FAULT DIAGNOSTIC SYSTEM FOR PREDICTIVE MAINTENANCE ON A BRAYTON CYCLE POWER PLANT

C. Vorster B.ing

Dissertation submitted in partial fulfillment of the requirements for the degree Magister in Engineering in Electronic Engineering of the Potchefstroom University of Christian Higher Education

Supervisor: Prof. C.P. Bodenstein

May 2004

Potchefstroom

Acknowledgement

To the One and only Almighty source of motivation, knowledge and love...

Abstract

Model-based fault detection and diagnostic systems have become an important solution (Munoz & Sanz-Bobi, 1998:178) in the industry for preventive maintenance. This not only increases plant safety, but also reduces down time and financial losses. This paper investigates a model-based fault detection and diagnostic system by using neural networks.

To mimic process models, a normal feed-forward neural network with time delays is implemented by using the MATLAB® neural network toolbox. By using these neural network models, residuals are generated. These residuals are then classified by using other neural networks. The main process in question is the Brayton cycle thermal process used in the pebble bed modular reactor. Flownet simulation software is used to generate the data, where practical data is absent.

Various training algorithms were implemented and tested during the investigation of modelling and classification concepts on two benchmark processes. The training algorithm that performed best was finally implemented in an integrated concept.

Opsomming

Modelgebaseerde foutopspoor- en diagnoseringstelsels het van kardinale belang geword in die industrie ten opsigte van voorkomende onderhoud (Munoz & Sanz-Bobi, 1998:178). Dit bevorder nie net fabrieksveiligheid nie, maar verminder ook aftye en finansiële verliese. Hierdie skripsie ondersoek `n modelgebaseerde foutopspoor- en diagnoseringstelsel deur van neurale netwerke gebruik te maak.

Om prosesse na te boots, word van normale vorentoevoer neurale netwerke met tydvertragings gebruik gemaak deur middle van MATLAB® neurale netwerk gereedskap. Residue word verkry deur van neurale netwerk modelle gebruik te maak. Die residue word dan geklassifiseer deur van ander neurale netwerke gebruik te maak. Die hoof proses onder die vergrootglas is die Brayton-siklus termiese proses wat in die korrelbed modulere reaktor gebruik word. Flownet word gebruik om die proses te simuleer en die nodige data te genereer.

Verskeie opleidingsalgoritmes is toegepas en getoets gedurende die ondersoek van die modellering en klassifiseringkonsepte op twee maatstafprosesse. Die algoritme wat die beste werkverrigting gelewer het, word dan uiteindelik geïmplementeer in die geïntegreerde foutopspoor- en diagnoseringstelsel.

Table of Contents

Acknowledgement I
Abstract II
Opsomming III
Table of Contents IV
List of Figures VI
List of Tables VIII
List of Abbreviations IX
List of symbols X

1 INTRODUCTION 1
1.1 Background 2
1.2 Problem statement 2
1.3 Proposed solution 3
1.4 Specific problems 3
1.5 Methodology 4
1.6 Research overview 4
2 FAULT DIAGNOSTICS AND RELATED RESEARCH 6
2.2 Diagnostic Methods 7
2.3 Some existing research 8
2.3.1 The radial basis function neural networks (RBF) 9
2.3.2 The probabilistic radial basis function network (PRBFN) 11
2.3.3 Kalman filters 14
3 NEURAL NETWORK THEORY 16
3.1 Background 17
3.2 Network topologies 19
3.2.1 Multilayer feed-forward networks 19
3.2.2 Recurrent networks 23
4 RESIDUAL CLASSIFICATION 25
4.1 Problem discussion 26
4.2 Detectability and isolability 27
4.3 Methodology 28
4.3.1 SIMULINK® 28
4.3.2 The data 29
4.3.3 Neural network architectures 31
4.3.3.1 Three-network topology 32
4.3.3.2 Single-network topology 32
4.4 Results 32
4.4.1 Three-network topology 32
4.4.2 Single-network topology 35
4.4.3 Multi-input single-output system 37
4.4.4 Noisy data 38
4.4.5 Quantised data 39
4.4 Conclusion 40

5 INDUCTION MACHINE BENCHMARK 42
5.1 Purpose 43
5.2 Background 43
5.3 Methodology 46
5.3.1 Simulation 46
5.3.2 Training data 47
5.3.3 Neural network topology 49
5.4 Results 50
5.5 Conclusion 53
6 FLOWNET MODEL OF THE BRAYTON CYCLE 54
6.1 Background 55
6.1.1 Introduction 55
6.1.2 Theory 55
6.1.2.1 The core 56
6.1.2.2 The compressors 57
6.1.2.3 The turbines 59
6.1.2.3 The heat exchanger. 60
6.1.2.5 Flownet 61
6.2 Methodology 62
6.2.1 Modelling 62
6.2.1.1 Brayton cycle 62
6.2.1.2 FDD system 63
6.2.1.2 Modelling results 65
6.2.2 Classification 67
6.2.2.1 Concept 67
6.2.2.2 Classification of data 67
6.2.2.3 Classification results 68
6.2.3 Decision logic 72
6.2.3.1 Methodology 72
6.2.3.2 Applied logic 73
6.2.3.3 Integrated concept 74
6.2.3.4 Decision logic results 75
6.3 Measured data 76
6.4 Conclusion 77
7 CONCLUSION 78
7.1 Conclusion 79
8 APPENDIX 81
REFERENCES 87

List of Figures

Figure 1.1: The Brayton cycle as used in the pebble bed modular reactor....3

Figure 2.1: Diagnostic family tree....8
Figure 2.2: Schematic representation of an FDD system....9
Figure 2.3: A generic fault detection method....10
Figure 2.4: PRBFN architecture....13
Figure 2.5: Fault detection by using Kalman filters....15

Figure 3.1: An actual neuron....17
Figure 3.2: A mathematical equivalent of the neuron....18
Figure 3.3: A multilayer feed-forward network architecture....20
Figure 3.4: The tansig (left) and purelin (right) activation functions....21
Figure 3.5: The log-sigmoid activation function....21
Figure 3.6: Information window while training a MATLAB® neural network....22
Figure 3.7: Basic recurrent network architecture....24

Figure 4.1: The parallel processes in SIMULINK®....26
Figure 4.2: Input to the classification networks....27
Figure 4.3: The input to the SIMULINK model....29
Figure 4.4: Signals representing a bias (left) and gain (right) fault....30
Figure 4.5: Fault windows for bias (left) and gain (right)....30
Figure 4.6: Time performance of the different topologies....34
Figure 4.7: Accuracy of the different topologies....34
Figure 4.8: A radar plot of the cumulative errors over 8 runs....34
Figure 4.9: Time average plots of different topology groups....36
Figure 4.10: Cumulative error of the different topology groups....36
Figure 4.11: Three similar possibilities that increase classification complexity....37
Figure 4.12: Neural network uncertainty with similar residuals....38
Figure 4.13: The same network with no noise and 13 dB SNR noise....39
Figure 4.14: 2-bit quantized data....40

Figure 5.1: The rotor cage of a squirrel-cage induction machine....43
Figure 5.2: A SIMULINK® model of the induction machine....47
Figure 5.3: An input/output pair sample of the training data....48
Figure 5.4: A feed-forward neural network with time delays [input at t0, t1,..]....49
Figure 5.5: Same inputs (red) with different outputs (blue)....51
Figure 5.6: Neural network simulation with insufficient delays....51
Figure 5.7: The step response of the 30-20-1 network....52
Figure 5.8: The step response of the 12-10-1 network....53

Figure 6.1: Schematic representation of a compressor....58
Figure 6.2: T-s diagram for a compressor....58
Figure 6.3: Schematic representation of a turbine....59
Figure 6.4: T-s diagram for a turbine....60
Figure 6.5: Schematic representation of a heat exchanger....60
Figure 6.6: Flownet model schematic....61
Figure 6.7: Input/output samples....62

Figure 6.8: FDD concept on part of the Brayton cycle. 63
Figure 6.9: Three neural network layout combinations. 64
Figure 6.10: The modelling capabilities of these neural networks. 65
Figure 6.11: Illustration of insufficient weights error for this scenario 66
Figure 6.12: An example of sensor fault residuals. 68
Figure 6.13: Misclassifications of the first classification network, C1. 69
Figure 6.14: Output of a 2-layer topology using both training sets. 70
Figure 6.15: Classification network trained with both training sets. 71
Figure 6.16: MATLAB® GUI developed to integrate all the FDD concepts 74
Figure 6.17: An example of a sensor 1 and 3 fault with sensor 3 detected first 75
Figure 6.18: Load rejection test of the PBMM 76

Figure A.1: Schematic illustration of the multi-input single output model. 86

List of Tables

Table 3.1: Commonly used MATLAB® training functions. 20

Table 4.1: Possible outputs of the residual neural networks. 31
Table 4.2: The output target format for the single-network scenario. 31
Table 4.3: Performance comparison of the three-network topology. 33
Table 4.4: Performance comparisons of the single-network topology. 36
Table 4.5: Cumulative MSE for quantised test set. 39

Table 5.1: Symbol definitions. 45
Table 5.2: Induction machine parameter definitions. 45
Table 5.3: Complex networks are insufficient 52

Table 6.1: Symbol definitions for the Brayton-cycle model. 56
Table 6.2: Accuracy (MSE) and speed of the three training algorithms investigated. 65
Table 6.3: Effect of the extra training set on classification performance. 70
Table 6.4: Single sensor error truth table. 73
Table 6.5: Multiple sensor error logic table. 73

Table A.1: Performance data for the 3-network topology 82
Table A.2: Performance data for the single-network 2-layer architecture. 83
Table A.3: Performance data for the single-network 3-layer architecture. 85
Table A.4: Performance with noise added for the three-network 6-3 architecture. 85

List of Abbreviations

FDD	Fault detection and diagnosis
GUI	Graphic user interface
HTR	High temperature reactor
MISO	Multi-input single-output
MNN	Multilayer neural network
MSE	Mean-square-error
OLS	Orthogonal least squares
PBMM	Pebble bed micro model
PBMR	Pebble bed modular reactor
PDF	Probability density function
PRBFN	Probabilistic radial basis function network
RBF	Radial basis function
SISO	Single-input single-output
SNR	Signal to noise ratio

List of symbols

Name	Description	Units
	Chapter 3	
u, x	System input related variables	-
y	System output related variable	-
w	Neural network connection weights or model parameters	-
ξ	Centre vector	-
d	Actual output of a parameter	-
X_i , x	State space variables	-
e	Error or residual	-
a	Activation functions	-
A, B, C	Constant matrices	-
K_i	i-th kalman filter gain	-
	Chapter 5	
v_a, v_b, v_c	Stator phase-voltages	V
i_a, i_b, i_c	Stator phase-currents	A
$\vec{v}_s(t)$	Stator voltage space-vector	V
$\vec{i}_s(t)$, $\vec{i}_r(t)$	Stator and rotor current space-vectors	A
v_{sd}, v_{sq}	Stator voltage vector components in rotational reference frame	V
$v_{s\alpha}, v_{s\beta}$	Stator voltage vector components in stator oriented frame	V
i_{sd}, i_{sq}	Stator current vector components in rotational reference frame	A
$i_{s\alpha}, i_{s\beta}$	Stator current vector components in stator oriented frame	A
i_{rd}, i_{rq}	Rotor current vector components in rotational reference frame	A
$\vec{\lambda}_s(t), \vec{\lambda}_r(t)$	Stator and rotor flux space-vectors	
$\lambda_{sd}, \lambda_{sq}$	Stator flux vector components in rotational reference frame	
$\lambda_{rd}, \lambda_{rq}$	Rotor flux vector components in rotational reference frame	
ω_d	Synchronous angular speed	
ω_{dA}	Slip angular speed	
ω_m	Rotor (mechanical) angular speed	
$\theta_d, \theta_{dA}, \theta_m$	Angles in radiance, corresponding to the above speed types	

Name	Description	Units
	Chapter 5	
P_n	Motor rated power	W
I_n	Rated phase current (rms value)	A
V_n	Rated phase voltage (rms value)	V
P	Number of pole-pairs	-
f_n	Rated frequency	Hz
T_n	Rated torque	Nm
T_L	Load torque	Nm
J_{eq}	Moment of inertia (equivalent)	Kg m^2
R_s	Stator resistance	Ω
R_r	Rotor resistance	Ω
L_m	Magnetizing inductance	H
L_{ls}	Stator leakage inductance	H
L_{lr}	Rotor leakage inductance	H
$L_s=L_{ls}+L_m$	Stator equivalent inductance	H
$L_r=L_{lr}+L_m$	Rotor equivalent inductance	H
	Chapter 6	
Q_n	Reaction power	Watt
m, m_a, m_b	Mass flow rate	kg/s
c_p, c_a, c_b	Specific heat capacitance of fluid at constant pressure	J/kgK
T_1, T_{a1}, T_{b1}	Inflow temperature	K
T_2, T_{a2}, T_{b2}	Outflow temperature	K
T_a	Ambient temperature	K
R_a	Thermal resistance to ambient	W/K
P_1	Inflow pressure	Pa
P_2	Outflow pressure	Pa
C, C_a,, C_b	Thermal capacitance	K/J

1 INTRODUCTION

A compact background discussion of the motivation for this research is given. The problem that is addressed in this paper is stated, followed by a proposed solution. Specific problems are highlighted. A methodological approach is proposed followed by an overview of the issues addressed in the different chapters.

1.1 Background

One of the two major factors forcing us to look at preventive maintenance is the safety of humans. Should an explosion occur due to a fault in a process, severe injuries or even fatalities could be sustained. To prevent such a scenario is of the utmost importance. The second major factor that contributes to the necessity for predictive maintenance is the financial factor. If the incipient failure of a component were detected in time, this component could be replaced during a planned shut-down. Catastrophic failure could cause failure of other components, depending on the severity of the consequential damage. By using fault detection and diagnostics (FDD), unscheduled down time could be reduced as well as damage to other components during failure. The financial implications are also of paramount importance, as Barringer and Woodrow (2002) point out that failures represent loss of money.

1.2 Problem statement

The above-mentioned factors stress the necessity of being able to predict failure of components or a faulty process status. The aim of this thesis is to investigate a fault detection and diagnostic (FDD) concept that could detect and recognise faulty behaviour of a process. A few known processes will be investigated before progressing to the Brayton thermal energy conversion cycle. The properties of neural networks will be put to the test during the investigation of an FDD system. For the diagnostic part, the classification capabilities are tested while the modelling capabilities of neural networks are tested in the detection part.

Detection of faults must ideally occur early enough so that actual failure of the process could be prevented. The Brayton cycle is a thermal energy-conversion process that is used in industrial applications to convert thermal energy to mechanical energy. The Brayton cycle is also implemented in the Pebble bed modular reactor (PBMR) nuclear power plant concept. See figure 1.1 (http://www.pbmr.co.za/2_about_the_pbmr/2_3how_it_works.htm). Since the PBMR has not yet been built, data is obtained from a simulation of the plant in a software package called Flownet where a model of the proposed plant was developed.

