



Minnesota State University, Mankato  
**Cornerstone: A Collection of Scholarly  
and Creative Works for Minnesota  
State University, Mankato**

---

All Graduate Theses, Dissertations, and Other  
Capstone Projects

Graduate Theses, Dissertations, and Other  
Capstone Projects

---

2020

## **Fault Detection and Classification of a Single Phase Inverter Using Artificial Neural Networks**

Ayomikun Samuel Orukotan  
*Minnesota State University, Mankato*

Follow this and additional works at: <https://cornerstone.lib.mnsu.edu/etds>



Part of the [Artificial Intelligence and Robotics Commons](#), and the [Electrical and Electronics Commons](#)

---

### **Recommended Citation**

Orukotan, A. (2020). Fault detection and classification of a single phase inverter using artificial neural networks [Master's thesis, Minnesota State University, Mankato]. Cornerstone: A Collection of Scholarly and Creative Works for Minnesota State University, Mankato. <https://cornerstone.lib.mnsu.edu/etds/1043>

This Thesis is brought to you for free and open access by the Graduate Theses, Dissertations, and Other Capstone Projects at Cornerstone: A Collection of Scholarly and Creative Works for Minnesota State University, Mankato. It has been accepted for inclusion in All Graduate Theses, Dissertations, and Other Capstone Projects by an authorized administrator of Cornerstone: A Collection of Scholarly and Creative Works for Minnesota State University, Mankato.

**Fault Detection and Classification of a  
Single Phase Inverter Using Artificial Neural  
Networks**

by

**Orukotan, Ayomikun Samuel**

A Thesis Submitted in Partial Fulfillment of the  
Requirements for the Degree of  
Master of Science  
In  
Electrical Engineering

Minnesota State University, Mankato

June 8, 2020

June 8, 2020

Fault Detection and Classification of a Single Phase Inverter using Artificial Neural Networks

Orukotan, Ayomikun Samuel

This thesis has been examined and approved by the following members of the student's committee.

---

Professor Vincent Winstead, P.E  
Advisor

---

Professor Jianwu Zeng  
Committee Member

---

Professor Xuanhui Wu  
Committee Member

## Acknowledgements

I would like to express my deepest gratitude to my supervisor Professor Vincent Winstead for his intelligent guidance and patient nurture for the past years. I have learned so much from his ways of critical thinking and his analytic insights into the problems helped greatly in the accomplishment of this thesis. I am proud to have such a great mentor on my way towards research and he has made my stay at Minnesota State University, Mankato a truly valuable experience. I want to also thank my colleague and friend, Somefun Oluwasegun for his numerous helpful comments and enlightening discussions throughout my research course. I would also like to dedicate this special thanks to my family, especially my parents, who have always been there for me, whenever and wherever. This research work will not be possible without their love and encouragements. Finally, thanks are dedicated to XCEL Energy for the funding support.

# Contents

Acknowledgments . . . . .	ii
List of Tables . . . . .	vii
List of figures . . . . .	viii
Abstract . . . . .	x
<b>1 Introduction</b>	<b>1</b>
<b>2 Background Information</b>	<b>7</b>
2.1 Important Artificial Neural Network Terminology and Concepts . . .	7
2.1.1 Activation Functions . . . . .	8
2.1.1.1 Linear Activation Function . . . . .	10
2.1.1.2 Sigmoid or logistic Activation Function . . . . .	11
2.1.1.3 Hyperbolic Tangent Activation Function . . . . .	11
2.1.1.4 Rectified Linear Unit (ReLU) . . . . .	12
2.1.1.5 Leaky ReLU . . . . .	13
2.1.1.6 SoftMax Activation Function . . . . .	13
2.1.2 Weights . . . . .	14
2.1.3 Layers . . . . .	14
2.1.4 Bias . . . . .	14

2.1.5	Units/Neurons . . . . .	15
2.1.6	Threshold . . . . .	15
2.1.7	Learning Rate . . . . .	15
2.1.8	Error Calculation . . . . .	16
2.1.8.1	Error Calculation under Supervised Training . . . . .	16
2.1.8.2	Error Calculation under an Unsupervised Training . . . . .	18
2.2	Artificial Neural Network Structure and Architecture . . . . .	19
2.2.1	Input Layer . . . . .	21
2.2.2	Hidden Layer . . . . .	21
2.2.3	Output Layer . . . . .	21
2.3	Learning in Neural Network . . . . .	22
2.3.1	Supervised Learning . . . . .	22
2.3.2	Unsupervised Learning . . . . .	23
2.3.3	Reinforcement Learning . . . . .	23
2.3.4	Techniques of Supervised Learning of a neural network . . . . .	24
2.3.5	Problems commonly solved with Neural Networks . . . . .	24
2.3.5.1	Classification . . . . .	24
2.3.5.2	Prediction . . . . .	25
2.3.5.3	Pattern Recognition . . . . .	25
2.3.5.4	Optimization . . . . .	26
2.4	Training Algorithms or learning rules for Artificial Neural Networks . . . . .	27
2.4.1	Delta Rule . . . . .	27
2.4.2	Generalized Delta Rule . . . . .	29

2.4.3	Gradient Descent (GD) Algorithm . . . . .	30
2.4.3.1	Stochastic Gradient Descent (SGD) . . . . .	30
2.4.3.2	Mini-batch Gradient Descent . . . . .	30
2.4.3.3	Batch Gradient Descent . . . . .	31
2.4.4	Back Propagation Algorithm . . . . .	31
2.4.4.1	Fundamental mathematics behind Back Propagation [1] . . . . .	32
2.4.5	Boltzmann Learning . . . . .	35
2.4.6	Competitive Learning Rule . . . . .	36
2.4.7	Memory Based Learning . . . . .	36
2.4.8	Hebb-Net Learning . . . . .	37
2.4.9	Forward Propagation . . . . .	38
2.5	Convergence of Gradient Descent . . . . .	38
<b>3</b>	<b>Methodology</b>	<b>40</b>
3.1	One-hot encoding . . . . .	40
3.2	Momentum [2] . . . . .	40
3.3	Node List . . . . .	42
3.4	Layer Space . . . . .	43
3.5	Weight Initialization . . . . .	43
3.6	Inverter Model or Snapshot of the studied model . . . . .	45
3.7	Fault Detection and Fault Generation . . . . .	46
3.8	Data Description . . . . .	47
3.9	Data Analysis . . . . .	47

3.9.1	Pre-processing of Data . . . . .	47
3.9.2	Power Spectral Density estimate of the raw data . . . . .	47
3.10	Flow diagram of the proposed Neural Network Process . . . . .	48
3.11	Performance Timing for Faults . . . . .	50
<b>4</b>	<b>Results and Discussion</b>	<b>52</b>
4.1	Training, validation, and testing of neural networks . . . . .	52
4.2	Evaluation Metrics . . . . .	53
4.2.1	Classification Accuracy . . . . .	53
4.2.2	Loss Function . . . . .	53
4.3	Fault Description . . . . .	54
4.4	Fault Classification and loss function results for 1,000 iterations . . .	55
4.5	Fault Classification and loss function results for 10,000 iterations . . .	65
4.6	Fault Classification and loss function results for 20,000 iterations . . .	75
	<b>Conclusion</b>	<b>83</b>
	<b>Appendix</b>	<b>94</b>
	<b>References</b>	<b>95</b>



# List of Tables

2-1	Analogy between the brain and Neural Network . . . . .	8
2-2	The XOR logical operation in Neural Network . . . . .	17
2-3	The Layer structure of different NN type . . . . .	20
2-4	Types of Training Methods and their Data . . . . .	23
3-1	Weight Initialization Scheme under Normal or Uniform Distribution .	44
3-2	Performance Timing Computations . . . . .	51
4-1	Detected Faults and Description . . . . .	54
4-2	Fault Classification rates(%) of different neural network models . . .	64
4-3	Loss Function of different methods over 1,000 Epochs . . . . .	65
4-4	Fault Classification rates(%) for 10,000 Simulation runs . . . . .	74
4-5	Loss Function of different methods over 10,000 Epochs . . . . .	75
4-6	Loss Function of three different methods . . . . .	80
4-7	Fault Classification rates(%) for 20,000 Simulation runs . . . . .	81

# List of Figures

2-1	Illustrative diagram explaining Neural Network Terminology . . . . .	8
2-2	General Overview of Activation functions in Neural Network . . . . .	10
2-3	The Structure of a Self Organizing Map . . . . .	18
2-4	The Layer structure of a Neural Network . . . . .	20
2-5	A single Layer Neural Network . . . . .	28
2-6	A single-layer neural network with three input nodes and one output node . . . . .	29
2-7	Gradient Descent paths in parameter space . . . . .	31
2-8	A deep multi-layer Neural Network . . . . .	33
2-9	The architecture of Hebb-Net Learning . . . . .	37
3-1	The Snapshot of the studied model . . . . .	45
3-2	The Simulink Model of the Circuit Under Test . . . . .	46
3-3	Welch Power Spectral Density Estimate . . . . .	48
3-4	Flow diagram of the proposed Neural Network Process . . . . .	50
4-1	Line Plots of Loss and Accuracy over 1000 Training Epochs . . . . .	56
4-2	Line Plots of Loss and Accuracy over 1000 Epochs for single switch . . . . .	57
4-3	Plots of Training error and Accuracy over 1000 Epochs for single switch . . . . .	58
4-4	Plots of error and accuracy over 1000 Epochs for double switch faults . . . . .	59
4-5	Plots of training error and Accuracy over 1000 Epochs for triple faults . . . . .	60

4-6	Plots of MSE Loss and Accuracy over 1000 Epochs for multiple faults	61
4-7	Line Plots of Cross-entropy Loss over 1000 Epochs under normal condition . . . . .	62
4-8	Confusion matrix over 1000 Training Epochs under faulty condition .	63
4-9	Line Plots of MSE Loss and Accuracy over 10000 Training Epochs . .	66
4-10	Line Plots of MSE Loss and Accuracy for s3 in Fault type 2 . . . . .	67
4-11	Line Plots of Loss and Accuracy typical of Fault type 3 . . . . .	68
4-12	Line Plots of Loss and Accuracy typical of Fault type 4 . . . . .	69
4-13	Line Plots of Loss and Accuracy typical of Fault type 5 . . . . .	70
4-14	Line Plots of Loss and Classification Accuracy typical of Fault type 5	71
4-15	Training Performance Plot of Cross Entropy Loss typical of Fault type 4	72
4-16	Confusion Matrix typical of Fault type 4 over 10000 training epochs .	73
4-17	Training Learning Curves of a single switch over 20,000 epochs . . .	76
4-18	Training Learning Curves of double switch faults over 20,000 epochs	77
4-19	Training Learning Curves of triple switch faults over 20,000 epochs .	78
4-20	Training Learning Curves of multiple switch over 20,000 epochs . . .	79

## Abstract

The detection of switching faults of power converters or the Circuit Under Test (CUT) is real-time important for safe and efficient usage. The CUT is a single phase inverter. This thesis presents two unique methods that relies on back propagation principles to solve classification problems with a two-layer network. These mathematical algorithms or proposed networks is able to diagnose single, double, triple and multiple switching faults over different iterations representing range of frequencies. First, the fault detection and classification problems are formulated as neural network based classification problems and the neural network design process is clearly described. Then, neural networks are trained over different epochs to perform fault detection or repeatedly trained with the training data until the error is reduced to the satisfactory level. The performance of neural networks for different test suites are examined using two evaluation metric (classification accuracy and training error loss) from the standpoint of stability and convergence. The classification performance of the proposed neural network between normal and abnormal conditions is within the range of 93% and 100%. The simulation results show that the proposed network can detect faults quite efficiently, with the ability to differentiate between switching fault types. The results of this analysis on training error and accuracy are identified in tabular forms of Fault IDs and corresponding results based on different network designs and architecture.

# Chapter 1

## Introduction

The Power Electronics Market is expected to garner \$ 25 billion by 2022, registering a Compound Annual Growth Rate (CAGR) of 8.9% , with the inverter market valued to reach US\$ 93.7 billion by 2024, at a CAGR of 5.8% [3]. The DC-DC converter market is projected to grow from US\$ 8.5billion in 2019 to US\$ 22.4 billion by 2025, at a CAGR of 17.5% [4]. The implication of these recent statistics is that as long as power converter market size continues to increase, whether according to type (high voltage or low voltage) or to applications (electronics, industrial and power utilities), power electronics devices like (IGBT, MOSFET, BJTs), which are susceptible to failure in different modes, must be carefully studied to reduce their failure rates in either a preventive or corrective manner or both.

Inverter drive systems have become ubiquitous in the industry for different high-powered applications. However, every coin has two sides. On one hand, this type of system has significant benefits and makes people's lives more convenient. On the other hand, due to various faults that can occur in such power systems, ensuring component or system reliability, safety and efficiency is increasingly becoming a difficult task. For instance, in a photovoltaic-Inverter generation system, the cost of failure is equal to the value of the energy that would be generated while the system is down plus the cost of repairing and replacing parts [5]. To develop innovative power electronic converters and systems for all relevant applications which are efficient, reliable

and cost-competitive through reduction in maintenance and operational costs, early detection of (process) faults is necessary to avert total system failure. Predicting faults can minimize plant downtime, extended equipment life, increase the safety and reduce manufacturing costs.

In line with the aforementioned, many engineers and researchers have focused on different methods of fault detection in the quest of finding answers to increasing demands for reliability and safety of power systems. Most of the methods in the fault detection literature are based on various linear methodologies or exact models whereas industrial processes are often difficult to model. They are complex, non-linear and not exactly known.

It is of importance to define the following terminology related to faults: fault, failure and malfunction, types of faults and fault detection. A fault (defect/bug) is the cause of a failure in the model while a failure is the situation where a model output deviates from the expected/correct output. In addition, a test suite is a set of test cases [6]. Fault detection determines the occurrence of fault in a monitored system while fault isolation determines the location and the type of fault. Fault identification determines the magnitude (size) of the fault. Intuitively, fault isolation and fault identification are together referred to as fault diagnosis [7].

In another survey, Semiconductor failures in device modules were estimated at 21% of converter system failures, according to a survey based on over 200 products from 80 companies [8]. The results of a recent questionnaire for industrial power electronics [9], also showed that only 50% of respondents were satisfied with reliability monitoring methods, showing that additional research efforts are needed in health management. Therefore, a number of researchers have considered artificial neural networks as an alternative way to represent knowledge about faults [10], [11] and [12].

In the work of [13], they presented an overview of fault detection and isolation tech-

niques on vibrational time series data using Neural Networks. Their study explored automated methods of detecting faults in the viscous damper bearings of a helicopter drive shaft. Specifically, they designed a system that correctly categorize time series data from a helicopter drive shaft into either good or bad classes. Based on these study, they had concluded that the best approach which presents the best possibility of success for a particular system is one that is monitored over its lifetime and faults are detected as deviation from normal behavior. Their study also established that an approach that relies on combining data from different systems is doomed to fail. In another research effort, [14] developed a method for obstacle recognition used by a mobile robot. In the presence of noise, their method took into account the physical behavior of reflected ultrasound waves in order to extract some features from the signals and then, to take decision using a neural network. However, the results of this work proved efficient for a small set of data because 100 percent of learning samples (26 measurements) were well classified for 6 types of obstacles and 92 % for an hundred samples.

Artificial Neural Networks (ANN) provide an excellent mathematical tool for dealing with non-linear problems. In the works of [15] using the data-set presented in [16], they analyzed a database with the characteristics of a liquid ultrasonic flow meter, their state (healthy or unhealthy) and established relevant factors that can cause system failure. The results of their study have shown that to decide the state of health of a device, some problem variables contribute more compared to others especially when dataset is slightly unbalanced.

In another study, [17] modeled the energy generated as a function of environmental and control variables and used the model to improve the performance of Combined Cycle Power Plants using Neural Network with four (4) inputs (temperature, exhaust vacuum, ambient pressure and relative humidity) and one output (Energy Output).

Corroborating the findings of [15], a practical example in [18] forecasted the power generated by solar plants as a scaled power generated as a function of all the input variables.

In the work of [19], they have used the Hopfield network for identification of system parameters. The obtained parameters were further passed to Adaptive Resonance Theory (ART) network for fault diagnosis. The problem with this method is the choice of the optimal window size to detect the system parameters. Circumventing this problem, [20] have proposed a model-based fault diagnosis method to detect the fault and isolate faults in the robot arm control system. The fault in this system is detected when the error (i.e. difference between the system output and the estimated output) exceeds a predetermined threshold. Once the fault is detected, the estimated parameters are transferred to the fault classifier to determine the output. In similar fashion, [21] proposed a new neural network for fault diagnosis of rotating machinery which synthesizes the ART and the learning strategy of Kohonen network.

In this work of [22], possible cyber-attacks to aircraft attitude sensors were investigated using a new approach based on neural network observer. It is capable of online detection of possible attacks on the UAV sensors in the Inertial Measurement Units (IMU). The proposed design used Extended Kalman Filter (EKF) to tune the NN weights which increases the learning speed of the NN, and subsequently, improves the ability of the detection of the sudden attacks. The simulation results show that the developed method can successfully and accurately detect the sudden and smooth attacks in the sensors. This detection can be further used to help the system to correct itself and to be robust against cyber-attacks.

This thesis therefore presents a knowledge-based and Artificial-Intelligence (AI) based method to detect single and multiple faults of semiconductor switches of a single-phase inverter system. The AI-based (non-parametric) techniques have several advantages



compared to data-based, analytically redundant and parametric or statistical methods. For example, AI-based techniques do not require explicit mathematical models which can be challenging to derive. Unlike many other techniques, ANN does not impose any restrictions on the input variables (as in how they should be distributed).

Additionally, many studies have shown that ANNs can better model heteroskedasticity i.e. data with high volatility and non-constant variance, given its ability to learn hidden relationships in the data without imposing any fixed relationships in the data [9].

We presume that to get a good model with high accuracy, it is expedient to find optimal values of “W” (weights) that minimizes the prediction error as low as reasonably possible. This is achieved through the application of Back propagation algorithm which makes ANN a learning algorithm. By learning from the errors, the model is improved.

The rest of this thesis is organized as follows. Chapter 2 provides some background information on Artificial Neural Network (ANN) structure and architecture, Artificial Neural Network Terminologies, Learning techniques in Neural Network, training algorithms and procedures for training and convergence of Gradient Descent. Chapter 3 describes the methodology of Neural Networks as it relates to fault detection and classification method used in this thesis. It describes how neural networks learn complex behaviors through ANN-based Algorithms, particularly, the back-propagation algorithm, which is an important and representative learning rule of the neural network used in this thesis. Chapter 3 also provides a mathematical foundation of Activation functions in ANN, and convergence which relates to how well a Neural Network learns. Chapter 4 contains the simulation studies and presents results and other relevant discussions. Chapter 5 outlines the conclusion and possible direction for the future. Subsequent sections capture the reference section and Appendices for other

important information.

# Chapter 2

## Background Information

In this chapter, this thesis presents some background information necessary to understand the techniques/approaches used, definition of technical terms and concepts, comprehensive explanation of learning rules and training algorithms, as well as the mathematics behind them as deemed fit. The content of this chapter is organized under the following headings:

- 1) Artificial Neural Network (ANN) structure and architecture.
- 2) Artificial Neural Network Terminology.
- 3) Learning in Neural Networks.
- 4) Training Algorithms or learning rules for ANN.
- 5) Convergence of Gradient Descent (GD) Algorithms.

### **2.1 Important Artificial Neural Network Terminology and Concepts**

The neural network imitates the mechanism of the human brain. As the human brain is composed of neurons or nerve cells which transmit and process the information received from our senses. The neural network is constructed with connections of nodes, which are elements that correspond to the neurons of the brain. The Neural Network simulates the connection of neurons using the weight value. The table below summarizes the analogy between the brain and neural network (technology).

Brain	Neural Network
Neuron	Node
Connection of Neurons	Connection of weights

Table 2-1: Analogy between the brain and Neural Network

To explain some terminology used in Artificial Neural Network(s), consider the diagram below which represents the nodes of a neural network and which is denoted by the input signals, weights of corresponding input signals, bias and other important terms.

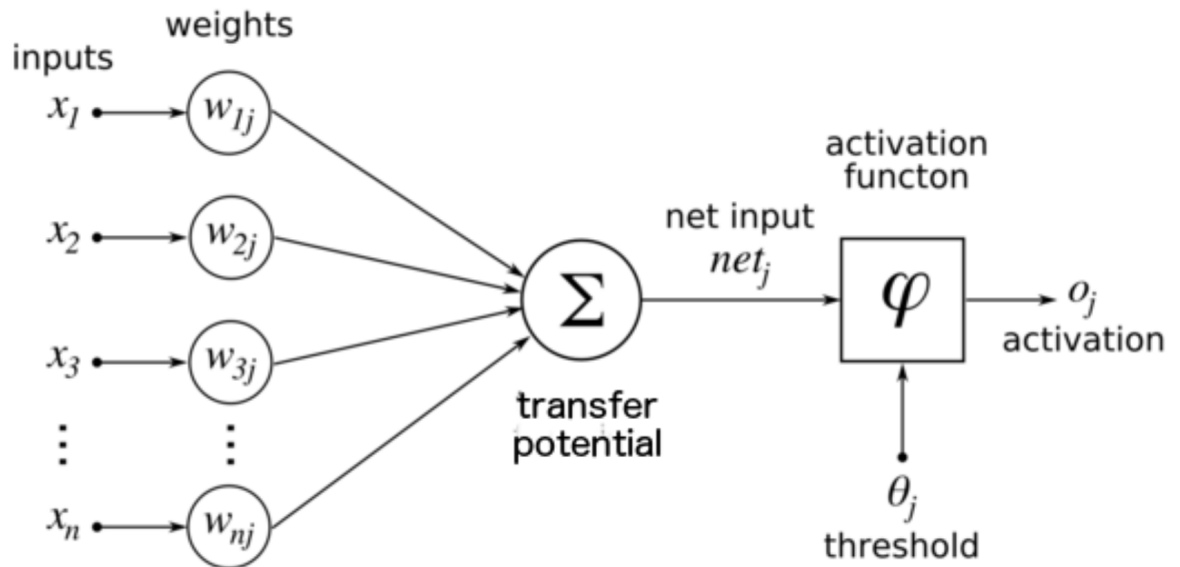


Figure 2-1: Illustrative diagram explaining Neural Network Terminology [23]

### 2.1.1 Activation Functions

Activation functions (mapping functions) are very important parts of the neural network used to calculate the output response of a neuron. The sum of the weighted input signal is applied with an activation function to obtain this response. The activation function in a Neural Network is analogous to the build-up of electrical potential in biological neurons that ignites once a certain activation potential is reached. In-

tuitively, this activation potential is mirrored in artificial neural networks using a probability measure. Depending upon the choice of activation function used, the properties of the network firing can be quite different but for neurons in the same layer, the activation function is usually the same. It is however worth noting that an activation function does two things;

- 1) Ensure non-linearity
- 2) Ensure gradients remain relatively large through the hidden unit.

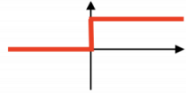
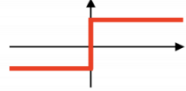
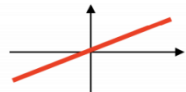
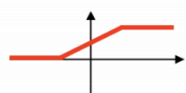
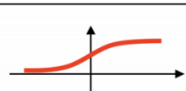
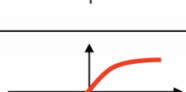
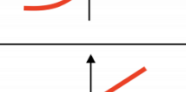

A neural network without any activation function would simply be a linear regression model, which is limited to the set of functions it can approximate. Using a non-linear activation, we are able to generate non-linear mappings from inputs to outputs. Hence, the need for connectionism. The basic idea of connectionism is to use simple neuron units which interconnect with each other and produce complex behavior. Without non-linearity, you cannot achieve this complexity.

**The general form of an activation function is shown below:**

$$h = f(W^T X + b)$$

$f(\cdot)$  represents the activation function acting on the weights and biases, producing  $h$ , the neural network output. Another important feature of an activation function is that it should be differentiable. The derivative of an activation function helps in calculating the backpropagation of a network which is to compute gradients of error (loss) with respect to the weights and updates the weight using gradient descent. The choice of an activation function is very important and can drastically improve or hinder the performance of a Neural Network. There are different types of activation functions that can be used to solve different problems depending on problem specifications or peculiarity. These functions include: Sigmoid, Linear, ReLu (Rectified Linear unit),

Leaky ReLU activation functions etc. The table below shows a summary of different activation functions, their graph and equations.

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Copyright © Sebastian Raschka 2016  
(<http://sebastianraschka.com>)

Figure 2-2: General Overview of Activation functions in Neural Network [24]

### 2.1.1.1 Linear Activation Function

The linear threshold function is the simplest of all the activation functions. In fact, it is not really a threshold or activation function at all, as whatever number is passed to it is returned unchanged. The drawback of this activation function is that it does not keep the input or output of the neuron within any sort of range. Furthermore, since there is no useful integral for it, the backpropagation training algorithm cannot be used with the linear activation function.

