A Framework of Hierarchy for Neural Theory

Oliver R. Vellacott

Thesis submitted for the degree of
Doctor of Philosophy
University of Edinburgh
October 1991
The research presented in this thesis is original and was conducted by the author.

Acknowledgements

This work is an unconventional approach to neural networks and goes against the mainstream of neural research. Because of this, I am indebted to Prof. Peter Denyer for his open-minded and totally unprejudiced attitude to the field. This has been vital. However, the freedom I have been afforded has been tempered with much needed restraint. In all, he has been a consistent source of stimulus and I have greatly enjoyed my time in this department.

Dr Alan Murray’s wide knowledge of the field has been of great value in relating this approach to previous work. Dr David Renshaw has been a never-failing source of encouragement. His enthusiasm for a project perhaps a little out of place in an engineering environment has been invaluable.

Finally, I am indebted to my fellow students for their constructive criticism of my work. I would like in particular to thank Mr Ken Sutherland for many fruitful discussions, and his unerring ability to maintain the morale of our group.
Abstract

There is currently no generally-accepted theory explaining how neural systems realise complex function. Indeed, it is believed by some that neural systems are fundamentally opaque. A framework of hierarchy is proposed as the basis of neural theory. By the application of hierarchy to neural systems it is possible to explain how complex function is computed. At the primitive (hardware) level it is only possible to understand the computation of primitive functions. To understand the computation of higher level function it is necessary to abstract primitive function, via an arbitrary number of intermediate levels of complexity, to the appropriate level of abstraction. Application of the framework is facilitated by a software tool which implements a specification as a neural system, to which training can then be applied. This specification is hierarchical, and is described in a fully distributed, object-oriented style. Networks constructed by this method are not restricted to any of the traditional neural models. The class of topologies which may be implemented is unrestricted. The framework is applied to the recognition of number-plates. This practical demonstration shows that (a) hierarchy enables neural computation of complex function to be understood; (b) the application of hierarchy allows the integration of specification and learning as methods of implementation; and (c) the framework facilitates the scaling-up of neural systems.
## Table of Contents

Chapter 1: Introduction ............................................................................. 1  
1.1 The Scarcity of Neural Theory ................................................... 1  
1.2 Artificial Neural Nets ................................................................ 2  
1.3 Overview ........................................................................ 3  
  1.3.1 A Summary of the Thesis .................................... 3  
  1.3.2 A Strategy for Substantiation of the Thesis ........... 4  
Chapter 2: Foundations ................................................................... 5  
  2.1 Basic Concepts ................................................................ 5  
    2.1.1 Neurons ............................................................ 5  
    2.1.2 Synapses ............................................................ 6  
    2.1.3 Connectivity Patterns .......................................... 7  
    2.1.4 Hierarchy .......................................................... 8  
    2.1.5 Formation ......................................................... 10  
    2.1.6 Very Artificial Neural Networks ....................... 11  
  2.2 A Brief History of Neural Nets ........................................ 13  
    2.2.1 Early Work ....................................................... 13  
    2.2.2 Single and Multilayer Perceptrons .................... 14  
    2.2.3 Associative Memories ......................................... 16  
    2.2.4 Classification and Clustering Models .................... 16  
    2.2.5 Recurrent Networks ........................................... 18  
    2.2.6 Hybrid Models ................................................... 19  
  2.3 State of the Art ...................................................................... 19  
    2.3.1 A Multitude of Models ....................................... 19  
    2.3.2 The Lack of Theory ........................................... 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3 What a Neural Theory might look like</td>
<td>21</td>
</tr>
<tr>
<td>2.3.4 Where this thesis fits in</td>
<td>21</td>
</tr>
<tr>
<td>Chapter 3: Deducing Neural Theory from Related Disciplines</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Connectionist Expert Systems</td>
<td>23</td>
</tr>
<tr>
<td>3.1.1 Expert System Concepts</td>
<td>24</td>
</tr>
<tr>
<td>3.1.2 Structure</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3 Dynamics</td>
<td>27</td>
</tr>
<tr>
<td>3.1.4 Primitive vs. High Level</td>
<td>29</td>
</tr>
<tr>
<td>3.1.5 Specification vs. Learning</td>
<td>30</td>
</tr>
<tr>
<td>3.2 Distributed Systems</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1 The Motivation for Distribution</td>
<td>31</td>
</tr>
<tr>
<td>3.2.2 The Neural Metaphor</td>
<td>32</td>
</tr>
<tr>
<td>3.2.3 Autonomy and Control</td>
<td>33</td>
</tr>
<tr>
<td>3.2.4 Communication</td>
<td>34</td>
</tr>
<tr>
<td>3.3 Object-Oriented Systems</td>
<td>35</td>
</tr>
<tr>
<td>3.3.1 Motivation</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2 Object-Oriented Concepts</td>
<td>35</td>
</tr>
<tr>
<td>3.3.3 'Pure' Object-Oriented Modelling</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Non-Neural Connectionism</td>
<td>37</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>37</td>
</tr>
<tr>
<td>Chapter 4: A Framework for Neural Theory</td>
<td>38</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>38</td>
</tr>
<tr>
<td>4.1.1 The Concept of Levels</td>
<td>40</td>
</tr>
<tr>
<td>4.1.2 Abstraction and Implementation</td>
<td>41</td>
</tr>
<tr>
<td>4.1.3 Levels for Neural Representations</td>
<td>42</td>
</tr>
<tr>
<td>4.2 The Framework</td>
<td>43</td>
</tr>
<tr>
<td>4.2.1 Functions</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2 Connections</td>
<td>44</td>
</tr>
</tbody>
</table>
4.2.3 Interfaces ................................................................. 45
4.2.4 Instances ............................................................... 47
4.2.5 Summary ............................................................... 48

