1995

An application of Hamiltonian neurodynamics using Pontryagin's maximum (minimum) principle

Takamasa Koshizen
University of Wollongong

Follow this and additional works at: <https://ro.uow.edu.au/theses>

University of Wollongong

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following: This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author. Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material.

Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Recommended Citation

Koshizen, Takamasa, An application of Hamiltonian neurodynamics using Pontryagin's maximum (minimum) principle, Master of Science (Hons.) thesis, Department of Computer Science, University of Wollongong, 1995. <https://ro.uow.edu.au/theses/2791>

An Application of Hamiltonian Neurodynamics Using Pontryagin's Maximum (Minimum) Principle

A thesis submitted in fulfilment of the requirements for the award of the degree

Master of Science (Honours)

from

UNIVERSITY OF WOLLONGONG

by

**Takamasa Koshizen, Bachelor of Science in Physics
University of Ryukyu, Japan**

Department of Computer Science
1995

Abstract

Hamiltonians can generate Artificial Neural Dynamical systems dependent on time. Classical methods from optimal control theory, notably Pontryagin's Maximum (Minimum) principle (PMP) can be employed, together with Hamiltonians, in order to determine the optimal weights. Today, although several extended-backpropagation methods using optimization theory have been developed based on the well known standard backpropagation algorithm (SBP), feedforward multilayer perceptron (MLP) neural networks are here employed on differential equations which have characteristics such as admitting neurons and time dependent weight vectors . In this thesis, it is shown that the PMP learning rule obtained using PMP compares favourably with SBP. As a result, the PMP learning rule provides new results with feedforward networks; it can also be applied to recurrent networks, in both continuous-time and discrete-time.

Acknowledgments

I first would like to thank my supervisor, Mr. John Fulcher, for his technical suggestions and helps with this research generally as well as assistance with preparation of this thesis. For the mathematical advice, I wish to express my sincere appreciation to Professor Phil Broadbridge of the Department of Mathematics. Thanks are also due to Assoc. Professor Greg Doherty who assisted me with numerical techniques, and Minh Tam Tran for the C-code SBP implementation.

There are so many people that I met in the Internet Newsgroups during the course of this study. I also wish to express my appreciation to them, since this project could not have been completed without them.

Lastly, I cannot express enough thanks to my parents and to one of my deepest friends, Indriati Wonosoebekti, they provided encouragement and support throughout this project.

Contents

1	Introduction	1
1.1	Biological Foundation and Artificial Neural Networks (ANNs)	1
1.1.1	The structure of the brain	1
1.1.2	Artificial neural networks and the brain	2
1.2	Neurodynamics	4
1.3	Optimization Methods	4
1.3.1	First order derivative	5
1.3.2	Second order derivative	5
1.4	Thesis aims	6
1.5	Thesis Outline	6
2	Mathematical Background	8
2.1	Backpropagation	8
2.1.1	The Multilayer Perceptron (MLP)	8
2.1.2	MLPs and the Standard Backpropagation Algorithm (SBP)	10
2.1.3	Error Surfaces	11
2.1.4	The SBP learning algorithm	14
2.2	Extended Backpropagation	19
2.2.1	Conjugate Direction Methods (CD)	19
2.2.2	Conjugate Gradients and SBP	20
2.2.3	Newton's method	21

2.3	Optimal Control Problems	24
2.3.1	The state equations	25
2.3.2	The performance index	26
2.3.3	State variable representation of systems	27
2.3.4	Solution of the optimal control problem	29
2.3.5	The Hamilton-Jacobi Equation	29
2.3.6	The Pontryagin's Minimum Principle	34
2.3.7	The Discrete-time Maximum Principle	39

3	The New Learning algorithm based on Pontryagin's Principle	44
3.1	Optimal Control Problems and ANNs	44
3.1.1	The form of the state equations in continuous-time and discrete-time systems	44
3.1.2.	The performance index in Continuous-time ANN systems	46
3.1.3.	The performance index in Discrete-time ANN systems	47
3.2	Learning Potential Functions and Pontryagin's Principle	48
3.2.1	Squared error function	49
3.2.2	The learning potential function in Continuous-time and Discrete-time systems	50
3.2.3	Linear regulator problems and Resulting optimal weight equations	51
3.3	The New Learning algorithm	55

3.3.1	Optimal weight equations	56
3.3.2	Learning Dynamics and its equation	57
4	Results and Conclusion	60
4.1	Numerical Example and Performance Evaluation	60
4.1.1	The form of the second order derivative	60
4.1.2	The application of the continuous first order derivatives	61
4.1.3	Numerical Result	61
4.1.4	Discussion	68
4.2	Summary and Suggestions for Further work	68

APPENDIX

- Appendix A. Hamiltonian Control Systems and PMP
- Appendix B. PMP and the Hamiltonian-Jacobi Equation
- Appendix C. Proof of Equation (3.2.8)
- Appendix D. Program Code

BIBLIOGRAPHY

Chapter 1

Introduction

1.1 Biological Foundation and Artificial Neural Networks (ANNs)

Computers are far superior to humans when it comes to number crunching. However, computers often fall short when it comes to tasks where some sort of intelligence needs to be displayed. The human brain is a very complicated device, which has been studied in detail over many years. The low level operation of the brain is described below.

1.1.1 The structure of the brain

Figure 1-1(a) shows a biological neuron, the basic unit of the brain, which is a stand-alone analogue logical processing unit [1]. The neuron receives chemical signals from the synapses on its dendrites (inputs), and yields an electrical action potential which travels down the axon (output). The axon also branches and forms synapses on the dendrites and cell bodies of other neurons. Thus, small electrical pulses can be transmitted by electrochemical processes from one neuron to another. The synapses determine to what degree a signal from one neuron influences the activation of a postsynaptic neuron, more specifically the degree of signal amplification. The human brain can be perceived as a large number (10^{10} to 10^{11}) of neurons interconnected through a large number (10^{13} to 10^{14}) of synapses. Even though the transmission of a signal through a neuron is slow (1 to 10 ms), the massively parallel nature of the interconnections ensures that intensive

analog computation can be performed quickly. Furthermore, the biological brain system learns by association and is very tolerant to noise.

1.1.2 Artificial neural networks and the brain

An artificial neural network (ANN) is a mathematical idealisation of a biological neural network. The neurons are abstracted into processing elements, called nodes, the axons and dendrites into connections between the nodes, and the inhibitory or excitatory potential of a synapse into a scalar, called the weight, associated with each connection. An artificial neuron model is shown in Figure 1-1(b) [1]. Here, $x_1, x_2, \dots, x_n$ represent the inputs (voltages) received by neuron j , the w_{ji} 's are the synaptic strengths, called weights, and y_j the output of the neuron. The left half of the neuron model, containing the summation symbol, is an operational amplifier configured as an adder. The quantity f is any suitable nonlinear squashing function, which will be described later. Learning in a biological nervous system is achieved by adjusting the transmission rate of the synapses, whereas learning in an ANN is done by a process of finding a connectivity pattern and set of weights (weight space) which matches the mapping to be learned. Moreover, many ANN models group the individual nodes into layers.

We can conclude that the human brain is the most remarkable computing device known to man. The ANN approach to computing is to simulate or modernize the understanding of the brain at the microlevel, as discussed above. For an overview or detailed description of various ANN models, the reader is referred to [2] [3] [4].

(a) Biological neuron.

(b) Mathematical abstraction of neuron j .

Figure 1-1: Biological and mathematical neurons

1.2 Neurodynamics

In order to develop the science and technology of neural networks to their full potential, it will be important to cultivate a thorough understanding of neurodynamics [5]. In general, neurodynamics can be constructed by the network topology, input patterns, and weight dynamics attributed to learning, notably associative memory models as a function of time. Fixed point attractors serve as associative memory models typified by Hopfield [6], whereas [7][8][9] use applied oscillatory or chaotic attractors for this purpose. Moreover, [10] suggests that chaos, especially of the intermittent variety, might model an *idle* state in which the mind is prepared to trigger a wide variety of associative memories, which would correspond to the quasi-periodic parts of an intermittently chaotic attractor. Furthermore, the study of neurodynamics seems to be mixed with several optimization methods derived from an understanding of classical optimal control problems, such as first-order and second-order algorithms and dynamical programming methods [11] [12].

1.3 Optimization Methods

Learning methods in ANNs can basically be regarded as optimization methods. Training a fixed size network is a form of numerical optimization where the purpose is to minimize or maximize an objective or a cost (error) function ψ . The objective function ψ at a given point $\mathbf{w} + \Delta\mathbf{w}$ in weight space can be expressed by its Taylor series expansion, as follows :

$$\psi|_{\mathbf{w}+\Delta\mathbf{w}} = \psi|_{\mathbf{w}} + \frac{\partial\psi}{\partial\mathbf{w}}|_{\mathbf{w}}^T \Delta\mathbf{w} + \frac{1}{2} \Delta\mathbf{w}^T \frac{\partial^2\psi}{\partial\mathbf{w}\partial\mathbf{w}^T}|_{\mathbf{w}} \Delta\mathbf{w} + \dots \quad (1.3.1)$$

The matrix $\partial^2\psi / \partial\mathbf{w}\partial\mathbf{w}^T$ is called the Hessian matrix.

Numerical analysis has traditionally focused on methods using not only the local gradient of the function but also the second derivative. In recent times, the backpropagation method, which is widely used for learning in multilayered neural networks, uses steepest gradient-descent minimization of a global error function.

In summary, learning algorithms can be classified by their order. Training a network can be viewed as optimizing a cost function ψ . First order methods only use the first derivative of the function in order to perform this optimization. Second order methods are characterized by the utilization of information about the curvature of the function (second derivative) as well as the gradient (first derivative).

1.3.1 First order derivative

First order optimisation methods (steepest descent) only use the first two terms of equation (1.3.1) to approximate ψ around the current point in weight space.

1.3.2 Second order derivative

Second order optimisation methods make use of an additional second order term involving the Hessian matrix, thus locally approximating ψ with a paraboloid instead of a plane. The Hessian answers the question of how the gradient changes if one moves from the current point in a certain direction, and may be useful for calculating both search direction and step size [13]. Actually, there are several techniques that can implicitly calculate the Hessian, rather than calculating it explicitly [13][14]. In addition, several second order methods approximate the Hessian by differentiating the gradient between successive steps.

1.4 Thesis aims

Of the existing training algorithms for feedforward neural networks, we first discuss the standard backpropagation algorithm (SBP) in Chapter 2. The disadvantages of this algorithm is that it converges very slowly, behaves very “badly” on large-scale problems, and can become stuck in local minima [15][16]. Therefore, an important direction of research is to devise learning schemes for neural networks that are considerably faster than conventional SBP.

The main approach to developing fast methods for training ANNs is to consider a fixed size network and to focus on methods for minimizing (or maximizing) a given objective function (error cost function) ψ as fast as possible.

- The first aim of this thesis is to develop a new learning algorithm for determining optimal weight equations based on classical optimal control theory.
- The second aim is to compare the new learning algorithm with traditional SBP.

1.5 Thesis outline

The structure of the thesis is as follows:

Firstly, Chapter 2 defines and derives the learning algorithm known as SBP. This is followed by an extension of SBP. A brief presentation follows of feedforward multilayer perceptrons (MLP), a network topology used in later implementation work. This chapter also describes Classical Optimal Control theory, more specifically, Pontryagin's Minimum (Maximum) principle

(PMP) [17], used later in this thesis to derive important characteristics of weight solutions.

Chapter 3 describes how a new learning algorithm can be derived for obtaining weight equations, using PMP. In addition, mathematical definitions of the weight equations for both continuous- and discrete-time ANN systems are presented using PMP.

Chapter 4 contains results from comparative simulations of both SBP and the new PMP learning method developed in Chapter 3. These comparative simulations show that convergence to a given tolerance using the PMP learning algorithm is significantly faster (approximately 1.5 times) than with SBP. This chapter also describes how to choose second order derivatives (the Hessian) in quadratic form, such that only *one* iteration is needed to reach the global minimum.

Chapter 5 draws conclusions based on the results presented in the earlier chapters, and suggests several avenues for further research.

Formal mathematical suggestion and proof are presented in the appendices.

Chapter 2

Mathematical Background

2.1 Standard Backpropagation

Multilayer, feedforward networks are currently the most widely used neural network architecture [18]. This section describes the standard backpropagation (SBP) learning algorithm used in such networks, but firstly we examine the history and evolution of SBP.

Originally, SBP was introduced by Bryson and Ho in 1969 [18] and independently rediscovered by Werbos in 1974, by Parker in the mid 1980's and by Rumelhart, Williams and other members of the PDP group in 1985 (the work of Bryson and Ho was reported in 1988 by le Cun [18]). SBP is also known as the generalized delta rule.

2.1.1 The Multilayer Perceptron

This section explores aspects of the feedforward multilayer perceptron (MLP), in particular the modifications needed to solve complex problems such as XOR. The 3-2-4 network of Figure 2.1 is referred to as a three layer network; it consists of an input layer, an output layer, and a *hidden* layer. The network is also fully connected with weights initialized to small random values prior to learning.

In feedforward MLPs, connections are formally allowed to be made between any node in layer Ω and any node in layer Ξ , as long as $\Omega < \Xi$ and there are

no resulting feedback loops. This means that units in each respective layer distribute the values they receive to the next layer.

Figure 2-1: Three-Layer Back Propagation with Feed-forward network (3-2-4)

The MLPs used in subsequent comparative studies in this thesis are assumed to have hidden nodes with symmetrical sigmoidal activation functions (Figure 2-2). The input nodes are always assumed to be linear, and we can regard these as performing online data preprocessing. For mapping tasks, the output nodes are usually taken to be linear.

Figure 2-2 Nonlinear activation functions. Linear (A); sigmoid (B)

The mathematical neuron used in MLPs comprises n inputs (i.e., input vector $\mathbf{x} = (x_1, \dots, x_n)$) and a single output z . $\mathbf{w} = (w_1, \dots, w_n)$ are the weights (weight vector) applied to the inputs, h is a threshold, and $f: \mathbf{R} \rightarrow \mathbf{R}$, acts as the non linear sigmoidal activation function. The neuron's input-output relation is given by the following equation :

$$output = f\left(\sum_{i=1}^n w_i x_i - h\right) \quad (2.1.1)$$

with the sigmoidal activation function :

$$f(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{x})} \quad (2.1.2)$$

varying smoothly from 0 at $-\infty$ to 1 at ∞ . This nonlinear activation function (2.1.2) provides two computationally useful properties, in contrast to linear systems such as $f(\mathbf{x}) = \mathbf{x}$ [19]. One of these properties is a guarantee that the node outputs remain within a bounded range (0,1). The other is the ability to approximate a very general class of system. It can also be shown that a feedforward neural network model can approximate *any* mapping, given enough hidden nodes [20].

2.1.2 MLPs and the standard backpropagation algorithm

In general, the standard backpropagation (SBP) method is applied to feedforward MLP neural networks in order to evaluate the derivatives of the network error with respect to its connections strengths (weights). The basic component in a feedforward network is the single "neuron" model of Section 2.1.1. The feedforward neural network is constructed by interconnecting several such neurons so as to form a network in which all connections are made only in the forward direction from input to output, but without feedback (recurrent) loops. Such networks are described by the following equations, for the discrete-time case :

$$x_j(t+1) = f(y_j) \quad (2.1.3)$$

where

$$y_j = \sum_{i=1}^n w_{ji} x_i(t), \quad (2.1.4)$$

based on (2.1.1) with $h = 0$ (Figure 2-3). (2.1.4) is the total input to the j -th neuron; where x_i is the output from the i -th neuron in the previous layer.

The thresholding function is alternatively known as the "step", or "Heaviside" function

Figure 2-3: The thresholding function

2.1.3 Error Surfaces

First, we define the function $\psi_p(\mathbf{w})$, called the mean squared error function of the network:

$$\psi_p(\mathbf{w}) \equiv \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{l=1}^N \psi_p^l \quad (2.1.5)$$

with

$$\psi_p^l = \left| f(\mathbf{x}_p^l) - \Theta(\mathbf{x}_p^l, \mathbf{w}) \right|^2 \quad (2.1.6)$$

where, l denotes layer number and Θ is a function of the input vector x , and the network weight vector w . In addition, ψ_p^l is the square of the approximation error made on the l -th iteration and p is the number of training patterns.

Notice that $\psi(w) \geq 0$, since ψ is the average of non-negative quantities [18].

The error surface of SBP is the surface defined by the equation $\psi = \psi(w)$ in the $(\Gamma + 1)$ -dimensional space of vectors (w, ψ) , where Γ is the dimension of the vector w . The variable w ranges over its Γ -dimensional space, and for each Γ a non-negative surface height ψ is defined by $\psi(w)$.

Now if we consider the shape of this error surface, SBP (or generalized delta rule) has the property that, given any starting point w_0 on the error surface that is not a minimum, the learning rule will modify the weight vector w so that $\psi(w)$ decreases.

Figure 2-4 shows the error surface (2.1.5) for a single (symmetrical) sigmoidal node attempting to learn the mapping $x \rightarrow \sin(x)$ for 32 different values of x uniformly distributed in the interval $[0, 2\pi]$ [21]. Even though the error surface in Figure 2-4 is from a single node, some of the properties of the depicted surface are believed to carry over to higher dimensions [21]. It is not unusual for error surfaces of the kind represented by equation (2.1.5) to contain large plateaus, cliffs, troughs and ridges [22].

Now backpropagation error surfaces will contain local minima at error levels above the levels of the global minima for that surface. However finding the global minima is not easy. Moreover, several techniques exist for avoiding local minima. The existence of the global minimum can be proven by showing that all of the first order partial derivatives of the mean squared error function reach zero at a fixed point, and the Hessian (the

matrix of second order partial derivatives) is strongly positive-definite at this fixed point.

Figure 2-4: Error surface for a single symmetrical sigmoidal node
From Aleksander Ohm (1993); reprinted with permission

2.1.4 The SBP learning algorithm

Here we describe the SBP learning algorithm, which in its most common form uses gradient descent. To assist in the derivation of this learning algorithm, we introduce the following notation:

- x_j^l : output of the j -th node in layer l
- w_{ji}^l : weight which connects the j -th node in layer $l-1$ to i -th node in layer l
- $\mathbf{u}_p$: p -th training sample
- x_{0i} : i -th component of the input vector
- $d_j(\mathbf{u}_p)$ desired response of the j -th output node for the p -th training sample
- N_l : number of nodes in layer l
- L : number of layers
- P : number of training patterns

For convenience we let the 0-th layer of the network hold the input vector components i.e., in our notation $x_{0i} = x_j$, where x_j is the j -th component of the current input vector. Moreover, we define the 0th component of the input vector to each layer to be equal to 1, i.e., $x_0^l = 1$ and w_{j0}^l = the bias weight.