### 2.1.1.2 Sigmoid or logistic Activation Function

The sigmoidal activation function is useful with input positive numbers going into the neural network. The backpropagation algorithm requires the integral of the activation function. One of the most significant limitations of the sigmoidal activation function is that it is only capable of producing positive output. If a negative output is required, then the hyperbolic tangent activation function is considered. It ranges from 0 to 1. However, the drawback is that a saturated neuron causes the gradient to vanish and since it is not a zero-centered function, it makes convergence slower. The logistic function is given as:

$$f(x) = \frac{1}{1+e^{-x}} = y$$

Its derivative is given as:

$$f'(x) = \frac{\partial y}{\partial x} = \frac{-1}{(1+e^{-x})^2} \cdot (-e^{-x})$$

$$f'(x) = \frac{1}{1+e^{-x}} \left(1 - \frac{1}{1+e^{-x}}\right)$$

$$f'(x) = y \cdot (1 - y) \text{ (see[25])}$$

### 2.1.1.3 Hyperbolic Tangent Activation Function

The hyperbolic tangent activation function is a zero-centered function which allows for both positive and negative output. The output ranges from -1 to 1. This activation function circumvents the non-zero centric issue associated with logistic activation functions. Hence optimization becomes comparatively easier than with logistic. However, a hyperbolic tangent activated neuron may lead to saturation and cause vanishing gradients. It is similar to a logistic activation function with a mathematical equation shown below:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

$$f(x) = 2 \cdot \frac{1}{1+e^{-2x}} - 1 = 2 \cdot \text{logistic}(2x) - 1$$

Where logistic function is defined as the Logistic Activation function which can be expressed as:

$$\text{logistic}(x) = f(x) = \frac{1}{1+e^{-x}} = y$$

The derivative of Hyperbolic Tangent (tanh) activation function is given as:

$$\begin{aligned} f'(x) &= \frac{\partial f(x)}{\partial x} = \frac{\partial \frac{e^x - e^{-x}}{e^x + e^{-x}}}{\partial x} \\ f'(x) &= \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} \\ f'(x) &= 1 - \left( \frac{e^x - e^{-x}}{e^x + e^{-x}} \right)^2 = 1 - f(x)^2 \text{ see}([26]) \end{aligned}$$

#### 2.1.1.4 Rectified Linear Unit (ReLU)

It is used as a standard activation function in Convolutional Neural Network because it is computationally efficient and avoids the vanishing gradient problem. In practice, even though it is not a zero-centered function, it converges much faster compared to logistic and hyperbolic tangent activation functions. If the input is a positive number, the function returns the number itself and if the input is a negative number then the function returns 0. It is mathematically expressed as:

$$f(x) = \max(0, a) = \max\left(0, \sum_{i=1}^{i=n} w_i x_i + b\right)$$

The derivative of ReLU activation function is given as:

$$f'(x) = \frac{\partial f(x)}{\partial x} = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } x > 0. \end{cases} = \text{sgn}(\text{ReLU}(x))$$

Mathematically, sgn is represented as the signum function. According to [27], the sign function or signum function is an odd mathematical function that extracts the sign of a real number.

The signum function of a real number x is defined as follows:

$$\text{sgn}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$



$$\text{Alternatively: } \text{sgn}(x) = \frac{d}{dx} |x|, \quad x \neq 0.$$

Considering this: a case of bias initialized to a large negative value, then the weighted sum of inputs is close to 0 and the neuron is not activated. The implication of this instance is that up to 50% of ReLu activated neurons (may) die during the training [26]. In practice, to circumvent this problem, bias is initialized to a large positive value or another variant of ReLu known as Leaky ReLu is used.

#### 2.1.1.5 Leaky ReLu

It was proposed to fix the dying neurons problem of ReLu. It introduces a small slope to keep the update alive for the neurons where the weighted sum of inputs is negative. If the input is a positive number, the function returns the number itself and if the input is a negative number then it returns a negative value scaled by 0.01(or any other small value) [26]. It doesn't have any saturation problem in both positive and negative region. The neurons do not die because "0.01x" ensures that at least a small gradient will flow through. Although the change in weight will be small but after a few iterations it may come out from its original value. It is mathematically defined as:

$$f(x) = \max(0.01a, a) = \max\left(0.01a, \sum_{i=1}^{i=n} w_i x_i + b\right)$$

The derivative of Leaky ReLu is given as:

$$f'(x) = \frac{\partial f(x)}{\partial x} = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

#### 2.1.1.6 SoftMax Activation Function

For a binary classification problem, the logistic activation function works well but not for a multiclass classification problem. So, SoftMax is used for multiclass classification problem. The soft-max activation function is again a type of sigmoid function. As

the name suggests, it is a “soft” flavor of the max function where instead of selecting only one maximum value, it assigns the maximal element to the largest portion of the distribution, and other smaller elements getting some part of the distribution.

The standard softmax function  $\sigma : \mathbb{R}^k \rightarrow \mathbb{R}^k$  is defined by the formula as shown below:  $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$  for  $i = 1, \dots, K$  and  $\mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K$  (see [28])

In other words, the standard exponential function is applied to each element  $z_i$  of the input vector  $z$  and these values are normalized by dividing the sum of all these exponentials. This normalization ensures that the sum of the components of the output vector  $\sigma(\mathbf{z})$  is 1. SoftMax is generally preferred in the output layer where we are trying to get probabilities for different classes in the output.

### 2.1.2 Weights

Weights are network parameters that can be set to zero or initialized using specific methods but changes in weight generally indicate the overall performance of the neural net.

### 2.1.3 Layers

A layer is the highest-level building block in Neural Network. A layer is a container that usually receives weighted input, transforms it with a set of mostly non-linear functions called activation function and then passes these values as output to the next layer. The first and last layers in a network are called input and output layers, respectively, and all layers in between are called hidden layers.

### 2.1.4 Bias

It is a factor associated with the storage of information. Bias acts exactly as a weight on a connection from a unit whose activation is always 1. As a rule of thumb,

increasing the bias increases the net input to the unit which in turn improves the performance of the neural network. If bias is present, the net input is calculated as:

$$Net_i = b + \sum_{i=0}^n w_i x_i$$

Where  $b$  = bias,  $Net_i$  = Net Input,  $x_i$  = Input from neuron  $i$  and  $w_i$  = weight of the neuron  $i$  to the output neuron.

### 2.1.5 Units/Neurons

They are functions containing weights and biases and wait for data. After the data arrives, they perform some computations and then use an activation function to restrict the data to a range.

### 2.1.6 Threshold

The threshold,  $\theta$ , is a factor used in calculating the activations in a given net. The activation function is a function of the threshold.

### 2.1.7 Learning Rate

Learning rate is a concept in Neural Network training algorithms that specifies how radically the weight matrix should be updated based on training results. Learning rate is otherwise referred to as the step size of a neural network algorithm. Specifically, the learning rate is a configurable hyperparameter (these are the values which you have to manually set) used in the training of neural networks that has a small positive value, often in the range between 0 and 1. The learning rate controls how quickly the model is adapted to the problem and properly setting the learning rate can have a profound impact on the speed with which the neural network learns. Setting it too low will impede performance; too high may cause the neural network to behave randomly and never converge on a solution. In practice, a learning rate that is too large can

cause the model to converge too quickly to an optimal solution and overshoot the minimum, but a learning rate that is too small will take too many iterations to get to the minimum and lead to slow convergence.

## **2.1.8 Error Calculation**

Error calculation is an important aspect of any neural network; whether the neural network is supervised or unsupervised, an error rate must be calculated. The goal of virtually all training algorithms is to minimize the rate of error. This thesis examines how the rate of error is calculated for both supervised and unsupervised neural network.

### **2.1.8.1 Error Calculation under Supervised Training**

There are two values that must be considered in determining the rate of error for supervised training. First, we must calculate the error for each element of the training set as it is processed. Second, we must calculate the average of the errors for all of the elements of the training set across each sample. Implementing logic gates using neural networks help understand the mathematical computation by which a neural network processes its inputs to arrive at a certain output. This neural network will deal with the XOR logic problem. An XOR (exclusive OR gate) is a digital logic gate that gives a true output only when both its inputs differ from each other. For example, consider the XOR logical operation below:

<b>A</b>	<b>B</b>	<b>A <math>\oplus</math> B</b>
0	0	0
0	1	1
1	0	1
1	1	0

Table 2-2: The XOR logical operation in Neural Network

From the truth table shown above, the XOR logical operation requires a slightly more complex neural network than the “AND” and “OR” logical operations, (both of which have only two layers— an input layer and an output layer) because it requires one or more hidden layers [29]. Typically, for a complex system, the process involves creating a random weight matrix and then testing each row in the training set. An output error is then calculated for each element of the training set and after all of the elements of the training set have been processed, the root mean square (RMS) error is determined for all of them.

The output error is simply an error calculation that is performed to determine how different a neural network’s output is from the ideal output. This value is rarely used for any purpose other than as a stepping-stone in the calculation of the root mean square (RMS) error for the entire training set. Once all of the elements of a training set have been run through the network, the RMS error can be calculated. This error acts as the global rate of error for the entire neural network. It is important to note that a global error is a capture of all local errors calculated for each iteration, and can be mathematically aggregated using any of the techniques listed below:

$$\text{Mean Square Error, } MSE = \frac{1}{n} \sum_{i=1}^n E_i^2$$

$$\text{Sum of Squares Error, } SSE = \frac{1}{2} \sum_{i=1}^n E_i^2$$

$$\text{Root Mean Square, } RMS = \sqrt{MSE}$$

### 2.1.8.2 Error Calculation under an Unsupervised Training

In the literature, one common procedure of calculating error in an unsupervised training is by using a Self-Organizing Map. According to [29] and [30], a Self-Organizing Map (SOM) is a Neural Network Architecture that only contains an input neuron layer, no hidden layer and an output neuron layer. The input to a self-organizing map is submitted to the neural network via the input neurons. The input neurons receive floating point numbers that make up the input pattern to the network. A self-organizing map requires that the inputs be normalized to fall between -1 and 1 such that only one of the output neurons produces a value which could either be true or false. The structure of a typical SOM is shown below:

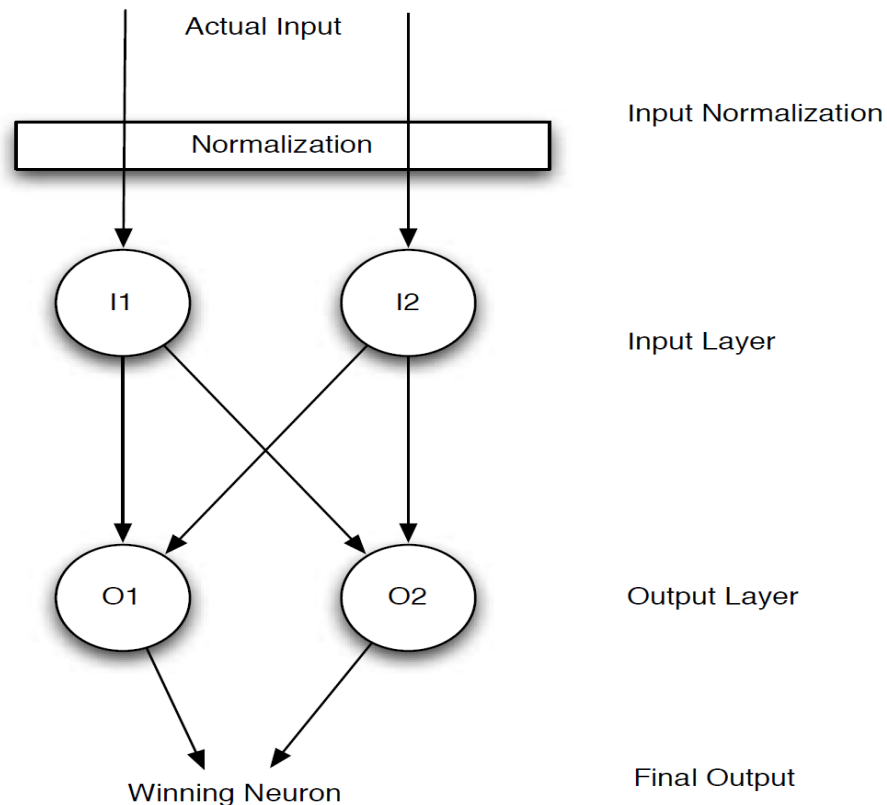


Figure 2-3: The Structure of a Self Organizing Map [30]

In reality, most unsupervised neural networks are designed to classify input data based on one of the output neurons. The degree to which each output neuron fires

for the input data is usually explored in order to produce an error for unsupervised training. Ideally, we would like a single neuron to fire at a high level for each member of the training set but when this is not the case, we adjust the weights to the neuron with the greatest number of firings, that is, the winning neuron consolidates its win. This training method causes more and more neurons to fire for the different elements in the training set.

## 2.2 Artificial Neural Network Structure and Architecture

As the “neural” part of its name suggests, they are brain-inspired systems which are intended to replicate the way humans learn (from its mistakes). It was intended to simulate the behavior of biological systems composed of neurons. Artificial Neural Networks (ANNs) are computational algorithms used as generalized non-linear function approximators. The artificial Neural network is typically organized in layers of many interconnected nodes which contain an activation function.

Neural Networks consist of input and output layers, as well as a hidden layer (in most cases) consisting of units or neurons that transform the input into a meaningful output. There are two major types of Neural networks which are single-layer neural network and multi-layer neural network. Initially, Neural Network pioneers had a very simple architecture with only input and output layers but no hidden layer. This topology is called single-layer neural networks. However, when hidden layers are added to a single-layer neural network, this produces a multi-layer neural network. Therefore, the multi-layer neural network consists of an input layer, hidden layer(s), and output layer.

On the other hand, it is necessary to distinguish between shallow neural network and deep neural network. The Neural Network (NN) that has a single hidden layer is called a shallow neural network or a vanilla neural network. A multi-layer neural network that contains two or more hidden layers is called a deep neural network. Most

of the contemporary Neural Networks used in practical applications are deep neural networks. The table below puts in perspective the difference between single-layer and multi-layer neural network as it relates to the types and layers of a network.

Neural Network Type	Layer Structure
Single layer NN	Input Layer $\rightarrow$ Output Layer
Multi-Layer Shallow NN	Input Layer $\rightarrow$ Hidden Layer $\rightarrow$ Output Layer
Multi-Layer Deep NN	Input Layer $\rightarrow$ Hidden Layers $\rightarrow$ Output Layers

Table 2-3: The Layer structure of different NN type

Generally, Neural Network contains three major layers, as it can be seen from the diagram below:

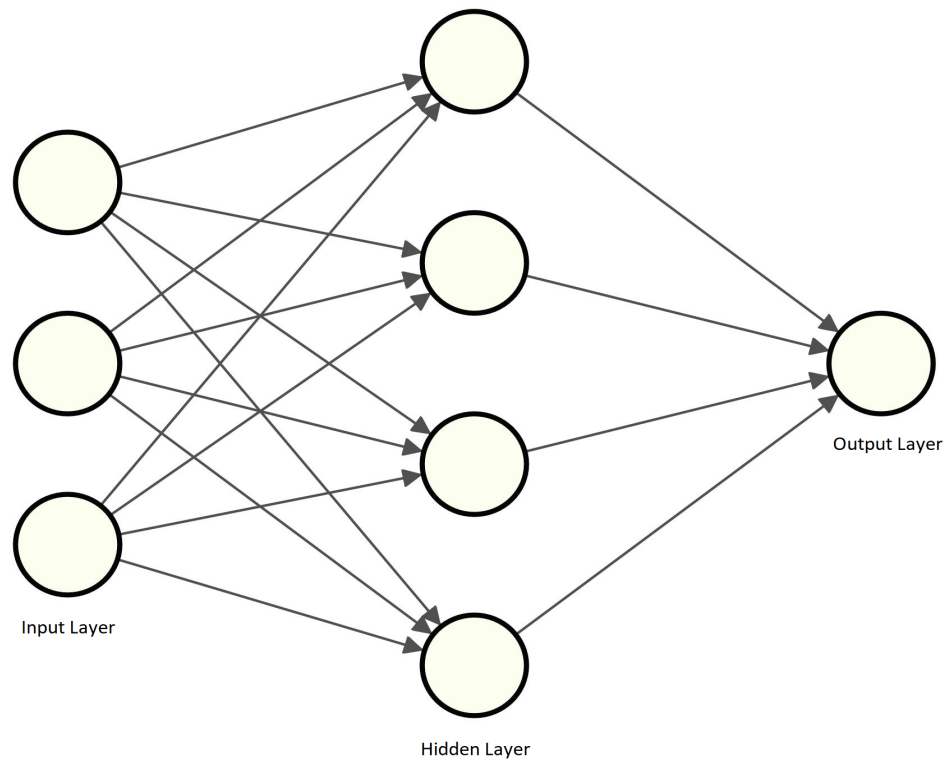


Figure 2-4: The Layer structure of a Neural Network [31]

1) Input Layer



2) Hidden Layer(s)

3) Output Layer

### **2.2.1 Input Layer**

This is the initial data for the NN. The objective of the input layer is to receive as input the values of the explanatory attributes for each observation. Usually, the number of input nodes in an input layer is equal to the number of explanatory variables. Specifically, input layer presents the patterns to the network, which communicates to one or more hidden layers even though the nodes of the input layer are passive, meaning they do not change the data. From the input layer, each value is duplicated and sent to all the hidden nodes.

### **2.2.2 Hidden Layer**

This is the intermediate layer between the input and output layers. The hidden layer applies given transformations to the input values inside the network. In a hidden layer, the actual processing is done via a system of weighted connections. There may be one or more hidden layers but the values entering a hidden node is usually multiplied by its weights. The weighted inputs are then added to produce a single number. There can be one or more hidden layers in the Architecture of ANN which makes it deep or shallow as the case may be. The hidden layers then link to an output layer.

### **2.2.3 Output Layer**

The Output layer produces the result for any given inputs. It receives connections from hidden layers or from the input layer by returning an output value that corresponds to the prediction of the response variable. In classification problems for

instance, there is usually only one output node that might give different possibilities. The active nodes of the output layer combine and change the data to produce the output values. The ability of the neural network to provide useful data manipulation lies in the proper selection of the weights and also, affect the output whether positively or negatively.

### **2.3 Learning in Neural Network**

Training or Learning is the process by which connection weights are assigned. Most training algorithms begin by assigning random numbers to a weighted matrix. However, the type of learning is determined by the manner in which the parameter changes takes place and the set of well-defined rules for the solution of a learning problem is called a learning algorithm. Each Learning algorithm differs from others in the way in which the adjustment to a synaptic weight of a neuron is formulated. Also, the manner in which a neural network is made up of a set of inter-connected neurons relating to its environment has to be considered. Generally, there are three types of learning. They are;

- 1) Supervised Learning
- 2) Unsupervised Learning
- 3) Reinforcement Learning

#### **2.3.1 Supervised Learning**

Supervised learning is a type of training method or algorithm that takes a known set of input data and known responses to the data (output) and trains a model to generate reasonable predictions for the response to new data [32]. As adaptive algorithms identify patterns in data, a computer “learns” from different observations. When exposed to more observations, the computer improves its predictive performance.

### 2.3.2 Unsupervised Learning

Unsupervised Learning is a type of training method that does not provide the neural network with expected outputs. In a neural net, if for the training input vectors, the target output is not known, the training method adopted is called an unsupervised training. The net may modify the weight so that the most similar input vector is assigned to the same output unit. Of course, unsupervised networks are far more complex and difficult to implement and that is why it is otherwise called self-learning or self-organizing networks because of its ability to carry out self-triggered learning.

### 2.3.3 Reinforcement Learning

Reinforcement learning is a general approach to learning that can be applied when the knowledge required to apply supervised learning is not available. Reinforcement learning attempts to learn the input-output mapping through trial and error with a view to maximize a performance index called the reinforcement signal. The system knows whether the output is correct or not but does not know the correct output. Reinforcement learning employs sets of input, associated outputs, and grade as training data. It is generally used when optimal interaction is required, such as control and game plays [2]. The table below depicts the different training methods and how its data are formulated.

<b>Training Method</b>	<b>Training Data</b>
Supervised Learning	{ input, correct output }
Unsupervised Learning	{ input }
Reinforced Learning	{ input, some output, grade for this output }

Table 2-4: Types of Training Methods and their Data

### **2.3.4 Techniques of Supervised Learning of a neural network**

In this thesis, supervised learning techniques are considered and hence, we focus on this type of training method. Moreover, it is used for more applications compared to unsupervised learning and reinforcement learning [2]. This training method is closely connected with a specific network topology which is explained in subsequent sections. The common ones include: Recurrent Cascade Correlation, Learning Vector Quantization, Feed forward propagation, Backpropagation through time and Real-time recurrent Learning.

### **2.3.5 Problems commonly solved with Neural Networks**

Neural networks are particularly useful for solving problems that cannot be expressed as a series of steps, such as recognizing patterns, classification, series prediction, and data mining. There are many different problems that can be solved with a neural network and these problems are categorized into types which include:

- 1) Classification
- 2) Prediction
- 3) Pattern recognition
- 4) Optimization

#### **2.3.5.1 Classification**

Classification is the process of classifying input data into groups. For example, IT Solutions at Minnesota State University, Mankato, a technology-based platform, may want its emailing system to classify incoming mail into groups, of either spam or non-spam messages. In the same vein, the neural network is usually trained by assigning it to group of data and as to which group each data element belongs. This

allows the neural network to learn the characteristics that may indicate each group membership. Neural networks are used in a broad range of classification problems: Examples include image classification [33], digit and character classification [34–37], or even medical diagnosis [38, 39]. A comprehensive survey on classification by neural networks can be found in [40].

### **2.3.5.2 Prediction**

Prediction is another common application for neural networks. Given a time-based series of input data, a neural network will predict future values. However, the accuracy of the guess will be dependent upon many factors, such as the quantity and relevance of the input data.

### **2.3.5.3 Pattern Recognition**

Pattern recognition is one of the most common uses for neural networks. Pattern recognition is simply the ability to recognize a pattern. The pattern must be recognized even when it is distorted. For example, every person who holds a driver’s license in the United States should be able to accurately identify a traffic light which is a *sine qua non* for safety of self and other drivers on the road. This is an extremely critical pattern recognition procedure carried out by countless drivers every day. Many variations of the traffic light recognition exist today, owing to differences in human perception, vision differences and eye defects. Still, recognizing a traffic light is not a hard task for a mentally stable driver. How hard is it to write an algorithm that accepts an image and tells a driver if it is a traffic light or other road signs? Without the use of neural networks, this could be a very complex task.

#### 2.3.5.4 Optimization

Another common use for neural networks is optimization. Optimization can be applied to many different problems for which an optimal solution is required. The neural network may not always find the optimal solution; rather, it seeks to find an acceptable solution. Optimization problems include circuit board assembly, resource allocation, and many others. Perhaps one of the most well-known optimization problems is the Traveling Salesman Problem (TSP).

A salesman must visit a set number of cities. He would like to visit all cities and travel the fewest number of miles possible. With only a few cities, this is not a complex problem. However, with a large number of cities, brute force methods of calculation do not work nearly as well as a neural network approach [30]. Basically, the two most common types of application for supervised learning are classification and regression. In classification, the goal is to assign a class (or label) from a finite set of classes to an observation. That is, responses are categorical variables. Applications include spam filters, advertisement recommendation systems, and image and speech recognition. Classification algorithms usually apply to nominal response values. However, some algorithms can accommodate ordinal classes. In regression, the goal is to predict a continuous measurement for an observation. That is, the responses variables are real numbers. Applications include forecasting stock prices, energy consumption, or disease incidence [32].

It is important to note that using an algorithm is a function of memory consumption, training speed and predictive accuracy on new data. These algorithms common to supervised training include: Classification Trees, Regression Trees, Discriminant Analysis (classification), k-Nearest Neighbors (classification), Naive Bayes (classification), Support Vector Machines (SVM) for classification and SVM for regression.

## 2.4 Training Algorithms or learning rules for Artificial Neural Networks

A neural network learns about its immediate environment through an interactive process of adjustments applied to its bias levels and weights. One common input to any learning rule is the error. The error is the degree to which the actual output of the neural network differs from the anticipated output. In supervised training, the neural network is constantly adjusting the weights to attempt to better align the actual results with the anticipated outputs that were provided. There are various learning rules in the literature. However, this thesis explains the following common ones.

- 1) Delta Rule and Generalized Delta rule
- 2) Boltzmann Learning
- 3) Gradient Descent Algorithm
- 4) Back Propagation Algorithm
- 5) Forward Propagation
- 6) Memory-based Learning
- 7) Competitive Learning rule
- 8) Hebbian or Hebb-Net learning Rule

### 2.4.1 Delta Rule

This is the representative learning rule of the single-layer neural network. It can sometimes be referred to as Adaline rule or the Widrow-Hoff rule. Although, it is not capable of multi-layer neural network training, it is very useful for studying the important concepts of the learning rule of the neural network. It is gradient descent

training technique that adjusts a network's weights based on differences between output and the ideal output. Back-propagation is a form of the delta rule. The delta learning rule is valid for only continuous activation functions and in the supervised training mode. In fact, the aim of the delta rule is to minimize the error over all training patterns.

