San Jose State University
SJSU ScholarWorks

Master's Theses | Master's Theses and Graduate Research

2007

# Neural networks and differential equations

Kathleen J. Freitag
*San Jose State University*

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Freitag, Kathleen J., "Neural networks and differential equations" (2007). *Master's Theses*. 3539.
DOI: https://doi.org/10.31979/etd.h2n8-mb9r
https://scholarworks.sjsu.edu/etd_theses/3539

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

# NEURAL NETWORKS AND DIFFERENTIAL EQUATIONS

A Thesis

Presented to

The Faculty of the Department of Mathematics

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Kathleen J. Freitag

December 2007

UMI Number: 1452051

Copyright 2007 by
Freitag, Kathleen J.

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 1452051
Copyright 2008 by ProQuest LLC.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

© 2007

Kathleen J. Freitag

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF MATHEMATICS

Dr. Roger Dodd

Dr. Leslie Foster

Dr. Mohammad Saleem

APPROVED FOR THE UNIVERSITY

11/21/07

# ABSTRACT

## NEURAL NETWORKS AND DIFFERENTIAL EQUATIONS

by Kathleen J. Freitag

This thesis investigates the use of neural networks for approximating solutions to differential equations and compares the method to existing finite element methods. The thesis begins with a discussion about traditional finite element methods. A detailed description of neural networks is then presented. In particular, the architecture of neural networks and how data is processed by the network is explained. The parameters associated with a neural network and methods for training the neural network are discussed. Also, the topic of error minimization and the associated challenges are addressed.

The main topic of the thesis is a discussion of differential equations in the context of neural networks. The thesis describes an implementation of numerical approximation via neural networks which involves a non-traditional minimization technique. The thesis concludes with examples of the neural network method applied to specific differential equations and a discussion of future research in this area.

# CONTENTS

CHAPTER


1 INTRODUCTION 1

2 FINITE ELEMENT METHODS 3

2.1 Galerkin Method . . . . . . . . . . 4

2.1.1 Triangulation . . . . . . . . . . 5

2.1.2 Assembly . . . . . . . . . . 8

2.1.3 Error Estimation . . . . . . . . . . 10

2.2 Collocation Method . . . . . . . . . . 11

2.2.1 Mesh . . . . . . . . . . 12

2.2.2 Assembly . . . . . . . . . . 12

2.2.3 Error Estimation . . . . . . . . . . 14

3 NEURAL NETWORKS 16

3.1 History . . . . . . . . . . 16

3.2 Architecture . . . . . . . . . . 21

3.2.1 Transfer Functions . . . . . . . . . . 24

3.3 Training Networks . . . . . . . . . . 28

3.4 Minimization Techniques . . . . . . . . . . 30

3.4.1 Backpropagation . . . . . . . . . . 30

3.4.2 Steepest Descent Method . . . . . . . . . . 32
3.4.3 BFGS Method . . . . . . . . . . 33
3.4.4 Radial Basis Function Methods . . . . . . . . . . 33
3.5 Overlearning and Generalization . . . . . . . . . . 35

**4** DIFFERENTIAL EQUATIONS AND NEURAL NETWORKS 37
4.1 Mesh . . . . . . . . . . 38
4.2 ANN Assembly . . . . . . . . . . 39
4.2.1 The Minimization Problem . . . . . . . . . . 39
4.3 Error Estimation . . . . . . . . . . 45
4.3.1 Neural Networks and Function Approximation . . . . . . . . . . 46

**5** IMPLEMENTATION 49
5.1 Neural Net Software . . . . . . . . . . 49
5.1.1 ANNADES . . . . . . . . . . 50
5.2 Example Problems . . . . . . . . . . 51
5.2.1 Ordinary Differential Equation, Example 1 . . . . . . . . . . 53
5.2.2 Ordinary Differential Equation, Example 2 . . . . . . . . . . 56

**6** CONCLUSION 61

**BIBLIOGRAPHY** 62

## TABLES

**Table**

2.1 Finite Element Method Error Norms . . . . . . . . . . . . . . . . . . . . 11

3.1 ANN Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

## FIGURES

**Figure**


2.1 Hat Function, equation (2.7) . . . . . . . . 6

2.2 Bilinear Example, equation (2.8) . . . . . . . . 7

3.1 Perceptron Structure . . . . . . . . 18

3.2 XOR Problem . . . . . . . . 19

3.3 Connected Graph with Weighted Edges . . . . . . . . 21

3.4 Feed Forward ANN; the input nodes are on the left . . . . . . . . 22

3.5 Recurrent ANN . . . . . . . . 23

3.6 Neural Net Data Flow . . . . . . . . 24

3.7 Logistic Curve, $m = 0, a = n = \tau = 1$ . . . . . . . . 26

3.8 Radial Basis Curve, $\rho(x, 0) = g(x, 1) = \exp(-x^2)$ . . . . . . . . 27

4.1 Representative ANN for a PDE . . . . . . . . 40

5.1 ANNADES Main Function Flow . . . . . . . . 51

5.2 ANNADES Optimization Function Flow . . . . . . . . 52

5.3 Absolute error approximation for problem 5.1, $h = 0.1$ . . . . . . . . 54

5.4 Absolute error approximations for problem 5.1, $h < 0.1$ . . . . . . . . 55

5.5 Absolute error approximation for problem 5.1, $N_H = 15$ . . . . . . . . 57

5.6 Absolute error approximation for problem 5.1, using $\tilde{\Psi}_t$ . . . . . . . . 58

5.7 Absolute error approximation for problem 5.5 with a mesh size = 0.1 . . . 60

# CHAPTER 1

# INTRODUCTION

The study of differential equations is a broad field centered around rates of change in a physical behavior. Physicists, engineers, etc. define physical behaviors of the world using differential equations. Mathematicians study the existence and uniqueness of solutions to differential equations and discover analytical and numerical techniques for solving them. The goal of solving a differential equation is to find the closest, if not exact, solution given some restrictions on the behavior. The majority of real world problems are not likely to have a closed form analytical solution. Therefore, algorithms for approximating solutions are required and each differential equation has characteristics that lend itself to different numerical approximation methods. The goal of this thesis is to find continuity in the finite element methods of numerically approximating solutions and introduce a method of similar nature using neural networks.

Chapter 2 will center around the traditional finite element methods summarizing the process of finding a solution, the parameters involved and the method for determining its viability through error estimation. These methods were chosen as a basis of comparison due to their similarity to the neural network approximation method. The traditional methods discussed are the *collocation method* and the *Galerkin method.* These methods share the concepts of defining the domain on which to approximate the differential equation, the assembly of an approximate solution and estimating the error between the approximated solution and the exact solution.

Chapter 3 discusses neural networks in depth. The discussion will begin with a history of the subject, the traditional uses of neural networks, the architecture and parameters of a neural network, minimization techniques, and the potential challenges in neural network training.

Chapter 4 examines the application of neural networks for solving differential equations and the similarities to the traditional methods presented previously.

Chapter 5 addresses available neural network software, the application developed for this study, and the implementation of particular example problems.

The thesis concludes with the findings and results of the comparison including directions for further research.

# CHAPTER 2

# FINITE ELEMENT METHODS

Before introducing a new method for approximating solutions to differential equations it is important to provide a foundation for comparison with existing robust methods. The finite element methods provide the best analogy to the the Artificial Neural Network Approximation Method. More specifically a summary of the Galerkin and collocation methods are presented for their similarities. Both methods are described using an example partial differential equation with Dirichlet boundary conditions as a model problem. Good references for the extension of the methods on other types of differential equations are given in [Sew88], [GL88] and [Hug00].

First the partial differential equation is given. For domain $P \subset \mathbf{R}^2$, $f : P \to P$, and a non-constant coefficient, $\kappa$

$$\begin{aligned} -\nabla \cdot (\kappa \nabla \Psi) &= f \\ \Psi &= 0 \text{ on the boundary of } P. \end{aligned} \tag{2.1}$$

The goal of finite element methods is to approximate $\Psi$ using a projection of the solution onto an approximating subspace of finite dimensions. Given, $\{\phi_i\}_1^\infty$ forms a basis for the entire solution space and the finite element approximating space, $S_n = \{\phi_i\}_1^n$, the approximation is

$$\Psi_n(x) = \sum_{i=1}^{n} U_i \phi_i. \tag{2.2}$$

For both the Galerkin and the collocation method, the approximation (2.2) has

a unique representation when a number of conditions are met. An outline of these conditions are given for the Galerkin method next.

## 2.1 Galerkin Method

In order to find an approximation to the Dirichlet problem (2.1), the following definitions are needed.

**Definition 2.1.1.** A set of functions $S = \{\phi_i(\mathbf{x})\}_{i=1}^{\infty} \subset H^1(P)$ is said to be *complete* if these functions, $\phi_i(\mathbf{x})$ are linearly independent and any function, $\Psi \in H^1(P)$ can be written as

$$\Psi = \sum_i U_i \phi_i. \tag{2.3}$$

$S$ is also said to be a *completion* of $P$.

**Definition 2.1.2.** The *Hilbert space* $H^1(P)$, $P \subset \mathbf{R}^n$ is the space of complex functions $\Psi, \Phi \in H^1(P)$ with *inner product*

$$\langle \Psi, \Phi \rangle = \int_P \left( \Psi \bar{\Phi} + \nabla \Psi \bar{\nabla} \Phi \right) d\mathbf{x}. \tag{2.4}$$

It is a complete, normed linear space, i.e. *Banach Space* with respect to the associated *norm*

$$\|\Psi\|^2 = \langle \Psi, \Psi \rangle. \tag{2.5}$$

Now with the above definitions, given a finite element approximation space $S_n \subset H^1$, for every $\Psi \in H^1$ and $\Psi_n \in S_n$

$$\|\Psi - \Psi_n\| \to 0 \text{ as } n \to \infty. \tag{2.6}$$

More simply the finite element approximation approaches the exact solution as the subspace $S_n$ nears a completion of $H^1$. With the conditions on the functions $\phi$ and the space $P$ met, the coefficients, $U_i$ can be determined by a linear system of equations. The core steps in this approximating algorithm are,

- *triangulation*, partitioning the domain, $P$,
- *assembly*, forming the approximation function, $\Psi_n$,
- *error estimation*, calculating the distance $||\Psi - \Psi_n||$, between the exact solution and the approximation.

The next three sections outline these steps in some detail.

### 2.1.1 Triangulation

The triangulation of the domain into $n$ *elements* involves two steps. First, the shape and size of the elements or *mesh* are defined. For example, in a one dimensional problem, the elements are $n$ intervals that can be equal in length or have unequal lengths. For our 1-D problems we used an equally spaced mesh for the domain.

For the 2-D problems, the domain must be divided into a mesh of polygonal shaped elements. These elements must adhere to size restrictions, meaning they can't be too "skinny." They can be any polygon with straight or curved edges. Curved edges require a special application of finite elements called isoparametrics. For illustration purposes, we will be using a quadrilateral element with equal straight edges. These elements work well on the non-curved boundaries.

Next, a function is defined for each interval/element. This function is called the *element basis function* and tends to be a piecewise polynomial. These polynomials are the simplest to integrate and fit over a given domain. A basis function for an element has the property that it is zero at every vertex of the element except one where it has the value one.

For the 1-D problems, the *hat* function satisfies these properties by combining two

element basis functions. On an interval, $I = [x_i, x_{i+1}]$ the hat function

$$\phi(x_i) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}} & x \in [x_{i-1}, x_i], \\ \frac{-x + x_i}{x_{i+1} - x_i} & x \in [x_i, x_{i+1}], \\ 0 & x = x_{i-1}, x = x_{i+1}. \end{cases} \tag{2.7}$$

This piecewise linear function is illustrated by the example, $x_i = i$ in figure 2.1 for $i = 1 \ldots 6$.

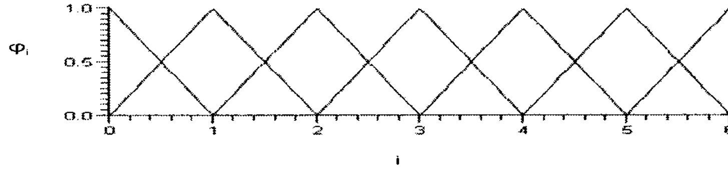


Figure 2.1: Hat Function, equation (2.7)

Thus, each interval, $I$, is defined by both $\phi_i(x) = a_1 + a_2 x$ and $\phi_{i+1}(x) = a_1 + a_2 x$. The $a_i$'s are then determined by the endpoints of interval $I$ and the set of the $\phi_i$'s form the finite element basis for approximating $\Psi : P \to P$. The approximation is defined as, $\Psi_n = \sum_{i=1}^{n} U_i \phi_i$ where $n$ is the finite number of $x_i \in P$ and the $U_i$ are yet to be determined.

For the 2-D problems a similar approach can be taken for defining a basis. In 2-D the *hats* become *tents*. To demonstrate the tent functions, a bilinear function can be assigned to each element. The bilinear functions are defined as $\phi_{jk}(x, y) = a_1 + a_2 x + a_3 y + a_4 xy$. The coefficients are determined by using the four corner points of the quadrilateral element and the general finite element basis properties of $\phi$ must hold.

**Example 2.1.3.** Consider $P = [0, 1]^2$, and split $P$ into four equal sized quadrilateral

elements. This results in nine vertices, $n = 9$. Define a tent function, $\phi_{jk}(x, y) = a_1 + a_2x + a_3y + a_4xy$. The following equation satisfies $\phi_{jk}\left(\frac{1}{2}, \frac{1}{2}\right) = 1$ when $x_j = \frac{1}{2}$ and $y_k = \frac{1}{2}$. Notice the equations agree at the edges of the quadrilaterals. This example is illustrated in figure 2.2.

$$v(x,y) = \begin{cases} 4xy & x, y \in \left[0, \frac{1}{2}\right]^2, \\ 4x(1-y) & x, y \in \left[0, \frac{1}{2}\right] \times \left[\frac{1}{2}..1\right], \\ 4(1-x)y & x, y \in \left[\frac{1}{2}, 1\right] \times \left[0..\frac{1}{2}\right], \\ 4(x-1)(y-1) & x, y \in \left[\frac{1}{2}, 1\right]^2. \end{cases} \tag{2.8}$$

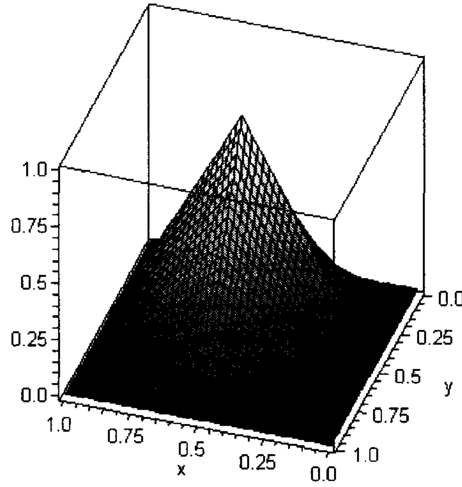


Figure 2.2: Bilinear Example, equation (2.8)

This construction of the $\phi_{jk}$ for each node $(x_j, y_k)$ results in the finite element basis defined on $P$ and an approximation $\Psi_n$. Each of the four elements produce four $\phi_{jk}$ but due to equality at the element boundaries the sixteen $\phi_{jk}$'s reduce to only nine $\phi_i$'s. The approximation $\Psi_9 = \sum_{i=1}^{9} U_i\phi_i$ is the finite element approximation of $\Psi$.

### 2.1.2 Assembly

The term assembly is used rather loosely in reference to the finite element method. One interpretation is that forming the approximation to the solution of a differential equation requires assembling a matrix of values called the *stiffness* matrix and a vector of values called the *load* vector. Determining the coefficients $U_i$ is then done by solving the system of equations. The representation of the differential equation in (2.1) is called the *strong* form of the equation. In order to determine the stiffness matrix and the load vector, the *weak* or *variational* form of the differential equation must be derived from the strong form.

