrix. The resulting variances are plotted in Figure 1.9. It can be seen that one parameter has a considerably higher variance than all others, showing some heteroscedasticity in the model. However, as most parameters have the same variance, the homoscedastic assumption is satisfied.

$$\begin{aligned}\mathbb{E}\left\{(\hat{\theta}-\theta)(\hat{\theta}-\theta)^T\right\} &= \mathbb{E}\left\{(A^+(x)\varepsilon)(A^+(x)\varepsilon)^T\right\} \\ &= A^+(x)\mathbb{E}\left\{\varepsilon\varepsilon^T\right\}\left(A^+(x)\right)^T \\ &= A^+(x)\frac{\varepsilon\varepsilon^T}{n-k}\left(A^+(x)\right)^T\end{aligned} \tag{1.9}$$

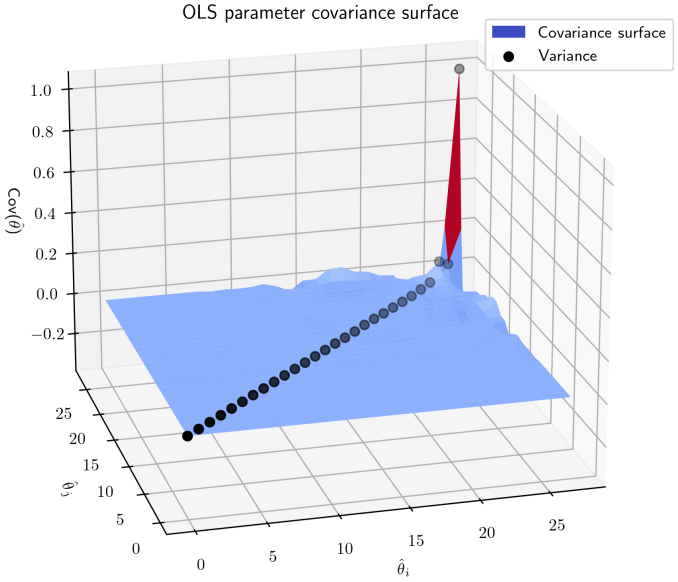

Figure 1.9: *Covariance surface of the OLS parameter estimates using the validation dataset, with the variances of the OLS parameters highlighted in black.*

To perform a model-error based validation, the autocorrelation function can be analyzed. This was calculated using python's `numpy.correlate` method, and the results can be found in Figure 1.10. From the autocorrelation, the whiteness of residual correlations can be assessed, as most points fall within the 95% confidence interval around a correlation of 0, showing that the assumption that the residuals are modelled after zero-mean white noise.

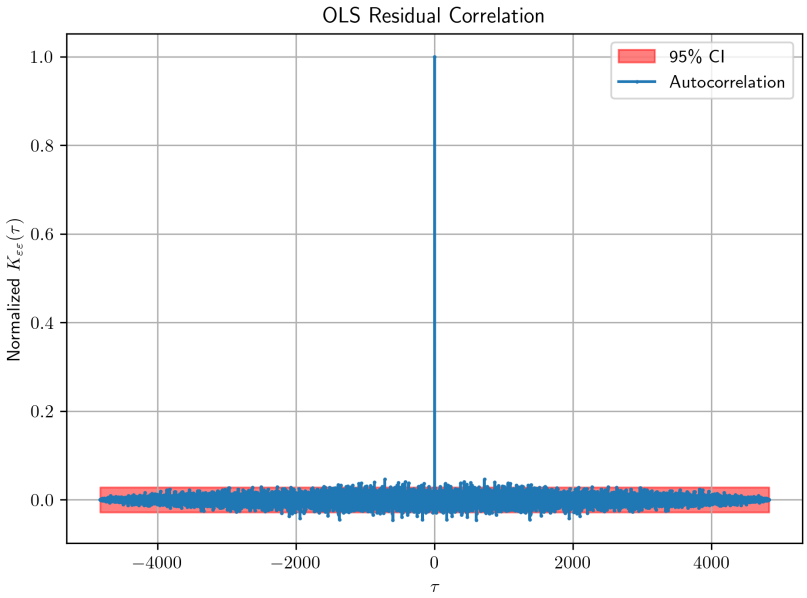

Figure 1.10: *Normalized autocorrelation function of the OLS model residuals.*

2

Radial Basis Function Neural Network Model

2.1. RBF network implementation

In the previous section, OLS linear regression was used as a method for parameter estimation. Another method used for parameter estimation concerns the use of neural networks, due to their property of being 'universal function approximators'. A specific neural network architecture that can be used is the radial basis function (RBF) neural network. Figure 2.1 shows the layout of the RBF network, with three input neurons (corresponding to the three states used to estimate C_m), n-hidden neurons (the number which is to be optimized) and one output neuron (corresponding to the parameter to be estimated).

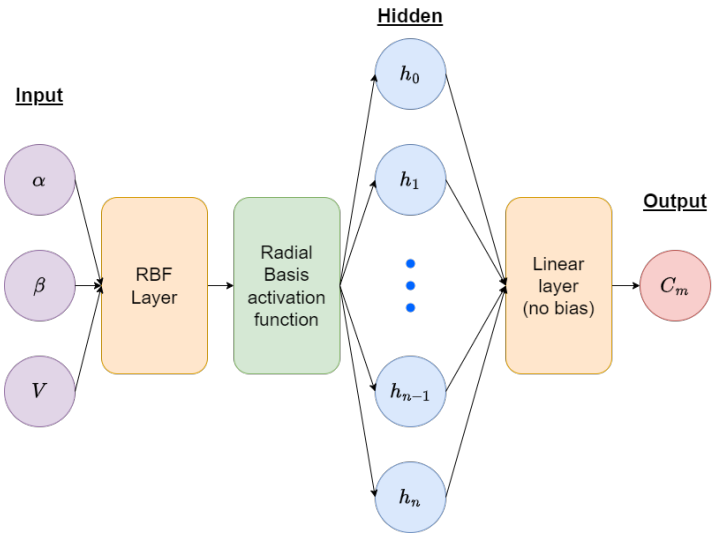

Figure 2.1: *RBF network architecture.*

In between the input and hidden neurons lie the RBF layer (described in Equation 2.1) followed by the radial-basis activation function (Equation 2.2). Connecting the hidden neurons to the output, there is a linear layer (Equation 2.3) that does not contain a bias term.

$$\nu = w_1^2 \cdot (x - c)^2 \tag{2.1}$$

$$\phi(\nu) = a \cdot \exp(-\nu) \tag{2.2}$$

$$h = \phi\left(w_1^2 \cdot (x - c)^2\right) \qquad \hat{y} = w_2 \cdot h \tag{2.3}$$

From the equations describing the RBF network, two hyperparameters appear which need to be chosen, namely the RBF centers and amplitude. Initially, the amplitude is set to one, though this shall be revisited and optimized in later sections. Regarding the centers, one must decide the number and placement of the RBFs using the input data. One common technique is to apply a clustering method

to the input data, and assigning the centers of the clusters to the RBF centers. The number of RBF centers will be the number of clusters chosen, which corresponds to the size of the hidden layer (i.e. number of hidden neurons) - a hyperparameter in itself. The clustering method chosen was Kmeans clustering, which resulted in the following clustering of the input data, and respective RBF centers, seen in Figure 2.2.