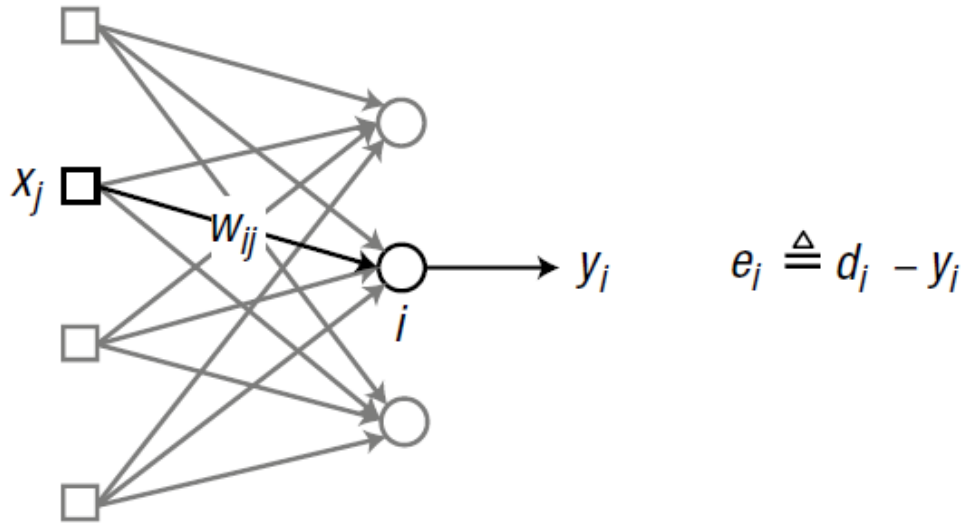


Figure 2-5: A single Layer Neural Network [2]

Consider a single-layer neural network, shown above. In the figure,  $d_i$  is the correct output of the output node  $i$ .  $x_j$  is input value and  $e_i$  is the output error. The delta rule can be expressed in equation as:

$$w_{ij} = w_{ij} + \alpha e_i x_j \quad (\text{Equation 2.1})$$

where  $x_j$  = The output from the input node,  $j$  ( $j = 1, 2, 3$ )

$e_i$  = The error of the output node,  $i$  = error of the output,  $y_i$ , from the correct output,  $d_i$ , to the input.  $w_{ij}$  = The weight between the output node  $i$  and input node,  $j$  and  $\alpha$  = Learning rate.

The learning rate,  $\alpha$ , determines how much the weight is changed per time. If this value is too high, the output wanders around the solution and fails to converge.



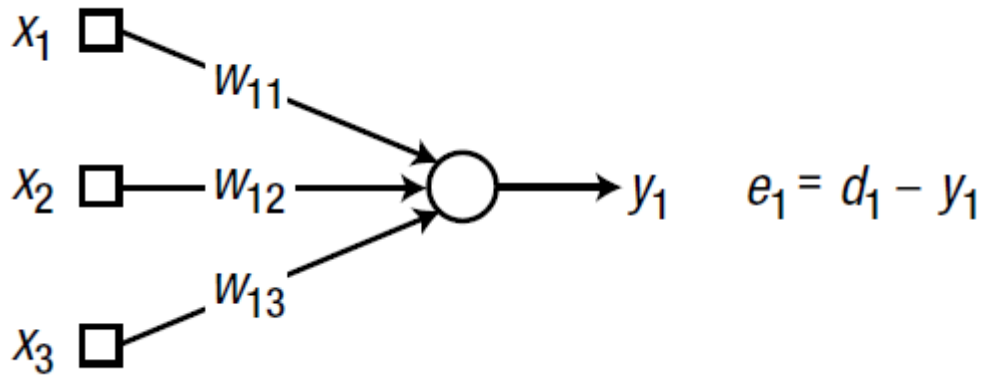


Figure 2-6: A single-layer neural network with three input nodes and one output node [2]

Considering, a single-layer neural network with three input nodes and one output node. Applying (Equation 2.1) to the neural network above, the adjustment of the weights is calculated as:

$$w_{11} \leftarrow w_{11} + \alpha e_1 x_1$$

$$w_{12} \leftarrow w_{12} + \alpha e_1 x_2$$

$$w_{13} \leftarrow w_{13} + \alpha e_1 x_3$$

Where:  $w_{11}$ ,  $w_{12}$  and  $w_{13}$  are weights.  $e_1$  is the error of the output,  $y_1$ , from the correct output,  $d_1$ , to the input. The weight updates according to delta rule is the multiplicative factor of the learning rate,  $\alpha$ , output node error,  $e_i$  and input node value,  $x_j$ . This can be mathematically expressed as:

$$\text{Weight Updates, } \Delta w_{ij} = \alpha e_i x_j \quad (\text{Equation 2.2})$$

The delta rule is a type of numerical method called gradient descent. The gradient descent starts from the initial value and proceeds to the solution.

### 2.4.2 Generalized Delta Rule

Delta rule has variants and as such as, there exists a generalized delta rule which can be expressed as:

$$w_{ij} = w_{ij} + \alpha \delta_i x_j \quad (\text{Equation 2.3})$$

In (Equation 2.3) above,  $\delta_i$  is defined as:

$$\delta_i = \phi' (V_i) e_i$$

where  $e_i$  = The error of the output node  $i$

$V_i$  = The weighted sum of the output node  $i$

$\phi'$  = The derivative of the activation function of the output node  $i$

### 2.4.3 Gradient Descent (GD) Algorithm

This is the simplest training algorithm used in supervised training model. There are different variations of GD algorithm. They are:

- 1) Stochastic Gradient Descent (SGD)
- 2) Mini Batch GD
- 3) Batch GD

#### 2.4.3.1 Stochastic Gradient Descent (SGD)

The Stochastic Gradient Descent (SGD) calculates the error for each training data and adjusts the weights immediately. With every GD iterations, it is imperative to shuffle the training set and pick a random training example from it but the draw back is that since you only use one training example, your path to the local minima will be noisy like a drunk man with an unstable or zig-zag pattern of movement.

#### 2.4.3.2 Mini-batch Gradient Descent

The mini batch method is a blend of the SGD and batch methods. Instead of iterating over all training examples and with each iteration only performing computations on

a single training example, we process ‘n’ training examples at once. This is a good choice for very large data sets.

### 2.4.3.3 Batch Gradient Descent

In the batch method, each weight update is calculated for all errors of the training data, and the average of the weight updates is used for adjusting the weights. This method uses all of the training data and updates only once. Vanilla or Batch gradient descent may be too slow or unstable due to the differences between the dimensions.

The figure below shows the comparison of the different variants of Gradient Descent.

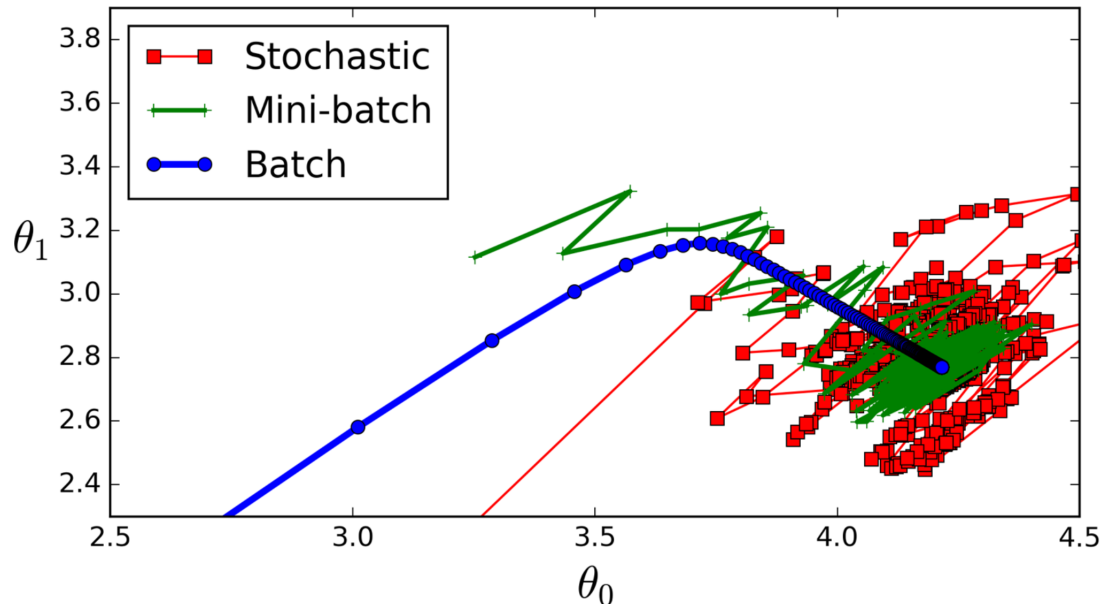


Figure 2-7: Gradient Descent paths in parameter space [41]

### 2.4.4 Back Propagation Algorithm

Backpropagation is a powerful training tool used by most Artificial Neural Networks to learn the knowledge weights of the hidden units or simply put, learning from mistakes. It is a feeding back mechanism that captures the rate of change of error with respect to the weight. It is only used in supervised learning since it requires a known output for training to help in determining the gradient descent of the loss

function. Mathematically, it is an extension of the gradient-based delta learning rule and it also uses gradient descent for training. The back-propagation algorithm defines the hidden layer error (the difference between desired and target) as it propagates the output error backward from the output layer. Once the hidden layer error is obtained, the weights of every layer are adjusted using the delta rule. The importance of the backpropagation algorithm is that it provides a systematic method to define the error of the hidden node. It is used mostly in the case of Multilayer Neural Network.

#### **2.4.4.1 Fundamental mathematics behind Back Propagation [1]**

In order to truly understand backpropagation as a training algorithm, we need to understand two (2) simple truths;

- 1) We cannot directly arrive at the rate of change of the error with respect to weights for all the units. Instead, we need to first compute the rate of change of error with respect to the activation functions of the hidden activities.
- 2) Once the rate of change of error with respect to the activation function is known, then using chain-rule, we can compute the rate of change of error with respect to the hidden units.

To explain the concept of backpropagation, let us consider a multi-layer ANN illustrated below.

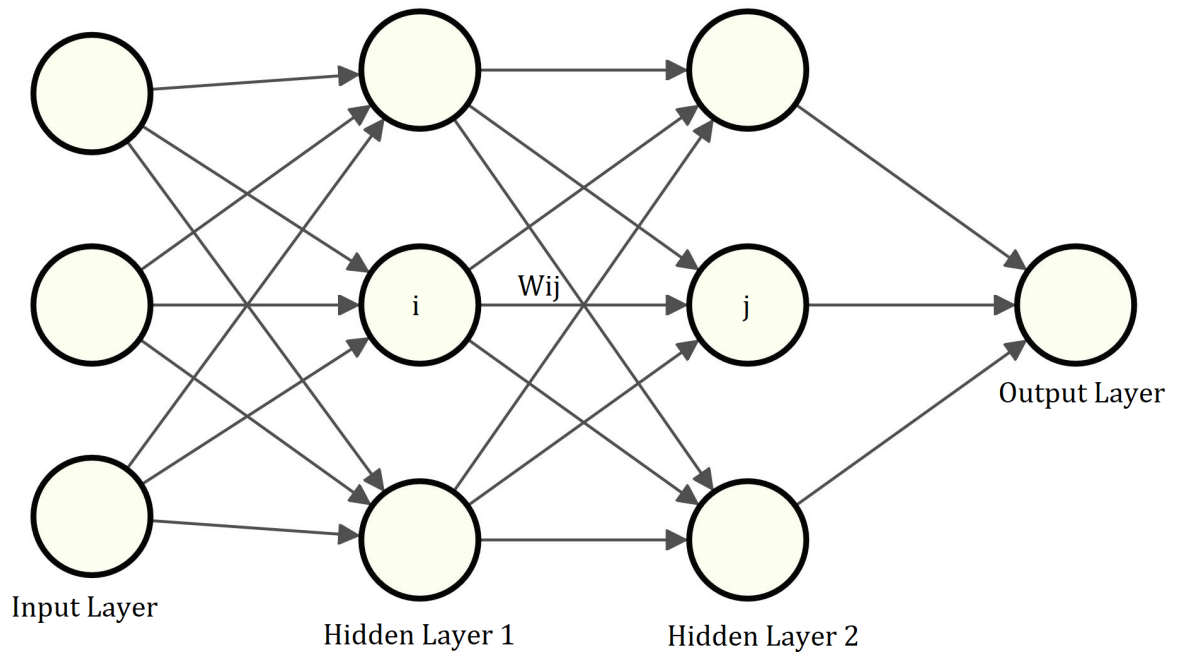


Figure 2-8: A deep multi-layer Neural Network [23]

In the above illustration, let us explain the different notations:

- 1)  $y_j$  is the output of the activation function in the j-layer.
- 2)  $y_i$  which is the output of the activation function in the i-layer.
- 3) Knowledge weight,  $w_{ij}$  which is the strength of the connections between neurons in layer 'i' and layer 'j'.
- 4) Theta,  $\theta$ , is the transfer potential into layer 'j'.

Let's assume an error,  $E$ , which is the difference between output  $y_j$  and some expected value  $t_j$  which can be expressed as:  $E(j) = t(j) - y(j)$

In order to compute the rate of change of error  $E$  with respect to weight  $w_{ij}$ , we must compute in the following order.

- (a) The rate of change of error  $E$  with respect to the transfer potential,  $\theta(j)$

- (b) The rate of change of error  $E$  with respect to the activation function of the hidden units in layer 'i' and
- (c) The rate of change of error  $E$  with respect to the hidden weights  $w(ij)$ .

Writing the procedures (a, b, and c) mathematically, we have:

$$\frac{\partial E}{\partial \theta_j} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{d\theta_j}$$

Assuming a logistic sigmoid activation function which is given by:

$$f(\theta) = \frac{1}{1+e^{-\theta}}$$

It is worthy to note that the Logistic sigmoid function is assumed to explain the mathematics behind back propagation because it is real-valued and differentiable, which is a primary requirement for being an activation function in ANN. Theta,  $\theta$ , is the transfer potential in the activation function above and it is mathematically expressed as:

$$\theta = \sum_{i=1}^n X_i * W_i$$

The '\*' operator is an activation function in the equation above. Therefore, the full form of the logistic sigmoid activation function which is the output of the activation function is expressed as:

$$f(\theta) = y' = \frac{1}{1+e^{-\sum_{i=1}^n (X_i * W_i)}}$$

where the Local or standard error,  $E$  is given as:

$$E = y' - y$$

$y'$  is the output from the activation function and  $y$  is the actual expected output.

From the equation above;

$$\frac{\partial E}{\partial \theta_j} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{d\theta_j}$$

Where  $\frac{dy_j}{d\theta_j}$  is the derivative of the logistic sigmoid function

$$\frac{dy_j}{d\theta_j} = y_j(1 - y_j)$$

$$\frac{\partial E}{\partial \theta_j} = \frac{\partial E}{\partial y_j} \cdot y_j (1 - y_j)$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial \theta_j} \cdot \frac{d\theta_j}{dy_i} = \sum_j \frac{\partial E}{\partial \theta_j} \cdot w_{ij}$$

$$\text{Recall that: } \theta_j = \sum_i (y_i \cdot w_{ij})$$

$$\frac{\partial \theta_j}{\partial w_{ij}} = y_i$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \theta_j} \cdot \frac{\partial \theta_j}{\partial w_{ij}} = \frac{\partial E}{\partial \theta_j} \cdot y_i$$

$$\therefore \frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} y_j (1 - y_j) \cdot y_i$$

If we choose ESS,  $E = \frac{1}{2} \sum_j (t_j - y_j)^2$  Where  $t_j$  is the expected output and  $y_j$  is the output of the activation function in the j-layer.

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$

$$\text{Hence, } \frac{\partial E}{\partial w_{ij}} = -y_i \cdot y_j (1 - y_j) (t_j - y_j)$$

So, the rate of change can now be directly applied to every knowledge weight or Weight Update,  $w_{ij}$  is expressed as:

$$w_{ij} = \frac{\partial E}{\partial w_{ij}} + w_{ij} \quad (\text{Equation 2.4})$$

### 2.4.5 Boltzmann Learning

This is a stochastic learning in which the neurons constitute a recurrent structure and they work in binary form. This learning is characterized by an energy function,  $E$  which is determined by the particular states occupied by the individual neurons of the machine. It is given by:

$$E = -\frac{1}{2} \sum_i \sum_j W_{ij} X_j X_i \quad i \neq j$$

Where  $X_i$  is the state of neuron  $i$  and  $W_{ij}$  is the weight of neuron  $i$  to neuron  $j$ . The  $i \neq j$  means that none of the neurons in the machine has self feedback.

This learning process has two kinds of neurons: Visible and hidden neurons. In visible neurons, there is an operating interface between the network and the environment whereas, in the hidden neurons, they operate independent of the environment. In a

nutshell, the learning rule of the multiclass classification neural network is identical to that of the binary classification neural network. Although these two neural networks employ different activation functions—the sigmoid for the binary and the SoftMax for the multiclass—the derivation of the learning rule leads to the same result.

### 2.4.6 Competitive Learning Rule

This rule is suited for unsupervised network training and it is such that the output neurons of a neural network compete among themselves to become active. This rule has a competitive mechanism that allows the neurons to fight for the right to respond to a given subset of inputs so that only one output neuron is active at any given time. This learning find application in learning the statistical properties of inputs and solving clustering problems.

### 2.4.7 Memory Based Learning

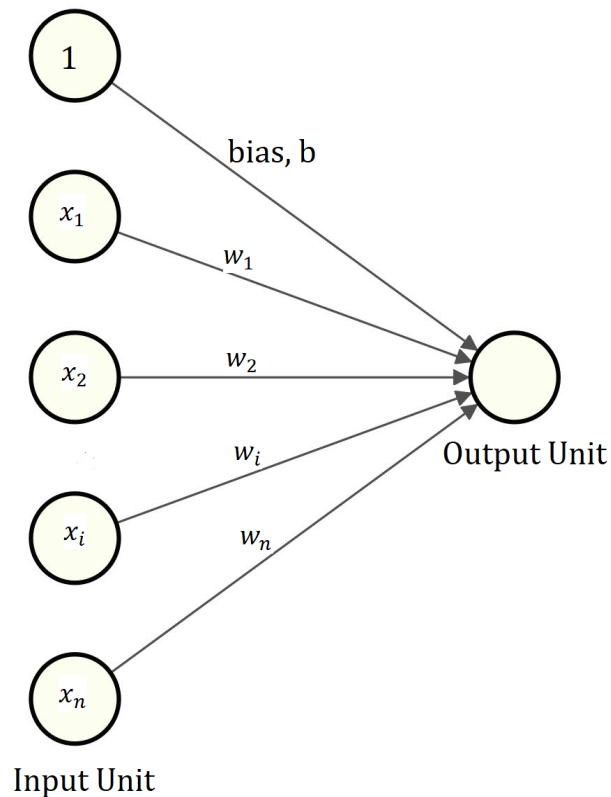
In memory-based learning, all the previous experiences are stored in a large memory of correctly classified relationship based on input vector and a scalar desired response. This algorithm find application in radial basis function network. According to [16], a radial basis function network is an artificial neural network that uses as activation functions.

It defines the local neighborhood of a test vector and a learning rule applied to the training in a local neighborhood which is popularly referred to as the nearest neighbor rule. The nearest neighbor rule is such that the local neighborhood is defined as the training example that lies in the immediate neighborhood of the test vector. This test vector is said to be the nearest neighbor of  $x$  if the minimum Euclidean distance between two immediate vectors is equal to the distance between the test vectors. A variant of the nearest neighbor classifier is the  $K$ -nearest neighbor classifier which usually acts like an averaging device.



### 2.4.8 Hebb-Net Learning

This is the first learning law for artificial neural network, and it was designed by Donald Hebb in 1949. The law states that if two neurons are activated simultaneously, then the strength of the connection between them should be increased. The architecture of the Hebb net consists of bias which acts as a weight on a connection from a unit whose activation is always 1. If the bias is increased, the net input of the unit also increases. Also, both the input and output data should be in bipolar form. For instance, if it is in binary form, the Hebb net cannot learn. This Hebb rule finds application in pattern classification problem.



**The Architecture of a Hebb net**

Figure 2-9: The architecture of Hebb-Net Learning [42]

### 2.4.9 Forward Propagation

According to Universal approximation theorem, a well-guided and engineered deep neural network can approximate any arbitrary complex and continuous relationship among the variables. The way neural network learns the true function is by building complex representations on top of simple ones. In a feedforward network, the information moves in only one direction – forward – from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network unlike recurrent Neural Networks or back propagation in which the connections between the nodes form a cycle. The input layer provides the initial data that then propagates to the hidden units at each layer and finally produce the output.

The architecture of a forward propagated network entails determining its depth, width, and activation functions used on each layer. In this case, depth is the number of hidden layers. Width is the number of units (nodes) on each hidden layer since the dimensions of neither input layer nor output layer cannot be controlled. Examples of Feedforward networks are single layer perceptron (no hidden layer) and multi-layer perceptron (has one or more hidden layers).

## 2.5 Convergence of Gradient Descent

An iterative algorithm is said to have converged to a solution if the value updates arrive at a fixed point where the gradient is zero and further updates do not change the estimate or jitter around the local minimum. Generally, Convergence is mathematically expressed as:

$$R = \frac{|f(x^{(k+1)}) - f(x^*)|}{|f(x^{(k)}) - f(x^*)|}$$

Where  $x^{(k+1)}$  is the  $k^{th}$  iteration and  $x^*$  is the optimal value of  $x$

If  $R$  is a constant or upper bounded, the convergence is linear. However, in reality, its arriving at the solution exponentially fast

$$|f(x^{(k)}) - f(x^*)| = C^k |f(x^{(0)}) - f(x^*)| \text{ (see [43])}$$

# Chapter 3

## Methodology

### 3.1 One-hot encoding

One-hot encoding is otherwise referred to as 1-of-N encoding and it is used as a method to quantify categorical data. According to [44], one hot encoding is a representation of categorical variables as binary vectors. This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1. Suppose, your features have normal, open circuit fault and closed fault labels and if you convert these features to nominal values yielding No Fault=2, Fault short=1, Fault Open=0. Therefore, in the case of One-hot encoding, we represent the three output nodes from the neural network by creating the classes as the following vectors: Class1  $\rightarrow$  [1, 0, 0] Class2  $\rightarrow$  [0, 1, 0] Class3  $\rightarrow$  [0, 0, 1]

### 3.2 Momentum [2]

Momentum is a method that helps accelerate Stochastic Gradient Descent in the relevant direction, dampens oscillations [45] [46] and improves both training speed and accuracy [47]. According to [48], the idea of momentum-based optimizers is to remember the previous gradients from recent optimization steps and to use them to help to do a better job of choosing the direction to move next, acting less like a drunk student walking downhill and more like a rolling ball.

Generally, Gradient Descent with momentum depends on two training parameters;

learning rate ( $\delta$ ) and momentum constant ( $mc$ ). The parameter,  $\delta$ , indicates the learning rate and the parameter,  $mc$  is the momentum constant that defines the amount of momentum and this is set between 0 (no momentum) and values close to 1 (lots of momentum). This section explores the variations of the weight adjustment. A critical look into the work of [2] has shown that the weight adjustment has relied on the simplest forms of (Equation 2.2) and (Equation 2.3). However, there are various weight adjustment forms available. The benefits of using the advanced weight adjustment formulas include higher stability and faster speeds in the training process of the neural network. These characteristics are especially favorable for Deep Learning as it is hard to train. This section only covers the formulas that contain momentum, which have been used for a long time. The momentum,  $m_k$ ;  $k \in N^+$ , is a term that is added to the delta rule for adjusting the weight. The use of the momentum term drives the weight adjustment to a certain direction to some extent, rather than producing an immediate change. It acts similarly to physical momentum, which impedes the reaction of the body to the external forces.

$$\begin{aligned}
 \Delta w(k) &= \alpha \delta x(k) \\
 m(k) &= \Delta w(k) + \beta m(k-1) \\
 w(k) &= w(k) + m(k) \\
 m(k-1) &= m(k)
 \end{aligned}
 \tag{Equation 3.1}$$

Where  $m(k-1)$  is the previously computed momentum and  $\beta$  is a positive constant that is less than 1. Let's briefly see why we modify the weight adjustment formula in this manner. The following steps show how the momentum changes over time:

$$\begin{aligned}
m(0) &= 0 \\
m(1) &= \Delta w(1) + \beta m(0) = \Delta w(1) \\
m(2) &= \Delta w(2) + \beta m(1) = \Delta w(2) + \beta \Delta w(1) \\
m(3) &= \Delta w(3) + \beta m(2) = \Delta w(3) + \beta \{ \Delta w(2) + \beta \Delta w(1) \} \\
&= \Delta w(3) + \beta \Delta w(2) + \beta^2 \Delta w(1) \\
&\vdots
\end{aligned}
\tag{Equation 3.2}$$

It is noticeable from these steps that the previous weight update, i.e.  $\Delta w(1)$ ,  $\Delta w(2)$ ,  $\Delta w(3)$ , etc., is added to each momentum over the process. Since  $\beta$  is less than 1, the older weight update exerts a lesser influence on the momentum. Although the influence diminishes over time, the old weight updates remain in the momentum. Therefore, the weight is not solely affected by a particular weight update value. Therefore, the learning stability improves. In addition, the momentum grows more and more with weight updates. As a result, the weight update becomes greater and greater as well. Therefore, the learning rate increases.

