OULU UNIVERSITY OF
APPLIED SCIENCES

Gavin Harper

Viability of Artificial Neural Networks in Mobile Health-care

Viability of Artificial Neural Networks in Mobile Healthcare

Gavin Harper
Bachelor's Thesis
Spring 2011
Degree programme in
Business Information Technology
Oulu University of Applied Sciences

Abstract

Oulu University of Applied Sciences
Degree in Business Information Technology

Author(s): Gavin Harper
Title of Bachelor's thesis: Viability of Artificial Neural Networks in Mobile Healthcare
Supervisor(s): Anu Niva
Term and year of completion: Spring 2011 Number of pages: 43 + 17 appendices

ABSTRACT:

This thesis describes the design process of an artificially intelligent algorithm capable of differentiating between two discrete electrocardiogram readings (a signal depicting the electrical activity of the heart at a given point in time). The paper culminates in a reduced form implementation of one such algorithm utilising the neural computing paradigm. Algorithms implemented under the neural computing paradigm mimic the as-currently perceived operation of the brain cells known as neurons. By creating a network of nodes (neurons) and altering the connections between them, an algorithm can display rudimentary forms of what may be classified as intelligence by displaying an ability to recognise patterns in data and adapt to better recall those patterns.

The theoretical section of this paper discusses first those concepts of electrocardiography that directly relate to the research in this paper. The remainder of the theoretical section is dedicated to neural computing. By reading this paper, a reader not previously familiar with the subject should understand the various factions of the artificial intelligence community and both the operation and purpose of neural computing and what role it may potentially play in future mobile healthcare developments.

Keywords:

Artificial Neural Networks, Electrocardiogram, ECG, Heart

Contents

List of Figures

1 Introduction

Mobile healthcare is a rapidly emerging subset of the healthcare industry focusing on providing medical assistance to those who may be unable, or lack the desire to submit to traditional healthcare methods carried out at one distinct geographical location such as a hospital. The premise driving the mobile healthcare industry centres around the notion that recent advances in technologies such as wireless communication and portable computation provide a powerful platform for automated procurement, analysis and transmission of patient data such as the activity of the heart at a given point in time. This paper focuses primarily on developing automated, mobile means of intelligently analysing the state of a patient's heart. It is hoped that should the technology powering mobile healthcare attain suitable accuracy and efficiency, such that wide scale deployment is possible, these technologies may serve to the betterment of those unable to submit to traditional medical treatment.

It is exciting to consider a device that may be worn by a patient in an unobtrusive, discrete manner capable of intelligently analysing the electrical activity of the heart and alerting at some early stage in the event of an anomaly or onset of *arrhythmia*. Such early warning under traditional methods requires the patient to be located in a hospital. However, if a device could conform to certain expectations such as extensive battery life, low mass, accurate analysis and high efficiency, it is not unreasonable to suggest that those at risk to a given heart condition can monitor their health over a long period of time while attending to their daily life.

Utilising additional technologies such as the Global Positioning System (GPS) and 3G/High Speed Downlink Packet Access (HSDPA) which may be trivially implemented in small devices as evidenced by the current generation of mobile phone technology, provides additional benefits. Accessing the internet by means of a cell phone tower allows a mobile electrocardiogram device to communicate with a centralised server and push updates and warnings in real-time so ensure that in the event of the onset of arrhythmia or another undesirable state. This functionality may be coupled with the Global Positioning System to push location updates to the server in the event that medical assistance is required.

This research in this paper is conducted on behalf of Domuset Oy, one such company that seeks to develop devices similar to those described above. This paper represents a small subset of the research conducted into developing intelligent signal analysis algorithms that may be deployed on low powered, portable devices. The research and problem set has been adapted to adhere to requirements that dictate both the spacial restrictions and the assumed knowledge of the reader. It is such that a reader with little to no background in mathematics, signal analysis or electrocardiography may still follow the progression of this paper as the writing style has been intentionally structured to introduce new concepts in a logical and progressive manner.

1.1 Research Methods

Printed material has been employed as the preferential method of information gathering for those sections relying on mathematical accuracy as there exists many highly respected texts with a proven track record in the field of adaptive learning algorithms. Journals have proven to be an invaluable source for obtaining a historical perspective on matters such as the early history of artificial intelligence. For such topics of research that printed material was not readily available in the time frame required, electronic sources such as internet based material were employed. It is important to note that the preference of printed material in no way represents the opinion that electronic material is sub par or any less valuable to the research process. Instead it should be noted that a familiarity with many of the printed sources pertaining to the field of artificial intelligence and adaptive learning algorithms was in existence prior to commencing this research project.

In addition to traditional methods of information gathering, regular meetings with the commissioner in attendance have proven to be instrumental in directing the research project. It is the case that many prior methods for analysing the datasets discussed in this paper have been discussed and over a period of time, optimal solutions became apparent. The informal nature of these meetings allowed a focus on the flow of ideas and as such served to simultaneously educate about the subject of electrocardiograms and the subsequent analysis methods and also to invigorate the need to continue researching the topics discussed previously. While this paper prefers to focus on a quantitative approach to data collection and analysis due to the highly nature of the research. It would, however, be factually inaccurate to state that a qualitative approach had no place during this research project. Simple encounters such as the meetings described were highly useful.

1.2 Research Objective

The objective of this research is to develop a proof-of-concept algorithm capable of differentiating between datasets consisting of distinct regions of electrocardiogram readings.

The resulting algorithm is to display characteristics that may be considered intelligent, such as pattern recognition and adaptive behaviour. In addition, it is important that the algorithm be designed in such a way that resources usage and complexity is minimised allowing deployment in low-powered devices such as mobile telephones and dedicated hardware.

The scope of this research extends to demonstrating the suitability and in some cases, the necessity of adaptive learning algorithms for analysing and interpreting nontrivial datasets such as those obtained from natural processes. This paper will present a subset of the research into classifying electrocardiogram readings with adaptive learning algorithms carried out for the larger. As such, the problem as it appears in this paper may be considered to be in reduced form for brevity.

All code samples and algorithms will be presented utilising the C programming language. C has been chosen due to the inherent portability of the language and the speed benefits that arise from employing a compiled language. In addition, the scripts required to build a working implementation of the algorithm will be provided in the form of a GNU Make Makefile.

Due to the nature of the objective, the proceeding sections will first discuss the required background knowledge in a logical, progressive and abstracted manner thus disambiguating the notion of adaptive learning software before formally introducing the objective in a parametrised manner prior to implementation and analysis.

2 Electrocardiograms

The Electrocardiogram (ECG) is a tool employed for the task of analysing the electrical activity of the heart. The electrocardiogram was invented by Dutch physiologist Willem Einthoven in the early 20th century. An early prototype of what eventually lead to the modern day electrocardiogram was based on a *string galvanometer*, a tool utilised for amplifying electric signals over subterranean cables such as those running undersea (MIT 2005, date of retrieval 03/05/2011). In 1901, a working model utilising a string galvanometer was produced that was capable of measuring the electrical activity of the heart. This working model eventually lead to a commercial model being introduced into the market in 1903. (MIT 2005, date of retrieval 03/05/2011). Einthoven was awarded a Nobel Prize for the conceptualisation and invention of the Electrocardiogram in 1924 (Cohen, date of retrieval 03/05/2011).

In modern times, typical Electrocardiogram readings are obtained by attaching 12 electrodes onto a patient at strategic points. Cohen, (date of retrieval 03/05/2011) states that 12 electrodes are attached as no single point is capable of providing a complete reference of the current state of the heart. However, for the purposes of mobile health care, it is excessive to attach 12 electrodes to a patient as this severely hampers mobility and is beyond the expected capabilities of patients to attach correctly all 12 nodes. As a result, in mobile solutions, far fewer electrodes are typically employed.

2.1 Cardiac Anatomy

The heart is the muscle responsible for circulating oxygen and nutrients contained within blood to the various cells in the body. The heart is comprised of four chambers, two *atria* (singular atrium) and two *ventricles*. Cohen (date of retrieval 03/05/2011) states that "[t]he heart is really 2 'half hearts,' the right heart and the left heart, which beat simultaneously". Each half contains exactly one atrium and one ventricle, located in the upper, smaller and lower, larger chambers respectively.

To induce circulation, the heart employs a beating mechanism. The *sinus node* located in the right atrium controls the beating of the heart by discharging at a given time, typically $60 - 100/minute$. During discharge, an electrical impulse traverses the heart to reach the *atrioventricular* node. Cohen (date of retrieval 03/05/2011) describes the atrioventricular node as "a relay point to further propagate the electrical impulse". Upon such time that the electrical impulse happens upon the ventricles, the ventricles contract and as a result of this contraction, blood is carried away from the heart via *arteries*. (Cohen, date of retrieval 03/05/2011)

2.2 Interpreting the Electrocardiogram

The electrocardiogram machine records visually the electrical activity of the heart as it surfaces on the *epidermis*. Readings are typically taken at intervals of $4ms$ or $250/sec$ are are visualised by means of the *electrocardiogram trace*. The electrocardiogram trace displays, at any given point in the represented time period, the total strength of the electrical activity of the heart for that point in time. (Ashley et al. 2004, date of retrieval 03/05/2011)

The Electrocardiogram readings contain five discrete points of interest, namely the *P, Q, R, S* and *T* waves. The first point of interest is the *P wave*. The P wave is the result of the sinus node *depolarising*. During this time the left and right atrium are in a state of contraction. Following the P wave is the *QRS Complex* (an amalgamation of the Q, R and S waves), representing the ventricular depolarisation (contraction). The final wave of interest is the *T wave*, signifying ventricular repolarisation. Atrial repolarisation is typically engulfed as part of the QRS complex and is not visible on the electrocardiogram strip under circumstances deemed normal. Figure 1 represents an idealised electrocardiogram reading of a single depolarisation and repolarisation of the heart, with each constituent wave emphasised. An additional point of interest is the *U wave*. This, however, serves no purpose in this paper so a discussion of this wave is bypassed. The characteristics of these five, PQRST points aid in diagnosing potential cardiac maladies. (Ashley et al. 2004, date of retrieval 03/05/2011)

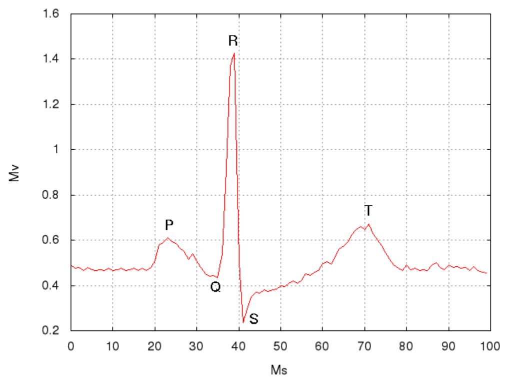

Figure 1: Idealised Representation of PQRST

3 Neural Networks

"I propose to consider the question, 'Can machines think?' " began Turing (1950) in his seminal paper, Computing Machinery and Intelligence. In this paper, Turing proposed a test called the "imitation game" (Turing 1950) though this has been since renamed the *Turing Test*, that could determine the relative intelligence of a machine by prompting the machine and analysing the responses. The Turing test is structured as follows: A researcher communicates with a single machine and a single human, isolated from view and communicating with written *natural language*. The researcher will interrogate both participants and attempt to determine which participant is human and which is machine. Assuming a machine could consistently dupe a researcher into believing the machine was a human responding, Turing argued that this would be a display of intelligence.