We define the output of a node in layer l by

$$x_j^l = f\left(\sum_{i=0}^{N_{l-1}} w_{ji}^l x_i^{l-1}\right) \quad (2.1.7)$$

where, $f(\cdot)$ is the sigmoid nonlinearity. This function has a simple derivative :

$$f'(\sigma) = \frac{df(\sigma)}{d\sigma} = f(\sigma)(1 - f(\sigma)) \quad (2.1.8)$$

The most common learning algorithm for the MLP uses a gradient search technique, in order to find the network weights that minimize an objective function. The objective function to be minimized is the following Sum-of-Squared-Error cost function :

$$\psi(\mathbf{w}) = \sum_{p=1}^P \psi_p(\mathbf{w}) \quad (2.1.9)$$

where, $\psi_p(\mathbf{w})$ is the total squared error for the p -th pattern (2.1.5) :

$$\psi_p(\mathbf{w}) = \frac{1}{2} \sum_{q=1}^{N_L} (x_q^L(\mathbf{u}_p) - d_q(\mathbf{u}_p))^2 \quad (2.1.10)$$

and N_L is the number of nodes in the output layer as defined above.

The weights of the network are determined iteratively :

$$\begin{aligned} w_{ji}^l(k+1) &= w_{ji}^l(k) - \mu \left. \frac{\partial \psi(\mathbf{w})}{\partial w_{ji}^l} \right|_{\mathbf{w}(k)} \\ &= w_{ji}^l(k) - \mu \sum_{p=1}^P \left. \frac{\partial \psi_p(\mathbf{w})}{\partial w_{ji}^l} \right|_{\mathbf{w}(k)} \end{aligned} \quad (2.1.11)$$

where μ is a positive constant, called the learning rate. In order to implement this algorithm we need to develop an expression for the partial derivative of ψ_p with respect to each weight in the network.

For with an arbitrary weight in layer l this can be obtained using the following chain rule :

$$\frac{\partial \psi_p(\mathbf{w})}{\partial w_{ji}^l} = \frac{\partial \psi_p(\mathbf{w})}{\partial x_j^l} \frac{\partial x_j^l}{\partial w_{ji}^l} \quad (2.1.12)$$

where,

$$\begin{aligned}
\frac{\partial x_j^l}{\partial w_{ji}^l} &= \frac{\partial}{\partial w_{ji}^l} \left[f \left(\sum_{m=0}^{N_{l-1}} w_{jm}^l x_m^{l-1} \right) \right] \\
&= f' \left(\sum_{m=0}^{N_{l-1}} w_{jm}^l x_m^{l-1} \right) \frac{\partial}{\partial w_{ji}^l} \left[\sum_{m=0}^{N_{l-1}} w_{jm}^l x_m^{l-1} \right] \\
&= f' \left(\sum_{m=0}^{N_{l-1}} w_{jm}^l x_m^{l-1} \right) x_i^{l-1}
\end{aligned} \tag{2.1.13}$$

Substituting from (2.1.8) for the first term, we obtain

$$\frac{\partial x_j^l}{\partial w_{ji}^l} = x_j^l (1 - x_j^l) x_i^{l-1} \tag{2.1.14}$$

Thus, (2.1.12) becomes

$$\frac{\partial \psi_p(\mathbf{w})}{\partial w_{ji}^l} = \frac{\partial \psi_p(\mathbf{w})}{\partial x_j^l} x_j^l (1 - x_j^l) x_i^{l-1} \tag{2.1.15}$$

The term $\frac{\partial \psi_p(\mathbf{w})}{\partial x_j^l}$ represents the *sensitivity* of $\psi_p(\mathbf{w})$ to the output of node x_j^l . The node x_j^l exhibits its influence on ψ_p through all of the nodes in the succeeding layer. Therefore, $\frac{\partial \psi_p(\mathbf{w})}{\partial x_j^l}$ can be expressed as a function of the *sensitivities* to nodes in the next highest layer as follows

$$\begin{aligned}
\frac{\partial \psi_p(\mathbf{w})}{\partial x_j^l} &= \sum_{m=1}^{N_{l+1}} \frac{\partial \psi_p(\mathbf{w})}{\partial x_m^{l+1}} \frac{\partial x_m^{l+1}}{\partial x_j^l} \\
&= \sum_{m=1}^{N_{l+1}} \frac{\partial \psi_p(\mathbf{w})}{\partial x_m^{l+1}} \frac{\partial}{\partial x_j^l} \left[f \left(\sum_{q=0}^{N_l} w_{mq}^{l+1} x_q^l \right) \right]
\end{aligned} \tag{2.1.16}$$

$$\begin{aligned}
&= \sum_{m=1}^{N_{l+1}} \frac{\partial \psi_p(\mathbf{w})}{\partial x_m^{l+1}} f' \left(\sum_{q=0}^{N_l} w^{l+1}_{mq} x_q^l \right) \frac{\partial}{\partial x_j^l} \left[\sum_{q=0}^{N_l} w^{l+1}_{mq} x_q^l \right] \\
&= \sum_{m=1}^{N_{l+1}} \frac{\partial \psi_p(\mathbf{w})}{\partial x_m^{l+1}} x_m^{l+1} (1 - x_m^{l+1}) w^{l+1}_{mj}
\end{aligned}$$

This process can be continued for $\frac{\partial \psi_p(\mathbf{w})}{\partial x_j^{l+1}}$ and so on, until we reach the output layer. At the output layer we reach a boundary condition where the sensitivities of the nodes in the last layer are derived from (2.1.10):

$$\frac{\partial \psi_p(\mathbf{w})}{\partial x_j^L} = x_j^L(\mathbf{u}_p) - d_j(\mathbf{u}_p) \quad (2.1.17)$$

This is called the output error. The corresponding expression for hidden layer nodes in (2.1.16) is often referred to as the hidden layer error.

Equation (2.1.15) seems to be working its way forward to the output layer, however, the sensitivity of a node is actually computed from the output layer backwards. Since the "hidden layer error" is computed from the output layer backwards, it has historically been called the backpropagation error, and the learning algorithm the (standard) *Backpropagation Algorithm*.

The results in (2.1.15)-(2.1.17) can be combined with (2.1.11) in order to implement a gradient search. Typically, the summation in (2.1.11) is replaced with an estimate of the gradient based on a single sample. This means, $\frac{\partial \psi(\mathbf{w})}{\partial w^l_{ji}}$ is approximated such that (2.1.11) becomes [21] :

$$w^l_{ji}(k+1) = w^l_{ji}(k) - \mu \frac{\partial \psi_{k \bmod p(\mathbf{w})}(\mathbf{w})}{\partial w^l_{ji}} \Big|_{\mathbf{w}(k)} \quad (2.1.18)$$

where $(k \bmod P)$ is the index of the pattern used to estimate the gradient at the k -th iteration.

(2.1.15)-(2.1.18) comprise the Backpropagation learning algorithm. The weights are initialized to small random values. Moreover, the learning rates can be chosen in a number of different ways. They can be the same for every weight in the network, different for each layer, different for each node, or different for each weight in the network. It is generally difficult to determine the best learning rate, but a useful rule of thumb is to make the learning rate for each node inversely proportional to the average magnitude of vectors feeding into the node[21]. In fact several attempts have been made to adapt the learning rate as a function of the local curvature of the surface[23][24][25]. The simplest approach is to add the following momentum term :

$$\alpha(w'_{ji}(k) - w'_{ji}(k-1))$$

to each weight update, where $0 < \alpha < 1$. This term also determines the effect of past weight changes on the current direction of movement in weight space. This provides a sort of momentum in weight space that effectively filters out high frequency variations of the error surface [3]. This is useful in spaces containing long ravines that are characterized by sharp curvature across the ravine and a gently sloping floor. In areas of the weight space where the gradient is approximately constant, adding the momentum term has the same effect as increasing the step size by a factor of $\frac{1}{1-\alpha}$ [26]. Consequently, adding momentum of the following form (2.1.19) usually speeds convergence.

$$\Delta \mathbf{w}(k) = -\mu \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)} + \alpha \Delta \mathbf{w}(k-1) \quad (2.1.19)$$

2.2 Extended Backpropagation

Numerical analysis has traditionally focused on methods using not only the local gradient of the function as described in Section 2.1, but also its second derivative. Standard backpropagation of the error gradient function $\psi_p(\mathbf{w})$ can be extended using the second derivative, as in conjugate direction and Newton Methods. Now it is well known that such extended backpropagation algorithms require much shorter training times compared with steepest gradient descent. We firstly consider the conjugate gradient method.

2.2.1 Conjugate Direction Methods (CD)

Recently, there has been a focus on training feedforward neural networks with conjugate (gradient) direction methods. References [28], [29] and [30] describe the theory of general conjugate gradient methods and how to apply the methods in feedforward neural networks. In addition, they conclude that the conjugate method is an order of magnitude faster than the SBP when tested on the parity problem.

Conjugate direction (CD) methods can be regarded as being somewhere between steepest descent and Newton's method. Their use is motivated by a desire to accelerate the slow convergence associated with steepest descent, while avoiding the storage and computation overhead associated with the Hessian matrix [31]. Unlike steepest descent, CD methods retain some memory of the directions they have previously searched in. At each successive step, CD methods choose a downwards direction on the error surface, at the same time preventing them going upwards in directions it has previously searched. This means the direction is always chosen such that the minimization steps in all previous directions are not spoiled.

2.2.2 Congugate gradients and SBP

A geometrical interpretation of the method of conjugate gradients (CG) might aid in understanding the algorithm. The idea is to assume a quadratic surface ψ in (1.3.1) and to transform the weight space $\mathbf{W}$ into a space $\Sigma(\mathbf{W})$ which causes elliptical contour curves of ψ in $\mathbf{W}$ to become scalar in $\Sigma(\mathbf{W})$. This transformation Σ is determined by the Hessian matrix $\mathbf{H}$ as follows

$$\Sigma(\mathbf{w}) = \sqrt{\mathbf{H}}\mathbf{w} \quad (2.2.1)$$

The CG algorithm also allows one to iteratively develop an orthogonal basis for $\Sigma(\mathbf{W})$, which results in a more direct path towards the minimum of ψ than with ordinary steepest descent.

Any point $\mathbf{w}$ can be written as a linear combination of the basis vectors. More specifically, a difference vector between two points can be written as :

$$\mathbf{w}^* = \mathbf{w} + \sum_{k=0}^{n-1} \mu(k)\mathbf{d}(k) \quad (2.2.2)$$

where, $\mathbf{d}(k); 0 \leq k \leq n-1$ form a basis of $\mathcal{R}^n$ (the Euclidean space).

Equation (2.2.2) can be viewed as an iterative algorithm to compute a solution point $\mathbf{w}^*$ from a starting point $\mathbf{w}$, each step of which consists of adding a term $\mu(k)\mathbf{d}(k)$. The vector $\mathbf{d}(k)$ represents the search direction at step k , whereas $\mu(k)$ corresponds to the step size.

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mu(k)\mathbf{d}(k) \quad (2.2.3)$$

Now this process is guaranteed to converge to the solution $\mathbf{w}^*$ within n steps.

If the error surface ψ is quadratic, the above CG algorithm can recursively yield expressions for $\mathbf{d}(k)$ that lead to a minimum of ψ in at most n steps. Furthermore, the search directions are conjugate :

$$\mathbf{d}(i)^T \mathbf{H} \mathbf{d}(j) = 0 \quad (\text{for all } i \neq j) \quad (2.2.5)$$

Equation (2.2.5) also implies that $\Sigma(\mathbf{d}(i))$ and $\Sigma(\mathbf{d}(j))$ are orthogonal. The vectors $\mathbf{d}(k)$ are determined by the following :

$$\mathbf{d}(k) = \begin{cases} -\frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)} & (k = 0) \\ -\frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)} + \beta(k-1) \mathbf{d}(k-1) & \text{otherwise} \end{cases} \quad (2.2.6)$$

where $\beta(k-1)$ is a scalar value which acts like an adaptive momentum term α in (2.1.19), and which can take on different forms [31].

Now the following two rules apply to the coefficient $\beta(k-1)$, the first being the Polak-Ribiere rule [32], and the second the Hestenes-Stiefel rule [33].

(1) The Polak-Ribiere rule:

$$\beta(k-1) = \frac{\left(\frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)} - \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k-1)} \right)^T \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)}}{\frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k-1)}^T \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k-1)}} \quad (2.2.7)$$

(2) The Hestenes-Stiefel rule:

$$\beta(k-1) = \frac{\left(\frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)} - \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k-1)} \right)^T \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)}}{\mathbf{d}(k-1)^T \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k-1)}} \quad (2.2.8)$$

The factor $\mu(k)$ in equation (2.2.3) can be approximated by a line search :

$$\mu(k) = -\frac{\mathbf{d}(k)^T}{\mathbf{d}(k)^T \mathbf{H}|_{\mathbf{w}(k)} \mathbf{d}(k)} \frac{\partial \psi}{\partial \mathbf{w}} \Big|_{\mathbf{w}(k)} \quad (2.2.9)$$

Now (2.2.9) is usually avoided due to the effort involved in finding the Hessian. However, techniques for computing the product $\mathbf{H}|_{\mathbf{w}(k)} \mathbf{d}(k)$ in $O(n)$ time have recently surfaced [14].

It can be immediately concluded that (2.2.6)-(2.2.9) implement the standard backpropagation learning method described in (2.1.18) through (2.1.19), with learning parameter μ and momentum β chosen such that subsequent search directions are conjugate [28]. Therefore, CG is an extension of SBP with momentum, which uses second-order information. In addition to CG, a version also exists in which the one-dimensional minimization is replaced by a scaling of the step size that depends on success in error reduction and goodness of fit of a one-dimensional quadratic approximation -this is Moller's Scaled CG [33].

2.2.3 Newton's method

Newton's method can be considered as the basic local method which uses second-order information. It is usually used for finding roots of equations, but can also be used for minimization (if we view minimizing ψ as finding a root of $\frac{\partial \psi}{\partial \mathbf{w}}$). The method is based on approximating ψ around the current point in weight space by a paraboloid in (1.3.1). The next point in weight space is then set directly to the minimum of the estimated paraboloid.

It is important to emphasize that its applicability to MLP is hampered by the fact that it requires calculation of the Hessian matrix.

The above process is repeated until some stopping criterion is reached. This results in the following iterative algorithm :

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \mathbf{H}^{-1} \left|_{\mathbf{w}(k)} \frac{\partial \psi}{\partial \mathbf{w}} \right|_{\mathbf{w}(k)} \quad (2.2.10)$$

where, $\mathbf{H}^{-1}$ is the inverse Hessian matrix.

If ψ has continuous second-order partial derivatives and the Hessian $\mathbf{H}$ is positive definite at points near a minimum point $\mathbf{w}^{\min}$, then the method is well defined near the solution $\mathbf{w}^{\min}$ [31].

It can be concluded that Newton's method is very unstable in the general case, but if the criterion that the starting point is sufficiently close to the solution point is met, it converges quadratically [31]. In addition, for certain problems, the set of points from which Newton's method converges to a solution has a fractal shape [32].

2.3 Optimal Control Problems

Optimal control problems have received a great deal of attention since the early 1960s. They often involve finding a control decision rule with respect to certain constraints, which minimizes some deviation from ideal behaviour. Such a measure is commonly known as a performance criterion or *performance index*. Performance index of an optimal control system is a measure or indicator of the cumulative deviation of the system from the desired or ideal state. When the performance index involves the economic consequences of a given control, the index is called the *cost function*.

The general structure of an optimal control system is straightforward. In the simplest version, there is a given dynamical system (linear or nonlinear, discrete-time or continuous-time) for which input functions can be specified. Solving optimal control problems involves selecting the input function so as to optimize (maximize or minimize) the performance index.

In formulating an optimal control problem, the quantities appearing in the optimization process are state variables, control variables, and system parameters. Briefly, an optimal control problem can be formulated using the following information:

- (a) state and output
- (b) control variables or vector
- (c) problem constraints
- (d) performance index
- (e) system parameters

In addition, a mathematical definition of an optimal control problem can be expressed in either continuous or discrete-time form.

2.3.1 The state equations

A nontrivial part of any control problem is modelling the process. The objective is to obtain the simplest mathematical description that adequately predicts the response of the physical system to all anticipated inputs. First we define the state variables of the process at time t . Our discussion here is restricted to systems described by the continuous-time case, because continuous-time problems are notationally simpler than discrete-time ones. However, discrete-time problems can be readily extended to continuous-time problems.

If

$$x_1(t), x_2(t), \dots, x_n(t)$$

are the state variables of the process at time t , and

$$w_1(t), w_2(t), \dots, w_m(t)$$

are the control inputs to the process at time t , then the system may be described by n first-order differential equations:

$$\begin{aligned} \dot{x}_1(t) &= g_1(x_1(t), x_2(t), \dots, x_n(t), w_1(t), w_2(t), \dots, w_m(t)) \\ \dot{x}_2(t) &= g_2(x_1(t), x_2(t), \dots, x_n(t), w_1(t), w_2(t), \dots, w_m(t)) \\ &\vdots \\ \dot{x}_n(t) &= g_n(x_1(t), x_2(t), \dots, x_n(t), w_1(t), w_2(t), \dots, w_m(t)) \end{aligned} \tag{2.3.1}$$

We define

$$\mathbf{x}(t) \equiv \begin{bmatrix} x_1(t) \\ x_2(t) \\ \bullet \\ \bullet \\ \bullet \\ x_n(t) \end{bmatrix}$$

as the state vector of the system, and

$$\mathbf{w}(t) \equiv \begin{bmatrix} w_1(t) \\ w_2(t) \\ \bullet \\ \bullet \\ \bullet \\ w_n(t) \end{bmatrix}$$

as the control vector. The state equations can thus be written

$$\dot{\mathbf{x}}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{w}(t)) \quad (2.3.1)'$$

where $\mathbf{g}$ is defined in (2.3.1).