### 3.3 Node List

A node list is a collection of nodes in a Neural Network. In this thesis, the node list object is used to explore the possibility of having different Neural Network architectures (whether deep or shallow) for the purpose of results evaluation. According to [12], the node-list is a set of  $s$  nodes represented by  $N$ . It is generally expressed as:

$$N = \{ n_1, n_2, n_3, \dots, n_s \}$$

### 3.4 Layer Space

This is a neural network methodology used to define the structure of spaces of neural network functions and it is otherwise referred to as space of weight.

### 3.5 Weight Initialization

A proper initialization of the weights in a neural network is critical to its convergence. The analysis of [49], provides a clear demonstration of the role of non-linearities in determining the proper weight initializations. According to this research, the weight initialization strategy for the Rectified Linear Unit (RELU), which is not differentiable at 0 is different from activation functions differentiable at 0. It is important to note that the results from [50–54], all rely on the Xavier initialization scheme which cannot be used to compute the optimal value of  $v^2$  as it is a poor choice with ReLU Activation functions [55]. In an important follow up paper, [55] argued that the Xavier initialization does not work well with the RELU activation function, and instead proposed an initialization methodology commonly referred to as the “He” initialization. In support of the “He” initialization, the authors provided an example of a 30-layer neural network which converges with the “He” initialization, but not under the Xavier initialization. According to [55] and corroborated by the findings of [49], the Optimal value of  $v^2$  is given as:

$$v^2 = \frac{2}{N}$$

where  $N$  is the number of nodes feeding into that layer. To put the aforementioned into a better perspective, according to [54], we initialized the biases to be 0 and the weights  $W_{ij}$  at each layer with the following commonly used heuristic:

$$W_{ij} = U \left[ -\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}} \right]$$

Where  $U [-a, a]$  is the uniform distribution in the interval  $(-a, a)$  and  $n$  is the size of the previous layer (i.e the number of columns of  $W$ ). The following formula is given

for calculating the value of  $\epsilon$  used to initialize  $W_{ij}$  with random values  $[-\epsilon, \epsilon]$ . The parameter,  $\epsilon$ , it is defined as:

$$\epsilon = \frac{\sqrt{6}}{\sqrt{L_{output} + L_{input}}}$$

Then,  $W_{ij}$  is initialized as:

$$W_{ij} = (2\epsilon) \text{ rand}(L_{output}, L_{input+1})$$

where  $L_{input} = n_j$  and  $L_{output} = n_{j+1}$  are the number of units in the adjacent layers and  $\text{rand}(\cdot)$  is the random function over the interval  $(L_{output}, L_{input+1})$  with uniform distribution.

Sometimes, we don't know anything about the fault data captured and as such assigning the weights that would work in that particular case becomes a Herculean task. One good practice, and also corroborated by [56], is to assign the weights from a Gaussian distribution characterized by a zero mean and some finite variance. It is important to note that numerator in the  $\epsilon$  formula above can either be  $\sqrt{2}$  or  $\sqrt{6}$  as deemed fit. If it is  $\sqrt{2}$ , it represents a normal distribution and  $\sqrt{6}$  implies a uniform distribution. However, some papers in the literature have provided strategies for different activation functions which is summarized below:

<b>Activation Function</b>	<b>Uniform Distribution</b> $[-\epsilon, +\epsilon]$	<b>Normal Distribution</b>
Hyperbolic Tangent Function	$\epsilon = \sqrt{\frac{6}{n_{in} + n_{out}}}$	$\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$
Sigmoid Function	$\epsilon = 4 * \sqrt{\frac{6}{n_{in} + n_{out}}}$	$\sigma = 4 * \sqrt{\frac{2}{n_{in} + n_{out}}}$
ReLu (and its variants)	$\epsilon = \sqrt{2} * \sqrt{\frac{6}{n_{in} + n_{out}}}$	$\sigma = \sqrt{2} * \sqrt{\frac{2}{n_{in} + n_{out}}}$

Table 3-1: Weight Initialization Scheme under Normal or Uniform Distribution [56]

This thesis employs the ‘‘He’’ weight initialization technique because the author finds that it converges in simulation better than any other initialization techniques.



### 3.6 Inverter Model or Snapshot of the studied model

The single phase voltage source inverter is designed using Simulink which is utilized to convert DC power from a battery source to AC by keeping the output voltage of the inverter at the rated voltage irrespective of a fluctuating load. The inverter model implements a full-bridge power converter which comprises of four (4) IGBT switches and modeled with IGBT/diode pairs controlled by firing pulses produced by a PWM generator (0/1 signals). This pulse-width modulation (PWM) technique is applied to control switches. Further, the measured or output voltage is passed through an RLC sub-circuit for filtering purposes. The carrier frequency of the single-phase inverter is 1620Hz and the sampling time for the simulation is 30ms.

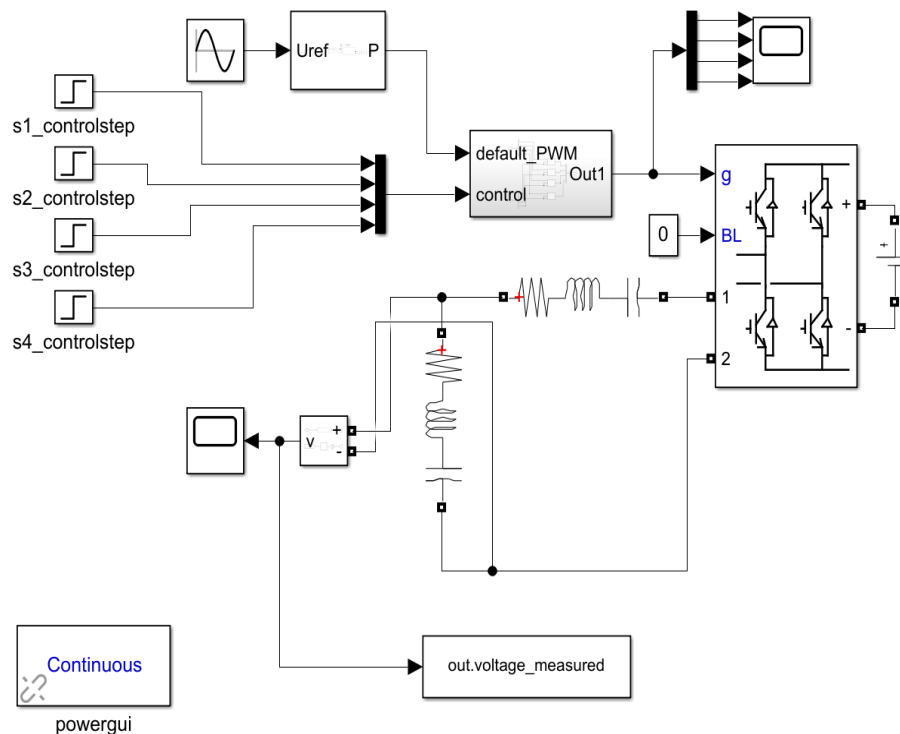


Figure 3-1: The Snapshot of the studied model

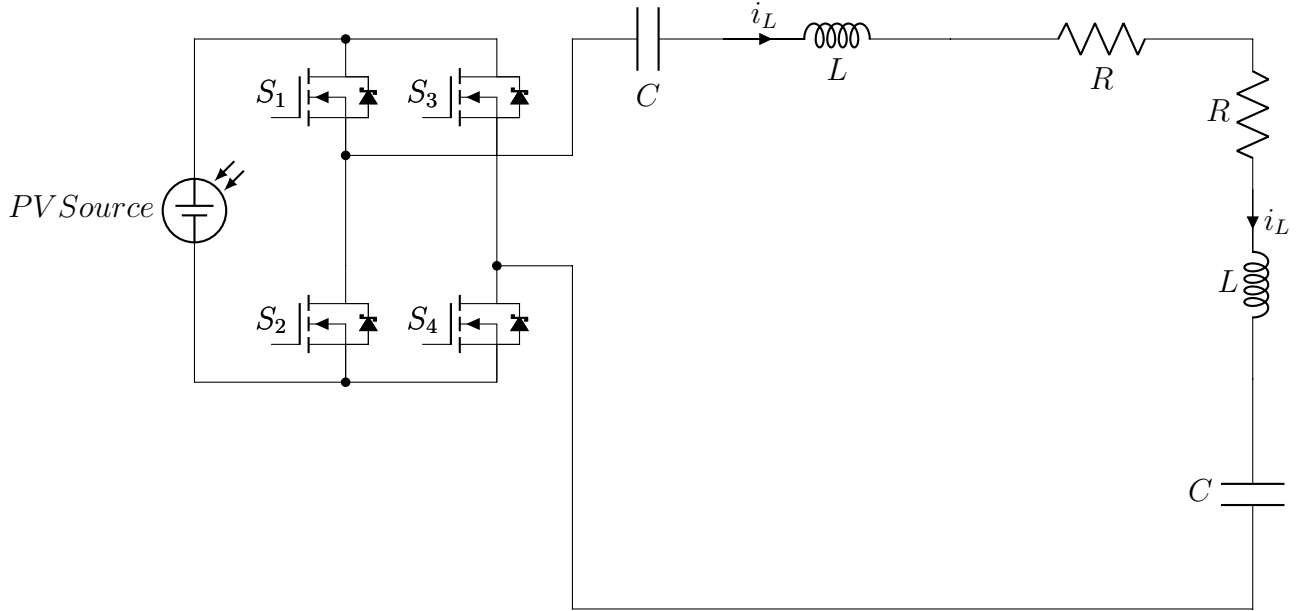


Figure 3-2: The Simulink Model of the Circuit Under Test

### 3.7 Fault Detection and Fault Generation

In this thesis, fault detection was formulated as a multi-classification problem where the labels are normal, open-circuit fault and short-circuit fault. According to [57], Fault Detection Rate (FDR), which is an accuracy index is defined as:

$$FDR = \frac{\# \text{ of faulty samples with faulty label}}{\# \text{ of faulty samples}}$$

On the other hand, fault capture or generation is generated in a single-phase inverter system to check the performance of the studied model under normal or different other fault conditions. We generate single and multiple faults by opening the IGBTs of the inverter so that the system can receive the input signal without respective phases.

In the case of double faults, by convention, there is a high possibility of fault in two gates used in the same phase (*Say, S<sub>1</sub>&S<sub>4</sub> or S<sub>2</sub>&S<sub>3</sub>*). In practice, Phase mismatch or complete phase missing can be noticed in the case of short-circuit or open-circuit faults.

### 3.8 Data Description

The data used in this study covers basically the power spectral estimates, encoded data and fault time measured which is directly dependent on the Output Voltage sensed by a voltage measurement block in MATLAB/Simulink. This collected data is used to train the Neural Network.

### 3.9 Data Analysis

In Neural Network, the data analysis stage involves data pre-processing and feature extraction. These processes will be explained in subsequent lines.

#### 3.9.1 Pre-processing of Data

The data used in this study is composed of power spectral estimates, specifically ten (10) different data points in the power spectral density plot, encoded data and recorded fault times for each simulation instance. The purpose of this pre-processing stage is generally to compensate for known or unknown distortions introduced by the sensor and/or the environment. This stage involves operations such as scaling and filtering. The only pre-processing performed on the fault capture data is to normalize the Power Spectral Density (PSD) estimate to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values.

#### 3.9.2 Power Spectral Density estimate of the raw data

This is a critical stage in feature engineering and data analysis. The pwelch-based method of spectral analysis was used in the thesis to measure the energy content of the input signal and used for feature extraction. The matlab syntax is given as:  $[p_{xx}, w] = \text{pwelch}(x, \text{window}, \text{noverlap}, w)$ . The Power Spectral Density (PSD)

returns a one-sided Welch PSD estimates at a normalized frequencies specified in the vector,  $w$ .  $x$  is the input signal. Window is the row or column vector or an integer. noverlap is the number of overlapped samples. The vector  $w$  must contain at least two elements, because otherwise the function interprets it as nfft. nfft specifies the number of discrete Fourier transform (DFT) points to use in the PSD estimate.

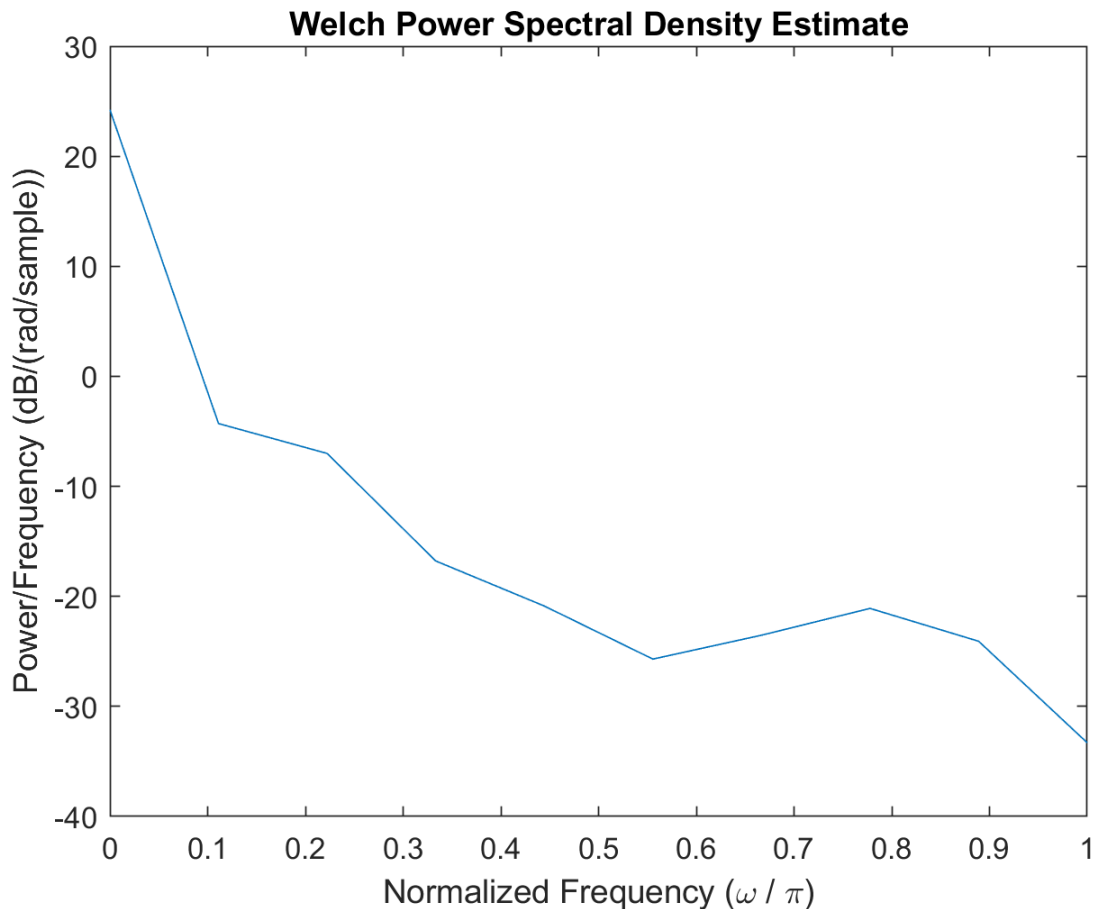


Figure 3-3: Welch Power Spectral Density Estimate

### 3.10 Flow diagram of the proposed Neural Network Process

Figure 3-4 shows the flow chart of the fault detection and classification system proposed in this thesis. The Single phase inverter which operates on the Pulse Width Modulation(PWM) technique receives DC Supply from a 20V ideal DC voltage source and then the inverter converts it to AC with a desirable frequency of 60Hz. The in-

verter produces an output voltage waveform and this sine wave which has harmonic distortions is subjected to an RLC filter. The resulting output voltage is collected which then becomes our raw data. This raw data goes through two major data analysis step which is: pre-processing and feature extraction. At the data pre-processing stage, the raw fault data is cleaned, smoothed and normalized to ensure accurate, efficient, or meaningful analysis. Thereafter, high level features or attributes are extracted from the data using the pwelch's method to produce the training data. Then, the process of model construction begins with offline training and appropriate tuning. The training data goes through the feed forward network which is made of the input layer, hidden layer and output layer with a 10-5-3 node configuration respectively. The nodes calculate the weighted sum of their input signals and output the result of the activation function with the weighted sum. The output at the output node becomes the open-loop result. According to the back-propagation methodology anchored on the generalized delta rule earlier discussed in this thesis, the error which is the difference between the correct output or target data and the resulting output is back-propagated until it reaches the hidden layer and weights are updated to reduce the error. This whole process is repeated for all training data points until the error reaches an acceptable tolerance level as retraining the model with the same data usually improve the model. The next stage is model selection and evaluation. At this stage, the best model is chosen and appropriately deployed for different classification applications as deemed fit.

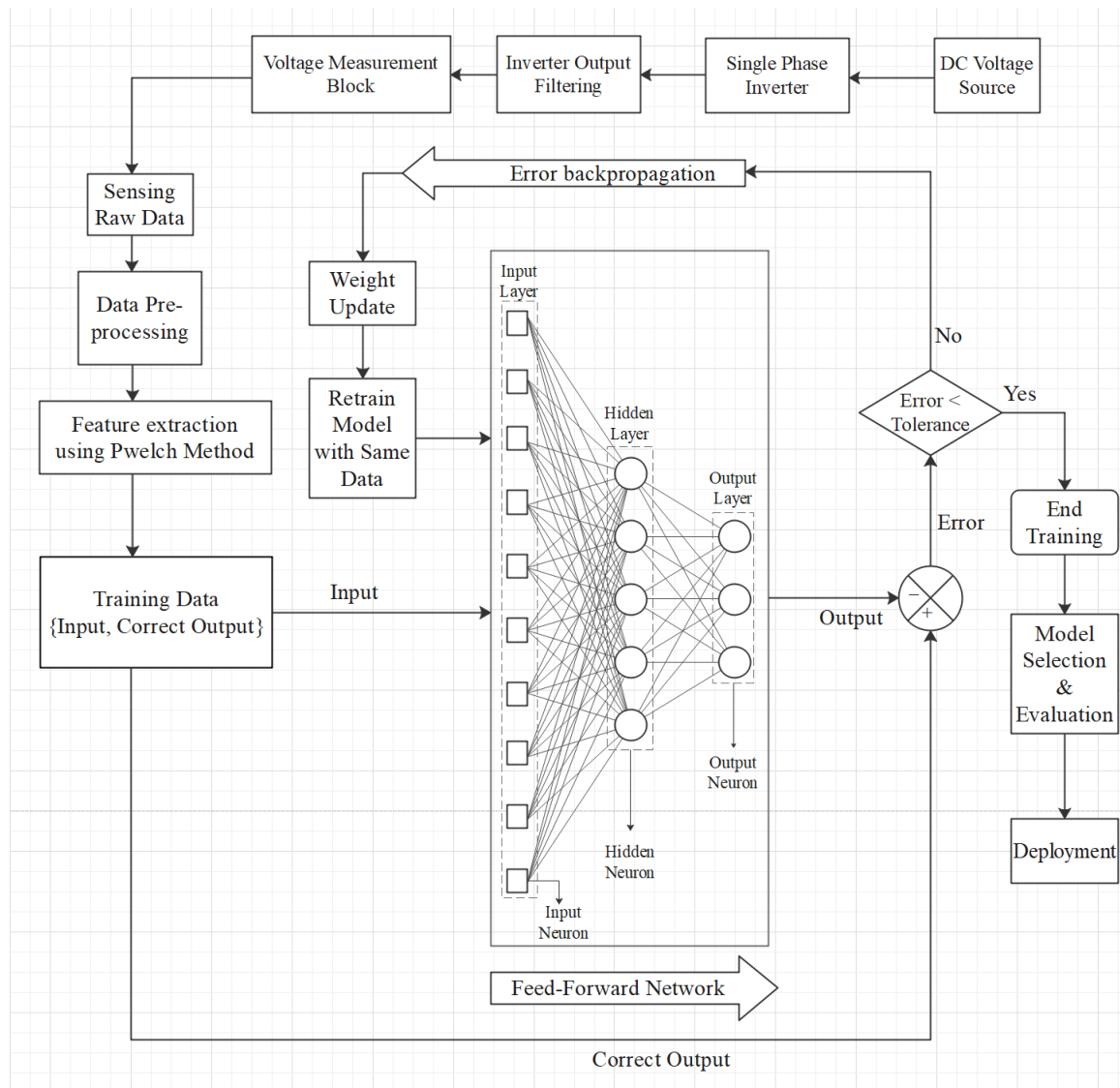


Figure 3-4: Flow diagram of the proposed Neural Network Process

### 3.11 Performance Timing for Faults

This section measures performance of the code in terms of the time elapsed using the `tic/toc`, `clock` and `CPU time` commands in MATLAB. The various performance timing functions help to estimate how long the feature extraction took from the standpoint of the raw voltage data collected. The “`tic/toc`” command allows “`toc`” read the elapsed time from the stopwatch timer initiated by the “`tic`” command. The “`clock`” command records current date and time as date vector but the `clock` function is based on the system time. The “`cputime`” command returns the total CPU time (in seconds) used

by the MATLAB application from the time it was started. A Dell G7 15 (7588) gaming laptop is used for calculation because the calculation time for the same program code is different but around an average value irrespective of computer's operating speed. Table 3-2 below shows gives a detailed performance timing information with respect to the different iterations considered. It took an approximate time of 4 hours, 2 hours and 9 minutes to compute 20,000, 10,000 and 1,000 iterations respectively.

	Elapsed Time (Secs)		
<b>Epochs</b>	<b>Stop watch Time</b>	<b>CPU Time</b>	<b>Real Time</b>
1,000	0.630	0.844	532.53
10,000	0.646	0.859	5848
20,000	0.504	0.650	13667

Table 3-2: Performance Timing Computations

# Chapter 4

## Results and Discussion

### 4.1 Training, validation, and testing of neural networks

The neural network model was trained for different epochs which ranges from 1,000, 10,000 and 20,000 training set respectively. After the training was completed, the performance of each neural network was evaluated using two indicators: the loss or error function and classification Accuracy. In this thesis, Cross-entropy and Mean Square Error ( $MSE$ ) are types of loss functions used when training neural network models. Conversely, the Neural Network toolbox in Simulink of MATLAB uses the entire data set in three parts; Training, Validation and Testing. Training data (which captures a 70% of the total data samples) are presented to the network during training and the network is adjusted according to its error. Also, validation data (which captures a 15% of the total data samples) measures network generalization, and halt training when generalization stops improving and testing data (which also captures a 15% of the total data samples) have no effect on training and so provide an independent measure of network performance during and after training.

The performance of the trained neural network is tested through the Confusion matrix or classification accuracy as the case may be. The Confusion matrix or error matrix of the training, validation and testing phases is a tabular visualization of the model predictions versus the ground-truth labels. Each row of confusion matrix represents the instances in a predicted class and each column represents the instances in an actual class. The confusion matrix shows that the efficiency of the trained neural



network in terms of its ability to check and identify correctly the three possible types of the fault.

## 4.2 Evaluation Metrics

Metrics are used to monitor and measure the performance of a model during training and testing. In this thesis, two typical classification-related metrics were considered. They are;

- (i) Classification Accuracy
- (ii) Loss function

### 4.2.1 Classification Accuracy

Classification Accuracy are learning curves defined as the number of correct predictions divided by the total number of predictions, multiplied by 100. It can be mathematically expressed as:

$$\text{Classification Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \times 100$$

### 4.2.2 Loss Function

Loss functions are functions or learning curves (not necessarily a metric) that show a measure of the model performance and are used to train a machine learning model, indicating the fraction of samples which are misclassified.

These learning curves provide an indication of three things: how quickly the model learns the problem, how well it has learned the problem, and how noisy the weight updates were to the model during training.

### 4.3 Fault Description

This section represents the different categories of fault with specific Fault ID which would be referred to in subsequent lines.

<b>Fault De- scription</b>	<b>Fault ID</b>	<b>Faulty Com- ponent</b>
No Fault	1	Normal Operation
Single Fault	2	S1
	3	S2
	4	S3
	5	S4
Double Fault	6	S2 & S3
	7	S1 & S2
	8	S3 & S4
Triple Fault	9	S1, S2 & S3
	10	S2, S3 & S4
Multiple Fault	11	S1, S2, S3 & S4

Table 4-1: Detected Faults and Description

Also, Fault IDs can be grouped into five (5) categories based on its Fault description or system condition states for better organization and reference:

- Fault type 1: {1}
- Fault type 2: {2, 3, 4, 5}
- Fault type 3: {6, 7, 8}

- Fault type 4: {9, 10}
- Fault type 5: {11}

#### 4.4 Fault Classification and loss function results for 1,000 iterations

The Fault classification problem is solved using an input data and target data derived with one-hot encoding for different iterations. The model of our neural network has a structure expressed as:  $nodelist = [Inod, 5, Onod]$ . Where  $Inod$  is the number of nodes in the Input Layer,  $Onod$  is the number of nodes in the Output Layer and 5 represents the number of nodes or neurons in the hidden Layer. In this analysis, we used the neural network with a single hidden layer whose structure is 10-5-3 and are trained using the Fault Data generated through simulation. In comparing our results with other data-driven method, Table 4-2 summarizes the results from Deep Learning Toolbox (DPLT) and our deep neural network model shows the best overall fault classification rate.

