System Identification of Aerospace Vehicles
Neural Networks Assignment

R.J. Bouwmeester
Delft University of Technology, 2629 HS, the Netherlands
https://github.com/gemenerik/sysid

September 29, 2020

1 Introduction

System identification provides the tools to create mathematical descriptions of systems from measurements. In the field of aircraft control & simulation, in particular, these mathematical descriptions (e.g., state space systems) are widely used. Implementation might be obvious, such as in flight simulators. But the models can also be used to control statically unstable aircraft, damaged aircraft. Really, these models stand at the core of control & simulation.

All techniques discussed in the field can be broadly summarized to the following; give both the real and mathematical system a known input and try to match your model's behavior to that of the real system. The required models might be highly non-linear, quasi-linear, or linearized (for a matrix-form state space system, for example). Measurements might be very noisy, so that must be taken into account. Unless one seeks to model noise!

In this report, neural networks will be used for system identification. Neural networks are a very powerful and accurate tool that can approximate many behaviors and functions, given that the right architecture and parameters are used, and the network is trained well. Each neuron is fed by the input and/or other neurons, to this they apply their basis functions, which has parameters such as a weight and bias. At the output of the network, the result is compared with the desired output using a cost function. Depending on the magnitude of the discrepancy, the previously mentioned parameters of the neurons are updated. The way these are updated depends on the chosen optimization method.

Truly, these methods provide a black box approach to the problem. This allows us to approximate the behavior of the F16 without knowing any of the underlying physical properties. In many cases it is hard to verify the results of a neural network. However, with the F16 being such a common aircraft, in this case, the results should be verifiable. Sadly, our data set is limited to 100 seconds of behavior. Obviously, our model will only be valid for the conditions during the recording of this data.

In this report, system identification methods using neural networks are applied on F16 training data. First, inaccuracies and noise are removed from the training data using a Kalman Filtering technique in section 2. In the same section, a parameter estimation is done to find a missing value. Next, in section 3 a polynomial regression fit of the data is made. In section 4 the general layout of the neural networks are introduced. In section 5 and 6 this architecture is deployed using Radial Basis Functions (RBF), and sigmoidal Feed Forward (FF) architectures, respectively. Both are tested with gradient descent and Levenberg-Marquardt optimizers, and for varying network parameters.

2 Kalman Filtering

In this section the data to be used in the Neural Networks is processed. The measured data is highly contaminated by noise, and the measured angle of attack α_m is known to be biased.

2.1 System Equations

Define state vector x_k, and input vector u_k as in equation 1 and 2, respectively. $C_{\alpha_{up}}$ is a constant, so its derivative is 0. Clearly, the system equation $\dot{x_k} = f(x_k, u_k, t)$ is trivial and follows directly from the input vector u_k. The resulting system equation is shown in equation 3.

$$x_k = \begin{bmatrix} u & v & w & C_{\alpha_{up}} \end{bmatrix}^T \tag{1}$$

$$u_k = \begin{bmatrix} \dot{u} & \dot{v} & \dot{w} \end{bmatrix} \tag{2}$$

$$\dot{x_k} = \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \\ \dot{C_{\alpha_{up}}} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} x_k + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} u_k \tag{3}$$

To find the system equation $z_k = h(x_k, u_k, t)$, combine equations 4 through 10, to arrive at equation 11.

$$z_k = \begin{bmatrix} \alpha_m & \beta_m & V_m \end{bmatrix}^T \tag{4}$$

$$\alpha_m = \alpha_{\text{true}} \left(1 + C_{\alpha_{\text{up}}}\right) + v_\alpha \tag{5}$$

$$\beta_m = \beta_{\text{true}} + v_\beta \tag{6}$$

$$V_m = V_{\text{true}} + v_V \tag{7}$$

$$\alpha_{true} = \tan^{-1}\left(\frac{w}{u}\right) \tag{8}$$

$$\beta_{\text{true}} = \tan^{-1}\left(\frac{v}{\sqrt{u^2 + w^2}}\right) \tag{9}$$

$$V_{\text{true}} = \sqrt{u^2 + v^2 + w^2} \tag{10}$$

$$z_k = \begin{bmatrix} \left(1 + C_{\alpha_{up}}\right) \tan^{-1}\left(\frac{w}{u}\right) \\ \tan^{-1}\left(\frac{v}{\sqrt{u^2+w^2}}\right) \\ \sqrt{u^2 + v^2 + w^2} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} u_k + \begin{bmatrix} v_{alpha} \\ v_{beta} \\ v_V \end{bmatrix} \tag{11}$$

2.2 Kalman filter selection

As the aircraft dynamics are non-linear, a regular Kalman filter cannot be used. Thus, an extended Kalman filter is used. These are not optimal, but by using an iterated extended Kalman filter we can reduce the linearization errors at the cost of increased computational requirements. A non-iterated version does not guarantee sufficient convergence speed or might not even converge at all. The iterated version of the extended Kalman filter introduces an iterative part within

2.3 Iterated Extended Kalman Filter

We construct the Kalman filter following the rules as given in 'Lecture 3: State Estimation'.

1. Make a one-step ahead prediction using the Runge-Kutta fourth order method.
2. Calculate the Jacobians of the state transition (F) and observation equations (H). F is a matrix of zeroes, as the system is trivial. H is calculated with equation 12
3. The state transition and input matrices are discretized using a continuous to discrete function inspired by Matlab's c2d function, resulting in the discrete state transition ($\Phi_{k+1,k}\left(\underline{x}^*(t), \underline{u}^*(t), t\right)$) and input matrices ($\Gamma_{k+1,k}\left(\underline{x}^*(t), \underline{u}^*(t), t\right)$), respectively.
4. Calculate the covariance matrix of the state prediction error using equation 13.

   Where $(\bullet) = (\underline{x}^*(t), \underline{u}^*(t), t)$
5. Iterative part
   (a) Recalculate measurement equation Jacobian
   (b) Recalculate the Kalman gain using equation 14.
   (c) Do a measurement & state estimate update, equations 15, 16
   (d) Repeat till improvement $< \epsilon$
6. Calculate the covariance matrix of the state estimation error
7. Repeat for every data point.