2.3.2 The performance index

In order to evaluate the performance of a system quantitatively, the designer selects a performance index measure. An optimal control is defined as one that minimizes (or maximizes) the performance index measure.

In all that follows it will be assumed that the performance of a system is evaluated by a measure of the following form :

$$J = h(\mathbf{x}(t_f)) + \int_{t_0}^{t_f} \psi(\mathbf{x}(t), \mathbf{w}(t))dt \quad (2.3.2)$$

where t_0 and t_f are the initial and final time; t_f may be specified or "free" depending on the problem statement [34].

Starting from a fixed initial state $\mathbf{x}(t_0) = \mathbf{x}_0$ and applying a control vector $\mathbf{w}(t)$ with respect to $t \in [t_0, t_f]$ causes a system to follow some state trajectory. This means the performance index assigns a unique real number to each trajectory of the system.

2.3.3 State variable representation of systems

The starting point for optimal control problems is a mathematical model in state variable form. In this section, we need to summarize the results and notation to be used in the subsequent discussion.

Why use state variables?

Having the mathematical model in state variable form is convenient because :

1. Differential equations are ideally suited for digital or analog solution.
2. The state form provides a unified framework for the study of both nonlinear and linear systems.
3. The state variable form is invaluable in theoretical investigations.
4. The concept of state has a strong physical motivation.

Definition of State of a system

In referring to the state of a system, the following definition is needed:

The state of a system is a set of quantities $x_1(t), x_2(t), \dots, x_n(t)$ which if known at $t = t_0$ are determined for $t \geq t_0$ by specifying the inputs to the system for $t \geq t_0$.

System Classification

Systems are described by the terms linear, nonlinear, time-invariant, and time-varying. We classify systems based on the form of their state equations.

Nonlinear, time-invariant systems are generally represented by state equations of the following form :

$$\dot{\mathbf{x}}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{w}(t)) \quad (2.3.3)$$

If a system is linear and time-varying, its state equations are

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{w}(t) \quad (2.3.4)$$

where $\mathbf{A}(t)$ and $\mathbf{B}(t)$ are $n \times n$ and $n \times m$ matrices with time-varying elements. Similarly, state equations for *linear, time-invariant* systems have the form :

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{w}(t) \quad (2.3.5)$$

where $\mathbf{A}$ and $\mathbf{B}$ are constant matrices.

2.3.4 Solution of the optimal control problem

In this section, the optimal control problem for a continuous-time system, defined by (2.3.1)' to (2.3.2) is solved. Both sufficient conditions (the Hamilton-Jacobi equation) and necessary conditions (the Minimum Principle) are used to obtain the solutions. Because of the fact that the derivation of the Minimum Principle is, in general, lengthy [35], we provide a complete derivation of the Hamilton-Jacobi equation first and then simply state the Minimum Principle.

2.3.5 The Hamilton-Jacobi Equation

We consider the system described by (2.3.1)' and its initial condition :

$$\mathbf{x}(t_0) = \mathbf{x}_0 \quad (2.3.6)$$

It is desired to find the optimal control $\mathbf{w}^{OPT}(t)$, $t_0 \leq t \leq t_f$, which minimizes the cost function (2.3.2), assuming that $\mathbf{g}(\cdot)$, $\psi(\cdot)$, and $h(\cdot)$ are smooth functions of their arguments. Other than this restriction, $\mathbf{g}(\cdot)$ can be arbitrary and $\psi(\cdot)$ and $h(\cdot)$ nonnegative to reflect their physical meaning in the optimal control formulation [35].

We provide the following definition:

$$J^*[\mathbf{x}(t)] = \min J[\mathbf{x}(t), \mathbf{w}(t)] \quad (2.3.7)$$

where $J^*(\cdot)$ represents the minimum value of $J(\cdot)$ which we are required to find. Notice that $J^*[\mathbf{x}(t)]$ is independent of $\mathbf{w}(t)$, because the knowledge of $\mathbf{x}(t_0)$ and t intuitively determine $\mathbf{w}(t)$, by the requirement that the control minimizes $J[\mathbf{x}(t), \mathbf{w}(t)]$. We can evaluate (2.3.7) for all t and $\mathbf{x}(t)$ to find the optimal control which minimizes $J(\cdot)$. Assuming that an optimum J^*

exists in terms of $\mathbf{x}(t)$ and t , together with the optimal control, we solve the optimization problem defined by (2.3.1)' and (2.3.2) by setting $t_0 = t$.

For an arbitrary value of $t_0 \leq t \leq t_f$ we can describe the optimal value J^* in (2.3.7) by

$$\begin{aligned} J^*[\mathbf{x}(t)] &= \min_{\mathbf{w}(t, t_f)} \left[\int_t^{t_f} \psi[\mathbf{x}(\tau), \mathbf{w}(\tau)] d\tau + h[\mathbf{x}(t_f)] \right] \\ &= \min_{\mathbf{w}(t, t_1)} \left[\int_t^{t_1} \psi[\mathbf{x}(\tau), \mathbf{w}(\tau)] d\tau + \int_{t_1}^{t_f} \psi[\mathbf{x}(\tau), \mathbf{w}(\tau)] d\tau + h[\mathbf{x}(t_f)] \right] \\ &= \min_{\mathbf{w}(t, t_1)} \left\{ \int_t^{t_1} \psi[\mathbf{x}(\tau), \mathbf{w}(\tau)] d\tau + \min_{\mathbf{w}(t_1, t_f)} \left[\int_{t_1}^{t_f} \psi[\mathbf{x}(\tau), \mathbf{w}(\tau)] d\tau + h[\mathbf{x}(t_f)] \right] \right\} \end{aligned} \quad (2.3.8)$$

Thus, (2.3.8) can be described as follows

$$J^*[\mathbf{x}(t)] = \min_{\mathbf{w}(t, t_1)} \left[\int_t^{t_1} \psi[\mathbf{x}(\tau), \mathbf{w}(\tau)] d\tau + J^*[\mathbf{x}(t_1)] \right] \quad (2.3.9)$$

Here, if we let $t_1 = t + \Delta t$ in (2.3.9), for small Δt and expand its right-hand side by a Taylor's series expansion, then we obtain :

$$\begin{aligned} J^*[\mathbf{x}(t)] &= \min_{\mathbf{w}(t, t+\Delta t)} \left\{ \Delta t \psi[\mathbf{x}(t + a\Delta t), \mathbf{w}(t + a\Delta t)] + J^*[\mathbf{x}(t)] \right. \\ &\quad \left. + \left[\frac{\partial J^*}{\partial \mathbf{x}}[\mathbf{x}(t)] \right] \frac{d\mathbf{x}(t)}{dt} \Delta t + \frac{\partial J^*}{\partial t}[\mathbf{x}(t)] \Delta t + O(\Delta t)^2 \right\} \end{aligned} \quad (2.3.10)$$

where $0 \leq a \leq 1$ is a constant.

Recalling (2.3.1) and solving for the last derivative term, this becomes :

$$\begin{aligned} \frac{\partial J^*}{\partial t}[\mathbf{x}(t)] = & - \min_{\mathbf{w}[t, t+\Delta t]} \left\{ \psi[\mathbf{x}(t+a\Delta t), \mathbf{w}(t+a\Delta t)] + \right. \\ & \left. + \left[\frac{\partial J}{\partial \mathbf{x}}[\mathbf{x}(t)] \right]^T \mathbf{g}[\mathbf{x}(t), \mathbf{w}(t)] + O(\Delta t) \right\} \end{aligned} \quad (2.3.11)$$

Now, if Δt approaches zero, then (2.3.11) becomes :

$$\frac{\partial J^*}{\partial t}[\mathbf{x}(t)] = - \min_{\mathbf{w}(t)} \left\{ \psi[\mathbf{x}(t), \mathbf{w}(t)] + \left[\frac{\partial J^*}{\partial \mathbf{x}}[\mathbf{x}(t)] \right]^T \mathbf{g}[\mathbf{x}(t), \mathbf{w}(t)] \right\} \quad (2.3.12)$$

Here, J^* is the unknown quantity, whereas $\mathbf{g}$ and ψ are known functions. Moreover, this is one form of the Hamilton-Jacobi equation. In fact, we can define the Hamiltonian H as follows :

$$H(\mathbf{x}(t), \mathbf{w}(t), \frac{\partial J^*}{\partial \mathbf{x}}) \equiv \psi[\mathbf{x}(t), \mathbf{w}(t)] + \left[\frac{\partial J^*}{\partial \mathbf{x}}[\mathbf{x}(t)] \right]^T \mathbf{g}[\mathbf{x}(t), \mathbf{w}(t)]$$

Thus, (2.3.12) can be represented as follows

$$\frac{\partial J^*}{\partial t}[\mathbf{x}(t)] = - \min_{\mathbf{w}(t)} H(\mathbf{x}(t), \mathbf{w}(t), \frac{\partial J^*}{\partial \mathbf{x}})$$

or

$$0 = \frac{\partial J^*}{\partial t}[\mathbf{x}(t)] + \min_{\mathbf{w}(t)} H(\mathbf{x}(t), \mathbf{w}(t), \frac{\partial J^*}{\partial \mathbf{x}}) \quad (2.3.12)'$$

This equation is the well known continuous-time form of the Hamilton-Jacobi equation.

Now the value of $w(t)$ which minimizes the right-hand side of (2.3.12) depends on $x(t)$, $\frac{\partial J^*}{\partial t}$ and t . We assume that this minimizing control can be denoted by

$$\tilde{w}\left[x(t), \frac{\partial J^*}{\partial t}, t\right]$$

Thus, (2.3.12) becomes

$$\frac{\partial J^*}{\partial t} = -\psi\left[x(t), \tilde{w}\left[x(t), \frac{\partial J^*}{\partial t}, t\right]\right] + \frac{\partial J^{*T}}{\partial x} g\left[x(t), \tilde{w}\left[x(t), \frac{\partial J^*}{\partial t}, t\right]\right] \quad (2.3.13)$$

The above equation is a first-order partial differential equation with J^* as the dependent variable and $x(t)$ and t as independent variables (since $\psi(\cdot)$, $g(\cdot)$, and $\tilde{w}$, and accordingly, the minimum value of this performance index with respect to $w(\cdot)$ is also $h[x(t_f)]$), i.e.

$$J^*[x(t_f)] = h[x(t_f)] \quad (2.3.14)$$

As with the Hamilton-Jacobi equation, we have to seek the optimal control for the problem defined by (2.3.1) to (2.3.2). Here, it is assumed that a solution to (2.3.12) and (2.3.13) exists such that $J^*[x(t)]$ is a known function of its arguments. This solution $w^{Sol}[x(t)]$ is defined as follows :

$$w^{Sol}[x(t)] \equiv w^{Sol}\left[x(t), \frac{\partial J^*}{\partial x}(x(t))\right]$$

This new control function has two important attributes :

(1) $w^{Sol}[x(t)]$ represents the control which minimizes

$$J[x(t), w(\cdot)] = h[x(t_f)] + \int_t^{t_f} \psi[x(\tau), w(\tau)] d\tau \quad (2.3.15)$$

This indicates that in order to obtain the optimal performance index $J^*[\mathbf{x}(t)]$, we begin with control $\mathbf{w}^{sol}[\mathbf{x}(t)]$. This point is implicit in the arguments leading to (2.3.12), as previously shown.

(2) The optimal control $\mathbf{w}^{opt}$ for the optimal minimization problem is defined by (2.3.1) through (2.3.2), with t_0 as the initial time and t as an intermediate value of time. It is related to $\mathbf{w}^{sol}(\cdot, \cdot)$ by

$$\mathbf{w}^{opt}(t) = \mathbf{w}^{sol}[\mathbf{x}(t)] \quad (2.3.16)$$

where, $\mathbf{x}(t)$ is the state at time t arising from the application of $\mathbf{w}^{opt}(\cdot)$ over $[t_0, t_f]$.

A few further points need to be mentioned. The first is that $\mathbf{w}^{sol}[\mathbf{x}(t)]$ is independent of t_0 , which implies that the optimal control at an arbitrary time t for the minimization of

$$J[\mathbf{x}(\theta), \mathbf{w}(\cdot)] = \int_{\theta}^{t_f} \psi[\mathbf{x}(\tau), \mathbf{w}(\tau)] d\tau + h[\mathbf{x}(t_f)] \quad (2.3.17)$$

is also $\mathbf{w}^{sol}[\mathbf{x}(t)]$. This means the control $\mathbf{w}^{sol}[\mathbf{x}(\cdot)]$ is the optimal control for the whole class of problems (2.3.16) with respect to variables θ and $\mathbf{x}(\theta)$. The second point is that the optimal control at time t is expressed in terms of the state $\mathbf{x}(t)$ at time t , even though its functional dependence on the state may not be constant; it is generally a time-varying function of the state. Theoretically, the optimal control is implemented with a feedback law. It should be noted that other approaches (such as the Minimum Principle in the next section and the Euler-Lagrange equations for computing optimal control) are normally expressed as a certain function of time.

The final point is that the remarks leading to the Hamilton-Jacobi equation can be reversed. In other words, if a suitable solution to the equation is known, this solution must be the optimal performance index $J^*[x(t)]$.

2.3.6 The Pontryagin's Minimum Principle

The Hamilton-Jacobi equation (2.3.12) or (2.3.12)' provides a sufficient condition for the solution of the optimal control problem. If the set of admissible controls is not restricted, we may use the calculus of variations [34] in order to derive a set of necessary conditions for optimization. When the set of admissible control is bounded in some manner, unrestricted variations of $w(t)$ are not allowed [35]. If the minimum occurs at the boundary, then it is no longer true for the first variation because the slope vanishes at that point. Pontryagin's minimum principle, which was first produced by Ponryagin et al. [17] for control problems, can be used to find important characteristics of the optimal solution. It yields a necessary condition that an optimal control must satisfy, and has become the basic method for computing optimal controls. In a sense, it can also be thought of as an extension of the calculus of variations to the case of bounded control variables.

First we consider an autonomous system :

$$\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, \mathbf{w}) \quad (2.3.18)$$

with initial state,

$$\mathbf{x}(t_0) = \mathbf{x}_0 \quad (2.3.19)$$

a class of admissible controls,

$$\mathbf{w}(t) \in \mathbf{W} \quad (2.3.20)$$

and performance index,

$$J(\mathbf{x}_0, \mathbf{w}) = h[\mathbf{x}(t_f)] + \int_{t_0}^{t_f} \psi[\mathbf{x}(t), \mathbf{w}(t)] dt \quad (2.3.21)$$

where, $\psi(\cdot)$ is continuously differentiable in $\mathfrak{R}^n \times \mathbf{W}$.

Here, we provide the following definition for a scalar function, i.e.,

Let $H(\mathbf{x}, \mathbf{w}, \mathbf{p})$ denote the real-valued scalar function of the n -dimensional vector $\mathbf{x}$, the m -dimensional vector $\mathbf{w}$, and the n -dimensional vector $\mathbf{p}$, given by

$$H(\mathbf{x}, \mathbf{w}, \mathbf{p}) = \psi(\mathbf{x}, \mathbf{w}) + \mathbf{p}^T \mathbf{g}(\mathbf{x}, \mathbf{w}) \quad (2.3.22)$$

or using (2.3.18) :

$$H(\mathbf{x}, \mathbf{w}, \mathbf{p}) = \psi(\mathbf{x}, \mathbf{w}) + \mathbf{p}^T \dot{\mathbf{x}} \quad (2.3.23)$$

where $\mathbf{p} \in \mathfrak{R}^n$ is called the Lagrange multiplier or the costate vector; $H(\cdot)$ is the Hamiltonian.

Now, we can state Pontryagin's minimum principle by the following theorem.

Theorem (Pontryagin et al. 1962)

Suppose that $\mathbf{w}^{opt}$ is an optimal control for this problem and optimal states $\mathbf{x}^{opt}$ is the corresponding state trajectory. Then there exists a nonzero vector $\mathbf{p}(t)$ such that

$$\dot{\mathbf{x}} = \frac{\partial H}{\partial \mathbf{p}} = \mathbf{g}(\mathbf{x}, \mathbf{w}) \quad (2.3.24)$$

$$\dot{\mathbf{p}} = -\frac{\partial H}{\partial \mathbf{x}} = -\frac{\partial \psi}{\partial \mathbf{x}} - \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}} \right)^T \mathbf{p} \quad (2.3.25)$$

$$H(\mathbf{x}^{Opt}, \mathbf{w}^{Opt}, \mathbf{p}^{Opt}) = \min_{\mathbf{w} \in \mathbf{W}} H(\mathbf{x}^{Opt}, \mathbf{w}, \mathbf{p}^{Opt}) \quad (2.3.26)$$

where, the optimal costate vector is $\mathbf{p}^{Opt}$.

Futhermore, $H(\mathbf{x}^{Opt}, \mathbf{w}^{Opt}, \mathbf{p}^{Opt})$ is constant for $0 \leq t \leq t_f$. The costate vector differential (2.3.25) is accomodated with a final condition which is known as the *transversality condition*. This condition depends on the final condition $\mathbf{x}_f(t)$ and final time t_f , which is detailed in [35].

The minimum principle is principally set with the following boundary condition :

$$\mathbf{x}^{Opt}(t_0) = \mathbf{x}_0, \quad \mathbf{x}^{Opt}(t_f) \in S_1 \quad (2.3.27)$$

where S_1 is a *smooth k-fold* in $\mathfrak{R}^n$. This means, if a set $S(g_1, g_2, \dots, g_{n-k})$ is said to be a *smooth k-fold* in $\mathfrak{R}^n$ for every point $\mathbf{x}_0 \in S(\cdot)$, then the $n-k$ vectors $(\partial g_1 / \partial \mathbf{x})(\mathbf{x}_0), \dots, (\partial g_{n-k} / \partial \mathbf{x})(\mathbf{x}_0)$ are linearly independent.

For example, if $\mathbf{x}(t_f) = \mathbf{x}_f$, a fixed point, then regardless of t_f , these would be no boundary condition on $\mathbf{p}(t_f)$. For a free-end point situation, that is, $\mathbf{x}(t_f) \in \mathfrak{R}^n$ fixed and *free* t_f , then the boundary condition (2.3.25) would be :

$$\mathbf{p}(t_f) = \frac{\partial h(\mathbf{x}(t_f))}{\partial \mathbf{x}(t_f)} \quad (2.3.28)$$

The relation (2.3.25) consitutes a set of m static equations. This means, m inequalities for a constrained control $\mathbf{w}(t) \in \mathbf{W}$ or a set of m algebraic equations:

$$0 = \frac{\partial H}{\partial \mathbf{w}} = \frac{\partial \psi}{\partial \mathbf{x}} + \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}} \right)^T \mathbf{p} \quad (2.3.29)$$

for unconstrained $\mathbf{w}(t)$.