Recall that the set of functions, $\phi$ and the domain $P$ had to satisfy some constraints. Additionally, for each element in the domain with $n_v$ vertices, a function $v$ can be defined as

$$v = \sum_{k=1}^{n_v} a_k \phi i. \tag{2.9}$$

This *variational* function, $v \in H^1$ is multiplied on both sides of the equation. The two sides are integrated over the domain using integration by parts and a variational form is derived. The next example illustrates this.

**Example 2.1.4** (Weak Form of equation (2.1)). Recall, the Dirichlet equation from the beginning of the chapter. For domain $P \subset \mathbf{R}^2$, $f : P \to P$, and a non-constant coefficient, $\kappa$

$$\begin{aligned} -\nabla \cdot (\kappa \nabla \Psi) &= f \\ \Psi &= 0 \text{ on the boundary of } P. \end{aligned}$$

Beginning with the above strong form of the differential equation, multiply both sides by $v(x) \in H_0^1$ and integrate the equation over the domain, $P$.

$$-\int_P \nabla (\kappa \nabla \Psi) v = \int_P f v.$$

Using integration by parts, the divergence theorem from Calculus and the property that $v = 0$ on the boundary of $P$, i.e. $v|_{\partial P} = 0$ the weak form is derived.

$$\begin{aligned} -\int_P \nabla\left(\kappa\nabla\Psi\right)v &= \int_P fv \\ \int_P \kappa\nabla\Psi\nabla v - \int_{\partial P} \kappa v\frac{\partial\Psi}{\partial n} &= \int_P fv \\ \int_P \kappa\nabla\Psi\nabla v &= \int_P fv \end{aligned} \tag{2.10}$$

The integrals are then numerically approximated using a summation of estimates over the number of elements, $n_e$ and the number of nodes in the domain, $n$. Each element contains $n_v$ nodes which can be shared by other elements. This approximation is called a *quadrature rule* and is shown below.

$$\begin{aligned} \int_P \kappa\nabla v\nabla\Psi &= \sum_{i=1}^{n_e} \kappa\nabla v\nabla\Psi_i \\ &= \sum_{i=1}^{n_e}\sum_{j=1}^{n_v} \left(w_{ij}\kappa\nabla v\left(\mathbf{x}_j\right)\right)\nabla\Psi_i \end{aligned} \tag{2.11}$$

The entries of the stiffness matrix, $\mathbf{K}$, can then be determined by setting $k_{ij} = w_{ij}\kappa\nabla v\left(\mathbf{x}_j\right)$. Then the load vector, $\mathbf{L}$ is approximated using the same quadrature rule and the entries are $l_i = w_i f\left(\mathbf{x}_i\right)$. The result is a $n \times n$ matrix equation of the form, $\mathbf{K}\cdot\mathbf{U} = \mathbf{L}$. See equation (2.12).

$$\begin{bmatrix} k_{11} & . & . & k_{1j} & . & k_{1n} \\ . & k_{22} & & & & . \\ . & & . & & & . \\ k_{i1} & & & k_{ij} & & . \\ . & & & & . & \\ k_{n1} & . & . & . & . & k_{nn} \end{bmatrix} \begin{bmatrix} U_1 \\ . \\ . \\ U_i \\ . \\ U_n \end{bmatrix} = \begin{bmatrix} l_1 \\ . \\ . \\ l_i \\ . \\ l_n \end{bmatrix}. \tag{2.12}$$

The stiffness matrix contains a *band* of values along the diagonal of size, $n_v$. Once this matrix and vector are created the task is then to find the coefficients, $U_i$ for each basis function. This is accomplished by inverting the stiffness matrix and multiplying it by the load vector. Once the coefficients are found the approximation of the solution is created by summing the coefficients found multiplied by the basis function over all the intervals or elements of the domain.

There are many details left out of this discussion of the finite element method for approximating a solution to a differential equation. The goal is to provide the reader with a general idea of the steps associated with constructing the approximation and noting some key features that will be referred to when discussing the neural network method in Chapter 3.

The next section will illustrate the value of this approximation by explaining the different types of errors that can be calculated.

### 2.1.3 Error Estimation

The space on which the above approximation is defined is by definition in the same space or subspace as the exact solution of the differential equation. This space has a number of different ways to define how close to elements of the space are to each other. This is known as a *norm* of the space and is represented by $\|\cdot\|$. A subscript also

accompanies the norm notation defining the space that the norm is operating in. There are four norms to be considered.

Table 2.1: Finite Element Method Error Norms

| **Name** | **Representation** | **Definition** | **Space** |
|---|---|---|---|
| Sup Norm | $\|\|\Psi\|\|_{\infty}$ | $max_{x\in\Omega}\{\|\Psi(x)\|\}$ | $l_2$ |
| $L^2 Norm$ | $\|\|\Psi\|\|_{L^2}$ | $\left(\int_{\Omega}\Psi^2\right)^{\frac{1}{2}}$ | $L_2$ |
| $H^1$ Norm | $\|\|\Psi\|\|_{H^1}$ | $\left(\int_{\Omega}\Psi^2+\nabla\Psi^2\right)^{\frac{1}{2}}$ | $H^1$ |
| Energy Norm | $\|\|\Psi\|\|_E$ | $\sqrt{a(\cdot,\cdot)}$ | $L_2$ and $H^1$ |

The value $a(\cdot,\cdot)$ are the $k_{ij}$'s in the stiffness matrix. Also, the norm is not limited to the $L_2$ and $H^1$ spaces. In general, the actual error of the approximation is not available. Thus, only a sense of the size of the error or *order* of the error can be determined. For finite element methods the order is often given in relation to the size of the mesh. Knowing the order of the error also allows different error estimations to be compared with each other. Later it is shown in § 5.2 that comparing the finite element approximation with the new neural network approximation using the mesh size is inconclusive and that the similarity between the two methods is in the algorithm for defining the approximations rather than in the methods for estimating the error of the approximations.

## 2.2 Collocation Method

The Collocation Method provides a better comparison to the neural network method for numerically approximating solutions to differential equations. The method can be considered a special form of the more general Galerkin Method in which a specialized test function is used. With this in mind the basic steps of forming the approximation are

- *mesh* definition, partition the domain, $P$,
- *assembly*, create the approximation function, $\Psi_n$,

- *error estimation*, calculate the distance $\|\Psi - \Psi_n(x)\|$, between the approximation and the exact solution and the approximation.

Recalling the model differential equation, (2.1), the first step is to define a *mesh* on the domain $P$.

### 2.2.1 Mesh

The mesh of evaluation points is a finite number of points in a bounded domain, $\mathbf{X}$. These points are locations where the approximation exactly satisfies the differential equation. The goal is to *collocate* the solution with the approximation for each of these domain values. The collocation points can be chosen from any location but in general they are equally spaced or chosen for minimum error depending on the basis functions. Gaussian points have been shown to generate the best approximation [Sun96] for a particular set of basis functions.

### 2.2.2 Assembly

The basis functions best approximated by the Gaussian points are the Hermite Cubic functions on each interval for the 1-D case or the Hermite Bicubic functions on each rectangle in the domain. There are 4 unknowns in the 1-D case and 16 unknowns in the 2-D case. In the 1-D case, the Hermite basis functions and it's derivatives are evaluated at two collocation points located within the boundary of the interval. In the 2-D case the Hermite basis functions and its derivatives are evaluated at 8 collocation points.

The Hermite cubic basis functions for a domain consisting of $n$ collocation points are $\{S_0, H_1, \ldots, S_{n-1}, H_n, S_n\}$ and are expressed by the following equation.

$$H_k(x) = \begin{cases} 3\left(\frac{x-x_{k-1}}{x_k-x_{k-1}}\right)^2 - 2\left(\frac{x-x_{k-1}}{x_k-x_{k-1}}\right)^3 & x_{k-1} \leq x \leq x_k \\ 3\left(\frac{x_{k+1}-x}{x_{k+1}-x_k}\right)^2 - 2\left(\frac{x_{k+1}-x}{x_{k+1}-x_k}\right)^3 & x_k \leq x \leq x_{k+1} \\ 0 & elsewhere \end{cases} \tag{2.13}$$

$$S_k(x) = \begin{cases} -\frac{(x-x_{k-1})^2}{(x_k-x_{k-1})} - \frac{(x-x_{k-1})^3}{(x_k-x_{k-1})^2} & x_{k-1} \leq x \leq x_k \\ \frac{(xk+1-x)^2}{(x_{k+1}-x_k)} + \frac{(xk+1-x)^3}{(x_{k+1}-x_k)^2} & x_k \leq x \leq x_{k+1} \\ 0 & elsewhere \end{cases} \tag{2.14}$$

For a two dimensional domain consisting of $n \times n$ Gaussian points, the Hermite bicubic functions are used and are defined as the product of two Hermite cubic functions.

As in the Galerkin method, the strong form is multiplied by a test function In this case the Dirac Delta function is the special test function. However, when the product of the Dirac Delta function and the strong form are integrated over the domain the outcome of this procedure results in something different than in the Galerkin method. The Dirac Delta or unit impulse functions [SN96] are often considered an operator function since it has the property of determining a given function at a specific value. These functions are represented using the following definition.

If $\psi(x)$ is a continuous function on $[a, b]$, $x_0 \in [a, b]$ then the inner product of $\psi(x)$ with the Dirac Delta function, $\delta(x - x_0)$ is,

$$\int_a^b \Psi_i(x)\,\delta(x - x_0)\,dx = \Psi_i(x_0). \tag{2.15}$$

The operation of the Dirac Delta function on the strong from results in a simple evaluation of $\Psi_i$ at each Gaussian point in the domain where

$$\Psi_i = H_i(x_i)\,U_i + S_i(x_i)\,U_i'. \tag{2.16}$$

The big assumption is that the exact solution, $\Psi$ must be continuous on the domain, $P$ for the impulse function to apply over the entire domain. However, since the method is only concerned about a finite set of points, the assumption doesn't apply.

Next, the matrix equation $\mathbf{KU} = \mathbf{L}$ is formed. The vector

$$\mathbf{U} = \begin{bmatrix} U'_0 \\ U_1 \\ U'_1 \\ \cdot \\ \cdot \\ U_i \\ U'_i. \\ U_n \\ U'_n \end{bmatrix}.$$

The entries of $\mathbf{K}$ are computed by the $H_i$ and $S_i$ for each Gaussian vertex, $x_i \in P$. Second, the entries of the load vector $\mathbf{L}$ are generated using the values of $f(x_i)$. The coefficients, $U_i$ are then found by inverting the stiffness matrix and multiplying it with the load vector. The product of the coefficients with the basis functions summed over the intervals then combine to form an approximate solution to the differential equation. The resulting formulation of the approximation is,

$$\Psi_n(x) = \Psi(x_0) + \sum_{i=1}^{n} U_i H_i(x) + \sum_{i=0}^{n} U'_i S_i(x) \tag{2.17}$$

where $\Psi(x_0)$ defines the boundary condition [Sew88].

### 2.2.3 Error Estimation

Due to the discrete nature of the collocation method, most of the norms defined for the Galerkin Method are not applicable. All but one of those norms are reserved for continuous approximations in their respective spaces. For the collocation method, the Sup Norm coupled with the Gaussian collocation points delivers the error estimation [Sew88]. The discrete formulation of the error estimation is also applicable for estimating the error in the new neural network method, see § 4.3.

Next, the concept of using neural networks for numerically solving differential equations is introduced with the objective of illustrating its similarities to the traditional approximations and eventually providing it with a place on the menu of desired numerical solutions.

# CHAPTER 3

# NEURAL NETWORKS

An artificial neural network (ANN) is a computer simulation of the human brain's problem solving processes. Their application to solve various problems is vast [Sam07] [FVF04] [Bha99]. Before introducing the application of neural networks for numerically approximating differential equation solutions, a history of neural networks and a discussion of their traditional uses is presented.

## 3.1 History

ANNs began with the collaboration of Warren McCulloch and Walter Pitts in 1943. McCulloch, a neurophysiologist, and Pitts, a mathematician, were both interested in the mechanics of learning in the human brain. Their initial work resulted in a hardwired circuit board that emulated the function of neural processes [AM92]. Shortly thereafter at the IBM Research Lab located at Columbia University in New York, Nathanial Rochester initiated work on the first neural network software application [RG58]. Though his efforts produced a simple implementation, the limitations of the computers meant that the work wasn't pursued.

In 1956 the Dartmouth Summer Research Program's topic was centered around machine learning. The term artificial intelligence was coined at this conference and neural networks were at the core of the artificial intelligence discussions [JMS55]. From this program, many scientists began research in this area, including Frank Rosenblatt.

His contribution to the neural network community was the creation of the Perceptron, a computer built to model neural net processing. It consisted of two layers of nodes, an input layer for receiving $N_I$ signals and an output layer consisting of a single node that delivered the output signal. Mathematically, the perceptron is defined below.

**Definition 3.1.1.** Given $\mathbf{w}, \mathbf{x} \in \mathbf{R}^n$, and *threshold* $u \in \mathbf{R}$, the *perceptron* $\phi : \mathbf{R}^n \to \mathbf{R}$ is defined by

$$\phi(\mathbf{x}) = \sum_{i=1}^{N_I} w_i x_i - u. \tag{3.1}$$

The single output of the perceptron is determined by a classification function $\Psi : \mathbf{R} \to \mathbf{R}$

$$\Psi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The perceptron is a linear function and an affine subspace of $\mathbf{R}^n$ [AB99]. One of the outcomes of Rosenblatt's perceptron development was the reinforcement of parallel processing for computer hardware [Bri07]. The perceptron was the precursor to the ANN and the ANN's architecture reflects the perceptron's parallel processing structure. This structure can be viewed in illustration 3.1.

At the close of the 1950's, Bernard Widrow and Marcian Hoff at Stanford created the first neural nets applied to the real world problem of eliminating echo in phone lines. The implementations were named ADALINE (Adaptive Linear Elements) and MADALINE (Multiple Adaptive Linear Elements) both of which are still in use today [AM92].

During this time one of the challenges that surfaced with neural nets was the inability to handle the Boolean function XOR. To explain the problem we first define OR, AND, and XOR.

**Definition 3.1.2.** Given $x_1, x_2 \in \{0, 1\}$.

$$\mathrm{OR}(x_1, x_2) = \begin{cases} 0 & \text{if } x_1 = x_2 = 0 \\ 1 & \text{otherwise} \end{cases}$$

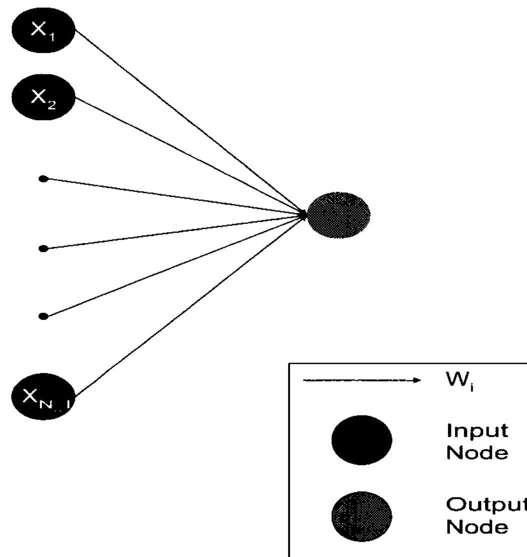


Figure 3.1: Perceptron Structure

**Definition 3.1.3.** Given $x_1, x_2 \in \{0,1\}$.

$$\text{AND}\,(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 = x_2 = 1 \\ 0 & \text{otherwise} \end{cases}$$

**Definition 3.1.4.** Given $x_1, x_2 \in \{0,1\}$.

$$\text{XOR}\,(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 = 1, x_2 = 0 \\ 1 & \text{if } x_1 = 0, x_2 = 1 \\ 0 & \text{otherwise} \end{cases}$$

The real issue was that the perceptron could not linearly separate what was a combination of two functions. Recall that the perceptron is represented by a single linear function where the separation of domain $\mathbf{R}^n$ is the hyperplane defined by $\sum_i w_i\,(x_i) - u = 0$. Figure 3.2 illustrates how a single line, $l = w_1 x_1 - u$ can separate the data in the AND and OR functions in $\mathbf{R}^2$ but how the XOR solution needs two independent lines to separate the solutions, $l_1 = w_1 x_1 - u$ and $l_2 = w_2 x_2 - u$. The pair of lines is impossible to represent in $\mathbf{R}^2$ using the perceptron as defined.