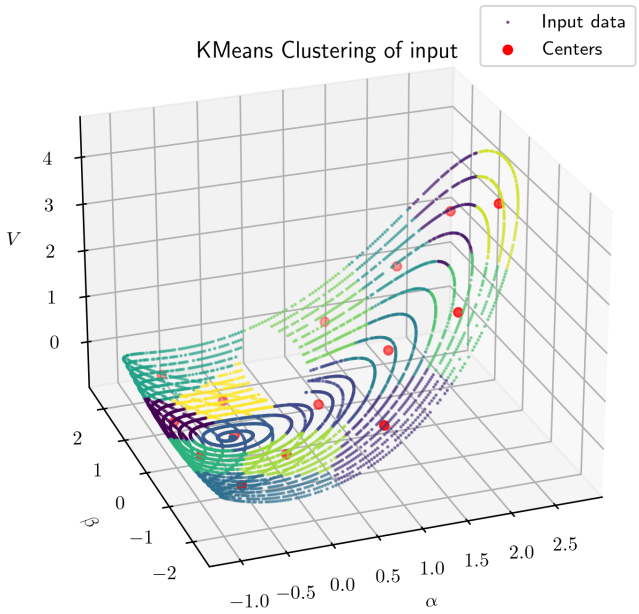

Figure 2.2: *RBF centers found through Kmeans clustering of input data, each cluster being shown in a different color, for 15 RBFs.*

2.2. Linear regression approach

Having described the architecture of the RBF network, it can now be trained to best estimate the C_m model from input data. One approach for 'training' an RBF network uses linear regression to calculate the weights of the network, namely for the linear output layer, as it depends linearly on the previous layer and does not contain a bias. Equation 2.4 can be used to calculate the weights of the linear output layer, and the RBF weights are initialized randomly and left as such. Applying this equation to the RBF network yields the updated model for C_m, shown in Figure 2.3

$$w_2 = h^+ y = \left[h^T h\right]^{-1} h^T y \tag{2.4}$$

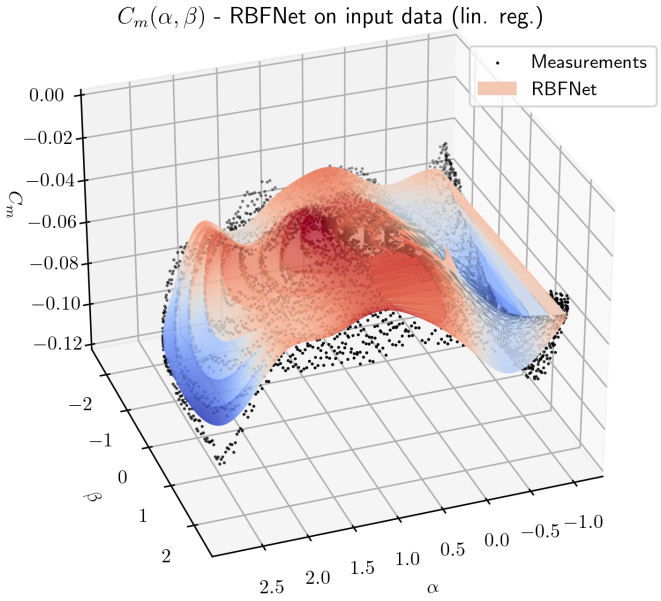

Figure 2.3: $C_m(\alpha, \beta)$ *model estimated using an RBF neural network 'trained'with linear regression, using 20 hidden neurons / RBFs.*

This procedure leaves one optimization parameter to be designed - the RBF amplitude a. This was done by running the parameter estimation for a series of different a values, and finding which one lead to the highest accuracy (using mean-squared error as the measure for this). This procedure, however, also depends on the number of RBFs. As a demonstration case, 20 RBFs were used leading to an optimal a of 0.98. Looking at Figure 2.4, one can see that the relationship between accuracy and RBF amplitude seems practically random, and for most amplitudes, a good accuracy is achieved. This randomness comes from the fact that the linear regression approach only updates the weights of the linear layer and not of the RBF layer (which have a nonlinear relationship to the input). This causes the RBF network to essentially reduce to an OLS problem and the RBF layer to serve as a random initialization to the problem. Therefore, to properly leverage the power of RBF networks, a different training algorithm should be used.

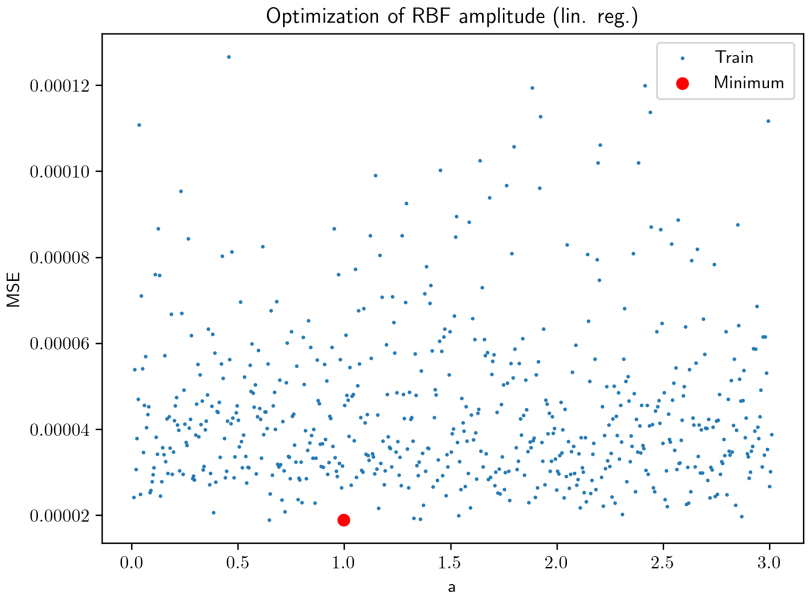

Figure 2.4: *Optimization of RBF amplitude a using 20 RBFs and a linear regression approach.*

2.3. Levenberg-Marquardt learning algorithm

Given the conclusion of the linear regression approach to training RBF networks, a different training algorithm should be used, namely the Levenberg-Marquardt algorithm (LMA). This algorithm updates the model parameters using the error of the prediction and the ground truth (i.e. actual C_m measurements) and the Jacobian matrix of this error w.r.t. the model parameters. The definition of the error and weight vectors can be seen in Equation 2.5 and Equation 2.6 respectively. For the weights, both the RBF amplitude and centers are considered trainable parameters and will be estimated by the training algorithm to reduce the error.