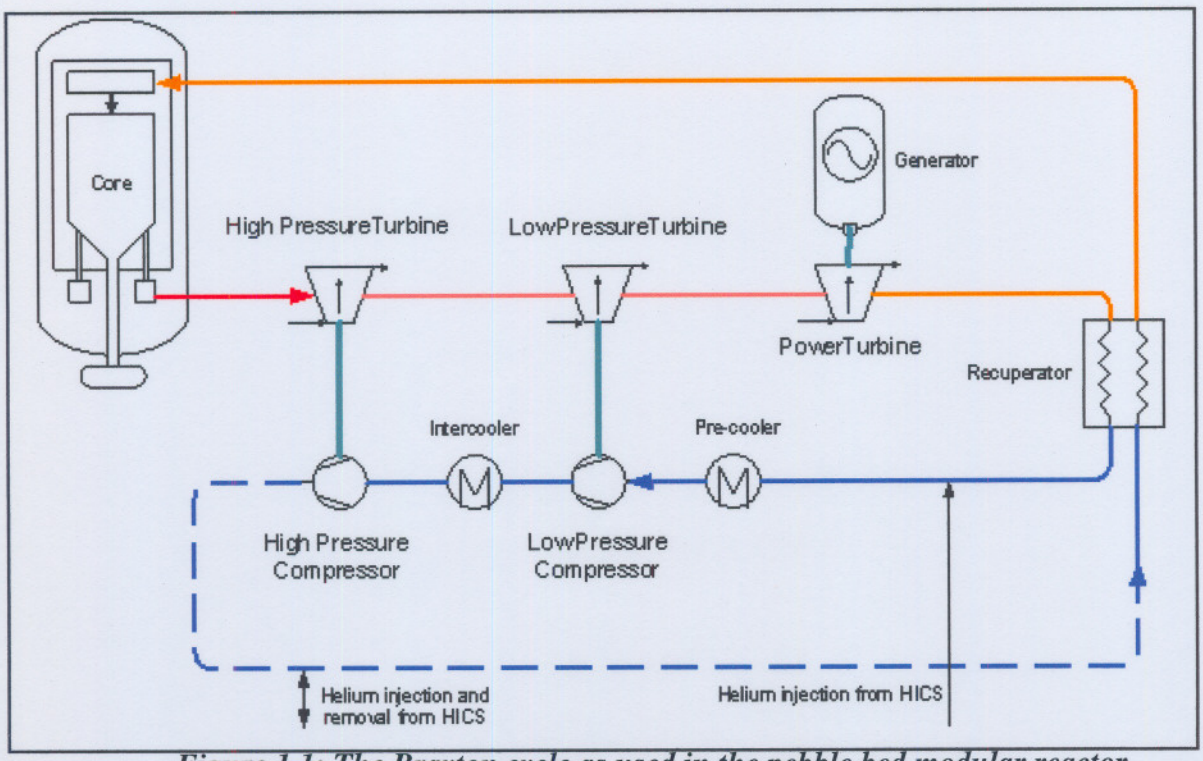

Figure 1.1: The Brayton cycle as used in the pebble bed modular reactor.

1.3 Proposed solution

A neural network, of which a discussion is given later, is implemented to firstly model the process response and secondly, by using another neural network, to classify identified errors in the process. The principles of fault detection and diagnostics will be discussed in Chapter 2. MATLAB® has a neural network toolbox that simplifies the process of investigating the different neural network topologies. The feasibility of the FDD concepts are investigated by implementing it on benchmark models to gain an understanding of the underlying principles. The general conclusions obtained from the simulations are then applied to a model of the PBMR.

1.4 Specific problems

-Diagnosing faults by using neural networks to classify residuals;

-modelling a process using neural networks for fault detection;

-Integrating the above mentioned solutions;

-Successfully implementing the FDD concept on the simulated pebble bed micromodel, PBMM.

1.5 Methodology

A major part in the development of the FDD will be the investigation of neural networks. These networks must be able to identify and classify faulty behaviour in the response data. Although multiple errors are discussed the focus is mainly on single errors. A discussion of neural networks is given in Chapter 3. MATLAB® will be used to develop this system because it has a neural network toolbox that makes development relatively easy. Finding the neural network topology and learning algorithm that generate the best results will be a major part of the work. Published research results will be consulted to find the appropriate neural network topology.

Data needed to train the networks will be obtained by means of simulation. This data could be manipulated to gain the most efficient results. To establish the tendencies of the data one is working within, might prove to be equally important.

When the concepts have been developed on the benchmark processes and the experimental results prove to be satisfactory, the Flownet model's simulated outputs will be used to train the network. The benchmark processes in this case are transfer function models and a model of an induction machine.

1.6 Research overview

Chapter 2: Background theory of fault detection and diagnostic systems and a literature survey of what has already been done.

Chapter 3: Theory of neural networks as applicable to this investigation.

Chapter 4: Investigation of neural networks as fault identifiers. Residuals are obtained by implementing two transfer function models that run parallel, one without errors and the other with errors. These transfer function models are implemented in SIMULINK®.

Chapter 5: Modelling a dynamic process with neural networks. An induction machine was chosen to serve as a benchmark for a dynamic non-linear system. The model was developed in SIMULINK and the neural network toolbox of MATLAB® is used to implement the neural network. This is a known process with some well-known faults and symptoms.

Chapter 6: Implementing the integrated concept on a FLOWNET model of the PBMM.

Chapter 7: A final conclusion on this research is given.

2 FAULT DIAGNOSTICS AND RELATED RESEARCH

A background is given on fault diagnosis and diagnostic methods. The difficulties that are faced when developing a fault detection and diagnostic system are discussed. An overview is given of some related research that has been done.

2.1 Fault diagnosis

The need to detect and diagnose faulty behaviour in a process, especially in its early stages, has been mentioned. A brief discussion of some practical challenges and some attributes of a diagnostic system are now given.

Process dynamics, lack of adequate models, incomplete and uncertain data and diverse sources of knowledge are amongst others, factors that make predictive fault detection and diagnosis (FDD) challenging (Dash & Venkatasubramanian, 2000). In industry the wide variety of processes leads to an equally wide variety of control systems that exist. This also applies to fault diagnostic systems.

For a diagnostic system to be an effective tool, such a system should have certain attributes. Foremost is the fact that the system should be able to detect and diagnose the fault as *early* as possible to be effective. When the system detects a faulty situation, it should be able to *discriminate* between various possible failures. However, the degree of *isolability* must not be so high that the system sees modelling uncertainties as faulty behaviour. The *robustness* of the system to noise and uncertainties will allow the system performance to *degrade gradually*, instead of falling away suddenly.

An important attribute is an explanation facility that would give possible origins and causes of the identified process faults. This facility would need a very descriptive set of historical data or experienced know-how. When the process is updated or changed the system should be able to adapt to. These changes should include minor changes, such as changes in the environment.

2.2 Diagnostic Methods

There are various diagnostic methods from which to choose. However, it is not that simple: an application might only allow certain types of methods. In figure 2.1 (Dash & Venkatasubramanian, 2000) the different methods are shown in a schematic diagram to illustrate the separate directions one can follow. Combining different concepts is sometimes necessary to get a robust and generalising system. The two main differences between these methods relate to methods that use process model information and those that use process history data.

With the process model-based method a fundamental understanding of the process is needed. This is used to describe the interactions of various process variables and input and output relations. The process history-based method uses, as the name implies, historical data that has been collected from an existing process. Each of these methods is then again divided into quantitative and qualitative methods as shown in figure 2.1.

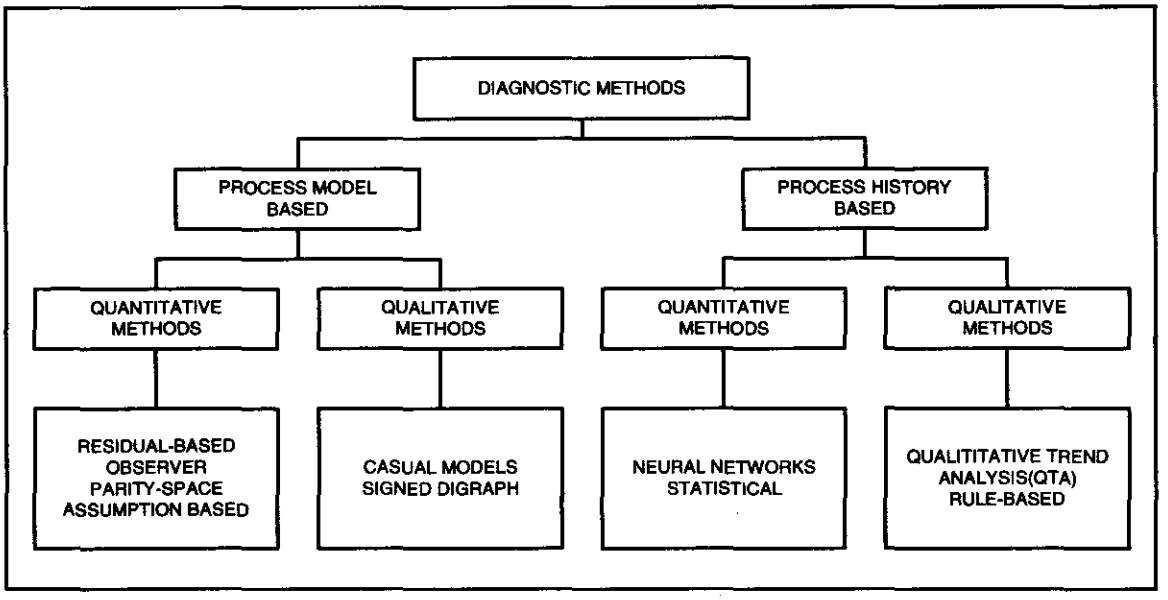

Figure 2.1: Diagnostic family tree.

Neural networks are shown as process history-based quantitative methods. In this investigation the process history will be obtained by simulation, since the physical plant has not yet been built. This investigation also shows whether MATLAB® and its neural network toolbox will suffice to provide the core of a diagnostic system.

2.3 Some existing research

Although there are FDD systems that use statistics and methods other than neural networks, the emphasis in this work is on neural networks. The reason for the popularity of neural networks is the generalising ability for non-distinct behaviour of a system (Fuente & Vega, 1999). A few applications of interest are discussed below.

2.3.1 The radial basis function neural networks (RBF)

Chen and Lee (2001) use radial basis function (RBF) networks for online state estimation, trained with the orthogonal least squares (OLS) algorithm. To train for detection and diagnosis, they use the back-propagation algorithm. They investigated fault detection and diagnosis on an airplane so that the flight control system could monitor the plane. When the system detects a fault on a component, the flight control system reacts appropriately to prevent an accident.

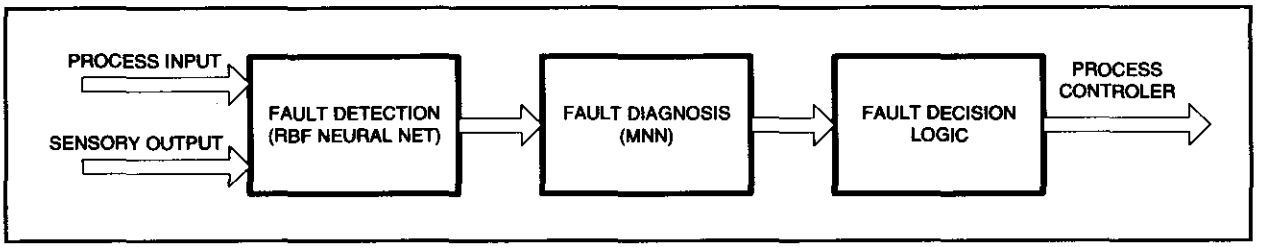

Figure 2.2: Schematic representation of an FDD system.

Figure 2.2 illustrates how they implemented a fault detection and diagnostic system. The two major parts of their system are *detecting* a fault by using the RBF network and the *classification* of the fault by using the multilayer neural network (MNN). The RBF uses less computation time (Moody & Darken, 1989) during training than topologies like the MNN and is also a more compact architecture (Lee & Kil, 1991).

Apart from that, RBF networks are highly suited for non-linear approximation. Using data that is taken from the system itself or a simulation of that system, the RBF network is trained to act like the system. A model of the process that is as realistic as possible is used to generate data for fault scenarios that are not available in the real process.

The RBF network approach in representing time-varying non-linear dynamic system is to use a combination of system input and output with some time-delay units. It is assumed that a time-varying non-linear dynamic system is described by:

$$y(k) = f(y(k-1),..., y(k-n_y), u(k-d-1),..., u(k-d-n_u)) \qquad (2.1)$$

with f an unknown non-linear function, u and y are the system's input and output and n_y, n_u and d represents time delays and orders of the model respectively. By using an RBF network that has one hidden layer and one output layer, its response can be expressed as:

$$\hat{y} = f(x) = \sum_{\alpha} w_{\alpha} \phi(\| x - \xi_{\alpha} \|) \qquad (2.2)$$

Here ϕ can be selected from many radial functions and in general this radial function is chosen as the Gaussian function. ξ_{α} is the center vector of the α-th hidden unit and w_{α} is the connection weight between the α-th hidden unit and one output layer. This RBF network becomes a function approximation problem of the type

$$\hat{y}[k] = f(x[k], n) \qquad (2.3)$$

that will simulate the system.

The detection component of the FDD system generates residuals by comparing the actual system variables with those simulated by the RBF network. If no faulty behaviour occurs, the residuals that are generated will originate from noise and system disturbances. There is a threshold that must be exeeded before a residual qualifies to indicate a fault in the system. This threshold should be chosen so that it will not wrongly mistake noise for a failure, or such that only extreme faults are detected with many misses on less severe faults.

Residuals generated by the system are stored in a database and serve as training data. The multilayer perceptron neural network implements the back-propagation learning algorithm for learning to classify these residuals. The aim is to classify the faults with a certain degree of accuracy.

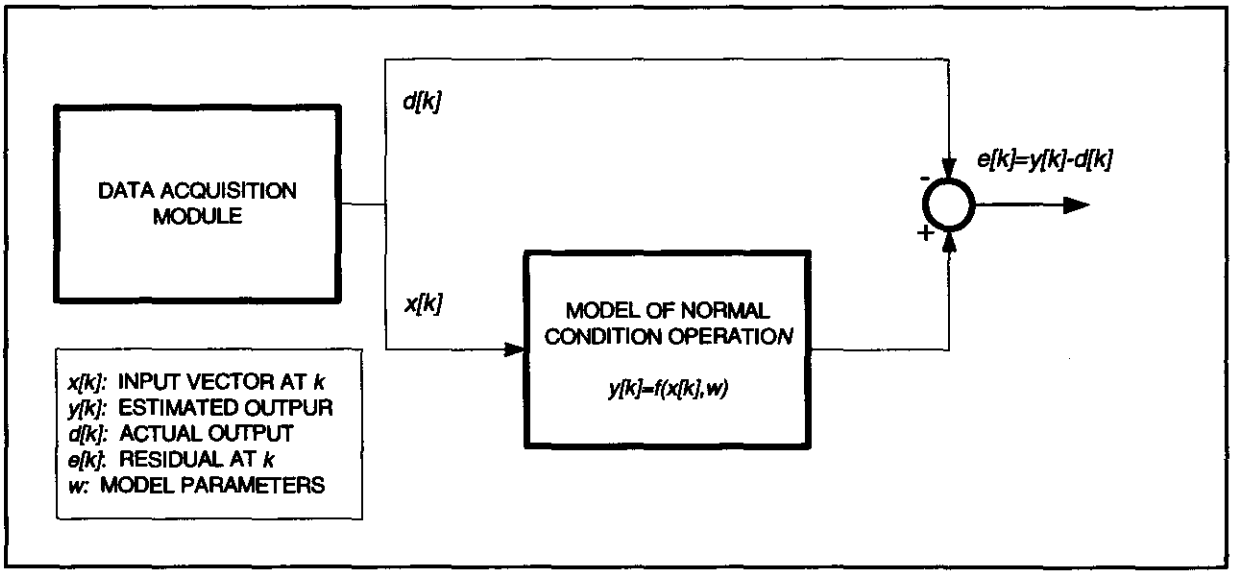

Figure 2.3: A generic fault detection method.