4.3 A State-Oriented Analysis of the Framework ..................... 48
  4.3.1 Foster’s State-Sequence Characterisation of Function ....... 49
  4.3.2 Hierarchical State-Sequences .................................... 52
  4.3.3 A State-Oriented Description of the Framework ............... 52
  4.3.4 State-Oriented Abstraction and Implementation ............... 55

4.4 Biological Considerations ............................................. 57
  4.4.1 Genetic Encoding of Hierarchical Specifications ............. 57
  4.4.2 Hierarchical Learning ............................................. 58

4.5 Discussion on Representations ....................................... 59
  4.5.1 What is a Representation? ....................................... 59
  4.5.2 What representations are natural to neural realisation? ....... 61

4.6 Summary ...................................................................... 62

Chapter 5: ANNECS: A Neural NEtwork Compiler and Simulator 63

5.1 Introduction ............................................................. 63

5.2 Features of ANNECS .................................................. 64
  5.2.1 Visualisation ......................................................... 64
  5.2.2 Libraries ............................................................. 65
  5.2.3 Macro Expansion .................................................... 66
  5.2.4 Specification ........................................................ 67

5.3 Compilation: Formation by Specification ......................... 70
  5.3.1 Compilation Method #1: Flattening ......................... 70
  5.3.2 Compilation Method #2: Resolution for High Level Simulation 73
### Chapter 6: Case Study: Automatic Numberplate Recognition

#### 6.1 Introduction

#### 6.2 Basic Image Processing Techniques

- **6.2.1 Image Capture**
- **6.2.2 Edge Detection**
- **6.2.3 Thresholding**
- **6.2.4 Segmentation**
- **6.2.5 Measurements/Classification**
- **6.2.6 Miscellaneous Techniques**

#### 6.3 Neural Techniques in Image Processing

- **6.3.1 Neocognitron**
- **6.3.2 Silicon Retina**
- **6.3.3 Connectionist Models**
- **6.3.4 Self-Organisation in Primitive Vision**
- **6.3.5 Grossbergian Boundary and Feature Contour Systems**
- **6.3.6 Head-Centred Frame of Reference**
- **6.3.7 Binocular Disparity**

#### 6.4 Overview of Numberplate Recognition

- **6.4.1 Requirements for Numberplate Recognition**
- **6.4.2 Problems involved in Numberplate Recognition**
- **6.4.3 Potential Solutions**