$$E = \frac{1}{2}(y - \hat{y})^2 \tag{2.5}$$

$$W = \begin{bmatrix} w_1 & w_2 & a & c \end{bmatrix}^T \tag{2.6}$$

The Jacobian is defined in Equation 2.7, using the chain-rule to find the gradient of the error vector to each element of the weight vector. Finally, the weight/model parameter update formula (i.e. the LMA) is found in Equation 2.8.

$$\begin{aligned} J = \frac{\partial E}{\partial W} &= \begin{bmatrix} \frac{\partial E}{\partial w_1} & \frac{\partial E}{\partial w_2} & \frac{\partial E}{\partial a} & \frac{\partial E}{\partial c} \end{bmatrix} \\ \frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \phi} \cdot \frac{\partial \phi}{\partial \nu} \cdot \frac{\partial \nu}{\partial w_1} &= -(y - \hat{y}) \cdot w_2 \cdot (-h) \cdot 2w_1(x - c)^2 \\ \frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2} &= -(y - \hat{y}) \cdot h \\ \frac{\partial E}{\partial a} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \phi} \cdot \frac{\partial \phi}{\partial a} &= -(y - \hat{y}) \cdot w_2 \cdot \exp(-\nu) \\ \frac{\partial E}{\partial c} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \phi} \cdot \frac{\partial \phi}{\partial \nu} \cdot \frac{\partial \nu}{\partial c} &= -(y - \hat{y}) \cdot w_2 \cdot (-h) \cdot -2w_1^2(x - c) \end{aligned} \tag{2.7}$$

$$W_{t+1} = W_t - (J^T J + \mu I)^{-1} J^T E \tag{2.8}$$

One of the most important parameters in the LMA s the damping parameter (μ). This value controls the step size of the weight updates, wherein a large μ will lead to a small step (more similar to gradient descent methods such as Adam), and conversely for a low μ. This value can also be controlled dynamically throughout the training process, depending on the output of the RBF network. The process for dynamically updating the damping value is detailed in Algorithm 1. This consists of first computing the updated parameters using Equation 2.8, checking if the residual sum of squares (RSS) is greater than the previous network output. If so, then the damping factor is reduced and the parameters are in fact updated. If the new RSS is greater, then the damping factor is increased and the parameters are not updated. This is performed until the damping factor reaches a very large number, which means the network has converged.

With the weight update and the dynamic update of the damping factor defined, the RBF neural network can be trained with the LMA to model the C_m dataset, the results of which can be seen in Figure 2.5.

Algorithm 1 Levenberg-Marquardt learning algorithm (with μ update)

for epoch in N_epochs **do**
  if update is *True* **then**
    $l \leftarrow \sum(y - \hat{y})^2$
    $J \leftarrow$ calculate Jacobian
    $E \leftarrow$ calculate error
  end if
  $W_{t+1} \leftarrow W_t - (J^T J + \mu I)^{-1} J^T E$
  $\hat{y}_{new} \leftarrow$ RBFNet.forward()
  $l_{new} \leftarrow \sum(y - \hat{y}_{new})^2$
  if $l_{new} < l$ **then**
    update $\leftarrow$ *False*
    $\mu \leftarrow \mu / \Delta\mu_{dec}$
    $W_{t+1} \leftarrow W_t - (J^T J + \mu I)^{-1} J^T E$
    $l \leftarrow \sum(y - \hat{y})^2$
  else if $l_{new} > l$ **then**
    update $\leftarrow$ *True*
    $\mu \leftarrow \mu \cdot \Delta\mu_{inc}$
  else if $\mu > 1 \times 10^8$ **then**
    break
  end if
end for

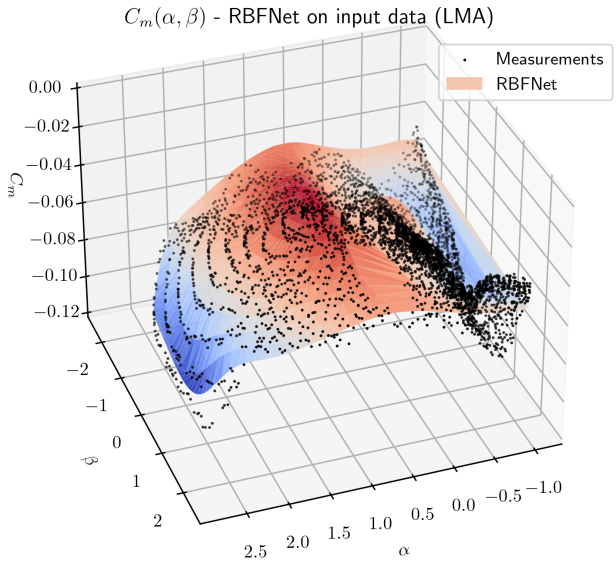

Figure 2.5: $C_m(\alpha, \beta)$ *model estimated using an RBF neural network trained with the Levenberg-Marquardt learning algorithm for* ~ 150 *epochs, with 30 RBFs.*

Comparing the LMA and linear regression approaches, these have clear differences in terms of sensitivity to initial conditions and convergence speed. The linear regression approach converges in a single time step, as a single calculation is required to find the model, which also makes this method insensitive to initial conditions. On the other hand, the LMA approach can require >100 epochs to converge to an acceptable loss value. However, as seen in Figure 2.6, there is a high sensitivity to initial conditions, as when using the same network configuration, some runs converge to too high loss values. In terms of approximation accuracy, the linear regression approach can achieve better accuracies, though at the cost of high overfitting due to the large polynomial orders required, and very low model flexibility, as solely a single parameter can be actually trained (namely the RBF amplitude).

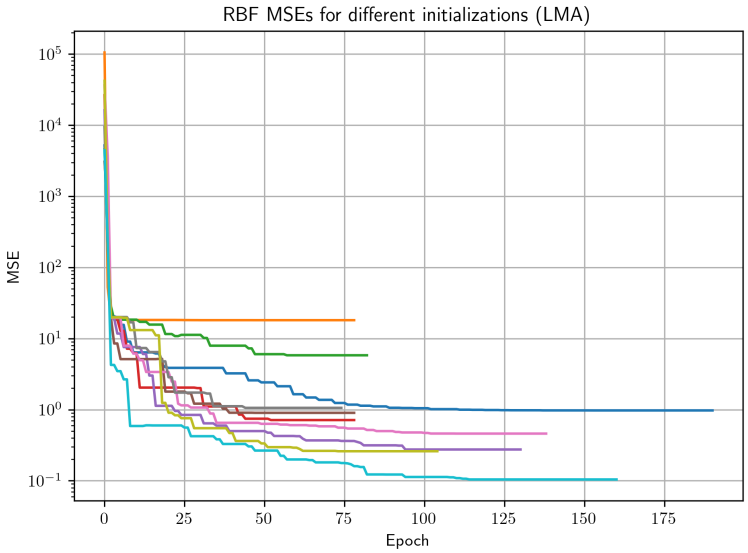

Figure 2.6: *MSEs of different RBFNet (LMA) runs with different initial conditions, but same network configuration (30 hidden neurons).*

To demonstrate why a dynamic updating of the damping factor is important, the network can be trained with a static damping, and the RSS curve can be plotted to see the convergence of the network. This can be done for a range of μ values to show the effect of damping on network convergence, which can be seen in Figure 2.7.