2.3.2 The probabilistic radial basis function network (PRBFN)

The detection method investigated by Munoz and Sanz-Bobi (1998) is discussed. This method is illustrated in figure 2.3. This fault detection method is based on a connectionist characterisation of the normal behaviour of a system or subsystem. A probabilistic radial basis function network (PRBFN) is used as a function approximator in the form of a dynamic black box model. Analytical redundancy is applied for this characterisation. The definition of the reliable domain of the model generally improves model-based fault detection methods. This means that by restricting the residuals generated within practical boundaries, uncertainties decrease.

Limit checking of individual plant variables is another method used for fault detection. However, the thresholds must be set conservatively so that all the normal conditions of the plant are covered. Fault isolation becomes intricate because a single fault may cause many plant variables to exceed their limits. Here pattern recognition was chosen for their flexibility (Munoz & Sanz-Bobi, 1998).

There are pattern-recognition techniques such as Bayes classifiers (Rengaswamy & Venkatasubramanian 2000) or Bayes' classification by using the K-nearest neighbour algorithm (http://www.eece.unm.edu/controls/papers/ACC97-paper1.pdf) that is not neural network based. However, as mentioned before, the generalising ability of neural networks has made them popular. The generation of residuals to classify process faults, predominate recent literature.

When using state equation or transfer functions for simulation models, there are three different strategies for residual generation: parity equations, the diagnostic observer and the Kalman filter. Keep in mind that transfer functions only apply to linear systems. Firstly the variables $\{X_1,X_2,...,X_{n'+1}\}$ are said to be representative of the state of the component, if under normal operation it is possible to define a set of parity relations of the form

$$G_i(X_1, X_2, ..., X_{n'+1}) = 0, \ i = 1, ..., m, \qquad (2.4)$$

which shows the relations between the lags of the state variables X_j. When a fault is detected at least one of the equations will not be satisfied. By selecting one variable of each G_i as the output variable of the model, the dynamic system can be modelled. Assume that the parity relations G_i ($i=1,...,m$) can be expressed as

$$d_i[k] = g_i(d_i^{\{k-1\}}, \boldsymbol{u}_i^{\{k\}}, \varepsilon_i^{\{k-1\}}) + \varepsilon_i[k] \qquad (2.5)$$

where $d_i[k] \in \Re$ is the value of the selected output variable for the ith parity equation at time k (consider as present time), $d_i^{[k-1]}$ the vector containing lagged or past values of $d_i[k]$, $\boldsymbol{u}_i^{[k]}$ is a vector containing present and past values of the remaining n input state variables, and $\varepsilon_i[k]$ is a white-noise process.

If $d_i[k]$ can be estimated by an unbiased non-linear model

$$y_i[k] = f_i(d_i^{[k-1]}, \boldsymbol{u}_i^{[k]}, e_i^{[k-1]}) \tag{2.6}$$

then the estimation errors

$$e_i[k] = d_i[k] - y_i[k], \ i = 1,...,m \tag{2.7}$$

are the residuals that one uses to diagnose the fault (Munoz & Sanz-Bobi, 1998).

Another important issue is what a reliable domain of the model would be. The black box model is realised by fitting a function approximator to a set of the input/output relations to act as training data. The reliable area would then be inside the input space of $X \subset \Re^n$. This is called the reliable domain of the model. The PRBFN gives an estimation of the probability density function (pdf), $p_x[k]$ of the input vector $\boldsymbol{x}[k]$. This pdf of $\boldsymbol{x}[k]$ is a good representation of the environment of the training set and thus a good characterisation of the residual $e[k]$. If $p_x[k]$ is low, the reliability of the estimated $e[k]$ is also low.

When to take notice of a residual is another question that needs to be answered. A proposal is to estimate the standard deviation, $s_e[k]$ of the residual as a function of the input vector (Munoz & Sanz-Bobi, 1998). A residual is significantly high if its absolute value exceeds the residual upper bound

$$e_{\max}[k] = 2 \cdot s_e[k] \tag{2.8}$$

The structure of the PRBFN is illustrated in figure 2.4 (Munoz & Sanz-Bobi, 1998). The output could take on one of two forms. For a given input vector $x \in \Re^n$ with its desired output $d \in \Re$, a joint pdf type of expression is obtained given by equation 2.9 (Munoz & Sanz-Bobi, 1998).

$$p(\boldsymbol{x}, d) = \frac{1}{h}\sum_{k=0}^{h}\left[\left(\prod_{k=0}^{n}\frac{1}{\sqrt{2\pi\,|\,\sigma_{ik}\,|}}\right)\exp\left(-\sum_{k=1}^{n}\frac{1}{2\sigma_{ik}^2}(x_k - r_{ik})^2\right)\right] \times \exp\left(-\frac{1}{2\sigma_{i0}^2}(d - v_i)^2\right) \tag{2.9}$$

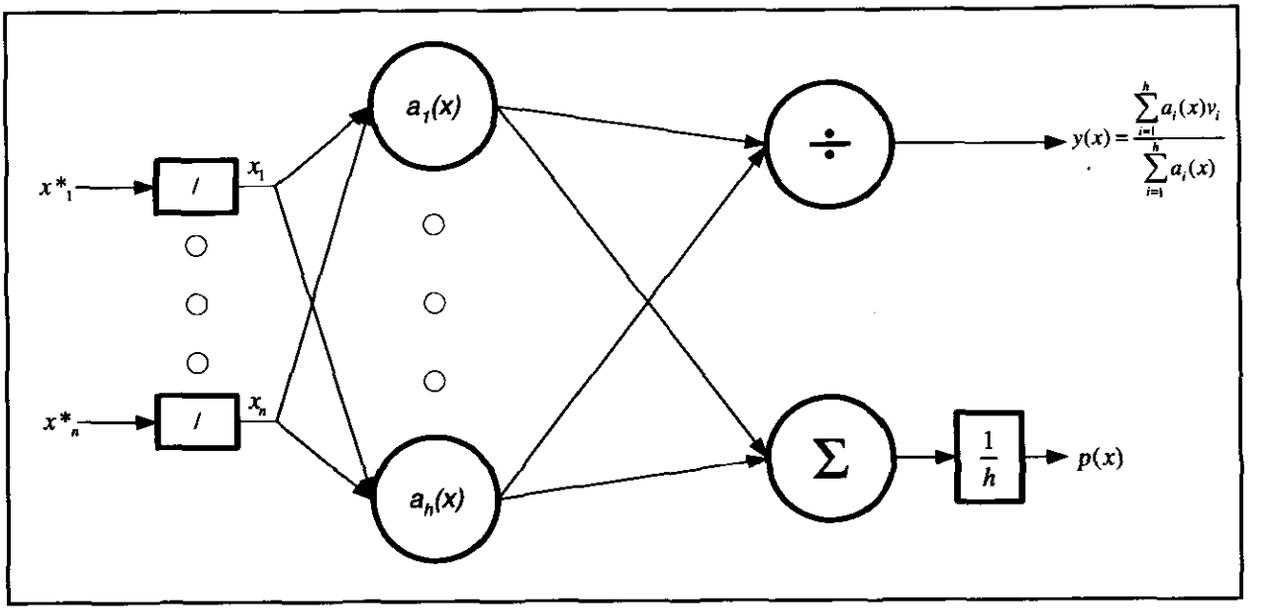

Figure 2.4: PRBFN architecture.

Assume that $p(\boldsymbol{x},d)$ is an estimation of the underlying joint pdf. This estimator can be used as a function approximator of the input/output mapping $x \mapsto d$ of the system. By applying the general regression principle it is seen in figure 2.4 that

$$y(\boldsymbol{x}) = E\left(\frac{d}{\boldsymbol{x}}\right) = \frac{\int_{-\infty}^{+\infty} d \cdot p(\boldsymbol{x},d)\mathrm{d}d}{\int_{-\infty}^{+\infty} p(\boldsymbol{x},d)\mathrm{d}d} = \frac{\sum_{i=1}^{h} a_i(\boldsymbol{x})v_i}{\sum_{i=1}^{h} a_i(\boldsymbol{x})}. \tag{2.10}$$

The activations a_i are given by

$$a(\boldsymbol{x}) = \left(\prod_{k=1}^{n} \frac{1}{\sqrt{2\pi}\,|\sigma_{ik}|}\right) \exp\left(-\sum_{k=1}^{n} \frac{1}{2\sigma_{ik}^2}(x - r_{ik})^2\right) \tag{2.11}$$

and all this can be structured as a two-layer neural network that gives rise to the PRBFN in figure 2.4. This network can be trained either as a function approximator or an estimator of the pdf of the input vector or it can be trained for both these functions, depending on the learning algorithm. A low-memory quasi-Newton method and cross validation can be used to optimise this procedure (Munoz & Sanz-Bobi, 1998).

2.3.3 Kalman filters

Simani and Fantuzzi (2000) use an FDD system similar to that used in section 2.3.2. Neural networks are used to classify the residuals that are generated from the sensors. The process is assumed to be described by a discrete time, time-invariant linear dynamic model of the type

$$\begin{aligned} x(t+1) &= Ax(t) + B\hat{u}(t), \\ \hat{y}(t) &= Cx(t), \quad t = 1,2,..., \end{aligned} \tag{2.12}$$

where $x(t) \in \Re^n$ is the state vector and the output vector is given by $\hat{y}(t) \in \Re^m$ with the control input vector $\hat{u}(t) \in \Re^r$. The constant matrices A, B and C are obtained by modelling techniques or detection procedures. The actual measurements would then be given by

$$\begin{aligned} u(t) &= \hat{u}(t) + \tilde{u}(t) + f_u(t), \\ y(t) &= \hat{y}(t) + \tilde{y}(t) + f_y(t), \end{aligned} \tag{2.13}$$

Here $\tilde{u}(t)$ and $\tilde{y}(t)$ represent the noise in the sensors that are normally modelled as white-noise Gaussian processes. The last term in each of these equations, $f_u(t)$ and $f_y(t)$, are signals that assume non-zero values when faults are present in the process.

Kalman filters are used to estimate signals that are compared to the actual signals from the sensors. Figure 2.5 (Simani & Fantuzzi, 2000) illustrates the method of using classical Kalman filters to generate residuals. Take notice that each of the *m* output sensors has its own filter, which simplifies the detection of faults. This situation is, however, not the same for the input sensor values, because every filter receives them all.

The time-invariant, discrete time, linear dynamic system referred to earlier has for the *i*th Kalman filter the structure (Jazwinski, 1970)

$$\begin{aligned} x_F^i(t+1 \mid t) &= A(I - K_i(t)C_i)x_F^i(t \mid t-1) + Bu(t) + AK_i(t)y(t), \\ y_F^i(t \mid t) &= C_i(I - K_i(t)C_i)x_F^i(t \mid t-1) + C_iK_i(t)y(t). \end{aligned} \tag{2.14}$$

The one-step prediction of the *i*th component of the state $x(t)$ is given by $x_F^i(t+1 \mid t)$. The estimate of the *i*th component of the output, $y_i(t)$ is given by $y_F^i(t \mid t)$, while C_i is the *i*th row of the output distribution matrix *C*. Simani and Fantuzzi (2000) use a Riccati equation to compute the time-variant gain of the filter, $K_i(t)$, using the covariance matrix of the output vector noise ü(*t*) and the variance of the *i*th component of the output noise ÿ(*t*).

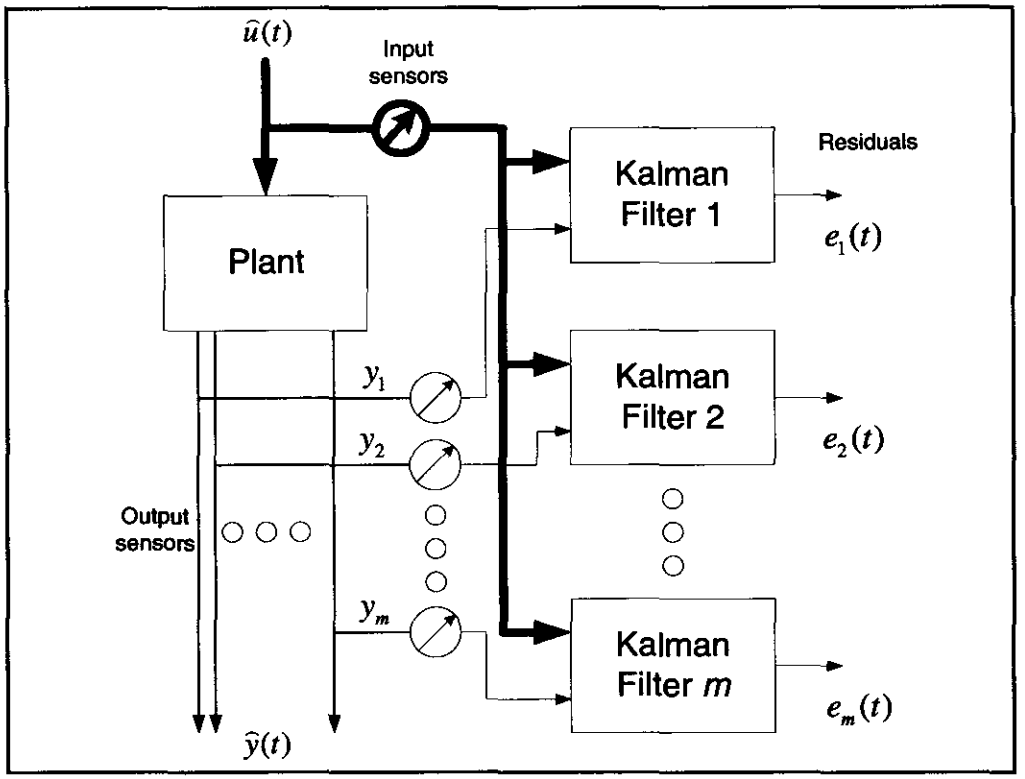

Figure 2.5: Fault detection by using Kalman filters.