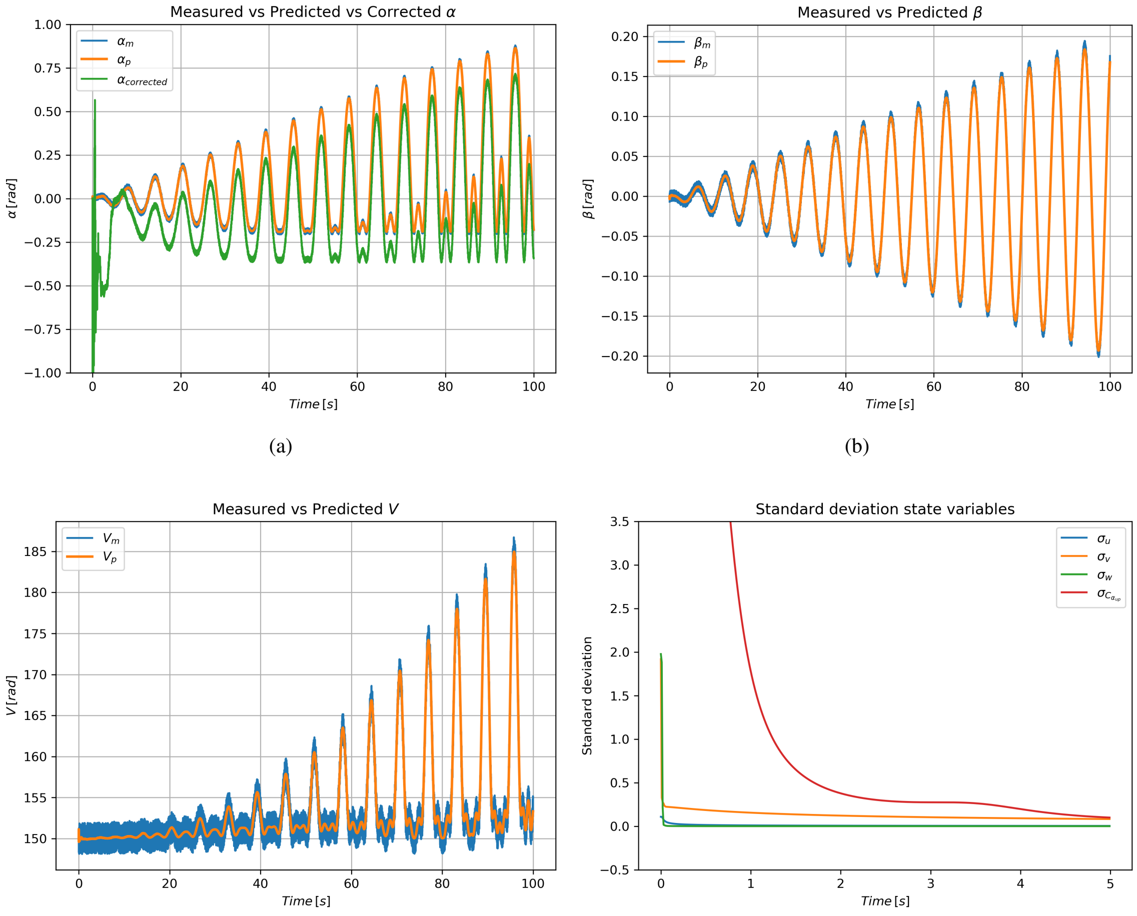
$$H_x(\underline{x}^*(t), \underline{u}^*(t), t) = \frac{\delta}{\delta x} h(\underline{x}(t), \underline{u}(t), t) \tag{12}$$

$$P_{k+1,k}(\bullet) = \Phi_{k+1,k}(\bullet) P_{k,k} \Phi^T_{k+1,k}(\bullet) + \Gamma_{k+1,k}(\bullet) Q_{d,k}(\bullet) \Gamma^T_{k+1,k}(\bullet) \tag{13}$$

$$K_{k+1} = P_{k+1,k} H^T_{k+1} \left(H_{k+1} P_{k+1,k} H^T_{k+1} + R_{k+1}\right)^{-1} \tag{14}$$

$$\eta_{i+1} = \hat{\underline{x}}_{k+1,k} + K_{k+1}\left(\underline{\eta}_i\right)\left(\underline{z}_{k+1} - \underline{h}\left(\underline{\eta}_i, \underline{u}_{k+1}\right) - H_x\left(\underline{\eta}_i\right)\left(\hat{\underline{x}}_{k+1,k} - \underline{\eta}_i\right)\right) \tag{15}$$

$$\hat{\underline{x}}_{k+1,k+1} = \underline{\eta}_l \tag{16}$$



(a)

(b)

(c)

(d)

Figure 1: Measured, predicted and corrected α, β, V, and standard deviations of state variables

The filter is used to estimate $C_{\alpha_{up}}$ at 0.1650959. With this, the true α (without bias, $\alpha_{corrected}$), is reconstructed using equation 17. The result is visualized in figure 1a. Figures 1b and 1c show the measured and predicted (using the Kalman filter) α, V. Figure 1d shows to standard deviation of the state variables, all converging to zero. This proves that the filter has converged.