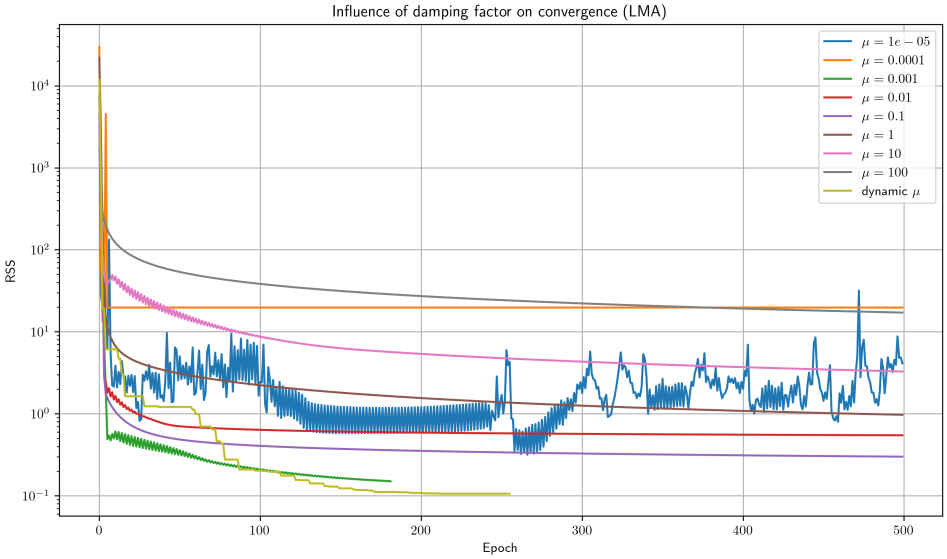

Figure 2.7: *Influence of damping factor μ on the RSS curves (i.e. network convergence) for the same network structure (30 hidden neurons).*

As can be seen from Figure 2.7, having too low a damping factor (e.g. $\mu = 1 \times 10^{-5}$) can lead to instability in the LMA, as the gradient update step is too large, leading to sudden increases in RSS. On the other hand, if the damping factor is too high (e.g. $\mu > 0.01$), the network will not converge to an acceptable RSS in a reasonable number of epochs. Considering the curve for $\mu = 0.001$, though it has some instability, it does relatively quickly converge to a good RSS value. However, this could be quite variable depending on the network structure, and careful retuning would have to be performed each time the number of hidden neurons is changed. To counteract this, and to combine the speed of low damping with the stability of high damping, a dynamic damping factor can be employed, which according to Figure 2.7, can even yield better accuracy.

The final parameter that can be modified in the LMA-trained RBF network is its structure, in this case, the number of hidden neurons. This hyperparameter can be used to tune the network to achieve the best balance between complexity and performance, in other words, achieve both low training and validation accuracy. To optimize the network in terms of total number of neurons, the number of hidden neurons can be varied, and the network trained on the three main datasets (training, validation and special validation). Then, the best network structure will be the one where the sum of the MSEs across all datasets is minimized. This is done to ensure both a good training and validation accuracy. Figure 2.8 shows the MSE of the RBFNet when trained on the three main datasets for a series of hidden dimension sizes, yielding the optimal number of hidden neurons at 42. Comparing this new structure to the one that was previously used (i.e. 30 hidden neurons), the approximating accuracy of the model has increased, as can be seen from the lower MSE for all datasets, as well as a lower RMS (0.00721 as opposed to 0.0144)

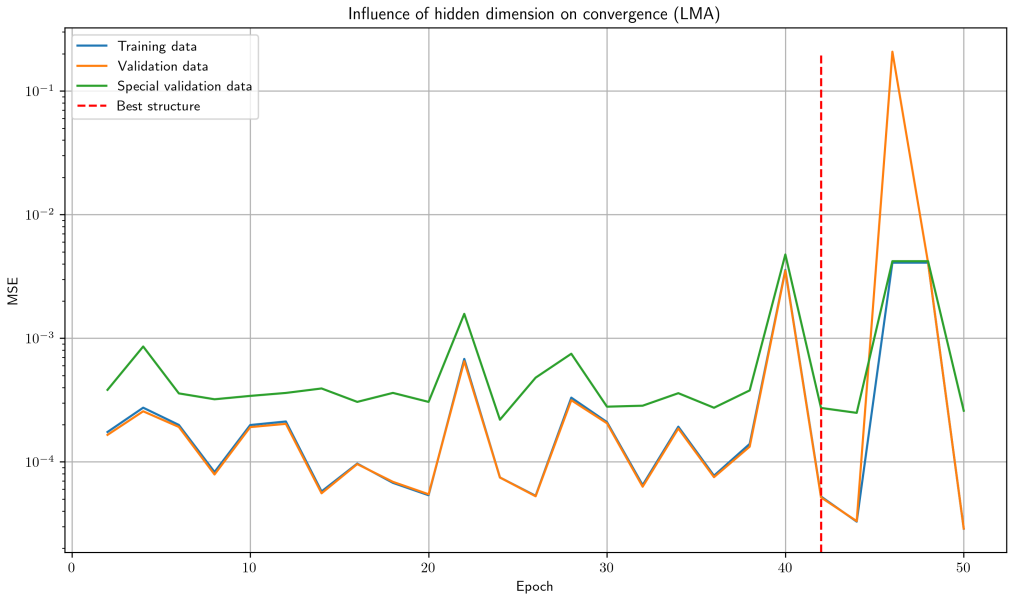

Figure 2.8: *Optimizing RBFNet in terms of number of neurons. Best structure with 42 hidden neurons.*

Finally, this optimized RBF network, trained using the LMA, can be validated on the special validation set, the results of which are shown in Figure 2.9. This estimated model yields a relatively high RMS of 0.0165, higher than that of the OLS estimation model of 0.0155.