The problem, however, is the fact that the process needs to be described with a discrete time, time-invariant dynamic model as described above. This is unnecessary for neural networks, which is another advantage. There are many way to implement the FDD concept, but the basic idea remains the same.

3 NEURAL NETWORK THEORY

The basic theory of neural networks is discussed with the emphasis on how to use the MATLAB® neural network toolbox.

3.1 Background

The basic building block of the brain is the neuron, figure 3.1 (http://faculty.washington.edu/chudler/cells.html). These neurons are connected in networks with their synaptic terminals connected to the dendrites of other neurons. A network can also have feedback; one example is when the synaptic terminal of a neuron is connected to its own dendrites. There are countless types of network architectures made of basic feed-forward and feed-back network topologies. These two basic networks are discussed later on.

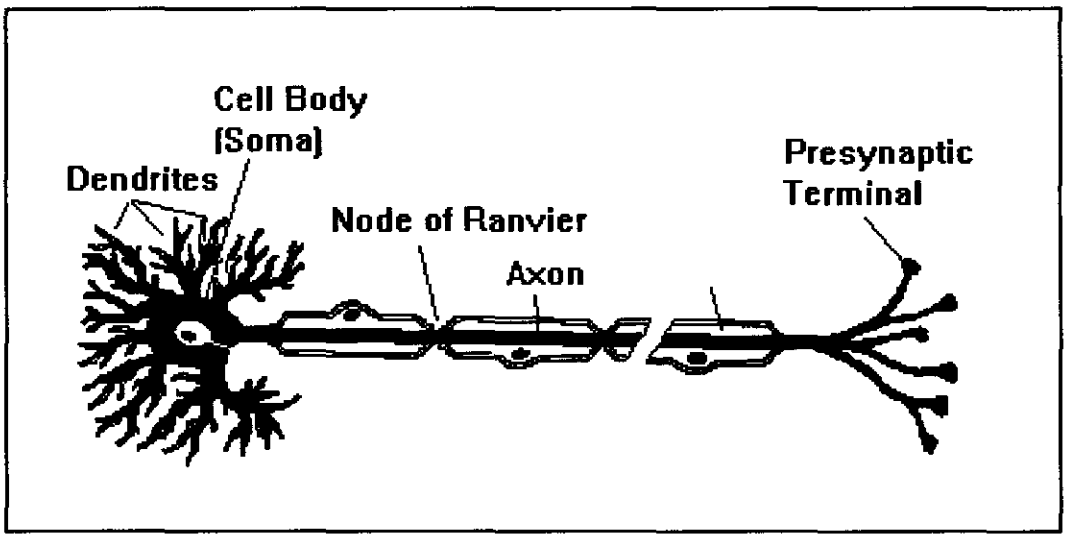

Figure 3.1: An actual neuron.

The connection strengths of the synapses lie in the chemistry of the synaptic terminal and these strengths are manipulated when the brain is learning something. The brain learns from inputs and responses. An example is when a child looks at an *a*, writes it down and tries to remember that it sounds like *a* (/e/), the desired output, to be able to use it later on. The knowledge is imbedded within the network structure via the synaptic connection strengths. The human brain is a highly parallel-operating system. This characteristic makes it ideal for recognising patterns like faces or any image for that matter almost immediately (Haykin, 1999).

Neural networks attempt to simulate some functions of the brain, which would have useful computational advantages. A more descriptive definition is given by Haykin (1999):

> *A neural network is a massively parallel-distributed processor consisting of simple processing units, which has a natural propensity for storing experimental knowledge and making it available for use. It resembles the brain in two respects:*

1. *Knowledge is acquired by the network from its environment through a learning process.*
2. *Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.*

The learning process is an algorithm and quite a number of these were developed, of which some will be discussed later on. The function of these algorithms is to change the synaptic weights between the neurons to integrate within the network, the knowledge it is being *fed.*

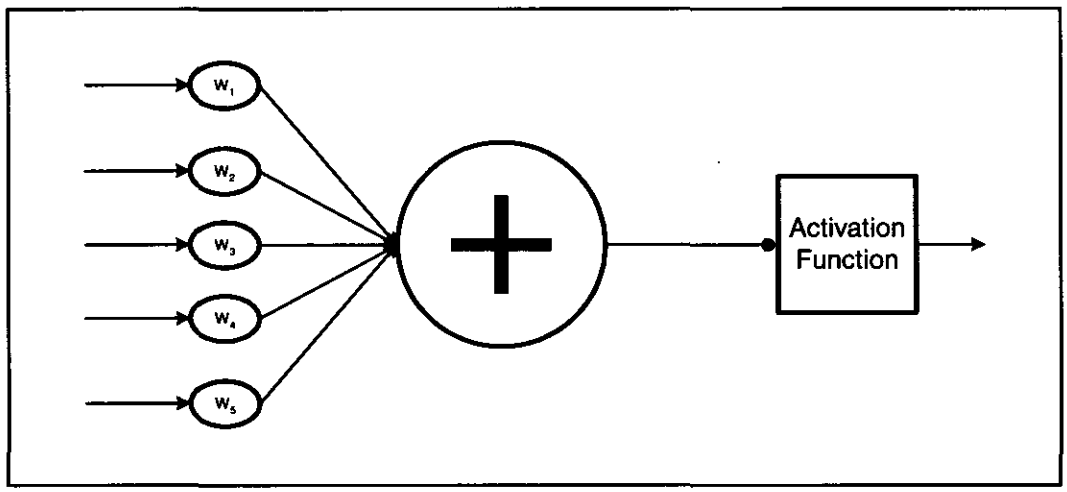

Figure 3.2: A mathematical equivalent of the neuron.

Figure 3.2 provides a mathematical representation of a neuron with the weights representing the synaptic connection strengths. In equation form:

$$Output = f(\sum_{n=1}^{5} i_n W_n), \qquad (3.1)$$

where f represents the activation function and i_n the input values. This is only a simplified representation of a real neuron found in the human brain. A bias, which acts like an input with a fixed value, may also be added. The bias has a weight that is adapted by the training algorithm.

The logic of the neural network, which is similar to that of the brain, differs from the basic logic that is implemented in computers. However, neural networks can be implemented on a higher level in a computer. For a human to recognise a face takes only a fraction of a second, but a computer with the correct software will take much longer. Pattern recognition, which is only one of many abilities that the neural network tries to copy from the human brain, is extremely time-consuming when using normal Boolean logic. When the neural network is implemented on a computer, the advantages gained from solving highly non-linear problems make it a viable option, however.

3.2 Network topologies

Certain network architectures tend to be more efficient with certain training or learning algorithms. A few topologies are mentioned below together with a discussion of the training algorithms. These algorithms are implemented in the MATLAB® neural network toolbox, since MATLAB® was used in this study to develop neural networks to solve the various problems.

3.2.1 Multilayer feed-forward networks

The MATLAB® function to create a feed-forward network, either single or multilayer, is *newff*. The application of this function will be explained in the following experiments. These networks implement neurons in layers (See figure 3.3). Each layer could have an arbitrary number of neurons. For more complex problems the number of neurons is increased to retain the *knowledge* that is needed. Inserting another layer of neurons may also enable the network to see more complex behaviour in its training data.

The number of times the training data is fed to the network, called the number of epochs, is also important to ensure that the network captures the behaviour of the data. There is a trade-off between the number of neurons, the number of layers and the time required to train the network. Furthermore, problems are sometimes better solved with a less complex network. Figure 3.3 illustrates the multilayer feed-forward network topology.

According to the MATLAB® helpfile on multilayer architectures, the multilayer neural network is quite powerful, since it states that a two-layer network can approximate any function to an acceptable degree of accuracy (Also refer to Haykin (1999:229)).

To illustrate how the multilayer network fits together, one can visualise the neuron of figure 3.2 in the place of each neuron in figure 3.3. The number of inputs and thus the weights depend on either the number of inputs for the first layer, or the number of neurons in the previous layer in the case of a hidden layer. The first layer is normally called the input layer and the following layers are called the hidden layers. The last layer is sometimes called the output layer.

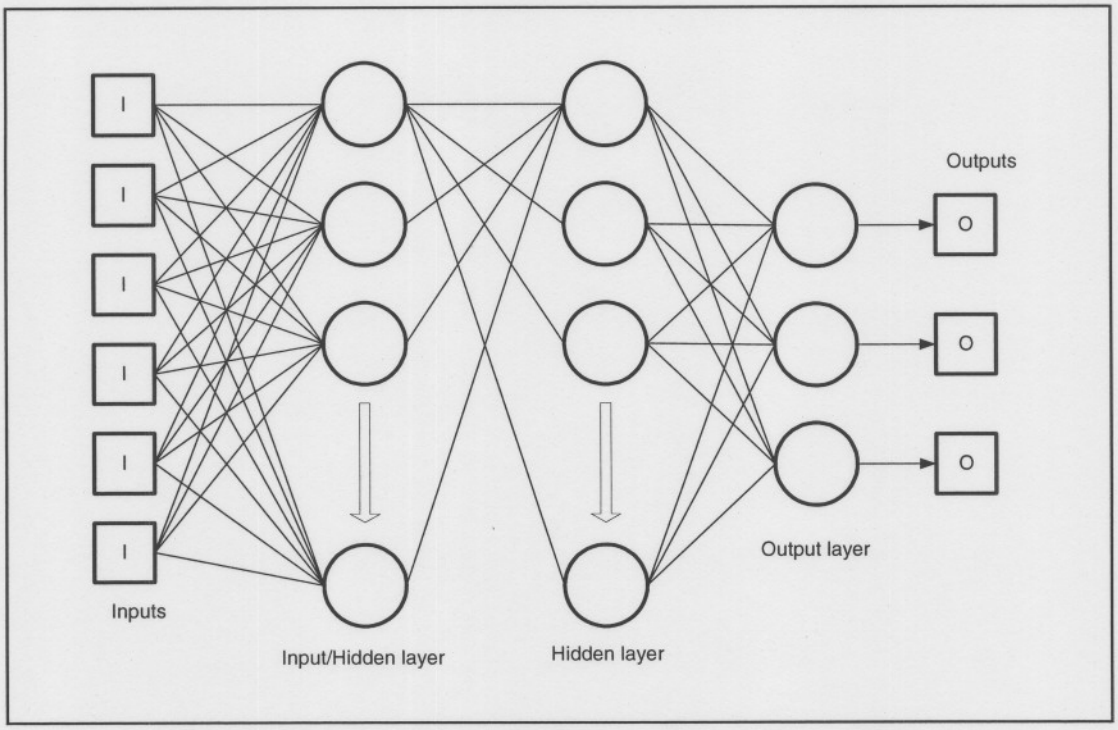

Figure 3.3: A multilayer feed-forward network architecture.

The connections in figure 3.3 between the first hidden layer and the second are not fully completed to keep the figure uncluttered. For the normal multilayer feed-forward network the neurons are fully interconnected.

Function	Description
Traingd	Gradient descent back-propagation
Traingdx	Gradient descent with momentum and adaptive learning rate back-propagation
Trainlm	Levenberg-Marquardt back-propagation
Trainrp	Resilient back-propagation
Trainbfg	BFGS quasi-Newton back-propagation

Table 3.1: Commonly used MATLAB® training functions.

Of all the available training algorithms in MATLAB®, the more popular ones are listed in table 3.1. These are different kinds of back-propagation algorithms. Back-propagation training is a very common, though effective method. It is seen in most of the literature on neural networks.

The purpose of the activation function is explained with reference to figure 3.4. After the inputs have been multiplied with their respective weights and added together, the results are fed into this activation function. Non-linear functions like the log-sigmoid function, *logsig*, and the tan-sigmoid function, *tansig*, allow the network to map non-linear patterns. The linear function *purelin*, is not fit for mapping non-linear patterns.

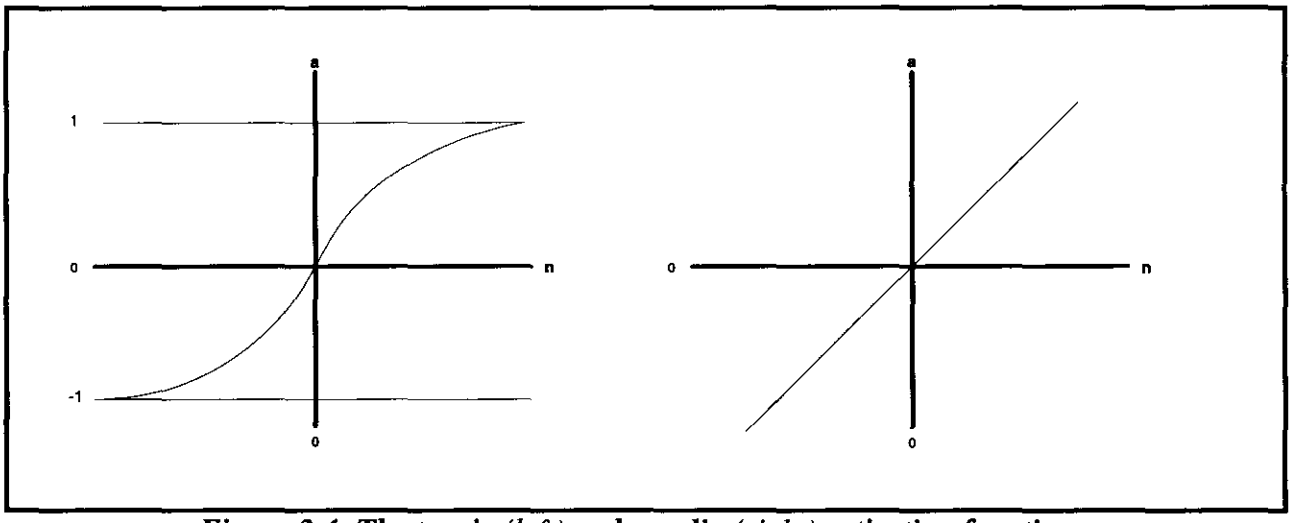

Figure 3.4: The tansig (left) and purelin (right) activation functions.

Throughout the experiments in this paper it was found that *trainrp* and *traingdx* were the best algorithms in terms of time and effectiveness. The *trainlm* algorithm was considered but as it takes much longer to train and uses much more memory, it was not used. MATLAB® also suggests that those algorithms listed should be applicable for multilayer networks.

Resilient back-propagation, *trainrp*, disposes of an inherent problem when using *logsig* activation functions in the network. The derivative between two large absolute values on the *n*-axis, figure 3.5, is very small. Algorithms using this derivative to adjust connection weights will thus take very long to train because this derivative is so small. Note that the sigmoid function allows only outputs between 0 and 1.