$$\alpha_{corrected} = \alpha_m - C_{\alpha_{up}} \tag{17}$$

3 Polynomial Regression Fit

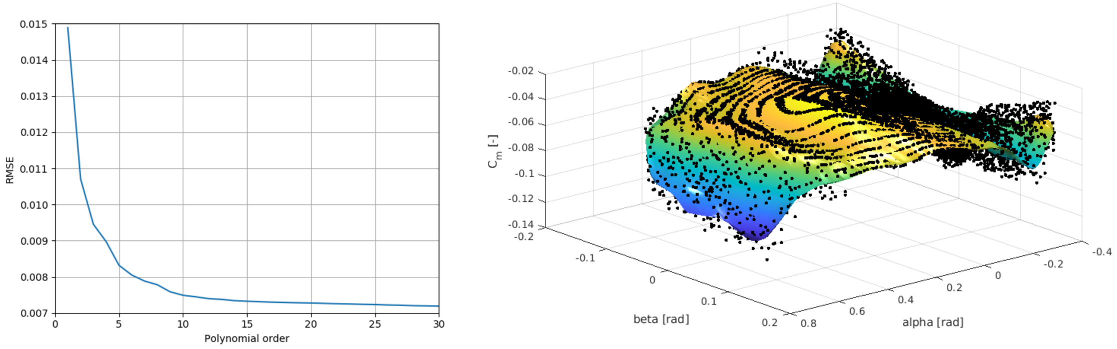
A polynomial regression fit can be made. The polynomial model can be found in equation 18. m is the degree of polynomial used. The sum of squared residuals is minimized, see equation 19. In figure 2, a polynomial fit of order 15 is shown on the $alpha$, $beta$, C_m-3D plot.

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m + \varepsilon_i (i = 1, 2, \ldots, n) \tag{18}$$

$$\min \sum_{i=1}^{n} \epsilon_i^2 \tag{19}$$

3.1 Influence of polynomial model order

To assess the influence of the polynomial order the RMSE of the polynomial fit is plotted for orders 1 through 30, see figure 2a. The RMSE does not decrease as much after a polynomial order of approximately 15, so that is used. In figure 2b, the polynomial regression fit is shown.



(a) RMSE for varying polynomial order

(b) Polynomial fit of order 15. $RMSE = 0.007325$

Figure 2

4 Neural Network Layout

In this section, the general layout of the neural networks used in this paper is introduced. In Figure 3, this layout is visualized. The input and output activation functions, ϕ_i and ϕ_k, are linear activation functions for all networks in this paper. In all networks, a root mean-squared error is used as loss function, as presented in Equation 20, where the hat indicates the output of the network, i is the current evaluated data-point, and N is the total number of data-points. $\alpha_{corrected}$, β_p, and C_m result from section 2. 80% of data is used for training, 20% for testing. Test data is used to avoid overfitting. The data is normalized.

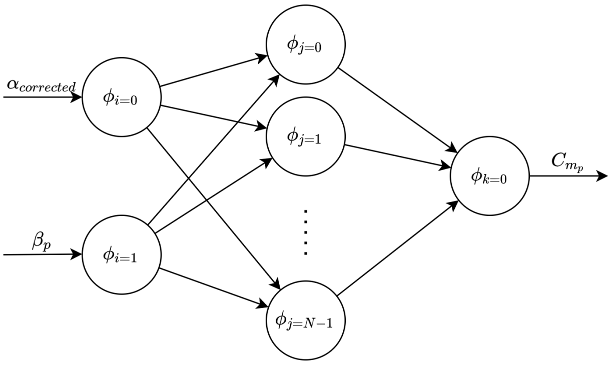


Figure 3: General neural network layout used in this paper

$$\lambda = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(C_{m_i} - C_{m_{p_i}})^2} \tag{20}$$

The network output is evaluated using equations 21, 22, and 23, where the values of i, j, k, indicate the selected input-, hidden-, and output node, respectively. From here on, weights w_{ij} are referred to as input weights, and weights w_{jk} as output weights. These are randomly initialized with a fixed seed.

This general layout can be equipped with varying activation functions for the hidden layer, the number of neurons in the hidden layer can be varied, the batch size can be varied, and different optimization algorithms can be applied. The setup does not allow to increase the number of hidden layers. A network with a single hidden layer of sufficient (sigmoidal) artificial neurons can approximate any function [1]. This does not imply that it is as easy to learn with a single hidden layer, as with a multi-layer network. However, for ease of implementation and modularity, it was decided to stick with just one layer. Learning is capped at a maximum of 250 epochs. The goal of this report, after all, is to explore the effects of varying parameter settings, optimizers, etc, and not to train state of the art networks. In the rest of the report, various adaptions of this layout are introduced, evaluated and compared.

$$\hat{C_m} = \phi_k(v_k) = v_k \tag{21}$$

$$v_k = \sum_j w_{jk} y_j = \sum_j w_{jk}\phi_j(v_j) \tag{22}$$

$$v_j = \sum_i w_{ij} y_i = \sum_i w_{ij}\phi_i(v_i) = \sum_i w_{ij} x_i \tag{23}$$

$$x = \{\alpha_{corrected}, \beta_p\} \tag{24}$$

5 Radial Basis Function Network

In this section a Radial Basis Function (RBF) network is implemented and evaluated. First, the properties of the network are introduced in section 5.1. Next, in section 5.2, a linear regression approach is applied to the network. In section 5.3, a Levenberg-Marquardt optimizer is implemented, and the effects of varying network parameters are evaluated.

5.1 RBF network properties

This network is equipped with radial basis functions as hidden layer activation functions.

$$\phi_j\left(v_j\right) = a \cdot \exp\left(-v_j\right); \quad v_j = \sum w_{ij}^2 \left(x_i - c_{ij}\right)^2 \tag{25}$$

The network is optimized, either by gradient descent or a Levenberg-Marquardt (LM) algoritm. Both require the gradient of the loss function with respect to the to be updated weights (equations 30, 36).

$$E = \frac{1}{2}\sum_k (d_k - y_k)^2 \tag{26}$$

$$\frac{\partial E}{\partial y_k} = \frac{\partial E}{\partial e_k}\frac{\partial e_k}{\partial y_k} = e_k \cdot -1 \tag{27}$$

$$y_k = \phi_k(v_k) \tag{28}$$

$$\frac{\partial y_k}{\partial v_k} = \frac{\partial \phi_k(v_k)}{\partial v_k} = 1 \tag{29}$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial w_{jk}} = e_k \cdot -1 \frac{\partial \phi_k(v_k)}{\partial v_k} y_j = e_k \cdot -1 y_j \tag{30}$$