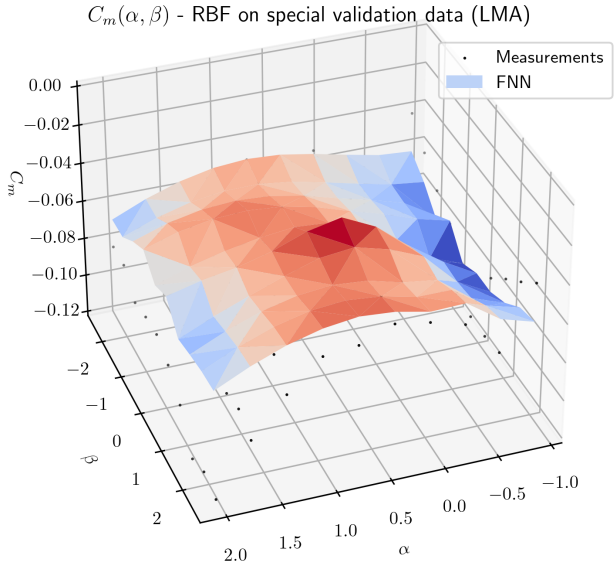

Figure 2.9: $C_m(\alpha, \beta)$ *model estimated using RBFNet (LMA), using the special validation dataset.*

3

Feed-Forward Neural Network Model

3.1. FNN implementation & backpropagation

As explored in the previous section, neural networks can be used for parameter estimation. A different architecture that can be used is the feedforward neural network (FNN). In a sense, RBF networks are also feed-forward networks, as they are structured as a sequence of layers where information flows in one direction. However, for RBF networks differ as they use a Gaussian function as the activation, unlike other typical FNNs that use ReLU, sigmoid, or tanh. In this report, the FNN architecture is given in Figure 3.1, consisting of a linear layer with tanh activation function connecting the input and hidden neurons (Equation 3.1 and Equation 3.2), and another linear layer connecting the hidden to the output neurons (Equation 3.3).

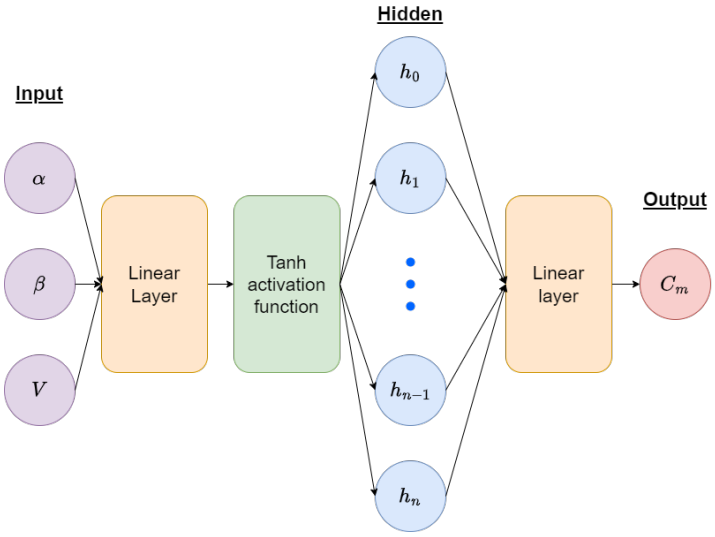

Figure 3.1: *Feed-forward network architecture.*

$$h = \phi\left(w_1 \cdot x + b_1\right) \tag{3.1}$$

$$\phi(\nu) = \frac{2}{1 + \exp\left(-2\nu\right)} - 1 = \tanh\left(\nu\right) \tag{3.2}$$

$$\hat{y} = w_2 \cdot h + b_2 \tag{3.3}$$

To train this FNN (i.e. update the model parameters to achieve a good estimation) a backpropagation algorithm can be used. Backpropagation is an efficient method of calculating the gradient of the error w.r.t. the model parameters, and then this gradient can be used to update the model parameters according to an update law. The update law chosen was the Adam optimizer[1], which uses estimates of the mean and variance of the gradients to calculate the updates of the weight. This update law can

[1] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

be seen in Equation 3.4, where ∇_W is the gradient of the loss function (mean-squared error) w.r.t. the model parameters, η is the learning rate and the ρ values are coefficients usually set t 0.9 and 0.99 respectively.

$$\begin{aligned} v_t &= \rho_v v_{t-1} + (1-\rho_v)\nabla_W \\ r_t &= \rho_r r_{t-1} + (1-\rho_r)(\nabla_W)^2 \\ W_{t+1} &= W_t - \eta \frac{v_t}{\sqrt{r_t}} \end{aligned} \tag{3.4}$$

As such, an FNN can be trained with a backpropagation algorithm (namely the Adam optimizer, which is built into the PyTorch [2] Python library) to estimate the C_m model. A series of hyperparameters can be varied, namely the size of the hidden layer and the learning rate, though an example of such a configuration can be found in Figure 3.2. After some manual tuning of the learning rate, it was found that a decaying η value was advantageous for reducing the MSE at an acceptable convergence time, as starting with a high η speeds up convergence, while reducing to a lower η over time improves stability at lower MSE values.

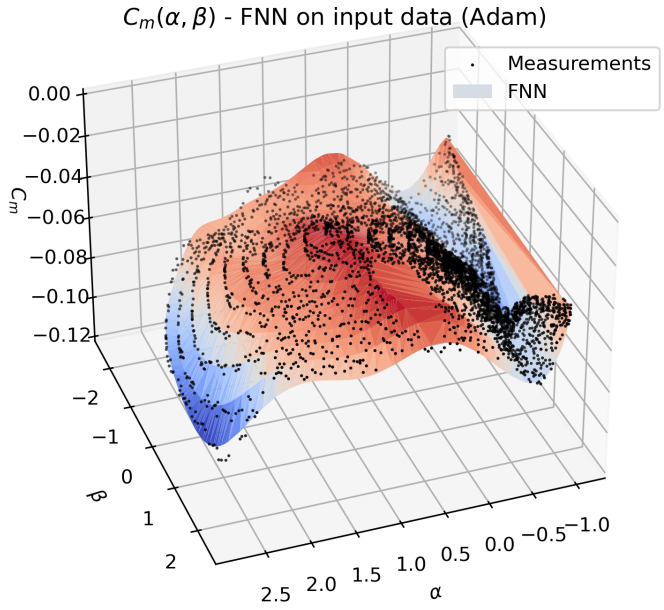

Figure 3.2: $C_m(\alpha, \beta)$ *model estimated using an FNN trained with an Adam optimizer for* ~ 10000 *epochs, decaying* $\eta : 0.01 \rightarrow 0.0001$*, with ten hidden neurons.*

By looking at the effect of learning rate on the network convergence, it can be seen why having a decaying learning rate is advantageous. Figure 3.3 shows the same network trained with different learning rates, and how that affects the loss curve. At low values of η (0.0005-0.005), convergence requires a lot of training epochs, though the MSE is kept stable as it slowly decreases. As the learning rate increases, the initial drop in MSE becomes steeper, though at the cost of stability as the training epochs increase (seen in spiking of the loss curves). Eventually, η becomes too high, and the network becomes so unstable that it cannot converge, as it wildly oscillates between high and low MSE due to the large changes to the model parameters (for $\eta = 0.5$). Therefore to capture the speed of a high learning rate and the stability of a low one, a decaying learning rate (going from $\eta = 0.01 \rightarrow 0.0001$) was chosen, to achieve a faster and more accurate convergence.

[2] PyTorch Adam optimizer, last accessed 13/05/2024.