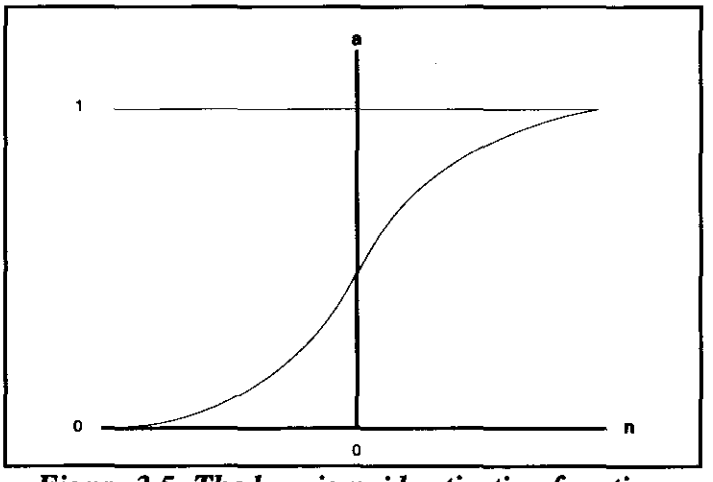

Figure 3.5: The log-sigmoid activation function.

Resilient back-propagation uses the derivative only to obtain the sign of the derivative. This shows to which side the weight should be adjusted. When the derivatives at two successive points have the same sign, the weight is incremented by *delt_inc* and is decremented by *delt_dec* when they have different signs. Should the sign change every other derivative, the value *delt_dec* is decreased to prevent the algorithm from oscillating. These parameters are inherent to this training algorithm. Of course, when the derivative is zero, the weight is left unchanged. If the signs of a few successive derivatives were the same, *delt_inc* is increased to speed up the process. The network will stop training when:

- The number of *epochs*, set by programmer, is reached;
- the maximum amount of *time* has expired;
- the performance has reached the predetermined *goal*;
- the gradient of the performance is the predetermined level, *mingrad*;.
- the "Stop Training" button is pressed (See figure 3.6).

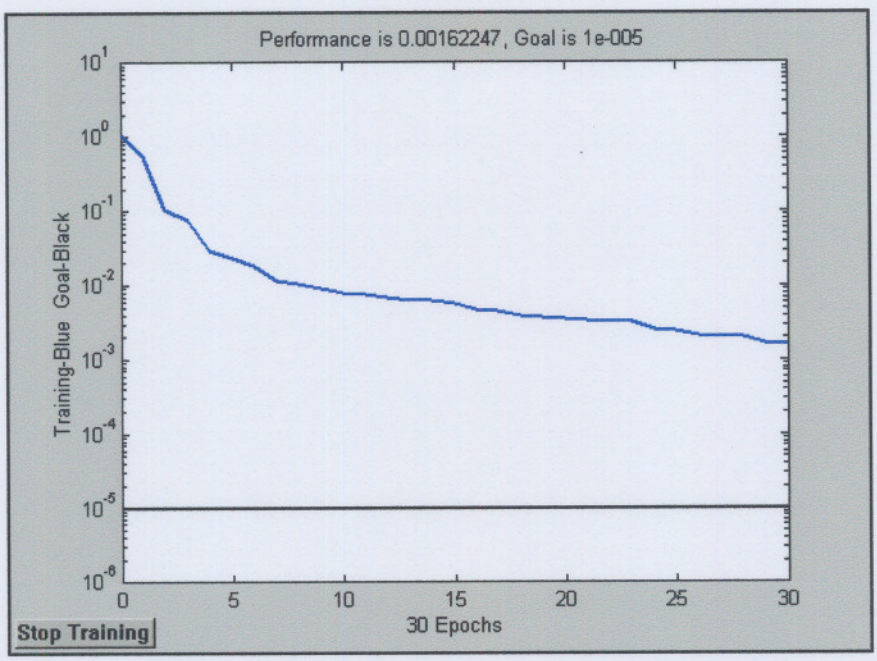

Figure 3.6: Information window while training a MATLAB® neural network.

In figure 3.6, the maximum number of epochs was set at 30. The performance *goal*, which is the mean-square-error between the network output and desired output, was set at 10^{-5}. When the gradient of the training performance is below *mingrad*, the algorithm knows that the performance curve will not reach the *goal*. That is at least within the set number of *epochs* or amount of *time* available. This applies to all the training algorithms.

Gradient descent with momentum and adaptive learning rate back-propagation or *traingdx* is another training algorithm that is often applied in this paper. Were it not for the momentum and adaptive learning rate, gradient descent would not have been attractive. It is very slow when the learning rate is too small, while it will oscillate if the learning rate is too big. The learning rate is simply the step size factor by which the weights are changed.

The momentum *mc* in equation 3.2 (MATLAB® helpfile) is multiplied with the previous change of weights *dXprev*, the learning rate *lr*, and the derivative of the performance *perf*. The performance is with respect to the weights and the bias weights if implemented

$$dX = mc \times dXprev + lr \times mc \times dperf/dX \qquad (3.2)$$

The momentum increases the adaptation of the weights when the derivative of the error surface has the same sign for a number of successive training steps. Note that if the momentum *mc* is too big the algorithm might end up oscillating as well.

The learning rate *lr* is adapted during training. If the performance decreases toward the goal, the learning rate of the following epoch is increased by the factor *lr_inc*. The performance is determined by the generated output's deviancy from the output. If the performance increases by more than the parameter *max_perf_inc*, the learning rate is adjusted by the parameter *lr_dec* and the change, which increased the performance, is ignored. The parameters printed in italics are preset in MATLAB® to optimal values, although these could be edited by the programmer.

3.2.2 Recurrent networks

A recurrent neural network only differs from a normal feed-forward network in that it has feedback. Figure 3.7 illustrates the basic architecture of a recurrent network. The MATLAB® neural network toolbox has two types of recurrent network topologies, Elman and Hopfield networks, of which only the Elman network is considered. The Hopfield network is a self-organising map topology.

The Elman neural network in the MATLAB neural net toolbox has a basic structure of two layers, one hidden layer and an output layer. Because of the recurrent connection, this recurrent network is able to detect and generate time-varying patterns (MATLAB® helpfile on recurrent networks). It

also recommends that when using the Elman network, one should use *traingdx* for the learning function and the *tansig-purelin* activation function combination for the two layers.

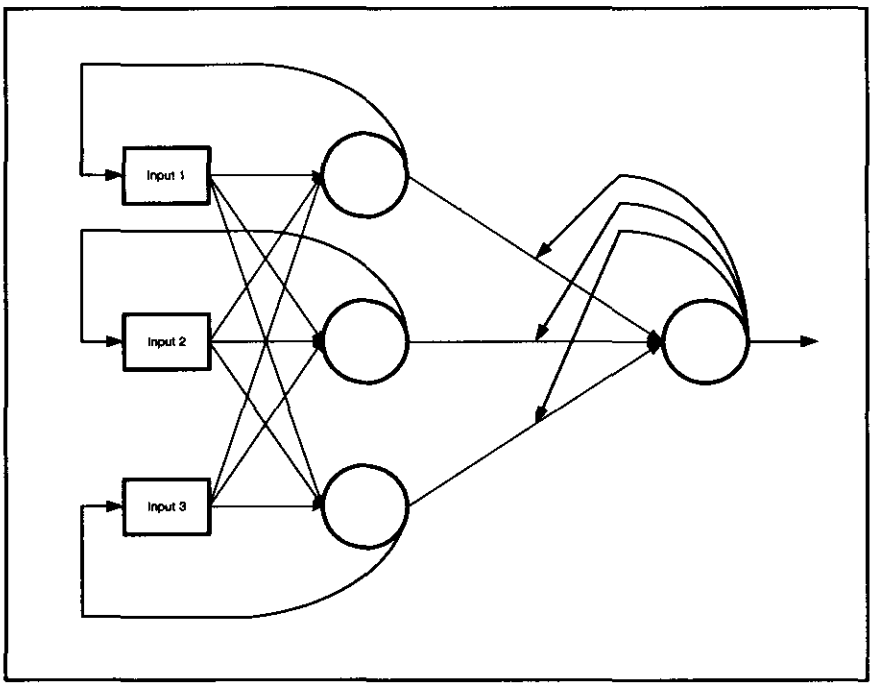

Figure 3.7: Basic recurrent network architecture.

4 RESIDUAL CLASSIFICATION

MATLAB SIMULINK® models are developed to model a plant from which residuals are generated. An investigation is done on normal, static, feed-forward neural networks for classifying these residuals. These networks are tested by adding noise to the residuals or quantising the residual data. These networks were developed on a single-input single-output and a multi-input single-output model.

4.1 Problem discussion

The idea of this phase is to investigate and test a ***classification*** neural network. This network will classify residuals from the model in figure 4.1. A static neural network, which is a neural network without time delays, will be used for the classification.

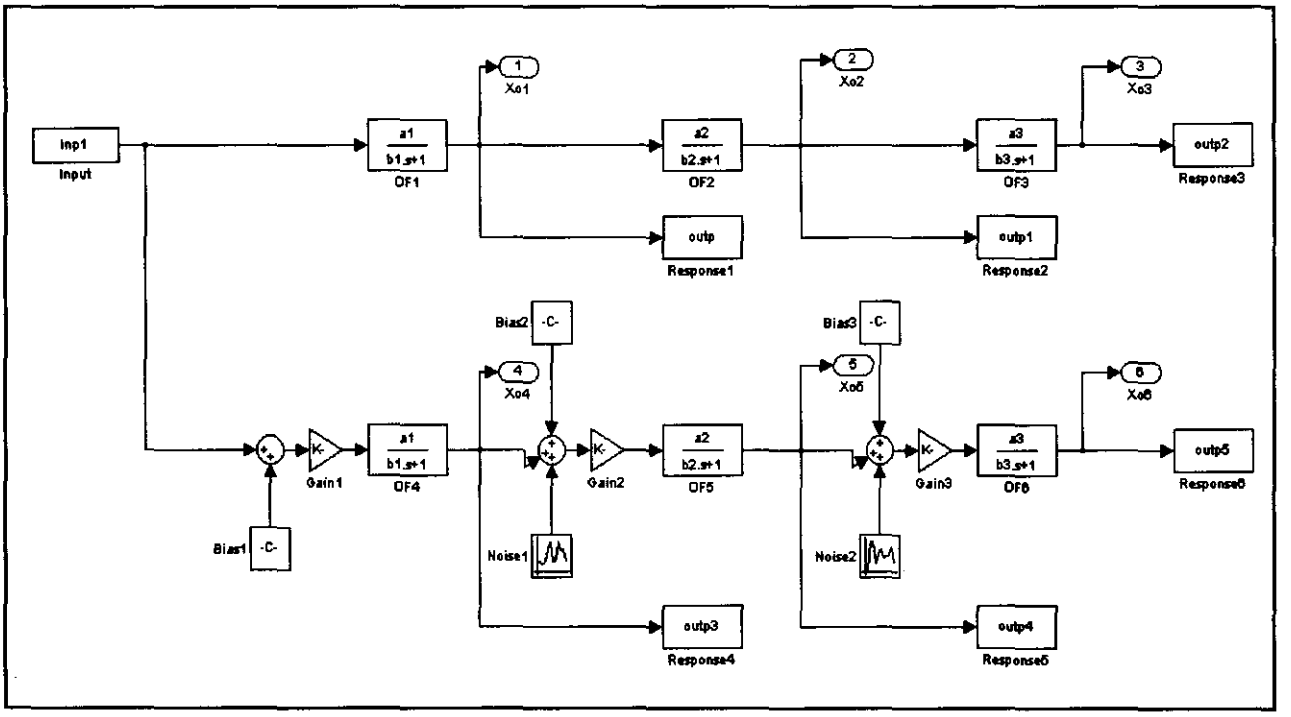

Figure 4.1: The parallel processes in SIMULINK®.

A SIMULINK® model consisting of two similar processes that are in parallel with each other is implemented. The only difference between these two processes is that in the lower process bias and gain errors can be inserted (Figure 4.1). The idea is to take measurements at corresponding points in these two systems. By subtracting these corresponding measurements one gets residuals that could be used to identify the fault that might be present in the process (Figure 4.2). These residuals can either be directly used or some statistical manipulation could be done beforehand. Conceptually, this FDD method could be implemented with the physical plant as the lower system and the reference model as the upper system.

The upper process in figure 4.1 models the system with no faults and can be replaced by a dynamic neural network that emulates the process. This dynamic neural network, which is a network with either feedback or time delays or both, is discussed in the next chapter.

The process at the bottom represents the actual plant where system faults can artificially be inserted. Note that for FDD systems one may use any suitable model.

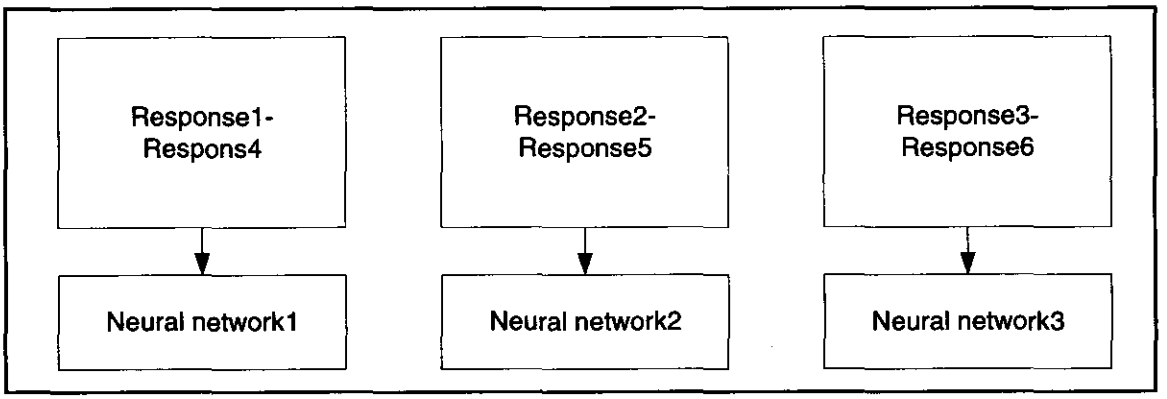

Figure 4.2: Input to the classification networks.

4.2 Detectability and isolability

Detectability is how easily a fault can be picked up or a significant residual can be generated. Isolability is how easily a fault can be distinguished from others. These concepts are important for the classification of residuals.

Detectability is highly dependent on the difference between the faulty and the non-faulty situation, while one can also look at the fault-to-noise ratio (Basseville, 1999). The isolability is again dependent on the information contained in the residual about that specific fault. When using statistical methods for an FDD system, these concepts play an important role in designing an effective system.

With neural networks, the detectability of the residuals depends on how accurately normal operation of the plant is modelled by the neural net. It follows that an inaccurate model may cause the difference between faulty and non-faulty operation to be either too small to detect or so large that no information about any fault may be obtained. The detrimental influence of noise on the residual will be considered in later sections.

Noise may cause the level of a residual to rise, creating a false indication of a fault. The sensitivity of the system depends on the set tolerance (Dummermuth, 1998). If the system tolerance is too high, many deviations might be overlooked, giving false negative indications of faults.

On the other hand, if the tolerance or threshold were too low, many false indications of faults will be generated. The issue as (Dummermuth, 1998) stated is the tolerance of the system. This will be regarded in detail in chapter 6.

4.3 Methodology

4.3.1 SIMULINK®

The system in figure 4.1 is a model that is based on a fault detection system that could be implemented on an actual process. Sets of training data and testing data are obtained from this SIMULINK® model. A MATLAB® programme was developed that prepares the system for certain faults (or no faults) and then activates the SIMULINK® model. The faults and their target values are set in the programme. The residuals from the SIMULINK® model are saved in a MAT file together with their target values. These residuals are then used to train and validate the classification networks.