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial y_j} \tag{31}$$

$$\frac{\partial v_k}{\partial y_j} = w_{jk} \tag{32}$$

$$\frac{\partial y_j}{\partial v_j} = \frac{\partial \phi_j(v_j)}{\partial v_j} \tag{33}$$

$$\frac{\partial \phi_j(v_j)}{\partial v_j} = -a \cdot exp(-v_j) \tag{34}$$

$$\frac{\partial v_j}{\partial w_{ij}} = y_i \tag{35}$$

$$\frac{\partial E}{\partial w_{ij}} = \sum_k \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial y_j}\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ij}} \tag{36}$$

Gradient descent

The following equations show how weights are updated for the gradient descent method (a back-propagation optimizer, the term back-propagation itself does not imply gradient descent [2]). η is the learning rate.

$$w_{t+1} = w_t + \Delta w \tag{37}$$

$$\Delta w = -\eta \frac{\partial E}{\partial w_t} \tag{38}$$

Levenberg-Marquadt

The following equations show how weights are updated for the Levenberg-Marquadt method. μ is the damping parameter.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \left(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \mu\mathbf{I}\right)^{-1} \mathbf{J}^{\mathrm{T}}\mathbf{e} \tag{39}$$

$$\mathbf{e} = \left[\begin{array}{cccc} e(1) & e(1) & \cdots & e(N) \end{array} \right]^T \tag{40}$$

$$\mathbf{J} = \frac{\partial \mathbf{e}}{\partial \mathbf{w}_t} = \left[\begin{array}{cccccc} \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{ij}(1)} & \cdots & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{ij}(K)} & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{jk}(1)} & \cdots & \frac{\partial \mathbf{e}(1)}{\partial \mathbf{w}_{fk}(M)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{ij}(1)} & \cdots & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{ij}(K)} & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{jk}(1)} & \cdots & \frac{\partial \mathbf{e}(N)}{\partial \mathbf{w}_{jk}(M)} \end{array} \right] \tag{41}$$

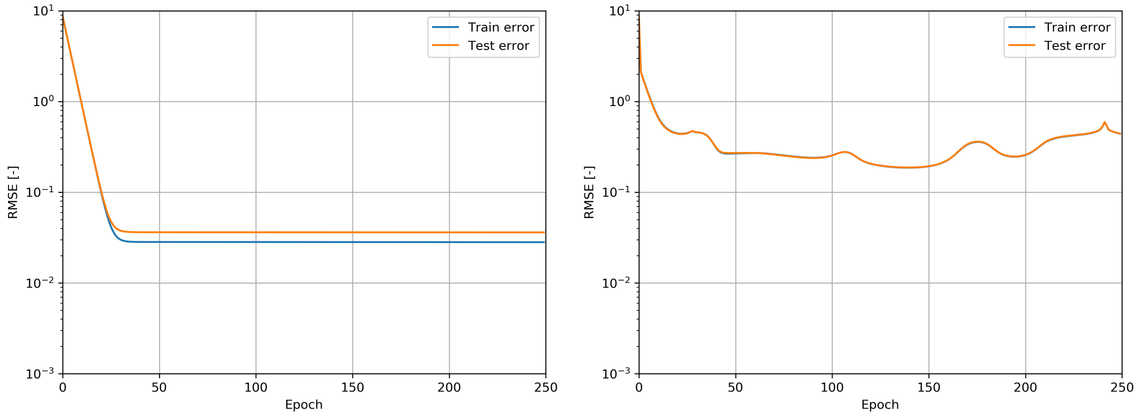
5.2 Linear regression approach

To emulate a regular linear regression, the batch size is set to match the length of the data. This way, all data is evaluated at the same time. Gradient descent is used to update the weights, with $\eta = 0.001$. 10 hidden neurons are used.

Figure 4a shows the resulting training graph. Table 1 shows information about the epoch with the lowest test error. Lowest test error is chosen in stead of lowest training error, as this gives a better indication of real application performance, and in case of overfitting would the lowest training error and test error not correspond. Compared to the 'regular' polynnomial regression fit of section 3, this method has a higher RMSE. Note however, that zero tweaking has been done, so far.

Table 1: Network results for minimum test error, for regression on RBF network, gradient descent

Test error	0.036087
Train error	0.028150
Epoch	250



(a) Gradient descent

(b) Levenberg-Marquardt

Figure 4: Regression using RBF network

5.3 Levenberg-Marquardt approach

To compare learning algorithms, the same network is trained using a Levenberg-Marquardt optimizer. For proper comparison, all other parameters are kept constant. The damping parameter μ is set at 0.01.

Figure 4b shows the resulting learning curve. The optimizer shows very fast initial convergence, but fails to converge as far as the gradient descent method does. In fact, in the current state, the optimizer turns out to be quite unstable. Clearly, the parameters of the LM learning algorithm can use some tweaking. In the following sections, a sensitivity analysis of various parameters is presented.

Table 2: Network results for minimum test error, for regression on RBF network, Levenberg-Marquardt optimization

Test error	0.187314
Train error	0.186290
Epoch	140

Damping parameter (μ) sensitivity

To assess the effects of the damping parameters (μ), the algorithm is run for a range of values for μ. As $\mu = 0.01$ already showed unstability, the network is only tested for that and larger damping parameters. The learning curves (showing only test error, to avoid clutter, no overfitting occurred, training graphs follow closely) can be seen in figure 5. $\mu = 0.1$ quickly dips into a minima, but fails to remain close to it, so a larger damping is required. $\mu = 1$ performs best out of all damping parameters. $\mu = 10$ and $\mu = 100$ are too large for sufficiently fast convergence. Note that 'the optimal' damping depends on the setting of all other network parameters. For instance, with an increased maximum number of epochs a larger damping might result in a lower error. However, from here on we will continue with $\mu = 1$.

The lower the damping, the faster the learning. On the other hand, the lower the damping, the more instability. The damping factor has the inverse effects of learning rate in a gradient descent method.