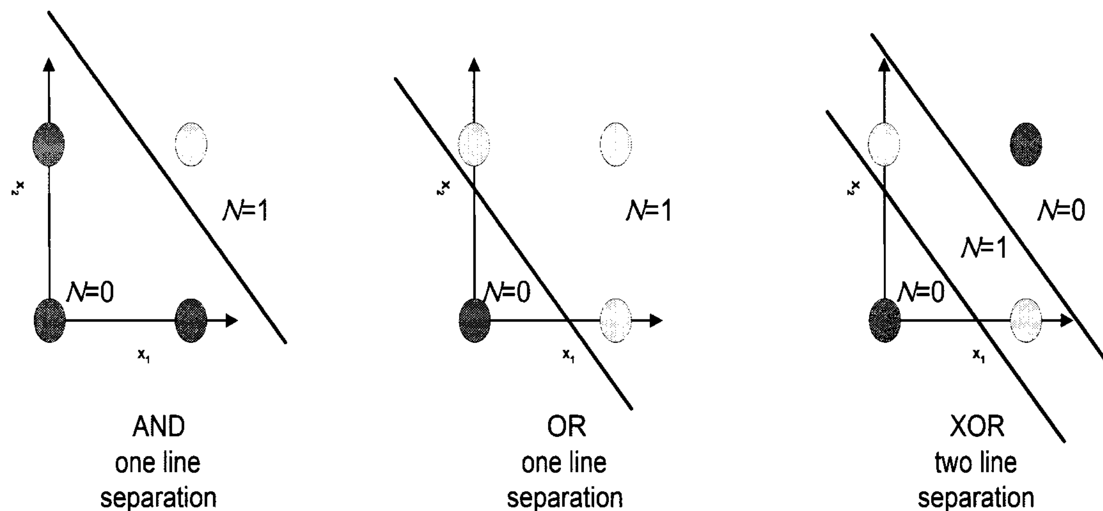


Figure 3.2: XOR Problem

This roadblock halted the evolution of neural networks until Marvin Minsky and Seymour Papert's book Perceptrons, [MP87], published in 1969. A solution to the XOR problem was presented. The solution was to incorporate another layer, a *hidden layer*, in the network which is described later in § 3.2.

Unfortunately, Minsky and Papert's solution arrived just at the time panic was instilled in the public by literary works. These works included Arthur C. Clarke's textit2001:A Space Odyssey with the self motivated learning machine, HAL and the self organizing robots of Isaac Asimov's science fiction novels [AM92]. The result was a general halt in U.S. government funding for any projects related to artificial intelligence, including neural nets. However, artificial intelligence research continued elsewhere in the world especially in Japan.

The artificial learning community finally found its break in 1982 when the charismatic John Hopfield from Caltech presented a paper and spoke in favor of further development in artificial intelligence [Hop82]. That same year, the Japan conference on Cooperative/Competitive Neural Networks also announced it's 5th generation implementation of a neural network . This fueled the competitive nature of the US government and new

funding began to flow.

Recent developments in computer hardware have enabled the construction of massive parallel computers at a reasonable cost. This has enabled implementations of neural networks on parallel processors with thousands of nodes so that machine learning can now more accurately model the processes of the human brain. there have also been software advances. For example in 1996, Dimitri Bertsekas and John Tsitsiklis introduced a marriage of dynamic programming and neural networks called neuro-dynamic programming (NDP). A result of their work was a machine that learned to play Tetris and Backgammon [BT96]. Another success of the increasing power of artificial intelligence techniques has been the solution to checkers. Checkers was solved by a computer called Chinook [S+07]. Chinook was programmed by a team lead by Dr. Jonathan Schaeffer at University of Alberta. The computer has been *learning* the good and bad plays of numerous checkers games for the last 18 years and optimizing it's search paths along a decision tree toward a solution. The work by Schaeffer's team perpetuates the development of neural networks as a framework for "thinking" computers.

Although the results of neural networks being applied to specific areas of game theory, engineering [GZ07], biomechanics [Hah07] and finance forecasting [HT06] has moved forward, the more general use of applying neural nets to solving classes of mathematical problems has had only limited attention. The study of ANNs and their architecture is still in its infancy. Architectures are still being developed and the best way to use ANNs for a wide field of problems is far from being understood.

As the applications of neural networks evolve, the need for the development of better techniques in other mathematical areas, such as function approximation, emerges. More specifically, methods for solving differential equations using ANNs requires better techniques for minimization and a larger class of activation functions. The first application of ANNs to solving differential equations appears to be in an article by Isaac Lagaris

and Aristadis Likas from the University of Ionnina, Greece in 1997. In order to understand their work, the architecture, operation, and challenges of using neural networks is first presented.

## 3.2 Architecture

An artificial neural net is a collection of perceptrons or nodes that are connected via edges to form a network. The edges of the neural net are assigned weights which modify the data. This is analogous to a connected graph with weighted edges and directional data flow, see figure 3.3.

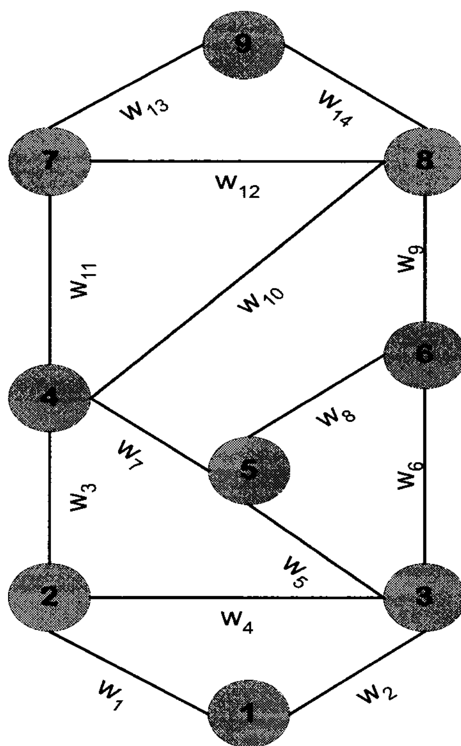


Figure 3.3: Connected Graph with Weighted Edges

A graph $G$ of this form can be represented by a set of $N$ nodes or vertices $V = \{n_i\}_1^N$, and edges $E \subset V \times V$. If the graph is directed, then $(n_i, n_j) \in E$ is distinct from $(n_j, n_i)$ when both edges are in $E$. We shall exclude from consideration edges which

connect to the same node, that is $(n_i, n_i) \notin E$ for all $1 \leq i \leq N$. The weights assigned to the edges are defined by a function $w : E \to \mathbf{R}$. We shall denote the value of a weight by $w_{ij}$.

The topology of a weighted directed graph $G = (V, E; w)$ is determined by the set of edges $E$. A complete weighted directed graph for example is one in which every node is connected to every other node in the graph.

One difference between the connected graph and the neural net is that the nodes are grouped in layers. There is an input layer, output layer, and any number of hidden layers. Data flows from the input nodes through the hidden layer nodes culminating in the output nodes. Typically, each layer of nodes is connected to the immediately following layer but not any layers beyond that. Figure 3.4 shows the difference.

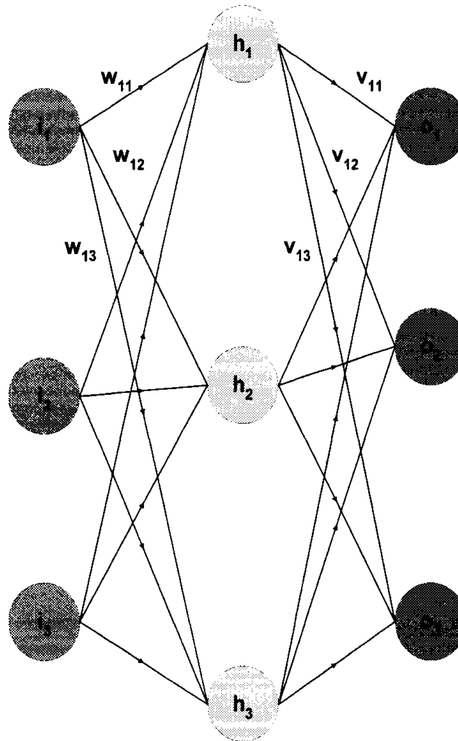


Figure 3.4: Feed Forward ANN; the input nodes are on the left

To describe the neural network we let $I = \{\, \mathrm{i}_j\}_1^{N_I}$ denote the input nodes, $O =$

$\{ o_j \}_1^{N_O}$ denote the output nodes and $H = \{ h_j \}_1^{N_H}$ denote a layer of hidden nodes. For simplicity we shall restrict the discussion to a single hidden layer, as shown in Figure 3.4. It is easy though to generalize to the case of several hidden layers. Let $E_{IH} \subset I \times H$ denote the directed edges which connect the input nodes to the layer of hidden nodes, and similarly $E_{HO} \subset H \times O$ represents the directed edges connecting the hidden layer to the output nodes. The graph for this case is given by $G = (I, H, O, E_{IH}, E_{HO}; w)$.

If backward flow of data exists the network is called a recurrent neural network. In recurrent networks data is shared with other nodes in feedback edges so for example there may be additional weighted edges which connect the output nodes and hidden nodes to the input nodes. In this case there are additional edges, $E_{OH} \subset O \times H$, and $E_{HI} \subset H \times I$. The corresponding graph is $G = (I, H, O, E_{IH}, E_{HO}, E_{OI}, E_{HI}; w)$. Figure 3.5 illustrates diagrammatically an example of this type of recurrent neural network.

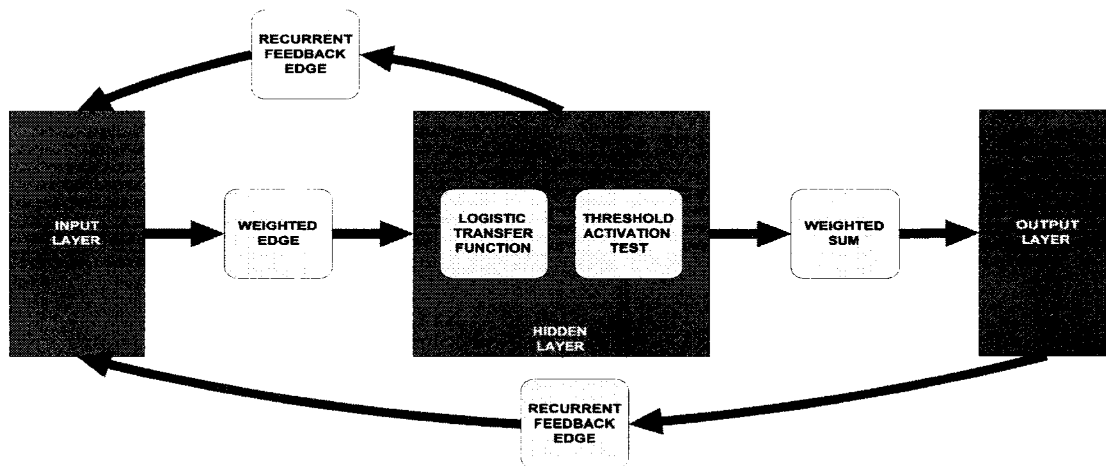


Figure 3.5: Recurrent ANN

Networks with non-recurrent directed data flow are called feed-forward networks and are the simpler of the two types to model. In a basic model, the ANN would have a few input data nodes, a single hidden layer of nodes, and a single output node. In contrast to the data flow figure depicting a neural net allowing feedback loops figure 3.6

shows a feed-forward network.

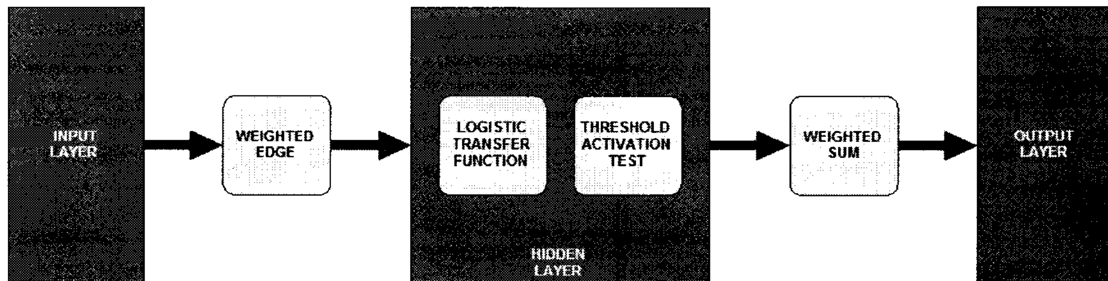


Figure 3.6: Neural Net Data Flow

The flow of data begins at the input nodes, is modified by the weighted edges to the hidden layer nodes, and is then subject to an activation threshold test. The values that pass the activation test are "squashed" by a logistic transfer function before the weighted sum is passed to the output node. This process is summarized by the following neural network output equation (3.2) and table 3.1 of parameters.

For input $\mathbf{x} \in \mathbf{R}^{N_I}$ the data processed to the $i$th hidden node is $z_i = \sum_{j=1}^{N_I} w_{ij}x_j + u_i$ where $w_{ij} = w(i_j, h_i)$. The output of a neural net, $\mathcal{N}$, is

$$\mathcal{N}(\mathbf{x}) = \sum_{i=1}^{N_h} v_i \sigma(z_i) \tag{3.2}$$

where $v_i = w(h_i, o)$ and $\sigma$ is a function similar to the classification function, (3.1), defined for the perceptron. The functions available for $\sigma$ are known as the transfer functions. The $\sigma$ function lies at the heart of a neural network and modifies the data received by the hidden layers. Some examples are described in detail next.

### 3.2.1 Transfer Functions

Transfer functions correspond to the activation potential of a real neuron. Data received by the neuron through its dendrites is transmitted across the synapse between the axon and a dendrite of a communicating neuron only if the potential along the

Table 3.1: ANN Parameters

| **Parameter** | **Definition** |
|---|---|
| $N_I$ | number of input nodes |
| $N_H$ | number of hidden nodes |
| $\{x_j\}_1^k$ | input data value |
| $w_{ij}$ | weight from input unit $j$ to hidden unit $i$ |
| $u_i$ | bias of hidden unit $i$ |
| $z_i$ | $i$th hidden unit value |
| $v_i$ | weight from the $i$th hidden unit to the output node |
| $\sigma(z_i)$ | sigmoid transfer function of the $i$th hidden unit |

axon exceeds a threshold value and the neuron fires. The function is called either the activation or transfer function. The terms are sometimes used interchangeably, but they have different roles. The activation function is typically a binary decision algorithm based on a threshold value. The most common of these activation functions are step functions. Data which exceeds the threshold value is transmitted onwards, otherwise it is dropped. A transfer function on the other hand modifies the data smoothly by restricting its range of values. It can be viewed as a smoothed out version of an activation function. Its operation is therefore less drastic than an activation function. Recent physiological studies indicate that neurons actually work in this way [AB99].