Turing's test for machine intelligence was challenged in 1980 by Berkeley professor John Searl. In his paper, Minds, brains and, programs (1980), Searl asserted that if a machine may pass the Turing test and is able to provide serviceable responses that may be interpreted as extruding intelligence, there is no guarantee that the same machine understands the responses it generates. To illustrate this, Searl proposed a further thought experiment named The Chinese Room. In this thought experiment, Searl proposes that, with no prior knowledge of written Chinese, nor spoken Chinese, he may be locked in a room and fed sheets of paper containing Chinese symbols and English instructions for manipulating those symbols. In the event that rules of sufficient quality and detail were provided, Searl could generate responses that may be indistinguishable from a native speaker of the Chinese language while not understanding the nature of what was communicated.

These conflicting thought experiments serve to fuel the debate as to what constitutes true intelligence. Two distinct camps, each with ardent proponents, have arisen in the artificial intelligence research community. The two camps are *strong* and *weak* artificial intelligence. Strong artificial intelligence, first coined by Searl (1980) when discussing the Chinese Room thought experiment, promotes the notion that any machine extruding strong artificial intelligence may "literally ha[ve] cognitive states and that the programs thereby explain human cognition". Proponents of strong artificial intelligence envision a machine that may be able to match or surpass human abilities in areas such as reasoning, planning and learn. Weak artificial intelligence, by contrast, does not aim to match or surpass human intelligence, but rather to implement characteristics that demonstrate intelligence in specific situations.

In addition to strong and weak artificial intelligence, two other factions exist within the artificial intelligence community, that is those that practise *symbolic* artificial intelligence and those that practise *connectionist* artificial intelligence. The former relates to attaining perceived intelligence by, as described by Tveter (1998, 12), "emphasiz[ing] list processing and recursion". Symbolic artificial intelligence is often referred to as top-down artificial intelligence. An issue, as stated by Mitchell (1999, 4), with Symbolic or top-down artificial intelligence is that "[e]arly on, AI practitioners believed that it would be straightforward to encode the rules that would confer intelligence on a program". Mitchell (1999) continues that "[n]owadays, many AI researchers believe that the "rules" underlying intelligence are too complex to encode by hand in a "top-down" fashion". This notion allowed the rise of connectionist, or bottom-up, artificial intelligence. In the bottom-up paradigm, "humans write only very simple rules, and complex behaviors such as intelligence emerge from the massively parallel application and interaction of these simple rules".

Artificial neural networks may be classified under the connectionist, bottom-up paradigm and in this paper, will be portrayed as weak artificial intelligence. For the purposes of the research contained within paper, it is sufficient that the intelligence quota of the neural network implementation need not achieve sentience but rather achieve adeptness in pattern recognition and function approximation.

3.1 Biological Neural Networks

The following description aims to provide the reader with an intuition of the underlying neural processes that will serve as a model for the adaptive artificial neural network systems described and implemented in the remainder of this paper. It is important to note that it is unfeasible to model any of the biological processes discussed in this paper in their entirety due to complexity and problem scope. Instead, desirable behaviourisms will serve as a model from which to design an intelligent, adaptive system as opposed to replicating directly the biological model.

The human brain consists largely of *neurons.* Neurons are described by Masters (1992, 2) as "a single cell capable of a sort of crude computation". A neuron may receive input from other neurons and in turn may produce an output that is sent to other neurons. In a biological neural network, input refers to an electrical or chemical signal that, depending on the input connection, may serve to excite or inhibit a neuron. A Neuron consists of many component parts. Input is received over protoplasmic projections that branch, similar to a tree, from the *soma* (cell body) named *dendrites* and is directed toward the *axon terminal* from which the signal is transmitted via the *synapses.*

"[B]iological neural networks are analog" writes Heaton (2008, 41), "[b]ecause this signal is analog, the voltage of each signal will vary. If the voltage is within a certain range, the neuron will fire". When a neuron fires, it transmits a signal to other neurons. The decisions made by a neural network are based on whether a given neuron has fired or remained inert.

3.2 Artificial Neural Networks

Artificial neural networks, hereinafter referred to as neural networks, were first pioneered by Frank Rosenblatt in 1957 while working at the Cornell University Aeronautical Laboratory. Rosenblatt proposed a single layer neural network dubbed the *perceptron.* "[The perceptron] was computationally feasible on the hardware of that time, was based on biological models, and was capable of learning", writes Masters (1993, 4). The purpose of the perceptron was to illustrate the potential for cognitive abilities to be attained by means of nonlinear units massively interconnected (Schalkoff 1997, 27). The perceptron had an intriguing quality. Rosenblatt presented an algorithm demonstrating what was referred to as the "perceptron convergence theorem" (Masters, 1993, 4). The perceptron convergence theorm demonstrated a training mechanism for the perceptron whereby upon satisfying the condition that a given set of training weights were learnable, the algorithm would converge to a series of weights that provide correct response to input data. The perceptron suffered one fundamental flaw. In order for the perceptron to learn a pattern, it had to be *linearly separable.* (Masters 1993, 4)

After an upsurge in neural network research, additional network architectures and topologies were borne. Important architectures include the Hopfield network, the multilayer feedforward neural network and the self-organising map (SOM). One common trait among all neural network architectures is the need to be trained before they may serve a purpose. Training involves exposing a neural network to data that falls within a similar class as the live, production data it is expected to operate upon. This serves as a distinction between typical algorithms and computers. As Heaton (2008, 83) writes, "ordinary programs can only automate repetitive tasks, such as balancing checkbooks or calculating the value of an interest portfolio. While a program can easily maintain a large collection of images, it cannot tell us what is illustrated in any of those images. Programs are inherently unintelligent and uncreative".

Neural networks, however, do not provide a 'silver bullet' for every problem that may be represented in software or hardware. While it is true that neural networks excel in many areas, such as classifying data or predicting trends. Heaton (2008, 43) states that the qualities of neural networks are most prominent when a given problem cannot be expressed as a series of steps. Neural networks are most beneficial in situations requiring the processing of unpredictable non linear data, the patterns are subtle and not trivially distinguishable or the data may be non-absolute (such as consumer opinions) (Masters 1993, 7). In many situations, however, a neural network may serve to the detriment of performance and accuracy. Such cases may include problems that tend to feature constant data sets such as an unchanging business rule (Heaton 2008, 43) or the researcher requires knowledge not just of the result but how the result was derived from the input data. This last point is fundamental to the operation of neural networks. Neural networks are trained to learn and identify a given input. A neural network cannot, however, provide an intuition to the researcher what steps it took to produce a given output (Heaton 2008, 43). This paper is concerned with classifying digital signals. It is not imperative that the reasoning be deduced providing the classification is accurate.

3.3 Neural Network Infrastructure

There exists many proven neural network topologies that differ from their counterpart topologies in areas such as size, intention and topology. Common to all neural networks are certain core components that define the neural computing modus operandi. This section identifies and discusses these components, demonstrating the operation of neural networks. Initially, the Perceptron is discussed as a foundation and gradually a generic neural network topology is built as each required component is discussed and implemented in place.

3.3.1 Generic Topology

As a result of the inherently connected nature of biological neurons, it becomes trivial to represent these networks as a graph. Graphs are a powerful tool utilised in the field of discrete mathematics for visually representing and modeling the relationships between objects. A simple graph consists of *vertices* (singular *vertex*) and *edges.* A vertex represents a node in the network. An edge represents a relationship between two vertices. For the purposes of this paper, neurons are represented by vertices and the relationships (connections) between nodes are represented as edges.

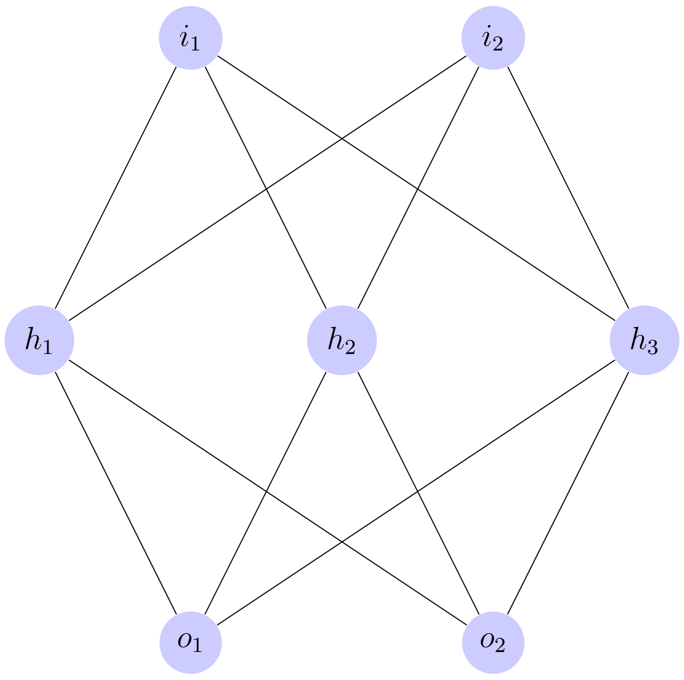

Figure 2: Generic Neural Network Topology Graph

Figure 2 shows a generic neural network structure consisting of three discrete layers. An input layer, a hidden layer and an output layer. The neurons that comprise the input layer (denoted i_n) perform no processing but instead serves to introduce input data to the network. Masters (1993, 8) describes the input layer as "a semantic construct used to represent the input sample". The hidden layer acts on the data presented by the input layer, allowing remapping of the input data. The term 'hidden layer' is somewhat of a misnomer as these neurons exist and are of great importance in many neural network topologies. The term 'hidden', refers to a 'black box' situation in which the operation of the hidden layer may be abstracted from view. The output layer is the point at which the overall mapping of the network input is available (Schalkoff 1997, 158). Heaton (2008, 146) advocates that in situations where the neural network is utilised for classification (sorting inputs into discrete groups based on inherent characteristics or patterns), one output neuron should be included for each potential classification group. However, in such situations where a neural network is employed to perform tasks such as noise reduction, it is beneficial to correlate the number of output neurons with the number of input neurons, that is $o_n = i_n$.

When transferring a neural network topology to the *computational domain*, one simple method for representing a graph is by means of a *matrix* (plural *matrices*). Matrices provide a fundamental tool in *linear algebra* whereby numerical values may be contained in a rectangular array. The values contained within a matrix are referred to as *elements* and may be referenced by their horizontal and vertical position within the matrix. It is possible to perform many mathematical operations on matrices, many of these will be discussed in subsequent sections of this paper as it becomes pragmatic to do so. A matrix of *order* (dimensions) 7×7 is drawn below. Additional notations have been included that demonstrate referential points. Figure 3 describes an *adjacency* matrix (a matrix indicating 1 is a connection exists between two neurons, 0 if no connection exists) for the network topology shown in figure 3. Note, $\boldsymbol{\Psi}$ symbol serving as an identifier for the matrix is arbitrary.

$$\boldsymbol{\Psi} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Figure 3: Adjacency Matrix

3.3.2 Weights, Bias and Directionality

The topology discussed in this section has a topology not dissimilar to the generic topology discussed in earlier sections, however, certain key characteristics will be introduced and receive a formal definition. A neural network is deemed *'feedforward'* if each neuron may only communicate with a neuron or neurons in the next layer (i_i may communicate only with h_i, h_i may communicate only with o_i). This unidirectional nature of the multilayer feedforward network is represented by means of a directed graph (directionality is represented by arrows).