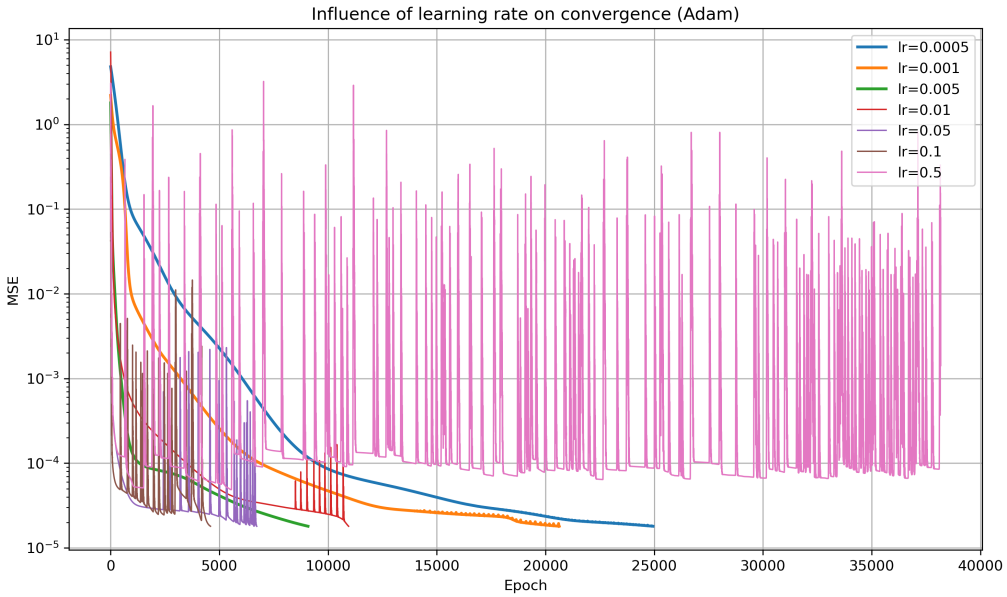

Figure 3.3: *Influence of learning rate on the MSE curves (i.e. network convergence) for the same network structure (ten hidden neurons).*

Neural network parameter estimation problems can sometimes suffer from issues of reproducibility when it comes to the initial conditions of the network. Each weight and bias is initially randomly sampled, and thus the initial conditions can change from run to run. Figure 3.4 shows the MSE curves for ten runs of the FNN for the same network structure. As can be seen from Figure 3.4, the network will output a slightly different result depend on the initial configuration, though always with the same order of magnitude of error, so the network can be deemed somewhat insensitive to changes in initial conditions. What does vary between runs is the convergence time, as some runs can take much longer to reach an acceptable MSE, though due to the simplicity of the neural network, all runs still have a relatively fast wall-clock time.

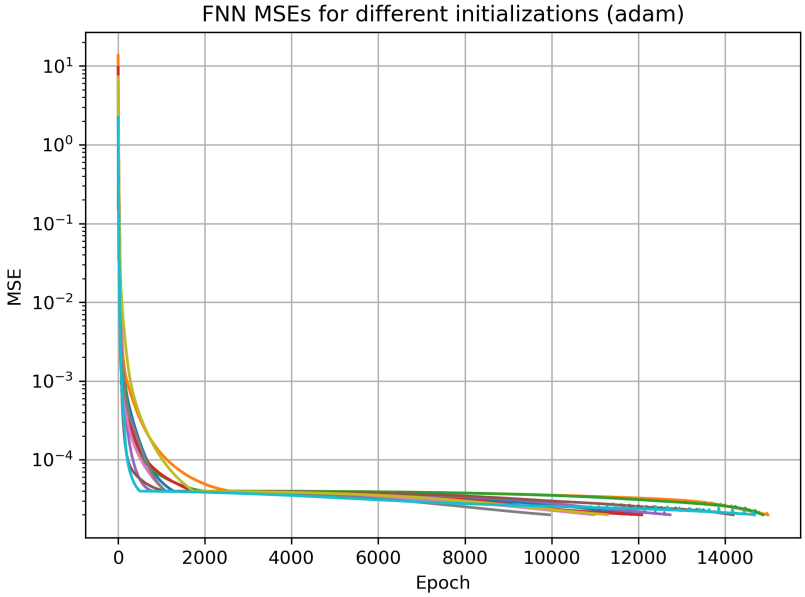

Figure 3.4: *MSEs of different FNN runs with different initial conditions, but same network configuration (five hidden neurons and $\eta = 0.01$).*

The last hyperparameter that can be modified for this FNN is the number of neurons. As the input and output neurons are fixed, the total number of neurons can be altered by changing the size of the hidden layer. By having a larger hidden layer, the network should be able to learn more complex

relationships between input and output, thus reducing the MSE/increasing the approximation accuracy. However, just having a very large number of hidden neurons will lead to overfitting of data, leading to poorer performance (similarly to OLS case, where too high of an order increases the validation MSE). Thus, a balance needs to be reached between having high training and validation accuracy. The best model structure can be found by minimizing both the MSE of the validation/test data and the MSE of the special validation data, to check how the model performs with data that does not overlap with the main dataset. This result can be seen in Figure 3.5, where the best number of hidden neurons is 12.

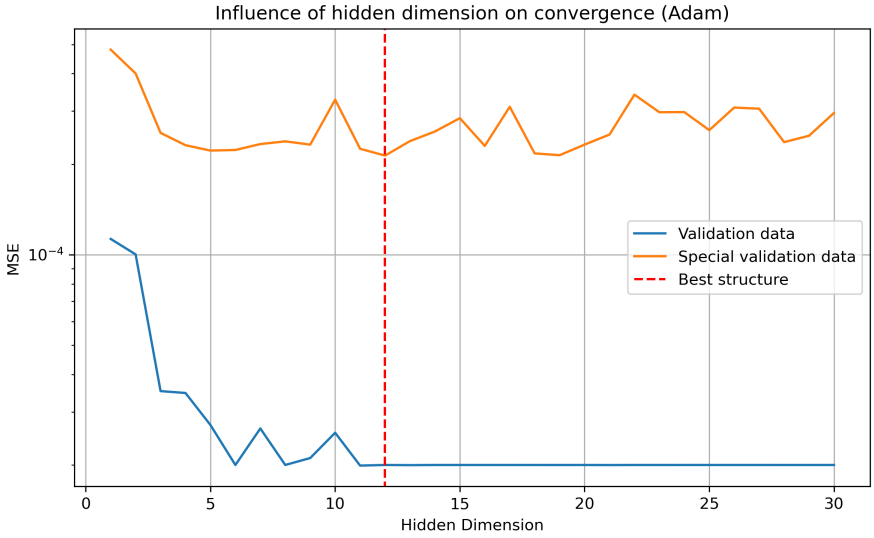

Figure 3.5: *Optimizing FNN in terms of number of neurons. Best structure with 12 hidden neurons.*