#### 6.5 Conventional Numberplate Recognition with near-100% accuracy

- **6.5.1 Introduction and Results**
6.5.2 Thresholding ................................................................. 96
6.5.3 Cleaning ........................................................................ 99
6.5.4 Edge Detection .............................................................. 99
6.5.5 Object Detection ........................................................... 99
6.5.6 Object Filter ............................................................... 100
6.5.7 Character Classification ............................................... 101

6.6 Neural Numberplate Recognition ........................................ 102

6.6.1 Introduction .................................................................... 102
6.6.2 Neural Local Adaptive Histogram-Based Thresholding .................. 103
6.6.3 Neural Edge Detection ..................................................... 107
6.6.4 Neural Character Location ............................................. 108
6.6.5 Neural Character Classification ...................................... 109

6.6.6 Summary ....................................................................... 114

Chapter 7: Conclusions .......................................................... 115

7.1 A Brief Review ................................................................. 115
7.2 Conclusion: Hierarchy is Foundational to Neural Theory .............. 117
7.3 Speculation: Hierarchical Learning ...................................... 118
7.4 Directions for Future Research .......................................... 119

References .............................................................................. 121

Appendices ............................................................................. 143

Appendix A : Formal Presentation of Abstraction/Implementation ........ 143
Appendix B : Neural Specification of Local Adaptive Histogram-based Thresholding ............................................................... 144
Appendix C : Neural Specification of Edge Detection ...................... 149
Appendix D : Neural Specification of Character Classification .......... 150
Appendix E : 'C' Implementation of Histogram-based Local Adaptive Thresholding .......................................................... 152
Appendix F: Published Work ................................. 158
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Biological Neuron</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Artificial Neuron (with sigmoidal activation function)</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Biological Synapse</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>A Hypothetical Artificial Neural Network</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Use of Specification in Constraining Search Space</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Structural Similarity between Neuron and Rule</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Structural Similarity between Neural Models and Rule Bases</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Relationship between Rule-Based Uncertainty Management and Neural Dynamics</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Relationship between levels, abstraction and implementation</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Concepts of the Framework: Definition of a Function</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Definition of ‘if...then...else’ function type</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Definition of ‘image’ connection type in terms of quadrants</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Definition of multiple functions in terms of a common instance</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>Neural implementation of exclusive-or function</td>
<td>51</td>
</tr>
<tr>
<td>4.7</td>
<td>Using state-sequences to describe hierarchy</td>
<td>53</td>
</tr>
<tr>
<td>4.8</td>
<td>Interface of some hypothetical function and state-sequence describing</td>
<td>54</td>
</tr>
</tbody>
</table>
4.9 Example implementation of hypothetical function ........................................ 55

4.10 State-sequence characterisation of functions which form a valid implementation of the hypothetical function ........................................ 56

4.11 Illustration of the way in which a specification might be genetically encoded ........................................ 58

4.12 Illustration of how a network may develop from a genetically- encoded hierarchical specification ........................................ 59

5.1 Icons of Function and Connection Types ........................................ 65

5.2 Example of Hierarchical Menu Layout: Functions ................................ 67

5.3 Generation of textual description for function containing nested if...then...else ........................................ 68

5.4 Stages in compilation of if...then...else ........................................ 72

5.5 Diagram showing compilation of neural specification by Netlist-of-Netlist Formation ........................................ 73

5.6 High Level Simulation of Image Thresholding Function ....................... 75

5.7 Specification of Simple Robot Controller in ANNECS ......................... 76

5.8 Compiled Simple Robot Controller ........................................ 77

5.9 Alternative Specification of Simple Robot Controller ......................... 78

6.1 Histogram and potential thresholds for a block of an image .................. 85

6.2 Main stages in conventional numberplate recognition algorithm ........... 87

6.3 Illustration of difficulty in locating numberplate ................................ 93

6.4 Images at main stages in conventional numberplate processing ............ 97

6.5 Bar chart showing relative execution times of stages in processing ........ 98
6.6 Main stages in potential neural numberplate recognition system .......... 104

6.7 Top Level Specification of Neural Numberplate Recognition System ................................................................................... 105