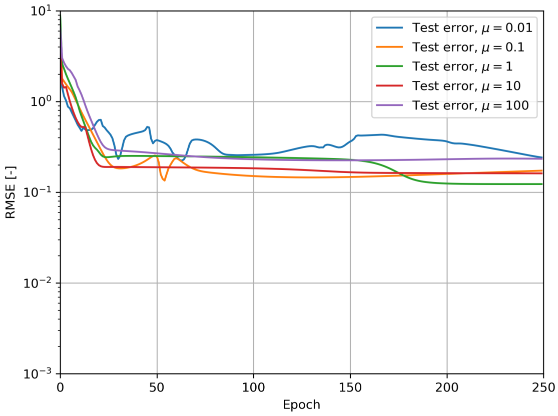


Figure 5: Test error for μ, regression using RBF network, LM

Table 3: Network results for varying μ, for minimum test error, for linear regression on RBF network, LM

	$\mu = 0.01$	$\mu = 0.1$	$\mu = 1$	$\mu = 10$	$\mu = 100$
Test error	0.2231	0.1344	**0.1229**	0.1614	0.225
Train error	0.2209	0.1389	0.1236	0.1592	0.2261
Epoch	64	55	235	250	147

Weight initialization sensitivity

Random initialization
To assess the effects of weight initialization, training is done for two additional random (weights between 0 and 1) initializations. Figure 6a shows the learning curve of the resulting three random weight initializations. The second random initialization fails to converge as far as the others. The third random initialization converges relatively slowly early on, but then finds the same minimum as the first initialization early on, and even converges a bit further. Table 4 shows, for each initialization, the epoch with the lowest test error.

Table 4: Network results for **random** initializations, for minimum test error, for linear regression on RBF network, LM

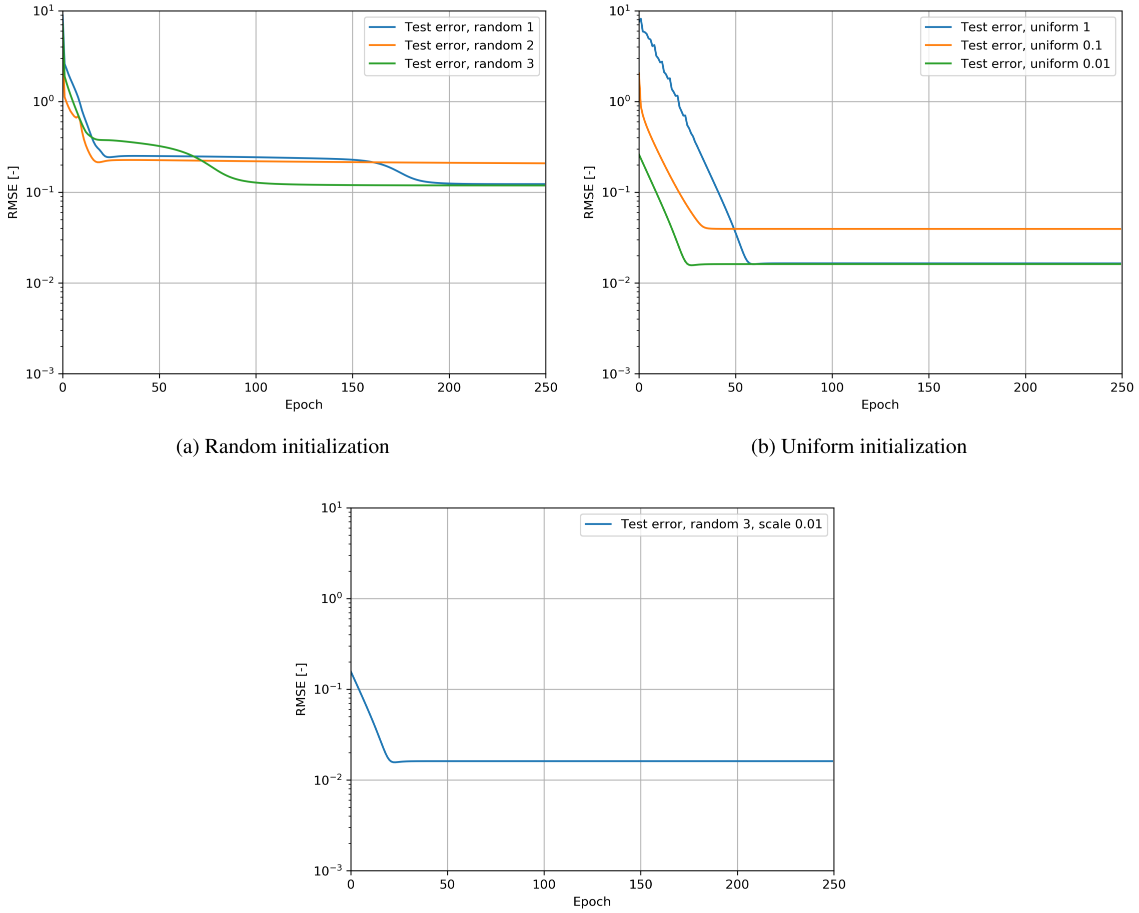
	Initialization 1	Initialization 2	Initialization 3
Test error	0.1229	0.2083	**0.1187**
Train error	0.1236	0.2093	0.1179
Epoch	235	250	250

Uniform initialization
The RBF network can also be initialized uniformly. To assess, the network weights are initialized with all 1s, 0.1s and 0.01s. Figure 6b shows the learning curve of the uniform initializations. All uniform initializations perform better than random initializations. Curiously, uniform 1 converges further than uniform 0.1. Uniform 0.01 converges furthest, and fastest. Smaller values were tested but did not improve results. The results suggest that smaller starting weights are advantageous, which might not be surprising considering the small values of C_m (between -0.14 and -0.02. The best test epochs per initialization are shown in table 5.

With these findings, we train the random initialization 3 again, but with all weights scaled by 0.01. The results can be found in figure 6c, and are close to identical to the uniform 0.01 initialization, which we will use from here on.