In most neural network topologies, merely connecting neurons is not sufficient for achieving a learning capability. Referencing the biological model of a neural network, it is such that when a neuron fires, it emits an electrical or chemical signal of a given strength based on the level of excitation exhibited by that neuron. This signal may be implemented in artificial neural networks by means of *weights.* A weight is a real value ($w_{ij} \in \mathbb{R}$) that is associated with a given connection between two neurons. In the event that a neuron fires, the connected neurons receive an input equal to the weight of the neuron(s) that have fired. To formally define this notion, Schalkoff (1997, 71) writes "w_{ij} represents the strength of the connection to neuron unit i from (either) neuron unit j or input j". Schalkoff (1997, 71) continues that "a large positive value of w_{ij} indicates a strong excitory input from unit or input j to unit i, whereas a large negative weight value may be used to represent a highly inhibitory input". Weights allow a certain value to be associated to a given neuron. This is of importance when multiple input datum are presented to a network. Masters (1993, 188) discusses a situation in which one neuron may have a higher precedence than another neuron due to the contribution that neuron provides to the output of the network. "[S]uppose we train a neural network to classify people as overweight or normal weight, based on four variables: Height, Hair Color, Eye Color and Weight. We may find that Hair Color and Eye Color are not important to the network's decision. The other two variables need more careful analysis, though. Certainly Height alone tells us nothing about obesity. Even Weight alone tells us little. But taken together, they tell all."

One special type of weight is a *bias*. As written by Schalkoff (1997, 86), "[b]iases may be used, for example, to selectively inhibit the activity of certain neurons". A bias provides a constant value of $b_i = 1.0$ into each layer subsequent to the input layer. Inclusion of a bias into a neural network topology is optional and is dependent on the nature of the problem encountered.

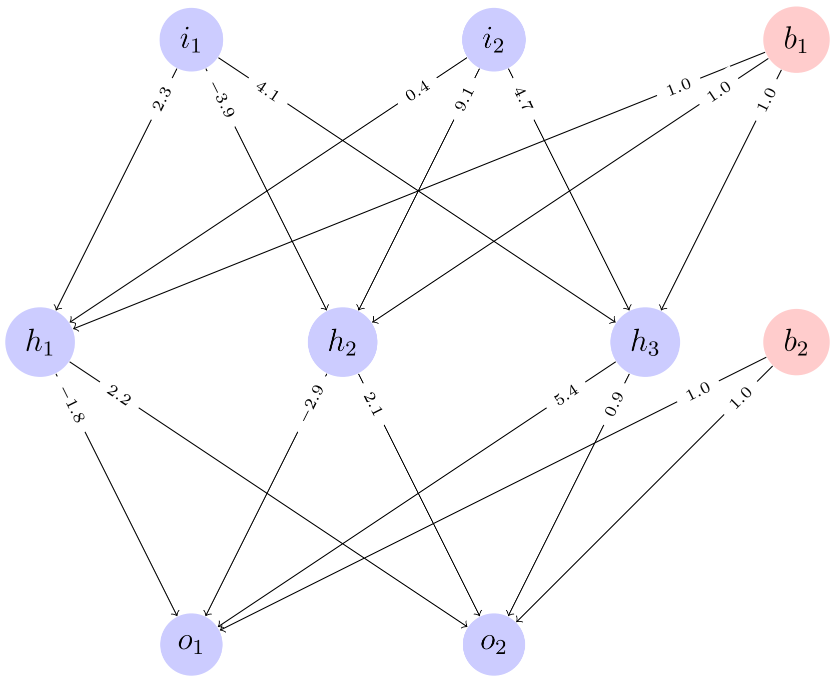

Figure 4: Inclusion of Weights, Bias and Directionality into the Network Topology

Heaton (2008, 85) describes the weight matrix as "the 'memory' of the neural network, and will allow the neural network to recall certain patterns when they are presented". There exists some notational differences in figure 5 from figure 3. The symbol ∞ replaces 0 for indicating that a connection between two neurons does not exist. This serves the purpose of conveying that a given connection between two neurons has an infinitely large weight and thus may not be traversed. Note that while this convention is employed in this paper, other conventions exist. Additionally, the *boolean* values that indicate a connection are replaced with values such that $w_{ij} \in \mathbb{R}$. The method in which elements are indexed is not reciprocal. When referencing a connection, rows in the weight matrix represent the start node and columns represent the end node. Consider briefly the connection between i_1 and h_1. Since this is unidirectional, a value exists at (i_1, h_1) but not at (h_1, i_1). A bias, if it exists, will not be included in the weight matrix as the value is constant ($b_i = 1.0$) and if included, the bias of a given layer may be assumed to be connected to each node in the following layer.

$$\boldsymbol{\Psi} = \begin{bmatrix} \infty & \infty & 2.3 & -3.9 & 4.1 & \infty & \infty \\ \infty & \infty & 0.4 & 9.1 & 4.7 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 1.8 & 2.2 \\ \infty & \infty & \infty & \infty & \infty & -2.9 & 2.1 \\ \infty & \infty & \infty & \infty & \infty & 5.4 & 0.9 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Figure 5: Weight Matrix

One consideration to be made when dealing with lower powered devices is handling the nonreciprocal nature of connections in a feedforward neural network. The weight matrix for this simple $2 \times 3 \times 2$ network requires 49 allocations for a *floating point* data type, of which 12 are required. One possible method of combating this misuse of resources is to amalgamate the collective weight matrix into a *row matrix.*

$$\textbf{Weight} = \begin{bmatrix} 2.3 & -3.9 & 4.1 & 0.4 & 9.1 & 4.7 & 1.8 & 2.2 & -2.9 & 2.1 & 5.4 & 0.9 \end{bmatrix}$$

Figure 6: Row Matrix for Input and Hidden Layers

3.4 Nonlinear Activation Functions

The degree to which a neuron may or may not fire is influenced heavily by an *activation function*. In the case of the original Perceptron, the activation function employed resembled a *piecewise function*. The Perceptron applied a *threshold* value ($t \in \mathbb{R}$) whereby if the weighted sum of the input exceeded the threshold value, the output from the neuron would be 1, otherwise 0 (Masters 1993, 80). For non-trivial problems, however, the boolean nature of the Perceptron limits the ability of the neural structure to classify problems that cannot be represented as a 1 or a 0 but instead rely on *fuzzy* definitions ($0 \leq i \leq 1, i \in \mathbb{R}$ or $-1 \leq i \leq 1, i \in \mathbb{R}$). For this reason, many modern neural network implementations employ a *continuous*, nonlinear, real-valued function. Two common activation functions will be discussed.

The most common activation function employed in modern networks is a *sigmoid* function named the *logistic activation function* (Masters 1993, 81). The logistic function scales the output of the neural network into proper ranges (Heaton 2008, 150). An inherent property of the logistic activation function is that the possible range of possible $y = f(x)$ values that may result is $0 \leq f(x) \leq 1$. The logistic activation function and associated visual representation can be seen in figures 7 and 8 respectively.

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

$x =$ Input

Figure 7: Logistic Activation Function

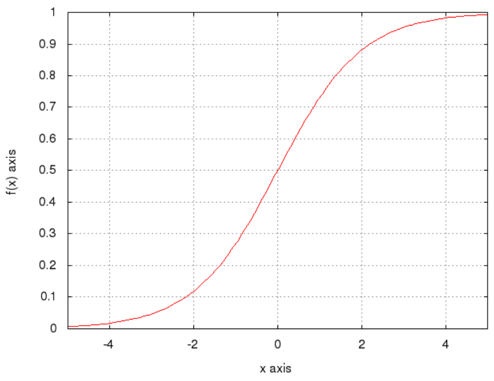

Figure 8: Logistic Activation Function Plot

In situations where a neural network is required to produce negative values, other activation functions must be considered. One such function sigmoidal function is the *hyperbolic tangent* or $tanh(x)$ function (figure 9. The hyperbolic tangent function is represented visually in figure 10.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$x =$ Input

Figure 9: Hyperbolic Tangent Activation Function

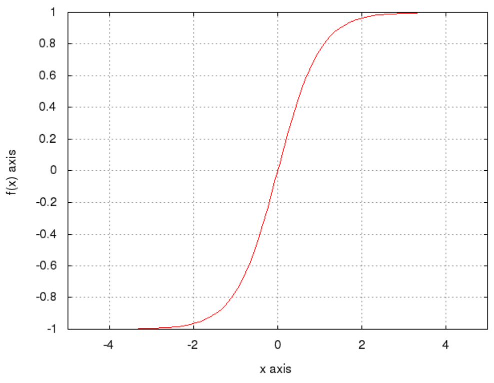

Figure 10: Hyperbolic Tangent Activation Function Plot

An important characteristic of sigmoid functions is that a sigmoid function never reach their theoretical maximum or minimum (Masters 1993, 82). As a result of this, Masters (1992, 82) suggests that neurons that utilise the logistic activation function be considered fully activated at values surrounding 0.9 and fully inhibited at values surrounding 0.1.

3.5 Neural Network Operation

The fundamental operation of a neural network may be described with the equation described in figure 11. Consider the neural network described in figure 14. The operations carried out to calculate the output of this network follow the operations described in figure 11. The example given describes the series of calculations carried out within a neural network for one hidden neuron. This process is repeated n times where n is directly proportional to the number of hidden and output neurons.

$$o_i = \left(\sum_{i=0}^{n-1} x_i w_i \right) + b_i$$

o_i = Output
x_i = Input
w_i = Weight
b_i = Bias

Figure 11: Calculating the Output of a Neuron

Figure 12 represents the input to the network for the purposes of this demonstration. As stated previously, these input values are arbitrary and serve no purpose other than to demonstrate the reaction of a neural network to some external stimulus over the input nodes.

$$\mathbf{Input} = \begin{bmatrix} 5.4 & 3.5 & 8.2 \end{bmatrix}$$

Figure 12: Neural Network Calculation Example Input Matrix

An additional step that can be incorporated into the calculation of the neural network, while not strictly necessary, is scaling the input to fall within the range $0 \leq input_i \leq 1$. Scaling may be implemented in future neural network calculations as the values in the implemented neural network in a future section of this paper do not necessarily occur within the range $0 \leq input_i \leq 1$. Scaling becomes beneficial as with large input values for the activation function, the output value may occupy the upper limits of the function and thus providing less of a variance in output between large samples. One common method for linearly scaling the input values is to use the equation $\frac{1}{input_i}$. This ensures a range of $0 \leq input_i 1$ that ensures that the ratios between input values are not altered, only the range. Figure 13 demonstrates the output of passing the unscaled and scaled values to the logistic activation function. Note that the scaled input allows a wider range of values to take advantage of the range of the logistic activation function. This difference is not necessarily visible on a single example, however, scaling may prove beneficial over the course of network output calculation. The use of input scaling is determined on a case-by-case basis and should only be implemented if the results of the network deem that a positive benefit is introduced by the use of scaling.

$$f(6.21) = \frac{1}{1+e^{-6.21}} = 0.99$$

$$f(0.16) = \frac{1}{1+e^{-0.16}} = 0.64$$

Unscaled $= 6.21$
Scaled $= \frac{1}{Unscaled} = 0.16$

Figure 13: Example of Logistic Activation Function Output - Scaled and Unscaled

Figure 14 highlights a simplified neural network with the resulting calculations involved in determining the output of this network. In this example, the input matrix will be scaled to ensure the input values fall within the range $0 \leq input_i \leq 1$.