The overall Classification Accuracy and loss function for each fault state for 1000 Epochs is summarized in Table 4-2 and Table 4-3 respectively. The Classification accuracy for normal and some selected faulty state, representing single, double, triple and multiple faults is shown in the figures below, and only one confusion matrix is provided here for brevity.

Figure 4-1 shows good convergence behavior for both mean square error loss and classification accuracy under a normal operating condition of the studied system. From the figure, we can see the model performed well, achieving a classification accuracy of about 100% on the training dataset.

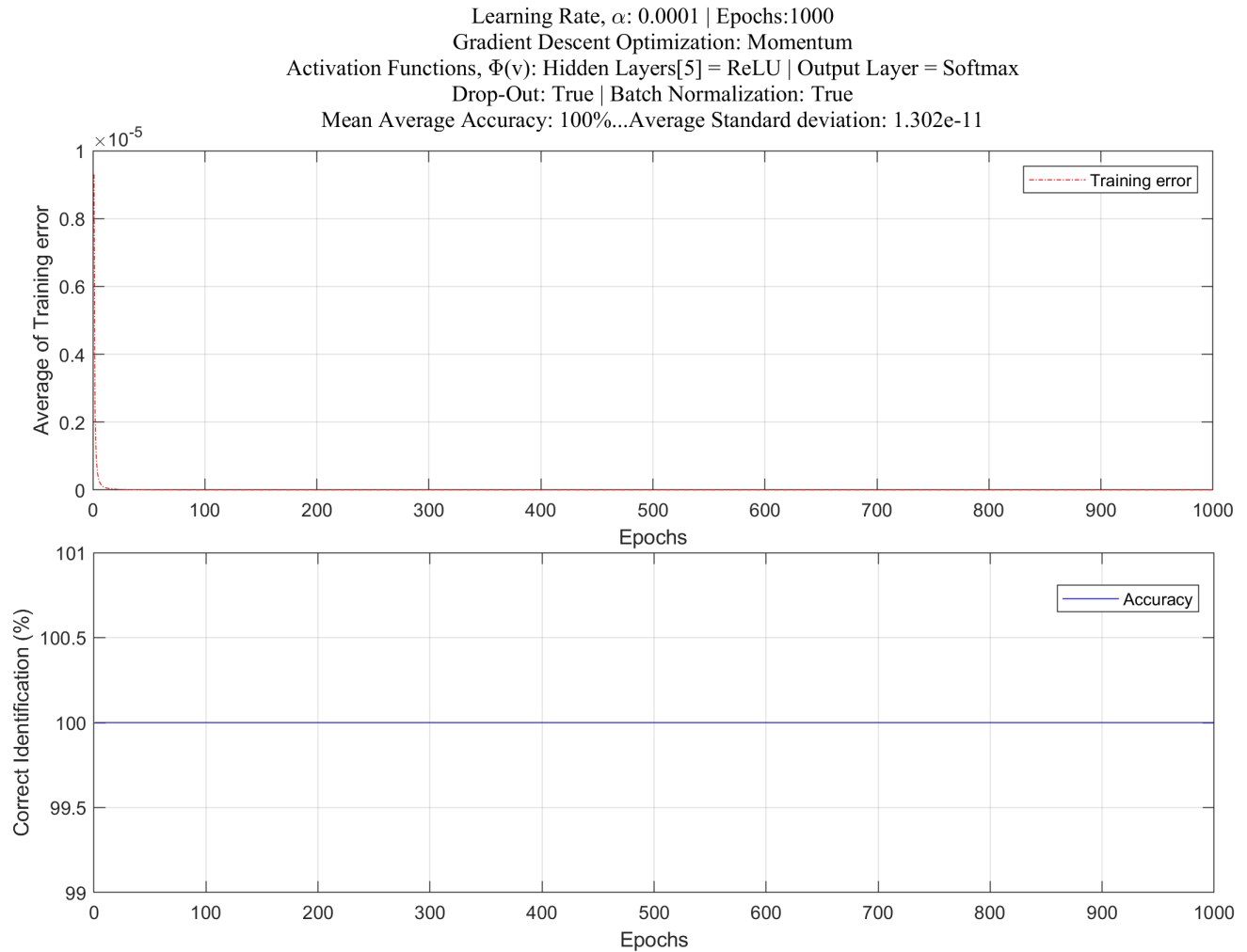


Figure 4-1: Line Plots of Loss and Accuracy over 1000 Training Epochs

Figure 4-2 represent the behavior of a single switch fault (s1) with method 2. From the figure, we can see that the fault classification accuracy for Fault type 2 rises all the way up to 100% which indicates that our network correctly classifies all 1,000 training samples. We can also see that a shallow network is sufficient to detect correctly this fault type. In the first 50 epochs the accuracy rises to just under 94% percent. Finally, at around epoch 100 the classification accuracy pretty much stops improving. Later epochs merely see small stochastic fluctuations near the value of the accuracy at epoch 100. In other words, what our network learns after epoch 100 no longer generalizes to the test data and so it's not useful learning. We say the network is over-fitting or

over-training beyond epoch 100.

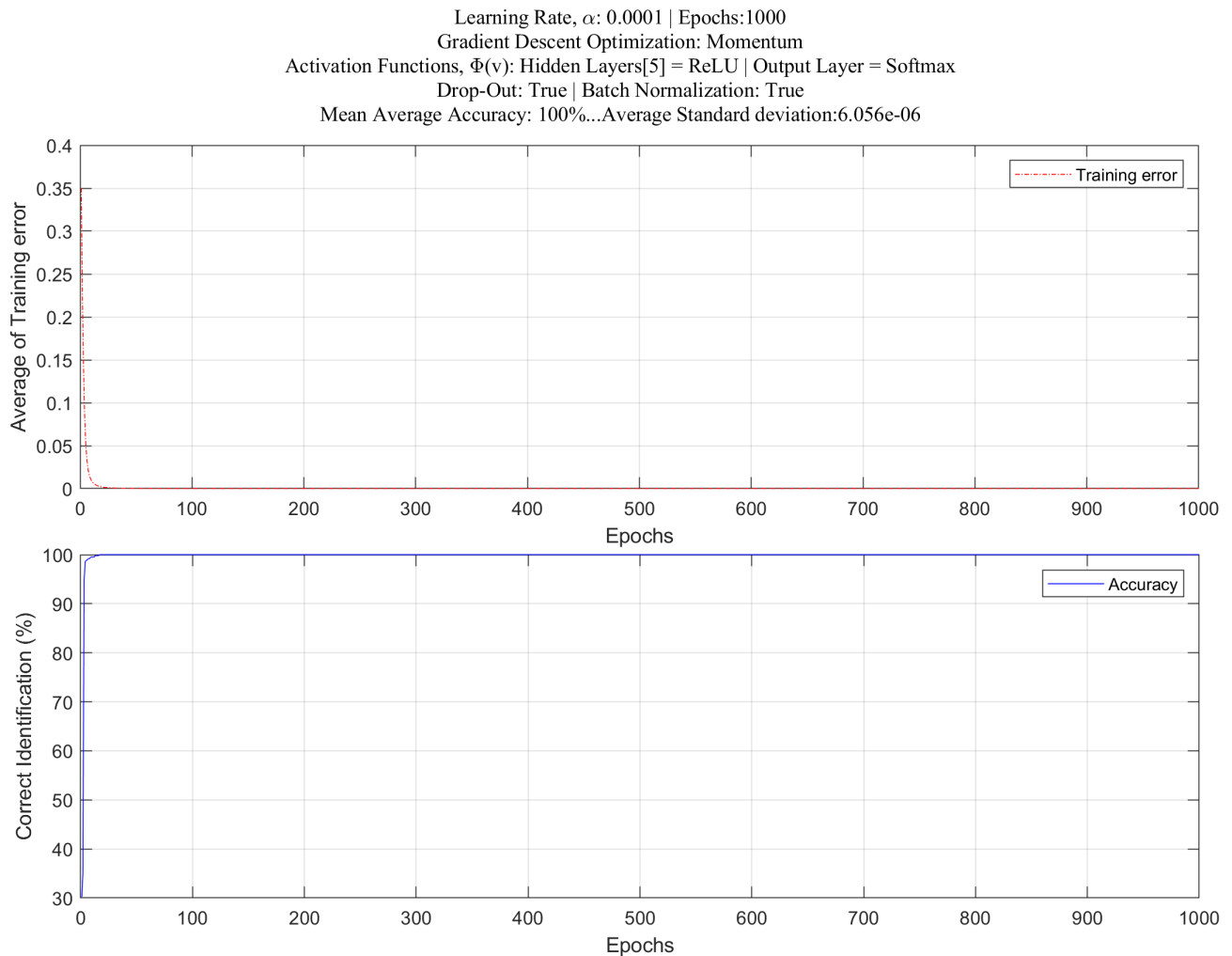


Figure 4-2: Line Plots of Loss and Accuracy over 1000 Epochs for single switch

Figure 4-3 shows the result of (s1) switch which is typical of a single switch fault when training with Method 1. The plot also shows the unstable nature of the training process with the chosen configuration. From the figure, we can see that the plot depicts that the noisy updates result in noisy or poor performance throughout the duration of training. The shape of the error surface is bumpy (not as smooth) as other loss functions (trained with the second method) where small changes to the weights are causing large changes in loss.

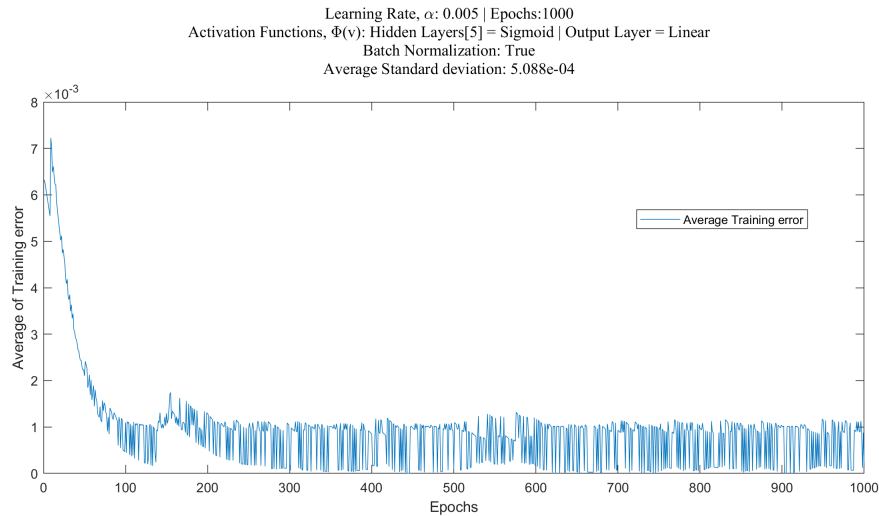


Figure 4-3: Plots of Training error and Accuracy over 1000 Epochs for single switch

Figure 4-4 is created showing two line plots of training error and classification accuracy, all of which are indicative of the Fault type 3. From the figure, we can see that the Fault Classification Accuracy of Fault type 3 ranges from (98.6% to 100%). Therefore, this implies that, our shallow network model is enough to effectively classify this Fault-type for 1000 simulation runs.

Learning Rate,  $\alpha$ : 0.0001 | Epochs:1000  
 Gradient Descent Optimization: Momentum  
 Activation Functions,  $\Phi(v)$ : Hidden Layers[5] = ReLU | Output Layer = Softmax  
 Drop-Out: True | Batch Normalization: True  
 Mean Average Accuracy: 100%...Average Standard deviation: 0.002261

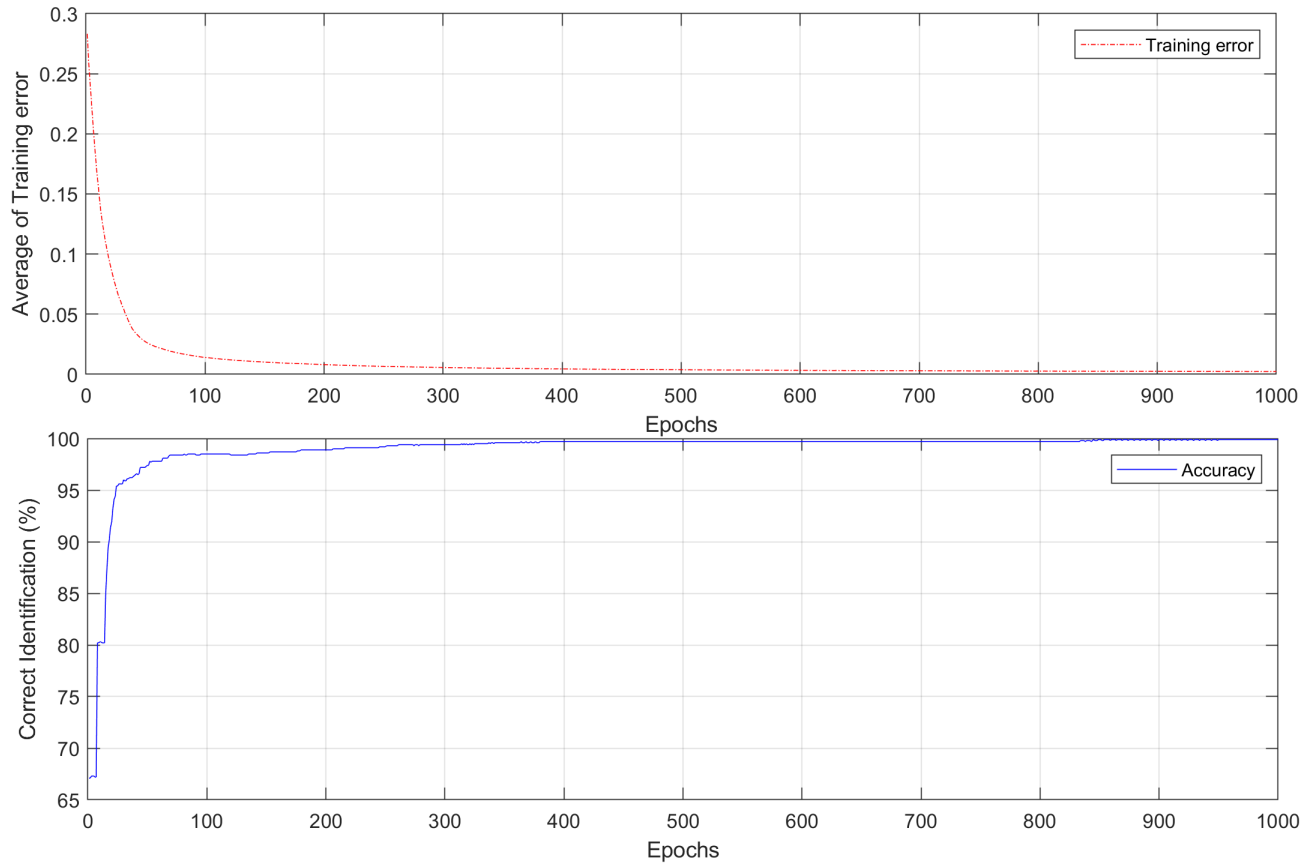


Figure 4-4: Plots of error and accuracy over 1000 Epochs for double switch faults

Figure 4-5 belongs to the family of Fault type 4. From the figure, the Fault classification Accuracy is between the range of (99.4% to 99.8%), depending on training or re-training which actually helps to optimize network on inputs and targets. As we can see, our shallow network model is enough to effectively classify this Fault-type for 1000 simulation runs.

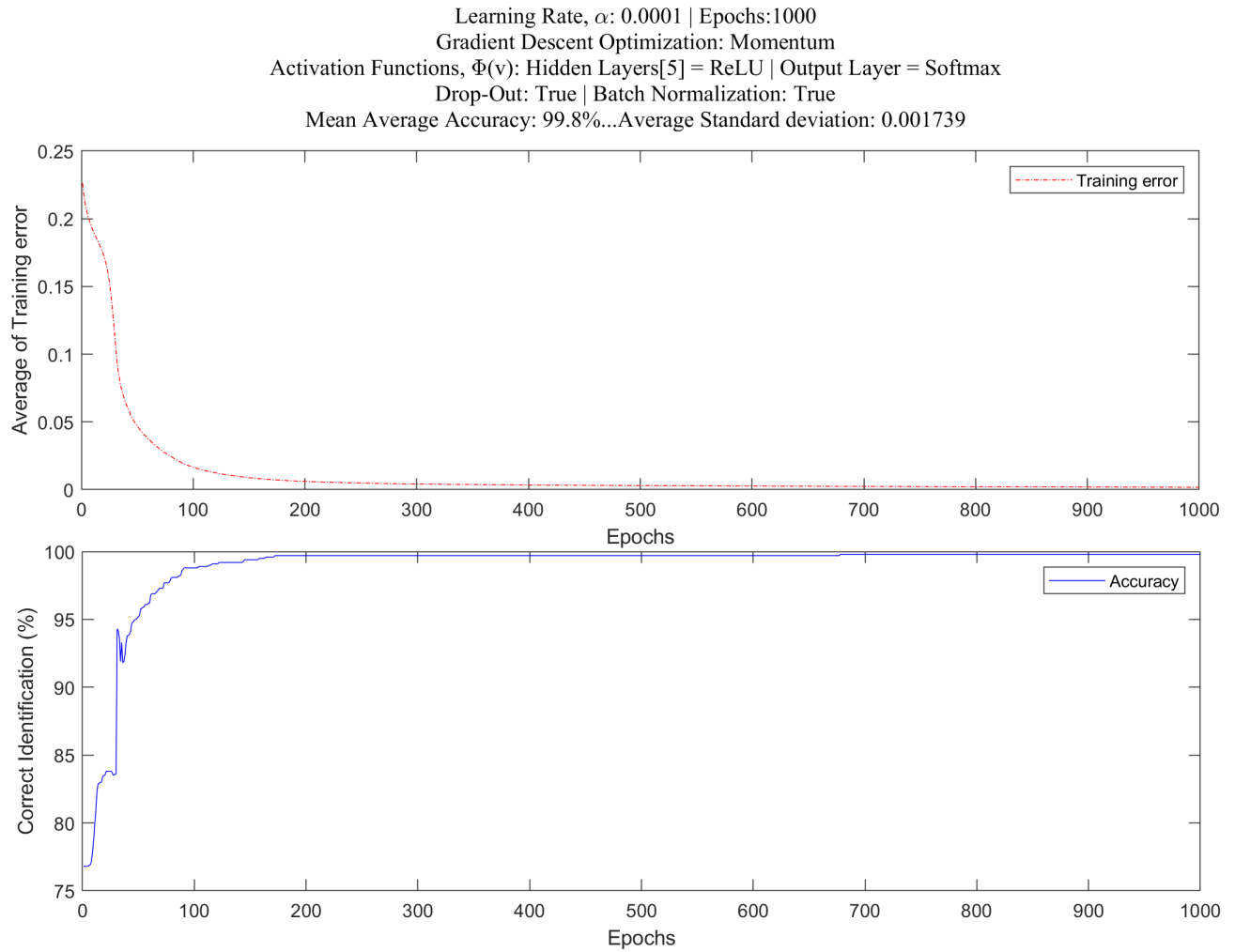


Figure 4-5: Plots of training error and Accuracy over 1000 Epochs for triple faults

Figure 4-6 belongs to the family of Fault type 5. In the first 150 epochs, the accuracy rises up to 90%. From the figure, the overall Fault Classification Accuracy is around 98.6%.



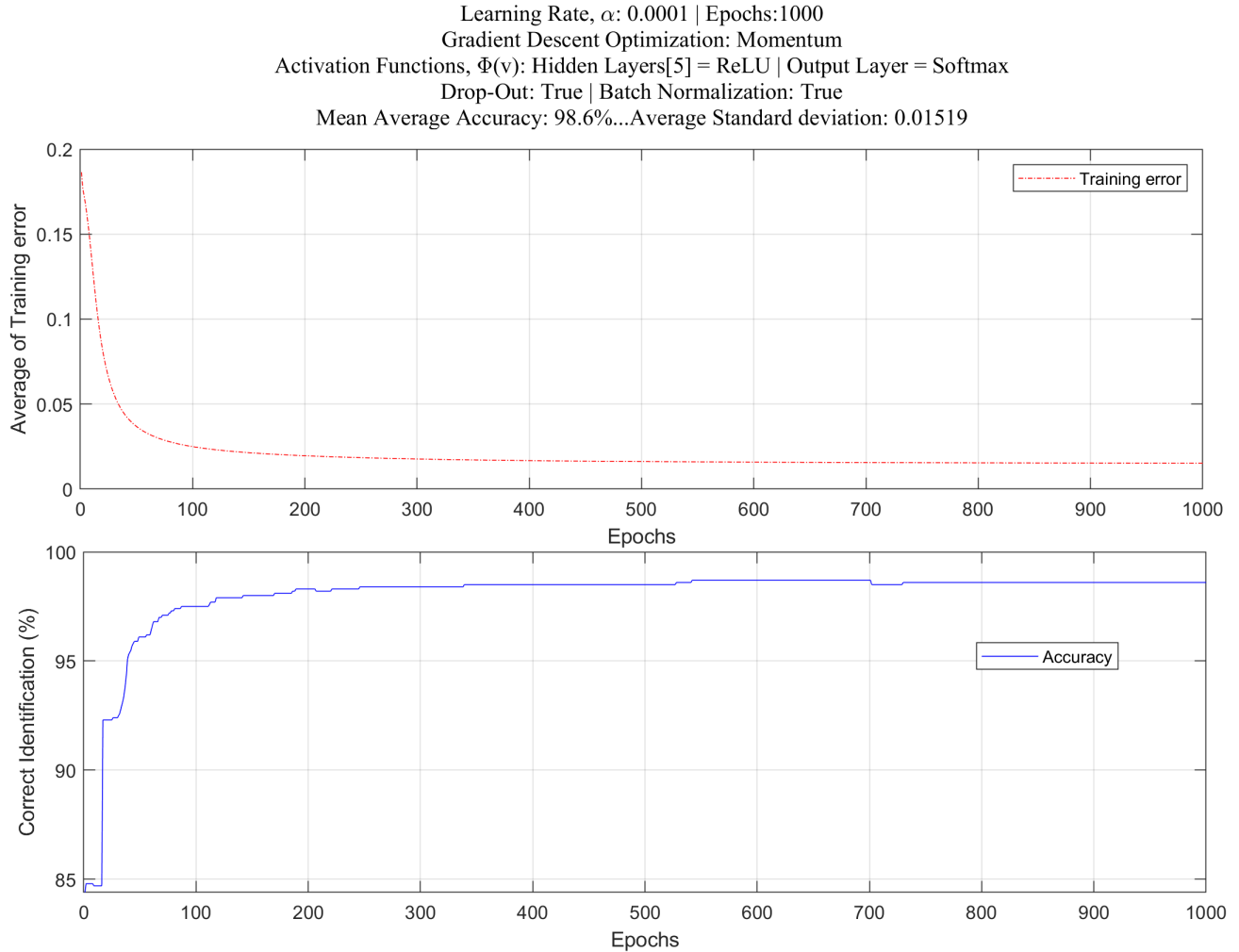


Figure 4-6: Plots of MSE Loss and Accuracy over 1000 Epochs for multiple faults

Figure 4-7 shows the training performance plot using the Deep Learning Toolbox (DPLT). As a basis for comparison, the figure indicates the result of the Fault type 2 (which in this case is the s1 switch). The overall cross-entropy error of the trained neural network is  $4.2026e - 07$  and it can be seen from the figure that the testing and the validation curves have similar characteristics which is an indication of efficient training. The result also shows that our method had a better accuracy in terms of classification and training error plots. This training stopped when the validation error occurred at iteration 23. From the figure, the result is reasonable because of the following considerations; the final cross-entropy error is small, the test set error and the

validation set error have similar characteristics and there is no significant overfitting that occurred by iteration 23 (where the best validation performance occurs).