Typical transfer functions are members of the logistic family of functions or *squashing* functions. These functions limit the range of values with which the net uses [Kin95]. The general form of which is,

$$p(t) = a\frac{1 + m\exp^{-\frac{t}{\tau}}}{1 + n\exp^{-\frac{t}{\tau}}} \tag{3.3}$$

where $t, a, m, \tau, n \in \mathbf{R}$

An example of this function can be viewed in figure 3.7.

The transfer function is used in the following way. Suppose that the input to the

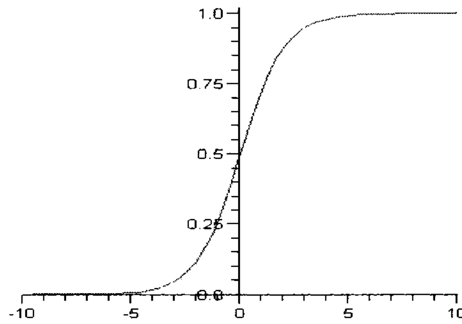


Figure 3.7: Logistic Curve, $m = 0, a = n = \tau = 1$

$i$th node in the hidden layer is

$$z_i = \sum_{j=1}^{N_I} w_{ij} x_j + u_i, \tag{3.4}$$

where $x_j$ is the input from the $j$th input node, $w_{ij}$ the weight associated with the edge ( $i_j$, $h_i$). The number $u_i$ is called the bias of the $i$th hidden node. The bias determines where the transfer function has its steepest descent. If $f$ is the transfer function for the net then the output from the $i$th hidden node will be $f(z_i)$.

Another increasingly popular family of transfer functions are the radial basis functions. This is because they offer the possibility of a direct calculation of the weights $w_{ij}$ and biases $u_i$. Suppose that we have data points $(\mathbf{x}_j, y_j)_1^M$ for a function $f : \mathbf{R}^M \to \mathbf{R}$. Select a radial basis function $g : \mathbf{R}_+ \to \mathbf{R}_+$ ( $\mathbf{R}_+$ is the set of real numbers greater or equal to 0). The Gaussian function is a commonly used function for this purpose. It has the additional properties that $g(0)$ is the maximum of $g$ and the function rapidly decays to 0. An example of a Gaussian radial basis function can be viewed in Figure 3.8. Radial basis functions are discussed further in § 3.4.4.

ANNs can also incorporate pre-processing and post-processing algorithms to modify the incoming and outgoing data, respectively. For example, incoming data could be riddled with known signal noise, the scale of the data could be wrong, or lexicographical

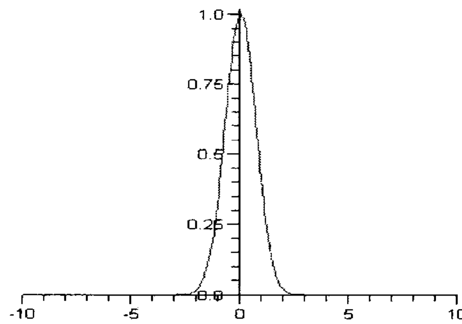


Figure 3.8: Radial Basis Curve, $\rho(x, 0) = g(x, 1) = \exp(-x^2)$

values need translating to numerical values. The pre-processing algorithms address these situations. Post-processing of neural network output may help in combining individual output data for classification, regression of output data to provide a single continuous output value, and sorting the information for display purposes.

The number of input nodes and the number of output nodes of a neural network is determined by the numerical approximation being conducted by the net. The rest of the components that comprise an artificial neural net, the number of layers, the number of hidden nodes, the connections from one layer to the next, the weights of the edges, the activation functions, and the direction of data flow have generally been chosen by what seems to generate the best solution. The works of Hornik concluded that feedforward networks with multiple layers are universal approximators for any "measurable function to any desired degree" [KHW89]. Thus, a neural net with an input layer, at least one hidden layer, an output layer, sigmoid transfer functions, and feed-forward data flow can approximate any function. This is the foundation on which to build a method for approximating the solutions to differential equations. The next step is to determine how this approximation is performed in a neural net.

## 3.3 Training Networks

Over time the human brain gets better at taking inputs and classifying or solving problems through learning. As in the human brain from which it is modeled, the neural net evaluates it's inputs giving greater weight to the more important features of the input in an attempt to classify a problem and ultimately solve it. This method of learning is modeled in ANNs by training algorithms.

Training neural networks falls into two categories, supervised and unsupervised. Supervised training requires a training data set with *known* output values for the problem to be approximated. The training data set is a list of selected values from the domain of a known function with a known solution. The neural net is then trained by comparing the computed outputs of the net with the known outputs of the problem for any given set of inputs. The method by which this set of input data is selected is a subfield of its own. The ideas that are available have been predominately influenced by the works of Hopfield [Hop82], Hinton [DHAS85], Bertsekas and Tsitsiklis [BT96]. These works address the issues of erroneous data and apply classification or filtering techniques prior to using the data in the main net. For the method of using ANN's to solve differential equations, the output data are unknown with the exception of the equation, initial and boundary data information. For this type of problem, an unsupervised training method is needed.

Unsupervised training methods do not require training data sets with known output values in order to train the ANN. Instead the measure of how well the ANN is approximating a solution is defined using other given information. For the differential equation problem, the initial and the boundary data provide a subset of output values and the differential equations provide information for finding the remaining values.

In either method, the ANN uses an error minimization function with a set of conditions. In supervised training the learning conditions are typically set by the user and

can, for example, either require minimizing the estimated error or specifying the number of times the user wants to train the network. In unsupervised training, the training function is more complicated because the desired output is not known. For solving differential equations, the search for an optimal solution to the differential equation requires finding the derivative of the ANN with respect to all of its parameters. Minimization techniques are applied and the parameters of the net are modified to produce the closest approximation to the solution of the differential equation. This theory is discussed in § 4.2.1

After training, it is important to test the neural net with a new set of inputs. The net's behavior on the new set of test data is the last indicator of whether or not the net is accurately modeling the problem. In the supervised training approach, this data should be an independent set of known inputs and outputs of the problem. The user can choose how closely the network approximates the data or it can train the net until the net produces the exact output i.e. zero error. If there is any interpolation done after the net approximates the exact data then the error of the ANN solution is dependent on the interpolation technique. The supervised ANN model is successful if it can accurately approximate the test data when it has been trained on the training data.

For the unsupervised training approach, testing the net involves selecting a new set of input values (i.e. differential equations with known exact solutions) and computing the resulting cost with the ANN output. If the cost is still within the range of desired error then the ANN approximation is interpolating the differential equation solution well. If the error lies outside the desired range then the neural net has not reached a desired solution approximation. The error calculations are straightforward since different classes of differential equations have unique solutions that can be solved exactly.

## 3.4 Minimization Techniques

As noted before, there are two approaches used by an ANN to improve its problem solving. The methods are either supervised or unsupervised learning. Currently most ANNs are implemented using a supervised learning model. The goal of the training in the supervised model is to minimize the error between the neural net output and the known output of the system over a given domain.

### 3.4.1 Backpropagation

In supervised training methods there is a set of, $N$ known input-output pairs called the *training set*,

$$\mathcal{X} = \left\{ \left( \mathbf{x}_k, f\left(\mathbf{x}\right)_k \right) \right\}_{k=1}^{N}. \tag{3.5}$$

In this case the function $f\left(\mathbf{x}\right) : \mathbf{R}^{N_I} \rightarrow \mathbf{R}$ is unknown but specific values that are known are used to help approximate $f$.

Given this set of input-output pairs defined in (3.5), the ANN parameters, $\mathbf{p}$ are then updated as the error estimate is minimized. The error estimate is

$$E\left(\mathcal{X}, \mathbf{p}\right) = \sum_{k=1}^{N} \left| \mathcal{N}\left(\mathbf{x}_k, \mathbf{p}\right) - f\left(\mathbf{x}_k\right) \right|^2. \tag{3.6}$$

The parameters $\mathbf{p}$ of the net are the weights $w_{ij}, v_i$, and the biases $u_i$, see table 3.1. *Backpropagation* is the process of updating the ANN with new parameters that minimize the above estimated error function. This process is outlined below. For each iteration, $i$

(1) Calculate the neural network outputs, $\mathcal{N}\left(\mathbf{x}_k\right)$ for each input $\mathbf{x}_k$.

(2) Calculate the error using function (3.6).

(3) Calculate the change in the parameter vector, $\Delta\mathbf{p}$ that minimizes $E\left(\mathcal{X}, \mathbf{p}\right)$

(4) Update the ANN parameter vector, $\mathbf{p}$,

(5) Continue iterating until $E(\mathcal{X}, \mathbf{p})$ is less than a user specified tolerance or until a user specified number of iterations has been reached.

When the exact output data is not known, as is the case in solving differential equations, the algorithm for updating the weights is the same but the method of determining the value of the $\Delta\mathbf{p}$ is different. This will be discussed further in § 4.2.1.

There is a wide range of error minimization techniques available for finding the value of $\mathbf{p}$ such that $E(\mathcal{X}, \mathbf{p})$ is minimal. In particular, steepest descent and Broyden-Fletcher-Goldfarb-Shanno (BFGS), methods can be used. Each of these methods relies on the idea that the direction perpendicular to the gradient moves toward a local minimum or maximum [Rar98]. This direction is defined as follows.

**Definition 3.4.1.** Direction $\Delta\mathbf{p}$ is improving for the minimization function, $E(\mathcal{X}, \mathbf{p})$, if $\nabla E(\mathcal{X}, \mathbf{p}) \cdot \Delta\mathbf{p} < 0$.

The gradient $\nabla E(\mathcal{X}, \mathbf{p})$ is found using,

$$\nabla E(\mathcal{X}, \mathbf{p}) = \begin{bmatrix} \frac{\partial E(\mathcal{X},\mathbf{p})}{\partial p_1} \\ \cdot \\ \cdot \\ \frac{\partial E(\mathcal{X},\mathbf{p})}{\partial p_i} \\ \cdot \\ \frac{\partial E(\mathcal{X},\mathbf{p})}{\partial p_m} \end{bmatrix}. \tag{3.7}$$

However, there are limitations when using these methods. First a few more definitions are needed.

**Definition 3.4.2.** A function $E(\mathcal{X}, \mathbf{p})$ is *unimodal* if the straight line direction from every point, $(\mathcal{X}, \mathbf{p})$ to $(\mathcal{X}, \tilde{\mathbf{p}})$ in the domain is an improving direction [Rar98].

**Definition 3.4.3.** A solution, $\mathbf{p}$ is a *global minimum* if it doesn't violate any constraints and $E(\mathcal{X}, \mathbf{p}) < E(\mathcal{X}, \tilde{\mathbf{p}})$ for all $\tilde{\mathbf{p}}$. A solution, $\mathbf{p}$, is a *local minimum* if it doesn't violate

any constraints and $E(\mathcal{X}, \mathbf{p}) < E(\mathcal{X}, \tilde{\mathbf{p}})$ for all $\tilde{\mathbf{p}}$ in a small neighborhood surrounding $\mathbf{p}$ [Rar98].

The limitations encountered by the typical minimization techniques are that if the function $E(\mathcal{X}, \mathbf{p})$ is not *unimodal* then any of the following methods could stumble upon a local minimum instead of a global minimum. The study of finding a global minimum in numerical analysis, optimization techniques, and in neural networks is beyond the scope of this thesis. However, in practice many neural nets have been successful in achieving errors below user specified tolerances. A good reference to investigating this further is [AB99].

### 3.4.2 Steepest Descent Method

The first minimization technique is a local improving search algorithm called the steepest descent method. The model also requires a learning rate $r \in [0, 1]$, which represents how quickly the network *learns*, and a step size $\lambda$, which represents how much of a step the algorithm will allow in an improving direction. At each iteration of the steepest descent algorithm, the error is minimized by using the gradient, $\nabla E(\mathcal{X}, \mathbf{p})$, as the improving direction since $\nabla E \cdot \nabla E > 0$ for a $\nabla E \neq 0$. The actual step taken then is $r\lambda\nabla E$. The higher the learning rate the quicker the net will move toward a minimum error. However, if the rate is too large it could *overstep* a minimum. In theory the steps should be infinitesimal but convergence would never be reached. In practice, the steps tend to be very small and they vary, i.e. get larger when the direction is constant over several iterations and return to the initial size when the direction changes at each iteration. With this in mind, typical learning rates vary with numbers generally below 0.1 in an attempt to balance both criteria.

### 3.4.3 BFGS Method

An improvement to the Steepest Descent Method is to determine the best step size, $\lambda$, to take at each iteration. This is done by incorporating an approximation of the Hessian Matrix. First, we define this matrix.

**Definition 3.4.4.** The *Hessian matrix* is a symmetric matrix of second partial derivatives that describes the changes in slope of a function $E(\mathcal{X}, \mathbf{p})$ in the neighborhood of current values of $\mathbf{p}$ [Rar98]. Given $\mathbf{p} = \{p_1, \ldots, p_n\}$ the matrix is defined as

$$H(\mathcal{X}, \mathbf{p}) = \begin{bmatrix} \frac{\partial^2 E}{\partial p_1^2} & . & . & . & \frac{\partial^2 E}{\partial p_1 \partial p_n} \\ . & . & & & . \\ . & & \frac{\partial^2 E}{\partial p_i \partial p_j} & & . \\ . & & & . & . \\ \frac{\partial^2 E}{\partial p_n \partial p_1} & . & . & . & \frac{\partial^2 E}{\partial p_n^2} \end{bmatrix}. \tag{3.8}$$

The Hessian matrix can be approximated using quasi-Newton methods [Rar98]. At iteration, $i$, this matrix is identified as $D_i$ and called the *deflection matrix.*

As long as this deflection matrix is positive definite and symmetric a new minimizing direction at iteration, $i + 1$ can be computed using the following formula,

$$\Delta \mathbf{p}^{i+1} = -D_i \nabla E\left(\mathcal{X}, \mathbf{p}^i\right). \tag{3.9}$$

The benefit of using the deflection matrix is that it doesn't require inverting an $n \times n$ matrix for computing the improving direction. This is the method adopted for most neural net applications due to its superior success both in computation efficiency and accuracy.

### 3.4.4 Radial Basis Function Methods

Another method worth noting is the method of Radial Basis functions. This is due to its similarity to the collocation method discussed in § 2.2. The method differs from

the previously discussed minimization techniques in that it provides a direct calculation of the optimal ANN parameters as opposed to an incremental search for them.

As in the collocation method, we write $f$ as

$$f(\mathbf{x}) = \sum_{i=1}^{J} c_i g(|\mathbf{x}_i - \mathbf{d}_i|)$$

where $\mathbf{d}_i$ is the center of the $i$th Gaussian function. From the point of view of the neural net we only require to construct a function $f$ which is fairly smooth and passes as close as possible through the data points. The function is therefore determined by minimizing the error function $L[f]$,

$$L[f] = \sum_{i=1}^{J} (y_i - f(\mathbf{x}_i))^2 + \mu ||Pf||^2,$$

where $||\cdot||$ denotes the Euclidean norm on $\mathbf{R}^M$. The first term is obvious and the second term is a stabilizer which has been added to make $f$ as smooth as possible. The multiplier $\mu$ measures the relative effect of the stabilizer.

