Methodologies for Tracking of Load Extremes and Error Estimation using Probabilistic Techniques

R. Flores
E. Ortega
R. López

Publication CIMNE Nº-375, April 2012

Methodologies for Tracking of Load Extremes and Error Estimation using Probabilistic Techniques

R. Flores
E. Ortega
R. López

Publication CIMNE Nº-375, April 2012

International Center for Numerical Methods in Engineering
Gran Capitán s/n, 08034 Barcelona, Spain

Summary

This work, conducted at CIMNE under ALEF project task 1.2.3, presents an investigation about the potential capabilities of neural networks to assist simulation campaigns. The discrete gust response of an aircraft has been chosen as a typical problem in which the determination of the critical loads requires exploring a large parameter space.

A very simple model has been used to compute the aerodynamic loads. This allows creating a large database while at the same time retaining some of the fundamental properties of the problem. Using this comprehensive dataset the effects of network structure, training method and sampling strategy on the level of approximation over the complete domain have been investigated. The capabilities of the neural network to predict the peak load as well as the critical values of the design parameters have also been assessed. The applicability of neural networks to the combination of multi-fidelity results is also explored.

Table of contents

List of figures

List of tables

Nomenclature

Δt	Time increment
μ	Mean value
σ	Standard deviation
U_∞	Flight speed (true air speed)
U_G	Gust velocity
ADM	Aerodynamic Data Module
ALEF	Aerodynamic Loads Estimation at Extremes of the Flight Envelope
BFGS	Broyden-Fletcher-Goldfarb-Shano method
c	Airfoil chord
CG	Conjugate Gradient method
CIMNE	International Center for Numerical Methods in Engineering
CPU	Central Processing Unit
dof	Degrees of freedom
EAS	Equivalent Air Speed
h	Altitude
H	Gust gradient distance
I_G	Moment of inertia per unit span
L-BFGS	Limited memory BFGS method
m	Mass per unit span
MAC	Mean Aerodynamic Chord
MLW	Maximum Landing Weight
MSE	Mean Squared Error
MTOW	Maximum Takeoff Weight
MZFW	Maximum Zero-Fuel Weight
RMS	Root Mean Square

R_y	Radius of gyration
SM	Static Margin (fraction of chord)
x_G	Center of mass longitudinal position

1 Introduction

This report explores the applicability of neural networks as useful tools for building a detailed representation of the aerodynamic loads over the complete flight envelope, including the determination of the critical load cases. When the number of free parameters in the problem increases the so called “curse of dimensionality” comes into play (Bishop 1995). This means that it is not possible to obtain a comprehensive sampling of the parameter space because the number of points required grows geometrically with the dimensionality of the problem. Neural networks are known for their ability of generalization, that is, the capability to extract trends from a collection of sample points (so much in fact that they are used for such abstract task as shape recognition, for example). From a statistical point of view, it can be demonstrated that under certain conditions a neural network delivers the best representation of an unknown probability density function which is possible for a given number of degrees of freedom (Bishop 1995). In this sense, neural networks are optimum approximators and therefore very good interpolators. They are especially suited for cases where there are large gaps in the sample set. It seems then reasonable that neural networks can be useful in the process of building a database of aerodynamic loads, where the number of points that can be computed is limited due to time and cost constraints. Moreover, from the computational point of view the cost of training a neural network is very small; it can be used in combination with existing tools without increasing the cost of the data generation process. Thus, the networks could be a useful tool for building aerodynamic data modules (ADM’s)

The ability of neural networks to predict the extreme load cases from a small sample set is the main focus of this report. The document starts with a summary description of the characteristics of neural network and their basic working principles (section 2). Next a simple 2D model is introduced which will be the basis for exploring the capabilities of the method (section 4). The effects of network architecture, training algorithm and sampling strategy on the accuracy of the approximation are assessed starting with a very simple model of little practical interest but which has the advantage of allowing a graphical rendering of the response surface (section 5). Once some basic understanding of the approximator has been achieved the report moves to a higher dimensionality case (sections 6) where the capabilities of the neural network become more important due to the impossibility of visualizing the system response. Section 7 tests the ability of the networks as data fusion tool, combining multi-fidelity results into a single coherent approximation. Next, section 8 applies the results

of section 6 to higher fidelity model, to verify the general applicability of the method. Finally, the main conclusions drawn from the investigation are summarized in section 9.

2 Neural networks

A neural network is a biologically inspired computational model which consists of a network architecture composed of artificial neurons (Haykin 1994). These are information processing structures whose most significant property is their ability to learn how to perform certain tasks. A neural network can be subject to different learning paradigms, depending on how the knowledge available for training is represented. They include supervised, unsupervised and reinforcement learning (Haykin 1994).

The multilayer perceptron is an important kind of neural network, and much of the literature in the field is referred to that model (Bishop 1995). This type of neural network has found a wide range of applications, with data modelling (usually refered to as function regression in the specialized literature) being one of the most popular ones (Bishop 1995).

The function regression problem can be regarded as the problem of approximating a function from an input-target data set (Bishop 1995). The targets are a specification of what the output response to the inputs should be. From a statistical point of view, the basic goal in a function regression problem is to model the conditional distribution of the output variables, conditioned on the input variables (Bishop 1995). This function is called the regression function.

Other tasks for which neural networks have proven themselves valuable tools are pattern recognition (classification) and time series prediction (forecasting). While of great practical importance these applications fall outside the scope of this document and won’t be discussed in the following.

2.1 Multilayer perceptron

In this section we summarize the basic theory behind the multilayer perceptron. This type of neural network is based on four building blocks:

- A neuron model (the perceptron)
- A feed-forward network architecture
- The objective functional
- Training algorithm

2.1.1 The perceptron neuron model

A neuron is the basic information processing unit in a neural network. The perceptron is the characteristic neuron model in the multilayer perceptron (Sima & Orponen 2003). It computes a net input as a function of the input signals and some free parameters. The net input is then passed through an activation function to produce an output signal (Sima & Orponen 2003). Figure 2 shows a perceptron neuron model.

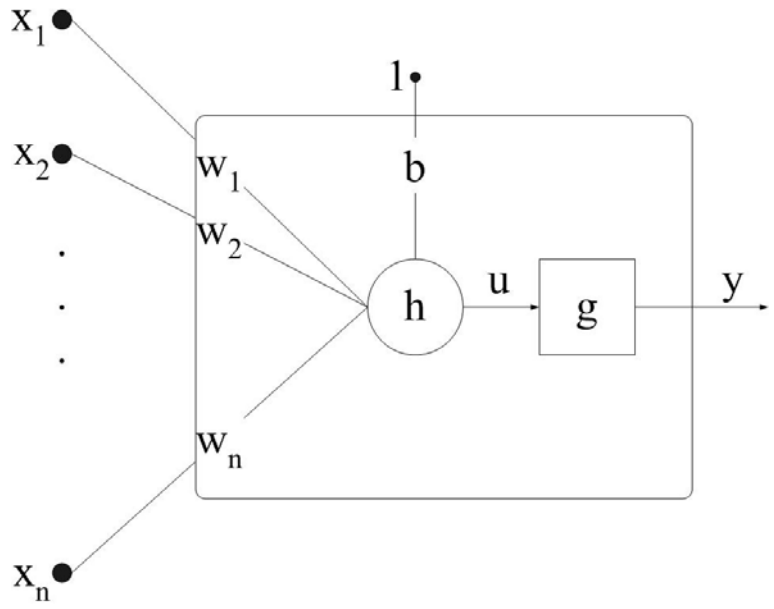

Fig. 1 – A perceptron neuron model

Mathematically, a perceptron neuron model spans a parameterized function space **V** which transforms an element of the input space $\mathbf{x} \in \mathbb{R}^n$ into a scalar output $\mathbf{y} \in \mathbb{R}$ (Lopez & Oñate 2006). The function space **V** is parameterized by the free parameters of the neuron $(b, \mathbf{w})$ and therefore the dimension n+1. The elements of the vector **w** are called the weights and the scalar b is known as the neuron bias. The weights are used to average the inputs while the bias mission is to offset the average value. The elements of the function space spanned by a perceptron are of the form:

$$y : \mathbb{R}^n \to \mathbb{R} \mid \mathbf{x} \to y(\mathbf{x}, b, \mathbf{w}) \tag{1}$$

where

$$y = g\left(b + \sum_{i=1}^{n} w_i x_i \right) \tag{2}$$

Some of the most commonly used activation functions are the logistic sigmoid $g(u) = \frac{1}{1+\exp(-u)}$, the hyperbolic tangent $g(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{-u}}$ and the linear function $g(u) = u$ (Bishop 1995). The logistic sigmoid function can approximate as extreme cases both a linear function and a Heaviside step thus delivering great versatility. It also allows for the probabilistic interpretation of the network outputs (Bishop 1995). The properties of the hyperbolic tangent are very similar to the logistic sigmoid but this activation function has been empirically shown to improve convergence of the training algorithms in many cases.