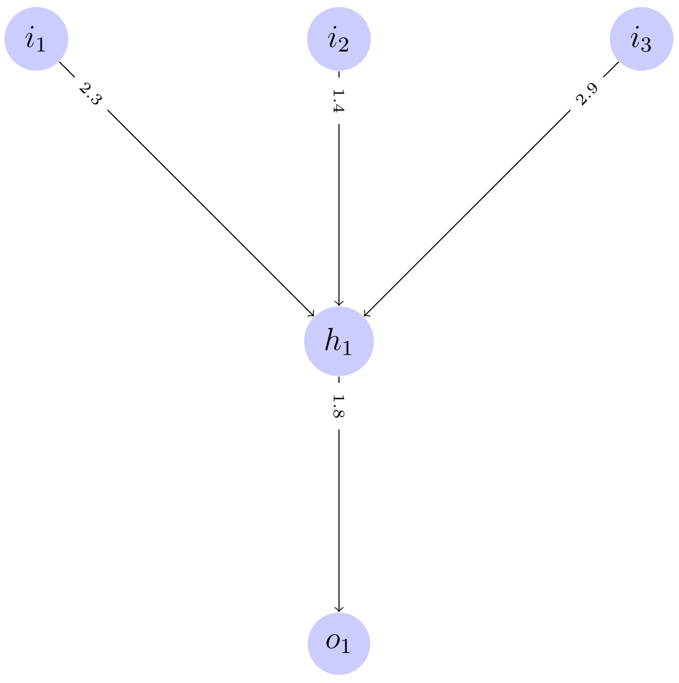

Figure 14: Neural Network Calculation Example Graph

To calculate the output of this simple network, the input matrix is first scaled to provide the values (rounded to 2 decimal places)

$$\textbf{Input} = \begin{bmatrix} \frac{1}{5.4} = 0.19 & \frac{1}{3.5} = 0.29 & \frac{1}{8.2} = 0.12 \end{bmatrix}$$

Add the product of each input and each weight $((x_i w_i) + b_i)$ with a bias of 1.0

$$h_1 = ((0.19 * 2.3) + (0.29 * 1.4) + (0.12 * 2.9) + 1.0) = 2.19$$

Applying the logistic activation function to this value provides

$$f(2.19) = \frac{1}{1 + e^{-2.19}} = 0.90$$

This new value serves as input to the output layer which in turn is multiplied by the weight of the connection

$$o_1 = ((0.90 * 1.8) + 1.0) = 2.62$$

Applying the logistic activation function to this value provides the output of the network for this given path

$$f(2.62) = \frac{1}{1 + e^{-2.62}} = 0.93$$

The output for this network and input combination is 0.93 which would, in most circumstances, be interpreted as a value of 1.0. This interpretation is due to the property of sigmoid functions dictating they only approach their maximum and minimum values of 0 and 1 as $\lim_{i \to \pm\infty}$.

3.6 Training and Validating a Neural Network

As a core requirement of a neural network, the adaptability of a neural network serves to distinguish 'intelligent' software from conventionally programmed software. The weights assigned to the neurons in a neural network determine the reaction of the network to input data samples. A network that has an optimal set of weights for each neuron, such that the network can accurately classify new (unseen) data to a low margin or error is considered ready for live use. The process of adjusting these weights is referred to as *training*. "One of the most important aspects of biological and artificial neurons is their learning capability", writes Cichocki et al. (1994, 47), "whereby synaptic (or connection) weights are adaptively changed according to an adaptive algorithm".

Neural networks are typically trained under one of three paradigms, *supervised* training, *unsupervised* and *hybrid* training. Under the supervised training paradigm, a *training set* is assembled that provides the neural network with a complete representation of the inputs that the network is required to learn (Masters 1993, 9). For each sample in the training set, an ideal output is included. As various subsets of the training set are presented to the neural network, the weights are adjusted based on how well the neural network performed and the validity of the result (Heaton 2008, 44). “One pass through the subset of training samples with an updating of the network’s weights is called an *epoch*”, writes Masters (1993, 10), “The number of samples in the subset is called the *epoch size*”.

Supervised training is the most common method for training a neural network (Heaton 2008, 44), however, there exists many situations in which ideal outputs are not available. Under the unsupervised training paradigm, a similar training set typical of the range of inputs is collected however no ideal output is presented to the network. Masters (1993, 10) describes the unsupervised training process as “[t]he process of training the network [,] letting it discover salient features of the training set, and using these features to group the inputs that it (the network) finds distinct”.

Heaton (2008, 45) describes hybrid training as “combin[ing] aspects of both supervised and unsupervised learning”. One such example of hybrid learning is *reinforcement learning*. During training that is carried out by reinforcement learning, a training set is presented to the network but no ideal outputs are provided. Instead, the neural network is informed manually as to the validity of the response it generated. (Masters 1993, 10) Training a neural network is a non-trivial task. Masters (1993, 94) states “Training a practical network is usually a numerical nightmare”. In most cases of neural network training, whether supervised or unsupervised, an initial weight matrix is populated with random values (Heaton 2008, 123). The optimum weight matrix is achieved when the neural network output error is minimised. Error is loosely defined as the difference between expected output and actual output. Heaton (2008, 123) advocates calculating the Root Mean Squared Error at such time when all samples in the training set have been presented to the network. Figure 15 describes the formula for the Root Mean Square Error calculation.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(t_i - o_i)^2}$$

n = Number of Epochs
t_i = Expected Output
o_i = Actual Output

Figure 15: Root Mean Squared Error (RMSE)

Several issues may arise in training, including that of *false minima.* False minima are those points where a training algorithm may settle with a given weight matrix and assume that the error achieved is the lowest error possible. It is the goal of any training algorithm to locate the *global minima*, the set of weights that provide the lowest error possible. *Local minima* are those points that are preceded by a download slope (decreasing error) and resolve to a slope of near zero (that is, any attempts to 'escape' this minima are met by slopes that provide a worse-case error). It is in these points that a naive training algorithm may land. The local minima is by definition, not the optimal minimum error. To combat this phenomenon, specialised algorithms for training neural networks to optimise while avoiding local minima have been introduced. Three such algorithms are discussed for the remainder of this section.

At such time that the training of a neural network is deemed complete, the network must then be *validated.* Network validation requires that the network is presented with unseen data to ensure correct operation (Heaton 2008, 45). Masters (1993, 10) stipulates that "[i]n many respects, proper validation is more important than proper training". During training, a network may 'overfit' the data, as Masters (1993, 10) describes, "[r]ather than learning the basic structure of the data, enabling it to generalize well, it learns irrelevant details of the individual cases. In such cases that the validation phase performs poorly, Heaton (2008, 46) suggests that potential cause may include unique validation data not represented by the training set. The process of validating a neural network will be discussed in further detail once the neural network has been implemented. Postponing a formal analysis of the validation stage until such time is beneficial as it is possible to track the progress of the neural network training algorithms by means of gathering diagnostic data from the neural network.

3.6.1 Backpropagation

Training by backpropagation is typically associated with the training of multilayer feed-forward neural networks, not least because the backpropagation training method was, as Masters (1993, 100) states, "the first practical method for training a multiplelayer feed-forward network". Masters (1993, 100) attributes the resurgence of interest into the field of artificial neural networks to the backpropagation training method. The backpropagation algorithm, as described by Masters (1993, 100) is a *gradient descent* algorithm, the operation of which is discussed for the remainder of this section.

During the training process of a neural network, the network is presented with a training set. The training set, as discussed in previous sections, contains a selection of data representative to the data that a neural network is expected to encounter during normal operation. The purpose of training a neural network is to minimise the error produced when a network is exposed to data. Training by backpropagation seeks to minimise this error by modifying the weights and thresholds associated with the neural network. The backpropagation algorithm works by first calculating the error of the output layer and determining the contribution of each neuron in the layer to the error returned. It follows that neurons with a higher weight value contribute more significantly and thus are to be adjusted more. This process then moves through the preceeding layers and adjusts each layer in turn. (Heaton 2008, 162-163)

For backpropagation to operate, the activation function must be *differentiable*. Differentiation represents the instantaneous rate of change of a given function as the input to the function changes. It is not required for the reader to be familiar with differential calculus to understand this paper as the *derivative* of any function is provided for the reader. The importance of derivatives with regard to training by backpropagation are described by Masters (1993, 95). "Derivatives are priceless to the numerical analyst who must optimize a function. If we know the partial derivative of the error with respect to each weight, we know (at least on a local scale) which way the weights must move in order to reduce the error". Figure 16 demonstrates the derivative for the logistic activation function.

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

x = Input

Figure 16: Derivative of the Logistic Activation Function

There are two additional important parameters to consider when training by backpropagation, *learning rate* and *momentum*. The learning rate is the rate at which the values are adjusted. A higher learning rate results in more radical adjustments being made in a shorter number of iterations. The momentum determines the degree to which training from a previous iteration may affect the current iteration. (Heaton 2008, 421)

3.6.2 Simulated Annealing

Annealing is a process employed in the field of *metallurgy*. The annealing process involves heating metal to a high temperature, at which point, the atoms have a greater degree of freedom than at lower temperatures. During the cooling process, the temperature is gradually lowered allowing the atoms to form stable structures. If the temperature is lowered too rapidly (*rapid quenching*), anomalies in the structures formed by the atoms may occur and thus the metal may be defective or brittle. Simulated annealing, as pioneered in the mid 1970s by Scott Kirkpatrick, applies these principles to such problems as function optimisation or searching a *nonlinear search space.* Simulated annealing, in addition, serves simultaneously as a highly effective training algorithm for artificial neural networks. The remainder of this section will discuss in more detail the operation of the simulated annealing process with regards to artificial neural networks. (Masters 1993, 118)

The simulated annealing algorithm operates on the weight matrix of an artificial neural network attempting to produce an optimal set of weights that minimise the error of the network. Press et al. (1999, 451) state that "[t]he basic ideas of simulated annealing are also applicable to optimization problems with continuous N-dimensional control spaces, e.g., finding the (ideally global) minimum of some function $f(x)$". A typical simulated annealing algorithm consists of two main processes (Heaton 2008, 201). A brief natural language description of the simulated annealing algorithm will be discussed as it pertains to artificial neural networks, utilising annealing nomenclature where required for the purposes of maintaining a relationship between the physical process and the simulated process.

Prior to discussing the operation of the simulated annealing algorithm, it is advantageous to first discuss the key parameters in use. The first parameter to be discussed is the *temperature* parameter. The temperature value represents the standard deviation of the employed random number generator (Masters 1993, 119). Initially, a high standard deviation is chosen to allow a large degree of potential values for the elements of the weight matrix to assume. This standard deviation is gradually reduced. Another important parameter to consider is the *current best solution.* The current best solution represents the *seed* of the random number generator that has performed most desirable (that is, produced the lowest neural network error) weight matrix and is replaced in such cases that another matrix produces a lower error. A final parameter to define prior to discussion of the operation of the algorithm is the *lower bound temperature.* The lower bound temperature provides a value determining the minimum temperature the algorithm may operate at. This may serve as a failsafe to prevent an infinite loop. (Masters 1993, 119)

The operation of the simulated annealing algorithm is simple (Press et al. 1999, 444).

An initial randomised weight matrix is produced as per the range permitted by the current temperature. The initial weight matrix is marked as the current best solution and provided to the artificial neural network. Upon completion of the required training data, the error is compared to that generated by the current best solution. In such cases that the error shows improvement over the existing current best solution then the previous solution is replaced with the solution containing more favourable characteristics. For each temperature, a predetermined number of iterations may occur. After this number of iterations has been met, the temperature is reduced and the process repeats, using the current best solution for each temperature as a central point for the random number generator. This process continues until such time that the lower bound temperature is met. (Heaton 2008, 201)

One important calculation performed in the simulated annealing process is the rate at which the temperature decreases. It is often beneficial to calculate a non-static step value to ensure that the temperature decreases in such a way that a similar number of steps may occur for all ranges of start and end temperatures. The equation commonly employed for calculating the step value for temperature decrease can be found in figure 17. (Heaton 2008, 204).

$$step = e^{\frac{\ln\left(\frac{start}{end}\right)}{cycles-1}}$$

$start$ = Temperature Upper Bound
end = Temperature Lower Bound
$cycles$ = Number of cycles

Figure 17: Step Function for Temperature Decrease in Simulated Annealing

It should be noted that simulated annealing is the preferred training method for the artificial neural networks deployed as part of this research.