If (2.3.29) is satisfied, then the matrix

$$\frac{\partial^2 H}{\partial \mathbf{w}^2}(\mathbf{x}^{Opt}, \mathbf{w}^{Opt}, \mathbf{p}^{Opt})$$

becomes crucial.

If this matrix is positive definite, this is sufficient to guarantee that $\mathbf{w}^{Opt}$ causes H to be a local minimum.

If the Hamiltonian H can be expressed in the following form :

$$\begin{aligned} H(\mathbf{x}(t), \mathbf{w}(t), \mathbf{p}(t)) &= \mathbf{g}(\mathbf{x}(t), \mathbf{p}(t)) \\ &+ [\mathbf{c}(\mathbf{x}(t), \mathbf{p}(t))]^T \mathbf{w}(t) + \frac{1}{2} \mathbf{w}^T(t) \mathbf{R}(t) \mathbf{w}(t) \end{aligned} \quad (2.3.30)$$

where $\mathbf{c}$ is an array that does not have any terms containing $\mathbf{w}(t)$, then the satisfaction of (2.3.30) and a positive definite $\partial^2 H / \partial \mathbf{w}^2 \geq 0$ are necessary and sufficient conditions for $H(\mathbf{x}^{Opt}, \mathbf{w}^{Opt}, \mathbf{p}^{Opt})$ to be a global minimum.

If H is shown in (2.3.30), then we have

$$\frac{\partial^2}{\partial \mathbf{w}^2} H(\mathbf{x}^{Opt}, \mathbf{w}^{Opt}, \mathbf{p}^{Opt}) = \mathbf{R}(t) \quad (2.3.31)$$

Here, if $\mathbf{R}(t)$ is positive definite, then we obtain :

$$\mathbf{w}^{Opt}(t) = -\mathbf{R}^{-1}(t) \mathbf{c}(\mathbf{x}^{Opt}(t), \mathbf{p}^{Opt}(t)) \quad (2.3.32)$$

which globally minimizes the Hamiltonian.

Several well-known numerical techniques exists for obtaining the optimal controls $\mathbf{w}^{opt}(t)$ for example the two-point boundary-value (TPBV). The TPBV is constituted by (2.3.24) and (2.3.25) together with initial and final conditions on $\mathbf{x}(t)$ and $\mathbf{p}(t)$, such as (2.3.19) and (2.3.29). In other words, neither a complete set of initial nor final conditions for all equations is available [35].

Additional Necessary Conditions

Pontryagin and his co-workers derived other necessary conditions for optimality that we will find useful. We now state, without proof, two of these necessary conditions [34] :

1. If the final time is fixed and the Hamiltonian does not depend explicitly on time, then it must be a constant when evaluated on an extreme trajectory :

$$H(\mathbf{x}^{opt}, \mathbf{w}^{opt}, \mathbf{p}^{opt}) = const \quad \text{for all } t \in [t_0, t_f]$$

2. If the final time is free, and the Hamiltonian does not explicitly depend on time, then it must be identically zero when evaluated on an extreme trajectory :

$$H(\mathbf{x}^{opt}, \mathbf{w}^{opt}, \mathbf{p}^{opt}) = 0 \quad \text{for all } t \in [t_0, t_f]$$

Moreover, the Hamiltonian-Jacobi equation of Section 2.3.5 is another formulation of PMP (Appendix B).

2.3.7 The discrete-time Maximum principle

Our discussion so far has been devoted to the optimal control of continuous-time systems. Pontryagin's minimum principle of Section 2.3.6 is extended to discrete-time systems here. The discrete maximum principle can be considered as an extension of Pontryagin's minimum principle. Strictly speaking, the application of the discrete maximum principle reduces to an investigation of the system convexity. First, we consider the following nonlinear discrete-time system :

$$\mathbf{x}(k+1) = \mathbf{g}[\mathbf{x}(k), \mathbf{w}(k)] \quad (2.3.33)$$

with initial state :

$$\mathbf{x}(k_0) = \mathbf{x}_0 \quad (2.3.34)$$

and performance index

$$J = h[\mathbf{x}(k_f)] + \sum_{k=k_0}^{k_f} \psi[\mathbf{x}(k), \mathbf{w}(k)] \quad (2.3.35)$$

The term $h[\mathbf{x}(k_f)]$ is known as the terminal cost. This term would be required as the terminal condition, if $\mathbf{x}(k_f)$ is not fixed [35]. The design problem is to find the optimal control $\mathbf{w}^{opt}$ on $[k_0, k_f]$ such that the performance index (2.3.35) is minimized with respect to the equality constraints (2.3.33) and (2.3.34).

We define an $n \times 1$ costate vector $\mathbf{p}(k)$, the above optimization is equivalent to minimizing

$$J_1 = h[\mathbf{x}(k_f)] + \sum_{k=k_0}^{k_f-1} \left\{ \psi[\mathbf{x}(k), \mathbf{w}(k)] + \mathbf{p}^T(k+1)[\mathbf{x}(k+1) - \mathbf{g}[\mathbf{x}(k), \mathbf{w}(k)]] \right\} \quad (2.3.36)$$

Here, we define a scalar Hamiltonian, for the discrete-time case :

$$H[\mathbf{x}(k), \mathbf{w}(k), \mathbf{p}(k+1)] = \psi[\mathbf{x}(k), \mathbf{w}(k)] - \mathbf{p}^T(k+1)\mathbf{g}[\mathbf{x}(k), \mathbf{w}(k)] \quad (2.3.37)$$

As with the maximum principle, the Hamiltonian must be maximum along the optimal trajectory [35].

Substituting $H(\cdot)$ of (2.3.37) into (2.3.36) leads to the following objective function :

$$J_1 = h[\mathbf{x}(k_f)] + \sum_{k=k_0}^{k_f-1} \{H[\mathbf{x}(k), \mathbf{w}(k), \mathbf{p}(k+1)] - \mathbf{p}^T(k+1)\mathbf{x}(k+1)\} \quad (2.3.38)$$

This is the maximum principle. We let the state vectors $\mathbf{x}(k), \mathbf{x}(k+1)$, control $\mathbf{w}(k)$ and costate vector $\mathbf{p}(k+1)$ have the following variations :

$$\mathbf{x}(k) = \mathbf{x}^{Opt}(k) + \varepsilon \boldsymbol{\zeta}(k) \quad (2.3.39)$$

$$\mathbf{x}(k+1) = \mathbf{x}^{Opt}(k+1) + \varepsilon \boldsymbol{\xi}(k+1) \quad (2.3.40)$$

$$\mathbf{w}(k) = \mathbf{w}^{Opt}(k) + \vartheta \boldsymbol{\rho}(k) \quad (2.3.41)$$

$$\mathbf{p}(k+1) = \mathbf{p}^{Opt}(k+1) + \nu \boldsymbol{\lambda}(k+1) \quad (2.3.42)$$

Now, (2.3.38) can be described as follows :

$$\begin{aligned} J_1 = & h[\mathbf{x}^{Opt}(k_f) + \varepsilon \boldsymbol{\zeta}(k_f)] + \sum_{k=k_0}^{k_f-1} H[\mathbf{x}^{Opt}(k) + \varepsilon \boldsymbol{\zeta}(k), \mathbf{w}^{Opt}(k) \\ & + \vartheta \boldsymbol{\rho}(k), \mathbf{p}^{Opt}(k+1) + \nu \boldsymbol{\lambda}(k+1)] \\ & - [\mathbf{p}^{Opt}(k+1) + \nu \boldsymbol{\lambda}(k+1)]^T [\mathbf{x}^{Opt}(k+1) + \varepsilon \boldsymbol{\xi}(k+1)] \end{aligned} \quad (2.3.43)$$

Moreover, expanding $h(\mathbf{x}(k_f))$ into a Taylor series about $h(\mathbf{x}^{Opt}(k_f))$ yields

$$h(\mathbf{x}(k_f)) = h(\mathbf{x}^{Opt}(k_f)) + \varepsilon \boldsymbol{\zeta}^T(k_f) h_{\mathbf{x}^{Opt}}^{Opt} + \dots \quad (2.3.44)$$

Where $h_{\mathbf{x}^{OpI}} \equiv \frac{\partial h^{OpI}}{\partial \mathbf{x}^{OpI}}$.

Similarly, the above Hamiltonian can be extended into a Taylor series about

$\mathbf{x}^{OpI}(k), \mathbf{w}^{OpI}(k), \mathbf{p}^{OpI}(k+1)$ and $\mathbf{x}^{OpI}(k+1)$,

$$\begin{aligned} H[\mathbf{x}(k), \mathbf{w}(k), \mathbf{p}(k+1)] &= H[\mathbf{x}^{OpI}(k), \mathbf{w}^{OpI}(k), \mathbf{p}^{OpI}(k+1)] \\ &+ \varepsilon \zeta^T(k) H_{\mathbf{x}^{OpI}}^{OpI}(k) + \vartheta \rho^T(k) H_{\mathbf{w}^{OpI}}^{OpI}(k) + \nu \lambda^T(k+1) H_{\mathbf{p}^{OpI}}^{OpI}(k+1) + \dots \end{aligned} \quad (2.3.45)$$

$$\text{where} \quad H^{OpI}(k) = H[\mathbf{x}^{OpI}(k), \mathbf{w}^{OpI}(k), \mathbf{p}^{OpI}(k+1)] \quad (2.3.46)$$

Substituting (2.3.45) and (2.3.46) into (2.3.38), we perform the following necessary conditions for minimizing J_1 :

$$\left. \frac{\partial J_1}{\partial \varepsilon} \right|_{\varepsilon=\vartheta=\nu=0} = 0 \quad (2.3.47)$$

$$\left. \frac{\partial J_1}{\partial \vartheta} \right|_{\varepsilon=\vartheta=\nu=0} = 0 \quad (2.3.48)$$

$$\left. \frac{\partial J_1}{\partial \nu} \right|_{\varepsilon=\vartheta=\nu=0} = 0 \quad (2.3.49)$$

This results in

$$\zeta^T(k_f) h_{\mathbf{x}^{OpI}}^{OpI}(k_f) + \sum_{k=k_0}^{k_f-1} \zeta^T(k) H_{\mathbf{x}^{OpI}}^{OpI}(k) - \sum_{k=k_0}^{k_f-1} (\mathbf{p}^{OpI})^T(k+1) \zeta(k+1) = 0 \quad (2.3.50)$$

$$\rho^T(k) H_{\mathbf{w}^{OpI}}^{OpI}(k) = 0 \quad (2.3.51)$$

$$\lambda^T(k+1) H_{\mathbf{p}^{OpI}}^{OpI}(k+1) - \mathbf{x}^{OpI}(k+1) = 0 \quad (2.3.52)$$

The last equation (2.3.52) leads to

$$\mathbf{x}^{Opt}(k+1) = H_{\mathbf{p}(k+1)}^{Opt}(k+1) \quad (2.3.53)$$

$$H_{\mathbf{w}}^{Opt}(k) = \frac{\partial H^{Opt}(k)}{\partial \mathbf{w}(k)} = 0 \quad (2.3.54)$$

which represents the necessary condition for the maximization (or minimization) of the Hamiltonian along the optimal trajectory with respect to the optimal control.

The final set of necessary conditions for optimality stems from the last term on the left-hand side of (2.3.50) which can be described as follows :

$$\begin{aligned} \sum_{k=k_0}^{k_f-1} (\mathbf{p}^{Opt})^T(k+1) \zeta(k+1) &= \sum_{k=k_0+1}^{k_f} (\mathbf{p}^{Opt})^T(k) \zeta(k) \\ &= \sum_{k=k_0}^{k_f-1} (\mathbf{p}^{Opt})^T(k) \zeta(k) + (\mathbf{p}^{Opt})^T(k_f) \zeta(k_f) \\ &\quad - (\mathbf{p}^{Opt})^T(k_0) \zeta(k_0) \end{aligned} \quad (2.3.55)$$

Now $\mathbf{x}(k_0)$ is known. Moreover, if $\zeta(k_0) = 0$, then (2.3.55) becomes

$$\sum_{k=k_0}^{k_f-1} (\mathbf{p}^{Opt})^T(k+1) \zeta(k+1) = \sum_{k=k_0}^{k_f-1} (\mathbf{p}^{Opt})^T(k) \zeta(k) + (\mathbf{p}^{Opt})^T(k_f) \zeta(k_f) \quad (2.3.56)$$

Now, by substituting this into (2.3.50) and rearranging terms, we obtain the following equation :

$$[h_x^{Opt}(k_f) - \mathbf{p}^{Opt}(k_f)]^T \zeta(k_f) + \sum [H_x^{Opt}(k) - \mathbf{p}(k)]^T \zeta(k) = 0 \quad (2.3.57)$$

Since the variations of $\mathbf{p}(k)$ and $\mathbf{p}(k_f)$ are mutually independent [35], the only way to satisfy (2.3.57) is respectively

$$\mathbf{p}^{Opt}(k_f) = h_{\mathbf{x}}^{Opt}(k_f) \quad (2.3.58)$$

and

$$\mathbf{p}^{Opt}(k) = H_{\mathbf{x}}^{Opt}(k) \quad (2.3.59)$$

Thus, we precisely provide the following necessary conditions for the discrete-time maximpum principle :

$$\mathbf{p}^{Opt}(k) = \frac{\partial H^{Opt}(k)}{\partial \mathbf{x}^{Opt}(k)} \quad (2.3.60)$$

$$\mathbf{p}^{Opt}(k_f) = \frac{\partial H^{Opt}(k_f)}{\partial \mathbf{x}^{Opt}(k_f)} \quad (2.3.61)$$

$$\mathbf{x}^{Opt}(k+1) = \frac{\partial H^{Opt}(k)}{\partial \mathbf{p}^{Opt}(k+1)} \quad (2.3.62)$$

$$\mathbf{x}^{Opt}(k_0) = \mathbf{x}_0, \text{ known} \quad (2.3.63)$$

$$H_w^{Opt}(k) = 0 \quad (2.3.64)$$

Equations (2.3.60) through (2.3.63) represent a discrete-time TPBV problem and are referred to as *canonical state difference equations*. More specifically, (2.3.63) provides a means for finding the optimal control, while equations (2.3.61) and (2.3.63) represent the transversality and the initial state, respectively. Notice that if any component of the final state $\mathbf{x}(t_f)$ is fixed, the corresponding transversality condition of $\mathbf{p}^{Opt}(k_f)$ no longer applies [35].

Chapter 3

The New Learning algorithm based on Pontryagin's principle

3.1 Optimal Control Problems and ANNs

Classical control system design by ways of optimal control problems is generally a trial-and error process in which various methods of analysis are used iteratively to determine the design parameters of an "acceptable" system [36]. In the previous chapter, we saw that the objective of optimal control theory is to determine the control signals that cause a process to satisfy the physical constraints and at the same time minimize (or maximize) some performance index. In ANN research, optimal control problems are mainly applied in order to determine optimal weights for ANNs whose task is defined as the optimization of a specific performance index, as will be discussed in this section. We shall begin with the previous work on neurodynamics in optimal control problems, including that of Ramacher et al. [37], who used a single first-order Hamilton-Jacobi equation (as described previously in Section 2.3.5). It will then be shown how this theory fits into the framework of neural optimal control theory.

3.1.1 The form of state equations in continuous-time and discrete-time systems

Ramacher [37] considered the following neuron differential equation :

$$\dot{\mathbf{x}}(t) = \mathbf{g}(\mathbf{in}(t), \mathbf{x}(t), \mathbf{w}(t)) \quad (3.1.1)$$

where $in(t)$ represents the neuron inputs, $\mathbf{x}(t)$ the neuron outputs, and $\mathbf{w}(t)$ the weights.

We consider an n -dimensional continuous-time neural network with describing state equation as follows

$$\dot{\mathbf{x}}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{w}(t)) \quad (3.1.2)$$

with initial condition :

$$\mathbf{x}(0) = \mathbf{x}_0 \quad (3.1.3)$$

Moreover, (3.1.2) is a quite general form, which readily extends into ANNs, by reformulating the following system of differential equation, called *recurrent networks* by [38] :

$$\tau \dot{x}_j(t) = -x_j(t) + f(y_j) \quad (3.1.4)$$

subject to (2.1.4), where $f(\cdot)$ is the sigmoidal activation function which is represented by (2.1.2).

The types of training problem which can be presented to recurrent neural networks are more numerous than those which make sense for feedforward networks. Moreover, it is possible to train a recurrent temporally continuous generalization of back-propagation networks.

Equation (3.1.4) is apparently for the continuous-time case, but notice that (2.1.3) is indeed a discrete version of (3.1.4). Alternatively, an ANN can be represented by the following system equation :

$$\tau \dot{x}_j(t) = -x_j(t) + \sum_{i=1}^n w_{ji}(t) f(x_i(t)) \quad (3.1.4)'$$

Accordingly, we adopt the following forms for equation (3.1.2) :

$$\dot{\mathbf{x}}(t) = s(\mathbf{x}(t)) + \mathbf{w}(t)f(\mathbf{x}(t)) \quad (3.1.5)$$

and

$$\dot{x}_j(t) = s(x_j(t)) + \sum_{i=1}^n w_{ji}(t)f(x_i(t)) \quad (3.1.6)$$

Similarly, for an n -dimensional discrete-time system, the state equation can be approximated (see section 2.3.7) by

$$\mathbf{x}(k+1) = s(\mathbf{x}(k)) + \mathbf{w}(k)f(\mathbf{x}(k)) \quad (3.1.7)$$

and

$$x_j(k+1) = s(x_j(k)) + \sum_{i=1}^n w_{ji}(k)f(x_i(k)) \quad (3.1.8)$$

3.1.2 The performance index in Continuous-time ANN systems

We consider the performance J^C in continuous-time ANN systems which is defined by (2.3.21) :

$$J^C = h(\mathbf{x}(t_f)) + \int_{t_0}^{t_f} \psi(\mathbf{x}(t), \mathbf{w}(t))dt \quad (3.1.9)$$

where $\mathbf{x}$ represents the (neuron) state variable and $\mathbf{w}$ the controls, which in our application are ANN weight vectors. The performance index, which includes the objective function $h(\mathbf{x}(t_f))$, is to be minimized at final time t_f , which may be *specified* or *free* depending on the problem statement [35].