2.1.2 The multilayer perceptron network architecture

Although a single perceptron can perform certain simple tasks, the power of neural computation comes from connecting many neurons in a network architecture. Neurons. The architecture of a neural network is determined by the number of neurons, their arrangement and connectivity (Sima & Orponen 2003). The characteristic network architecture of the multilayer perceptron is the so called feed-forward architecture.

A feed-forward architecture typically consists on an input layer of sensorial nodes, one or more hidden layers of neurons, and an output layer of neurons (Fig. 2). Communication proceeds layer by layer from the input layer via the hidden layers up to the output layer (Sima & Orponen 2003). In this way, a multilayer perceptron is a feed-forward network architecture of perceptron neuron models.

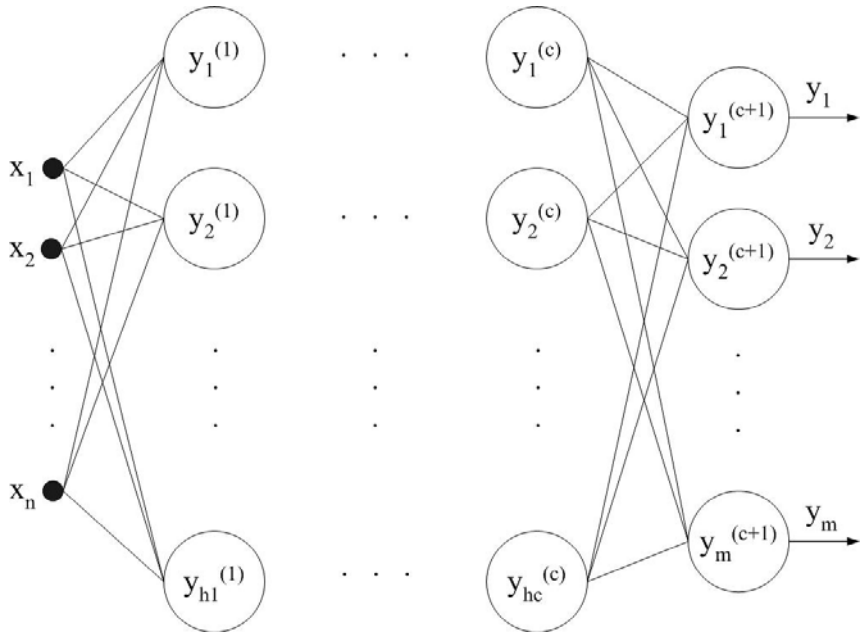

Fig. 2 - Multilayer perceptron network architecture

In a similar way as it happens with a single perceptron, a multilayer perceptron spans a parameterized function space **V** from an input $\mathbf{x} \in \mathbb{R}^n$ to an output $\mathbf{y} \in \mathbb{R}^m$ (Lopez & Oñate 2006). Elements of **V** are parameterized by the free parameters in the network, which can be grouped together in a s-dimensional free parameter vector $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_s)$. Note that the vector includes both the weights on the connections as well as the biases of all the neurons. In fact, it is common practice to consider the bias values as additional weights and include an extra virtual neuron at each layer with the output fixed to one. This is advantageous from the point of view of the programmer because it allows for a unified treatment of all the free parameters. In this line the vector $\boldsymbol{\alpha}$ may be referred to as is "vector of network weights" when no specific distinction is necessary. The dimension of the function space **V** is therefore s. The elements of the function space spanned by the multilayer perceptron in Fig. 2 are of the form:

$$\mathbf{y} : \mathbb{R}^n \to \mathbb{R}^m \mid \mathbf{x} \to \mathbf{y}(\mathbf{x}, \boldsymbol{\alpha}) \tag{3}$$

where the state of any neuron can be evaluated from the outputs of the neurons in the previous layer:

$$y_j^{(i)} = g^{(i)}\left(b_j^{(i)} + \sum_{k=1}^{h_{i-1}} w_{kj}^{(i-1)} y_k^{(i-1)} \right) \qquad \text{for } i = 1, 2, ..., c+1 \tag{4}$$

In the expression above h_{i-1} denotes the number of neurons in layer $i-1$ and $w_{kj}^{(i-1)}$ is the weight connecting the output of neuron k of layer $n-1$ to neuron j of layer i. Note that (4) assumes that the inputs are treated as the first layer of the network $\left(y_j^{(0)} = x_j\right)$. Using expression (4) the state of the network can be advanced layer-by-layer from the input all the way to the output layer.

A multilayer perceptron with as few as one hidden layer of sigmoid neurons and an output layer of linear neurons provides a general framework for approximating any function from one finite dimensional space to another up to any desired degree of accuracy, provided sufficiently many hidden neurons are available. In this sense, multilayer perceptron networks are a class of universal approximators (Hornik et al. 1989).

2.1.3 The objective functional

The objective functional defines the task that the network is required to accomplish and provides a measure of the quality of the representation that it is required to learn. In this way,

the choice of a suitable objective functional depends on the particular application (Lopez & Oñate 2006). An objective functional for the multilayer perceptron is of the form

$$F : V \to \mathbb{R} \,|\, \mathbf{y}(\mathbf{x},\boldsymbol{\alpha}) \to F\left[\mathbf{y}(\mathbf{x},\boldsymbol{\alpha})\right] \tag{5}$$

The learning problem for the multilayer perceptron can then be formulated in terms of the minimization of an objective functional of the function space spanned by the neural network (Lopez & Oñate 2006).

One of the most common objective functionals used in function regression is the sum-of-squares error, which is measured on an input-target data set (Bishop 1995). It is written as a sum, over all the samples in the data set, of a squared error calculated for each sample separately:

$$E\left[\mathbf{y}(\mathbf{x},\boldsymbol{\alpha})\right] = \frac{1}{2}\sum_{q=1}^{Q}\left(\sum_{k=1}^{m}\left(y_k\left(\mathbf{x}^{(q)},\boldsymbol{\alpha}\right) - t_k^{(q)}\right)^2\right) \tag{6}$$

where $\mathbf{t}^{(q)}$ is the q-th member of the target sample set while $\mathbf{x}^{(q)}$ is the corresponding input sample. Q denotes the target sample size and the $\frac{1}{2}$ factor is included in order to make the gradient of the error functional directly proportional to the factor $y_k\left(\mathbf{x}^{(q)},\boldsymbol{\alpha}\right) - t_k^{(q)}$. Closely related to (6) and widely used too is the mean squared error (MSE), which is simply the sum of the squared errors normalized with the number of samples in the data set in order to remove the dependency with the set size:

$$MSE\left[\mathbf{y}(\mathbf{x},\boldsymbol{\alpha})\right] = \frac{1}{Q}\sum_{q=1}^{Q}\left(\sum_{k=1}^{m}\left(y_k\left(\mathbf{x}^{(q)},\boldsymbol{\alpha}\right) - t_k^{(q)}\right)^2\right) \tag{7}$$

From the MSE functional and considering a fixed training-target set, an error function can be defined whose argument is the vector of free parameters in the network

$$f : \mathbb{R}^s \to \mathbb{R} \,|\, \boldsymbol{\alpha} \to f(\boldsymbol{\alpha}) \tag{8}$$

The minimum of the objective functional is achieved when a vector $\boldsymbol{\alpha}$ is found for which the objective function f reaches a minimum value. Therefore, the learning problem for the multilayer perceptron, formulated as a variational problem, can be reduced to a function optimization problem (Lopez & Oñate 2006).

2.1.4 The training algorithm

The training algorithm is in charge of solving the function optimization problem by adjusting the free parameters in the network so as to minimize the objective function. More specifically,

the training algorithm searches in a s-dimensional space for a parameter vector $\boldsymbol{\alpha}^*$ at which the objective function takes a minimum value. A minimum can be either a global minimum, the smallest value of the function over its entire range, or a local minimum, the smallest value of the function within some local neighborhood. Finding a global minimum is, in general, a very difficult problem (Wolpert & McReady 1997) so algorithms in common use generally yield only local extrema.

Training algorithms might require information from the value of the objective function only, its gradient (first derivatives) vector or its Hessian (second derivatives) matrix (Press et al. 2002). These methods, in turn, can perform either global or local optimization.

Zero-order training algorithms evaluate the objective function only (no derivatives are needed). Usually these methods call for a larger number of iterations than gradient-based schemes. They are especially useful when the derivatives of the error function are difficult to compute (or if the function is not even differentiable).

First-order training algorithms use the objective function and its gradient vector. Examples of these are gradient descent, conjugate gradient and quasi-Newton methods (Press et al 2002). They are all local optimization methods.

Second-order training algorithms make use of the objective function, its gradient vector and its Hessian matrix. An example is the Newton's method (Press et al 2002), which is a local optimization method. In spite of its fast rate of convergence, the complexity associated with the computation of the Hessian often outweighs the benefits of the method.

3 Software implementation

For this activity the PUMI -NEURAL library of Neural Network procedures has been developed in FORTRAN 95 language. This choice of language allows for seamless integration into CIMNE's existing analysis software while also delivering increased performance compared with existing alternatives (e.g. Demuth & Beale 2002).