The design of the training sets involved some trial and error. A description of how the training sets were chosen will be shown briefly later on. The neural network programme loads the training data from the MAT file and trains with it. The SIMULINK® model is used to simulate a plant that could develop faults as well as a fault-free reference plant.

The SIMULINK® model is a third-order process, where each stage is represented by a transfer function. These transfer functions are all first-order functions like equation 4.1, and have variables, a and b, that can be changed to represent an internal failure of some kind.

$$G(s) = \frac{a}{sb+1} \qquad (4.1)$$

The reference model and test model enable one to insert faults at different locations. Gain and offset or bias faults are investigated since these are symptoms of a wide variety of plant and instrumentation faults. Zero drift in sensors, incorrect setting of stroke of control valves, fouling of heat exchangers and the effects of ambient temperature are examples. These faults are inserted in the lower part of the parallel system in figure 4.1. The input to the model is put in the workspace of MATLAB® from where it is passed to the models. The six outputs are also stored in the workspace by SIMULINK® and are thus passed on to MATLAB® where it can be manipulated.

This system is a single-input single-output (SISO) system with all three states measured. To test the networks with a more complex situation, a multi-input single-output (MISO) system will be investigated in the latter stages of this section.

After the networks have proved that faults can be isolated in the ideal situation, classification of the residuals will also be tested when noise is added. For the more complex MISO system, the data will also be quantised to simulate quantising noise.

4.3.2 The data

A train of 20 pulses similar to the example in figure 4.3 is fed to the SIMULINK® model. These pulses vary in amplitude, duration and sign, the sign of each pulse being the negative of the previous pulse.

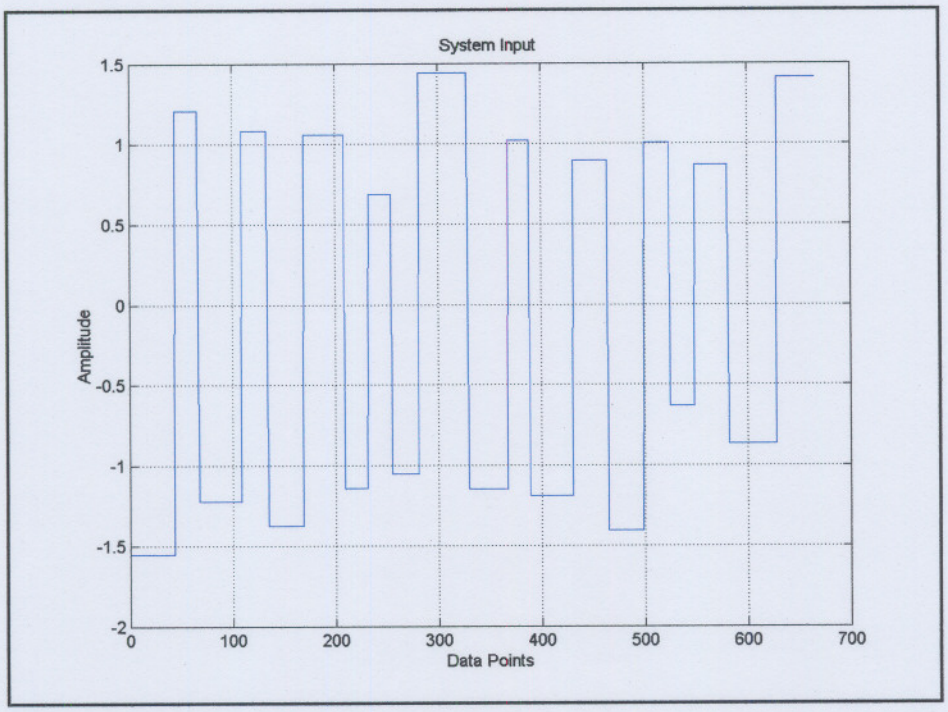

Figure 4.3: The input to the SIMULINK model.

This input represents the signal to an actuator that is used to control the input of the system represented by the SIMULINK® model. The pulse duration and amplitude is partly random. The six outputs are all of the same length and the outputs of the lower and upper system may differ in amplitude or dc level. The residuals are generated from these differences by subtracting the corresponding outputs of the upper system from those of the lower system.

At this stage there are three residuals. From these residuals the system can be monitored for faulty behaviour. Figure 4.4 illustrates these signals, firstly for the zero offset or bias fault and secondly for a gain fault. In this simulation the effect of noise has not yet been investigated, but will be addressed later.

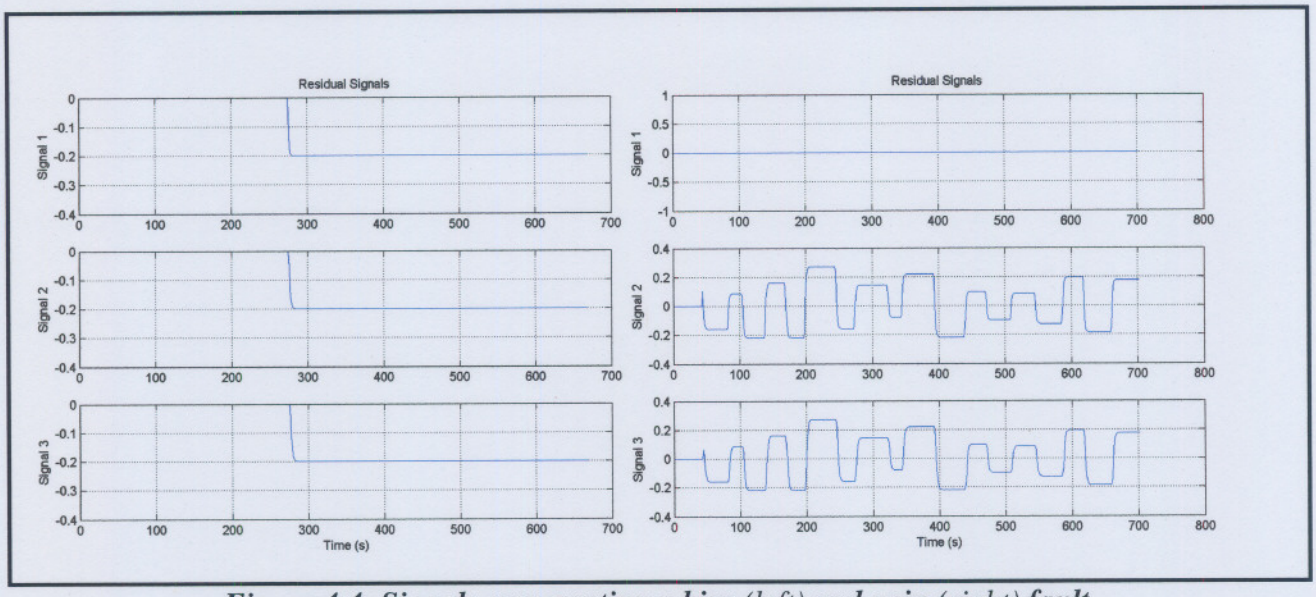

Figure 4.4: Signals representing a bias** (left) **and gain** (right) **fault.

When the programme detects a faulty situation the signal is stored in three fault windows. In this example, a total of 14 points taken every 5 steps are sampled and stored. If 14 consecutive points were taken, the fault windows might only contain the first transition of the gain fault causing it to look like a bias fault. The reason is that the first transition of the gain and the bias fault looks the same. Figure 4.4 shows the gain fault with its multiple transitions, while the bias fault window has only one transition.

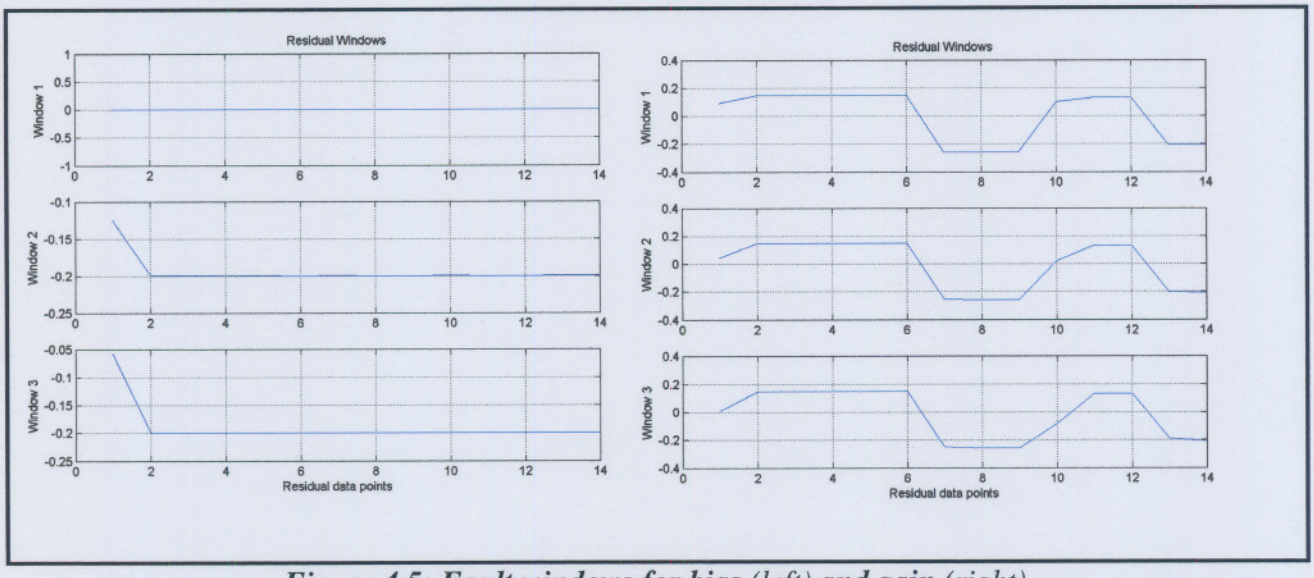

***Figure 4.5: Fault windows for bias** (left) **and gain** (right).*

Outputs for each residual window network			
No fault	1	0	0
Bias fault	0	1	0
Gain fault	0	0	1

Table 4.1: Possible outputs of the residual neural networks.

The inputs to the neural network consist of three windows similar to those in figure 4.5. In total there are 6 faulty conditions and then a faultless condition. Each window has three states, which is given in table 4.1. The fault status of the whole system depends on all three windows. For instance, there may be no fault in the first or even the second window, while the remaining window may contain a fault.

For the first scenario where there is a neural network for each window, the target values of the training data take the form of table 4.1. The output targets where one network is implemented for all three fault windows are illustrated in table 4.2.

Faults	Fault window 1	Fault window 2	Fault window 3	Target example (fault 3)
Fault 1(no fault)	No fault	No fault	No fault	0
Fault 2	Bias	Bias	Bias	0
Fault 3	No fault	Bias	Bias	1
Fault 4	No fault	No fault	Bias	0
Fault 5	Gain	Gain	Gain	0
Fault 6	No fault	Gain	Gain	0
Fault 7	No fault	No fault	Gain	0

Table 4.2: The output target format for the single-network scenario.

The faults in figure 4.4 are Fault 3 (left) and Fault 5 (right) respectively. To change the data in table 2 into the form of table 4.2, a conversion programme is implemented. The topology of these neural networks is discussed in the following section.

4.3.3 Neural network architectures

Two possible topologies are investigated. This investigation determines the fastest, most stable and most accurate topology in terms of training. The results of these investigations are given in the next section.

4.3.3.1 Three-network topology

Each of the three networks is equivalent in terms of the number of inputs, neurons, layers and outputs. Each network is faced with unique data generated in different parts of the process. Although in this instance the data may look similar, it will not be trained with exactly the same data and end up with exactly the same weight matrix. Each network is set up in MATLAB® as illustrated below.

```
net=newff(z,[6 3],{'tansig','purelin'},'traingdx');
net.trainParam.lr = 0.05;
net.trainParam.epochs = 10;
net.trainParam.goal = 1e-5;
```

A *for* loop is implemented to change the input/output training pair. This loop is inside another *for* loop that determines how many times the network will be trained with the whole training set. In this instance the number of epochs or the amount of times the network will train the specific input/output training pair is 10. If the outside loop runs 30 times, each training pair is trained 300 times.

4.3.3.2 Single-network topology

Here the network has 7 outputs instead of three due to the change in target format. A two-layer, 15-7 combination is illustrated as an example.

```
net1=newff(z,[15 7],{'tansig','purelin'},'traingdx');
net1.trainParam.lr = 0.05;
net1.trainParam.epochs = 50;
net1.trainParam.goal = 1e-5;
```

The input to this network is the three fault windows appended, one to the other, which gives a 42-point input instead of three 14-point inputs. If one divided the input neurons by three, each fault window would be *assigned* 5 neurons. However, each input is connected to all the input neurons - 15 in this case - and thus the previous statement is not valid.

4.4 Results

4.4.1 Three-network topology

For all the combinations in the three-network experiments the deviation from the desired output is small. The issue of the investigation amounts to which network architecture is the most efficient in terms of speed. The best of these results will then be compared to the single-network topology.

The time that the network needs to train depends on the complexity of the architecture. There is a trade-off between having a network with a very high complexity. To illustrate, consider two network architectures: one is a two-layer 3-3 combination and the other is a 30-3 combination.

View the simple network as a *dumb* child and view the complex network as an *intelligent* child. The *dumb* child will take much longer to achieve the *goal* than the *intelligent* child does. When the simple network is given 300 epochs with which to train, it might never pass the performance *goal* and then only the time limit, performance gradient or epoch-limit, would stop the training. The complex network might learn the solution much faster and only need 150 epochs before crossing the performance goal. The trade-off is when the network becomes more complex and the computation time for each epoch increases. Another issue is whether or not more complex networks solve the problem better.

An experimental run in which six different architectures were investigated was made. Each experiment was run 10 times. The measures by which the networks were compared are time in seconds and a cumulative error. The mean-squared-error of test samples are added together to generate the cumulative error. The test data is a set of 14 samples that comprises two sets of the possible 7 faults that could occur.

Architecture	**3-3**(1)	**6-3**(2)	**9-3**(3)	**14-3**(4)	**18-3**(5)	**24-3**(6)
Time (s)	1188.8	894.5	845.3	748.2	699.1	635.4
Cum. Error	1.32554	0.10524	0.14262	0.22883	0.26836	0.35748

Table 4.3: Performance comparison of the three-network topology.

As illustrated in figure 4.6, the speed by which the network train improves as the input neurons are increased. However, the accuracy decreases as the number of input neurons are increased (Figure 4.7). The number of epochs was 300 in each experiment.

The network that gave the best accuracy was compared with the network that took the least amount of time to train. The number of epochs was set at 1050 epochs. The idea was to see if the tendency remained when the networks were given more epochs. After 8 runs the answer was obvious. Figure 4.8 illustrates this clearly.