6.8 An image thresholded by a histogram-based neural implemention ........ 106

6.9 An edge-detected image produced by a specified neural system .......... 108

6.10 Features learnt from unspecified initial conditions ...................... 111

6.11 Features after learning from specified initial conditions .................. 113
Table of Tables

2.1 Correspondence between neural models and classical algorithms .......... 17
Chapter One

Introduction

1. The Scarcity of Neural Theory

The biological nervous system is capable of remarkable function, yet how it does this is an enigma. Every creature, from the cockroach to Man, depends utterly on its brain for its survival. By means of their nervous systems these creatures hunt, feed, avoid predators, find a mate and play. These functions rise to astoundingly complex levels (consider a lion intercepting its prey); yet this subtle and diverse function arises, in some mysterious way, out of the interaction of vast numbers of relatively simple and homogeneous nervous components.

The components from which nervous systems are constructed are fairly well understood. Moreover, the principles by which these primitive nervous elements may be interrelated are conceptually simple. Not only this, but for some systems, the precise way in which these components are put together – their wiring diagram – is known. Yet even this description of what is occurring at the hardware level does not explain how complex function emerges. No coherent and general explanation of how higher functioning arises out of primitive neural interactions has yet been described. Neural systems remain, in reality, black boxes.

Neural Theory is the set of principles by which the realisation of higher function by neural systems may be understood. These principles would explain, for example, how recognition of prey, calculation of trajectory and interception of a target may all be realised by the appropriate interconnection of neurons. By this test, neural theory is undoubtedly scarce.
Chapter One - Introduction

2. Artificial Neural Networks

The development of Artificial Neural Networks is inspired by the capabilities of biological neural systems. Artificial neural systems employ a medium of computation (or representation) which approximates that adopted by nature. It is thought that, by so doing, functions natural to brains will be reproduced artificially. Whether this is possible is not a concern of this thesis. However, it is argued that lack of neural theory is responsible for the failure of artificial neural network development to achieve this aim.

Hitherto, the approach has been to assemble a certain configuration of neural components (e.g. by lining the neurons up in rows and fully interconnecting between adjacent rows), defining rules by which the system may evolve, presenting training and test data, and observing what happens. This empirical approach has achieved moderate success in a relatively small class of applications. This failure to realise the expectations aroused by biological capabilities is largely due to an irrational emphasis on learning, at the expense of specification. This imbalance has come about as a result of the historical development of synthetic mediums of computation. The microprocessor is so much more suited (than a neural network) to silicon implementation and programming, that the main justification for the use of neural networks has been that they have learning abilities.

This preponderence of empirical derivation, however, is not found in biological neural systems. Approximately 70% of the genetic code, according to some estimates, is devoted to specifying brain composition.\(^{38-44}\) One theme of this thesis is that it is this specification which enables learning to take place, in artificial as well as biological systems. Whether or not this is true, it is clear that the empirically-biased line of research has attempted to sidestep the issue concerning how higher function emerges within neural systems.

Indeed, perhaps the most striking feature of the current state of the art in neural nets is the lack of any unifying theory. A multitude of models and successful specialised applications abound, each with its own, piecemeal ‘theory’ explaining how it works. No overarching principles exist, however, for bringing together these diverse models and enabling a common understanding. This lack of theory was pointed out by von Neumann as far back as 1956\(^2\) yet his paper seems as relevant today as it was then. More recently,
Patricia Churchland has described her own search for neural theory, and has concluded that none is yet available, though one is much needed. Her book, *Neurophilosophy*, gives an excellent description of the role and requirements of a theory of neural networks.

This lack of theory is stifling the development of artificial neural nets. The field seems to have become trapped in a local minima of its own making. The empirical bias has even led some researchers to believe that there is no theory of neural nets. Perhaps this is correct! Maybe there is no method for saying what the $i$th neuron in the $j$th layer of a multilayer perceptron actually does. Maybe there is no computational theory underlying neural cognition which can identify the role of each neuron and each connection. Perhaps the only ‘explanation’ of neural processing is some principle such as *Neural Darwinism* which explains how a system evolves but not how it performs its function at any non-primitive level. Is it sufficient to continue to treat neural systems as black boxes, unconcerned that there is no framework for understanding how they realise complex function? The current attitude seems to be that not only are neural systems not understood, but that it is not necessary to understand how they work. The poor delivery of results belies this attitude.