The code allows experimenting with general multilayer perceptrons with an arbritary number of layers, different types of activation functions (which can be mixed in any combination across the different layers) as well as error measures.

Two methods to calculate the gradients of the error function with respect to the network parameters are provided:

- Finite differences. A very general method but with limited efficiency.

- Back propagation. Computationally efficient but requires a derivable form for the error norm (note that this is not the same as requiring a closed form for the complete error functional). This is no limitation in data modelling activities when the MSE norm is used. This algorithm computed the rate of change of the activations of each layer as a function its connections with the previous one. The information is then back propagated from the output down to the network input in order to compute the gradient vector. This method has a computational complexity $O(s)$ versus $O(s^2)$ for finite differences making, it the prime choice for software implementations.

In order to solve the minimization problem associated with training three different methods are provided: conjugate gradient (CG), Broyden-Fletcher-Goldfarb-Shano (BFGS) and L-BFGS (Bishop 1995). Both BFGS and L-BFGS (Limited Memory-BFGS) are quasi-Newton methods that iteratively build an approximation to the Hessian matrix without directly computing it.

The training sequence also requires some kind of line search minimization once a suitable descent direction has been found. Recomputing the gradient at every step would be too costly to be practical so instead Brent's method is used to locate the extreme along a search direction (Brent 1973).

4 A simple test case

In order to explore the capabilities of the method and to determine the best strategy for application it is useful to start with a simple problem where the solution can be easily found. In this case the discrete gust response of a a simple 2-dof bidimensional system has been chosen. On one hand the model still retains some of the physics of a real problem and, on the other hand, its numerical complexity is simple enough as to allow for an exhaustive search of the parameter space. This way the accuracy of the neural network approximation can be tested across the complete design space.

To obtain a quick estimate of the aerodynamic loads an unsteady potential model is used. While compressibility and viscous effects are neglected, it is important to remark that the model is not linearized (i.e. geometric changes are not assumed small and wake rollup is accounted for). Thus, the system response is not easy to predict beforehand and constitutes a good test of the generalization capabilities of the neural network.

The degrees of freedom of the system are the pitch angle (θ) and the vertical displacement of the center of gravity (δ). Its dynamic behaviour is characterized by its mass (m) and moment of inertia (I_G) per unit length along the span and center of mass position (x_G).

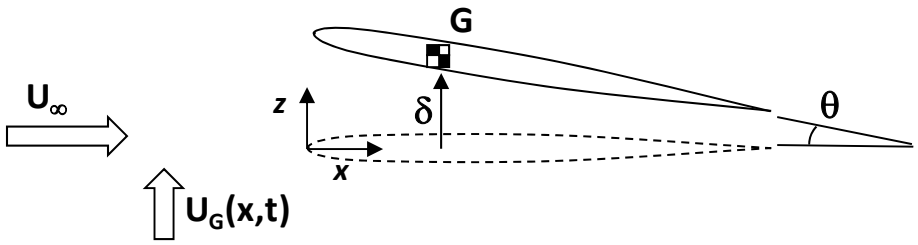

Fig. 3 - Simple 2-dof system for gust response analysis

The discrete gust profile is taken from the EASA CS-25.341 definition. Constant Equivalent Air Speed (EAS) is assumed at any altitude (h) and air density is obtained from the ISA atmosphere. Under these assumptions it is clear that the maximum vertical acceleration due to the gust has the following functional dependence:

$$\ddot{\delta}_{\max} = f\left(m, I_G, x_G, c, EAS, h, H, R_1, R_2\right) \tag{9}$$

where c denotes the airfoil chord and H the gust gradient distance. Parameters R_1 & R_2 are defined in CS-25 and depend on the aircraft weights (MLW, MZFW & MTOW). From dimensional analysis it follows that

$$\frac{\ddot{\delta}_{\max} c}{EAS^2} = f'\left(\frac{I_G}{mc^2}, \frac{x_G}{c}, \frac{h}{c}, \frac{H}{c}, R_1, R_2\right) \tag{10}$$

As for a given aircraft the values of R_1 and R_2 are constant, the maximum load factor due to the gust can be characterized as a function of four parameters only. In the analysis that follows the parameters chosen as free are

- Flight altitude (h)
- Gust gradient distance (H)
- Radius of gyration $R_y = \sqrt{I_G / m}$
- Aircraft static margin $SM = (0.25 - x_G)/c$ (x_G is measured from the leading edge)

In order to obtain results which are at least plausible the values of the different parameters are based on real world aircraft. Taking as reference a regional transport turboprop (ATR-72 class) the following values were chosen:

Parameter	Value
h	0 - 7000m
H	9 – 107m
EAS	110m/s
c	2.25m
m	700kg/m

Ry	3.9 – 5.1m
R1	0.97
R2	0.91
SM	0.05 – 0.40

Table 1 – Parameter values for 2-dof model

Taking 10 equally spaced values for each parameter within the bounds stated above a database of 10^4 points is constructed. While such a comprehensive coverage is probably unnecessary, it will allow obtaining detailed distributions of the approximation error and choosing the best strategy to accurately model the sample data.

5 Initial test with two free parameters

Given the difficulty visualizing the error distribution when the parameter space is four-dimensional it is interesting to first analyze the behaviour of the neural network when there are only two free parameters.

To this effect the rotational degree of freedom has been constrained rendering the moment of inertia and center of mass position irrelevant. Under these assumptions the maximum (positive) load factor increase due to an upward gust is plotted in Fig. 4.

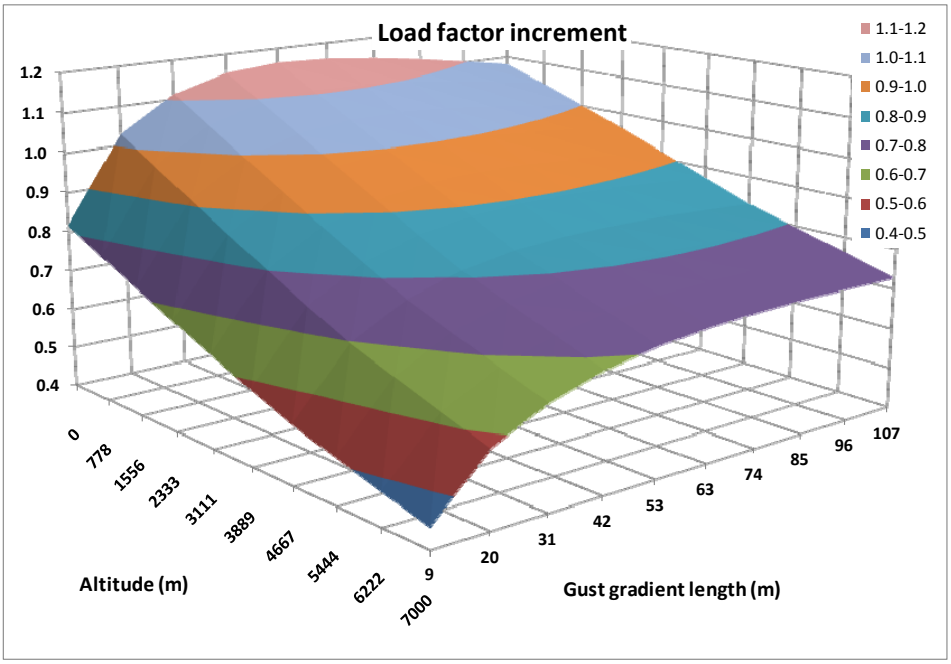

Fig. 4 - Load factor increase as a function of gust size and altitude

5.1 Data normalization

While a neural network can, in theory, approximate an arbitrary distribution like that shown in Fig. 3, common practice dictates that all the variables should have comparable ranges.

When the different parameters have vastly different orders of magnitude (as is the case here) the magnitude of the required network weights will also have a large spread. This by itself does not degrade the quality of the approximation bust has the potential to cause problems during the training process. Given the highly nonlinear behaviour of the network the weights must be obtained using iterative algorithms which might become slow or even fail to converge if the initial guess is too far off from the solution.

While it is not possible to provide a good initial guess of the correct weights, it is at least desirable to start with the correct order of magnitude. To this effect it is common practice to first normalize the training data so that both the inputs and outputs values have zero mean and unit variance.