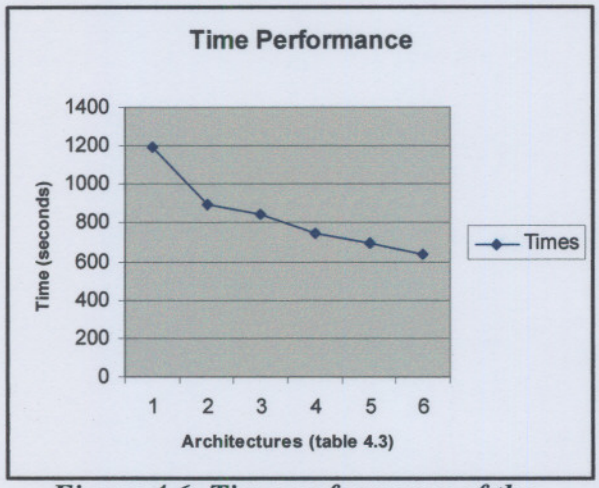

Figure 4.6: Time performance of the different topologies.

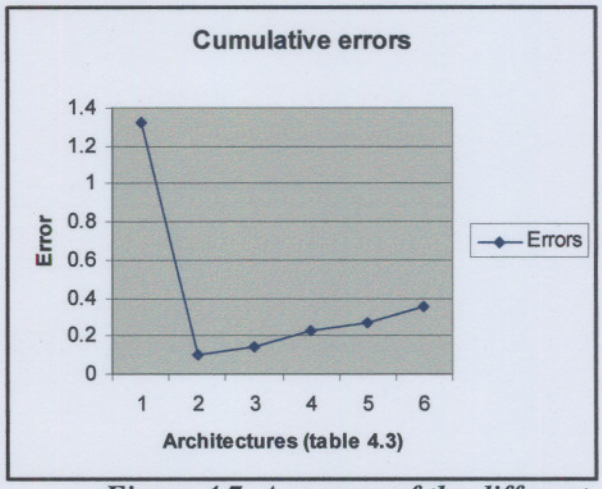

Figure 4.7: Accuracy of the different topologies.

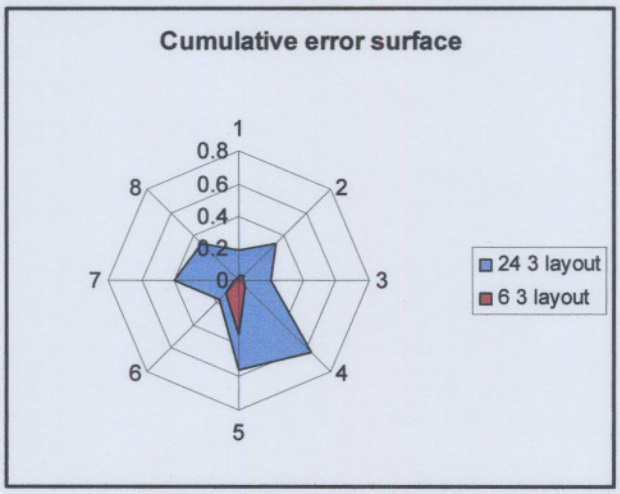

Figure 4.8: A radar plot of the cumulative errors over 8 runs.

Even though the 6-3 architecture took some 300 seconds (5 min) longer than the 24-3 network, the average cumulative errors over the 8 runs of this architecture is visibly much better. Numerically, it reads as follows: the 6-3 network has an error of 0.08246 and the 24-3 network has an error of 0.3469. Figure 4.8 also illustrates the consistency of the smaller network except for the one outlier at sample 5. The more complex network has less consistency if one looks at the angular surface in figure 4.8 (the ideal consistent surface would be a perfect octagon in this case).

Why is the complex network that trains 5 minutes faster the least accurate network? The reason might be *overtraining*. *Overtraining* is when the network trains the data too exact. If an input that varies just a little from its training data were given, the network cannot generalise and gives an inaccurate answer. Why would the larger network *overtrain* and not the smaller network?

The simple network has 84 input weights and the complex network has 336 input weights. The complex network has almost twice the number of neurons than it has inputs, while the simple network has less than half the number of neurons than it has inputs. These characteristics logically allow the complex network to see the input in much more detail. It also has four times more input weights than the simple network to store the characteristics of the input. This might allow the larger network to learn the training data too exactly and might decrease its generalisation.

The test data comprises two sets of the 7 possible errors and it was obvious that the complex network struggled with the second set of gain errors. It was less obvious in the case of the simple network. This would mean that the second gain error set is different in some way to those in the training data. The simple network classified this variant much better than the complex network did. The conclusion is made that the complex network *overtrained* because of its *resources* and did not have the same generalisation ability as the simple network had.

4.4.2 Single-network topology

Will the single network perform better than the three-network topology? A single window of 14 points is less complex as an input than three windows together as an input. The network becomes more complex in the single-network scenario even if the total number of input neurons were less. The number of input weights in the three-network topology is, for 6 neurons in each network, $14 \times 6 \times 3 = 252$ input weights for all 3 networks together. For the single network the number of input weights for 15 input neurons is $15 \times 42 = 630$. The single-network topology is a more complex problem.

The *single-network* scenario was tested with architectures of two and three layers. A cumulative error and time, in seconds, were again taken as performance measures. The number of epochs was kept at 1500, while the number of neurons and layers were varied. The trade-off between accuracy and speed was investigated by varying the complexity of the architecture. Detailed results are given in table A.2 and A.3 in the appendix.

Architecture	7 7 7(1)	9 7 7(2)	12 7 7(3)	9 9 7(1)	12 9 7(2)	15 9 7(3)
Time (s)	2163.1	2170.3	2136	1824.2	1986.5	2008.5
Cum. Error	1.11928	1.00877	0.702375	0.65277	0.77254	0.68951
Architecture	7 12 7(1)	9 12 7(2)	12 12 7(3)	15 12 7(4)	7 15 7(1)	9 15 7(2)
Time (s)	1606	1626	1729.6	1719.5	1449.9	1511.3
Cum. Error	0.82208	0.61401	0.61811	0.50778	0.6766	0.60638
Architecture	12 15 7(3)	15 15 7(4)	9 18 7(1)	12 18 7(2)	15 18 7(3)	
Time (s)	1545.2	1528.5	1377.9	1393.7	1471.7	
Cum. Error	0.64614	0.61474	0.85687	0.56688	0.75755	

Table 4.4: Performance comparisons of the single-network topology.

When three layers are used there are more combinations as illustrated in table 4.4. These values are averages taken over 10 experiments. To simplify the comparison of the different architectures, the architectures were grouped according to the number of neurons in the middle layer. In figures 4.9 and 4.10 they are plotted as the neurons in the input layer increases from architecture 1 to 3 or1 to 4 (See table 4.4).

Figure 4.10 shows that the more complex network architectures do not necessarily provide the best solution, although in terms of time performance the larger networks trained faster on average. Architectures that are less or more complex had inferior performances and were left out. With architecture 1 in figure 4.10, the network with 9 neurons in the middle layer performed the best. When the input neurons were increased, the performance of the network with 12 neurons in the middle layer outperforms the rest.

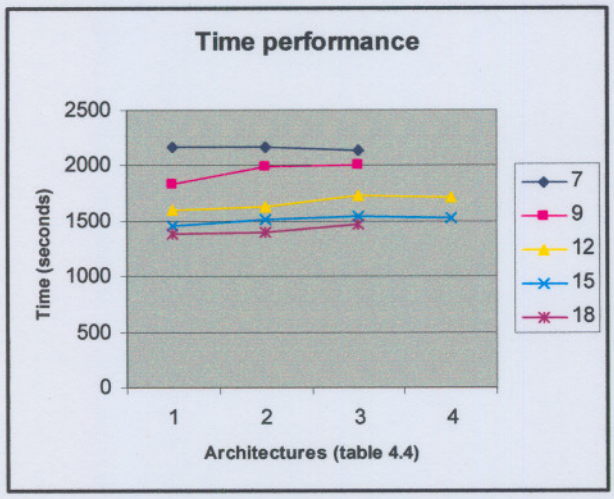

Figure 4.9: Time average plots of different topology groups.

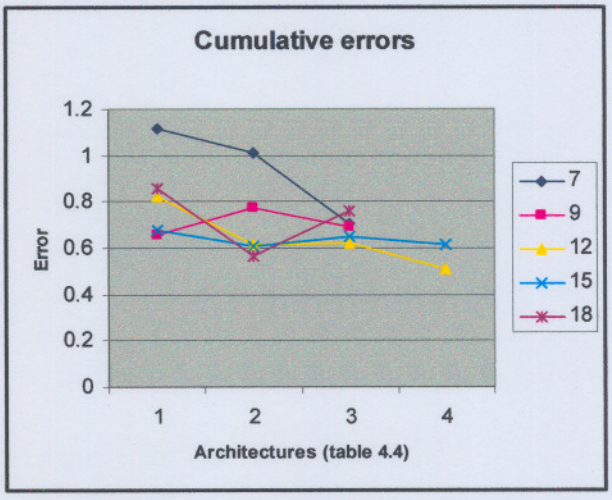

Figure 4.10: Cumulative error of the different topology groups.

4.4.3 Multi-input single-output system

The MISO system is investigated to test whether residual classification using neural networks is efficient in a more complex environment. The process residuals up to now have been too simple to test the classification performance with quantised data. The schematic illustration of the model is given in the appendix, figure A.1. The first input is the one used in the SISO model and the second input is a constant block-wave.

The three residuals in this instance differ from each other and not only from those in the previous section. Two of them have three possible states and the third has five possible states. The final residual may contain (1) no residual, (2) a bias residual, (3) a constant block-wave that has 1 or –1 amplitudes, (4) a non-constant block wave with varying amplitude and (5) a block wave that has zero regions in-between a non-constant block-wave. Examples of the last three possibilities are illustrated in figure 4.11.

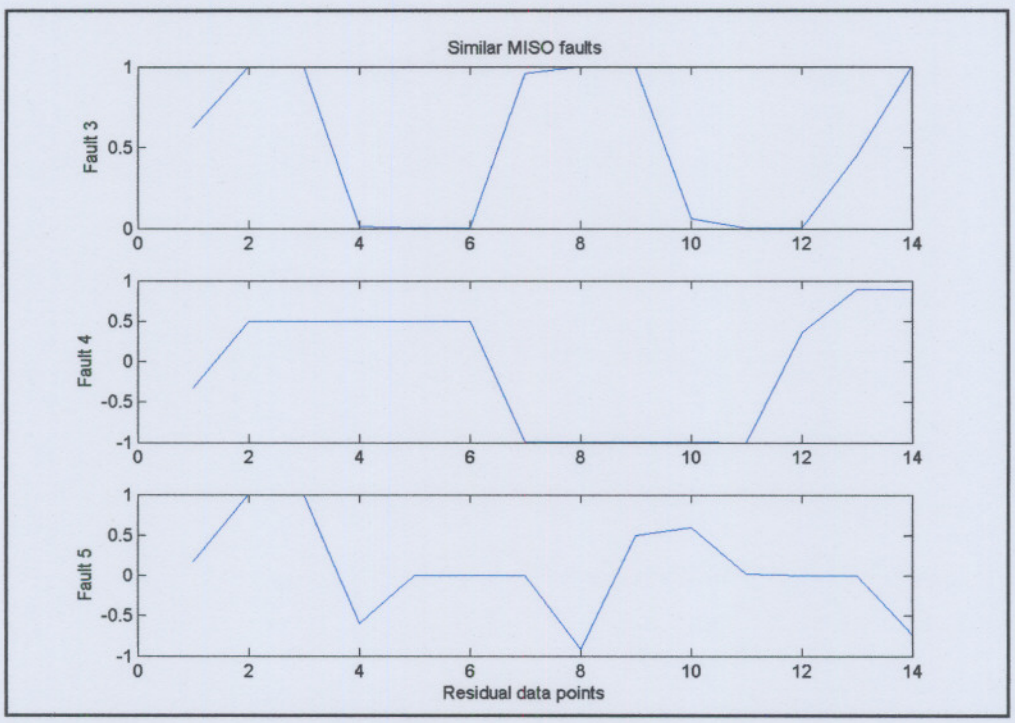

Figure 4.11: Three similar possibilities that increase classification complexity.

The similarity between these residuals causes uncertainty with regard to their classification. There were no misclassifications but the neural network struggled to generate definite or certain classifications in some instances. An example of such uncertainty is shown in figure 4.12.

The "combined NN outputs" axis in figure 4.12 has 11 possible positions. This is the possibilities of all three classification networks one after the other. The composition is 3-3-6, which represents three faults for the first two classifications and 6 faults for the third classification network. The "combined test data" axis represents the three inputs, one after the other, which gives 42 data points in total. The figure layout is chosen to emphasise the uncertainty.

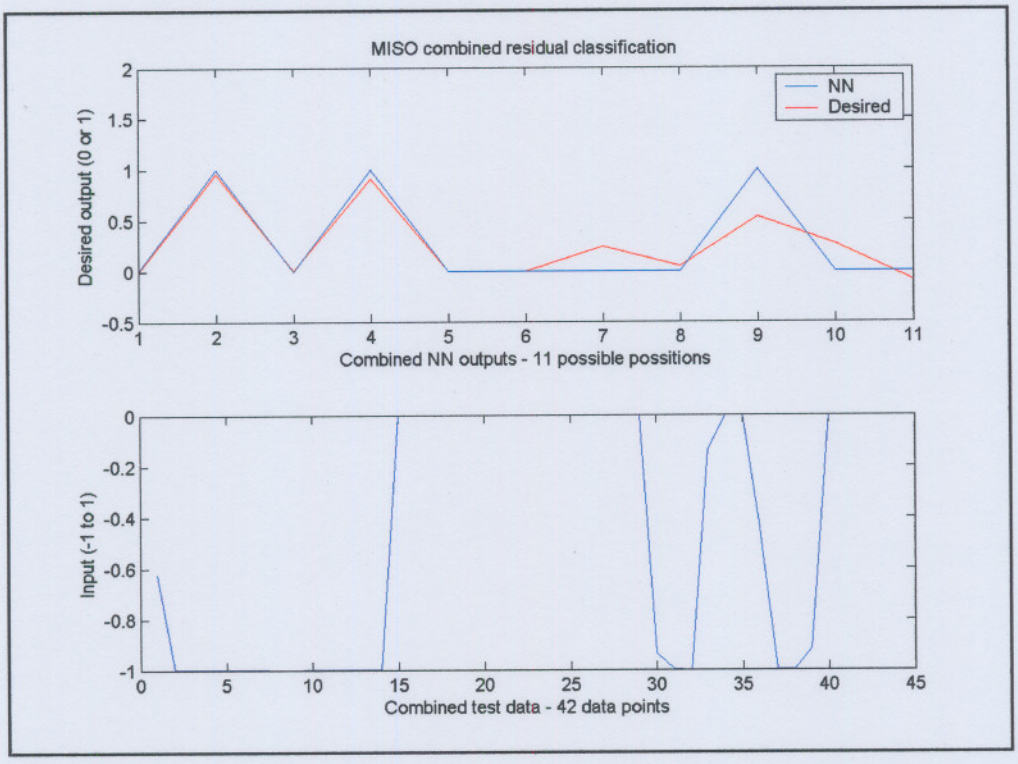

Figure 4.12: Neural network uncertainty with similar residuals.