3.6.3 Genetic Algorithms

Genetic algorithms are a prominent subset of the field of evolutionary computing. Evolutionary computing is an adaptive paradigm, mimicking natural processes such as natural selection and applying those principles to complex computational problems that may extrude large run-times under conventional methods. Genetic algorithms were originally devised by John Holland, along with his students and colleagues at the University of Michigan in the 1960's (Mitchell 1999, 2). As Mitchell (1999, 2) states, Genetic algorithms were initially intended to model biological evolution and adaptation, their contributions to problems in computer science such as optimisation and searching were incidental. Genetic algorithms, similarly to all evolutionary computational models, begin with a random *population* of candidate solutions. These samples combine and mutate, allowing those candidate solutions that provide superior character traits (read: better solutions) to continue to reproduce new candidate solutions (Heaton 2008, 177). This section first discusses the biological processes that will be modelled, then applies that understanding to computationally feasible algorithms that allow the searching nonlinear spaces for optimal threshold and weight values for a neural network.

A typical genetic algorithm begins by creating a *population* that are typically initialised to random values. Each member of the population represents one potential solution to the problem being studied. Upon successful generation of a population, the *fitness* of each solution is tested to determine to what degree a given solution performed. Following Darwinian theory of evolution, those potential solutions that performed most desirably have a higher probability of reproducing new potential solutions. Those solutions that reproduce share material defined by a random cutoff point, additionally, random *mutation* occurs to ensure new material is introduced. Mutations may or may not provide desirable characteristics, however, this is a moot point as the fitness verification will determine if these mutations proved successful. (Heaton 2008, 176-177)

4 Implementation

This section discusses the implementation specifics of an artificial neural network capable of differentiating between two conditions of the heart. Prior to specific implementation details being discussed, the objective detailed in section 1.2 will be expanded upon, formally defining the problem. It is deemed beneficial to the reader to formally introduce the objective at this later stage as the reader, assuming a logical and progressive order of reading, is now aware of the theoretical constructs that underpin the research in this paper. Promptly following this formal objective parametrisation is a discussion regarding data collection techniques. This section will discuss the source of the data and how it is to be prepared for exposure to the neural network. The construction and training of the neural network, including relevant C code samples are included and discussed detailing the construction and operation of the neural network as it is implemented and trained.

4.1 Formal Objective Parametrisation

The implementation stage of this research project will continue in three discrete parts. First, two samples will be obtained from Physiobank ATM, one representing a healthy signal and serving as a control, and another representing an onset of arrhythmia. Specific points from each sample are to be extracted. To ensure a large, consistent training and validation set, further input samples from each case are to be randomly generated based on ranges of the respective values over a 10 second interval from the original samples. The values obtained will represent the peak values of the *P, Q, R, S* and *T* waves.

The input values, coupled with expected output, are to be presented to the neural network. The neural network will employ simulated annealing as the training method. Training will occur in cycles and iterations until such time that the weight matrix in use by the network is sufficiently optimised, that is, the error is reduced to an acceptable rate.

The final stage of involves validating a new dataset against the trained neural network. The outcome expected is that the neural network can differentiate between the healthy signal and the unhealthy signal by means of outputting close to 0 when the healthy signal is detected and close to 1 if characteristics of the unhealthy signal are detected.

It is the purpose of the algorithm to identify and react to the diminished P wave of the ‘unhealthy’ sample. If the neural network is able to assimilate this diminished P wave characteristic into a weight matrix and provide a $\geq 70\%$ detection accuracy, this algorithm will be considered successful.

4.2 Input Data Preparation

The data generated is to serve as training data and as validation data so it follows that two discrete batches from each sample set are to be generated. To ensure consistent sample sets that may scale in size in the event that current training data proves inadequate or of insufficient length, a sample of peak values from the relevant data points and random values are to be generated within the maximum and minimum range of the sample values taken. This mimics potential fluctuations in a live signal within a small range that fall within an expected range for that given signal.

The data points of interested are peak values from the *P, Q, R, S* and *T* waves. These values are to be collected and hard coded into the resulting software implementation to alleviate system incompatibilities between systems such as Unix based systems (Linux, BSD, Macintosh) and Windows based systems. Included in each given sample in the training set is an expected value from which the accuracy of the classification made by the neural network may be measured.

4.2.1 Sample A - Healthy

The sample chosen to serve as the control sample, serving as the baseline from which deviations are considered unhealthy. This electrocardiogram readings chosen to represent the healthy control sample were obtained from Physiobank ATM, with the particular sample set being the *Challenge 2010 Test Set C* sample set. Only the readings from the *II* node were used for generating a base set of training and validation data. Noteworthy points in this sample include the existence of well-defined and formed P and T waves and an obviously discernible QRS complex.

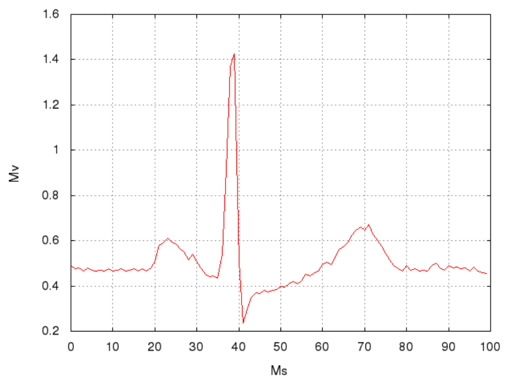

Figure 18: Sample A - Healthy Visualisation

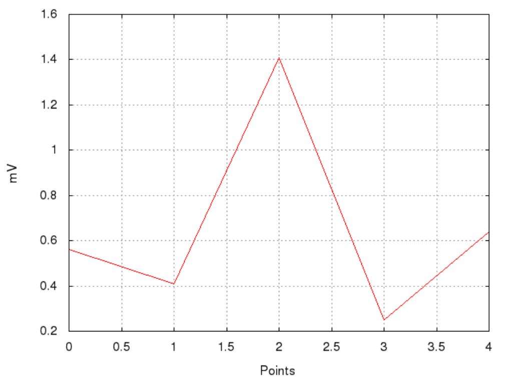

Figure 19: Sample A - Extracted Points

4.2.2 Sample B - Unhealthy

The unhealthy sample set employed to serve as an anomalous reading also originated from the Physiobank ATM database, specifically the *Intracardiac Atrian Fibrillation Database*. Similarly to Sample A, only the *II* node readings were obtained. This sample serves as an apt pattern for neural network classification due to the reduced P wave and elevated QRS complex peaks.

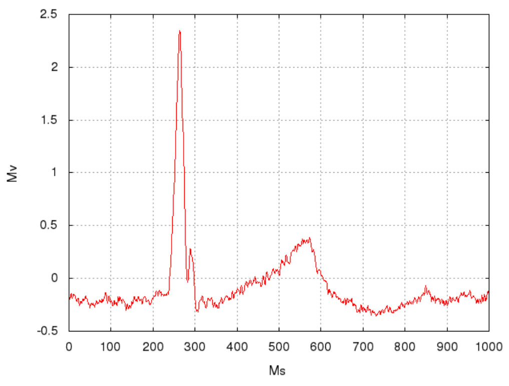

Figure 20: Sample B - Unhealthy Visualisation

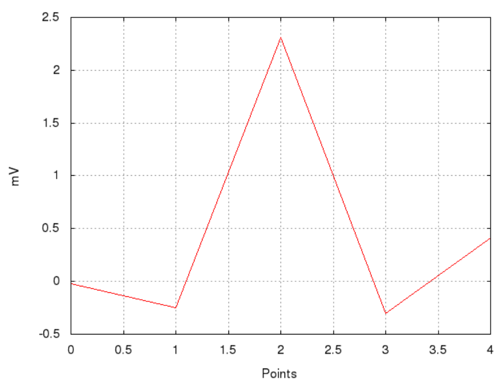

Figure 21: Sample B - Extracted Points

4.3 Neural Network Training

The training algorithm employed for this neural network implementation was the simulated annealing training method. Simulated annealing was chosen for the interesting properties it extrudes. Figure 22 provides provides a visual representation of the decrease in error from those weight matrices generated during the simulated annealing training algorithm that displayed increasingly optimal weight values. Figure 23 represents the error fluctuation over the entire training cycle. Displayed below is the output from the neural network algorithm that generated these error function plots. It is encouraging to note that the resulting accuracy of the network was 95.3%. A behaviour of this particular implementation that may be considered somewhat noteworthy is that while the simulated annealing training method produced an accuracy far in excess of the anticipated 70%, the algorithm ceased to optimise after a relatively short period. This is likely due to the trivial nature of the training data, and as a result, such positive results should not be expected so quickly from larger networks with considerably more complex input patterns.

```
Arch% bin/ecgnn
|--------------------------------------------|
| PQRST Analysis Artificial Neural Network  |
|   Thesis Work by Gavin Harper 08.05.2011   |
|--------------------------------------------|

(New Best Error: 0.68201713), (Temp: 400.00)
(New Best Error: 0.67112941), (Temp: 400.00)
```

```
(New Best Error: 0.65418777), (Temp: 400.00)
(New Best Error: 0.65175272), (Temp: 400.00)
(New Best Error: 0.64616436), (Temp: 400.00)
(New Best Error: 0.59962134), (Temp: 400.00)
(New Best Error: 0.54911686), (Temp: 400.00)
(New Best Error: 0.53239386), (Temp: 400.00)
(New Best Error: 0.49881412), (Temp: 400.00)
(New Best Error: 0.49048498), (Temp: 400.00)
(New Best Error: 0.48953939), (Temp: 400.00)
(New Best Error: 0.48944191), (Temp: 400.00)
(New Best Error: 0.48889536), (Temp: 400.00)
(New Best Error: 0.46721348), (Temp: 398.98)
(New Best Error: 0.33996150), (Temp: 387.78)
(New Best Error: 0.22568789), (Temp: 387.78)
(New Best Error: 0.21086273), (Temp: 387.78)
(New Best Error: 0.17777691), (Temp: 386.77)
(New Best Error: 0.13902444), (Temp: 386.77)
(New Best Error: 0.09757576), (Temp: 386.77)
(New Best Error: 0.05510005), (Temp: 386.77)
(New Best Error: 0.04694315), (Temp: 386.77)

|------Training Accuracy------|
(Best: 95.3057%)

|---Optimal Training Matrix---|
([00]: 3.51)
([01]: -5.86)
([02]: -5.35)
([03]: 1.69)
([04]: 9.15)
([05]: -1.81)
([06]: -0.09)
([07]: 1.87)
([08]: 1.29)
([09]: 1.10)
([10]: 2.80)
([11]: 3.84)
```

```
([12]: 15.66)
([13]: -2.61)
([14]: -6.26)
([15]: 17.80)
([16]: -9.56)
([17]: 1.87)
```