It can be shown under certain conditions that the coefficients $c_i$ can be determined from the condition that $L[f]$ is minimal. The formula obtained this way is a generalization of the Moore-Penrose inverse. Let $G_{ij} = g(|\mathbf{x}_i - \mathbf{d}_j|)$ denote the $ij$th entry in the $M \times J$ matrix $G$ and similarly let $G_\square$ denote the $J \times J$ matrix contained in the upper left corner of $G$ (it is assumed that $M > J$). The parameters $c_i$ which determine the activation function are given by

$$\mathbf{c} = \left(G^T G + \mu G_\square\right)_{-1} G^T \mathbf{y}.$$

The method of radial basis functions can be represented by a three layer feed forward neural network with a single output node. The $M$ input nodes communicate with $J$ hidden layer nodes and also directly with a single output node. The weights $w(\mathrm{i}_i, \mathrm{h}_j) = d_{i,j}$ are the components of the centers $\mathbf{d}$ of the radial basis functions. The weights $w(\mathrm{h}_i, \mathrm{o}) = c_i$ are the components of the vector $\mathbf{c}$. For a given input vector $\mathbf{x}$

the hidden nodes calculate the Euclidean distances $|\mathbf{x} - \mathbf{d}_i|$ and the output node receives the value

$$z = \sum_{j=1}^{M} c_j g(|\mathbf{x} - \mathbf{d}_j|) + u$$

where $u$ is a bias which can be added to modify the characteristics of the function $g$. We can compare this directly with the equation (3.4) in the case of a single output node ($k = 1$). The radial basis function can be modified to include a bias for a node in the hidden layer. If $p$ denotes the bias parameter, then we can let $g(x, p)$ denote the modified function. For example in the case of the Gaussian function we can take $g(x, p) = \exp(-px^2)$.

This method can be easily extended to $N_I$ input nodes, $N_H$ hidden nodes and $N_O$ output nodes by defining

$$z_k(\mathbf{x}) = \sum_{j=1}^{N_H} c_{j,k} h(|\mathbf{x} - \mathbf{d}_j|, p_j) + u_k \qquad 1 \leq k \leq N_O.$$

In this case the final output of $o_k$ is calculated by applying a transfer function such as the sigmoid function $p(z_k)$.

As was mentioned earlier the advantage of using radial basis functions is the direct calculation of the weights $c_{jk}(= w_{jk})$. The standard method of training a net using incremental search algorithms differs from the method used to train a net when it is used to solve differential equations. In fact as we outline in § 4.2.1 the method is very similar in principal to the method for training the neural net when radial basis functions are used.

## 3.5 Overlearning and Generalization

One drawback of an ANN is its propensity to *overlearn* or *generalize* a problem. Overlearning is the result of the training process being too narrow in its scope. The data used to train the net is selected from only a small subset of features and doesn't

capture enough details of the problem. One interesting story of a neural net generalizing a problem occurred in a classification problem for the U.S. Army. The idea was to train a neural net to determine whether there were tanks hidden in the bushes. The generalization occurred when the ANN focused on the fact that the tank pictures during training were taken on a cloudy day and the pictures without tanks were taken on a sunny day [Fra98]. This problem could have been avoided by some preprocessing of the picture data to equalize the lighting.

Once the topology, parameterization, training and testing of a net has been completed the net is ready for operation. We can now consider how neural networks can be used to approximate solutions to differential equations.

# CHAPTER 4

## DIFFERENTIAL EQUATIONS AND NEURAL NETWORKS

Here begins the goal of the thesis which is to present an alternative method for approximating solutions to differential equations using ANNs. The method as outlined by Lagaris *et al.* is a marriage of the collocation method and the backpropagation algorithm. The collocation method provides the procedures for defining a finite mesh on the domain and for approximating a solution for only the finite set of points. The backpropagation algorithm provides the iterative approach for minimizing the error of this approximation as a training method.

We shall develop the theory for an $m$th order differential equation in $n$ independent variables $\mathbf{x} = (x_1, ... x_n)$, and a single dependent variable $\Psi$ defined on a region $P \subset \mathbf{R}^n$. The simplest way to express the equation is to use multi-index notation. A vector $\alpha = (\alpha_1, ..., \alpha_n)$, where $\alpha_j \in \mathbf{Z}_+$, is called a *multi-index of order*

$$|\alpha| = \alpha_1 + ... + \alpha_n.$$

Given a multi-index $\alpha$ define

$$D^{|\alpha|}\Psi(\mathbf{x}) := \frac{\partial^{|\alpha|}\Psi(\mathbf{x})}{\partial_{x_1}^{\alpha_1} ... \partial_{x_n}^{\alpha_n}} = \partial_{x_1}^{\alpha_1} ... \partial_{x_n}^{\alpha_n}\Psi(\mathbf{x}).$$

Then for $m \in \mathbf{Z}_+$ define

$$D^m\Psi(\mathbf{x}) := \{D^\alpha\Psi(\mathbf{x}) : |\alpha| = m\}. \tag{4.1}$$

Impose some ordering on the derivatives in $D^k\Psi(\mathbf{x})$. For example we could order the multi-indices so that $\alpha < \beta$ if either $|\alpha| < |\beta|$ or if $|\alpha| = |\beta|$ and there exists $j$, such that $\alpha_i = \beta_i$ for $1 \leq i < j$, and $\alpha_j < \beta_j$. With this notation $D\Psi$ is for example the gradient vector,

$$D\Psi = (\Psi_{x_1}, ..., \Psi_{x_n}) = \nabla\Psi.$$

If the entries in $D^2\Psi$ are arranged in a matrix then the Hessian is obtained

$$D^2\Psi = (\Psi_{x_i x_j}).$$

Thus the Laplacian is given by

$$\nabla^2\Psi = \mathrm{Tr}(D^2\Psi).$$

The multi-index notation enables the general $m$th order equation to be written as

$$G(\mathbf{x}, \Psi(\mathbf{x}), D\Psi(\mathbf{x}), ..., D^m\Psi(\mathbf{x})) = 0. \tag{4.2}$$

The next few sections will outline the process of approximating a solution, $\Psi_t$, to equation (4.2). The process begins by selecting the values of $\mathbf{x}$ as input values to train the neural network to learn $\Psi_t$.

## 4.1 Mesh

In order to begin the approximation of $\Psi(\mathbf{x})$ a set of input values is needed. As in the collocation method a set of *collocated* points or a *mesh* of points are selected from a uniform grid in $P \subset \mathbf{R}^{N_I}$. Let $\{\mathbf{x}_1, ..., \mathbf{x}_n\} \subset P$ denote the input data.

For example in the 1-D case, given $P = [a, b]$ then we take the collection of mesh points $\{x_1, x_2, \ldots, x_N\} \subset P$ where $x_{i+1} = x_i + h$, $x_1 = a$ and $h = (b - a) / (N - 1)$.

In the 2-D case, given $P = [a, b] \times [c, d]$ we can take as the collection of mesh points,$\{(x_1, y_1), (x_1, y_2), \ldots, (x_{N_x}, y_{N_y - 1}), (x_{N_x}, y_{N_y})\}$. Each $x_{i+1} = x_i + h_x$ where $h_x = (b - a) / (N_x - 1)$ and each $y_{i+1} = y_i + h_y$ where $h_y = (c - d) / (N_y - 1)$.

The selection of these points could be chosen more optimally as in the collocation method. However, in an effort to manage the scope of the thesis and maintain parameters for replicating Lagaris *et al.*'s results, we have left this study for future endeavors. Next, the neural network needs to be configured.

## 4.2 ANN Assembly

We associate with the differential equation (4.2) a neural network which has $N_I = n$ input nodes and $N_O = 1$ output node. The number of hidden nodes $N_H = h$ is at the disposal of the user. We let $w_{ij} = w(\mathrm{h}_i, \mathrm{i}_j)$ and $v_i = w(\mathrm{h}_i, \mathrm{o}_1)$. The weights and biases are initialized with a random number generator.

Figure 4.1 is an example of the network topology used for solving PDEs. The number of nodes in the hidden layer is $h = 10$. In this case there are two input nodes and one output node. The activation function is chosen to be the sigmoid function, for which the general form is (3.3). The parameters of the logistic function are chosen to result in the equation

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

which is shown in figure 3.7. This function maps the line into the unit interval, $\sigma : \mathbf{R} \to [0, 1]$.

With the mesh selected and the neural net configured, the next step is to form the approximation equation, $\Psi_t$ and begin the algorithm to minimize the error of the approximate solution.

### 4.2.1 The Minimization Problem

For the neural network approach the approximation will be represented by

$$\Psi_t(\mathbf{x}) = A(\mathbf{x}) + F(\mathbf{x}, \mathcal{N}(\mathbf{x}, \mathbf{p})) \tag{4.3}$$

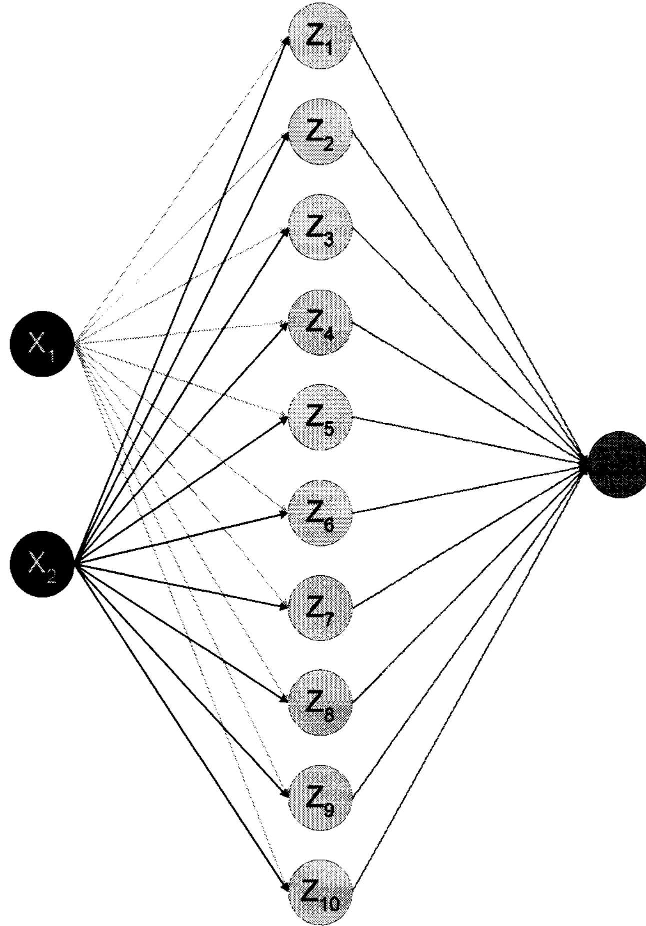


Figure 4.1: Representative ANN for a PDE

where $A$ and $F$ incorporate the constraints of the boundary conditions. For a single boundary condition, $\Psi(b_0) = \beta_0$ equation (4.3) becomes

$$\Psi_t(\mathbf{x}) = \beta_0 + (x - b_0)\mathcal{N}(\mathbf{x}, \mathbf{p}). \tag{4.4}$$

The minimization of the approximation error is then a modified version of equation (3.6).

$$E(\mathcal{X}, \mathbf{p}) = \sum_i \left| D^m \Psi_t(\mathbf{x}, \mathbf{p}) - f\left(x_i, \Psi_t, D^1\Psi_t, \ldots, D^{m-1}\Psi_t\right) \right|^2. \tag{4.5}$$

In order to minimize this error the backpropagation algorithm is employed. As mentioned previously, the backpropagation algorithm is done using an unsupervised method,

§ 3.4, one that does not require a set of output values for the exact solution. In general the only given values of the exact solution are either boundary values or initial values. This set of values would not be sufficient to train a neural net on the entire domain. The neural network, if trained on the boundary values, would definitely suffer from generalization, § 3.5. Every solution would behave as a boundary value behaves.

The backpropagation algorithm is similar to the Galerkin method of approximating differential equations when using the Ritz approach [Goc06]. In the Ritz algorithm, the minimization problem is to minimize the difference between the load vector and the inner product of a finite dimensional approximating function. In order to minimize the error of the solution using ANN's traditional minimization techniques discussed in § 4.2.1 are used to minimize the error in equation (4.5) given the differential equation,(4.2), over a finite set of mesh points.

For each cycle of the backpropagation algorithm the neural network processes each input data vector and obtains a corresponding output value before the network is updated with new parameters. Subsequent cycles use the same input data but the output changes for each cycle because the parameters are modified by a user defined minimization technique. The minimization algorithm measures the error defined in (4.5). Since, $F$ and $A$ are represented by some product of some polynomial in $\mathbf{x}$ with constant coefficients and $N$. The derivative of the approximation is found by using the chain rule.

$$D^m \Psi_t = D^m A(\mathbf{x}) + D^m F(\mathbf{x}, \mathcal{N}(\mathbf{x}, \mathbf{p}))\, D^m \mathcal{N}(\mathbf{x}, \mathbf{p}). \tag{4.6}$$

Since the above equation's only variable parameters are the neural network parameters, $\mathbf{p}$, these are the parameters that are updated using the minimization technique. These iterations continue until a minimum is attained, a user defined error tolerance is reached, or the number of iterations has surpassed the user defined maximum number of iterations.

In order to execute the minimization technique the derivatives of the neural network

with respect to its parameters need to be explained. Recall from section § 3.2 that the neural network is a linear sum of constant coefficients and transfer functions, $\sigma$.

$$\mathcal{N}(\mathcal{X}) = \sum_{i=1}^{h} v_i \sigma(z_i)$$

and

$$z_k = \sum_{j=1}^{N_I} w_{jk} x_j + u_k.$$

The derivatives of the output with respect to any of its inputs as well as the gradient of the network derivatives with respect to their inputs must be calculated. To find the derivative of $\mathcal{N}$ with respect to $x_j$, $w_{ij}$, $v_i$ and $u_k$ the chain rule is again used. First we show the derivative of the ANN, $\mathcal{N}$, with respect to any of its input nodes, $x_j$. That is,

$$\frac{\partial^k \mathcal{N}}{\partial x_j^k} = \sum_{i=1}^{h} v_i w_{ij}^k \sigma_i^{(k)}. \tag{4.7}$$

*Proof.* For $k = 1$:

$$\begin{aligned}
\frac{\partial \mathcal{N}}{\partial x_j} &= \frac{\partial}{\partial x_j}\left(\sum_{i=1}^{h} v_i \sigma\left(\sum_{j=1}^{n} w_{ij} x_j + u_i\right)\right) \\
&= \sum_{i=1}^{h} v_i \sigma'\left(\sum_{j=1}^{n} w_{ij} x_j + u_i\right) \frac{\partial}{\partial x_j}(w_{ij} x_j + u_i) \\
&= \sum_{i=1}^{h} v_i \sigma'(z_i)\, w_{ij} \\
&= \sum_{i=1}^{h} v_i \sigma^{(1)}(z_i)\, w_{ij}^1
\end{aligned}$$

Assume for $k = n - 1$:

$$\frac{\partial^{n-1} \mathcal{N}}{\partial x_j^{n-1}} = \sum_{i=1}^{h} v_i \sigma^{(n-1)}(z_i)\, w_{ij}^{n-1}$$

For $k = n$:

$$\begin{aligned}
\frac{\partial^n \mathcal{N}}{\partial x_j^n} &= \frac{\partial}{\partial x_j}\left(\sum_{i=1}^{h} v_i \sigma^{(n-1)}(z_i)\, w_{ij}^{n-1}\right) \\
&= \sum_{i=1}^{h} v_i \sigma^{(n-1)}\left(\sum_{j=1}^{n} w_{ij}x_j + u_i\right) w_{ij}^{n-1} \frac{\partial}{\partial x_j}(w_{ij}x_j + u_i) \\
&= \sum_{i=1}^{h} v_i \sigma^{(n)}(z_i)\, w_{ij}^{n-1} w_{ij} \\
&= \sum_{i=1}^{h} v_i \sigma^{(n)}(z_i)\, w_{ij}^{n}
\end{aligned}$$

Thus, by induction equation (4.7) holds true for all $n$. □

Next the gradient of the ANN with respect to it's input nodes, $\mathcal{N}_g$, is determined. First, let $P_i = \prod_{k=1\ldots n} w_{ik}^{\alpha_k}$ and the derivative index $n$ is defined in (4.1) then

$$\mathcal{N}_g = D^n \mathcal{N} = \sum_{i=1}^{n} v_i P_i \sigma_i^{(n)}. \tag{4.8}$$

*Proof.* For $k = 1$:

$$\begin{aligned}
D^1 \mathcal{N} &= \frac{\partial^{\alpha_1} \mathcal{N}}{\partial x_1^{\alpha_1}} \\
&= \sum_{i=1}^{h} v_i w_{ij}^{\alpha_1} \sigma^{(\alpha_1)}(z_i) \qquad (4.9) \\
& \qquad (4.10)
\end{aligned}$$

from previous proof.