The statistics of the data shown in Fig. 4 are given by:

Parameter	μ (mean)	σ (standard deviation)
Altitude (m)	3500	2250
Gust gradient length (m)	58.0	31.3
Load factor increase	0.867	0.165

Table 2 - Parameter distribution statistics

The scaled values are computed according to:

$$x_{scaled} = \frac{x_{original} - \mu}{\sigma} \tag{11}$$

The scaled data set is shown in Fig. 5

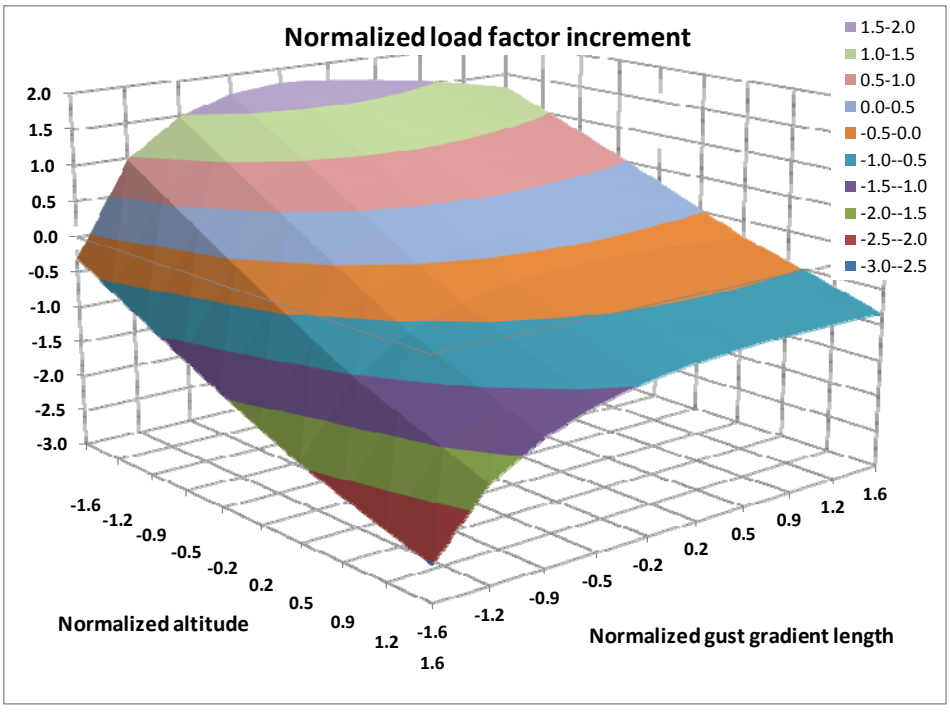

Fig. 5 - Normalized system response

Once the inputs have been transformed to zero mean and unit variance it is at least possible to find an initial set of networks weights which keeps the neurons from saturating and therefore maintain the ability of the network to approximate the target data. Therefore the initial weights are chosen in such a way that the net input of each neuron (in the initial state) has also zero mean and unit variance. This is achieved (assuming random uncorrelated network inputs) if the weights for each neuron are taken form a distribution such that (Bishop 1995)

$$\mu_w = 0 \quad \sigma_w = \frac{1}{\sqrt{d}} \tag{12}$$

where d is the number of neurons in the previous layer (that is, the number of weights used to calculate the input to the neuron being considered). Given that the minimization algorithms find local rather that global extrema, starting with a random set of weights means that the result of the training process is non-deterministic. That is, repeating the training several times yields different network weights. Given the reduced computational cost of the training process this is no serious limitation, several runs can be performed and the best result kept (Bishop 1995).

5.2 Effect of the choice of network configuration

The choice of the number of hidden layers and neurons in the network must strike a balance between accuracy and level of generalization. A network with a sufficiently large number of free parameters can fit exactly an arbitrary training set. This is however no guarantee that any real trend has been extracted from the training data. In fact, it may well happen that the approximated results present large oscillations (overfitting) predicting unrealistic values outside of the training points. On the other hand, if the network is not given enough flexibility it will be impossible to fit the training data with a reasonable accuracy. There is no general rule that suggests the best network configuration for the problem at hand so some investigation is required.

The data of Fig. 5 is smooth and does not contain localized peaks so it is easy to approximate. Assuming that a single hidden layer is used a number of neurons on the order of three should be adequate (Fig. 6).

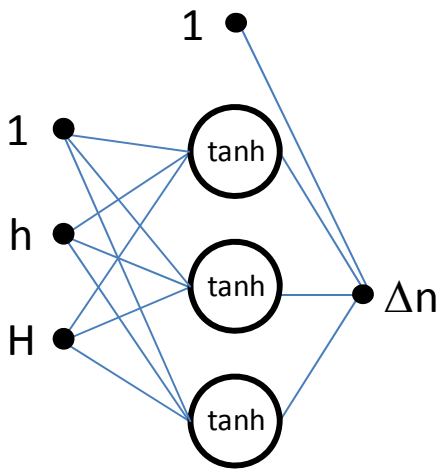

Fig. 6 - Example network with three neurons in the hidden layer

On the example shown in Fig. 6 two nodes labelled "**1**" are included. These represent virtual units with fixed activation that allow the code to treat biases in the same manner as the network weights. The hidden neurons have hyperbolic tangent activation functions whereas the output layer is linear. The total number of free parameters in the example network is 13.

When the data of Fig. 5 is used to train networks with different number of neurons in the hidden layer the error distributions of Table 3 - Distribution of approximation error Table 3 are obtained

Hidden Neurons	Average	RMS-Average	Minimum	Maximum
2	-6.95E-06	1.26E-02	-2.91E-02	4.14E-02
3	-2.05E-05	5.97E-03	-2.07E-02	1.45E-02
4	1.77E-05	2.68E-03	-6.79E-03	8.47E-03

Table 3 - Distribution of approximation error (not scaled)

Please note that, for ease of interpretation, the values in Table 3 are the actual errors in the load factor increment, not the errors in the normalized value. In this case, as far as gust loads are involved, a 3-neuron network seems enough to achieve a sufficient precision over the parameter space (RMS error below 1% and maximum deviation of some 2%). Because this simple example is very easy to visualize, it is also worth plotting the actual error values to investigate how the error is distributed.

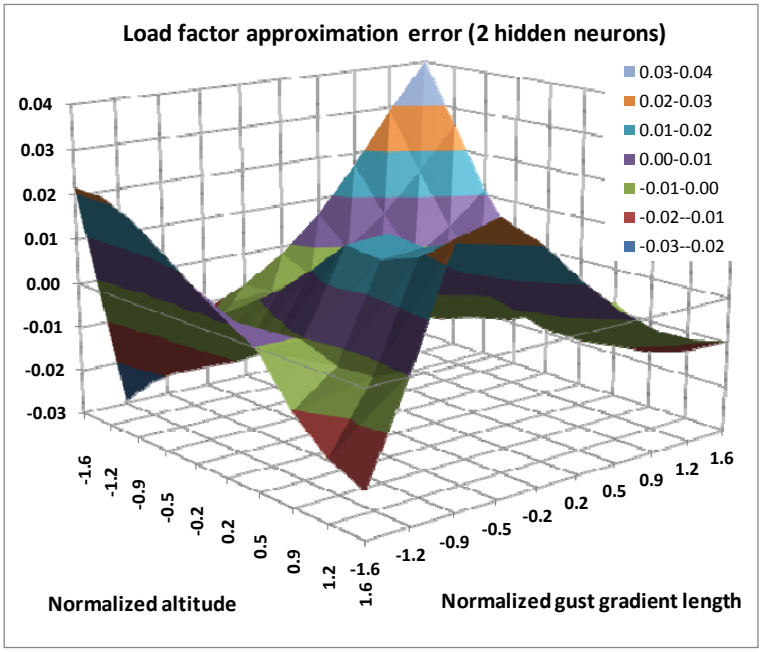

Fig. 7 - Error distribution (2 hidden neurons)

For the simplest network (Fig. 7) it is seen that error is fairly well distributed (no overwhelmingly large peak) with the largest values found at boundaries of the parameter space. As the number of hidden neurons is increased the distribution becomes more complex (as expected from the larger number of free parameters in the network) but the errors remain spread across the domain (Fig. 8 & Fig. 9).

The error plots shown do not reveal, however, if the network is overfitting the training data. To discard this possibility the approximation error has to be measured using an input sample different from the training data. To this end a new sample of the same parameter domain has been built using a grid of 17x17 points instead of 10x10. Given that 17 is a prime number, all the points thus obtained will be different from the original set (except for, obviously, the four corners of the domain). If overfitting has taken place large increase in approximation errors shall appear on the new sample.