3.2. Levenberg-Marquardt learning algorithm

As previously seen in Chapter 2, another useful learning algorithm is the LMA, which can also be applied to feed-forward networks. For this algorithm, the error vector is defined as the square residuals (Equation 3.5) and the weight vector has as its components the trainable parameters of each layer (i.e. the weight matrix and bias vector for each layer), seen in Equation 3.6. The Jacobian of the error w.r.t. the weights is also defined in Equation 3.7 by considering the chain rule of the partial derivatives. The same update law (Equation 3.8) is then applied to the respective weights to train the model.

$$E = \frac{1}{2}(y - \hat{y})^2 \tag{3.5}$$

$$W = \begin{bmatrix} w_1 & b_1 & w_2 & b_2 \end{bmatrix}^T \tag{3.6}$$

$$\begin{aligned} J = \frac{\partial E}{\partial W} &= \begin{bmatrix} \frac{\partial E}{\partial w_1} & \frac{\partial E}{\partial b_1} & \frac{\partial E}{\partial w_2} & \frac{\partial E}{\partial b_2} \end{bmatrix} \\ \frac{\partial E}{\partial W_1} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \phi} \cdot \frac{\partial \phi}{\partial \nu} \cdot \frac{\partial \nu}{\partial w_1} = -(y - \hat{y}) \cdot w_2 \cdot (1 - h^2) \cdot x \\ \frac{\partial E}{\partial b_1} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \phi} \cdot \frac{\partial \phi}{\partial \nu} \cdot \frac{\partial \nu}{\partial b_1} = -(y - \hat{y}) \cdot w_2 \cdot (1 - h^2) \cdot 1 \\ \frac{\partial E}{\partial w_2} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2} = -(y - \hat{y}) \cdot h \\ \frac{\partial E}{\partial b_2} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_2} = -(y - \hat{y}) \cdot 1 \end{aligned} \tag{3.7}$$

$$W_{t+1} = W_t - (J^T J + \mu I)^{-1} J^T E \tag{3.8}$$

As with the RBF network, special care has to be given to updating the damping factor in the LMA. This was also done according to Algorithm 1. The FNN was then trained, an example of the results can be found in Figure 3.6

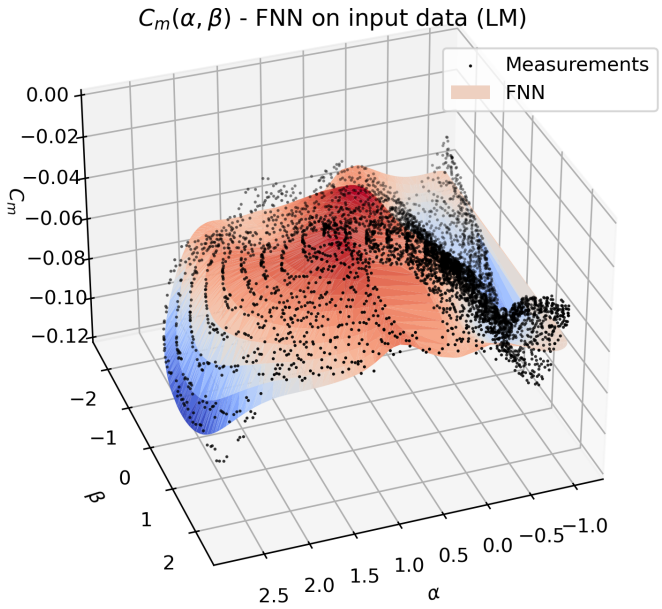

Figure 3.6: $C_m(\alpha, \beta)$ *model estimated using an FNN trained with the LMA for* ~ 80 *epochs, with 30 hidden neurons.*

Comparing the FNN results of the backpropagation (Adam) and Levenberg-Marquardt algorithms, these differ considerably in terms of approximation accuracy, required number of iterations and sensitivity to initial conditions. Figure 3.7 shows the MSE loss curves of the two methods, and it is immediately apparent that the LMA converges considerably faster than Adam. This comes from the automatic tuning of the damping factor in LMA as opposed to the slower decay of the learning rate in Adam. With regards to the approximation accuracy, Adam yields around half the MSE of LMA, though both these values are appropriately low (1.8×10^{-5} and 2.9×10^{-5} respectively). It must also be noted that to achieve this MSE, the LMA had to be trained with many more hidden neurons than Adam, which could achieve the lower MSE with around 12 neurons already.

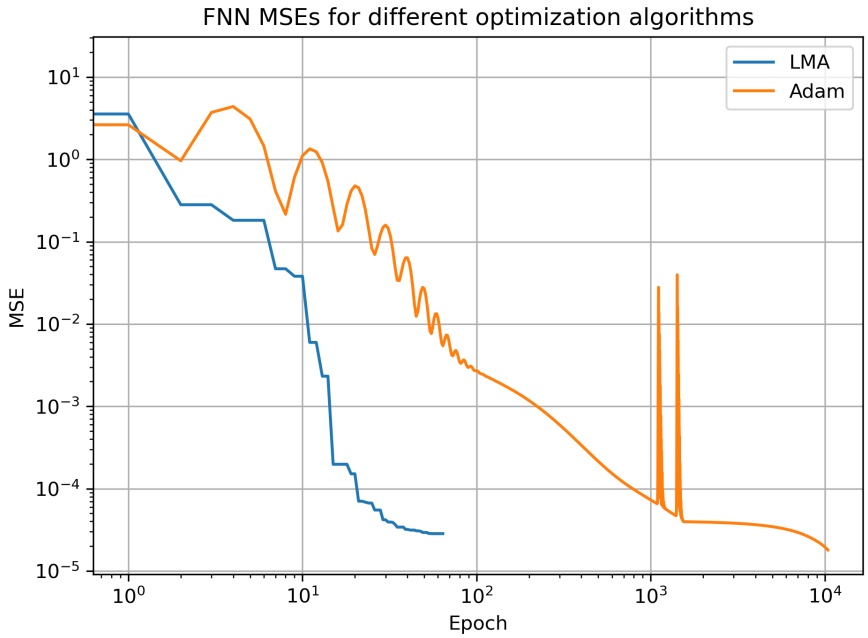

Figure 3.7: *Comparison between MSE loss curves of FNN trained with two different learning algorithms (Adam and LMA), both with the same network structure (30 hidden neurons).*

In terms of sensitivity to initial conditions, Figure 3.4 showed that Adam had a relative insensitivity to

initial conditions. On the contrary, the LMA showed a very high sensitivity to initial conditions, with around only less than half of the runs actually converge to an acceptable MSE, as seen in Figure 3.8.