Assume for $k = n - 1$:

$$D^{(n-1)} \mathcal{N} = \sum_{i=1}^{h} v_i \prod_{j=1}^{n-1} w_{ij}^{\alpha_j} \sigma_i^{(n-1)}(z_i).$$

For $k = n$:

$$
\begin{aligned}
D^n\mathcal{N} &= \frac{\partial^{\alpha_n}}{\partial x_{n^{\alpha_n}}}\left(D^{n-1}\mathcal{N}\right) \\
&= \frac{\partial^{\alpha_n}}{\partial x_{n^{\alpha_n}}}\left(\sum_{i=1}^{h} v_i \prod_{j=1}^{n-1} w_{ij}^{\alpha_j}\sigma_i^{(n-1)}(z_i)\right) \\
&= \sum_{i=1}^{h} v_i \prod_{j=1}^{n-1} w_{ij}^{\alpha_j}\left(\frac{\partial^{\alpha_n}}{\partial x_{n^{\alpha_n}}}\sigma_i^{(n-1)}(z_i)\right) \\
&= \sum_{i=1}^{h} v_i \prod_{j=1}^{n-1} w_{ij}^{\alpha_j}\sigma_i^{(n-1)+\alpha_n}(z_i)\frac{\partial^{\alpha_n} z_i}{\partial x_{n^{\alpha_n}}} \\
&= \sum_{i=1}^{h} v_i \prod_{j=1}^{n-1} w_{ij}^{\alpha_j}\sigma_i^{(n-1+1)}(z_i)\frac{\partial^{\alpha_n}\left(\sum_{m=1}^{n} w_{im}x_m + u_i\right)}{\partial x_{n^{\alpha_n}}} \\
&= \sum_{i=1}^{h} v_i \prod_{j=1}^{n-1} w_{ij}^{\alpha_j}\sigma_i^{(n)}(z_i)\, w_{in}^{\alpha_n} \\
&= \sum_{i=1}^{h} v_i \prod_{j=1}^{n} w_{ij}^{\alpha_j}\sigma_i^{(n)}(z_i) \\
&= \sum_{i=1}^{h} v_i P_i \sigma_i^{(\Lambda)}(z_i)
\end{aligned}
$$

Therefore, by induction, equation (4.8) holds true for all $n$. □

It is also necessary to figure out the derivative of $\mathcal{N}$ with respect to $v_i$, $u_i$, and $w_{ij}$. Finding the derivatives of $\mathcal{N}_g$, with respect to the other parameters becomes straightforward.

$$\frac{\partial \mathcal{N}_g}{\partial v_i} = P_i \sigma_i^{(\Lambda)}. \tag{4.11}$$

$$\frac{\partial \mathcal{N}_g}{\partial u_i} = v_i P_i \sigma_i^{(\Lambda+1)}. \tag{4.12}$$

Finally, finding the partial derivative of $\mathcal{N}$ with respect to $w_{ij}$ requires using the product rule.

$$\frac{\partial \mathcal{N}_g}{\partial w_{ij}} = x_j v_i P_i \sigma_i^{(\Lambda+1)} + v_i \Lambda_j w_{ij}^{\Lambda_i - 1} \left( \prod_{k=1, k \neq j} w_{ik}^{\Lambda_k - 1} \sigma_i^{(\Lambda)} \right). \tag{4.13}$$

Now that all of the derivatives of the gradient network have been found with respect to all of its network parameters, the techniques for minimizing non-linear surfaces can be used. These are the same methods employed by the supervised training methods. Though any minimization technique can be used, BFGS is chosen due to its superior performance [ILF98].

The ANN parameters are then modified by changes that the BFGS algorithm suggests and the iterations continues producing an approximation, $\Psi_t$, that nears the exact solution, $\Psi$.

## 4.3 Error Estimation

Optimally, the error between the approximation and the exact solution would go to zero in a finite number of backpropagation iterations, $n_i$. However, there are two obstacles to this occurring and eventually obtaining $\Psi_t = \Psi$. As was mentioned in § 3.4.3 an optimal solution is only guaranteed by BFGS if the error surface, dependent on the number of parameters $n_p$ is *unimodal.* Since this is not generally known, one approach to insuring this behavior in the differential equation solution is to divide the domain of the differential equation solution into sections or elements known to have convex properties as in the Galerkin method. In an effort to stay focused on the method, this tangent would be an excellent subject for further research. Instead, it is known that for an $n_p$ dimensional problem, typical BFGS error estimates are defined by the product of $n_i$ and the number of calculations made at each iteration [Rar98]. The majority of calculations is in determining the direction toward the minimum error and finding the gradient with respect to the parameters. Finding a direction involves determining a deflection matrix of size $n_p$ and calculating the gradient of size $n_p$. Each hidden node has three parameters

associated with it, $w_{ij}, u_i$, and $v_i$ producing an $n_p = 3N_H$ where $N_H$ is the number of hidden nodes. This gives the minimization portion an error of order dependent on $N_H$.

The second obstacle in approximating the error is the ability of a neural net to approximate a function. This area of research is an open ended question in the study of ANNs and has been a deterrent for users of neural nets. The next section describes the current status of the subject with respect to approximating differential equations. It is determined that the error of approximating the error function is also dependent on $N_H$. In chapter 5 reasonable approximations are achieved in the examples performed and within a small number of iterations of the backpropagation algorithm.

### 4.3.1 Neural Networks and Function Approximation

The use of a feed forward neural network to solve differential equations raises some fundamental questions. The most important of these is whether such a neural network can in fact approximate an unknown function $\Psi : \mathbf{R}^n \to \mathbf{R}$ (the solution of the differential equation) and its derivatives. If this is the case then does it require some specific or optimal number of hidden layers? Finally we can inquire about the class of transfer functions which are applicable to this problem. For example if the transfer function is a step function as discussed in § 3.2.1 then it is a piecewise continuous function which has derivative zero almost everywhere. Thus although this transfer function can be used to approximate the solution of the differential equation, it cannot be used to approximate its derivative. This type of transfer function is therefore not applicable to solve differential equations where successive approximations to the solution and its derivatives are calculated at each cycle through the network.

The original papers which investigated function approximation by feed forward neural networks were due to Hornik *et al.* [MHW90] and Leshno *et al.* [MLS93]. More recent papers refine the original results, but do not essentially change the conclusions of

the earlier works. It turns out that a feed forward neural network with a single hidden layer is capable of approximating a function and its derivatives to an arbitrary level of accuracy for relatively mild smoothness conditions on the activation function [MHW90]. In fact even more is possible. The network is capable of approximating functions which only have generalized derivatives (such as piecewise continuous functions).

In order to appreciate what is involved let us consider a neural network with $N_I$ input nodes, a single hidden layer of $N_H$ nodes, and one output node. The input to the $k$th hidden node $z_k$ is defined in (3.4). Let $f : \mathbf{R} \to \mathbf{R}$ denote the transfer function and let $\mathcal{N} : \mathbf{R}^{N_I} \to \mathbf{R}$ denote the output function of the neural network (the dependence on the weights and biases has been suppressed). Then if $\mathbf{x} \in \mathbf{R}^{N_I}$ and $v_k = w(\text{ h}_k, \text{ o})$ is the weight on the edge from the $k$th hidden node to the output node the output function from the net is

$$\mathcal{N}(\mathbf{x}) = \sum_{k=1}^{N_H} v_k f\left(\sum_{j=1}^{N_I} w_{jk} x_j - u_k\right) = \sum_{k=1}^{N_H} v_k f(\mathbf{x}.\mathbf{w}_k - u_k) \tag{4.14}$$

where $\mathbf{w}_k = (w_{1k}, ..., w_{N_I k})$. The first partial derivatives of the network output function are

$$\frac{\partial \mathcal{N}(\mathbf{x})}{\partial x_i} = \sum_{k=1}^{N_H} v_k w_{ik} Df(\mathbf{x}.\mathbf{w}_k - u_k) \tag{4.15}$$

where $Df$ is the derivative of $f$. Thus both (4.14) and (4.15) have the same functional form. It is well known (we briefly review some of the results later), that an unknown function $\Psi$ can be well approximated by an appropriate choice of weights and biases. The same is therefore true of $\Psi_{x_i}$. However it was not clear until the papers [MHW90], [MLS93] appeared that the same set of weights and biases could be chosen for which both $\Psi$ and its derivatives could be arbitrarily closely approximated.

There is however a big snag in the proof; it assumes a *continuum* of hidden nodes. This idea was first proposed by Irie & Miyake [IM88] and enables the use of Fourier transforms to considerably simplify the calculations. For example suppose $\Psi \in C^r(\mathbf{R}^n)$,

which we assume to be the unknown function, where $n = N_I$ corresponds to the number of input nodes. Then if

$$\hat{f}(\mathbf{a}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbf{R}^n} f(\mathbf{x}) e^{-i\mathbf{x}.\mathbf{a}} d\mathbf{x}$$

in the transform space the derivatives of $f$ can be simply expressed as

$$\widehat{D^\alpha f}(\mathbf{a}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbf{R}^n} D^\alpha_{\mathbf{x}} e^{-i\mathbf{x}.\mathbf{a}} d\mathbf{x} = (i\mathbf{a})^\alpha \hat{f}(\mathbf{a}).$$

Let $L_p(\mathbf{R})$ denote the space of Lebesgue measurable functions on $\mathbf{R}$ with the norm

$$||f||_p = \left[ \int_{\mathbf{R}} |f(x)|^p dx \right]^{1/p}.$$

The Sobolev space $S^m_p(\mathbf{R})$ is defined as the collection of functions $g \in C^m(\mathbf{R})$ such that $||D^\ell g||_p < \infty$ for all $\ell \leq m$. We also introduce the notation $C^\infty_\downarrow(\mathbf{R})$ to denote functions in $C^m(\mathbf{R})$ of *rapid decrease*. This means that for any $g \in C^\infty_\downarrow(\mathbf{R})$ and $j, l \geq 0$ $x^j D^\ell g(x) \to 0$ as $|x| \to \infty$.

The main result of Hornik *et al.* is that provided the transfer function $f \neq 0$ belongs to the Sobolev space $S^m_1(\mathbf{R}, \lambda)$, $\mathcal{N}$ can approximate any function belonging to $C^\infty_\downarrow(\mathbf{R})$ and its derivatives up to order $m$ arbitrarily closely on compact sets. However these conditions are too strong for most transfer functions; it rules out logistic and hyperbolic tangent squashing functions, and indeed any sigmoid function (3.3)

It turns out that the conditions on the transfer function can be considerably weakened to $\ell$-finite functions. These are functions $g \in C^\ell(\mathbf{R})$ for which

$$0 < \int_{\mathbf{R}} |D^\ell g(x)| dx < \infty.$$

That is if the transfer function $f \in C^\ell(\mathbf{R})$ and $D^\ell f \in L_1(\mathbf{R})$, for some $\ell \geq 0$, then $\mathcal{N}$ can approximate any function $\Psi \in C^\infty_\downarrow(\mathbf{R})$. The logistic and hyperbolic tangent transfer functions are now covered by this result. Unfortunately the result does not give any insight into the optimal number of hidden nodes required to approximate $\Psi$.

# CHAPTER 5

# IMPLEMENTATION

Lagaris *et al.* make little mention of the software used or developed for performing the neural net approximation of differential equations. They do direct the reader to a program called Merlin. However, Merlin is solely an optimization package. Discussion of the neural net piece was absent. This required searching for or creating a tool to use in applying the neural net approximation theory.

## 5.1 Neural Net Software

Many challenges existed when searching for neural net software. First, the majority, if not all, of neural net software applications available in the market use supervised training methods. This means the applications are hard-coded to train the neural net using training data sets, i.e. two vectors of input and output values.

Second, software implementations of multi-layer feed-forward neural networks range from the extremely expensive software of Wardsystem, NeuroShell 2 for stock trading predictions, to freeware by Sylvain Muise, to a simple Maplet that simulates an ANN to solve Boolean problems. These applications tend to have specific user interfaces catering to the problem being addressed. Since, solving differential equations with an ANN is a relatively new idea and the algorithm used was unsupervised, an implementation need to be created or another implementation needed to be modified.

The initial thought was to modify the existing program SNNS, Stutgart Neural

Network Simulator, to do unsupervised training with a customized learning algorithm, i.e. the minimization problem outlined in § 4.2.1. The application had a decent user interface and graphical output. However, the structure of the code did not lend itself to unsupervised learning. Similar attempts were made with various other existing neural net software but the final evaluation was that the traditional backpropagation algorithm was intricately embedded in all of these applications.

The new goal became finding a set of simple classes for defining a neural net and for performing the optimization. These two pieces were found easily. A neural net class developed by Jasper Bedaux at the University of Amsterdam in 2002 along with the GNU scientific library for the optimization covered the foundation for what is deemed the Artificial Neural Net for the Approximation of Differential Equation Solutions (ANNADES).

### 5.1.1 ANNADES

The neural net classes and the optimization library provided the basic layer of ANNADES. The remaining work was to provide a link between them and incorporate the minimization function for the differential equations. Since each differential equation is different the details of the minimization had to be changed for each approximation. To limit user error in these modifications, all changes for each approximation are made in a small number of functions. The best way to illustrate ANNADES is by first looking at the functional flow diagram of the main program in figure 5.1.

The main program defines the mesh of the domain, creates the neural net, minimizes the parameters of the net, and outputs the minimization results.

The minimization work occurs in the *Optimize Neural Net* process. A similar functional flow diagram can be referred to in figure 5.2. Here is where the majority of the calculations are done. The minimizer is defined using the GSL classes. The weights

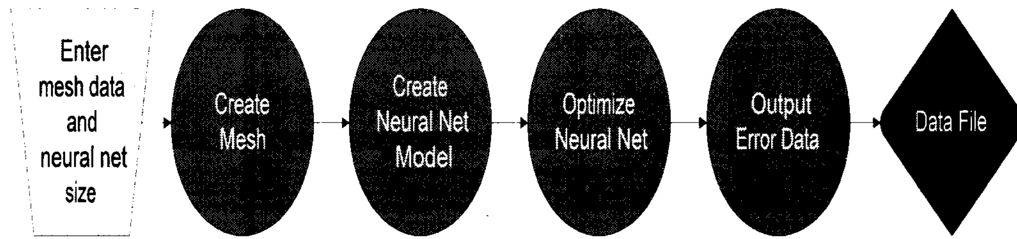


Figure 5.1: ANNADES Main Function Flow

and biases of the ANN are stored in a parameter vector, **p**. These parameters are then optimized using the user defined differential equation. The optimization occurs over the course of several iterations. At each iteration, the error function, (4.5), the derivative of the ANN with respect to the input, (4.8) and the gradient of the Error function with respect to **p**, using (4.11), (4.12) and (4.13), are calculated. The gradient is checked to determine if a minimum has been reached, the derivative of the error function is checked to determine if the step sizes are near zero and the number of iterations is checked to determine if the maximum number of iterations has been reached. If any of these conditions are met, the optimization process is exited and the minimization output is generated.