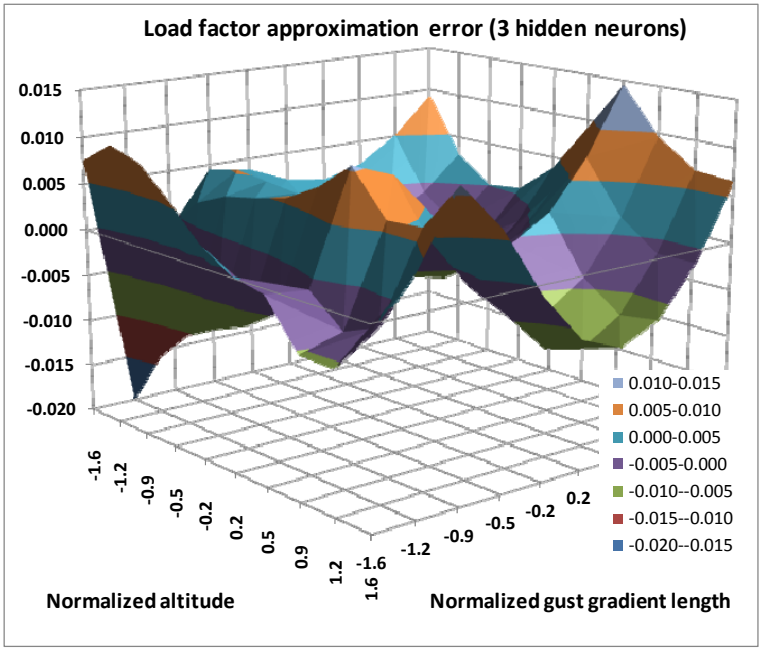

Fig. 8 - Error distribution (3 hidden neurons)

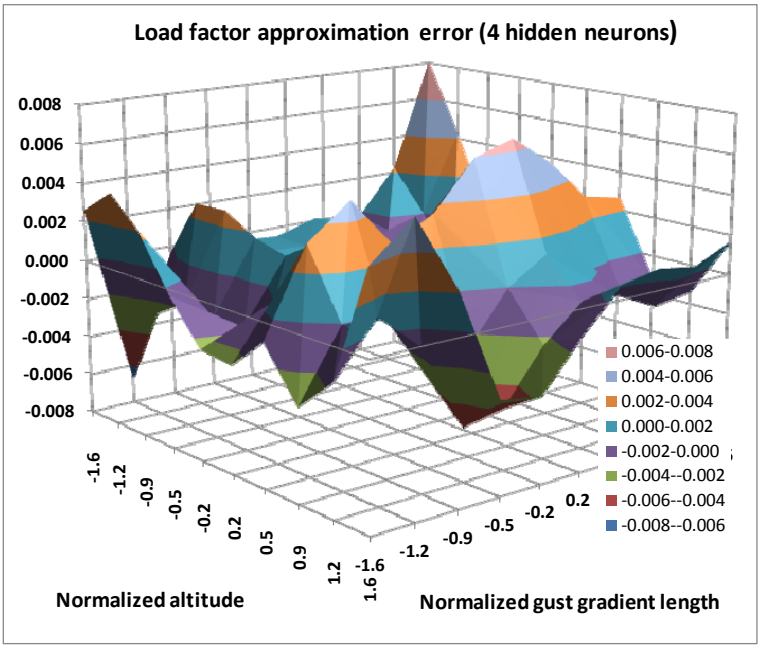

Fig. 9 - Error distribution (4 hidden neurons)

The error statistics for the 17x17 sample are given in Table 4. It shows that the performance of the three networks remains basically the same with the new sample so the possibility of overfitting can be discarded.

Hidden Neurons	Average	RMS-Average	Minimum	Maximum
2	-6.95E-06	1.26E-02	-2.91E-02	4.14E-02
3	-2.05E-05	5.97E-03	-2.07E-02	1.45E-02
4	1.77E-05	2.68E-03	-6.79E-03	8.47E-03

Table 4 - Distribution of approximation error (17x17 sample)

So far the network configurations explored have been restricted to a single hidden layer. This choice might seem arbitrary so it is interesting to explore the possible advantages gained from topologies with more hidden layers.

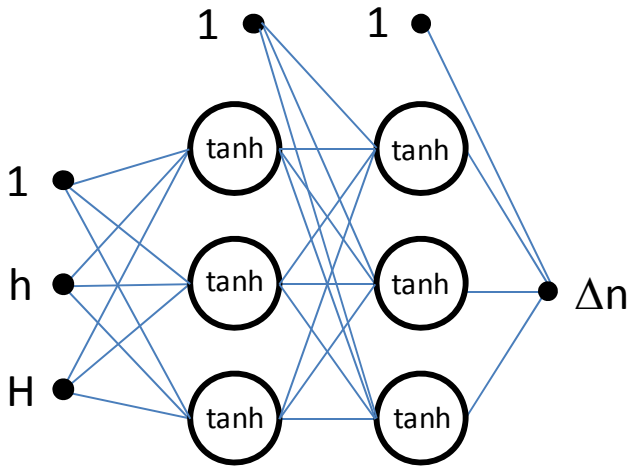

Fig. 10 - Network with two hidden layers of three neurons each

In Fig. 10 the two-hidden-layer equivalent of Fig. 6 is shown. The number of free parameters increases from 13 for the single-hidden-layer network to 25. Clearly, unless the approximation delivered by the more complex network is substantially better it makes little sense to use it. In Table 5 the performance of several networks having two hidden layers with the same number of neurons is summarized.

Hidden Neurons	Average	RMS-Average	Minimum	Maximum
2+2	3.09E-05	1.00E-02	-1.56E-02	3.25E-02
3+3	-2.27E-05	2.88E-03	-7.01E-03	8.61E-03
4+4	-1.28E-05	2.77E-03	-6.62E-03	9.31E-03

Table 5 - Approximation error with two hidden layers (not scaled)

Comparing the data of Table 5 with the values from Table 3 is becomes evident that the benefit of adding a neuron to a single hidden layer network is far greater than the advantage gained by including an additional hidden layer. What's more, for the network with 4 neurons in the first hidden layer there is no gain at all from adding a hidden layer. It follows that, for this simple problem, networks with a single hidden layer are the obvious choice (as they can deliver the same accuracy with fewer free parameters).

5.3 Training algorithm

Given that the quality of the approximation depends largely on the success of the training process, the latter deserves a careful examination. In particular, it is important to learn if

there is a training method which yields better approximation. It is also important to characterize the extent to which the non-deterministic character of the error minimization influences the end result.

Taking as an example the single layer network with four hidden neurons, repeating the training with the CG solver using different random initial weights each time yields the results of Fig. 11 are obtained. The network with the highest number of free parameters is chosen because it is expected to show a larger scatter of the results.

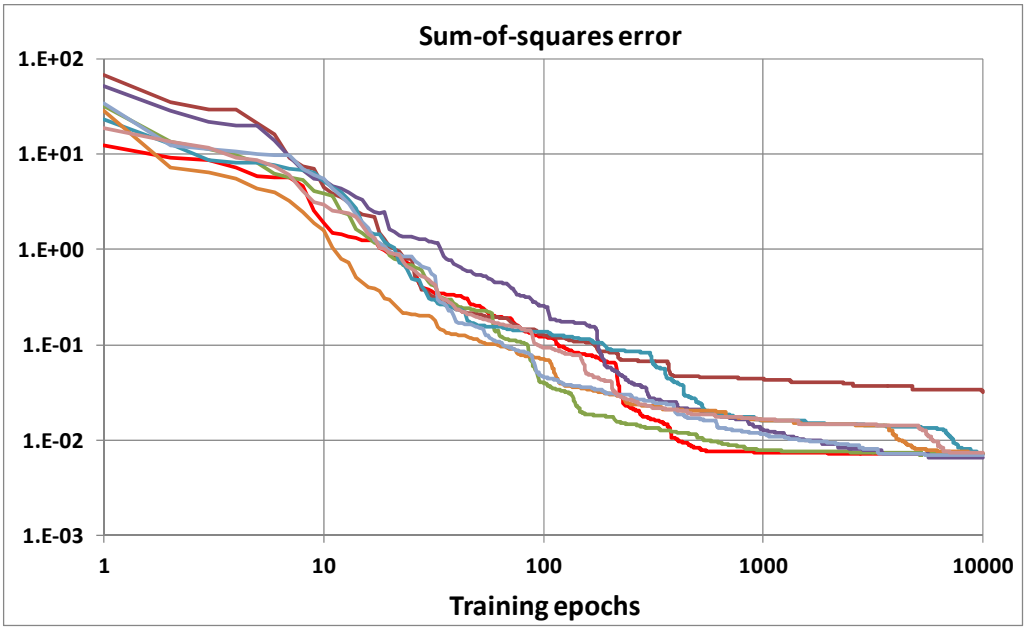

Fig. 11 - Convergence history for several training runs with the same dataset

The non-deterministic result of the training is clearly demonstrated in Fig. 11. However, all of the cases shown, save one, converge to a very similar error level. This means incorrect initial guesses can be easily eliminated by training the network several times and discarding the worst results. It is worth mentioning that the frequency with which a result as anomalous as that corresponding to the brown curve in Fig. 11 appears is much lower than what the chart might suggest. In fact, the number of runs that were needed to find such a bad convergence behaviour was well over 20. Not all the runs have been plotted in order to reduce clutter in the chart.

The effect of the nonlinear solver choice also deserves some attention. To isolate the effect of the solver several runs with the same set of initial random weights have been performed. The results are summarized in Fig. 12.

Fig. 12 - Effect of solver choice on training

The best rate of convergence is achieved by the BFGS quasi-Newton solver which reaches the minimum residual in 700 cycles approximately. The CG and the L-BFGS solver need, on the other hand, close to 5000 main iterations (excluding line search steps) to reduce the error to the same level. Given that all the methods reach the same level of approximation the choice will be based purely on efficiency considerations. In terms of CPU time per iteration the BFGS algorithm is roughly twice as expensive as the other two so in the end it delivers the solution in about one third of the time. Using current commodity hardware (Intel Core i3 M350) CPU time for the BFGS method is less than one second. CG & L-BFGS contain additional parameters that can be tuned in order to improve performance (e.g. the interval at which the search directions are reset). While this fine-tuning could yield slightly increased performance, it is not expected to change the overall result in a significant way.