4.4.4 Noisy data

In practice, the data that reaches the classification network will have some degree of noise present. This section takes the best network topologies from the previous section and adds noise to the testing data. The *three-network* topology's best *performer* is investigated. The 6-3 network was trained with noiseless data. Then noise was added to the testing data and the network was simulated with this noisy data. The noise added is Gausian noise that was generated with the *randn* function.

The network was trained a few times and simulated with noiseless data. The most accurate network was saved and the noise was added. The percentage of the noise to the signal was taken firstly at ± 36 dB SNR and taken up to ± 13 dB SNR. The actual data is tabulated in the appendix,

table A.4. Down to the 30 dB SNR mark there were no real deviation and no misclassifications. When the SNR reached the 24 dB mark, the network failed with some faults.

These misclassifications depended on where the larger noise components appeared. In some cases they appeared at places so that the network error was as low as 0.0289, which is just above the mean of the 30 dB SNR errors. In certain instances the cumulative error was large but there where no misclassifications. This is when the errors were spread over the whole test sample.

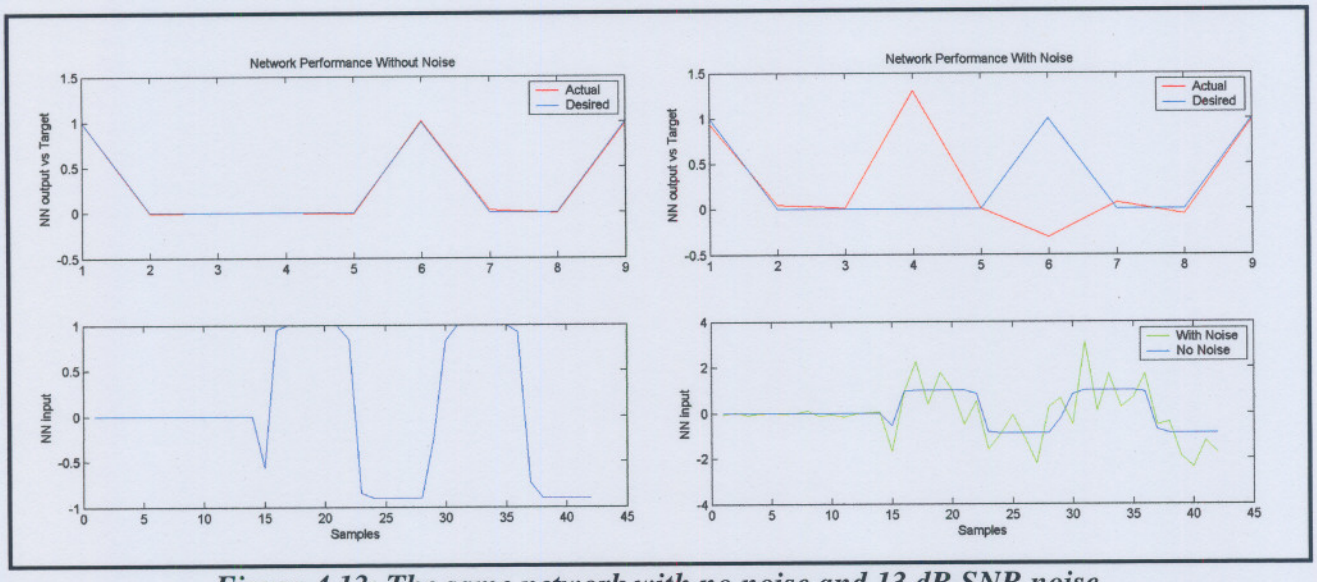

Figure 4.13: The same network with no noise and 13 dB SNR noise.

For the SISO process the input data is relatively simple and noise did not have a decisive effect on the output. The question might be whether the network performs with noise when the classification is not that clear-cut.

4.4.5 Quantised data

The effect that quantising of the input data has on the output of a neural network is investigated in this section. The MISO model residuals are used in this instance. The third neural network with the 5-output architecture is used. The five classes of residuals used for this network are sufficiently representative of the residuals that are used for the first two neural networks.

	No quant.	12-bit	8-bit	6-bit	4-bit	2-bit
Cum. MSE	0.2839	0.2839	0.2867	0.2979	0.4988	2.1432

Table 4.5: Cumulative MSE for quantised test set.

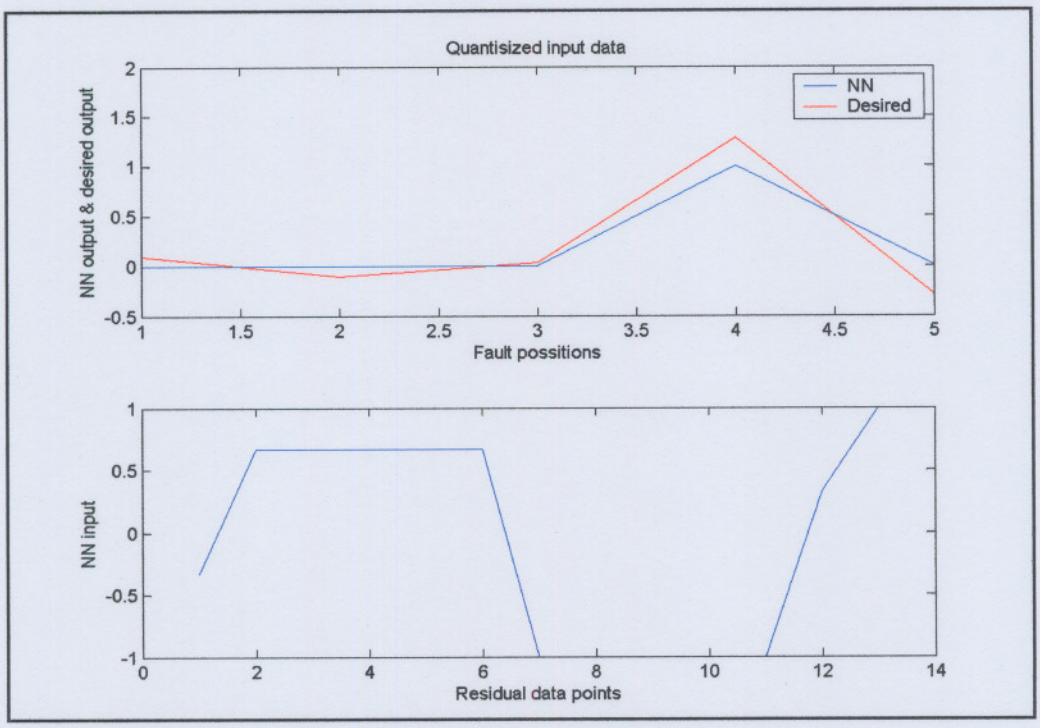

Figure 4.14: 2-bit quantized data.

The classification neural network was tested with 2-, 4-, 6-, 8- and 12-bit quantised input data. Table 4.5 numerically illustrates the effect of this quantising. The error that is present with no quantising is from the uncertainty discussed in section 4.4.4. The 12-, 8- and 6-bit quantising adds little to the existing error. The 4-bit quantizing almost doubles the existing error in this instance.

The 2-bit quantising does give a large error but some of the outputs were correctly classified in figure 4.14. The 2-bit case was the only experiment that gave misclassifications. The 4-bit experiment had one sample where the network gave fault 3 and fault 4 equal *likelihood*, when it should only have been fault 3.

4.4 Conclusion

The classification of the SISO model residuals posed no problem for the *three-network* topology. Even when noise was added, the generalising ability of the neural network prevented misclassification. It is also evident from this investigation that the use of more complex neural network architectures is not the probable solution. They might train faster, but they are more susceptible to *overtraining* and so generate bigger errors.

The neural networks did show little uncertainty with the residuals of the MISO model. Many experimental runs with different architectures only minimised this uncertainty to some test samples. More in-depth research is needed to locate the origin, which is beyond the scope of this paper. Finally, the neural network concept overcame the problem of classifying quantised data. In this instance, substantial problems only started below 4-bit quantising.

5 INDUCTION MACHINE BENCHMARK

The modelling ability of normal feed-forward neural networks with time delays is investigated. An induction machine model developed in SIMULINK® is implemented to obtain training data. The neural network's ability to model a dynamic non-linear process is investigated.

5.1 Purpose

A practical benchmark is chosen to gain knowledge about the feasibility of the modelling concept investigated below. In this instance a simulation of an induction motor was chosen as the benchmark. Panaitescu (Panaitascu R.C) developed an induction motor simulation for the steady state case of a squirrel-cage induction machine. Modelled data will be generated to train a dynamic neural network with the dynamic behaviour of this induction motor. The question is whether the neural network will be able to simulate the induction machine?

5.2 Background

This mathematical model of an induction machine is implemented in SIMULINK®. A brief discussion of the workings of the induction motor and the mathematical equations that are of importance are given.

A squirrel-cage induction motor is the specific machine that is simulated. This machine does not have rotor voltages, which simplifies the problem. Figure 5.1 (http://www.shef.ac.uk/materials/SSM/imi/squicage.jpg) gives an illustration of the basic idea behind a squirrel-cage induction machine.

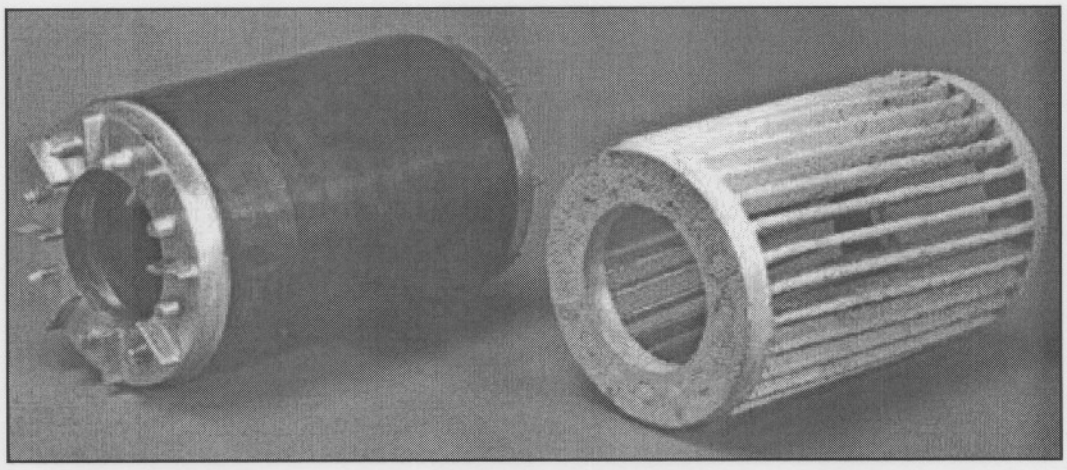

Figure 5.1: The rotor cage of a squirrel-cage induction machine.

The cage in figure 5.1 is commonly made of copper and is usually moulded within a less conductive metal. The magnetic fields produced by the 3-phase windings give rise to a current flow in the copper bars of the rotor. These currents in turn produce their own magnetic field that interacts with the stator field to produce an electromagnetic torque, which turns the rotor.

This magnetic torque is the output target that the neural network will be trained with. The equations that describe the motor mathematically are given below their symbol definitions in table 5.1.

- Voltage equations:

$$v_{sd} = R_s i_{sd} - \omega_d \lambda_{sq} + \frac{d}{dt}\lambda_{sd} \tag{5.1}$$

$$v_{sq} = R_s i_{sq} + \omega_d \lambda_{sd} + \frac{d}{dt}\lambda_{sq} \tag{5.2}$$

$$v_{rd} = 0 = R_r i_{rd} - \omega_{dA} \lambda_{rq} + \frac{d}{dt}\lambda_{rd} \tag{5.3}$$

$$v_{rq} = 0 = R_r i_{rq} + \omega_{dA} \lambda_{rd} + \frac{d}{dt}\lambda_{rq} \tag{5.4}$$

- Flux linkages:

$$\lambda_{sd} = L_s i_{sd} + L_m i_{rd} \tag{5.5}$$

$$\lambda_{sq} = L_s i_{sq} + L_m i_{rq} \tag{5.6}$$

$$\lambda_{rd} = L_r i_{rd} + L_m i_{sd} \tag{5.7}$$

$$\lambda_{rq} = L_r i_{rq} + L_m i_{sq} \tag{5.8}$$

- Electromagnetic torque:

$$T_{em} = L_m \left(i_{sq} i_{rd} - i_{sd} i_{rq} \right) \tag{5.9}$$

- Electrodynamics

$$J_{eq} \frac{d}{dt}\omega_m = T_{em} - T_L \tag{5.10}$$

$$\frac{d}{dt}\theta_m = \omega_m \tag{5.11}$$

or $\left(\frac{d}{dt}\theta_d = \omega_d ; \frac{d}{dt}\theta_{dA} = \omega_{dA} \right)$.

Name	Description	Units
v_a, v_b, v_c	Stator phase-voltages	V
i_a, i_b, i_c	Stator phase-currents	A
$\vec{v}_s(t)$	Stator voltage space-vector	V
$\vec{i}_s(t), \vec{i}_r(t)$	Stator and rotor current space-vectors	A
v_{sd}, v_{sq}	Stator voltage vector components in rotational reference frame	V
$v_{s\alpha}, v_{s\beta}$	Stator voltage vector components in stator oriented frame	V
i_{sd}, i_{sq}	Stator current vector components in rotational reference frame	A
$i_{s\alpha}, i_{s\beta}$	Stator current vector components in stator oriented frame	A
i_{rd}, i_{rq}	Rotor current vector components in rotational reference frame	A
$\vec{\lambda}_s(t), \vec{\lambda}_r(t)$	Stator and rotor flux space-vectors	Wb
$\lambda_{sd}, \lambda_{sq}$	Stator flux vector components in rotational reference frame	Wb
$\lambda_{rd}, \lambda_{rq}$	Rotor flux vector components in rotational reference frame	Wb
ω_d	Synchronous angular speed	rad
ω_{dA}	Slip angular speed	rad
ω_m	Rotor (mechanical) angular speed	rad
$\theta_d, \theta_{dA}, \theta_m$	Angles in rad, corresponding to the above speed types	-

Table 5.1: Symbol definitions.

Name	Description	Units
P_n	Motor Rated Power	W
I_n	Rated Phase Current (rms value)	A
V_n	Rated Phase Voltage (rms value)	V
P	Number of pole-pairs	-
f_n	Rated Frequency	Hz
T_n	Rated Torque	Nm
T_L	Load Torque	Nm
J_{eq}	Moment of Inertia (equivalent)	Kg m^2
R_s	Stator Resistance	Ω
R_r	Rotor Resistance	Ω
L_m	Magnetizing Inductance	H
L_{ls}	Stator Leakage Inductance	H
L_{lr}	Rotor Leakage Inductance	H
$L_s=L_{ls}+L_m$	Stator Equivalent Inductance	H
$L_r=L_{lr}+L_m$	Rotor Equivalent Inductance	H

Table 5.2: Induction machine parameter definitions.