Table 5: Network results for **random** initializations, for minimum test error, for linear regression on RBF network, LM

	Uniform 1	Uniform 0.1	Uniform 0.01
Test error	0.0161	0.0394	**0.0157**
Train error	0.0177	0.0406	0.0165
Epoch	141	250	25



(a) Random initialization

(b) Uniform initialization

(c) Random initialization, scaled

Figure 6: Linear regression using RBF network, LM

Number of hidden neuron optimization

Where logically the more neurons, the more accurate the approximation, such a large network has a hard time converging to the same degree as the smaller variants. This is because all tweaked parameters are interdependent. The effects on the output of updating one of two hidden neuron versus one of many hidden neurons, is much greater. For example, since we use a fixed damping of $\mu = 1$, the 10 hidden neuron network performs well, as we have tweaked the initialization of the network for it in the previous section. True optimization would require a multivariate optimization. In this section, however, we focus on an optimization with all parameters fixed, except for the number of hidden neurons.

Below the algorithm that optimizes the number of hidden neurons is shown in pseudocode, where n is the number of hidden neurons, i is the current iteration, e_0 is the previous iteration's error, e_1 is the current error, r is change rate (at which the number of hidden neurons changes), and C is a Boolean that defines whether last iteration the number of hidden neurons was increased (True) or decreased (False).

The algorithm is a variation on gradient descent (though the learning rate or change rate (r) is not proportional to the gradient between errors), with a learning rate schedule. The algorithm could be improved by adding further measures to prevent getting stuck in local minima.

Algorithm 1: Number of hidden neurons optimization

```
Input: $i = 0$,
$e_0 = 1e12$,
$n = 20$,
$C =$ False
$r = 5$
Output: Error log, training graph
while $i < 20$ do
    if $i > 5$ then
        $r = 2$
    if $i > 10$ then
        $r = 1$
    if $e_1$ already calculated then
        take result from error log
    else
        calculate error $e_1$
        store in error log
    if $C$ then
        if $e_1 < e_0$ then
            $n+ = r$, $C =$ True
        else
            $n- = r$, $C =$ False
    else
        if $e_1 < e_0$ then
            $n- = r$, $C =$ False
        else
            $n+ = r$, $C =$ True
    $i+ = 1$
```

Figure 7 shows the results for one such optimization, initialized as in the pseudocode. Results for 10 or fewer hidden neurons are virtually identical (note the shorter $RMSE$ axis compared to previous sections), with the best result being 7 nodes with $RMSE = 0.015648$. Next, 4 nodes with $RMSE = 0.015653$. Logically, with the same learning rate, the smaller networks converge faster. As expected, the number of hidden neurons the parameters were selected for performs well. Surprisingly, the network performs almost identical with fewer, even just 2, nodes. Clearly, there is room for improvement, as the network does not seem to take advantage of the 8 additional nodes.

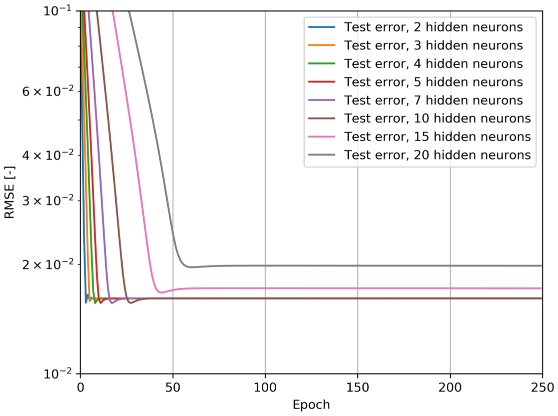


Figure 7: Number of hidden neuron optimalization

6 Feed forward

Now the network is fitted with sigmoid activation functions (equation 42). As in the previous chapter, we start by using gradient descent, and will vary multiple parameters to assess their effects. Finally, we will compare the sigmoid-based network to the RBF-network.

$$\phi_j\left(v_j\right) = \frac{2}{1+\exp\left(-2v_j\right)} - 1 \tag{42}$$

$$\begin{aligned} \frac{\partial \phi_j(v_j)}{\partial v_j} &= \frac{\partial}{\partial v_j}\left(2 \cdot (1+\exp(-2v_j))^{-1} - 1\right) \\ &= 4 \cdot \frac{\exp(-2v_j)}{1+\exp(-2v_j)} \frac{1}{1+\exp(-2v_j)} \\ = 4 \cdot & \frac{1}{1+exp(-2v_j)}\left(1 - \frac{1}{1+exp(-2v_j)}\right) \end{aligned} \tag{43}$$

Learning rate

$\eta = 10$ is unstable, and results in overflows in the activation function. $\eta = 1$ converges fastest, and no other learning rate manages to converge as far as this. However, the learning rate is too large to further converge than it manages at epoch 29.

A variable learning rate, to get a quick early drop and then further converge, could improve results. We could use momentum methods, or more advanced algorithms like Adam, but simplest to implement, and still quite effective is a custom set learning rate schedule. We emperically tweak the algorithm. We start with $\eta = 1$, reduce it by 90% at the 30th epoch, and again halve it at the 50th epoch. This brings down our maximum training error to 0.0601, however, we do get some overfitting and the test error goes back up to 0.0620.