When applicable, BFGS is usually chosen over CG or L-BFGS due to superior performance. It is worth mentioning, however, that the BFGS method does not scale well to very complex networks. The resources needed to store and update the approximate Hessian matrix grow rapidly with the number of free parameters so at some point a switch to CG of L-BFGS is required (Bishop 1995). Networks of this complexity fall however outside the scope of this document.

5.4 Number of samples required for training

Up to this point a large (100 points) sample has been used for training which delivers a comprehensive mapping of the system response. However, for the neural network to be

really useful it must yield an accurate prediction using a small sample size. For a single hidden layer perceptron, the network with 2 hidden neurons has 9 free parameters, while using 3 neurons requires setting 13 weight values (this figure includes the biases). The number of training points should be larger than the number of free parameters, otherwise the problem is underdetermined. In order to have a well-posed problem a reduced data set containing 4 equally-spaced points along each axis has been constructed (note that only the corner points coincide with the 10x10 sample). This has been used to train the networks yielding a final error as shown in Table 6.

Hidden Neurons	Sum-of-squares error
2	4.05E-02
3	2.23E-04

Table 6 – Error at the end of training with 4x4 sample set

The set of weights obtained from the training with 4x4 points was then used to approximate the 10x10 sample in order to determine the errors.

Hidden Neurons	Average	RMS-Average	Minimum	Maximum
2	-3.67E-02	8.69E-02	-2.09E-01	4.86E-02
3	-9.90E-03	2.61E-02	-9.02E-02	2.52E-02

Table 7 - Approximation error for 10x10 set after training with 4x4 samples (not scaled)

Comparison of Table 7 with Table 3 reveals an important increase in the approximation error. For the 3-neuron network the maximum deviation is now 9% while the RMS-average error is 3% approximately. While this loss of accuracy might not be acceptable depending on the application, the information is still valuable if it correctly points to the critical combination of parameters. Fig. 13 shows the load factor increment approximation over the complete domain recovered from the 4 by 4 point sample. While not identical to Fig. 4, the shape of the distribution is well captured and the position of the maximum load is predicted accurately. Table 8 lists the peak load factor and its location for both the complete 10x10 set and the recovered data using 3 hidden neurons and a 4x4 sample.

Sample	Maximum load factor	Altitude (m)	Gust gradient length (m)
10x10	1.16	0	53
Interpolated from 4x4	1.19	0	53

Table 8 – Maximum load factor obtained with a 10x10 set and interpolated from a 4x4 sample

From Table 8 it is apparent that the approximation delivered by the network is very good in this case. Not only the position of the peak load factor has been correctly identified, the value of the critical load factor has been predicted with less than 3% error. Thus, it is sensible to expect that neural networks might be useful when a simulation campaign is undertaken in order to determine critical load cases. As computations progress and enough sample points

become available, a network can be trained to provide an approximation to the system response across the complete parameter space. This has two potential advantages:

1. It provides a general picture of the response surface giving insight on the underlying physics.
2. The approximate values (which can be computed at almost zero cost) can be used to explore the parameter domain in a very short time in order to estimate the location of the critical load cases. This information can then be used to guide the collection of new sample points, reducing the time needed to identify the load extremes.

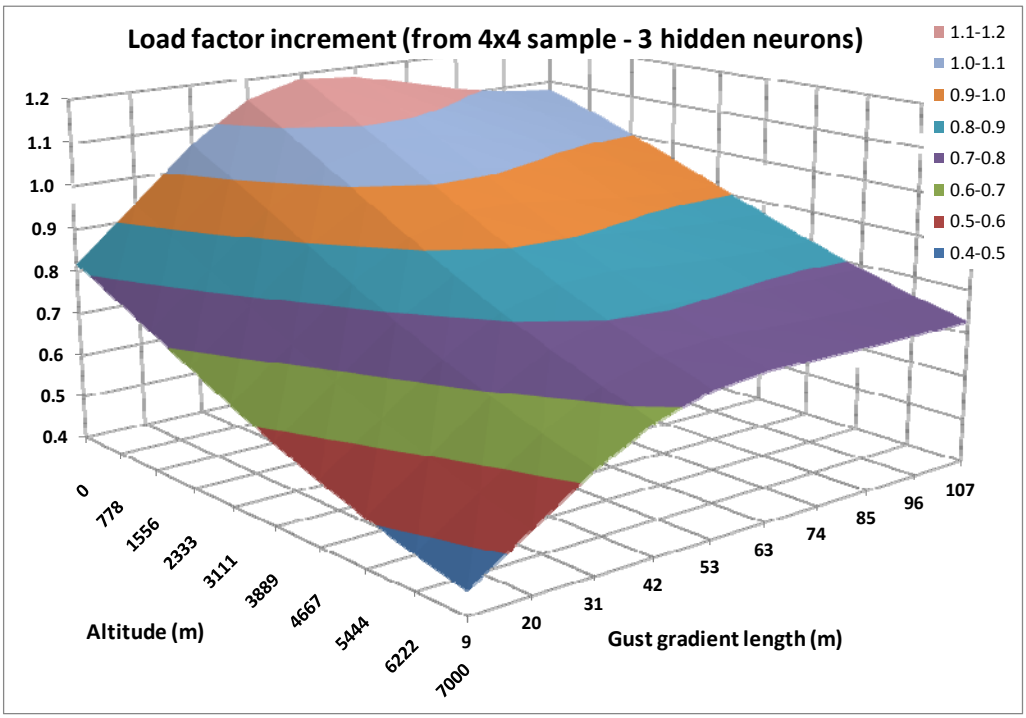

Fig. 13 - Approximate gust load factor recovered from a 4x4 sample

6 Test with four free parameters

Once the general behaviour of the approximation has been examined, the same technique can be applied to a more complex case. Using the example of section 4 the effect of the so called “curse of dimensionality” becomes evident. While a comprehensive coverage of the parameter space required just 100 sample points for the single-dof system, the same resolution needs 10000 samples when the rotational degree of freedom is active. This remains feasible because a very simple model has been chosen, but would be prohibitive in the case of a high fidelity model. It is therefore a case where neural networks might provide an important advantage with respect to “blind” computations. As it is not possible to plot a

scalar function (the load factor) defined in a four-dimensional space, from this point on the analysis shall rely on the statistical analysis of the output data.

6.1 Choice of a suitable network configuration

When only two free parameters where taken into account, a network with a single hidden layer of three neurons delivered an acceptable performance. Given the increased complexity of the problem now under consideration a higher number of hidden neurons might be needed. Tests similar to those of section 5.2 haven been performed to determine the optimum network configuration. To follow a process more representative of what a real-life workflow would be, the networks will be trained using only a small subset of the 10000 data points available. To assess the effect of the subsample size two different datasets have been used:

1. Set of 81 samples with three equally spaced points for each free parameter
2. Set of 256 samples, using four points along each direction

While these number of data points might seem large, they are comparable to the number of evaluations a nonlinear solver would require in order to locate the critical condition. A common choice for minimizing/maximizing functions whose gradient cannot be computed is Powell's method (1964). Each step of the requires in this case line searches along four independent search directions plus an additional search along the newly computed search direction. In total, five line searches are needed per iteration, each of which requires several function evaluations. Assuming five evaluations per search yields a total over 20 evaluations per iteration. Thus, the sample sizes chosen are not unreasonable. It must be stresses that minimization algorithms can get "stuck" easily in local extrema failing to capture the critical condition. Thus, some additional exploration of the complete parameter space is still required in order to discard this eventuality. If the neural network is able to predict the position of the critical load condition while delivering a reasonably accurate picture of the system response over the parameter space, it would prove a useful tool for simulation campaign monitoring

Networks with 2, 3 and 4 neurons in the hidden layer have been trained using the reduced datasets and then the error approximating the complete dataset has been evaluated. Results are shown in Table 9 & Table 10.

Hidden Neurons	Average	RMS-Average	Minimum	Maximum
2	-1.36E-02	4.62E-02	-1.66E-01	6.87E-02
3	-1.32E-02	4.93E-02	-1.28E-01	6.90E-02
4	-2.60E-04	1.37E-02	-5.58E-02	3.73E-02

Table 9 - Approximation error after training with 81 samples (not scaled)

Hidden Neurons	Average	RMS-Average	Minimum	Maximum
2	-7.96E-03	3.02E-02	-1.32E-01	8.38E-02
3	9.68E-03	2.26E-02	-5.21E-02	8.29E-02
4	-7.59E-03	2.40E-02	-8.42E-02	3.49E-02

Table 10 - Approximation error after training with 256 samples (not scaled)

The data in Table 9 & Table 10 shows that the approximation accuracy for all the networks is similar with the 4 neuron network delivering a slightly better accuracy (especially for the smaller training set). Keep in mind that small differences are not significant due to the non-deterministic character of the training process. Re-training a network with different initial weights would change the results. In fact, to mitigate the uncertainty associated with the choice of initial weights, each network has been trained five times using the same data sample but with different random initial weights each time. The results shown in the tables belong to the best set of weights (those yielding the minimum MSE for the training data). It was found that the likehood of finding a "bad" initial weight combination was far greater than in the case of two free-parameters. Therefore, the extra training runs were considered essential to achieve good accuracy. Given that the CPU time for each training run is very short (around one second) this does not increase the cost in any significant way.