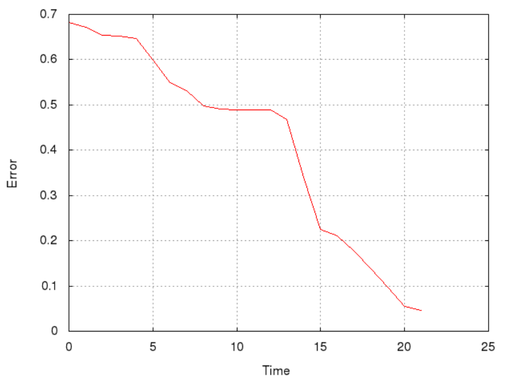

Figure 22: Simulated Annealing Best Error Function

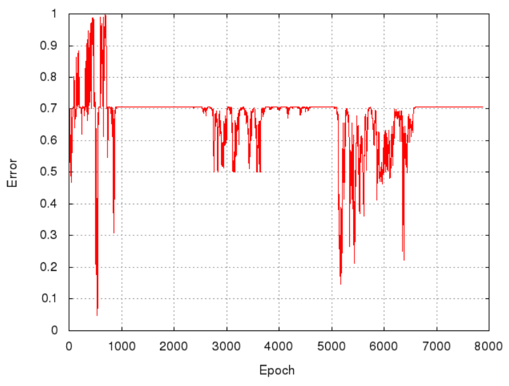

Figure 23: Simulated Annealing Global Error Function

5 Results

The results gathered during the training stage of the neural network implementation far surpass the original expectations placed upon the network in the design stage. Upon verifying the simulated annealing training algorithm operated correctly and efficiently, verification measures were then introduced. To verify the correct operation of the network, newly generated samples that had not been previously exposed to the network were introduced into the topology. To reiterate the purpose of the verification of a neural network, verification ensures that the network has identified generic patterns and formed generic rules with regards to the data, as opposed to learning only the training set. In this case, four tests were set up with threshold values of 70%. In almost all cases, the neural network correctly identified and differentiated between both samples at an accuracy and certainty of $\geq$ 70%. With such a trivial training set and small neural network topology, difficulties arise in determining if the neural network is *overfitting* data. However, in cases such as this, the results show definite potential.

```
|------Neural Validation------|

Entering a subset of Sample A (Expecting 0.0)
Actual: 0.12734469
PASS

Entering a subset of Sample B (Expecting 1.0)
Actual: 0.77359891
PASS

Entering a subset of Sample A (Expecting 0.0)
Actual: 0.16785368
PASS

Entering a subset of Sample B (Expecting 1.0)
Actual: 0.90986073
PASS
```

Detailed above is an except from a training cycle demonstrating correct identification to within 70%. One important consideration is the fundamental nature of *stochastic* algorithms. The results are likely to fluctuate between different executions and as a result, the figures obtained on one complete training cycle will vary from those in subsequent cycles. These results do not, however, conclusively prove or disprove the benefits or necessities of artificial neural networks in mobile health care beyond the trivial example discussed in this paper. Further research should be carried out, implementing artificial neural networks in relevant use cases before any commercial product should be introduced into the market. In many cases, the success of the simulated annealing training algorithm varied at run-time with values ranging from 50% to 90%, however this was remedied by better implementing the function that inflicted random modifications to the weight matrix. This research should be considered successful as the algorithm performed most admirably, exceeding original expectations.

6 Conclusion

Neural computing provides a solid foundation for designing digital signal analysis algorithms. Compared to traditional methods, there exists some initial complexity in the training and validation stages that may deter those unfamiliar with neural computing from considering a neural network as a candidate solution to a problem. An argument that may be deduced from this paper, however is more apparent with larger networks searching larger spaces, is that the accuracy and adaptability provided by a well-designed and well-trained neural network can provide significant benefits over longer periods of time.

While other, perfectly capable, neural network topologies exist such as the Self-Organising Map (SOM) or the Hopfield Network, the decision to utilise the multilayer feedforward neural network topology for this paper was borne from this specific topology serving as the archetypal neural network structure. Lessons learned from the multilayer feedforward neural network apply across a wide number of disciplines and topologies in neural computing and it serves as a sound basis from which neural computing may be introduced to readers unfamiliar with the paradigm.

It is important to note that the example provided in this paper does not constitute a real-world electrocardiogram analysis algorithm. It is not enough to present a neural network with three peak values from the P wave, QRS complex and T wave and expect that under laboratory conditions a suitable detection rate with a real-time signal could be achieved. This research intends to demonstrate the characteristics of neural computing that make it a suitable candidate to the problem of analysing a digital signal for anomalies. This paper, however, has suitably demonstrated that given appropriate conditions (to be discussed in a subsequent paragraph), a neural network is able to classify and identify data presented in a coherent manner to a high degree of accuracy.

To elaborate further on what may be deemed appropriate conditions, it is beneficial to consider first the nature of a digital electrocardiogram signal and then how a neural network operates. It can be stated that the challenge in analysing any digital signal is appropriately and consistently quantifying desired characteristics of the signal so they can be presented to a neural network in a predictable manner. To achieve this, the signal must be filtered and operated on using further mathematical techniques that are beyond the scope of this paper though are employed in the continuation of this research project post-thesis. Obtaining desired data from a real time signal, which may essentially

be considered as a one dimensional matrix of floating point numbers, is no trivial task. The success or lack thereof of this project depends entirely on an appropriate filtering technique being designed and implemented.
One final consideration is the suitability of artificial neural networks for mobile healthcare. Based soley on the knowledge that while this paper has demonstrated the adaptability and classification qualities of artificial neural networks, this example does not constitute a real-world case and as a result, should undergo further testing and experimentation before a sound recommendation can be made. This scepticism is based entirely on a lack of empirical data supporting the viability of artificial neural networks in a real-world case, but in no way seeks to cast down on the viability if, and only if, further specialised testing is carried out. It is the intention of this author to continue researching this matter and over a period of some years, correctly implement a neural computing solution capable of adapting to real-word cases.

7 Discussion

This paper and associated research has posed some curious and exciting challenges over the past weeks. It is always intriguing and holds a certain appeal to tackle challenges beyond that which is typically encountered at that given point in time with regards to education. All research and study into the topics discussed within the confines of this paper have been based entirely on a rigorous self-educating and experimentation process that has provided much insight into an area of computer science and engineering not previously considered.

This paper represents a subset of the research into developing artificially intelligent signal analysis software capable of differentiating electrocardiogram readings for Domuset Oy. Immediately after submission of this paper, the research process will resume with the shortcomings exemplified in this paper being met with new methods and approaches learned since the formalisation of the thesis topic. Research into the field of artificial intelligence and signal analysis provide exciting avenues for self-betterment and challenges beyond what one would normally encounter at this stage in the education process. It is such that this research project will continue, directly or indirectly relating to further studies, for some years to come. It is hoped that the problem of machine based, intelligent electrocardiogram analysis can be continued beyond Master of Science level education and into the realm of doctorate degrees.

In spite of the relative success in demonstrating the viability of artificial neural networks for mobile medical analysis, it is still such that the problem tackled in this paper was a reduced, trivialised form of the challenges faced external to this thesis. The respective fields of Artificial Intelligence, Signal Analysis and Electrocardiography are vast and interesting topics too large to be covered with any consistent accuracy in a short paper such as this. One challenge encountered during the writing of this paper was approaching what is considered a vastly complex subject in such a way that the relevant information may be provided to a reader with no background knowledge in artificial intelligence, signal analysis nor electrocardiography. It is the case that many fundamental operations in Neural Networks, to isolate one example, are constructed from multiple smaller, simpler parts. To cover all of these parts in any detail including tracing back the theory behind each part to prevent assumed knowledge being applied proved a challenge.

8 Bibliography

Ashley, E. A. & Niebauer, J. 2004. Conquering the ECG - Cardiology Explained. Date of retrieval 03/05/2011.
http://www.ncbi.nlm.nih.gov/books/NBK2214/

Cohen, J. S. Year Unknown. Electrocardiogram (ECG). Date of retrieval 03/05/2011.
http://www.emedicinehealth.com/electrocardiogram_ecg/article_em.htm
Cichocki, A & Unbehauen, R. 1994. Neural Networks for optimisation and signal porcessing. Stuttgart: John Wiley & Sons Ltd. & B. G. Teubner.

Cormen, T. H. & Leiserson, C. E. & Rivest, R. L. & Stein, C. 2009. Introduction to Algorithms. Third Edition. Massachusetts: The MIT Press.

Croft, A. & Davison, R. & Hargreaves, M. 2003. Engineering Mathematics: A Foundation for Electronic, Electrical, Communications and Systems Engineers. Third Edition. Essex: Pearson Education Limited.

Haykin, S. 1999. Neural Networks A Comprehensive Foundation. Second Edition. New Jersey: Prentice-Hall, Inc.

Heaton, J. 2008. Introduction to Neural Networks with Java. Second Edition. St Louis: Heaton Research, Inc.

Masters, T. 1993. Practical Neural Network Recipes in C++. San Diego: Academic Press, Inc.

MIT. 2005. Inventor of the Week: Archive. Date of retrieval 03/05/2011.
http://web.mit.edu/invent/iow/einthoven.html

Mitchell, M. 1999. An Introduction to Genetic Algorithms. Fifth Printing. Massachusetts: The MIT Press.

Peterson, J. & Kynor, D. 2003. Intracardiac Atrial Fibrillation Database. Date of retrieval 08/05/2011.
http://www.physionet.org/physiobank/database/iafdb/

Physionet. 2010. Challenge 2010 Test Set C. Date of retrieval 08/05/2011. *http://www.physionet.org/physiobank/database/challenge/2010/set-c/*

Press, W. H. & Teukolsky, S. A. & Vetterling, W. T. & Flannery, B. T. 1999. Numerical Recipes in C: The Art of Scientific Computing. Second Edition. Cambridge: The Press Syndicate of the University of Cambridge.

Schalkoff, R. J. 1997. Artificial Neural Networks. New York: McGraw-Hill

Searle, J. R. 1980. Minds, brains, and programs. Behavioral and Brain Sciences. 3 (3). 417-424.

Sedgewick, R. 1992. Algorithms in C++. Massachusetts: Addison-Wesley Publishing Company, Inc.

Turing, A. M. 1950. Computing Machinery and Intelligence. Mind. 59 (236), 433-460.

Tveter, D. R. 1998. The Pattern Recognition Basis of Artificial Intelligence. Los Alamitos: IEEE Computer Society Press.

A Constants and Notations

A.1 Constants

$e = 2.71828\ldots$

∞ = Infinity

A.2 Notations

α A value represented by some non-numerical character

α_n The '*nth*' value contained within α

α^β The product of a value, α multiplied by itself β times

$\sqrt{\alpha}$ Denotes the square root of α

$\alpha\beta$ The product (multiplication) of two values, α and *beta*

$\frac{\alpha}{\beta}$ The division of two values, α divided by β

$\alpha > \beta$ The value of α is greater than the value of β

$\alpha < \beta$ The value of α is less than the value of β

$\alpha \geq \beta$ The value of α is greater than or equal to the value of β

$\alpha \leq \beta$ The value of α is less than or equal to the value of β

$\alpha \neq \beta$ The value of α is not equal to the value of β

$0 \leq \alpha \leq 1$ The value of α is in the range $0 \rightarrow 1$

$\sum_{i=0}^{n} \alpha_i\beta_i$ Denotes the summation of the product for all α_i and β_i in the range $i \rightarrow n$

$\in$ Denotes set membership. $\alpha \in A$ is read as α is contained within some set A

$f(x)$ Denotes some function that receives an independent variable x and outputs exactly one dependent variable y

$f'(x)$ Denotes the derivative of some function $f(x)$

$\frac{dy}{dx}f(x)$ Denotes the derivative of some function $f(x)$

$\ln(\alpha)$ The base of the natural logarithm is e. The natural logarithm of α is the power e is raised to to equal α

$\lim_{x\to\infty}$ The limit of a function as x tends to infinity or some value

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ Denotes some 2×2 matrix.