Here, we define the Hamiltonian for the continuous-time case, as follows

$$H^C(\mathbf{x}(t), \mathbf{w}(t), \mathbf{p}(t)) \equiv \mathbf{p}^T(t) \mathbf{g}(t) + \psi(\mathbf{x}(t), \mathbf{w}(t)) \quad (3.1.10)$$

where $\mathbf{p}^T$ is a set of the costate vector as has been defined in section (2.2.6), and the cost function ψ can be interpreted as the potential which forces the system to move along the desired trajectory [36], as will be discussed later.

Furthermore, when the Hamiltonian H is also defined as in (3.1.10), the performance index becomes

$$J^C = h(\mathbf{x}(t_f)) + \int_{t_0}^{t_f} [H^C(\mathbf{x}(t), \mathbf{w}(t), \mathbf{p}(t)) - \mathbf{p}^T(t) \mathbf{g}(t)] dt \quad (3.1.11)$$

3.1.3 The performance index in Discrete-time ANN systems

Similarly, we can extend this treatment to optimal control problems in discrete-time systems (See section 2.3.7), as follows

$$J^D = h(\mathbf{x}(k_f)) + \sum_{k=0}^{k_f-1} \psi(\mathbf{x}(k), \mathbf{w}(k)) \quad (3.1.12)$$

where J^D means the performance index in discrete-time ANN systems. In addition, $\mathbf{x}$ represents the (neuron) state variable and $\mathbf{w}$ the controls, which in our application are ANN weight vectors. The performance index, which includes the objective function $h(\mathbf{x}(k_f))$, is to be minimized at final time k_f .

Here, we define the Hamiltonian for the discrete-time case, as follows

$$H^D(\mathbf{x}(k), \mathbf{w}(k), \mathbf{p}(k+1)) \equiv \mathbf{p}^T(k+1) \mathbf{g}(k) + \psi(\mathbf{x}(k), \mathbf{w}(k)) \quad (3.1.13)$$

where $\mathbf{p}^T$ is a set of the costate vector as defined in Section (2.2.6), and the cost function ψ can be interpreted as the potential which forces the system to move along the desired trajectory [36], as will be discussed later.

Furthermore, when the Hamiltonian H is also defined as in (3.1.10), the performance index becomes

$$J^D = h(\mathbf{x}(k_f)) + \sum_{k=0}^{k_f-1} [H(\mathbf{x}(k), \mathbf{w}(k), \mathbf{p}(k+1)) - \mathbf{p}^T(k+1)\mathbf{g}(k)] \quad (3.1.14)$$

3.2 Learning Potential Functions and Pontryagin's principle

As described previously in Section 2.3, the solution of an optimal control problem can be obtained through two basic approaches. The first is by the application of the *Hamiltonian equation* which constitutes a sufficiency condition. The other approach is *Pontryagin's minimum principle* which constitutes necessary conditions for optimality. However, the Hamiltonian-Jacobi equation is an alternative formulation of the minimum principle, as shown in Appendix B. Ramacher has applied a partial differential equation (PDE) of Hamilton-Jacobi type in order to describe neurodynamics [37], whereas we shall employ the minimum principle in order to find significant characteristics of the optimal solution. It yields a necessary condition that an optimal control must satisfy, and has become the basic method for computing optimal controls [39]. In a sense, it can also be thought of as an extension of the method of Lagrange multipliers (costate) to dynamical optimization and control problems. Firstly, we attempt to apply the quadratic cost function $\psi(\cdot)$ as the learning potential function, first proposed by Amari [40] for dynamical ANNs.

3.2.1 Squared error function

We let $\psi(\mathbf{w})$, defined in (2.1.9), be a loss function. When an input signal is processed by a layered net of feedforward connections, we generally apply the following squared error as the cost function $\psi(\cdot)$ (as described in Section 2.1.4) :

$$\psi(\mathbf{w}) = E(\mathbf{x}, \mathbf{d}) = \frac{1}{2} \|\mathbf{x} - \mathbf{d}\|^2 \quad (3.2.1)$$

where $\|\cdot\|$ is the Euclidean norm, $\mathbf{x}$ is the value of the output vector, and $\mathbf{d}$ the target vector or desired response of the output vector.

Alternatively, this can be represented by considering the number of training patterns :

$$\psi_p(\mathbf{w}) = E_p(\mathbf{x}, \mathbf{d}) = \frac{1}{2} \sum_{q=1}^{N_L} (x_q^L(\mathbf{u}_p) - d_q(\mathbf{u}_p))^2 \quad (3.2.2)$$

where, x_q^L is the output of the q -node in layer L , and d_q the desired response of the q -th output node. In addition, $\mathbf{u}_p$ is the p -th training sample.

Notice that if we here apply the first order algorithm (the steepest gradient descent method) of Section 2.1.4 as weight equations with (3.2.1) or (3.2.2), the standard backpropagation learning algorithm is obtained. However, we consider on alternative cost function of $\psi(\cdot)$ instead of (3.2.1) or (3.2.2) in the next section to yield a new algorithm.

3.2.2 The learning potential function in Continuous-time and Discrete time systems

We here define the following quadratic cost function as the potential function [34][41], in order to apply it to optimal control problems. This general cost function $\psi(\cdot)$ is capable of generating a versatile class of dynamical systems while providing some useful algebraic simplifications, as will be shown later.

In the continuous-time case, the learning potential is assumed to be of quadratic form, i.e.,

$$\psi^C(\mathbf{w}) = \frac{1}{2}[\mathbf{x}^T(t)Q\mathbf{x}(t) + \mathbf{w}^T(t)R\mathbf{w}(t)] \quad (3.2.3)$$

subject to (3.1.5).

Similarly, in the discrete-time case,

$$\psi^D(\mathbf{w}) = \frac{1}{2}[\mathbf{x}^T(k)Q\mathbf{x}(k) + \mathbf{w}^T(k)R\mathbf{w}(k)] \quad (3.2.4)$$

subject to (3.1.7).

Here, Q is a real symmetric positive semidefinite matrix, and R is a real symmetric positive definite matrix.

3.2.3 Linear regulator problems and Resulting optimal weight equations

In this section, we shall apply the linear regulator problem using the Pontryagin principle in order to obtain the optimal weight equations of

ANNs, as an alternative to the first-order and second-order methods described previously in Sections 2.1 and 2.2, respectively. In Section 2.3.6, the necessary condition of the minimum principle was denoted by equations (2.3.24), (2.3.25) and (2.3.26) :

$$\dot{\mathbf{x}} = \frac{\partial H}{\partial \mathbf{p}} = \mathbf{g}(\mathbf{x}, \mathbf{w})$$

$$\dot{\mathbf{p}} = -\frac{\partial H}{\partial \mathbf{x}} = -\frac{\partial \psi}{\partial \mathbf{x}} - \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}} \right)^T \mathbf{p}$$

$$H(\mathbf{x}^{opt}, \mathbf{w}^{opt}, \mathbf{p}^{opt}) = \min_{\mathbf{w} \in \mathbf{W}} H(\mathbf{x}^{opt}, \mathbf{w}, \mathbf{p}^{opt})$$

where, $\mathbf{x}^{opt}$ is the optimal state, $\mathbf{w}^{opt}$ the control vector and $\mathbf{p}$ the costate vector.

In addition, we previously considered the following system equation (3.1.5), for the continuous-time case :

$$\dot{\mathbf{x}}(t) = \mathbf{s}(\mathbf{x}(t)) + \mathbf{w}(t)f(\mathbf{x}(t))$$

The performance index (3.1.11) that we are considering is also

$$J^C = h(\mathbf{x}(t_f)) + \int_{t_0}^{t_f} [H^C(\mathbf{x}(t), \mathbf{w}(t), \mathbf{p}(t)) - \mathbf{p}^T(t)\mathbf{g}(t)]dt$$

The boundary condition, consistent with (2.3.25), is

$$\mathbf{p}(t_f) = \frac{\partial h(\mathbf{x}(t_f))}{\partial \mathbf{x}(t_f)}$$

For free t_f

Furthermore, the Hamiltonian (3.1.10) is then defined as follows :

$$H^C(\mathbf{x}(t), \mathbf{w}(t), \mathbf{p}(t)) \equiv \mathbf{p}^T(t) \mathbf{g}(t) + \psi(\mathbf{x}(t), \mathbf{w}(t))$$

If we now apply the *linear regulator problem* to (3.1.11), then we obtain the following equation [35] :

$$J^C = \frac{1}{2} \mathbf{x}^T(t_f) F \mathbf{x}(t_f) + \frac{1}{2} \int_{t_0}^{t_f} [\mathbf{x}^T(t) Q \mathbf{x}(t) + \mathbf{w}^T(t) R \mathbf{w}(t)] \quad (3.2.5)$$

without loss of generality, the matrix F may be assumed to be symmetric. In addition, we require it to be positive semidefinite.

The linear regulator problem is to find an optimal weight (control) vector $\mathbf{w}^{opt}(t)$ which satisfies the system equation (3.1.5) while minimizing the performance index J^C in (3.2.5). The Hamiltonian, in (3.1.10), is given by

$$\begin{aligned} H^C &\equiv \psi^C(\mathbf{w}) + \mathbf{p}^T \mathbf{g} \\ &= \frac{1}{2} [\mathbf{x}^T(t) Q \mathbf{x}(t) + \mathbf{w}^T(t) R \mathbf{w}(t)] \\ &\quad + \mathbf{p}^T(t) [s(\mathbf{x}(t)) + \mathbf{w}(t) f(\mathbf{x}(t))] \end{aligned} \quad (3.2.6)$$

with the costate equation (2.3.25), and the control equation (2.3.29) for the weights given as follows :

$$0 = \frac{\partial H}{\partial \mathbf{w}} = \frac{\partial \psi}{\partial \mathbf{x}} + \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}} \right)^T \mathbf{p}$$

Now, since we are considering the continuous-time case, it assumed that $H \equiv H^C$.

Thus, (2.3.29) will be

$$0 = \frac{\partial H^C}{\partial \mathbf{w}(t)} = R\mathbf{w}(t) + f^T(t)\mathbf{p}(t) \quad (3.2.7)$$

Then $\mathbf{w}^{Op_t}(t)$ can be eliminated from (3.2.7), which yields :

$$\mathbf{w}^{Op_t}(t) = -R^{-1}f^T(\mathbf{x}(t))\mathbf{p}(t) \quad (3.2.8)$$

which minimizes (globally) the Hamiltonian H^C defined in (3.2.6). This derivation is presented fully in Appendix III. Equation (3.2.8) is a general formulation for Linear-Regulator problems in optimal control theory. In addition, (3.2.8) satisfies the necessary condition (2.3.25) and the boundary condition (2.3.28), respectively.

Equation (3.2.8) is the *continuous-time* optimal equation for ANNs we have been seeking.

In a similar manner we can obtain the optimal equation for discrete-time ANNs. In Section 2.3.7, the necessary condition for the discrete-time Maximum principle was denoted by equations (2.3.62) and (2.3.60) :

$$\mathbf{x}^{Op_t}(k+1) = \frac{\partial H^{Op_t}(k)}{\partial \mathbf{p}^{Op_t}(k+1)}$$

$$\mathbf{p}^{Op_t}(k) = \frac{\partial H^{Op_t}(k)}{\partial \mathbf{x}^{Op_t}(k)}$$

where, $\mathbf{x}^{Op_t}(k)$ is the optimal state, $\mathbf{p}^{Op_t}(k)$ is the costate vectors, and $H^{Op_t}(k)$ is the optimal Hamiltonian

In addition, we considered the following system equation (3.1.7), for the discrete-time case :

$$\mathbf{x}(k+1) = s(\mathbf{x}(k)) + \mathbf{w}(k)f(\mathbf{x}(k))$$

The performance index (3.1.14) that we are considering is also

$$J^D = h(\mathbf{x}(k_f)) + \sum_{k=0}^{k_f-1} [H(\mathbf{x}(k), \mathbf{w}(k), \mathbf{p}(k+1)) - \mathbf{p}^T(k+1)\mathbf{g}(k)]$$

Then, the boundary condition (2.3.61) is

$$\mathbf{p}^{opt}(k_f) = \frac{\partial h^{opt}(k_f)}{\partial \mathbf{x}^{opt}(k_f)}$$

Furthermore, the Hamiltonian (3.1.13) was defined as follows :

$$H^D(\mathbf{x}(k), \mathbf{w}(k), \mathbf{p}(k+1)) \equiv \mathbf{p}^T(k+1)\mathbf{g}(k) + \psi(\mathbf{x}(k), \mathbf{w}(k))$$

If we now apply the *linear regulator problem* to (3.1.14), then we obtain the following equation [35] :

$$J^D = \frac{1}{2} \mathbf{x}^T(k_f) F \mathbf{x}(k_f) + \frac{1}{2} \sum_{k=0}^{k_f-1} [\mathbf{x}^T(k) Q \mathbf{x}(k) + \mathbf{w}^T(k) R \mathbf{w}(k)] \quad (3.2.9)$$

where matrix F is assumed to be positive semidefinite.

The linear regulator problem is to find an optimal weight (control) vector $\mathbf{w}^{opt}(k)$ which satisfies the system equation (3.1.7) while minimizing the performance index J^D in (3.2.9). The Hamiltonian (3.1.13) is given by

$$\begin{aligned} H^D &\equiv \psi^D(\mathbf{w}) + \mathbf{p}^T \mathbf{g} \\ &= \frac{1}{2} [\mathbf{x}^T(k) Q \mathbf{x}(k) + \mathbf{w}^T(k) R \mathbf{w}(k)] \\ &\quad + \mathbf{p}^T(k) [s(\mathbf{x}(k)) + \mathbf{w}(k) f(\mathbf{x}(k))] \end{aligned} \quad (3.2.10)$$

with the costate equation (2.3.60), and the control equation (2.3.64) for the weights given as follows,

$$H_w^{Opt}(k) = 0$$

Now since we are considering the discrete-time case, it assumed that $H \equiv H^D$.

Thus, (2.3.64) will be

$$0 = H_w^D(k) = R\mathbf{w}(k) + f^T(\mathbf{x}(k))\mathbf{p}(k) \quad (3.2.11)$$

Then $\mathbf{w}^{Opt}(k)$ can be eliminated from (3.2.11), which yields :

$$\mathbf{w}^{Opt}(k) = -Rf^T(\mathbf{x}(k))\mathbf{p}(k) \quad (3.2.12)$$

which minimizes (globally) the Hamiltonian H^D defined in (3.2.10). Equation (3.2.12) is a general formulation for *discrete-time* Linear-Regulator problems in optimal control theory. In addition, (3.2.12) satisfies the necessary condition (2.3.60) and the boundary condition (2.3.61), respectively.

Equation (3.2.12) is the *discrete-time* optimal equation for ANNs we have been seeking.

3.3 The PMP learning algorithm

In Section 3.2, the new optimal equations for weights were contrasted with the weight equations obtained using steepest gradient descent backpropagation (SBP) described in Section 2.1.4. The obtained weight equation comprises both the learning algorithm kernel as well as backpropagation, since the aim is to minimize the quadratic cost function, and thus give rise

to a modification for each connection weight. First of all, the conventional SBP weight equation is reviewed.

3.3.1 Optimal weight equations

SBP or the Generalized Delta Rule (GDR) for obtaining optimal weights proposed by Rumelhart et al [3], (Section 2.1.4) has become very popular since its initial publication (1986). The GDR for updating the weights $\mathbf{w}$ is given by the differential equation (2.1.19):

$$\Delta \mathbf{w}(k) = -\mu \left. \frac{\partial \psi}{\partial \mathbf{w}} \right|_{\mathbf{w}(k)} + \alpha \Delta \mathbf{w}(k-1)$$

Here, we set $\alpha = 0$:

$$\Delta \mathbf{w}(k) = -\mu \left. \frac{\partial \psi}{\partial \mathbf{w}} \right|_{\mathbf{w}(k)} \quad (3.3.1)$$

where, ψ is squared cost error function, and μ is the learning rate coefficient (or learning gain term).

Alternatively, we obtain

$$\Delta w_{ji}(k) = -\mu \nabla_{w_{ji}} \psi \quad (3.3.2)$$

where, $\nabla_{w_{ji}}$ is the partial derivative with respect to w_{ji} , connecting the j -th node in a layer to the i -th node in the subsequent layer.

Notice that equations (3.3.1) and (3.3.2) provide us with the insight that a positive definite matrix R in both optimal equations (3.2.8) and (3.2.11) will have the following analogous relationships with the learning rate coefficient μ .

Now, assuming for both (3.2.8) and (3.2.11) :

$$\begin{aligned} \mathbf{w}^{opt} &\cong \Delta \mathbf{w}, \\ R^{-1} &\cong \mu \end{aligned} \tag{3.3.3}$$

then we obtain the following optimal weight equation :

$$\Delta \mathbf{w}(k) = -\mu f^T(\mathbf{x}(k)) \mathbf{p}(k) \tag{3.3.4}$$

Similarly, we can consider the optimal weight equation for the continuous-time case :

$$\Delta \mathbf{w}(t) = -\mu f^T(\mathbf{x}(t)) \mathbf{p}(t) \tag{3.3.5}$$

Now it is possible to reformulate this optimal weight equation (3.3.4) using Pontryagin's principle, and thus replace Rumelhart's conventional SBP learning algorithm (Section 3.3.1).