It is interesting to note that while the number of free parameters has doubled with respect to the test of section 5, the networks deliver similar performance without the need for increased internal complexity (number of hidden neurons). This is a demonstration of the great flexibility of multilayer perceptrons in the role of interpolators. Using the 81-point sample and the perceptron with 4 hidden neurons an average accuracy of 2% over the complete domain is achieved with peak errors of 5%. This is quite acceptable as a first guess of the critical loads but, more importantly, the critical parameter combination is well predicted. Table 11 compares the extreme load factor obtained from the complete sample with the predictions of the different networks. The shaded cells indicate the peak load for each data set. All the networks attain their maximum load factor in the neighbourhood of the real critical condition[1].

					Training with 81 samples			Training with 256 samples		
SM	**R_Y(m)**	**h(m)**	**H(m)**	**Full dataset**	**2 neuron**	**3 neurons**	**4 neurons**	**2 neuron**	**3 neurons**	**4 neurons**
0.05	**5.1**	**0**	**41.7**	1.138	1.020	1.066	1.099	1.099	1.146	1.138
0.05	**5.1**	**0**	**52.6**	1.144	1.091	1.121	1.127	1.159	1.144	1.164
0.05	**5.1**	**0**	**63.4**	1.133	1.140	1.152	1.140	1.173	1.136	1.149

[1] Please note that the gust gradient distance step in the complete dataset (10000 samples) is 10.9m. Therefore, there are no additional data points between those shown.

0.05	**5.1**	**0**	**74.3**	1.113	1.159	1.157	1.135	1.154	1.122	1.111

Table 11 - Critical parameter combinations and peak load factor

Table 11 shows that the peak load factor is predicted in every case with better than 2% accuracy and it always takes place in the immediate neighbourhood of the real critical parameter combination. The neural networks therefore do a good job of mapping the complete parameter envelope while guiding the search for critical load cases. What's more, they do so at a negligible cost. In fact the CPU time needed to recover the 10000 approximate load cases is less than the cost of computing of a single data point (even with the simple model chosen).

6.2 Reducing the cost of training

Up to this point the samples used to train the network came from the nodes of a uniformly-spaced regular grid in parameter space. This is far from optimal from the sampling point of view. A very small number of values of each parameter can be sampled if the sample size is to remain acceptable. The question arises then about the effect of a more refined sampling strategy on the cost of training. To test this effect additional training sets have been prepared using the latin hypercube sampling technique (Iman 1981). This method generates a sample of size *n* using combinations of *n* different values for each free parameter. Thus, it explores a larger set of values for each parameter while keeping the total amount of data low. The parameter combinations are chosen at random with the restriction that any particular value of each parameter can appear only once in the complete data set. As a result, the size of the sample set is a function only of the number of values of each parameter, but does not depend on the problem dimensionality. Latin hypercube sampling is a purely stochastic sampling technique, there is no guarantee of uniformity in the sample. While multilayer perceptrons are very efficient interpolators, they are not reliable extrapolators. It is therefore convenient to include the extreme values of the parameter space in the training sample to avoid large errors. To this end an additional set of $2^4 = 16$ points has been included which contains the corners of the hypercube (all the possible combinations of maximum and minimum values of each parameter). Three latin hypercube samples have been created using 20, 30 and 40 equally-spaced values for each parameters; when supplemented with the corner points the modified datasets contain respectively 36, 46 and 56 points. Training the four-hidden-neuron network with these samples and then approximating the complete dataset (10000 points) yields the errors shown in Table 12.

Sample size	Average	RMS-Average	Minimum	Maximum
36	-8.26E-04	1.12E-02	-2.99E-02	5.06E-02
46	7.11E-05	1.30E-02	-3.79E-02	4.79E-02

56	1.28E-03	9.63E-03	-4.53E-02	2.81E-02

Table 12 - Approximation error for training with modified latin hypercube sampling (not scaled)

Comparison of Table 12 against Table 10 shows that the results of training with the modified latin hypercube sample are generally better than those obtained with the uniform 256-point sample. Even for the smallest dataset (36 points) the results fare well against those of table Table 10. A reduction by a factor of 7 in sample size has been achieved without degrading accuracy. It is important to verify also that the critical parameter combination is also well predicted when using the reduced training sample. The results are summarized in Table 13.

					Modified latin hypercube sample size		
SM	**R_Y(m)**	**h(m)**	**H(m)**	**Full dataset**	**36**	**46**	**56**
0.05	**5.1**	**0**	**41.7**	1.138	1.131	1.156	1.128
0.05	**5.1**	**0**	**52.6**	1.144	1.131	1.162	1.134
0.05	**5.1**	**0**	**63.4**	1.133	1.117	1.151	1.127
0.05	**5.1**	**0**	**74.3**	1.113	1.097	1.129	1.112

Table 13 - Critical parameter combinations and peak load factor

Once again, the neural network is able to predict accurately the critical parameter combination as well as the peak load factor (with less than 1% error) despite the vastly reduced sample size. We can therefore conclude that the modified latin hypercube sampling strategy is more efficient than “brute force” uniform sampling. Note that the number of evaluations required to build the smallest sample set (36) is comparable to the number of evaluations that a maximization algorithm would need in order to find a rough estimate of the critical load. With the same computational effort a neural network delivers not only the peak load, but also an accurate map of the system response over the complete parameter space.

7 Use of neural networks for data fusion

The interpolating capabilities of the multilayer perceptron can also be used to combine multi-fidelity results into a coherent dataset. Besides interpolating the system response, the neural network can be used to build a map of corrections to a low-order solution from a limited number of higher fidelity results.

To test this application a reduced fidelity dataset has been built by using a coarse time and space discretization. The settings for both the low-fidelity and high-fidelity samples are given in Table 14. The coarse discretization has been purposefully chosen in order to cause a noticeable loss of accuracy.

	Panels along airfoil	$U_\infty \Delta t / c$
Low-fidelity	12	0.1

High-fidelity	30	0.05

Table 14 – Setting for low-fidelity and high-fidelity models

Two datasets were built next, a small high-fidelity sample of 26 points and a large low-order database of 166 points. This is meant to be representative of a real-life situation, where high-fidelity data is scarce due to cost limitations.

The 26-point sample contains results from both models in order to map the discretization error across the domain. The sample is taken using the modified method (16 corner points plus 10 latin hypercube samples). The discretization error (difference between the load factor predicted by the high-order model and the low-fidelity solution) statistics for the 26-point sample are given in Table 15.

Average	RMS-Average	Minimum	Maximum
8.06E-02	8.83E-02	1.16E-02	1.39E-01

Table 15 – Discretization error for 26-point modified latin hypercuble sample

The low-order results are clearly biased, consistently underestimating the load factor increase due to the gust.

A neural network with three hidden neurons (the number of samples is not adequate for a more complex network) has been trained with the 26-point sample in order to obtain a representation of the discretization error across the complete parameter domain. This will serve to correct the low order approximation at a later stage.

Using the large 166-point low order dataset (obtained with 16 corner points and 150 latin hypercube samples) a four hidden-neuron network has been trained. It provides a low-order approximation over the complete parameter envelope. When compared against the full dataset (10000 high-order points) the errors are distributed according to Table 16.

Average	RMS-Average	Minimum	Maximum
8.89E-02	9.30E-02	2.82E-03	1.59E-01

Table 16 – Error distribution for network trained with 166 low-order samples

The errors in Table 16 are slightly larger than those of Table 15 because they contain an additional term due to interpolation error. However, the discretization error seems the dominant component.

Finally, the low order approximation is corrected using data from the three-neuron network trained with the discretization error sample (26 points). The error distribution for the fused data is given in Table 17.

Average	RMS-Average	Minimum	Maximum
-1.10E-04	7.08E-03	-1.78E-02	3.75E-02

Table 17 – Error distribution for low-order approximation (obtained from 166-sample set) corrected with error map interpolated from 26-point sample

The fused data is, as expected, more accurate than the approximation obtained using only low-order data (Table 16). More important, the response surface obtained with a small number of high-order results and a large low-order sample is a better approximation than what is possible using only an intermediate number of high-fidelity samples (see Table 12). Therefore, data fusion provides a synergistic approach which increases accuracy through more extensive coverage of the parameter space (large low-order sample) while reducing the total cost of the simulation campaign (because the number of high-fidelity computations is kept to a minimum).

It is also important to check the accuracy predicting the peak load factor and the critical parameter combination. The results are given in Table 18.

					Training sample	
SM	**R_Y(m)**	**h(m)**	**H(m)**	**Full dataset**	**166 low-order**	**166 low-order + 26 high-order**
0.05	**5.1**	**0**	**52.6**	1.144	1.012	1.160
0.05	**5.1**	**0**	**63.4**	1.133	1.013	1.148

Table 18 - Critical parameter combinations and peak load factor

Once again, the predictions of the fused dataset are superior to those obtained from low-order-only data and high-order-only modelling with an intermediate-size dataset (Table 13). The benefits of the approach are thus evident.