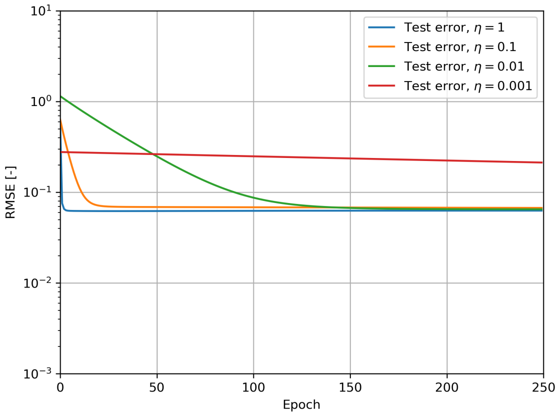


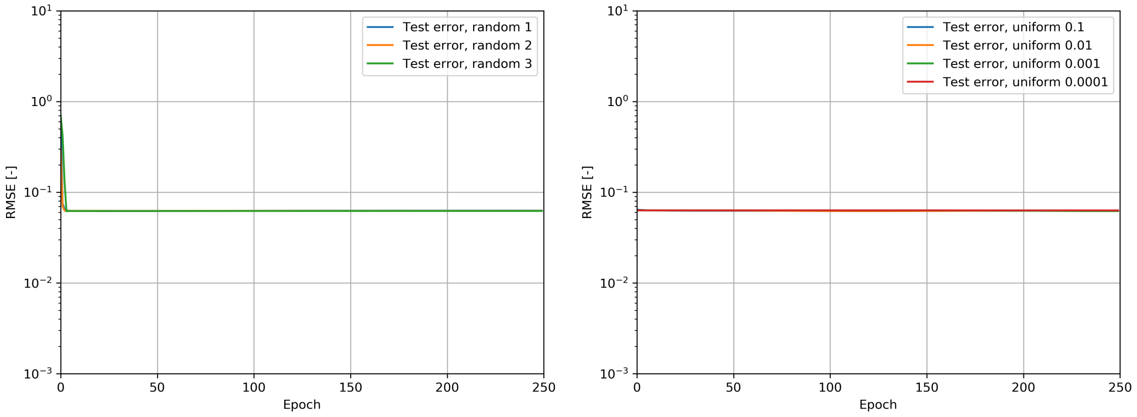
Figure 8: Test error for varying η, regression using FF network, gradient descent

Table 6

	$\eta = 1$	$\eta = 0.1$	$\eta = 0.01$	$\eta = 0.001$
Test error	**0.0619**	0.0672	0.0648	0.2124
Train error	0.0603	0.0667	0.0622	0.2036
Epoch	29	250	250	250

Weight initialization sensitivity

Contrary to the RBF-network, the initialization only significantly affects the convergence speed. It seems that we are getting stuck in a (local) minimum, despite the initialization.



(a) Test error for three random initializations

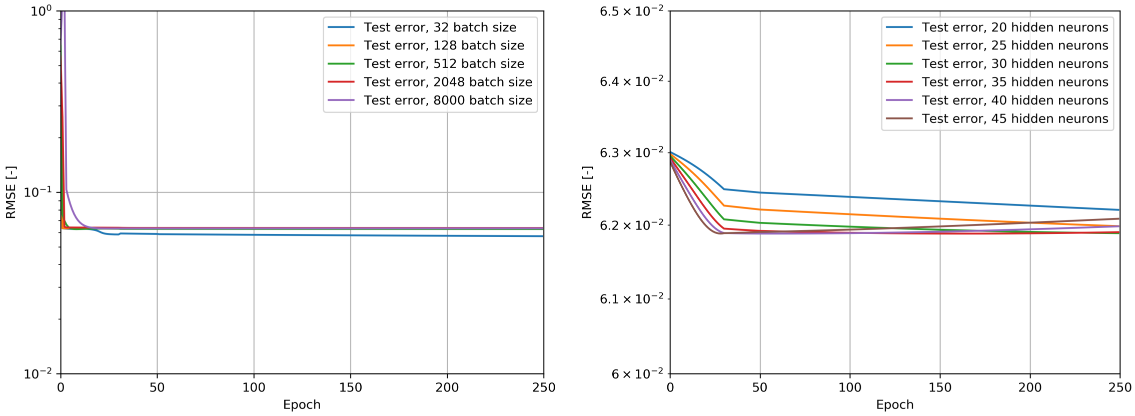
(b) Test error for uniform initializations

Figure 9

Batch size sensitivity

So far, the batch size was set to equal the data length, to simulate a normal regression problem. Theoretically, a larger minibatch size can result in a better estimate of the gradient. However, it can lead to overfitting. Very small batch sizes have an opposite effect. Additionally, several performance considerations should be made. Larger batch sizes require more memory [2] to be used. This is, however, not really a problem with our limited size network and data set. Power of 2 batches generally perform better, so all tested batch sizes are chosen as such, except for the batch size of the data length (8000). Though overfitting has not been a serious problem yet, with test error being very similar to train error, it is given a try.

In figure 10, one can see that a small batch size of 32 provides the best results. The improvement is not because of regularization, results for all batch sizes show approximately equal overfit. Smaller batch sizes might require lower learning rates to prevent instability because of the high variance in gradients [3], so I hypothesize that the seen improvement is simply because of a learning rate that is set too low (remember that the learning rate was picked empirically from a set of learning rates, each varying with a factor of 10; a learning rate of factor 10 higher caused instability, but that doesn't say it can't still be set significantly higher than 1) for the higher batch sizes. After testing, we identified stable behavior for a learning rate of $\eta = 3$ (that is 3 times the learning rate used before), with a convergence to $RMSE = 0.060755$ at the 16th epoch.



(a) Test error for varying batch size

(b) Number of hidden neuron optimalization

Figure 10

Number of hidden neuron optimization

The same algorithm as presented in section 5.3 is used. Differences are minimal, note the y-axis scale. Counterintuitively, the wider networks converge faster. The best result is achieved by the 36 node network with $RMSE = 0.061883$ at the 164th epoch. However, networks with 35 or more nodes start diverging halfway. Likely, in a few more epochs, the 30 node network would have improved on the 35's result, as it has a $RMSE = 0.061889$ at the 250th epoch.

6.1 Levenberg-Marquardt

Figure 11 shows a comparison between the training using gradient descent and Levenberg-Marquardt. For proper comparison, the same number of hidden nodes are used. The Levenberg-Marquardt optimizer converges slightly faster, but not as far as the gradient descent method.