3.3.2 Learning Dynamics and its equation

In order to find the optimal weights, we let the costate vector $\mathbf{p}$ be chosen to satisfy the boundary condition (2.3.28) for the continuous-time case ((2.3.61) for the discrete-time case), with respect to the final *free* time (Sections 2.3.6 and 2.3.7) :

$$\mathbf{p}(t) = \frac{\partial h(\mathbf{x}(t))}{\partial \mathbf{x}(t)}, \text{ in continuous time}$$

$$\mathbf{p}(k) = \frac{\partial h(\mathbf{x}(k))}{\partial \mathbf{x}(k)}, \text{ in discrete-time}$$

In this manner, we can obtain the following optimal weight equations with respect to (3.3.4) and (3.3.5), respectively:

$$\Delta \mathbf{w}(t) = -\mu \frac{\partial h(\mathbf{x}(t))}{\partial \mathbf{x}(t)} f(\mathbf{x}(t)) \quad (3.3.6)$$

and

$$\Delta \mathbf{w}(k) = -\mu \frac{\partial h(\mathbf{x}(k))}{\partial \mathbf{x}(k)} f(\mathbf{x}(k)) \quad (3.3.7)$$

An early learning equation was qualitatively formulated by Hebb in 1949, and constitutes the basic mechanism of neural learning [42]. This equation may be simply quantified using the discrete-time formula :

$$\mathbf{w}^{New}(k) = \mathbf{w}^{Old}(k) + \Delta \mathbf{w}(k) \quad (3.3.8)$$

Alternatively,

$$w_{ji}(k) = w_{ji}(k-1) + \Delta w_{ji}(k) \quad (3.3.9)$$

Thus, we obtain the following learning rule for (3.3.2) :

$$w_{ji}(k) = w_{ji}(k-1) - \mu \nabla_{w_{ji}} E \quad (3.3.10)$$

In general, this is well known as the (standard) back propagation analog of the *Widrow-Hoff rule* or *LMS* (least mean square) algorithm [43]. In addition, the continuous time version of (3.3.10) is :

$$\frac{dw_{ji}(t)}{dt} = -\mu \nabla_{w_{ji}} E$$

The above system of ordinary differential equations (ODEs) is usually *stiff*, and the stability of numerical methods for solving stiff systems of ODEs is discussed in detail by Gear [44].

Now we obtain the following rule in discrete-time, using (3.3.7) :

$$w_{ji}(k) = w_{ji}(k-1) - \mu \nabla_{x_j} h(\mathbf{x}) f(x_i) \quad (3.3.11)$$

which can be represented as follows :

$$w_{ji}(k) = w_{ji}(k-1) - \mu \frac{\partial h(\mathbf{x})}{\partial x_j} f(x_i) \quad (3.3.12)$$

This suggests that the development of the new learning law as above will provide a faster descent to the bottom of the error surface of Figure 2-4.

The objective function $h(\mathbf{x})$ is minimized at $\mathbf{x}^*$, such that $h(\mathbf{x}^*) = \min_{\mathbf{x}} h(\mathbf{x})$, where $h(\mathbf{x})$ has continuous second order derivatives (or twice-continuously differentiable), so the objective function can have the following gradient vector [45] :

$$\nabla_{x_j} h(\mathbf{x}) = \left[\frac{\partial}{\partial x_1} h(\mathbf{x}) \dots \frac{\partial}{\partial x_n} h(\mathbf{x}) \right]^T \quad (3.3.13)$$

To obtain the optimum convergence rate for this gradient vector, the update rule (3.3.13) should use the following Hessian matrix :

$$\nabla^2_{x_j} h(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial^2 x_1} h(\mathbf{x}) \dots \frac{\partial^2}{\partial x_1 \partial x_n} h(\mathbf{x}) \\ \dots \dots \\ \frac{\partial^2}{\partial x_n \partial x_1} h(\mathbf{x}) \dots \frac{\partial^2}{\partial^2 x_n} h(\mathbf{x}) \end{bmatrix} \quad (3.3.14)$$

In short, [45] suggests that *second order derivatives* be used in the numerical determination of the optimal state.

Chapter 4

Results and Conclusion

4.1 Numerical Example and Performance Evaluation

In this chapter, the performance of the PMP learning algorithm is evaluated by comparison with classical SBP. In order to gain a quantitative implementation, we used the C-language on a Sun SPARCstation. First we need to define the form for the continuous second-order derivatives $h(\cdot)$. Newton's method which utilizes the Hessian matrix is the well known basic local method [28] for determining the second order derivatives in ANNs. It is also far superior with respect to learning time (number of iterations) [16]. Before describing the quantitative implementation, we first define the form of second order derivatives $h(\cdot)$.

4.1.1 The form of second order derivative $h(\cdot)$

We consider the objective function $h(\cdot)$, to have the following quadratic form :

$$h(\mathbf{x}) \equiv \mathbf{x}^T \boldsymbol{\wp} \mathbf{x}$$

The description of the l -th layer will be :

$$h^l(\mathbf{x}) = \sum_{i=1}^n \sum_{k=1}^n \wp_{ik}^l x_i^l x_k^l \quad l = 1, 2, \dots, +\infty$$

where, a diagonal symmetric matrix $\wp = (\wp_{ik}), \wp_{ik} = \wp_{ki}, i, k = 1, 2, \dots, n$ can be either positive or negative, definite or indefinite. It determines both the gradient vector $\nabla \mathbf{h}(\mathbf{x}) = \wp \mathbf{x}$ and the constant diagonal Hessian matrix $\nabla^2 \mathbf{h}(\mathbf{x}) = \wp$, which can always be diagonalized by choosing a basis of eigenvectors. Sufficient conditions for a strong local minimum to the point $\mathbf{x}^*$ are that $|\nabla^2 \mathbf{h}(\mathbf{x}^*)| = 0$. Notice that if the Hessian matrix is positive definite and the quadratic model is correct, *one* iteration only is sufficient to reach the global minimum. However, even if the Hessian is not positive definite, it can be shown that the quadratic form can be effective on such minimization problems [28].

4.1.2 The application of the continuous first order derivative $f(\cdot)$

The activation function $f(\cdot)$ is adequately represented by the following nonlinear sigmoid equation [46][47][48] :

$$f(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{x})}$$

The main properties of sigmoidal activation have been discussed in Section 2.1.1.

4.1.3 Numerical Result

In order to evaluate the performance of the proposed learning algorithm, we compare it with the classical SBP algorithm for solving the *XOR problem*. The network architecture that we use has three layers as shown in Figure 4-1. Each input node in Figure 4-1 has a signal strength or activation function of the sigmoid type previously described in Section 2.1.1. For the current implementation, the network comprises eleven input nodes, two

Figure 4-1 : Three-Layer Back Propagation with Feed-forward network (11-2-1)

hidden nodes and one output node. The network is also fully connected with weights initialized randomly prior to learning. The value of any initial weight w is such that $0.03 \leq |w|$. In addition, the learning coefficients for the two methods are $\eta = 0.5$, and $\mu = 0.0005$, respectively.

Eleven inputs are presented to the network, and the output trained to reproduce the logical *exclusive--or* truth table. The 11-2-1 network can be described as follow :

This is an extension of XOR into 11 dimensions, where the eleventh dimension is obtained by successively ANDing the first two. For example :

and,

For inputs (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) and (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), the eleventh input (0) can be obtained in a similar manner.

The performance of the new PMP algorithm depends on the choice of the constant Hessian matrix $\mathcal{H} = (\mathcal{H}_{ik})$ previously described in Section 4.1.1., which is adjusted in order to obtain the best performance for this task. We here apply the following simple Hessian matrix :

$$\mathcal{H}_{ik}^l = |\mathcal{N}| \times \delta_{ik}^l \quad \mathcal{N}: \text{constants}$$

where, the rule δ_{ik}^l is the *Kronecker delta* :

$$\delta_{ik}^l \equiv \begin{cases} 1 & (i = k) \\ 0 & (i \neq k) \end{cases} \quad i, k = 1, 2, \dots, n$$

A recommended strategy for changing the Hessian in order to reach the minimum is, in fact, that of adding to it a simple diagonal matrix of the form $|\mathcal{N}|I$, I being the *identity matrix* as above [28]. An algorithm for efficiently computing just the diagonal of the Hessian [49]. Effective use of the optimum quadratic Hessian has been analyzed by [50][51].

The results are presented in Figure 4-2. Convergence to a tolerance of 0.001 took 640 and 1012 iterations for the new PMP learning algorithm and classic SBP respectively. This clearly demonstrates that traditional SBP requires more than 1.5 times as many iterations as the new PMP method. We next changed the positive constants $\aleph$ of the Hessian matrix. From Figure 4-3, we see that larger values of $\aleph$ result in faster convergence for a fixed error tolerance (0.001).

Figure 4-2: Comparison of the new algorithm and the SBP ($\kappa = 1.0 \times 10^4$)

Figure 4-3: Relationships of error and iterations as the parameter (κ) is varied

Moreover, the results of Figure 4.4 and table 1 are based on training using the same rule $\phi^{l_{ik}}(i, k = 1, 2, \dots, n)$, but different constants κ . In particular, notice that for $\kappa = 10 \times 10^4$ the network converges 220 times faster than for $\kappa = 1.5 \times 10^4$. Thus as κ increases, the number of iterations to convergence decreases exponentially.

Figure 4-4: Description of iterations as changing constants (κ)
Iteration to convergence (0.001)

Table 1

Constants (κ)	Iterations
15000	429
20000	324
50000	131
80000	37
90000	2
100000	2

Table 1 shows the number of iterations at convergence (0.001) when the constants (κ) are applied for training, based on Figure 4-4

4.1.4 Discussion

It is well recognised that the selection of initial weights in neural network training can significantly influence the speed of convergence. The usual procedure is to start with small, random values. Ideally, we would prefer to commence training with *optimal* weight values. Now PMP provides us with such, since this new learning algorithm is derived from minimisation theory (by regarding the weights as control variables). Moreover, such minimisation is readily represented in terms of either first or second-order derivative methods. Especially, the latter (which utilise the Hessian matrix). Similar second-order techniques are the basis for so-called “expanded” SBP methods, including PMP, all of which are capable of outperforming standard SBP.

4.2 Summary and Suggestions for Further work

We have shown that a method based on classical optimal control theory, notably Pontryagin's minimum (maximum) principle (PMP), is suited to Hamiltonian Neural Network systems. In Section 3.1, we showed that neurons and weights, being control variables in the network, are able to obey a continuous-time differential equation, generally called *the state equations*. The ANN's state equation (3.1.4) for recurrent (feedback) network models can be modelled as the neuron state e.g., (3.1.4)', being also continuous-time. Furthermore, (3.1.4)' can be extended to *discrete-time* systems under optimal control theory. As a result, the ANN's state equations, in both continuous-time and discrete-time, can yield optimal weights equations, using the new PMP learning method.

In Section 3.2, we showed that it is possible to describe the learning potential function as cost functions in Hamiltonian Neural Network systems. However, developing criteria for selecting appropriate learning potential functions is left for further research. Finally, the new learning algorithm

was numerically compared with the traditional SBP algorithm, and found to yield superior performance. PMP has yet to be compared against *second-order* extensions to SBP however (such as Conjugate Gradient or Newton's Method), this exercise is left for further research.

Our work suggests that we could obtain better rules for obtaining optimal network topology for weight dynamics attributed to learning. Although the *quadratic rule* leads to several successful results for *some* networks, alternative forms for the second order derivatives (Section 4.1.1) might be needed for others. Moreover, the obtained weight equation in continuous-time could be investigated (we have only implemented the discrete-time case here). Lastly, we expect applications using Hamiltonian control systems (Appendix A) would lead to further useful results with respect to neurodynamics research. For example, we would fully expect PMP to yield better generalisation ability, since Hamiltonian Control Theory is a *more general* formulation of (modern) Optimal Control Theory.

Appendix A

Hamiltonian control system and PMP

Here we show how classical optimal control theory such as PMP is organized under Hamiltonian (*affine*) control systems in "*mechanical nonlinear control system*" [52][53]. In short, the well known Euler-Lagrange or Hamiltonian equations of motion have been described in terms of "*Classical mechanics*" by [54][55], however here they will be discussed in the context of "*mechanical nonlinear control systems*". Firstly, we review the Euler-Lagrangian and Hamiltonian equations of motion from "*Classical mechanics*". Then we extend this control theory into Hamiltonian (*affine*) control systems.

Consider a mechanical system with n degrees of freedom: The following equations of motion are derived with n generalized configuration (position) coordinates $x = (x_1, x_2, \dots, x_n)$:

$$\frac{d}{dt} \left[\frac{\partial L_0}{\partial \dot{x}_i} \right] - \frac{\partial L_0}{\partial x_i} = 0 \quad i \in n \quad (\text{A.1})$$

Here, $L_0 = L_0(x_i, \dot{x}_i)$ (A.1) is called the Lagrange equation of motion, and is a set of n second order differential equations in which $\dot{x}_i = (\dot{x}_1, \dot{x}_2, \dots, \dot{x}_n)$ are the generalized velocities. Moreover, the Hamiltonian equations of motion in a mechanical system [54] are given by

$$\dot{x}_i = \frac{\partial H_0}{\partial \lambda_i} \quad (\text{A.2.a})$$

$$\dot{\lambda}_i = -\frac{\partial H_0}{\partial x_i} \quad i \in n \quad (\text{A.2.b})$$

$H_0 = H_0(x_i, \lambda_i)$ is defined as the Hamiltonian or canonical equation of motion, in which $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_n)$ are the generalized momenta.

Now the Euler-Lagrangian or Hamiltonian equations from general mechanics as above, which can be extended to control systems, or to be more precise, mechanical nonlinear control systems [52].

Consider a mechanical control system with n degrees of freedom, described by the Lagrangian $L(x, \dot{x}, w)$, which also depends directly on w , being the controls. Then the equations of motion are:

$$\frac{d}{dt} \left[\frac{\partial L(x, \dot{x}, w)}{\partial \dot{x}_i} \right] - \frac{\partial L(x, \dot{x}, w)}{\partial x_i} = 0 \quad i \in n \quad (\text{A.3})$$

where

$$L(x, \dot{x}, w) = L_0(x, \dot{x}) + \sum_{j=1}^n x_j w_j \quad (\text{A.4})$$

We call (A.3) a Lagrangian control system. In addition, defining $T(x, \dot{x})$ as the kinetic energy of the system and $V(x)$ as the potential energy function, we obtain the following form for $L_0(x, \dot{x})$ [54]:

$$L_0(x, \dot{x}) = T(x, \dot{x}) - V(x) \quad (\text{A.5})$$

As with the Lagrangian control system of (A.3), we define the generalized momenta as follows

$$\lambda_i = \frac{\partial L(x, \dot{x}, w)}{\partial \dot{x}_i} \quad i \in n \quad (\text{A.6})$$

Now the Hamiltonian function $H(x, \lambda, w)$ can be defined as the Legendre transform of $L(x, \dot{x}, w)$, as follows:

$$H(x, \lambda, w) = \sum_{i=1}^n \lambda_i \dot{x}_i - L(x, \dot{x}, w) \quad (\text{A.7})$$

where $\dot{x}$ and λ are related by (A.6). Moreover, it is well-known that (A.6) and (A.7) constitute the Euler-Lagrange equations of motion:

$$\begin{aligned} \dot{x}_i &= \frac{\partial H(x, \lambda, w)}{\partial \lambda_i} \\ \dot{\lambda}_i &= -\frac{\partial H(x, \lambda, w)}{\partial x_i} \end{aligned} \quad i \in n \quad (\text{A.8})$$

We call (A.8) a Hamiltonian control system. We now define the natural outputs of the Hamiltonian control system, as being

$$y_j = -\frac{\partial H(x, \lambda, w)}{\partial w_j} \quad j \in n \quad (\text{A.9})$$

In particular, if we confine ourselves to Hamiltonian (*affine*) control systems, equation (A.8) is defined with the following Hamiltonian form:

$$H(x, \lambda, w) = H_0(x, \lambda) - \sum_{j=1}^n H_j(x, \lambda) w_j \quad (\text{A.10})$$

Now we obtain the Hamiltonian (*affine*) control system as follows [52]:

$$\begin{aligned} \dot{x}_i &= \frac{\partial H_0(x, \lambda)}{\partial \lambda_i} - \sum_{j=1}^n \frac{\partial H_j(x, \lambda) w_j}{\partial \lambda_i} \\ \dot{\lambda}_i &= -\frac{\partial H_0(x, \lambda)}{\partial x_i} - \sum_{j=1}^n \frac{\partial H_j(x, \lambda) w_j}{\partial x_i} \\ y_j &= H_j(x, \lambda) \end{aligned} \quad \begin{aligned} i &\in n \\ & \\ j &\in n \end{aligned} \quad (\text{A.11})$$

where, defining the Hamiltonian function H_0 as the internal Hamiltonian in [52].

$$\text{Let } \frac{dH_0(x, \lambda)}{dt} \quad (\text{A.12})$$

Thus, we obtain the following result from (A.10) and (A.11):

$$\begin{aligned} \frac{dH_0}{dt} &= \sum_i \left(\frac{\partial H_0}{\partial x_i} \dot{x}_i + \frac{\partial H_0}{\partial \lambda_i} \dot{\lambda}_i \right) \\ &= \sum_i \left(\frac{\partial H_0}{\partial x_i} \frac{\partial H_0}{\partial \lambda_i} - \frac{\partial H_0}{\partial \lambda_i} \frac{\partial H_0}{\partial x_i} \right) + \sum_{i,j} \left(-\frac{\partial H_0}{\partial x_i} \frac{\partial H_j}{\partial \lambda_i} + \frac{\partial H_0}{\partial \lambda_i} \frac{\partial H_j}{\partial x_i} \right) w_j \\ &= \sum_{i,j} \left\{ \frac{\partial H_j}{\partial \lambda_i} \left(-\frac{\partial H_0}{\partial x_i} + \sum_k w_k \frac{\partial H_k}{\partial x_i} \right) + \frac{\partial H_j}{\partial x_i} \left(\frac{\partial H_0}{\partial \lambda_i} + \sum_k w_k \frac{\partial H_k}{\partial \lambda_i} \right) \right\} w_j \\ &= \sum_j \frac{dH_j}{dt} w_j \\ &= \sum_j \dot{y}_j w_j \quad k \in m \end{aligned} \quad (\text{A.13})$$

If $w_j = 0$ ($j \in n$), (A.13) is

$$\frac{dH_0(x, \lambda)}{dt} = 0 \quad (\text{A.14})$$

Therefore, $H_0(x, \lambda)$ is a *conserved quantity*, which expresses the conservation of total energy, and which may be represented as $H_0(x, \lambda) = T(x, \lambda) + V(x) = \text{total energy}$.

Notice that we are able to obtain the same result with (2.3.29) and (2.3.54), given the necessary conditions for PMP, when (A.9) is equal to 0 (Zero) with respect to the optimal controls w^{opt}_j ($j \in n$). end.