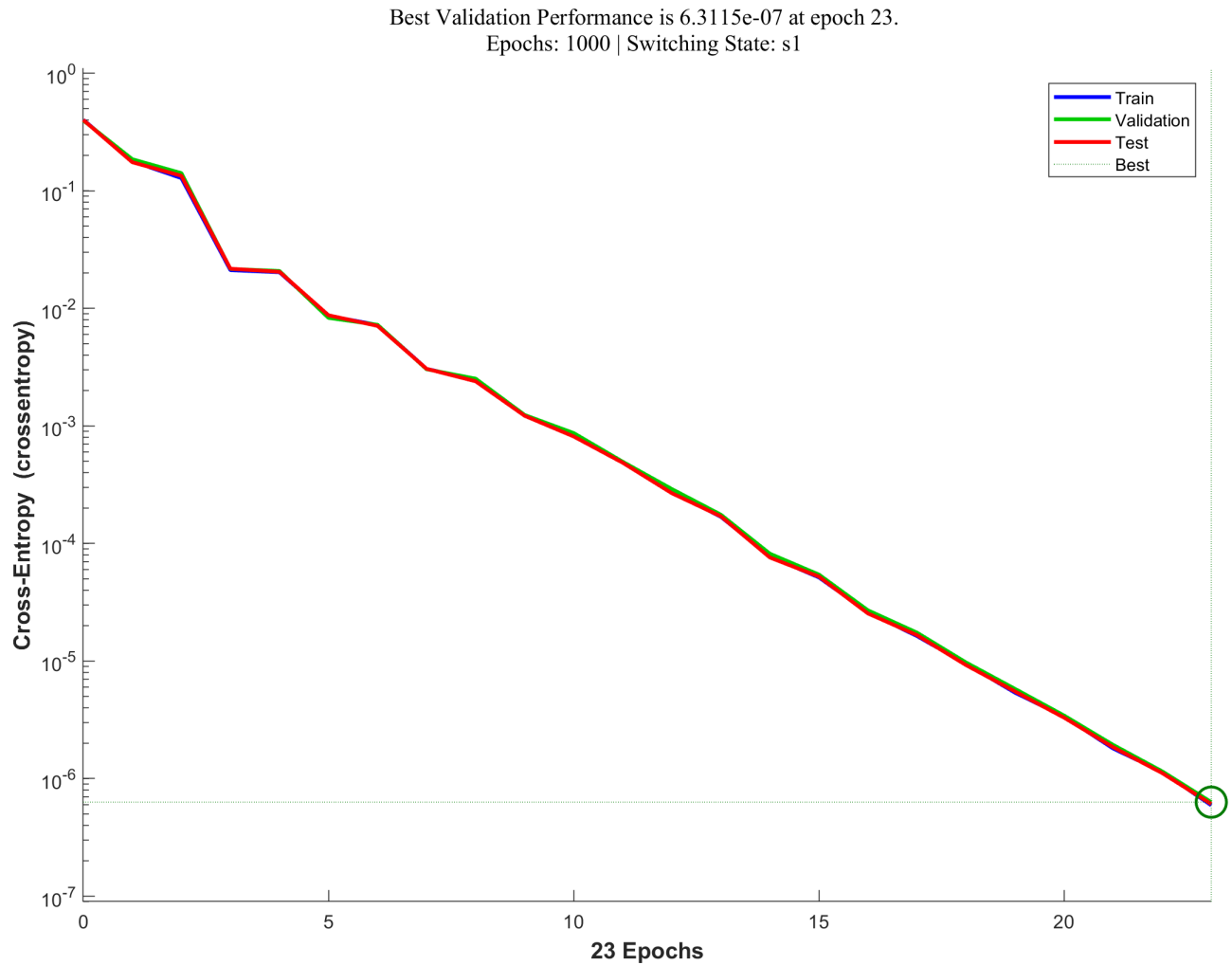


Figure 4-7: Line Plots of Cross-entropy Loss over 1000 Epochs under normal condition

Figure 4-8 shows that the efficiency of the trained neural network in terms of its ability to check fault is 100%.

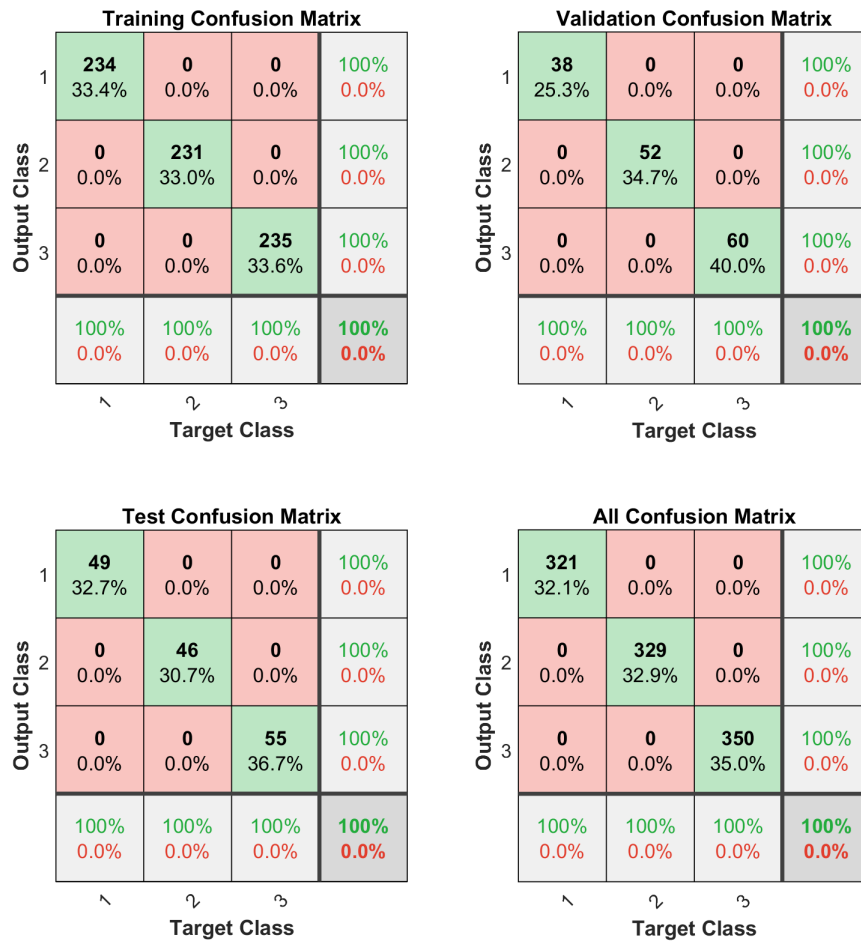


Figure 4-8: Confusion matrix over 1000 Training Epochs under faulty condition

Table 4-2 compares the results of two different methods for 1000 epochs. In terms of classification accuracy, our shallow neural network model shows the best overall fault classification rate.

<b>Fault ID</b>	<b>DPLT</b>	<b>Ours</b>
1	100	100
2	100	100
3	100	100
4	100	100
5	100	100
6	98.4	98.6
7	99.8	100
8	100	100
9	98.1	99.8
10	98.4	99.6
11	98.1	98.6
<b>Overall</b>	<b>99.35</b>	<b>99.69</b>

Table 4-2: Fault Classification rates(%) of different neural network models

Table 4-3 summarizes the results from different data-driven methods. The result obtained using DPLT and our methods were used for comparison. DPLT in Table 4-3 used Sigmoid and Softmax Activation functions for hidden layer and output layer respectively and the network is trained with scaled conjugate gradient back-propagation (trainscg). Method 1 in Table 4-3 used Sigmoid and Linear Activation functions for hidden layer and output layer respectively. Method 2 in Table 4-3 used ReLu and Softmax activation functions for their hidden layer and output layer respectively. The Method 2 was formulated to circumvent the problem of non-convergence and instability associated with Method 1. After thorough analysis, our proposed networks show

the best overall average training error.

<b>Average Training Error</b>			
Fault ID	DPLT	Our Method	
		Method 1	Method 2
1	7.402e-17	4.496e-07	1.212e-11
2	4.203e-07	5.088e-04	2.422e-07
3	1.884e-07	5.000e-04	6.156e-07
4	7.099e-07	6.314e-04	7.896e-07
5	4.997e-07	4.933e-04	3.737e-07
6	2.691e-02	1.147e-03	1.366e-02
7	3.910e-05	1.049e-03	2.261e-03
8	2.256e-04	7.396e-04	3.662e-04
9	1.095e-02	1.200e-03	1.738e-03
10	3.338e-02	1.331e-02	5.619e-03
11	2.224e-02	9.243e-04	1.519e-02
<b>Overall</b>	<b>8.523e-03</b>	<b>1.864e-03</b>	<b>3.531e-03</b>

Table 4-3: Loss Function of different methods over 1,000 Epochs

#### 4.5 Fault Classification and loss function results for 10,000 iterations

Now, let us consider the fault classification problem of our studied Inverter process.

The fault classification problem is solved by considering a data sample of 10,000 for

the different fault types aforementioned. In a similar fashion as the result analysis for 1,000 data samples, the classification accuracy and loss results are presented below. Figure 4-9 shows good convergence behavior for both Sum of Squared Error loss and classification accuracy under a normal operating condition of the studied system. From the figure, we can see that the model performed well, achieving a classification accuracy of 100% on the training dataset from the first to the last epoch.

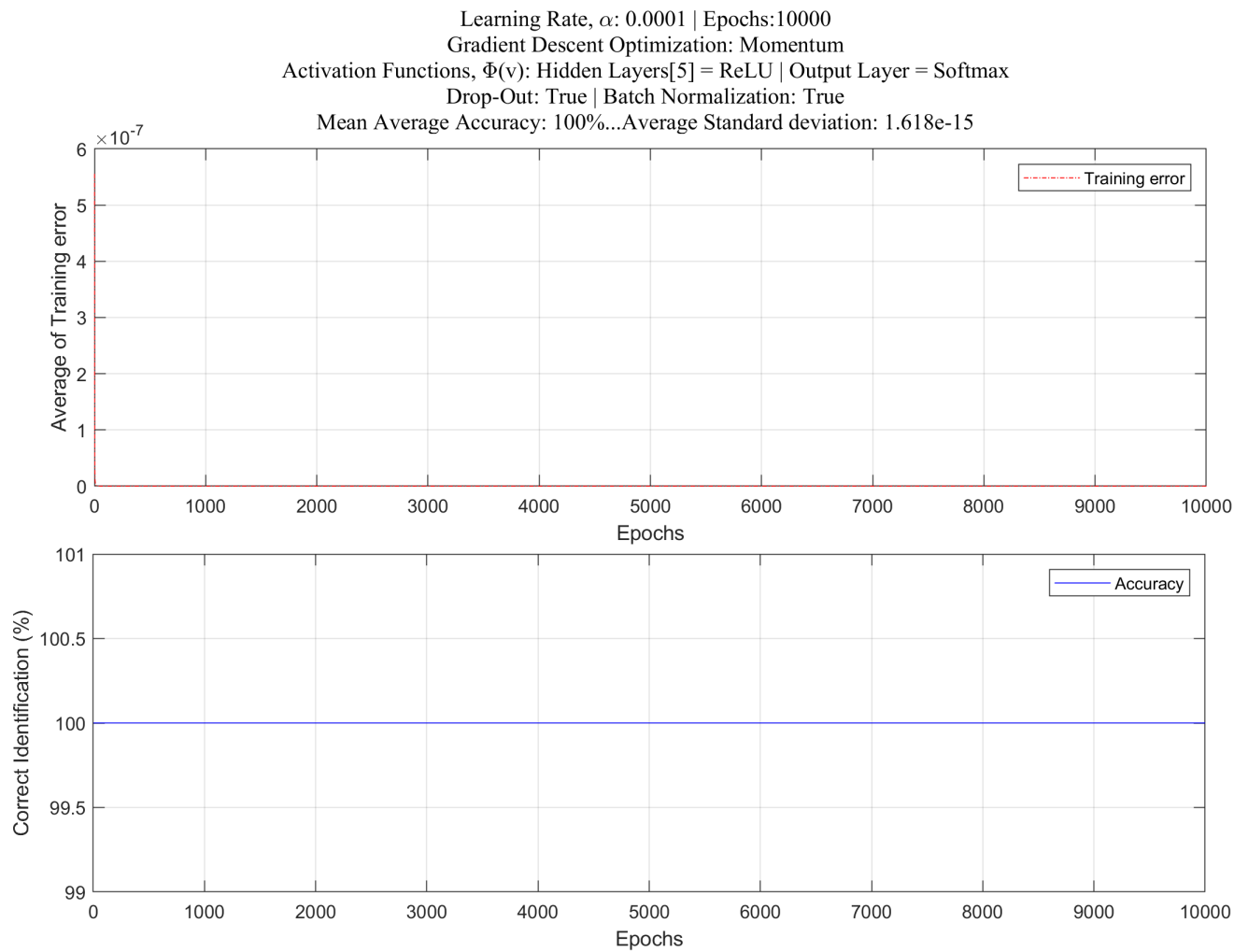


Figure 4-9: Line Plots of MSE Loss and Accuracy over 10000 Training Epochs

As you can see from Figure 4-10, we observed that for Fault type 2, the classification accuracies improve considerably for the first 100 Epochs, owing to the use of more training data.

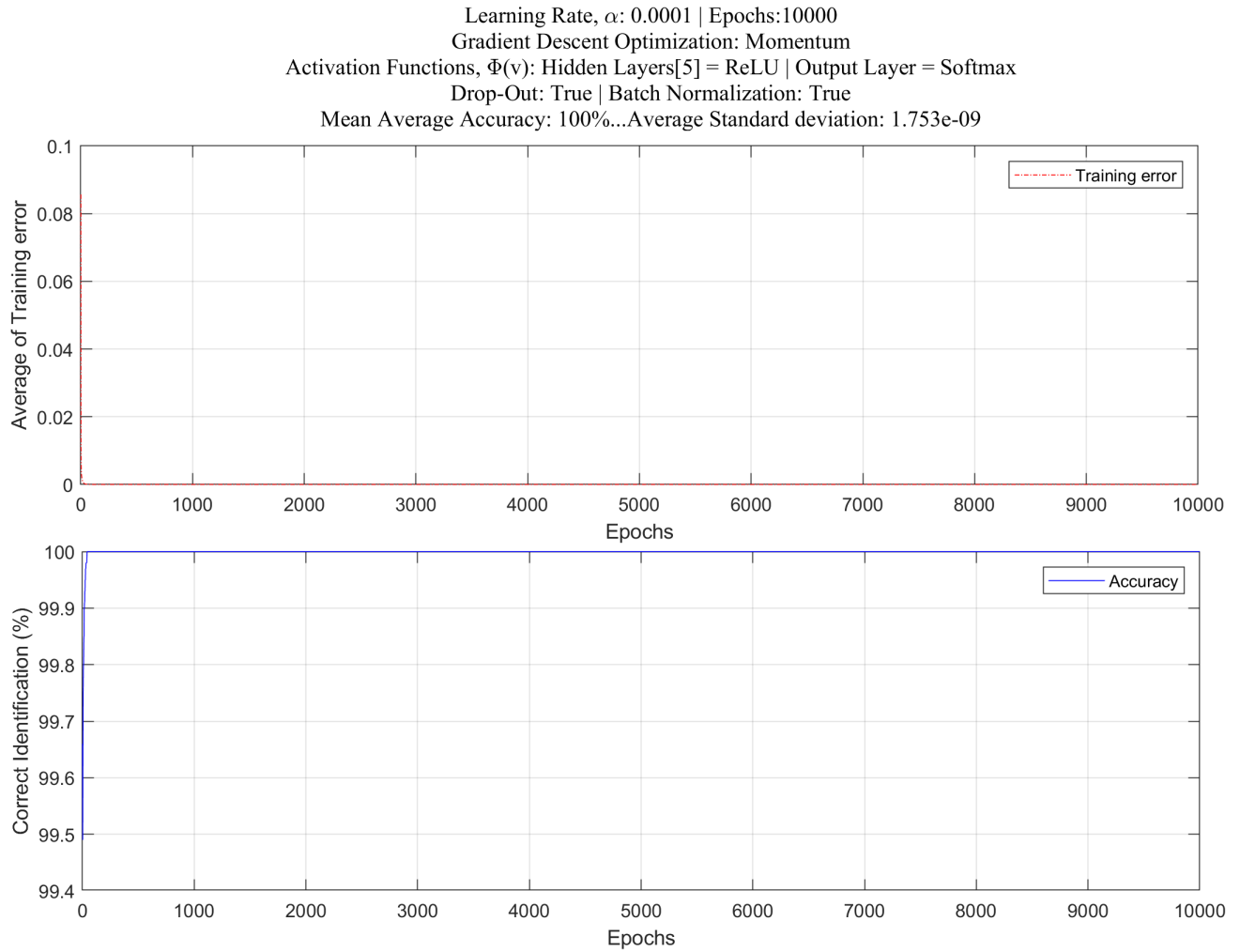


Figure 4-10: Line Plots of MSE Loss and Accuracy for s3 in Fault type 2

Figure 4-11 show how the classification accuracy on the training data changes over time. In the test suite, Fault type 3 which is typical of double switch fault really learns about peculiarities of the training set with an accuracy that rises all the way up to 100%.

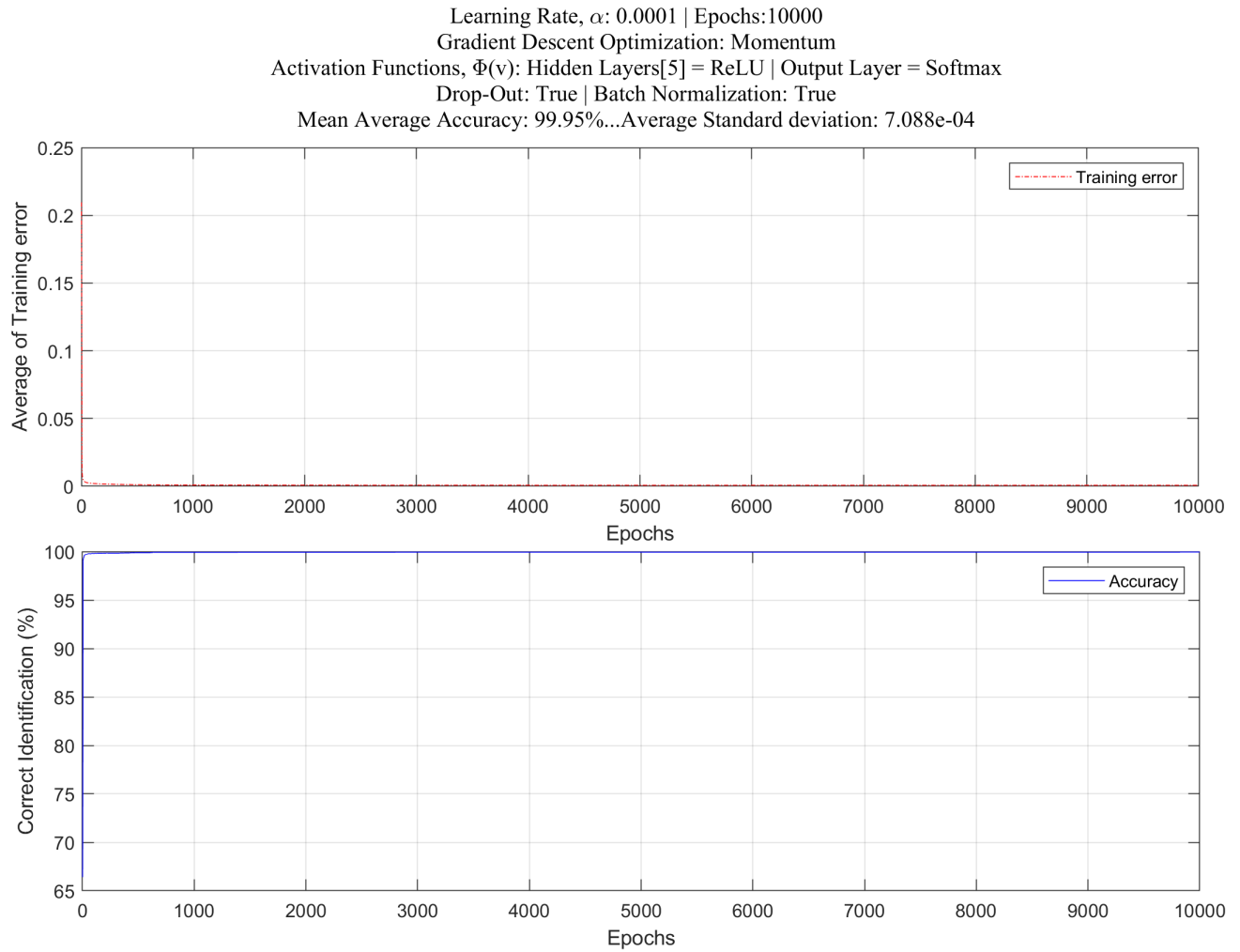


Figure 4-11: Line Plots of Loss and Accuracy typical of Fault type 3

Figure 4-12 represents the behavior of Fault type 4. In this case, we can see the model performed reasonably well, achieving a classification accuracy of about 98.77% on the training dataset.



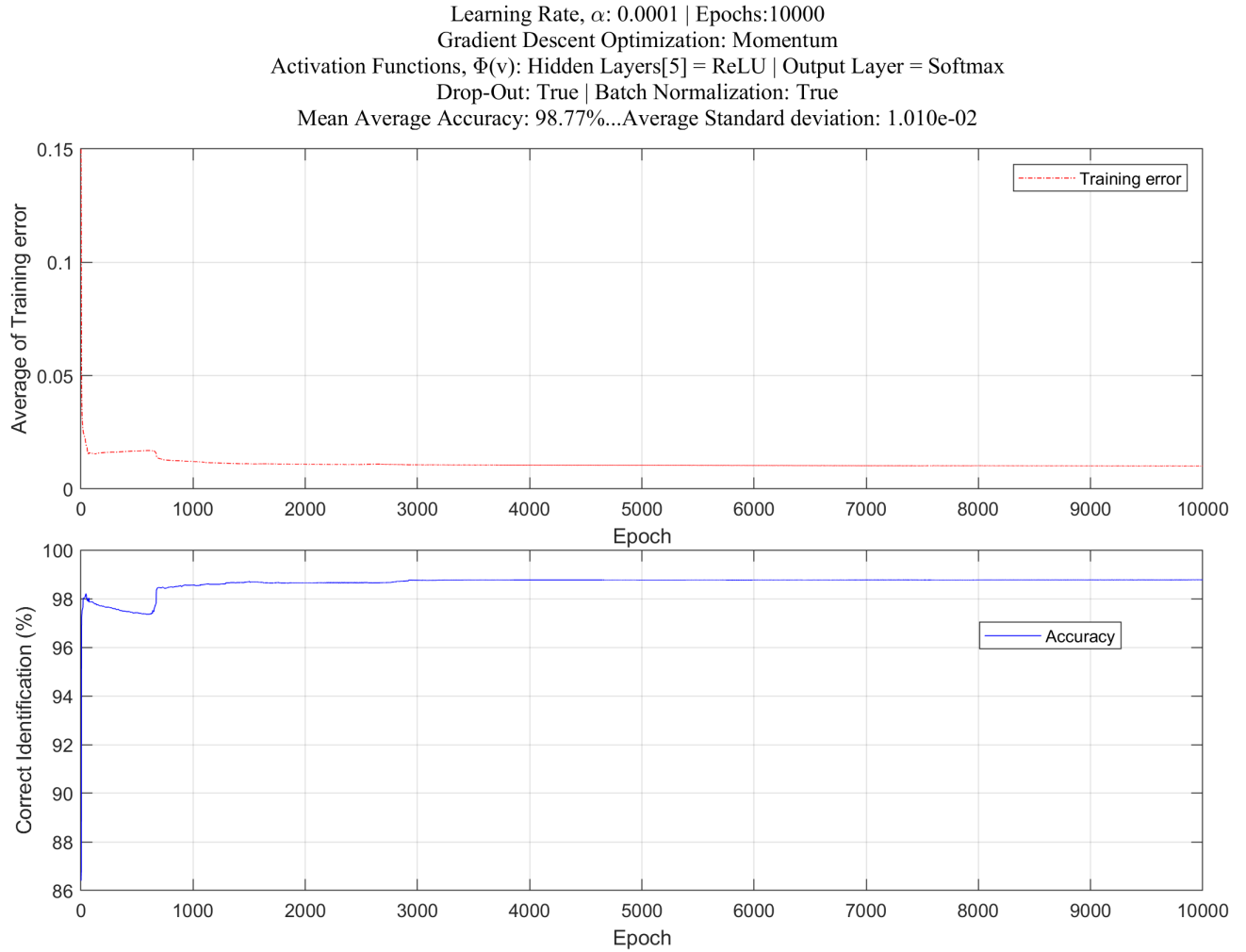


Figure 4-12: Line Plots of Loss and Classification Accuracy typical of Fault type 4

Figure 4-13 represents the behavior of Fault type 5 to training Loss with the method 1. It shows a deviation from the normal. The result is interesting because of its quick convergence and stability when the same configuration gave a bumpy training curve when applied to 1000 iterations and even double switching faults of 10,000 iterations.