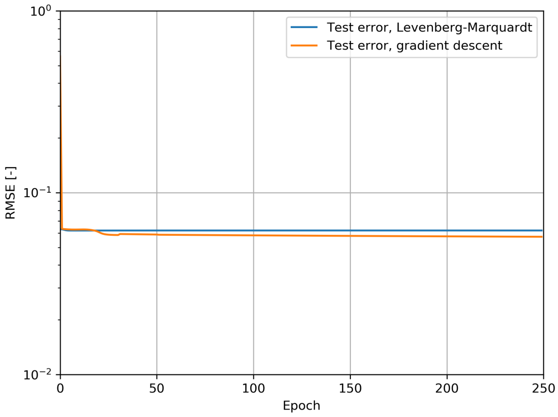


Figure 11: Gradient descent - Levenberg-Marquardt comparison

7 Conclusion

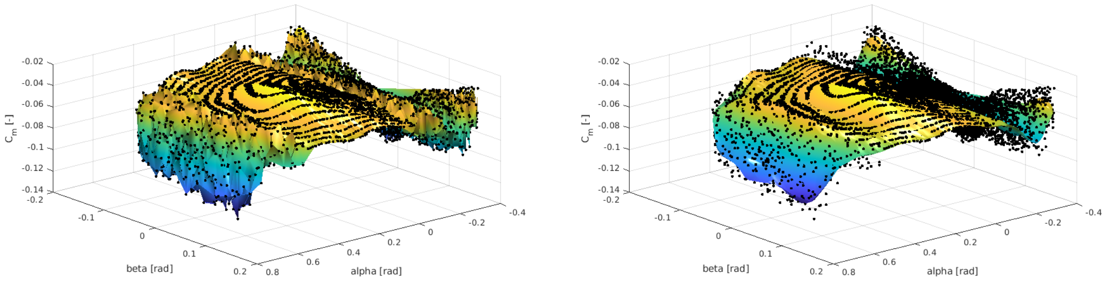
Optimization of parameters was highly interdependent, and results depend very much on the parameter initializations. As such, with limited time and computing power, very strong conclusions cannot to be drawn from the results. For a better comparison, all parameters need to be optimized simultaneously.

All in all, we had more success with the Levenberg-Marquardt optimizer, but in several cases the optimizers performed quite similarly.

In theory, both methods should be able to find the same minima, given the right initialization and settings. However, the Levenberg-Marquardt method, although in its basic form much more complex, is a lot easier to tweak to get to those minima. Without a variable learning rate, the gradient descent method easily overshoot the minima (too large learning rate) or get stuck in local minima (too small learning rate). According to [4], the Levenberg-Marquardt optimizer behaves much like a gradient descent method initially, until it is close to its optimal values, where it starts behaving more like Gauss-Newton.

Significantly better results were obtained using the RBF network. Again, results highly depend on initialization, application and parameter settings [5]. Both should be able to approximate the function accurately enough given enough nodes, but the RBF network is more easily trained with the current network setup. Looking at the shape of both activation functions, the RBF network has to place and scale the RBFs (which has a local effect), whereas the FF network is closer to the polynomial model, where each sigmoid affects a wide range of values. This implies that while the RBF network is more easy to train, the sigmoidal network generalizes better. In terms of run-times, both networks performed similarly, with the RBF network taking slightly more time each epoch.

Neither network got significantly close to the results of the polynomial model.



(a) Raw data with 3D triangular surface

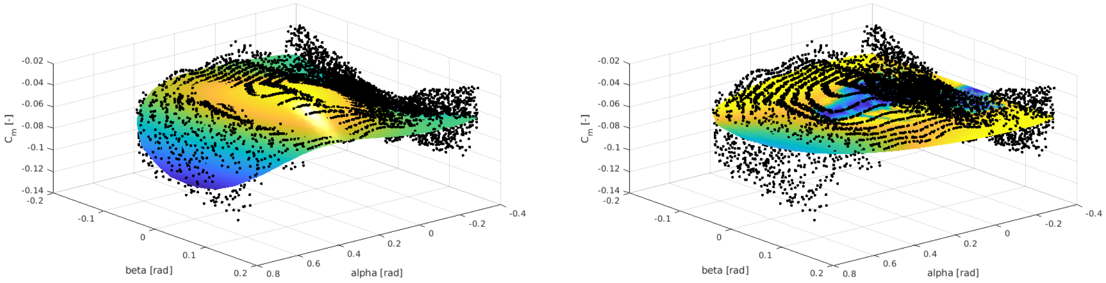
(b) Polynomial fit of order 15. $RMSE = 0.007325$

Figure 12

Recommendations

Several significantly influential parameters have not been looked at, yet. Potentially, a smaller test split could be used. Also, a random selection with constant seed was used to create this split; the effect of picking a different random test set should be evaluated.

For a fair comparison between neural networks and polynomial fitting one would have to fit both using the same data. In this case, the polynomial uses all data, whereas 20% of the data for the neural networks is



(a) Best RBF fit. $RMSE = 0.015648$

(b) Best FF fit. $RMSE = 0.060755$

Figure 13

sacrificed to detect potential overfitting. Due to the nature of polynomials (at least with relatively low order) there is less risk of overfitting.

Improving the performance of the network will allow to do significantly more optimization in the same amount of time. Training 250 epochs of the hand-written network takes in the order of 10 minutes on the used machine, whereas training the same network in TensorFlow takes in the order of 1 minute (both on CPU).

After running all experiments, TensorFlow was used to verify the results, which were close to identical for similar settings. Applying more advanced optimizers such as Adam quickly resulted in better results than obtained in the hand-made networks or than the polynomial fit, indicating that these networks could significantly improve with more tweaking.

References

[1] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

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

[3] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[4] Henri P Gavin. The levenberg-marquardt algorithm for nonlinear least squares curve-fitting problems. *Department of Civil and Environmental Engineering, Duke University http://people. duke. edu/˜ hpgavin/ce281/lm. pdf*, pages 1–19, 2019.

[5] Gary Russell and Laurene V Fausett. Comparison of function approximation with sigmoid and radial basis function networks. In *Applications and Science of Artificial Neural Networks II*, volume 2760, pages 61–72. International Society for Optics and Photonics, 1996.