8 Test with 3D model

Neural networks are purely mathematical tools, they can provide an approximation to a dataset irrespective of the underlying physics. Therefore, the results for the 2D model studied in the previous sections should be extensible to 3D problems. While this seems reasonable it deserves verification. To this effect the techniques already described shall be applied to the gust response of a 3D aircraft model. The load factor increments have been calculated using an unsteady doublet-lattice solver and a simple model of the ATR-72 geometry. As only load factor increments due to the gust are sought, the effects of camber and thickness of the lifting surfaces have been neglected, taking only their planform into account. Fig. 14 shows the general dimensions of the aircraft. Some relevant mass and aerodynamic data are given in Table 19 and the variation ranges for the free parameters are listed in Table 20.

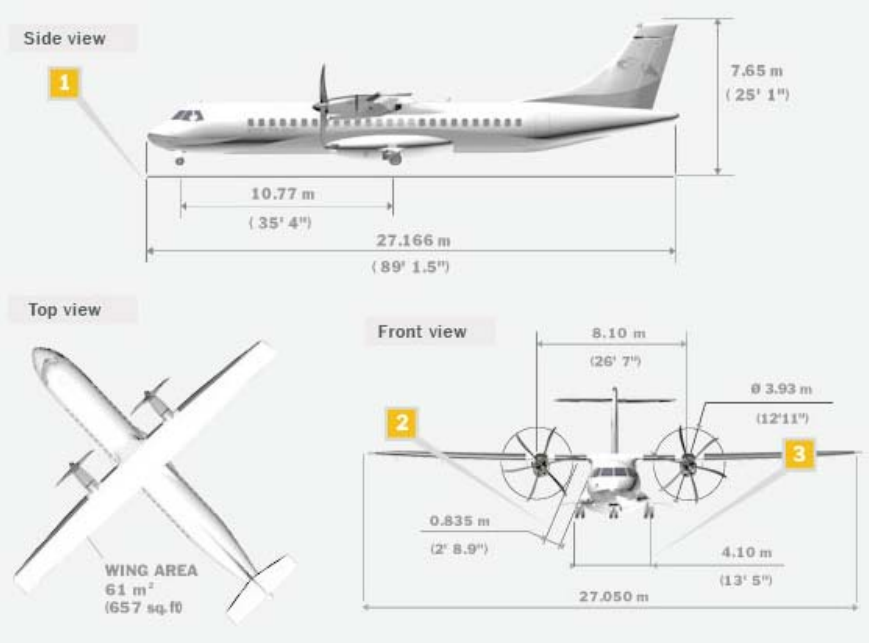

Fig. 14 - ATR-72 three-view schematic. Source: www.atraircraft.com

Mass	19000kg
R1	0.97
R2	0.91
Wing area	61m2
Mean aerodynamic chord (MAC)	2,29m
Wing span	27,1m
Fuselage length	27,2m
Neutral point position (from nose)	12,9m
HTP surface	11,8m^2

Table 19 - Summary of properties of 3D model

Radius of gyration Ry	4,5 - 5,5m
Gust gradient distance (per CS-25)	9 – 107m
Static margin (%MAC)	5 - 40
Altitude	0 - 7000m

Table 20 – Ranges of free parameters for 3D model

Following a strategy similar to section 6.2 a small sample of the response surface will be constructed using a combination of corner points and latin hypercube sampling. It is always reasonable to include the extremes of the parameter space (corner points) if a coverage of the complete domain is sought. Besides the 2^4 corner points, 10 additional latin hypercube samples have been taken. The cost of the complete dataset is thus of the same order as the cost of generating the basic corner sample.

Using the 26-point sample (16 corner + 10 latin hypercube) a network with three hidden.neurons has been trained. Next, the trained network has been used to contruct a 10^4

point map of the complete parameter envelope using uniform point spacing along the four axes. The predicted critical condition along with the actual value (computed a posterior) of the load factor is given in Table 21.

SM	R_Y(m)	h(m)	H(m)	Predicted Value	Computed Value
0.244	5.5	0	107	1.23	1.01

Table 21 - Predicted vs. actual load factor

The predicted value is off by more than 20% so the approximation is obviously of low accuracy. This may be due to the latin hypercube sample being chosen completely at random with the possibility of irregular coverage of the parameter space. In order to check whether the critical parameter combination is correctly predicted, computations were performed for five adjacent points, with the results shown in Table 22.

SM	R_Y(m)	h(m)	H(m)	Computed Values
0.24	5.50	0.00	96.11	1.04
0.24	5.50	777.78	107.00	0.97
0.24	5.39	0.00	107.00	1.01
0.21	5.50	0.00	107.00	1.04
0.28	5.50	0.00	107.00	0.98

Table 22 - Computed values near predicted extremum

Looking at the values of Table 22 there is no evidence for a local extremum so in this case the prediction of the neural network was completely wrong. However, the results from the new computations can be used to refine the training of the network in order to increase the accuracy of the predictions. Training with 32 points (26 original + 6 new) yields an improved prediction for the critical load factor (see Table 23)

SM	R_Y(m)	h(m)	H(m)	Computed Value	Predicted Value
0.05	5.50	0.00	64.44	1.19	1.26
0.05	5.50	0.00	74.33	1.19	1.26
0.05	5.50	0.00	85.22	1.18	1.25

Table 23 - Predicted vs. actual load factor after re-training

Table 23 shows the critical condition as well as the neighbouring points in the matrix. This time the perceptron has been able to capture the correct parameter combination while predicting the maximum load factor with increased accuracy (less than 6% error). We see that online training can improve the predictions of the network as more data becomes available. This offers the potential to adjust the simulation campaign as it proceeds in order to identify the critical load cases with a minimum number of computations. At the same time a global representation of the parameter envelope is constructed which gives further insight

into the underlying phenomena than a simple maximization strategy (which tends to provide a very uneven coverage of the parameter space) .

9 Conclusions

The application of neural networks to a load determination problem where the dimensionality of the parameter space makes comprehensive searches inefficient has been explored. It has been shown that using a relatively small number of samples an accurate representation of the complete parameter space can be obtained. Even in those cases where the accuracy of the predicted peak load was reduced due to insufficient sample size a reasonable approximation to the critical parameter combination was obtained. This has the potential to reduce the number of computations needed in order to determine the critical load cases. At the same time the neural network provides a global description spanning the complete design space, something that a maximization solver is not capable of.

The cost of training the network (once the samples are available) and recovering a large set of approximate values is negligible. Therefore the neural network can be included into existing process workflows at virtually no extra cost. It has the potential to improve the quality of the data sets available (by filling gaps).

It was found that the outcome of the training step is unpredictable. The training algorithm can become trapped into local minima and deliver reduced accuracy. This was overcome repeating the training process several times for each sample and retaining only the best set of weights. While this ad hoc solution is functional given the reduced cost of training, the matter deserves further study.

It has also been demonstrated that multilayer perceptrons can be effectively used for data fusion tasks. Two sets of different fidelity results were combined yielding a better approximation than what each dataset was capable of alone. This has the potential to deliver a highly accurate representation of the complete parameter envelope while maintaining the total cost of the simulation campaign within acceptable limits.

Finally, it has been shown that the method is applicable to different levels of approximation (in this case 2D & 3D computations). The fact that neural networks provide a “black box” description of the system response not related to the underlying physics (they can be interpreted as representations of the probability density function of the training data) makes them very general tools applicable to a vast array of processes.

10 References

Bishop, C., 1995. *Neural Networks for Pattern Recognition*. Oxford University Press.

Brent, R.P., 1973. *Algorithms form Minimization without Derivatives*. Prentice-Hall

Demuth H and Beale M, 2002. *Neural Network Toolbox for Use with MATLAB. User's Guide*. The MathWorks.

Haykin, S., 1994. *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Hornik, K. et al., 1989. *Multilayer feedforward networks are universal approximators*. In Neural Networks, Vol. 2, No. 5, pp. 359-366.

Iman, R.L. et al., 1981. *An approach to sensitivity analysis of computer models, Part 1. Introduction, input variable selection and preliminary variable assessment*. In Journal of Quality Technology, Vol. 13, No. 3, pp. 174–183.

Lopez, R. and Oñate, E., 2006. *A Variational Formulation for the Multilayer Perceptron*. Proceedings of the 16th International Conference on Artificial Neural Networks. Athens, Greece, Vol 1, pp. 159-168.

Powell, M.J.D., 1964. *An efficient algorithm for finding the minimum of a function of several variables without calculating derivatives*. In Computer Journal, Vol. 7, pp 155-162.

Press, W.H. et al., 2002. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press.

Sima, J. and Orponen, P., 2003. *General purpose computation with neural networks: a survey of complexity theoretic results*. In Neural Computation, Vol. 15, pp. 2727-2778.

Wolpert, D.H. and MacReady, W.G., 1997. *No free lunch theorems for optimization*. In IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1, pp. 67-82.