## 5.2 Example Problems

The difficulty in verifying the results given by Lagaris *et al.* is that the differences between their implementation and ANNADES is unknown. In an attempt to minimize these differences, the goal was to confirm the results of their example problems. The results are illustrated in the following section of examples.

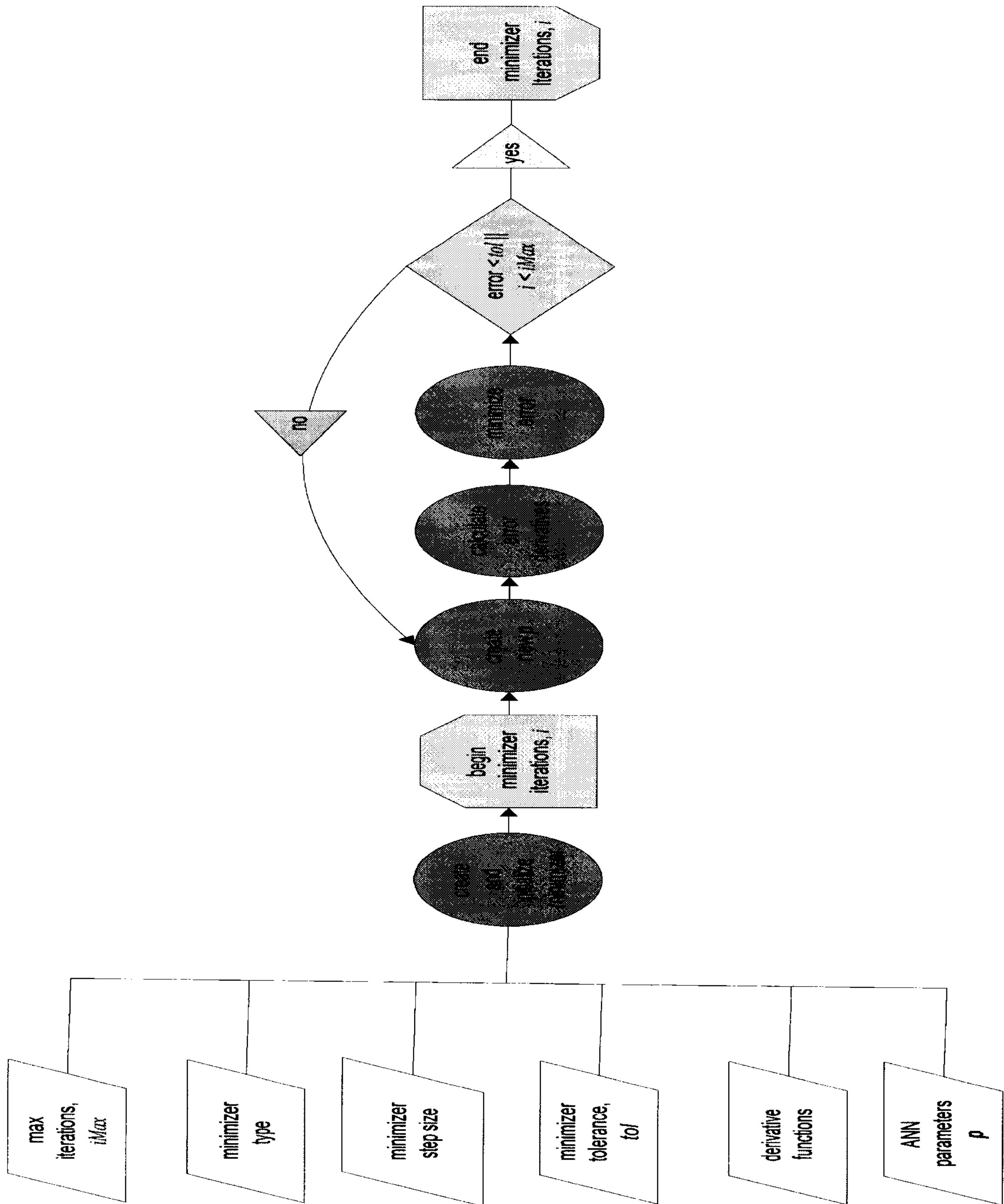


Figure 5.2: ANNADES Optimization Function Flow

### 5.2.1 Ordinary Differential Equation, Example 1

The first example is given as

$$\frac{d}{dx}\Psi + \left(x + \frac{1+3x^2}{1+x+x^3}\right)\Psi = x^3 + 2x + x^2\frac{1+3x^2}{1+x+x^3} \qquad (5.1)$$
$$\Psi(0) = 1, x \in [0,1].$$

The exact solution is

$$\Psi_a(x) = \frac{exp^{-\frac{x^2}{2}}}{1 = x + x^3} + x^2.$$

The neural net approximation is

$$\Psi_t(x, \mathbf{p}) = 1 + x\mathcal{N}(x, \mathbf{p}) \qquad (5.2)$$

where each parameter $p \in \mathbf{p}$ is initialized to a random value in $[-1, 1]$ and modified through the optimization algorithm.

For this problem, ten $x_i$ equally spaced in $[0, 1]$, i.e. the mesh size, $h = 0.1$. The ANN had ten hidden nodes. The optimization method was BFGS and the approximation equation was (5.2). The minimization ran for 35 iterations before the program determined that the optimization was not making any progress. This meant that the size of the steps in the direction of the minimum were near zero. The resulting sum of the squared error ($SSE$) for the gradient was 0.504079 which the same magnitude as $h$,i.e. $O(h)$. The absolute error for each $x_i$,

$$e(x_i) = |\Psi_a(x_i) - \Psi_t(x_i, \mathbf{p})| \qquad (5.3)$$

achieved errors as high as 0.57932389. See figure 5.3.

Next, as in the finite element methods, it was necessary to determine if the approximation would improve with smaller mesh sizes. The approximation routine was run for $h = 0.1$, $h = 0.05$, $h = 0.025$, $h = 0.0125$ and achieved results in 87, 26, 24, 51 iterations respectively. Unexpectedly, the approximation did not improve but worsened with no visible correlation to the mesh size. The results can be viewed in figure 5.4.

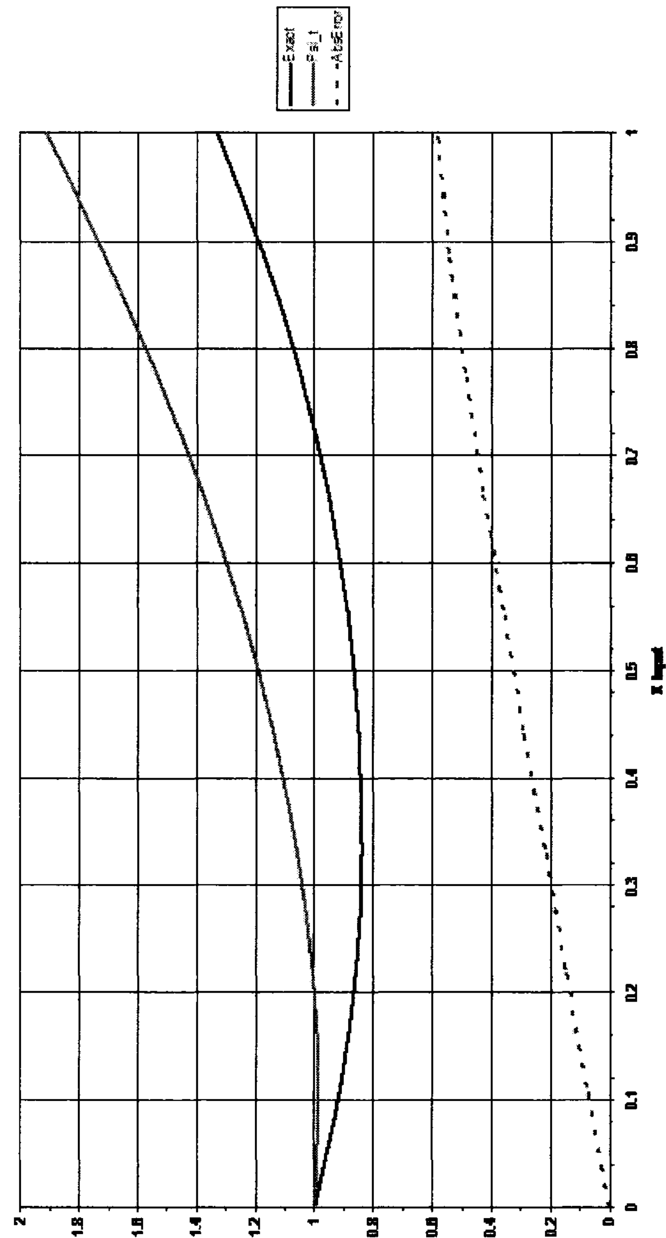


Figure 5.3: Absolute error approximation for problem 5.1, $h = 0.1$

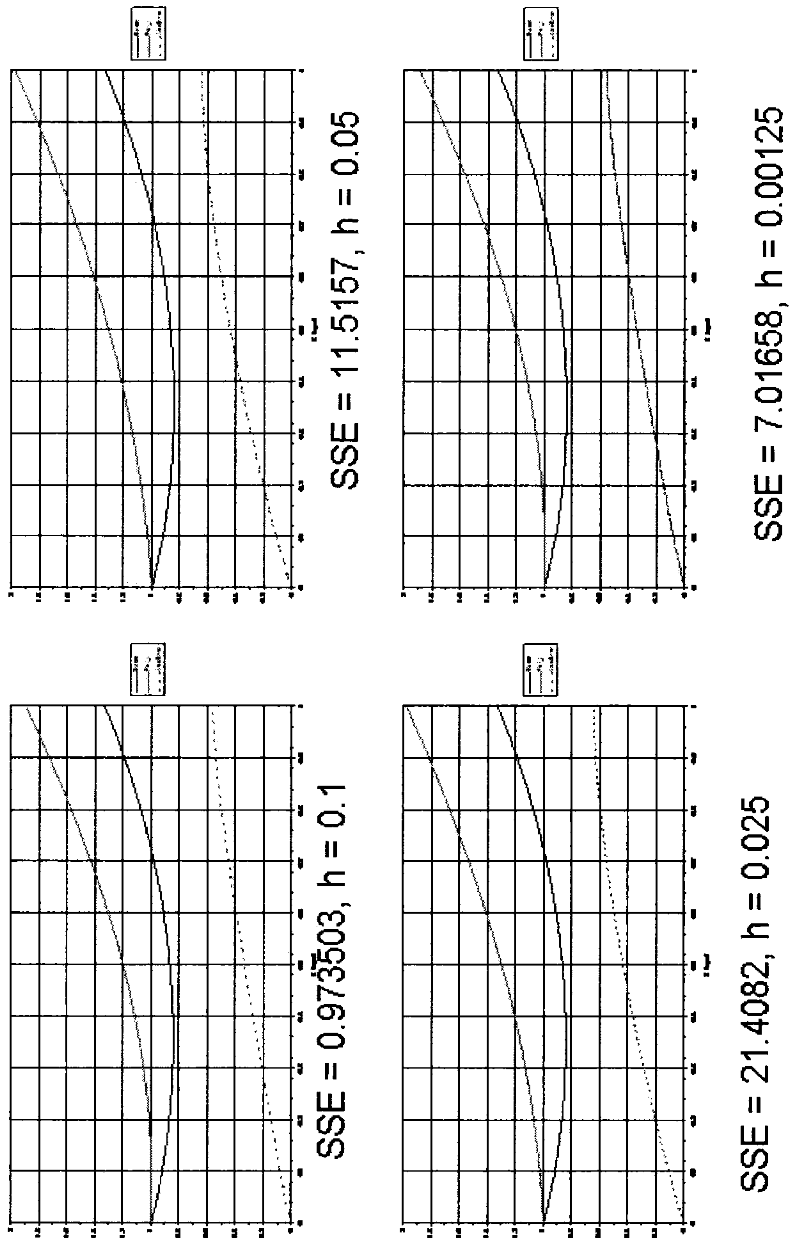


Figure 5.4: Absolute error approximations for problem 5.1, $h < 0.1$

Since the approximation did not improve by reducing the mesh size, the next idea was to increase the number of hidden nodes in the ANN. This simulates increasing the dimension of the basis since the neural network output is a summation of sigmoid functions, see equation (3.2). The result was that the SSE was reduced to 0.456032 but was still $O(h)$. This result was achieved in 31 iterations and can be viewed in figure 5.5.

Finally, a last attempt was made at reducing the magnitude of the error. After some consideration, it was noted that the approximation $\Psi_t$ was numerically a linear approximation and the general form of the differential equation had an $x^3$ term. This prompted another look at the neural net approximation and led to a new approximation of the form

$$\tilde{\Psi_t}(x, \mathbf{p}) = 1 + x^3 \mathcal{N}(x, \mathbf{p}). \tag{5.4}$$

The parameters of $\mathcal{N}(x, \mathbf{p})$ were optimized in 14 iterations using the new form, (5.4), and achieved an $SSE = 0.0450667$. This error was $\frac{1}{10}$ the magnitude of $h$ or $O(h^2)$ and the absolute errors were less than 0.2, a marked improvement over the original approximation model. See figure 5.6.

### 5.2.2 Ordinary Differential Equation, Example 2

The next differential equation to be approximated is the following equation.

$$\frac{d}{dx}\Psi + \frac{1}{5}\Psi = \exp^{-\frac{x}{5}} \cos(x) \tag{5.5}$$
$$\Psi(0) = 0, x \in [0, 2].$$

The exact solution is $\Psi_a(x) = exp^{-\frac{x}{5}} \sin(x)$.

The first neural net approximation tried was,

$$\Psi_t(x, \mathbf{p}) = x\mathcal{N}(x, \mathbf{p}) \tag{5.6}$$

where $\mathbf{p}$ is a set of optimized values ranging from $[-1, 1]$.