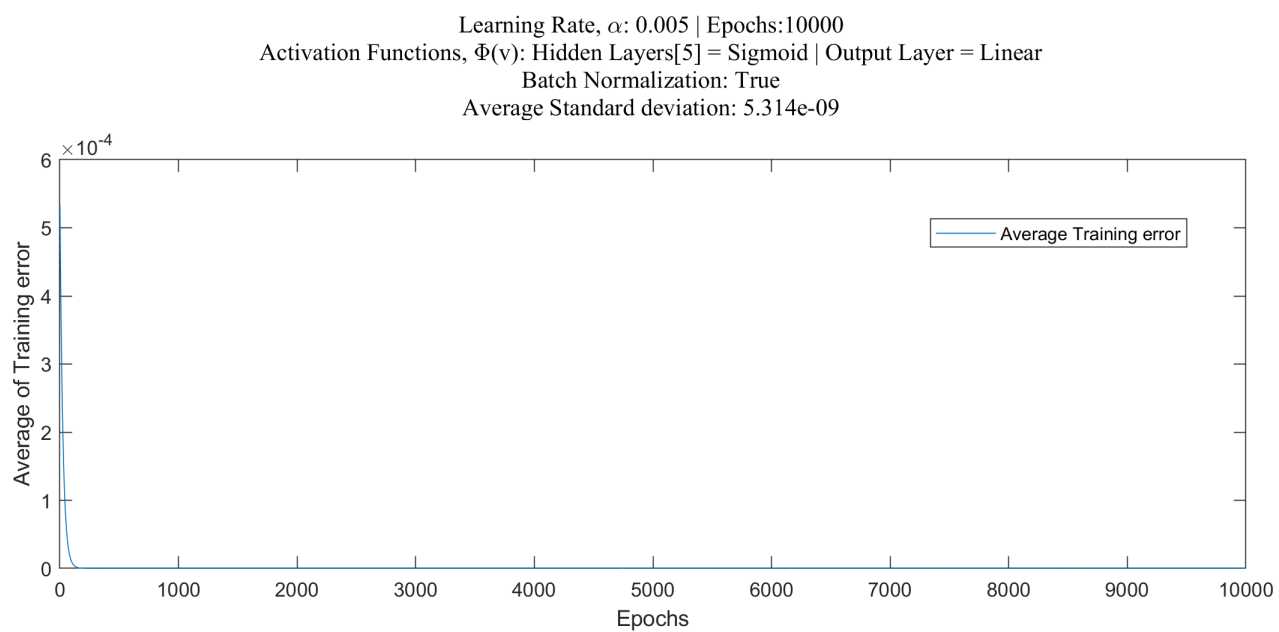


Figure 4-13: Line Plots of Loss and Classification Accuracy typical of Fault type 5

Figure 4-14 represents the behavior of Fault type 5. In this case, we can see the model performed relatively well, achieving a classification accuracy of about 93.14% on the training dataset.

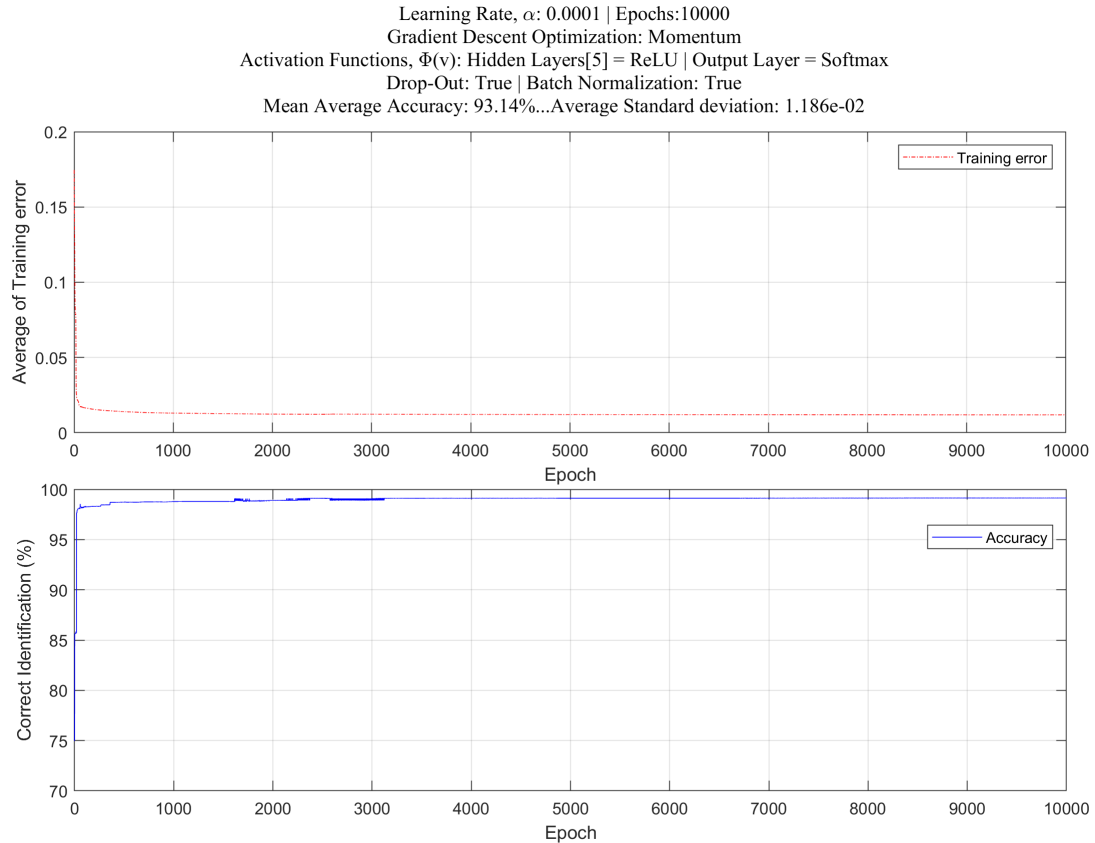


Figure 4-14: Line Plots of Loss and Classification Accuracy typical of Fault type 5

Figure 4-15 shows the training performance plot using the Deep Learning Toolbox (DPLT). The Time Delay Neural Network method contains 1 hidden layer and 5 neurons in each layer. As a basis for comparison, the figure indicates the result of the Fault type 4. The overall cross-entropy error of the trained neural network is  $2.639e - 02$  and it can be seen from the figure that the testing and the validation curves have similar characteristics which is an indication of efficient training. The result shows that our method had a better accuracy in terms of classification and training loss. This training stopped when the validation error occurred at iteration 111. From the figure, the result is reasonable because of the following considerations; the final cross-entropy error is small, the test set error and the validation set error have similar characteristics and there is no significant overfitting that occurred by

iteration 111 (where the best validation performance occurs).

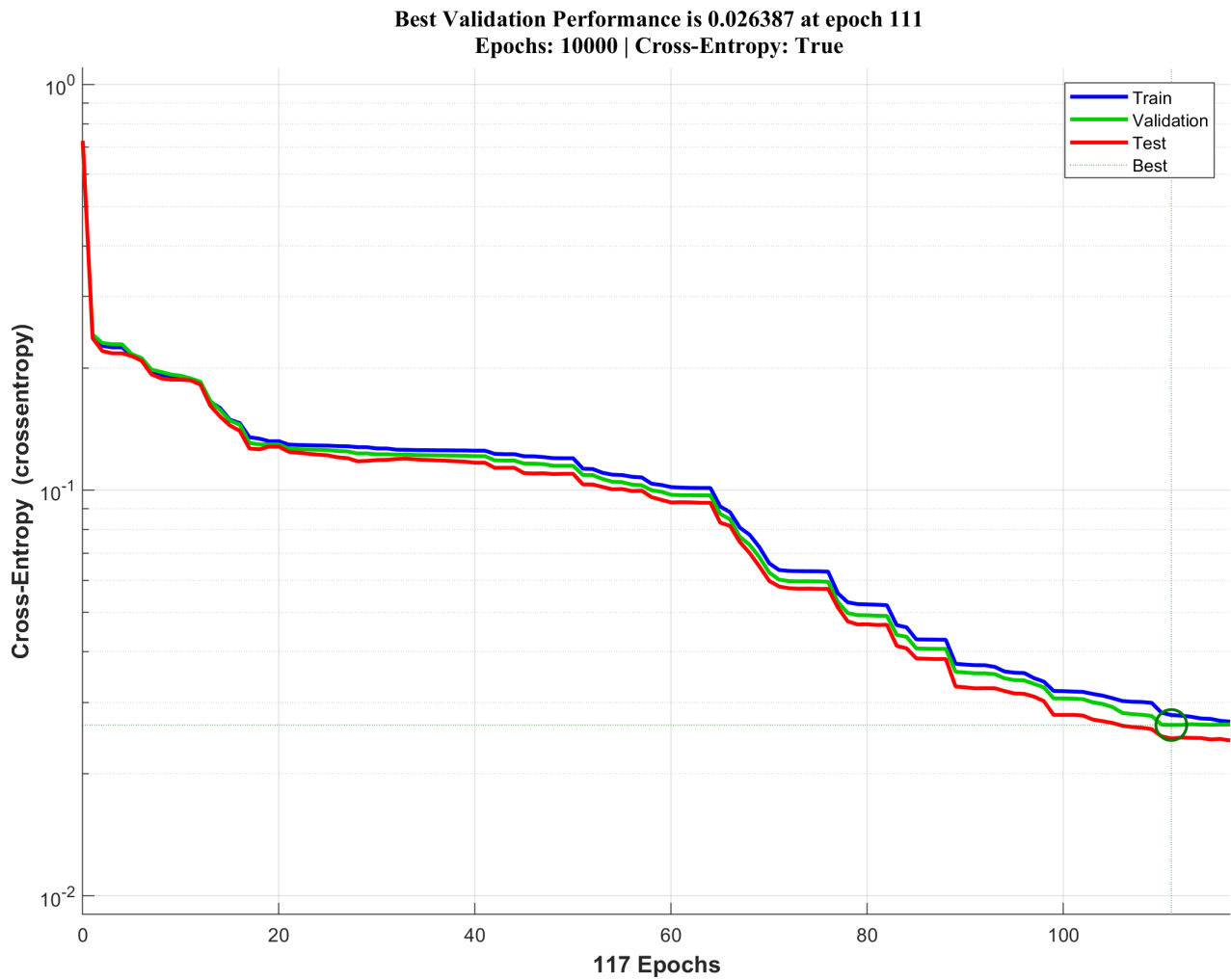


Figure 4-15: Training Performance Plot of Cross Entropy Loss typical of Fault type 4

Figure 4-16 shows that the efficiency of the trained neural network in terms of its ability to check fault type 4, which is about 98.3%.



Figure 4-16: Confusion Matrix typical of Fault type 4 over 10000 training epochs

Table 4-4 compares the results of two different methods for 10000 epochs. In terms of classification accuracy index, our shallow neural network model shows a good overall fault classification rate. Perhaps, could have shown the best overall classification rate if we had the luxury of re-training easily.

<b>Fault ID</b>	<b>DPLT</b>	<b>Ours</b>
1	100	100
2	100	100
3	100	100
4	100	100
5	100	100
6	98.30	96.75
7	98.60	100
8	98.90	99.95
9	98.30	96.38
10	98.50	98.77
11	97.50	93.14
<b>Overall</b>	<b>99.10</b>	<b>98.63</b>

Table 4-4: Fault Classification rates(%) for 10,000 Simulation runs

Table 4-5 detailed the average training error of different methods. However, the results sprang up some surprises. Method 1 which didn't converge well for the different fault types over 1,000 training epochs performed relatively well with 10,000 training epoch. Moreover, majority of the Fault ID converged well in a stable manner and has very small training error. The Neural Network model formulated with Method 1 shows an overall performance, with method 2 in second place and then DPLT in last place.

<b>Average Training Error</b>			
Fault ID	DPLT	Our Method	
		Method 1	Method 2
1	7.402e-17	5.423e-10	1.618e-15
2	5.670e-07	5.490e-05	1.753e-09
3	4.856e-07	5.024e-05	1.666e-10
4	9.419e-05	2.674e-09	1.753e-09
5	4.828e-07	5.053e-05	2.382e-09
6	1.561e-02	4.307e-14	2.068e-02
7	8.488e-03	7.301e-19	9.667e-07
8	1.175e-02	2.564e-09	7.088e-04
9	2.639e-02	6.741e-09	2.198e-02
10	1.489e-02	5.787e-10	1.010e-02
11	2.796e-02	5.314e-09	1.186e-02
<b>Overall</b>	<b>9.564e-03</b>	<b>1.864e-03</b>	<b>5.939e-03</b>

Table 4-5: Loss Function of different methods over 10,000 Epochs

#### 4.6 Fault Classification and loss function results for 20,000 iterations

Now, let us consider the fault classification problem of our studied Simulink model. The fault classification problem is solved by considering a data sample of 20,000 for the different fault types aforementioned. In a similar fashion, the result analysis for 20,000 data samples, the classification accuracy and loss results are presented below.

Figure 4-17 shows the expected behavior as we can see that the model rapidly learns the problem. At about 15 epochs or even less, the model leaped up to more than 90% accuracy. We could have stopped training at epoch 50 instead of epoch 20,000 due to learning speed or to avoid over-fitting. However, the model converged at a larger batch size and was stable during learning.

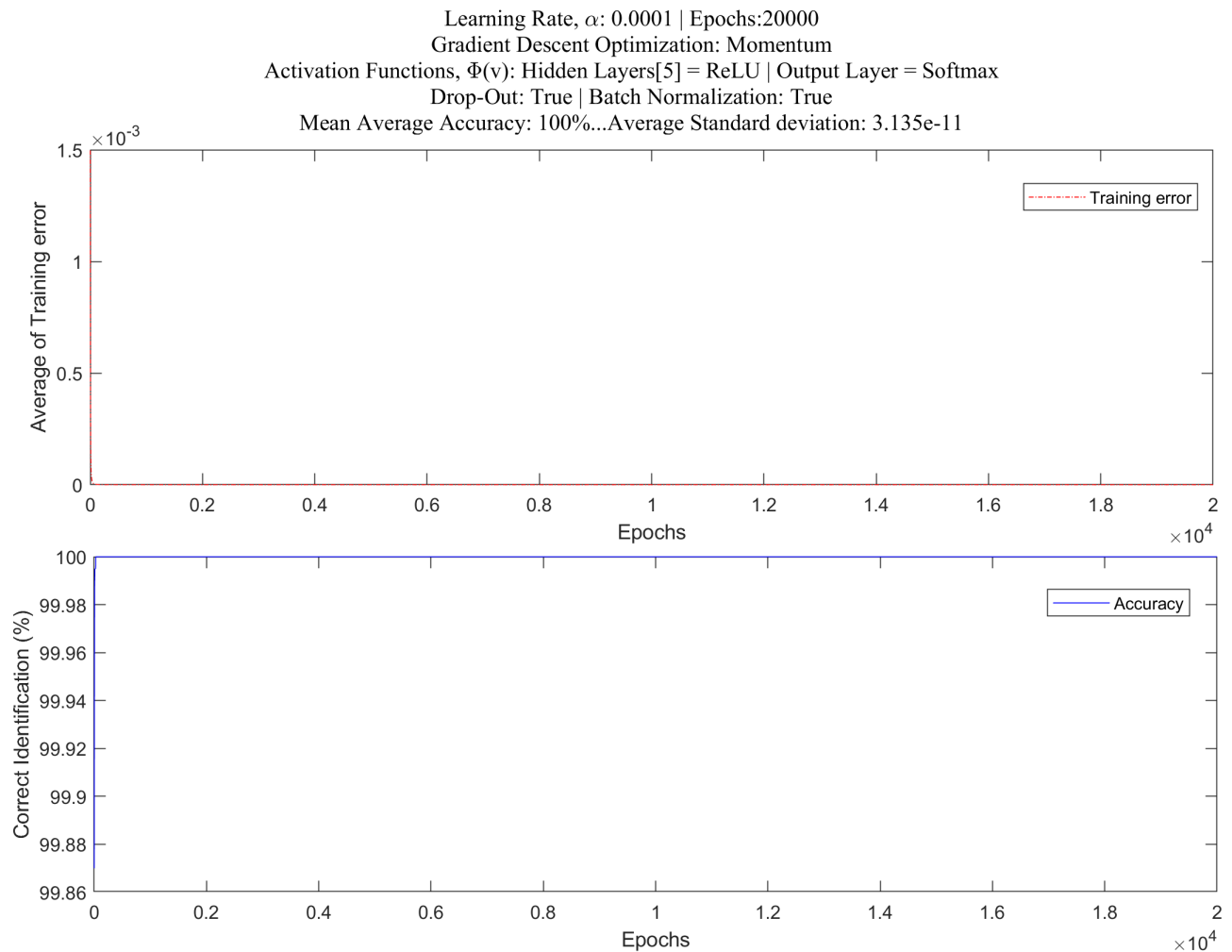


Figure 4-17: Training Learning Curves of a single switch over 20,000 epochs

Figure 4-18 exhibited an interesting learning curve which is slightly different from other category in Fault type 3. Before 10,000 epochs, it was impossible to stop training because it has a noisy weight updates. At about 8,000 epochs, this plot depicts clearly that the model is relatively slow to learn this problem, converging at



a solution after 10,000 epochs, but also stabilizes more towards the end of the run.

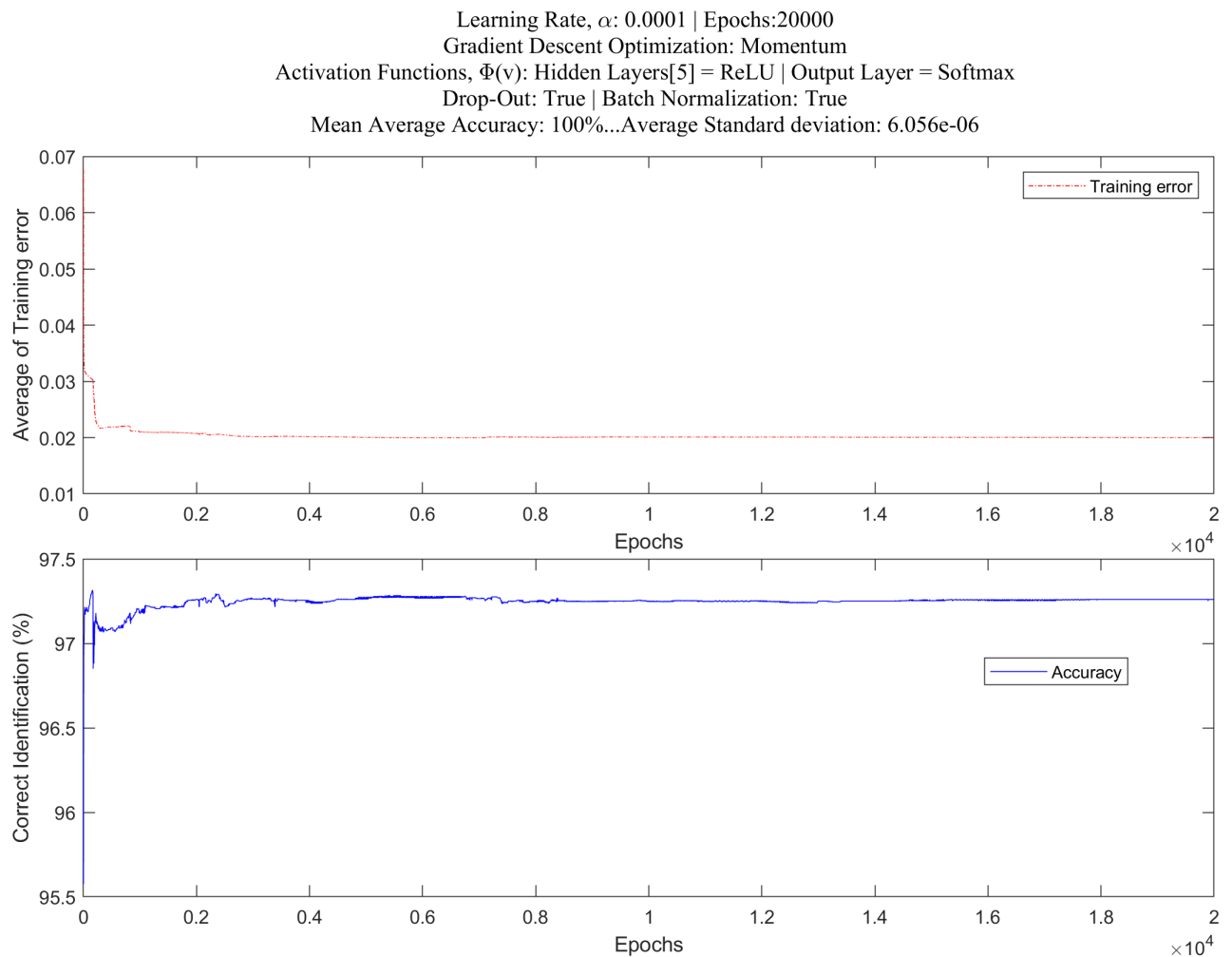


Figure 4-18: Training Learning Curves of double switch faults over 20,000 epochs

Figure 4-19 depicts another interesting learning curve. We can see that the model is relatively slow to learn this problem, thus difficult finding a point of convergence. This particular test suite deviates from other categories in Fault type 4. However, we can say that the noisy updates has a significant effect on its noisy performance throughout the duration of training. Nevertheless, the model almost converged and reached learning stability.

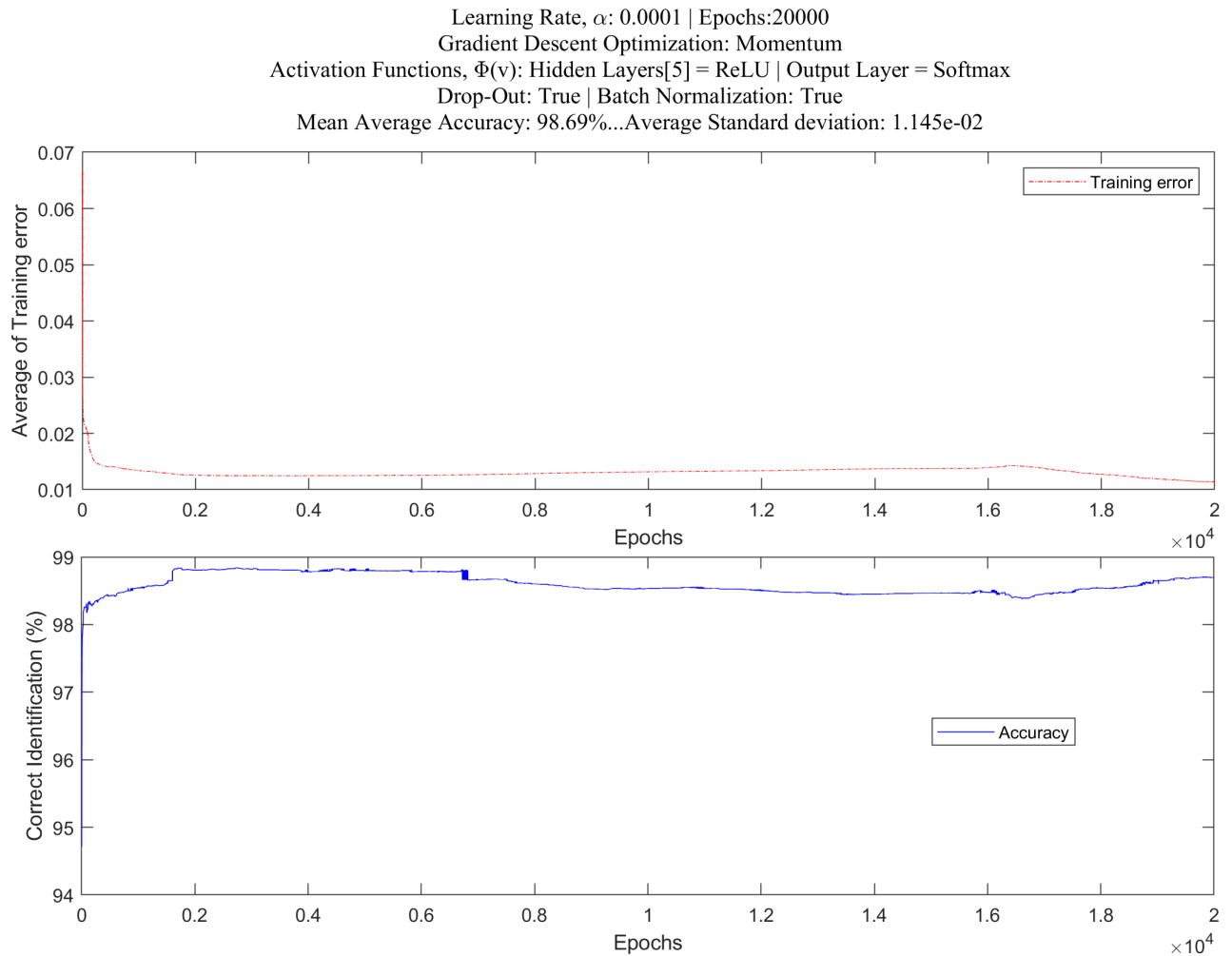


Figure 4-19: Training Learning Curves of triple switch faults over 20,000 epochs

Figure 4-20 leaped up to about 90% accuracy in about 60 epochs. From the learning curve, the model learns rapidly and converged quickly to a solution at about 1000 epochs but with relatively noisy weight updates.