A.3 Number Systems

$\mathbb{R}$ The set of all Real Numbers $(x \in \mathbb{R} : -\infty \leq x \leq \infty)$

B GIT Log

```
commit e0b9d4b51f4992301a8ece0445b4a7b261dae753
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 19:50:27 2011 +0300

    Final Commit

commit 1ec3d03cd08802791bc43d3c527a78d24fd6e834
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 16:47:59 2011 +0300

    Removed obsolete matrix duplication function

commit acb03d55706d2125f62970db4c243a42f86e227d
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 05:53:21 2011 +0300

    Pedantic OCD kicked in. Had to update the validation heading

commit e7a0bf865f3e36dc0d5223e285bc31d4e0171e22
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 05:34:40 2011 +0300

    Final version - commented

commit c89f649391c508fe390b89277413105458783a27
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 02:55:31 2011 +0300

    Removed redundant scale input function

commit 6c0a508936c89810c30ac45868015e753b042b4f
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 02:40:07 2011 +0300

    Network seems to be handling training data well

commit 776f645b89654d72cdc93f625932d77beaf054f4
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 00:29:21 2011 +0300

    General stability updates

commit 2e0880f5b4f8041e27a873fb9e9af3505ee4fa2d
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 00:28:31 2011 +0300

    Added Root Mean Squared Error function

commit 63f382de3ed5b0d62031d8cb2c2944a52657732b
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Mon May 9 00:28:10 2011 +0300

    Added sample generator
```

```
commit 3e19a4e19d130acf00dc9f293f0eb6a6f9f2e8b4
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 18:46:39 2011 +0300

    Fixed inconsistency in error display

commit cfec9da604f4720d3a1eafbb3fdd14b47a2657c5
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 18:38:27 2011 +0300

    Added annealing shake function

commit c5add29590896a686fbde85eb2813bda09276cec
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 17:48:52 2011 +0300

    Added error function

commit 7e9d28299f8a0da808cc9e240413270569656243
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 17:15:04 2011 +0300

    Fixed annealing matrix duplication

commit 38fdb7f031ae5f11070d330923ec260b8da4153e
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 16:50:57 2011 +0300

    Fixed header inclusion error in fnduplicatematrix.h

commit 302c746c6ddbe050e4d7835beddc3cf9eda91900
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 16:48:28 2011 +0300

    Added function for duplicating the weightmatrix

commit 2e2cace1e75404366d2e5d1f329965eb0aa3a9a0
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 16:40:19 2011 +0300

    Added step function for annealing temperature decrease

commit 2a02cfee1e90c5fdf47f8ffd96b20925a4e8ee85
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 16:27:08 2011 +0300

    Fixed inaccurate calculation values in fncalculatenetwork.c

commit bba9efa0d5f3acc61ea66232e07f2d3e9e80a635
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 16:08:29 2011 +0300

    Added simulated annealing training function

commit ec2273078b345b7b79e75a6beba8162cfb7946c6
```

```
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 14:42:55 2011 +0300

    Added random number functionality

commit c569cee9cf5d4ad8af3e4bb7f75541480735947d
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 14:22:23 2011 +0300

    Added sigmoid activation function

commit 2ef17fe10f28cadb4d4e922bd728119f95492e7a
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 14:21:59 2011 +0300

    Added function to scale input to range 0..1

commit f819813c2edfa22d2a7d2042a9f13e710d6e6091
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 14:21:26 2011 +0300

    Added function to dynamically assign a weight matrix

commit b8956998a72ed660b91031dad49ea6758908d47e
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 14:20:53 2011 +0300

    Added network calculation functionality

commit c33fd5bf2b353a1348b31ce7dd748b08ae6d711a
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 14:20:27 2011 +0300

    Added Makefile

commit 0ae9829aaec62eac0931d2aeef1dd4ad5f734b67
Author: Gavin Harper <k9haga00@students.oamk.fi>
Date:   Sun May 8 14:20:09 2011 +0300

    Repo initialisation
```

C Makefile

```
#
# Makefile for Thesis Work: PQRST Analysis (Multilayer Feedforward Neural Network)
#
# Gavin Harper - F8827
#
# Compiler: Gnu GCC 4.6.0 20110415
#

CC=gcc
CFLAGS= -lm -O3 -Wall
OBJ = obj/main.o obj/fngetweightmatrix.o obj/fncalculatenetwork.o \
    obj/fnactivationfunction.o obj/fngetrandom.o obj/fnanneal.o \
    obj/fnannealstep.o obj/fngeterror.o obj/fnannealshake.o \
    obj/fngetrmse.o


FNGETWEIGHTMATRIX = fngetweightmatrix.h fngetweightmatrix.c
FNCALCULATENETWORK = fncalculatenetwork.h fncalculatenetwork.c
FNACTIVATIONFUNCTION = fnactivationfunction.h fnactivationfunction.c
FNGETRANDOM = fngetrandom.h fngetrandom.c
FNANNEAL = fnanneal.h fnanneal.c
FNANNEALSTEP = fnannealstep.h fnannealstep.c
FNANNEALSHAKE = fnannealshake.h fnannealshake.c
FNGETERROR = fngeterror.h fngeterror.c
FNGETRMSE = fngetrmse.h fngetrmse.c

all: ecgnn

ecgnn: $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) -o bin/ecgnn

obj/main.o: main.c $(FNANNEAL)
    $(CC) $(CFLAGS) -c main.c -o obj/main.o

obj/fngetweightmatrix.o: $(FNGETWEIGHTMATRIX) $(FNGETWEIGHTMATRIX)
    $(CC) $(CFLAGS) -c fngetweightmatrix.c -o obj/fngetweightmatrix.o

obj/fncalculatenetwork.o: $(FNCALCULATENETWORK) $(FNACTIVATIONFUNCTION)
    $(CC) $(CFLAGS) -c fncalculatenetwork.c -o obj/fncalculatenetwork.o

obj/fnactivationfunction.o: $(FNACTIVATIONFUNCTION)
    $(CC) $(CFLAGS) -c fnactivationfunction.c -o obj/fnactivationfunction.o

obj/fngetrandom.o: $(FNGETRANDOM)
    $(CC) $(CFLAGS) -c fngetrandom.c -o obj/fngetrandom.o

obj/fnanneal.o: $(FNANNEAL) $(FNANNEALSTEP) $(FNANNEALSHAKE) \
        $(FNGETERROR) $(FNGETWEIGHTMATRIX) $(FNGETERROR) \
        $(FNGETRMSE)
    $(CC) $(CFLAGS) -c fnanneal.c -o obj/fnanneal.o

obj/fnannealstep.o: $(FNANNEALSTEP)
    $(CC) $(CFLAGS) -c fnannealstep.c -o obj/fnannealstep.o
```

```
obj/fngeterror.o: $(FNGETERROR)
	$(CC) $(CFLAGS) -c fngeterror.c -o obj/fngeterror.o

obj/fngetrmse.o: $(FNGETRMSE)
	$(CC) $(CFLAGS) -c fngetrmse.c -o obj/fngetrmse.o

obj/fnannealshake.o: $(FNANNEALSHAKE)
	$(CC) $(CFLAGS) -c fnannealshake.c -o obj/fnannealshake.o

clean:
	rm -rf bin/ecgnn dgn/* $(OBJ)
```