Appendix B

PMP and the HAMILTONIAN-JACOBI Equation

The Hamiltonian-Jacobi equation is defined as follows [56]:

$$-\frac{\partial J^*}{\partial t} = H^*(x(t), \lambda^T(t)) \quad (B.1)$$

where

$$H^*(x(t), \lambda^T(t)) = \min_w H(x(t), w(t), \lambda^T(t))$$

$$\lambda^T(t) \equiv \frac{\partial J^*}{\partial x}$$

$$J = h(x(t_f)) + \int_{t_0}^{t_f} \varphi(x(t), w(t)) dt$$

$$H(x, \lambda, w) = \varphi(x, w) + \lambda^T(t) f(x, w), \quad \dot{x}(x, w) \equiv f(x, w)$$

$$J^*(x) \equiv \min_w J(x, w)$$

$$\text{Let } \frac{d\lambda^T}{dt}$$

then we have

$$\frac{d\lambda^T}{dt} \equiv \frac{d}{dt} \left(\frac{\partial J^*}{\partial x} \right) = \frac{\partial^2 J^*}{\partial x^2} f + \frac{\partial^2 J^*}{\partial x \partial t} \quad (B.2)$$

Partial differentiation of the above Hamiltonian-Jacobi equation with respect to x yields

$$\begin{aligned} & \frac{\partial}{\partial x} \left(\frac{\partial J^*}{\partial t} + H^*(x(t), \lambda^T(t)) \right) \\ &= \frac{\partial^2 J^*}{\partial x \partial t} + \frac{\partial}{\partial x} H^*(x(t), \lambda^T(t)) \\ &= 0 \end{aligned}$$

Thus, we obtain the following equation, considering $w^* = w^*(x(t))$ and (B.1):

$$\begin{aligned} & \frac{\partial^2 J^*}{\partial x \partial t} + f^T \frac{\partial^2 J^*}{\partial x^2} + \frac{\partial \varphi}{\partial x} + \frac{\partial \varphi}{\partial w} \frac{\partial w^*}{\partial x} + \frac{\partial J^*}{\partial x} \left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial w} \frac{\partial w^*}{\partial x} \right) \\ & = 0 \end{aligned} \quad (\text{B.3})$$

The PMP necessary condition (2.3.29) has been stated as being

$$\frac{\partial H}{\partial w} = \frac{\partial \varphi}{\partial w} + \frac{\partial J^*}{\partial x} \frac{\partial f}{\partial w} = 0 \quad (\text{B.4})$$

Now, the coefficient of $\frac{\partial w^*}{\partial x}$ in (B.3) vanishes on an optimal path according to (B.4).

Consequently, we obtain the following equation:

$$\frac{\partial^2 J^*}{\partial x \partial t} + f^T \frac{\partial^2 J^*}{\partial x^2} + \frac{\partial \varphi}{\partial x} + \frac{\partial J^*}{\partial x} \frac{\partial f}{\partial x} = 0$$

Thus, (B.3) will become, taking into account (B.2)

$$\frac{d\lambda^T}{dt} = -\frac{\partial \varphi}{\partial x} - \lambda^T \frac{\partial f}{\partial x} \quad (\text{B.5})$$

(B.4) and (B.5) describe the Euler-Lagrange equation.

end.

Appendix C

Proof of Equation (3.2.8)

From equation (3.1.6),

$$\dot{x}_j(t) = s(x_j(t)) + \sum_{i=1}^n w_{ji}(t)f(x_i(t))$$

Now, (3.2.6) can be represented as follows

$$H^c(\cdot) = \sum_{j,i} \left[p_j \dot{x}_j + \frac{1}{2} Q_{ji} x_j x_i \right] + \frac{1}{2} \sum_{\alpha,\beta} R_{\alpha,\beta} W_\alpha W_\beta \quad (3.2.6)'$$

where α and β represent multi-indices :

$$\alpha \equiv i_\alpha j_\alpha, \quad \beta \equiv i_\beta j_\beta$$

Thus, (3.2.6)' can be represented from (3.1.6) as follows :

$$H^c(\cdot) = \sum_{j,i} \left[p_j (s(x_j) + W_{ji} f(x_i)) + \frac{1}{2} Q_{ji} x_j x_i \right] + \frac{1}{2} \sum_{\alpha,\beta} R_{\alpha,\beta} W_\alpha W_\beta \quad (3.2.6)''$$

$$\text{Let } 0 = \frac{\partial H(\cdot)}{\partial W_\alpha}$$

$$\begin{aligned} &= \sum_{j,i} (p_j f(x_i) \delta_{jj_\alpha} \delta_{ii_\beta} + R_{\alpha\beta} W_\beta) \\ &= p_{i_\alpha} f(x_{j_\alpha}) + R_{\alpha\beta} W_\beta \end{aligned}$$

Therefore,

$$0 = p_{j_\alpha} f(x_{i_\alpha}) + R_{\alpha\beta} W_\beta$$

$$(i.e) \quad W_\beta(t) = - \sum_{\alpha} \left(R_{\beta\alpha}^{-1} p_{j_\alpha}(t) f(x_{i_\alpha}(t)) \right)$$

Equation (3.2.12) can be derived in a similar manner.

end.

Appendix D

Program Code

This program based on SBP.C by Minh Tam Tran

```

/*****
/* Program : The PMP learning algorithm */
/* Module : hbp.c */
/* Purpose : This tool is used that the performance of the PMP learning */
/* algorithm is evaluated by comparison with classical Standard BP */
/* Author : Takamasa Koshizen, Minh Tam Tarn */
/* Date : October 1994 */
/*****
for declaration of C header files
*****/

#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>

/*****
define constants used throughout functions
*****/

#define MAX_LAYERS 5
#define MAX_VECTORS 1000
#define TRAINING_FILE "training.dat"
#define WEIGHTS_FILE0 "weights.dat0" /* for PMP */
#define WEIGHTS_FILE "weights.dat" /* for SBP */
#define OUTPUT_FILE "output.dat"
#define TEST_FILE "test.dat"
#define AVGERR_FILE "avgerr.dat"
#define Rlow -0.030
#define Rhigh 0.030
#define urand() ((float)rand() / 0x7fff * (Rhigh - Rlow) + Rlow)
struct layer
{
    int num_inputs; /* the number of inputs */
    int num_outputs; /* the number of outputs */
    float *hessians; /* pointer to array of hessians */
    float *outputs; /* pointer to array of outputs */
    float *inputs; /* pointer to array of inputs */
    float *weights; /* pointer to array of weights */
    float *output_errors; /* array of errors at output */
    float *back_errors; /* array of errors backpropagated */
};

/*****
Data base
*****/

struct layer *layer_ptr[MAX_LAYERS];
int number_of_layers;
float *buffer;
float *expected_values;
int layer_size[MAX_LAYERS];
unsigned training;

/*****
For declaration of functions
*****/

void get_layer_info(void);
void set_up_network(void);
void set_hessian(void);
void no_memory(void);

```



```

        l->back_errors[i] *= (*(l->inputs+i))*(1-(*(l->inputs+i)));
    }
}

/*****
Calculate error for output layer
*****/

void calc_error_output_layer(float * error)
{
    int i, j, k;
    struct layer *l;
    float accumulator = 0;
    float total_error = 0;

    l = layer_ptr[number_of_layers-1];

    for (j=0; j < l->num_outputs; j++)
    {
        l->output_errors[j] = (expected_values[j] - l->outputs[j]);
        total_error += l->output_errors[j];
    }

    *error = total_error;

    for (i=0; i < l->num_inputs; i++)
    {
        k = i * l->num_outputs;

        for (j=0; j < l->num_outputs; j++)
        {
            l->back_errors[i] = l->weights[k+j] * l->output_errors[j];
            accumulator += l->back_errors[i];
        }

        l->back_errors[i]=accumulator;
        accumulator = 0;

        /* multiply by derivative of sigmoid which is: input * (1 - input) */
        l->back_errors[i] *= (*(l->inputs+i))*(1-(*(l->inputs+i)));
    }
}

/*****
Fill memory with an array of input, output vectors up to MAX_VECTORS.
Return number of read vectors
*****/

int fill_IObuffer(FILE * inputfile)
{
    int i, k, count, veclength;
    int ins, outs;

    ins = layer_ptr[0]->num_outputs;
    outs = layer_ptr[number_of_layers-1]->num_outputs;

    if (training == 1)
    {
        veclength = ins + outs;
    }else veclength = ins;

    count = 0;

```

```

void    randomize_weights(void);
float    randomweight(void);
void    read_weights(FILE * weights_file_ptr, int layer_no);
int      fill_IObuffer(FILE * inputfile);
void    set_up_pattern(int buffer_index);
void    forward_prop(void);
float    squash(float input);
void    write_outputs(FILE * outfile);
void    backward_prop(float * toterror);
void    calc_error_output_layer(float * error);
void    calc_error_middle_layer(int layer_no);
void    update_weights(const float beta);
void    update_weights1(const float beta);
void    write_weights(FILE *weights_file_ptr);
void    list_weights(void);
void    list_errors(void);
void    list_outputs(void);
void    free_memory(void);
void    write_weights0(FILE *weights_file_ptr0);

/*****
Routine for backpropagation rule
*****/

void backward_prop(float *toterror)
{
    int i;

    /* error for the output layer */
    calc_error_output_layer(toterror);

    /* error for the middle layers */
    for (i=number_of_layers-2; i > 0; i--)
    {
        calc_error_middle_layer(i);
    }
}

/*****
Calculate error for middle layer
*****/

void calc_error_middle_layer(int layer_no)
{
    int i, j, k;
    float accumulator = 0;
    struct layer *l;

    l = layer_ptr[layer_no];

    for (i=0; i < l->num_inputs; i++)
    {
        k = i * l->num_outputs;

        for (j=0; j < l->num_outputs; j++)
        {
            l->back_errors[i] = l->weights[k+j] * (l->output_errors[j]);
            accumulator += l->back_errors[i];
        }

        l->back_errors[i] = accumulator;
        accumulator = 0;

        /* multiply by derivative of sigmoid which is: input * (1 - input) */

```

```

while ((count < MAX_VECTORS) &&
      (!feof(inputfile)))
{
    k = count*(veclength);

    for (i=0; i < veclength; i++)
    {
        fscanf(inputfile,"%f",&buffer[k+i]);
        fscanf(inputfile,"\n");
        count++;
    }

    if (!(ferror(inputfile)))
    {
        return count;
    }else return -1;
}

}

void free_memory(void)
{
    int i;

    free(layer_ptr[0]);
    free(layer_ptr[0]->outputs);

    for (i=1; i < number_of_layers; i++)
    {
        free(layer_ptr[i]->hessians);
        free(layer_ptr[i]->weights);
        free(layer_ptr[i]->output_errors);
        free(layer_ptr[i]->back_errors);
        free(layer_ptr[i]->outputs);
        free(layer_ptr[i]);
    }

    free(expected_values);
    free(buffer);
}

/*****
bottom up calculation of net for input pattern
*****/

void forward_prop(void)
{
    int i, j, k, m;
    struct layer *l;
    float accumulator = 0.0;

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];

        for (j=0; j < l->num_outputs; j++)
        {
            for (i=0; i < l->num_inputs; i++)
            {
                k = i * l->num_outputs;

                if (l->weights[k+j] * l->weights[k+j] > 1000000.0)
                {
                    printf("Weights too big. Use smaller learning rate\n");
                    exit(1);
                }
            }
        }
    }
}

```

```

        }
        l->outputs[j] = l->weights[k+j] * (*(l->inputs+i));
        accumulator += l->outputs[j];
    }

    l->outputs[j] = squash(accumulator);
    accumulator=0.0;
}
}

/*****
get layer sizes for the network
*****/

void get_layer_info(void)
{
    int i;

    printf("Enter number of layers for the network (min.3 max.5):\n");
    scanf("%d",&number_of_layers);
    printf("Enter the layer sizes, eg. 3 2 4 : \n");

    for (i=0; i < number_of_layers; i++)
    {
        scanf("%d",&layer_size[i]);
    }
}

void list_inputs(void)
{
    int m, i, j;
    struct layer *l;

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];
        printf("layer number : %d\n",m);

        for (i=0; i < l->num_inputs; i++)
        {
            printf("Back_error[%d] is : %f\n",i,l->back_errors[i]);
        }

        for (j=0; j < l->num_outputs; j++)
        {
            printf("Output_error[%d] is : %f\n",j,l->output_errors[j]);
        }
    }
}

void list_outputs(void)
{
    int m, i;
    struct layer *l;

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];
        printf("layer number : %d\n", m);

        for (i=0; i < l->num_outputs; i++)

```

```

        {
            printf("output[%d] is: %f\n", i, l->outputs[i]);
        }
    }
}

void list_weights(void)
{
    int m, i, j, k;
    struct layer *l;

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];
        printf("layer number : %d\n", m);

        for (i=0; i < l->num_inputs; i++)
        {
            k = i * l->num_outputs;

            for (j=0; j < l->num_outputs; j++)
            {
                printf("weight[%d,%d] is: %f\n", i, j, l->weights[k+j]);
            }
        }
    }
}

/*****
main body of learning
*****/

void main()
{
    float error_tolerance      = 0.1;
    float total_error          = 0.0;
    float avg_error_per_cycle  = 0.0;
    float error_last_cycle     = 0.0;
    float avgerr_per_pattern   = 0.0;
    float error_last_pattern   = 0.0;
    float learning_parameter   = 0.02;
    unsigned startup;
    long vectors_in_buffer;
    long max_cycles;
    long patterns_per_cycle    = 0;

    long total_cycles;
    long total_patterns;
    int i;

    FILE * training_file_ptr, * weights_file_ptr, * output_file_ptr;
    FILE * test_file_ptr,    * data_file_ptr;
    FILE * avgerr_file_ptr,   * weights_file_ptr0;

    if ((output_file_ptr = fopen(OUTPUT_FILE, "w")) == NULL)
    {
        printf("Problem opening output file\n");
        exit(1);
    }

    if ((avgerr_file_ptr = fopen(AVGERR_FILE, "w")) == NULL)
    {
        printf("Problem opening output file\n");
        exit(1);
    }
}

```

```

}

printf("Enter 1 for TRAINING, 0 for TESTING:\n");
scanf("%d",&training);

if (training == 1)
{
    printf("Enter error tolerance, learning rate. Eg. 0.1 0.5 :\n");
    scanf("%f %f",&error_tolerance, &learning_parameter);

    if ((training_file_ptr = fopen(TRAINING_FILE,"r")) == NULL)
    {
        printf("Error opening training file\n");
        exit(1);
    }

    /* training on */
    data_file_ptr=training_file_ptr;
}
else(

    if ((test_file_ptr = fopen(TEST_FILE,"r")) == NULL)
    {
        printf("Error opening test file\n");
        exit(1);
    }

    /* training off */
    data_file_ptr=test_file_ptr;

}

/**
TRAINING : loop until the total error is less than the tolerance specified
or the maximum number of cycles is exceeded.
TESTING : go through the input data set once in the forward
propagation phase only. Read the starting weights from a file.
***/

total_cycles =0; /* a cycle is once through all the input data */
total_patterns=0; /* a pattern is one entry in the input data */

get_layer_info(); /* get layer information */

set_up_network(); /* set up network connections */

set_hessians(); /* set up hessian matrices */

/* initialise the weights */
if (training == 1)
{
    if ((weights_file_ptr0 = fopen(WEIGHTS_FILE0,"w")) == NULL)
    {
        printf("Problems opening weights file\n");
        exit(1);
    }

    if ((weights_file_ptr = fopen(WEIGHTS_FILE,"w")) == NULL)
    {
        printf("Problems opening weights file\n");
        exit(1);
    }

    randomize_weights();
    write_weights0(weights_file_ptr0);
}

```

```

    }else{
        if ((weights_file_ptr = fopen(WEIGHTS_FILE,"r")) == NULL)
        {
            printf("Problems opening weights file\n");
            exit(1);
        }

        for (i=1; i < number_of_layers; i++)
        {
            read_weights(weights_file_ptr,i);
        }
    }

    /**
MAIN LOOP : If training, go through the input data until error
is acceptable or the max cycles is exceeded. If testing, go through
the input data once, write outputs to file output.dat
***/

    startup = 1;
    total_error = 0;

    vectors_in_buffer = fill_IObuffer(data_file_ptr);

    /* fill buffer */
    if (vectors_in_buffer < 0)
    {
        printf("Error in reading vectors, aborting.\n");
        exit(1);
    }

    while (((training == 1)
            && (avgerr_per_pattern > error_tolerance)
            && (vectors_in_buffer != 0))
            || ((training == 0)
            && (total_cycles < 1))
            || ((training == 1)
            && (startup == 1 ))
            )
    {
        startup = 0;
        error_last_cycle = 0;
        patterns_per_cycle = 0;

    /**
process all the vectors in the datafile, going through one buffer
at a time, pattern by pattern
***/

        for (i=0; i < vectors_in_buffer; i++)
        {
            set_up_pattern(i);

            total_patterns++;
            patterns_per_cycle++;

            /*-----*/
            /*      forward propagate      */
            /*-----*/
            forward_prop();

            /*-----*/
            /*  back propagate if appropriate  */
            /*-----*/

```

```

        if (training == 1)
        {
            backward_prop(&error_last_pattern);
            error_last_cycle += error_last_pattern *
            error_last_pattern;
            update_weights1(learning_parameter);

        }else write_outputs(output_file_ptr);

    }

    error_last_pattern = 0;
    avgerr_per_pattern = ((float)sqrt((double)error_last_cycle))/
    patterns_per_cycle;
    total_error += error_last_cycle;
    total_cycles++;

    printf("%ld\t%f\n",total_cycles, avgerr_per_pattern);
    fprintf(avgerr_file_ptr, "%ld\t%f\n", total_cycles, avgerr_per_pat

} /* end of main loop */

exit_while:

printf("\n\n\n\n\n\n\n\n\n\n\n");
printf("-----\n");
printf("          O N L I N E      U P D A T E S\n");
printf("-----\n");
printf(" Results in file output.dat\n");

if (training == 1)
{
    write_weights(weights_file_ptr);
    write_outputs(output_file_ptr);
    avg_error_per_cycle = (float)sqrt((double)total_error)/
    total_cycles;
    error_last_cycle = (float)sqrt((double)error_last_cycle);

    printf(" Initialising Weights saved in file weights.dat0\n");
    printf(" Updating Weights saved in file weights.dat\n");
    printf(" Error saved in file avgerr.dat\n");
    printf(" Average error per cycle = %f\n",avg_error_per_cycle);
    printf(" Error last cycle = %f\n",error_last_cycle);
    printf(" Error last cycle per pattern = %f\n",avgerr_per_pattern);
}

printf("\n Total cycles = %ld\n",total_cycles);
printf(" Total patterns = %ld\n",total_patterns);
printf("-----\n");

fclose(data_file_ptr);
fclose(weights_file_ptr);
fclose(output_file_ptr);
fclose(avgerr_file_ptr);
fclose(weights_file_ptr0);

free_memory();

}

void no_memory(void)
{
    printf("Not enough memory\n");
    exit(1);
}

/*****