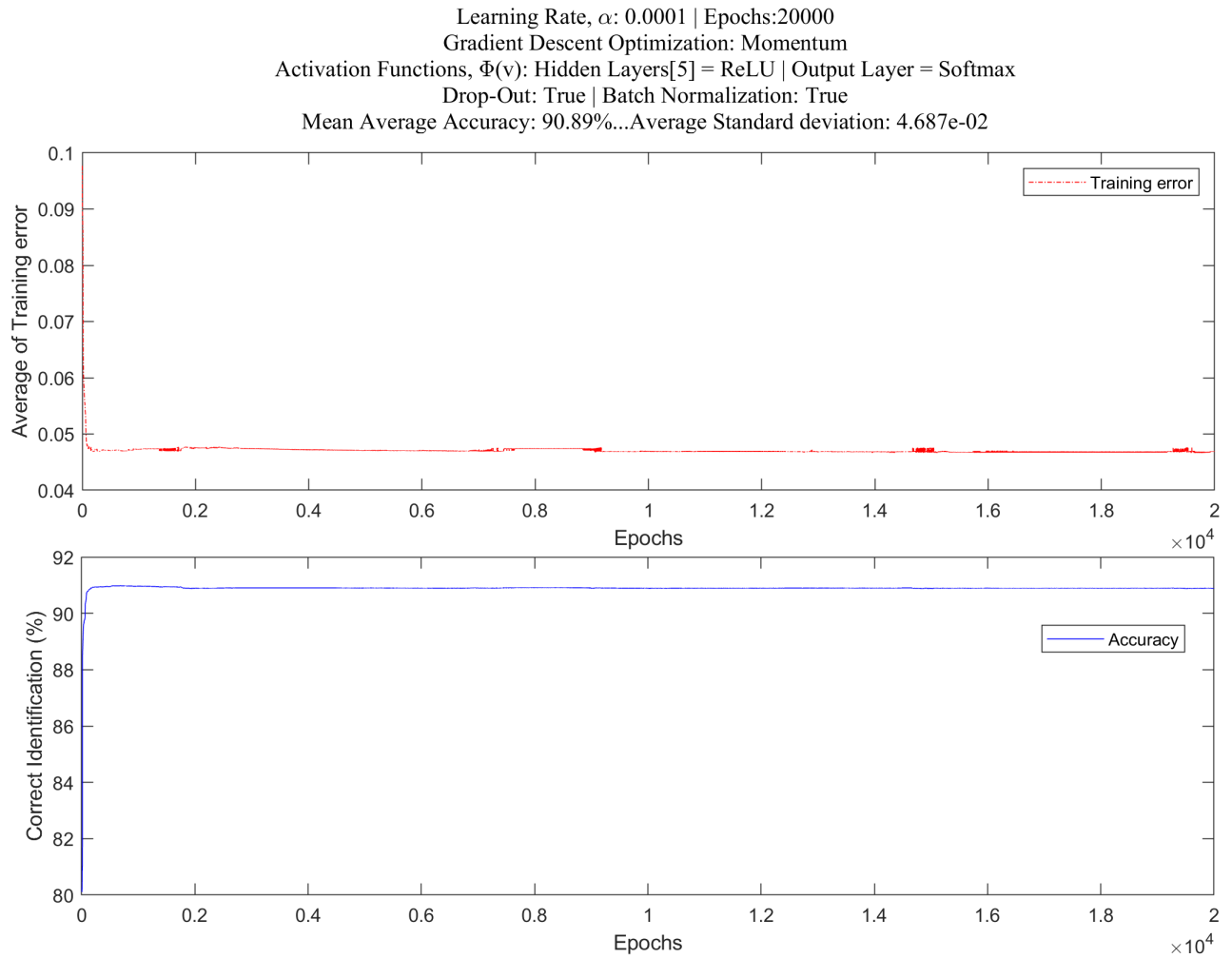


Figure 4-20: Training Learning Curves of multiple switch over 20,000 epochs

Table 4-6 and Table 4-7 summarizes the results of training error and classification accuracy over 20,000 training epochs.

<b>Average Training Error</b>			
Fault ID	DPLT	Our Method	
		Method 1	Method 2
1	7.402e-17	1.405e-12	8.638e-17
2	4.691e-07	4.994e-05	6.904e-12
3	3.734e-07	2.820e-07	5.456e-12
4	1.343e-05	4.649e-05	3.135e-11
5	5.272e-07	4.674e-05	4.299e-11
6	1.6021e-02	2.836e-05	1.998e-02
7	1.102e-03	4.863e-05	4.389e-07
8	6.828e-03	6.641e-05	1.016e-03
9	6.949e-03	8.573e-05	8.527e-04
10	1.277e-02	4.442e-05	1.145e-02
11	2.085e-02	8.418e-05	4.687e-02
<b>Overall</b>	<b>5.867e-03</b>	<b>4.556e-05</b>	<b>7.288e-03</b>

Table 4-6: Loss Function of three different methods

<b>Fault ID</b>	<b>DPLT</b>	<b>Ours</b>
1	100	100
2	100	100
3	100	100
4	100	100
5	100	100
6	98.20	97.26
7	99.9	100
8	99.70	99.91
9	99.40	99.91
10	98.70	98.69
11	98.20	90.89
<b>Overall</b>	<b>99.46</b>	<b>98.78</b>

Table 4-7: Fault Classification rates(%) for 20,000 Simulation runs

# Conclusion

In this thesis, we applied shallow neural networks to the problem of fault detection and classification for different training epochs and a built-in function “nprtool” in MATLAB was used to perform offline training of the neural network for the purpose of comparing methods. The proposed networks perform well with the selected training data set. The proposed network shows an 100% accuracy for single switching faults across 1000, 10,000 and 20,000 training iterations. This implies that the switching and conduction periods of the switch (single in this case) has zero power loss for significant training time. Conversely, the proposed network shows an average classification accuracy of about 99% for double switching faults but an accuracy of between 96.38% and 99.91% is observed in the case of triple switching faults. Even though the average accuracy for multiple switching faults was about 94.21%, relatively lower compared to other categories of fault considered, the general classification performance of the network is still high. Obviously, the classification performance between normal and abnormal conditions is quite satisfactory. Although, the results presented in this thesis looks promising, several points need to be addressed in future works. First, the disadvantages of prolonged processing time in neural networks, especially for 10,000 and 20,000 training epochs did not give too much room for repeated training of error and perhaps, classification accuracy to give better results. Otherwise, our method would have improved upon the classification accuracies of the DPLT method for both 10,000 and 20,000 training iterations. Second, the same technique can be used to analyze the faults in a three-phase inverter system with a pilot view of exploring different types of neural network (e.g. Convolutional Neural Network) to test and see if they perform better. Lastly, the networks developed in this thesis which have been

tested in simulation should be applied to an experimental system.

# Appendix

## Appendix A

Matlab Code for fault Data Capture

```
clear
close all

% parameters
%FT = 0.03 * rand() + 0.05;
FT = 0.03 * rand();
s1_set = 2; % 2=no fault, 1=fault short, 0=fault open
s2_set = 2;
s3_set = 2;
s4_set = 2;
my_max = 1000;

% Capture multiple data sets under the same fault condition with random
% fault time and random measurement error. Write to data file after each
% iteration.

fileID = fopen('NN_data_test_s1_1000.txt','w');

for x=1:my_max
```



```
% start simulation
%FT = 0.03 * rand() + 0.05; % new random fault time
FT = 0.03 * rand();
%my_pick = randi([1 4]);
my_pick= 1;
s1_set = 2; % initialize
s2_set = 2; % initialize
s3_set = 2; % initialize
s4_set = 2; % initialize
if (my_pick == 1)
    s1_set = randi([0 2]); % 2=no fault, 1=fault short, 0=fault open
end
if (my_pick == 2)
    s2_set = randi([0 1]); % 2=no fault, 1=fault short, 0=fault open
end
if (my_pick == 3)
    s3_set = randi([0 1]); % 2=no fault, 1=fault short, 0=fault open
end
if (my_pick == 4)
    s4_set = randi([0 1]); % 2=no fault, 1=fault short, 0=fault open
end

% To measure time, the tic/toc, cputime and clock commands
% tic--toc--> Stopwatch time
% c=clock---> Real time measurement
% t = cputime;e = cputime-t ---> CPU time
% use any of the time commands to measure the code performance
```

```

%tic
sim('NN_fault_detection_data_generator.slx')
current_data = ans.voltage_measured.Data;
[pxx,w] = pwelch(current_data, 100, [], 18);
%toc

plot(w/pi,10*log10(pxx))
xlabel('\omega / \pi')

%hold on
fprintf(fileID, '%6.2f%6.2f%6.2f%6.2f%6.2f%6.2f%6.2f%6.2f%6.2f', 10*log10(pxx))
fprintf(fileID, '%d %d %d %d %6.2f\r\n',s1_set,s2_set,s3_set,s4_set,FT);
end
hold off
fclose(fileID);

```

## Appendix B

### Training Algorithm of the Neural Network for Method 1

```

% NN implementation via backpropagation using
% one hidden layer (sigmoid) and one output layer (linear)
% For use with PSD data from 1-phase Simulink inverter with MOSFET faults
% Written by, V. Winstead
% Dec 1, 2019
%
% hidden layer fn --> logsig(n) = 1 / (1 + exp(-n))
% output layer fn --> purelin(n) = n

```

```

close all

clear

% configuration
n1 = 5; % number of neurons in layer 1 (hidden layer)
i_size = 10; % length of input vector
o_size = 4; % length of output vector
my_alpha = 0.005; % learning rate

s = rng;

% initial conditions on weights (W) and bias offsets (b)
%% adjusted starting weight and bias
W1 = randn(n1, i_size);
b1 = randn(n1, 1);
W2 = randn(o_size, n1);
b2= randn(o_size, 1);

% open data file
fileID = fopen('NN_data_test_s1_s2_s3_s4_1000.txt', 'r');
formatSpec = '%f %f %f %f %f %f %f %f %f %f %d %d %d %d %f';
A = fscanf(fileID,formatSpec, [15 Inf]);
fclose(fileID);

A = A'; % data read in transposed
p = A(:,1:10); % PSD data only
fault_device = A(:,11:14); % fault condition for each MOSFET
my_index = length(p); % number of iterations to train NN

```

```
for z = 1:my_index

    % compute output of hidden layer
    my_temp = W1 * p(z,:) + b1;
    a1 = 1 ./ (1 + exp(-my_temp));

    % compute output of output layer
    my_temp = W2 * a1 + b2;
    a2 = my_temp;

    % compute error
    my_e = fault_device(z,:) - a2;

    % compute partial differentials
    F1 = diag((1-a1).*a1);
    F2 = diag(ones(o_size, 1));

    % compute backpropagation starting with 2nd layer
    s2 = -2 * F2 * my_e;
    s1 = F1 * W2' * s2;

    % update weights and bias values
    W2 = W2 - my_alpha * s2 * a1';
    b2 = b2 - my_alpha * s2;
    W1 = W1 - my_alpha * s1 * p(z)';
    b1 = b1 - my_alpha * s1;
```

```
my_e_out(z) = norm(my_e); % store metric of errors

% Average Training error
E1(z) = my_e_out(z)/ my_index;
end

plot(E1)
xlabel('Epoch')
ylabel('Average of Training error')
legend('Average Training error')
title('model loss')
```

## Appendix C

### Training and Testing Algorithm of the Neural Network for Method 2

```
%% (DEEP)NN implementation
% For use with PSD data from 1-phase Simulink inverter with MOSFET faults
%
% hidden layer fn --> 1
% output layer fn --> 1
%% Author:Orukotan, Ayomikun Samuel <ayomikun.orukotan@mnsu.edu>
% Feb 7, 2020
%%
close all;
clear *;
```

```

clc;

% open data file
fileID = fopen('NN_data_test_s4_1000.txt','r');
formatSpec = '%f %f %f %f %f %f %f %f %f %f %d %d %d %d %f';
A = fscanf(fileID,formatSpec, [15 Inf]);
fclose(fileID);
A = A'; % data read in transposed
X = A(:,1:10); % PSD data only
fault_device = A(:,11:14); % fault condition for each MOSFET

% Choose
%DI = fault_device(:,1);
%DI = fault_device(:,2);
%DI = fault_device(:,3);
DI = fault_device(:,4);

% NN Config
[N1,N2] = size(X);
Onod = 3; % 3 output nodes representing the 3-classes: 0 1 and 2
Inod = N2; % nodes of each training data
N = N1; % length of training data
H = 1; % hidden layers (odd)
LS = H + 1; %layer space

% output encoding with One-hot encoding
D = zeros(N,Onod);

```

```

for id = 1:N
    if DI(id) == 0
        D(id,:) = [1 0 0];
    elseif DI(id) == 1
        D(id,:) = [0 1 0];
    elseif DI(id) == 2
        D(id,:) = [0 0 1];
    end
end

% node/neuron list in each layer
% 5 hidden nodes in H hidden layer(s)
nodelist = [Inod, 5, Onod]; % in-hid-...-hid-out;

s = rng;

W = cell(LS,1); % weights
dW = cell(LS,1); % prev weights
M = cell(LS,1); % momentum of weights
% initialize weights and its momentums
for id = 1:LS
    rb = sqrt(2/(nodelist(id+1)+nodelist(id))); % 6 for normal, 2 for uniform
    %W{id} = 1*rb*rand(nodelist(id+1), nodelist(id)) - 0; %normal [0 rb]
    W{id} = 2*rb*rand(nodelist(id+1), nodelist(id)) - rb; % uniform, [-rb rb]
    M{id} = zeros(nodelist(id+1), nodelist(id));
    dW{id} = zeros(nodelist(id+1), nodelist(id));
end

```

```

% train
horizon = 1000; % number of times to train NN
E1 = zeros(horizon, 1); % error metric
Cid = zeros(horizon, 1); % error metric
PCid = zeros(horizon, 1); % error metric

for epoch = 1:horizon
    [W,dW,M] = DeepPSDNN(W, dW, M, X, D, LS);

    % this epoch's training error
    e = 0;
    es1 = zeros(Onod,1);

    % Verify. inference
    Y = zeros(N,Onod);
    for k = 1:N
        x = X(k,:)';
        d = D(k,:)';

        u = x;
        y = cell(LS,1);

        % FORWARD
        for id=1:LS
            v = weightSum(W{id},u);

```



```

        if id == LS
            y{id} = Softmax(v);% out Sigmoid(v); Softmax(v)
        else
            y{id} = ReLU(v); % Sigmoid(v); ReLU(v); % hid
            u = y{id};
        end
    end
end

Y(k,:) = y{LS,1};

% sum of squared errors
es1 = es1 + (d - y{LS,1}).^2;

e = norm(es1); % norm of classification errors
end

% save learning process
E1(epoch) = e / N;

% percentage of correct fault identification
for k = 1:N
    for id = 1:3
        if abs( Y(k,id)) < 0.5 && abs(Y(k,id)) >= 0
            Y(k,id) = 0;
        elseif abs(Y(k,id)) <= 1 && abs(Y(k,id)) >= 0.5
            Y(k,id) = 1;
        end
    end
end

```

```
        end
        if norm(D(k,:)-Y(k,:)) == 0
            Cid(epoch) = Cid(epoch) + 1;
        end
    end
    PCid(epoch) = (Cid(epoch) / N ) * 100;

end

% plot training errors
subplot(211)
plot(E1, 'r-.' )
xlabel('Epoch')
ylabel('Average of Training error')
title('Model Loss')
legend('Training error')

% plot correct id
subplot(212)
plot(PCid, 'b')
xlabel('Epoch')
ylabel('Correct Identification (%)')
title('Classification Accuracy')
legend('Accuracy')
```

## References

- [1] V. Preetham, “Backpropagation — How neural networks learn complex behaviors,” *Medium*, 2016.
- [2] P. Kim, “Matlab deep learning: With machine learning,” *Neural Networks and Artificial Intelligence*. Apress, 2017.
- [3] A. More, “Power conversion market 2020 - global industry research update, future scope, size estimation, revenue, pricing trends, growth opportunity, regional outlook and forecast to 2025,” *Allied Market Research*, 2020.
- [4] Reportlinker, “The DC-DC converters market,” 2020.
- [5] A. Ristow, M. Begovic, A. Pregelj, and A. Rohatgi, “Development of a methodology for improving photovoltaic inverter reliability,” *IEEE Transactions on Industrial Electronics*, vol. 55, no. 7, pp. 2581–2592, 2008. tex.publisher: IEEE.
- [6] J. Korbicz, J. M. Koscielny, Z. Kowalczyk, and W. Cholewa, *Fault diagnosis: models, artificial intelligence, applications*. Springer Science & Business Media, 2012.
- [7] I. Castillo and T. Edgar, “Texas - wisconsin - california control consortium - Model-based fault detection and diagnosis,” 2008. Place: Austin, Texas tex.label: ICa08.
- [8] E. Wolfgang, K. Kriegel, and W. Wondrak, “Reliability of power electronic systems,” in *2009 13th european conference on power electronics and applications*, pp. 1–38, 2009. tex.organization: IEEE.

- [9] S. Yang, A. Bryant, P. Mawby, D. Xiang, L. Ran, and P. Tavner, “An industry-based survey of reliability in power electronic converters,” *IEEE transactions on Industry Applications*, vol. 47, no. 3, pp. 1441–1451, 2011. tex.publisher: IEEE.
- [10] B. Liu, *Automated Debugging and Fault Localization of Matlab/Simulink Models*. PhD thesis, University of Luxembourg, Luxembourg, 2017.
- [11] J. Mahanta, “Introduction to neural networks, advantages and applications,” *Medium*, 2017.
- [12] G. D. Smith, N. C. Steele, and R. F. Albrecht, *Artificial neural nets and genetic algorithms: Proceedings of the international conference in norwich, UK, 1997*. Springer Science & Business Media, 2012.
- [13] D. Hush, C. Abdallah, G. Heileman, and D. Docampo, “Neural networks in fault detection: a case study,” in *Proceedings of the 1997 american control conference (cat. No. 97CH36041)*, vol. 2, pp. 918–921, 1997. tex.organization: IEEE.
- [14] D. Diep, A. Johannet, P. Bonnefoy, and F. Harroy, “Obstacle identification by an ultrasound sensor using neural networks,” in *Artificial neural nets and genetic algorithms*, pp. 1–5, 1998. tex.organization: Springer.
- [15] K. S. Gyamfi, J. Brusey, A. Hunt, and E. Gaura, “Linear dimensionality reduction for classification via a sequential Bayes error minimisation with an application to flow meter diagnostics,” *Expert Systems with Applications*, vol. 91, pp. 252–262, 2018. tex.publisher: Elsevier.
- [16] S. Gyamfi, “UCI machine learning repository: Ultrasonic flowmeter diagnostics data set,” 2017. tex.institution: University of California, Irvine, School of Information and Computer Sciences.

- [17] P. Tüfekci, “Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods,” *International Journal of Electrical Power & Energy Systems*, vol. 60, pp. 126–140, 2014. tex.publisher: Elsevier.
- [18] R. Lopez, “Predict power generation | example | neural designer,”
- [19] F. N. Chowdhury and J. L. Aravena, “A modular methodology for fast fault detection and classification in power systems,” *IEEE transactions on control systems technology*, vol. 6, no. 5, pp. 623–634, 1998. tex.publisher: IEEE.
- [20] A. I. Megahed, A. M. Moussa, and A. Bayoumy, “Usage of wavelet transform in the protection of series-compensated transmission lines,” *IEEE Transactions on Power Delivery*, vol. 21, no. 3, pp. 1213–1221, 2006. tex.publisher: IEEE.
- [21] P. Bhowmik, P. Purkait, and K. Bhattacharya, “A novel wavelet transform aided neural network based transmission line fault analysis method,” *International Journal of Electrical Power & Energy Systems*, vol. 31, no. 5, pp. 213–219, 2009. tex.publisher: Elsevier.
- [22] A. Abbaspour, K. K. Yen, S. Noei, and A. Sargolzaei, “Detection of fault data injection attack on uav using adaptive neural network,” *Procedia computer science*, vol. 95, pp. 193–200, 2016. tex.publisher: Elsevier.
- [23] V. Preetham, “Mathematical foundation for activation functions in artificial neural networks,” *Medium*, 2017.
- [24] A. MOAWAD, “Dense layers explained in a simple way,” *Medium*, 2019.
- [25] N. Kumar, “Deep learning best practices: Activation functions & weight initialization methods — Part 1,” *Medium*, 2019.

- [26] v. luhanawal, “Analyzing different types of activation functions in neural networks — which one to prefer?,” *Medium*, 2019.
- [27] Wikipedia, “Sign function,” *Wikipedia*, 2020.
- [28] Wikipedia, “Softmax function,” *Wikipedia*, 2019.
- [29] J. Heaton, “Introduction to neural networks for c#, heaton research,” *Inc.: St. Louis, MO, USA*, 2008.
- [30] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.
- [31] A. Lenail, “Publication-ready nn-architecture schematics.,” *NN SVG*, 2020.
- [32] Mathworks, “Matlab documentation/ supervised learning workflow and algorithms,” 2001. tex.label: Mat01 tex.location: United States tex.publisher: Mathworks.
- [33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [34] I. Guyon, “Applications of neural networks to character recognition,” *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 5, no. 01n02, pp. 353–382, 1991. tex.publisher: World Scientific.
- [35] S. Knerr, L. Personnaz, and G. Dreyfus, “Handwritten digit recognition by neural networks with single-layer training,” *IEEE Transactions on neural networks*, vol. 3, no. 6, pp. 962–968, 1992. tex.publisher: IEEE.
- [36] G. L. Martin and J. A. Pittman, “Recognizing hand-printed letters and digits using backpropagation learning,” *Neural computation*, vol. 3, no. 2, pp. 258–267, 1991. tex.publisher: MIT Press.

- [37] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, “Handwritten digit recognition with a back-propagation network,” in *Advances in neural information processing systems*, pp. 396–404, 1990.
- [38] W. G. Baxt, “Use of an artificial neural network for data analysis in clinical decision-making: the diagnosis of acute coronary occlusion,” *Neural computation*, vol. 2, no. 4, pp. 480–489, 1990. tex.publisher: MIT Press.
- [39] H. Bölcskei, P. Grohs, G. Kutyniok, and P. Petersen, “Optimal approximation with sparsely connected deep neural networks,” *SIAM Journal on Mathematics of Data Science*, vol. 1, no. 1, pp. 8–45, 2019. tex.publisher: SIAM.
- [40] G. P. Zhang, “Neural networks for classification: a survey,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 30, no. 4, pp. 451–462, 2000. tex.publisher: IEEE.
- [41] Géron, Aurélien, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O’Reilly Media, 2019.
- [42] S. Sivanandam and S. Deepa, *Introduction to neural networks using Matlab 6.0*. Tata McGraw-Hill Education, 2006.
- [43] B. Raj, “Index of / bhiksha/courses/deeplearning/Spring.2018/www/slides,” tech. rep., Carnegie Mellon School of Computer Science, 2018.
- [44] J. Brownlee, “How to one hot encode sequence data in python,” *Machine Learning Mastery*, 2017.
- [45] S. Ruder, “An overview of gradient descent optimization algorithms,” *Sebastian Ruder*, 2016.

- [46] A. Kathuria, “Introduction to optimization in deep learning: Momentum, rmsprop and adam,” *Paperspace Blog*, 2018.
- [47] J. McCaffrey, “Neural network momentum using python -,” *Visual Studio Magazine*, 2017.
- [48] Gluon, “Momentum,” 2020.
- [49] S. K. Kumar, “On weight initialization in deep neural networks,” *arXiv preprint arXiv:1704.08863*, 2017.
- [50] Y. Kim, “Convolutional neural networks for sentence classification,” *arXiv preprint arXiv:1408.5882*, 2014.
- [51] A. Krizhevsky, G. Hinton, and others, “Learning multiple layers of features from tiny images,” 2009. tex.publisher: Citeseer.
- [52] E. L. Lehmann, *Elements of large-sample theory*. Springer Science & Business Media, 2004.
- [53] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proc. icml*, vol. 30, p. 3, 2013. tex.number: 1.
- [54] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.
- [55] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.
- [56] M. M. ARAT, “Weight initialization schemes - xavier (glorot) and he,” *Mustafa Murat ARAT*, 2019.



- [57] S. Heo and J. H. Lee, "Fault detection and classification using artificial neural networks," *IFAC-PapersOnLine*, vol. 51, no. 18, pp. 470–475, 2018. tex.publisher: Elsevier.
- [58] D. Himmelblau, "Use of artificial neural networks to monitor faults and for troubleshooting in the process industries," *IFAC Proceedings Volumes*, vol. 25, no. 4, pp. 201–206, 1992. tex.publisher: Elsevier.
- [59] Wikipedia, "Radial basis function network 2019," *Wikipedia*, 2019.
- [60] MarketWatch, "Power converters and inverters market analysis, growth by top companies, trends by types and application, forecast analysis to 2025," *MarketWatch*.
- [61] H. Kaya, P. Tüfekci, and F. S. Gürgen, "Local and global learning methods for predicting power of a combined gas & steam turbine," in *Proceedings of the international conference on emerging trends in computer and electronics engineering icetcee*, pp. 13–18, 2012.
- [62] P. Bhowmik, P. Purkait, and K. Bhattacharya, "A novel wavelet transform and neural network based transmission line fault analysis method," 2008. tex.publisher: IET.
- [63] S. Sumathi and S. Paneerselvam, *Computational intelligence paradigms: theory & applications using MATLAB*. CRC Press, 2010.
- [64] K. Bennett, *MATLAB: Applications for the practical engineer*. BoD–Books on Demand, 2014.
- [65] P. Researcher, Matthew Stewart, "Comprehensive introduction to neural network architecture," *Medium*, 2020.

- [66] D. S. Raschka, “Deep learning and machine learning research,” *Dr. Sebastian Raschka*, 2020.
- [67] Wikipedia, “Radial basis function network,” *Wikipedia*, 2019.