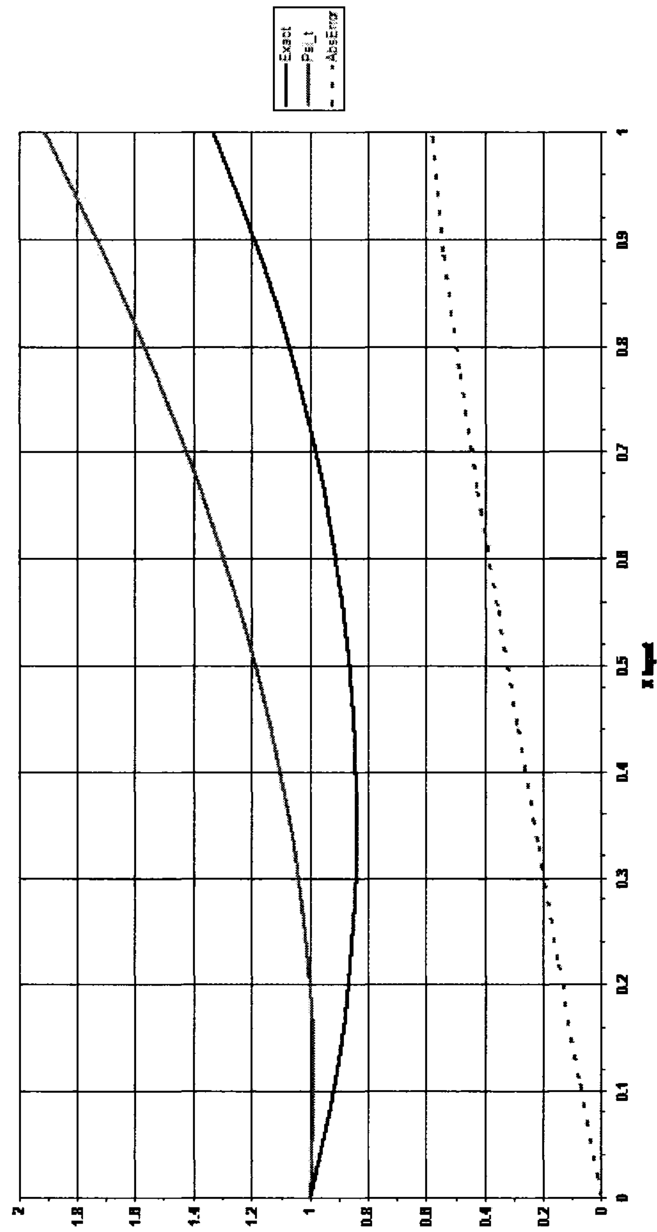


Figure 5.5: Absolute error approximation for problem 5.1, $N_H = 15$

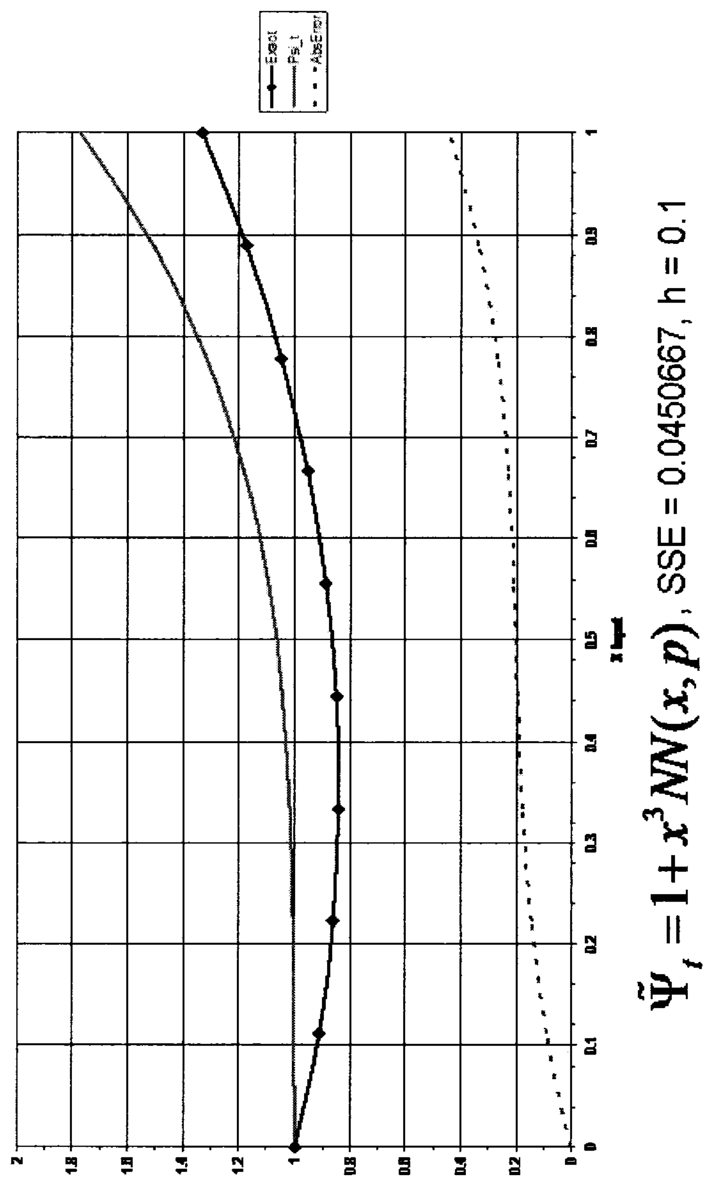


Figure 5.6: Absolute error approximation for problem 5.1, using $\tilde{\Psi}_t$

Note that the neural net approximation is similar to the first problem. There were ten $x_i$ equally spaced creating a mesh size of $h = 0.2$ since the domain interval was $[0, 2]$. The ANN had ten hidden nodes and the optimization method was BFGS. The minimization ran for 39 iterations before the program again determined that the optimization was not making any progress. However, in this case the resulting $SSE$ for the gradient error (3.7) was 0.0768531 which is $O(h^2)$. The figure 5.7 illustrates this result.

These results are promising in providing an alternative method for approximating solutions to differential equations. The speed and efficiency of the method is driven by the minimization technique as in other finite element techniques. The speed of the minimization techniques stems from the number of parameters, § 3.4. In other finite element techniques the results improve as the mesh size decreases which in turn slows the minimization algorithm. In contrast, the neural network approximation results seem to be independent of the size of the hidden layer and do not require increased mesh sizes for the approximation to improve. This keeps the number of parameters to be optimized a small constant value in the minimization. The neural network approximation presumably could achieve same or better results as the traditional methods.

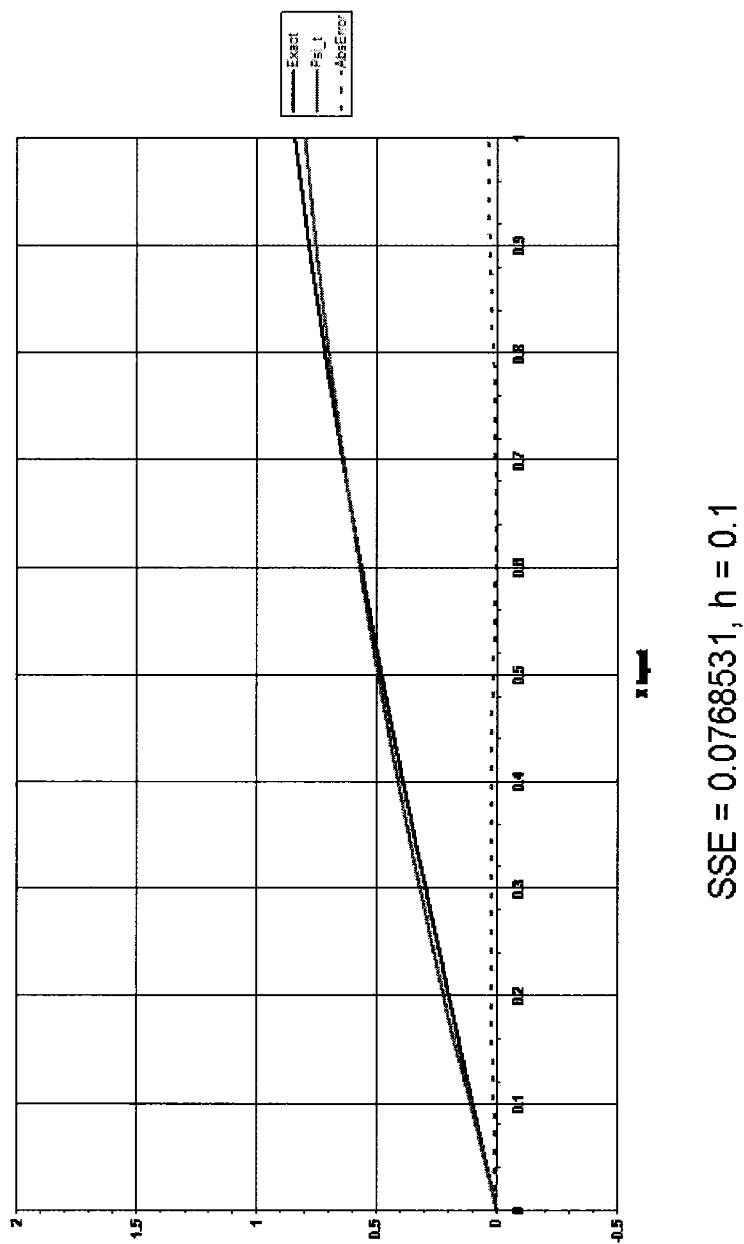


Figure 5.7: Absolute error approximation for problem 5.5 with a mesh size = 0.1

# CHAPTER 6

# CONCLUSION

One of characteristics of a relatively new field of science is that there are many unanswered questions. The idea of a neural network has only been around for a little over half a century with a thirty year hiatus. There is research being done in how to sample the domain of a neural network problem, § 3.1. There are questions surrounding the neural net architecture and the influences of the architecture on problem solving [MDRM03]. There are numerous ideas surrounding the activation functions of the hidden nodes [CW04]. There is also the widely researched area of the convergence of solutions using neural networks [Cao04].

Similarly, trying to corroborate Lagaris *et al.*'s results provided an alternative path of analysis regarding the neural net approximation models. Further research in the modification of these approximations and a future implementation of partial differential equation approximations will better the understanding of approximating differential equations using neural networks. With so many open ended avenues to pursue in the theory and implementation of neural networks models to approximate differential equations, the challenge is selecting a destination.

# BIBLIOGRAPHY


[AB99] Martin Anthony and Peter L. Bartlett, *Neural network learning: Theoretical foundations*, Cambridge University Press, 1999.

[AM92] Dave Anderson and George McNeil, *Artificial neural networks technology*, Tech. report, Data and Analysis Center for Software, 775 Daedalian Drive Rome, NY 13441-4909, August 1992, Prepared for Rome Laboratory, Grifiss Business Park, rome, NY 13441.

[Bha99] H. K. D. H. Bhadeshia, *Neural networks in material science*, ISIJ International **39** (1999), no. 10, 966–79.

[Bri07] Encyclopedia Britannica, *Artificial intelligence, creating an artificial neural network*, Encyclopedia Britannica (2007), 25.

[BT96] Dimitri P. Bertsekas and John Tsitsiklis, *Neuro-dynamic programming*, Athena Scientific, 1996.

[Cao04] Jinde Cao, *An estimation of the domain of attraction and convergence rate for hopfield continuous feedback neural networks*, Physics Letters A **325** (2004), 370–374.

[CW04] Jinde Cao and Jun Wang, *Absolute exponential stability of recurrent neural networks with lipschitz-continuous activation functions and time delays*, Neural Networks **17** (2004), 379–390.

[dBS73] Carl de Boor and Blair Swartz, *Collocation at gaussian points*, SIAM Journal of Numerical Analysis **10** (1973), 582–606.

[DHAS85] G. E. Hinton D. H. Ackley and T. J. Segnowski, *A learning algorithm for boltzmann machines*, Cognitive Science **9** (1985), 147–169.

[Fra98] Neil Fraser, *Neural network follies*, `http://neil.fraser.name/writing/tank/`, 1998.

[FVF04] Aristidis C. Likas Fotis Vartziotis, Isaac Elias Lagaris and Dimitrios I. Fotiadis, *A portable decision making tool for health professionals based on neural networks*, Health Informatics Journal **9** (2004), 273–82.

[GL88] Ronald B. Guenther and John W. Lee, *Partial differential equations of mathematical physics and integral equations*, Prentice Hall, 1988.

[Goc02] Mark S. Gockenbach, *Parital differential equations analytical and numerical methods*, Society For Industrial and Applied Mathematics, 2002.

[Goc06] ______, *Understanding and implementing the finite element method*, Society For Industrial and Applied Mathematics, 2006.

[GZ07] Mara P. Gonzalez and Jose L. Zapico, *Seismic damage identification in buildings using neural networks and modal data*, Computers and Structures (2007), http://dx.doi.org doi:10.1016/j.compstruc.2007.02.021.

[Hah07] Michael E. Hahn, *Feasibility of estimating isokinetic knee torque using a neural network model*, Journal of Biomechanics **40** (2007), 1107–1114.

[Hop82] J. J. Hopfield, *Neural networks and physical systems with emergent collective computational abilities*, Proceedings of the National Academy of Sciences, USA **79** (1982), 2554–2558.

[HT06] Yi-Chung Hua and Fang-Mei Tseng, *Functional-link net with fuzzy integral for bankruptcy prediction*, Neurocomputing (2006), http://dx.doi.org doi:10.1016/j.neucom.2006.10.111.

[Hug00] Thomas J. R. Hughes, *The finite element method, linear static and dynamic finite element analysis*, Dover Publications, Inc., 2000.

[ILF98] A. Likas I. Lagaris and D. Fotiadis, *Artificial neural networks for solving ordinary and partial differential equations*, IEEE Transactions on Neural Networks **9** (1998), 987–1000.

[ILP00] Aristidis C. Likas I. Lagaris and Dimitrios G. Papageorgiou, *Neural network methods for boundary value problems with irregular boundaries*, IEEE Transactions on Neural Networks **11** (2000), 1041–9.

[IM88] B. Irie and S. Miyake, *Capabilities of three-layered perceptrons*, Proceedings of the 1988 IEEE International Conference on Neural Networks (1988), 217–225.

[JMS55] N. Rochester J. McCarthy, M. L. Minsky and C.E. Shannon, *Proposal for the dartmouth summer resarch project on artificial intelligence*, 1955.

[KHW89] Maxwell Stinchcombe Kurt Hornik and Halber White, *Multilayer feedforward networks are universal approximators*, Neural Networks **2** (1989), 359–366.

[Kin95] S. E. Kinglsand, *Modeling nature*, Wikimedia Foundation, Inc., 1995, en.wikipedia.org/wiki/Logistic_function.

[MDRM03] Joel S. Parker Lance W. Hahn Marylyn D. Ritchie, Bill C. White and Jason H. Moore, *Optimization of neural network architecture using genetic programming improves detection and modeling of gene-gene interactions in studies of human diseases*, BMC Bioinformatics **4** (2003), 28.

[MDTC01] Nam Mai-Duy and Thanh Tran-Cong, *Numerical solution of differential equations using multiquadric radial basis function networks*, Neural Networks **14** (2001), 185–199.

[MHW90] M. Stinchcombe M. Hornik and H. White, *Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks*, Neural Networks **3** (1990), 551–560.

[MLS93] A. Pincus M. Leshno, Ya. Lin and S. Schocken, *Multilayer feedforward networks with a non-polynomial activation function can approximate any function*, Neural Networks **6** (1993), 861–867.

[MP87] Marvin L. Minsky and Seymour A. Papert, *Perceptrons - expanded edition: An introduction to computational geometry*, The MIT Press, 1987.

[MR90] B. Mueller and J. Reinhardt, *Neural networks, an introduction*, Springer-Verlag, 1990.

[NMNR06] M. R. Ransing N. M. Nawi and R.S. Ransing, *An improved learning algorithm based on the conjugate gradient method for back propogation neural networks*, Transactions on Engineering, Computing and Technology **14** (2006), 211–215.

[Rar98] Ronald L. Rardin, *Optimization in operations research*, Prentice Hall, 1998.

[RG58] Nathaniel Rochester and H. L. Gelernter, *Intelligent behavior in problem-solving machines*, IBM Journal of Research and Development **2** (1958), no. 4, 336–345.

[S+07] Jonathan Schaeffer et al., *Checkers is solved*, Science (2007).

[Sam07] Sandhya Samarasinghe, *Neural networks for applied sciences and engineering*, Auerback Publications, 2007.

[Sew88] Granville Sewell, *The numerical solution of ordinary and partial differential equations*, Academic Press, Inc, 1988.

[SHU00] K. Reif S. He and R. Unebehauen, *Multilayer neural networks for solving a class of partial differential equations*, Neural Networks (2000), 385–396.

[SN96] Gilbert Strang and Truong Nguyen, *Wavelets and filter banks*, Wellesley Colleg Press, Inc, 1996.

[Sta06] *Electronic statistics textbook*, `www.statsoft.com/textbook/stathome.html`, 2006.

[Sun96] W. Sun, *Block iterative algorithms for solving hermite bicubic collocation equations*, SIAM Journal of Numerical Analysis **33** (1996), 589–601.

[VSAI04] Sean F. McLoone Vijanth S. Asirvadam and George W. Irwin, *Memory efficient bfgs neural-network learning algorithms using mlp-network: A survey*, Proceedings of the 2004 IEEE International Conference on Control Applications (2004), 586–591.