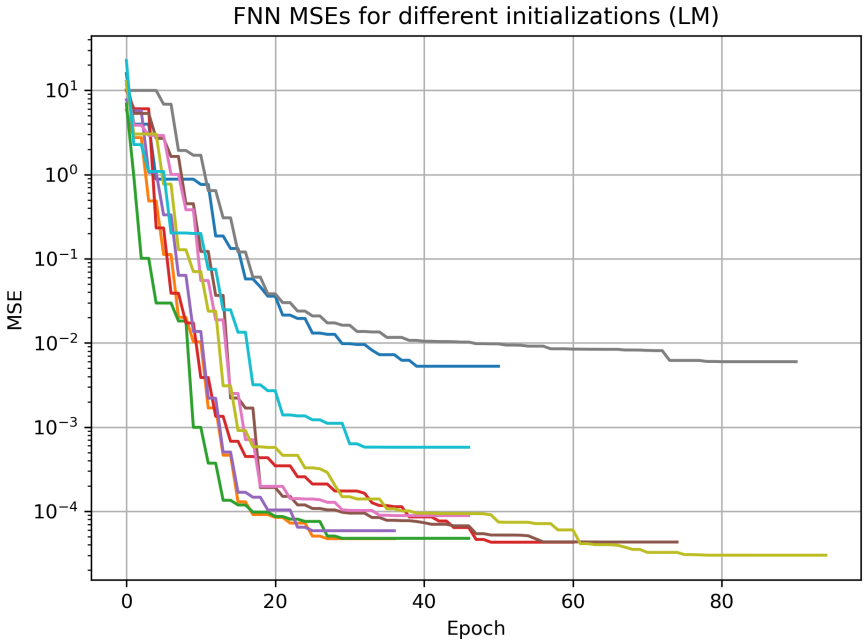

Figure 3.8: *MSEs of FNN runs with different initial conditions, but same network configuration (30 hidden neurons).*

3.3. Comparisons with other approaches

Given that two neural network architectures have been proposed to model the relationship between C_m, α and β, these two should be compared in terms of their approximation accuracy and required number of iterations to reach that accuracy. To better compare the two networks, the same number of parameters should be used for each network. This is, however, quite challenging, since the number of trainable parameters per network is different, even if they have the same structure (i.e. number of hidden neurons). Furthermore, the FNN and RBF approaches has distinct optimal structures, wherein the optimal number of hidden neurons for the RBFNet is over three times that of the FNN.

Nonetheless, if the number of hidden neurons is set to 12 (optimal for the FNN), the FNN outweighs the accuracy of the RBFNet by an order of magnitude (1.99×10^{-5} compared to 2.12×10^{-4}). If instead 42 hidden neurons are used (optimal for the RBFNet), the FNN still outperforms the RBFNet, though by a smaller amount (1.97×10^{-5} compared to 5.24×10^{-5}). In terms of the required number of iterations, for both configurations, the RBFNet requires considerably fewer iterations, around 70-180, while the FNN required 8852 iterations. It must be noted that even through careful configuration of the RBFNet hyperparameters, this network structure was not able to achieve the same approximation accuracy as the FNN, demonstrating the power of FNNs and gradient-descent learning algorithms as universal function approximators. The better accuracy of the FNN over the RBFNet can also be seen when the network is trained on the special validation dataset (the result of which is seen in **??**), yielding a lower RMS of 0.0154, compared to 0.0165 of the RBFNet.

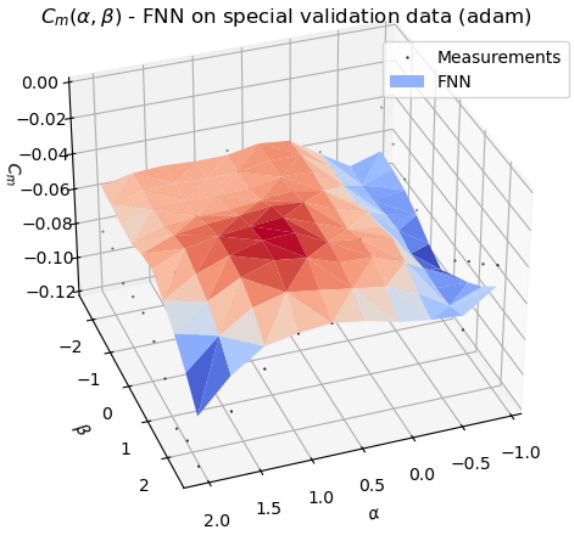

Figure 3.9: $C_m(\alpha, \beta)$ *model estimated using the FNN (Adam) with 12 hidden neurons, using the special validation dataset.*

These results can also be compared to the initial parameter estimation method that was presented, the OLS method. Comparing the accuracies, the optimal OLS model (of order 6) yields an even better test accuracy than the FNN model, with an MSE of 1.5×10^{-5}, and requiring a single iteration to converge. The power of this method over the FNNs may be attributed to the relative simplicity of the network structure used, as a deep neural network (i.e. multiple hidden layers), might have been able to better capture the details of the $C_m(\alpha, \beta)$ model (though at the computational cost of having many more parameters to be trained).

4

Conclusion

In sum, a series of parameter estimation methods were presented and applied to an F16 flight dataset, in order to model the relationship between the angle of attack and sideslip (α ad β) and the measured moment coefficient (C_m). Three main approaches used (each with different training methods), namely ordinary least squares (OLS), radial basis function neural networks (RBFNet) and feed-forward neural networks (FNN). The numerical results from the modelling can be seen in Table 4.1, while the strengths and weaknesses of each method are detailed in Table 4.2.

Table 4.1: *Comparison of main numerical results from modelling with the different parameter estimation approaches.*

Metric	**Modelling approach**				
	OLS	RBFNet (lin. reg.)	RBFNet (LMA)	FNN (Adam)	FNN (LMA)
Order/Hidden neurons	6	20	42	12	12
Parameters	135,268	20	295	61	61
Validation MSE	1.5×10^{-5}	2.41×10^{-5}	5.24×10^{-5}	1.99×10^{-5}	3.1×10^{-5}
Iterations	1	1	182	8852	76
Special validation RSS	0.0155	-	0.0165	0.0154	-

Table 4.2: *Summary of strengths and weaknesses of each parameter estimation approach to modelling.*

Approach	**Strengths**	**Weaknesses**
OLS	Very fast computation. High test and validation accuracy.	Requires a linearity assumption between input and output variables (satisfied in this case). For a best linear unbiased estimator, requires assumption of residuals to be normally distributed and homoscedastic (satisfied in this case).
RBFNet (lin. reg.)	Very fast computation. Good test and validation accuracy.	Essentially a differently posed OLS problem, hence does not leverage the power of RBF networks.
RBFNet (LMA)	Relatively small number of iterations. Better capability of modelling nonlinear relations.	Due to large sensitivity to initial conditions, training oftentimes will not converge to an acceptable loss value, hence computation must be repeated. Relatively poor validation accuracy.

FNN (Adam)	Good validation and test accuracy even when model is underparametrized compared to the size of input. Low sensitivity to initial conditions. Better capability of modelling nonlinear relations.	Requires considerably more iterations to converge. Can have very unstable convergence if learning rate is not properly tuned.
FNN (LMA)	Few iterations required for convergence. Better capability of modelling nonlinear relations.	Due to large sensitivity to initial conditions, training oftentimes will not converge to an acceptable loss value, hence computation must be repeated. Relatively poor validation accuracy.