D Source Code

```
                    ####################
                    #***** main.c *****#
                    ####################

#include <stdio.h>
#include <stdlib.h>

#include "fnanneal.h"

int main(int argc, char *argv[])
{
    printf("|——————————————————————————————————————————|\n");
    printf("| PQRST Analysis Artificial Neural Network |\n");
    printf("|   Thesis Work by Gavin Harper 08.05.2011  |\n");
    printf("|——————————————————————————————————————————|\n");
    printf("\n");

    fnAnneal();

    return 0;
}


                    ########################
                    #***** fnanneal.h *****#
                    ########################

#ifndef FNANNEAL_H_
#define FNANNEAL_H_

#include <float.h>
#include <stdio.h>

#include "fnannealstep.h"
#include "fnannealshake.h"
#include "fngetweightmatrix.h"
#include "fngeterror.h"
#include "fngetrmse.h"
#include "fncalculatenetwork.h"
#include "fngetrandom.h"

/* Train the network using simulated annealing. Due to simplicity,
network validation is included in this function to reduce
unnecessary files */

void fnAnneal();

#endif
```

```
                    ########################
                    #***** fnanneal.c *****#
                    ########################

#include "fnanneal.h"

void fnAnneal()
{
    int eSize = 20;           /* Size of error matrix */
    int nCycles = 40;         /* Number of cycles per temperature */
    double bErr = DBL_MAX;    /* High original best error */
    double sTemp = 400.0;     /* Starting temperature */
    double eTemp = 200.0;     /* Ending temperature */
    double cTemp = sTemp;     /* Current Temperature */
    double step;              /* Value by which temperaure is decreased */
    double cErr;              /* Current error */
    double oNet;              /* Output of Neural Network Calculation */

    double *wMtx = fnGetWeightMatrix(); /* Weight matrix */
    double *bMtx = (double *) malloc(30 * sizeof(double)); /* Best matrix */
    double *eMtx = (double *) malloc(eSize * sizeof(double)); /* Error matrix */

    initrand(); /* Seeds random number generator */

    while(cTemp > eTemp)
    {
        int i;
        for(i = 0; i < nCycles; i++)
        {
            fnAnnealShake(sTemp, cTemp, wMtx); /* Alter the weight matrix */

            /* Calculate output then error of neural network */
            oNet = fnCalculateNetwork(wMtx, 0.59, 0.41, 1.39, 0.22, 0.6);
            eMtx[0] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.03, -0.21, 2.22, -0.29, 0.54);
            eMtx[1] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.59, 0.41, 1.39, 0.23, 0.61);
            eMtx[2] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.05, -0.2, 2.42, -0.2, 0.37);
            eMtx[3] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.55, 0.41, 1.43, 0.23, 0.61);
            eMtx[4] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.08, -0.16, 2.36, -0.25, 0.44);
            eMtx[5] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.57, 0.41, 1.4, 0.25, 0.65);
            eMtx[6] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.09, -0.17, 2.2, -0.24, 0.46);
            eMtx[7] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.56, 0.43, 1.39, 0.24, 0.64);
            eMtx[8] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.05, -0.19, 2.2, -0.22, 0.53);
            eMtx[9] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.57, 0.4, 1.4, 0.23, 0.62);
            eMtx[10] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.08, -0.15, 2.19, -0.26, 0.51);
            eMtx[11] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.57, 0.43, 1.41, 0.26, 0.63);
            eMtx[12] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.05, -0.23, 2.25, -0.22, 0.39);
            eMtx[13] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.55, 0.41, 1.39, 0.25, 0.64);
            eMtx[14] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.05, -0.21, 2.25, -0.3, 0.47);
            eMtx[15] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.57, 0.43, 1.42, 0.25, 0.64);
            eMtx[16] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.06, -0.24, 2.2, -0.25, 0.41);
            eMtx[17] = fnGetError(oNet, 1.0);
            oNet = fnCalculateNetwork(wMtx, 0.58, 0.41, 1.4, 0.22, 0.64);
            eMtx[18] = fnGetError(oNet, 0.0);
            oNet = fnCalculateNetwork(wMtx, -0.05, -0.24, 2.38, -0.3, 0.49);
            eMtx[19] = fnGetError(oNet, 1.0);

            /* Return RMSE */
            cErr = fnGetRMSE(eMtx, eSize);

            /* Replace current error with best if current is lower */
            if(cErr < bErr)
            {
                bErr = cErr;
                /* Dump current matrix into best matrix */
                int i;
                for(i = 0; i < 30; i++)
                {
                    bMtx[i] = wMtx[i];
                }

                printf("(New Best Error: %.8f), (Temp: %.2f)\n", bErr, cTemp);
            }

            /* Write the global and best error values for plotting */
            FILE *fp;
            fp = fopen("dgn/global.err", "a");
                fprintf(fp, "%.8f\n", cErr);
            fclose(fp);
        }
        /* Calculate temperature decrease */
        step = fnAnnealStep(sTemp, eTemp, nCycles);
        cTemp -= step;
    }
    /* Statistics */
    printf("\n");
    printf("|------Training Accuracy------|\n");
    printf("(Best: %.4f\%)\n", (100.0 - (bErr * 100.0)));

    printf("\n");
    printf("|---Optimal Training Matrix---|\n");
    int j;
    for(j = 0; j < 18; j++)
    {
        if(j < 10)
        {
            printf("([0%d]: %.2f)\n", j, bMtx[j]);
        }
        else
        {
            printf("([%d]: %.2f)\n", j, bMtx[j]);
        }

    }

    /* Validation */

    /* 70% Accuracy expected. This is somewhat high for such a
    small training set and network size but it seems to pass
    consistently */
    printf("\n");
    printf("|------Neural Validation------|\n");
    printf("\n");

    printf("Entering a subset of Sample A (Expecting 0.0)\n");
    oNet = fnCalculateNetwork(bMtx, 0.55, 0.43, 1.41, 0.26, 0.61);
    printf("Actual: %.8f\n", oNet);
    if(oNet <= 0.3)
    {
        printf("PASS\n");
    }
    else
    {
        printf("FAIL\n");
    }
    printf("\n");

    printf("Entering a subset of Sample B (Expecting 1.0)\n");
    oNet = fnCalculateNetwork(bMtx, -0.02, -0.17, 2.4, -0.26, 0.38);
    printf("Actual: %.8f\n", oNet);
    if(oNet >= 0.7)
    {
        printf("PASS\n");
    }
    else
    {
        printf("FAIL\n");
    }
    printf("\n");

    printf("Entering a subset of Sample A (Expecting 0.0)\n");
    oNet = fnCalculateNetwork(bMtx, 0.57, 0.42, 1.43, 0.25, 0.65);
    printf("Actual: %.8f\n", oNet);
    if(oNet <= 0.3)
    {
        printf("PASS\n");
    }
    else
    {
        printf("FAIL\n");
    }
    printf("\n");
```

```
    printf("Entering a subset of Sample B (Expecting 1.0)\n");
    oNet = fnCalculateNetwork(bMtx, -0.05, -0.2, 2.27, -0.29, 0.51);
    printf("Actual: %.8f\n", oNet);
    if(oNet >= 0.7)
    {
        printf("PASS\n");
    }
    else
    {
        printf("FAIL\n");
    }
    printf("\n");

    free(wMtx);
    free(eMtx);
}
```

```
                    ##############################
                    #***** fnannealshake.h *****#
                    ##############################

#ifndef FNANNEALSHAKE_H_
#define FNANNEALSHAKE_H_

#include "fngetrandom.h"

/* Shake the weight matrix elements to generate randomness */
void fnAnnealShake(double sTemp, double cTemp, double *wMtx);

#endif
```

```
                    ##############################
                    #***** fnannealshake.c *****#
                    ##############################

#include "fnannealshake.h"

void fnAnnealShake(double sTemp, double cTemp, double *wMtx)
{
    int i;
    for(i = 0; i < 30; i++)
    {
        /* Determine random value change */
        double delta = 0.5 - fnGetRandom();
        delta *= (cTemp / sTemp);
        wMtx[i] = wMtx[i] + delta;
    }
}
```

```
                    ############################
                    #***** fnannealstep.h *****#
                    ############################

#ifndef FNANNEALSTEP_H_
#define FNANNEALSTEP_H_

#include <stdlib.h>
#include <math.h>

/* Reduce temperature value logarithmically */
double fnAnnealStep(double sTemp, double eTemp, int nCycles);

#endif


                    ############################
                    #***** fnannealstep.c *****#
                    ############################

#include "fnannealstep.h"

double fnAnnealStep(double sTemp, double eTemp, int nCycles)
{
    return exp((log(sTemp / eTemp)) / ((double)nCycles - 1));
}


                    #################################
                    #***** fngetweightmatrix.h *****#
                    #################################

#ifndef FNGETWEIGHTMATRIX_h_
#define FNGETWEIGHTMATRIX_h_

#include <stdlib.h>

#include "fngetrandom.h"

/* Return a random weight matrix to begin training */
double *fnGetWeightMatrix();

#endif
```

```
                    ##################################
                    #***** fngetweightmatrix.c *****#
                    ##################################

#include "fngetweightmatrix.h"

double *fnGetWeightMatrix()
{
    int i;
    /* 30 is the weight matrix size. Yes, this should be
    a #define but it is 3AM */
    double *wMtx = (double *) malloc(30 * sizeof(double));

    for(i = 0; i < 30; i++)
    {
        wMtx[i] = fnGetRandom();
    }

    return wMtx;
}


                    ###################################
                    #***** fncalculatenetwork.h *****#
                    ###################################

#ifndef FNCALCULATENETWORK_H_
#define FNCALCULATENETWORK_H_

#include <stdio.h>
#include <stdlib.h>

#include "fnactivationfunction.h"

/* Calculate neural network output */
double fnCalculateNetwork(double *wMtx, double i1, double i2 \
double i3, double i4, double i5);

#endif


                    ###################################
                    #***** fncalculatenetwork.c *****#
                    ###################################

#include "fncalculatenetwork.h"

double fnCalculateNetwork(double *wMtx, double i1, double i2, \
double i3, double i4, double i5)
{
    double h1 = 0.0; /* Hidden Neuron 1 */
    double h2 = 0.0; /* Hidden Neuron 2 */
    double h3 = 0.0; /* etc */
    double h4 = 0.0;
    double h5 = 0.0;
    double o1 = 0.0; /* Output Neuron 1 */
    /* Get product of all input values and respective weights
    then squash with the logistic activation function */
    h1 = ((i1 * wMtx[0]) + (i2 + wMtx[1]) + (i3 + wMtx[2]) + \
            (i4 * wMtx[3]) + (i5 * wMtx[4]) + 1.0);
    h1 = fnActivationFunction(h1);

    h2 = ((i1 * wMtx[5]) + (i2 + wMtx[6]) + (i3 + wMtx[7]) + \
            (i4 * wMtx[8]) + (i5 * wMtx[9]) + 1.0);
    h2 = fnActivationFunction(h2);

    h3 = ((i1 * wMtx[10]) + (i2 + wMtx[11]) + (i3 + wMtx[12]) + \
            (i4 * wMtx[13]) + (i5 * wMtx[14]) + 1.0);
    h3 = fnActivationFunction(h3);

    h4 = ((i1 * wMtx[15]) + (i2 + wMtx[16]) + (i3 + wMtx[17]) + \
            (i4 * wMtx[18]) + (i5 * wMtx[19]) + 1.0);
    h4 = fnActivationFunction(h4);

    h5 = ((i1 * wMtx[20]) + (i2 + wMtx[21]) + (i3 + wMtx[22]) + \
            (i4 * wMtx[23]) + (i5 * wMtx[24]) + 1.0);
    h5 = fnActivationFunction(h5);

    o1 = ((h1 * wMtx[25]) + (h2 * wMtx[26]) + (h3 * wMtx[27]) + \
            (h4 * wMtx[28]) + (h5 * wMtx[29]) + 1.0);
    o1 = fnActivationFunction(o1);

    return o1;
}
```

```
                    ###################################
                    #***** fnactivationfunction.h *****#
                    ###################################
```

```
#ifndef FNACTIVATIONFUNCTION_H_
#define FNACTIVATIONFUNCTION_H_

#include <stdlib.h>
#include <math.h>

/* Squash neuron output values to the range 0..1 */
double fnActivationFunction(double nVal);

#endif
```

```
                    ####################################
                    #***** fnactivationfunction.c *****#
                    ####################################

#include "fnactivationfunction.h"

double fnActivationFunction(double nVal)
{
    return ((1.0) / (1.0 + exp(-nVal)));
}


                    ##########################
                    #***** fngeterror.h *****#
                    ##########################

#ifndef FNGETERROR_H_
#define FNGETERROR_H_

/* Return simple error - Difference between actual and expected output */
double fnGetError(double actual, double expected);

#endif


                    ##########################
                    #***** fngeterror.c *****#
                    ##########################

#include "fngeterror.h"

double fnGetError(double actual, double expected)
{
    return (actual - expected);
}


                    #########################
                    #***** fngetrmse.h *****#
                    #########################

#ifndef FNGETRMSE_H_
#define FNGETRMSE_H_

#include <math.h>

/* Return Root Mean Squared Error for training cycle */
double fnGetRMSE(double *eMtx, int nCycles);

#endif
```

```
                    ##########################
                     #***** fngetrmse.c *****#
                    ##########################

#include "fngetrmse.h"

double fnGetRMSE(double *eMtx, int nCycles)
{
    double sigma = 0.0;

    int i;
    for(i = 0; i < nCycles; i++)
    {
        sigma += pow(eMtx[i], 2);
    }

    return sqrt(sigma / nCycles);
}


                    ############################
                     #***** fngetrandom.h *****#
                    ############################

#ifndef FNGETRANDOM_H_
#define FNGETRANDOM_H_

#include <stdlib.h>
#include <float.h>
#include <time.h>

/* Init random seed */
void initrand();
/* Get random float */
double fnGetRandom();

#endif


                    ############################
                     #***** fngetrandom.c *****#
                    ############################

#include "fngetrandom.h"

void initrand()
{
    /* Seed random number generator to increase random distribution */
    srand((unsigned)(time(0)));
}

double fnGetRandom()
{
    /* Return a random double precision value */
    return ((double)rand() / RAND_MAX);
}
```

```
##############################
#***** writesamples.py *****#
##############################
```

```python
#!/usr/bin/python2

# Python script to generate C function calls for sending
# generated ECG values to the neural network

from random import *     # Import random functionality

f = open('Smp-T.ecg', 'w')       # Dump values to a text tile
j = 0    # Loop counter to ensure that 40 values may be written

for i in range(20):

        p = str(round(uniform(0.55, 0.62), 2))  # Healthy P wave range
        q = str(round(uniform(0.40, 0.46), 2))  # Healthy Q wave range
        r = str(round(uniform(1.38, 1.45), 2))  # Healthy R wave range
        s = str(round(uniform(0.26, 0.21), 2))  # Healthy S wave range
        t = str(round(uniform(0.60, 0.66), 2))  # Healthy T wave range
        o = str(0.00)    # Expected output

        # Generate C function calls
        calcstr = "oNet = fnCalculateNetwork(wMtx, " + str(p) + ", " + str(q) + \
                ", " + str(r) + ", " + str(s) + ", " + str(t) + ");" + "\n"
        f.write(calcstr)
        errstr = "eMtx[" + str(j) + "] = fnGetError(oNet, 0.0);" + "\n"
        f.write(errstr) # Append healthy function call to file
        j += 1

        p = str(round(uniform(-0.09, -0.02), 2))          # Unhealthy <ecg wave> range
        q = str(round(uniform(-0.26, -0.14), 2))          # ...
        r = str(round(uniform(2.19, 2.42), 2))            # ...
        s = str(round(uniform(-0.2, -0.3), 2))            # ...
        t = str(round(uniform(0.37, 0.55), 2))            # ...
        o = str(1.00)    # Expected output

        # Generate C function calls
        calcstr = "oNet = fnCalculateNetwork(wMtx, " + str(p) + ", " + str(q) + \
                ", " + str(r) + ", " + str(s) + ", " + str(t) + ");" + "\n"
        f.write(calcstr)
        errstr = "eMtx[" + str(j) + "] = fnGetError(oNet, 1.0);" + "\n"
        f.write(errstr) # Append unhealthy function call to file
        j += 1

f.close() # Close file stream
```