```



```

read in the weight data file
*****/

void read_weights(FILE * weights_file_ptr, int layer_no)
{
    int i, j, k;
    struct layer *l;

    l = layer_ptr[layer_no];

    while(1)
    {
        fscanf(weights_file_ptr, "%i", &j);

        if ((j==layer_no) || (feof(weights_file_ptr))) break;
        else
        {
            while (fgetc(weights_file_ptr) != '\n')
                ({});
        }
    }

    if (!(feof(weights_file_ptr)))
    {
        i=0;

        for (j=0; j < l->num_outputs; j++)
        {
            fscanf(weights_file_ptr, "%f", &l->weights[j]);
            fscanf(weights_file_ptr, "\n");

            for (i=1; i < l->num_inputs; i++)
            {
                fscanf(weights_file_ptr, "%i", &layer_no);
                k = i * l->num_outputs;

                for (j=0; j < l->num_outputs; j++)
                {
                    fscanf(weights_file_ptr, "%f", &l->weights[k+j]);
                }

                fscanf(weights_file_ptr, "\n");
            }
        }
        }else printf("end of file reached\n");
    }

    /*****/
    random number generator
    *****/

    void randomize_weights()
    {
        int w, i, j, k;
        struct layer *l;

        /* seed the random generator */
        srand((unsigned) time(NULL));

        for (w=1; w < number_of_layers; w++)
        {
            l = layer_ptr[w];

            for (i=0; i < l->num_inputs; i++)

```

```

    {
        k = i * l->num_outputs;

        for (j=0; j < l->num_outputs; j++)
        {
            l->weights[k+j] = urand();
        }
    }
}

/*****
initialize weights with random numbers
*****/

float randomweight(void)
{
    int num;

    num=rand() % 100;
    return 2 * ( (float) (num/100.00))-1;
}

/*****
input the hessian
*****/

void set_hessians(void)
{
    int i, j, k, m;
    struct layer *l;

    printf("Set hessian matrices!\n");

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];

        for (i=0; i < l->num_inputs; i++)
        {
            k = i * l->num_outputs;

            for (j=0; j < l->num_outputs; j++)
            {
                printf("C[%d,%d] = ", m, j+1, i+1);
                scanf("%f", &(l->hessians[k+j]));
            }
        }
        printf("\n");
    }
}

/*****
construct the layers
*****/

void set_up_network(void)
{
    int i, j, k;
    struct layer *l;

```

```

layer_ptr[0] = (struct layer *) malloc (sizeof(struct layer));
layer_ptr[0]->num_inputs = 0;
layer_ptr[0]->num_outputs = layer_size[0];

if ((layer_ptr[0]->outputs=
    (float *) malloc(sizeof(float) * layer_size[0])) == NULL)
{
    no_memory();
}

for (i=1; i < number_of_layers; i++)
{
    layer_ptr[i] = (struct layer *) malloc (sizeof(struct layer));

    if (layer_ptr[i] == 0)
    {
        no_memory();
    }

    l = layer_ptr[i];
    l->num_inputs = layer_size[i-1];
    l->num_outputs = layer_size[i];
    l->hessians = (float *) malloc (sizeof(float) * l->num_outputs
        * l->num_outputs);
    l->weights = (float *) malloc (sizeof(float) * l->num_inputs
        * l->num_outputs);
    l->output_errors = (float *) malloc (sizeof(float) * l->num_outputs);
    l->back_errors = (float *) malloc (sizeof(float) * l->num_inputs);
    l->outputs = (float *) malloc (sizeof(float) * l->num_outputs);

    if ((l->weights == 0) || (l->output_errors == 0) ||
        (l->back_errors == 0) || (l->outputs == 0))
    {
        no_memory();
    }

    if ((expected_values = (float *) malloc (sizeof(float)
        * layer_ptr[number_of_layers-1]->num_outputs)) == 0)
    {
        no_memory();
    }
}

/****
Connect the Layers : set inputs to previous layer outputs for all
layers, except input layer
****/

    for (i=1; i < number_of_layers; i++)
    {
        layer_ptr[i]->inputs = layer_ptr[i-1]->outputs;
    }

/****
set output_errors to next layer's back_errors for all layers except
the output layer and the input layer
****/

    for (i=1; i < (number_of_layers - 1); i++)
    {
        layer_ptr[i]->output_errors = layer_ptr[i+1]->back_errors;
    }

/****
define the IObuffer that caches data from the datafile
****/

```

```

        i=layer_ptr[0]->num_outputs;
        j=layer_ptr[number_of_layers-1]->num_outputs;
        k=MAX_VECTORS;

        if ((buffer = (float *) malloc (sizeof(float) * (i+j)*k))==0)
        {
            no_memory();
        }
    }

}

/*****
read one vector into the network
*****/

void set_up_pattern(int buffer_index)
{
    int i, k;
    int ins, outs;

    ins = layer_ptr[0]->num_outputs;
    outs = layer_ptr[number_of_layers-1]->num_outputs;

    if (training == 1)
    {
        k = buffer_index * (ins+outs);
    }else k = buffer_index * ins;

    for (i=0; i < ins; i++)
    {
        layer_ptr[0]->outputs[i]=buffer[k+i];

        /* set target output values */
        if (training == 1)
        {
            for (i=0; i < outs; i++)
            {
                expected_values[i] = buffer[k+i+ins];
            }
        }
    }
}

/*****
sigmoid squashing function
*****/

float squash(float input)
{
    if (input < -50)
    {
        return 0.0;
    }else if (input > 50)
    {
        return 1.0;
    }else return (float)(1/(1+exp(-(double)input)));
}

/*****
update output with charged weights

```

```

*****/
void update_weights1(const float beta)
{
    int i, j, k, m;
    struct layer *l;
    float wt_chg;

    /***
    learning law: weight change = beta * hessian * outputs * squash(inputs);
    ***/

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];
        for (i=0; i < l->num_inputs; i++)
        {
            k = i * l->num_outputs;
            for (j=0; j < l->num_outputs; j++)
            {
                wt_chg = -(beta * (l->hessians[k+j] * (l->outputs[j]
                    * (squash(l->inputs[i])))));
                l->weights[k+j] += wt_chg;
            }
        }
    }

    /***
    write the output data file
    ***/

    void write_outputs(FILE *outfile)
    {
        int i, ins, outs;

        ins = layer_ptr[0]->num_outputs;
        outs = layer_ptr[number_of_layers-1]->num_outputs;

        fprintf(outfile, "for input vector:\n");

        for (i=0; i < ins; i++)
        {
            fprintf(outfile, "%f ", layer_ptr[0]->outputs[i]);

            fprintf(outfile, "\noutput vector is:\n");

            for (i=0; i < outs; i++)
            {
                fprintf(outfile, "%f ", layer_ptr[number_of_layers-1]->outputs[i]);
            }

            if (training == 1)
            {
                fprintf(outfile, "\nexpected output vector is:\n");

                for (i=0; i < outs; i++)
                {
                    fprintf(outfile, "%f ", expected_values[i]);
                }
            }
            fprintf(outfile, "\n-----\n");
        }
    }
}

```

```

}

/*****
write the weight data file: part1
*****/

void write_weights(FILE * weights_file_ptr)
{
    int i, j, k, m;
    struct layer *l;
    char *ch = "UPDATING WEIGHTS";

    fprintf(weights_file_ptr,"%s\n ", ch);

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];

        for (i=0; i < l->num_inputs; i++)
        {
            k = i * l->num_outputs;

            for (j=0; j < l->num_outputs; j++)
            {
                fprintf(weights_file_ptr,"w%d[%d,%d]= %f\n ",m,
                    j+1, i+1, l->weights[k+j]);
            }
        }
    }
}

/*****
write the weight data file: part2
*****/

void write_weights0(FILE * weights_file_ptr0)
{
    int i, j, k, m;
    struct layer *l;
    char *ch = "INITIALISING WEIGHTS";

    fprintf(weights_file_ptr0,"%s\n ", ch);

    for (m=1; m < number_of_layers; m++)
    {
        l = layer_ptr[m];

        for (i=0; i < l->num_inputs; i++)
        {
            k = i * l->num_outputs;

            for (j=0; j < l->num_outputs; j++)
            {
                fprintf(weights_file_ptr0,"w%d[%d,%d]= %f\n ",m, j+1,
                    i+1, l->weights[k+j]);
            }
        }
    }
}

```

Bibliography

- [1] V. Rao. Vemuri, Artificial Neural Networks: Concepts and Control Applications, IEEE Computer Society Press, Los Alamitos, California, 1992
- [2] J. Hertz, A., Krogh et al., Introduction to the Theory of Neural Computation, Addison-Wesley, 1991.
- [3] D. E. Rumelhart and J. L. McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition (Volumes 1 and 2), MIT Press, 1986.
- [4] R. Beale and T. Jackson, Neural Computing: An Introduction, Institute of Physics Publishing, Bristol and Philadelphia, 1990.
- [5] M. Y. Choi and B. A. Huberman, "*Dynamic behaviour of nonlinear networks*", Physical Reviews A, 28, pp. 1204-1206, 1983.
- [6] J. J. Hopfield, "*Neural networks and physical systems with emergent omputational abilities*", Proceedings of the National Academy of Sciences USA, pp. 2554, 1982.
- [7] M. Wilson and J. Bower, "*Computer simulation of oscillatory behaviour in cerebral cortical networks*", In D. S. Touretzky, editor, Advances in Neural Information Processing Systems, Vol. 2, pp. 84, San Mateo CA, Morgan Kaufmann, 1990.
- [8] Bill Baird, "*Associative memory in a simple model of oscillating cortex*", In D. S. Touretzky, editor, Advances in Neural Information Processing Systems, Vol. 2, pp. 68, San Mateo CA, Morgan Kaufmann, 1990.
- [9] Charles M. Gray, Peter Konig et al., "*Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties*", Nature, pp. 334-337, 1989.
- [10] Christine A. Skarda and Walter J. Freeman. "*How brains make chaos in order to make sense of the world.*", Behavioural and Brain Sciences, pp. 161-195, 1987.

- [10] R. Battiti, *"Accelerated backpropagation learning: Two optimization methods"*, Complex Systems, Vol. 3, pp. 331-342, 1989
- [12] D. M. Goodman et al., *"Backpropagation Learning for Multi-Layer Feed-Forward Neural Networks Using the Conjugate Gradient Method"*, International Journal of Neural Systems, Vol. 2, pp. 291-302, 1991.
- [13] T. T. Jervis and W. J. Fitzgerald, *"Optimization Schemes for Neural Networks"*, Technical Report CUED/F-INFENG/TR 144, Cambridge University, England
- [14] B. A. Pearlmutter, *"Fast Exact Multiplication by the Hessian"*, Neural Computation, Vol.?, pp. 147-160, 1994
- [15] Moller, M. F., *"A Scaled Conjugate Gradient Algorithm for fast supervised learning"*. Neural Networks, Vol.6, pp. 525-533, 1991
- [16] P. Patrick van der Smagt, *"Minimisation Methods for Training Feedforward Neural Networks"*, Neural Networks, Vol. 7, pp. 1-11, 1994
- [17] L. S. Pontryagin et al., *The Mathematical Theory of Optimal Processes*, Trans. by K. N. Trirogoff, Interscience Wiley, New York, 1962.
- [18] R. Hecht-Neilsen, *"Theory of the Backpropagation Neural Network"*, In Neural Networks for Human and Machine Perception, Volume 2, H. Wechsler, ed., pp. 65-93. Academic Press, Boston, MA.
- [19] R. Rohwer, *"The Time Dimension of Neural Network Models"*, SIGART Bulletin, Vol. 5, No. 3, pp. 36-44, July 1994.
- [20] K. Funahashi, *"On the approximate realization of continuous mappings by neural networks"*, Neural Networks, 2:183, 1989.
- [21] A. Ohn, *"Fast Learning Methods for Artificial Neural Networks"*, MSc thesis, Norwegian Institute of Technology (NTH), Trondheim, Norway, 1994
- [22] D. R. Hush and B. G. Horne, *"Progress in supervised Neural Networks"*, IEEE Signal Processing Magazine, pp. 8-39, January, 1993.
- [23] D. R. Hush, B Horne and J. M. Salas, *"Error Surfaces for Multilayer Perceptrons"*, IEEE Transactions on Systems, Man and Cybernetics, Vol. 22, 5, pp. 1152-1161, 1992.

- [24] S.Becker and Y. le Cun, *"Improving the convergence of back-propagation learning with second order methods."* In proceedings of the 1988 Connectionists Models Summer School, pp. 29-37. Morgan Kaufmann, 1988.
- [25] E. D. Dahl, *"Accelerated learning using superpositions of a sigmoidal function"* , In proceedings of the IEEE 1st International Conference on Neural Networks, Volume 2, pp. 523-530, San Diego, CA, 1987.
- [26] R. A. Jacobs, *"Increased rates of convergence through learning rate adaption"* , Neural Networks, 1(4):295-308, 1988.
- [27] R. L Watrous, *"Learning Algorithm for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization"*, Proceeding IEEE 1st International Conference on Neural Network, Vol.2, pp.619-628, 1987.
- [28] R. Battiti, *"First-and second-order methods for learning: Between steepest descent and Newton's method"*, Neural Computation, Vol.4, pp.141-166, 1992.
- [29] E. Barnard, *"Optimization for training neral nets"*, IEEE Transactions on Neural Networks, Vol. 3, No. 2, pp.232-240, 1992
- [30] A. J. Kinsella, *"Comparison and evaluation of variants of the conjugate gradient method for efficient learning learning in feed-forward neural networks with backward error propagation"*, Network, Vol.3, pp.27-35, 1992
- [31] D. G. Luemberger, Linear and Nonlinear Programming, 2nd Edition, Addison Wesley, 1984
- [32] W. H. Press, S. A. Teukolsky et al., Numerical Recipes in C, 2nd Edition, Cambridge University Press, 1992
- [33] M. F. Moller, *"A scaled conjugate gradient algorithm for fast supervised learning"*, Neural Networks, Vol. 6, pp. 525-533, 1991
- [34] D. E. Kirk, Optimal Control Theory, Prentice-Hall Inc., Englewood Cliffs New Jersey, 1970
- [35] M. Jamshidi, M. Tarokh et al., Computer-Aided Analysis and Desigh of Linear Control Systems, Prentice-Hall Inc., Englewood Cliffs New Jersey, 1992

- [36] R. B. Perez, "*Optimal parameter tracking and control of nonlinear systems with time varying parameters*", International Conference on Acoustics, Speech, and Signal Processing, Vol. 4, pp. 2224-2227, 1988.
- [37] U. Ramacher and P. Nachbar, "*The Hamiltonian Approach to neural networks dynamics*", Proc. IJCNN-91, Singapore, Vol. 3, pp. 1930-1936, Nov. 1991
- [38] F. Pineda, "*Generalization of backpropagation to recurrent neural networks*", Physical Review Letters, Vol. 19, No. 59, pp. 2229-2232, 1987
- [39] V. Vemuri et al., "*Neural Networks can be used for open-loop, dynamic control*", International Journal of Neural Networks, Vol. 3, No. 3, pp. 71-83, September 1992
- [40] S. Amari, "*Mathematical Theory of Neural Learning*", New Generation Computing, Vol. 8, pp. 281-294, 1991
- [41] H. Tokumaru et al., "*A Nonlinear Regulator Design in the Presence of System Uncertainties Using Multilayered Neural Networks*", IEEE Transactions on Neural Networks, Vol. 2, No. 4, pp.410-417, July 1991
- [42] E. Davalo et al., Neural Networks, London: Macmillan Education, 1991
- [43] Widrow, B., and E. M. Hoff, Adaptive Switching Circuits. Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, part 4, pp. 96-104, 1960
- [44] Gear, C.W., Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, New York, 1971.
- [45] D. G. Luenberger, Linear and Nonlinear Programming, 2nd ed, Reading, Mass: Addison Wesley, 1984
- [46] M. E. Savran, "*On the Design of Dynamic Associate Neural Memories*", IEEE Transactions on Neural Networks, Vol. 5, No. 3, pp. 489-492, May 1994.
- [47] M. W. Hirsch, "*Saturation at High Gain in Discrete Time Recurrent Networks*", Neural Networks, No. 3, Vol. 7, pp. 449-453, 1994.
- [48] L. G. Sotolino et al., "*Classification of Temporal Trajectories by Continuous-Time Recurrent Nets*", Neural Networks, No. 5, Vol. 7, pp. 767-776, 1994.

- [49] S. Becker and Y. Le Cun, *"Improving the Convergence of Back-Propagation Learning with Second Order Methods"*, in Proceedings of the 1988 Connectionist Models Summer School, D. S. Touretzky, G. E. Hinton, and T. J. Sejnowski, eds., pp. 29-37. Morgan Kaufmann, San Mateo, CA.
- [50] J. E. Dennis et al., *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, NJ, 1983.
- [51] P. E. Gill et al., *Practical Optimization*, Academic Press, London, 1981.
- [52] A. J. van der Schaft et al., *Nonlinear Dynamical Control Systems*, Springer-Verlag New York Inc., 1990.
- [53] A. J. van der Schaft et al., *Variational and Hamiltonian Control Systems*, Springer-Verlag, , New York, 1987.
- [54] H. Goldstein, *Classical Mechanics*, Addison-Wesley, Reading, Mass., 1950.
- [55] E. T. Whittaker, *A Treatise on the Analytical Dynamics of Particles and Rigid bodies*, 4th edition, Cambridge University Press, Cambridge, 1959.
- [56] A. E. Bryson and Y. C. Ho., *Applied Optimal Control*, New York: Hemisphere, 1975.