These objections seem unreasonable. Theory exists to explain the operation of many other areas of the universe. Why should neural computation be different? The field of neural nets cannot make do, as it may have assumed in the past, without theory. It cannot develop outside the confines of a framework which explains how higher functions are realised by neural processes.

### 3. Overview

**A Summary of the Thesis**

This thesis describes principles which enable understanding of the way in which neural systems realise complex function. These principles form a framework within which complex functioning of neural systems may be comprehended. The key concept underlying this is **hierarchy**. Thus, the framework provides a method for abstracting/realising function at one level of complexity as functioning at another level of complexity. At the primitive (neural hardware) level it is only possible to understand the
computation of primitive functions. In order to understand the computation of higher functions it is necessary to abstract primitive functioning to higher levels of abstraction. This explains the emergence of higher level function within neural systems. Results of applying this framework should include:

(a) understanding of how neural systems realise complex function;

(b) the integration of specification and learning as methods of construction;

(c) the scaling up of artificial neural nets.

A Strategy for Substantiation of the Thesis

Chapter Two makes a close examination of the field and looks for evidence in past neural research to support the thesis. Adjacent fields are then searched, in Chapter Three, for metaphors of neural computation. The purpose in this is to derive understanding of neural computation from the extant understanding of the metaphors.

Chapter Four then draws on these insights to propose a framework of hierarchy for neural theory. ANNECS is a software tool which facilitates the application of this framework, and is described in Chapter Five. This tool ‘compiles’ a hierarchical, object-oriented-style, distributed specification to a neural system which realises that specification.

Chapter Six describes the application of the framework, by use of ANNECS, to a real engineering problem within the field of image processing. Several stages in the recognition of a numberplate are implemented as a neural system and these experiments are used to determine the usefulness of the framework. Finally, Chapter Seven discusses the extent to which the thesis has been substantiated by the work carried out.
This chapter reviews the field of neural nets and, in particular, work relevant to this research. The basic concepts of neural nets are first described, followed by a brief history of the development of the field. A resumé of the state of the art in neural nets is then described, indicating where the research described in this thesis fits in.

1. Basic Concepts

Neurons

The biological neuron consists of a cell body and extensions from that body, along which it receives and transmits signals. The extensions consist of dendrites, along which input is received, and an axon, along which output is sent, though not all neurons have both. At birth, a human has virtually all the neurons it will ever have, approximately $10^{11}$, give or take an order of magnitude. Between 15-85% of these die in infancy in a method that appears to be part of development and in some way programmed in. There is no known reason for this though Neural Darwinism (described later in this chapter) offers one explanation, as does the neural generative process described in this work. A neuron operates at about 100 Hz – much slower than typical electronic devices. Thus, for the brain to recognise a face in, say, 1 sec there must be no more than 100 synaptic steps between retinal sensing and perception. This is known as the 100-step rule. Artificial neurons are modeled more or less on the biological neuron though, as we shall see later, this modelling is extremely crude in most networks. Our interest is not in the biological implementation of neural function, but in what that function is. Broadly speaking, a neuron performs summation and averaging of its inputs. In the short term, it acts as an all-or-none processor, emitting a constant-sized pulse whenever its state of excitement exceeds a threshold. Thus, in the long term, a neuron may be considered to
have a continuous-valued activity: its rate of firing over some period of time, which approximates the average of its inputs received during a preceding period.

### Synapses

Synapses are points of contact between neurons. A synapse normally forms a link between an axon and a dendrite but, in some cases, connects an axon to an axon or a dendrite to a dendrite. 5,6 Basically, there are two types of synapse: electrical and chemical. 8 Electrical synapses either transmit a pulse unattenuated from a sending axon to a receiving dendrite or act by inductance between two neighbouring axons. Chemical synapses perform transfer of a signal reaching the end point of an axon to a receiving dendrite. This signal is transferred, as the name implies, chemically (by means of a neurotransmitter) and its amplification/attenuation is dependent on both the type of neurotransmitter and the amount that is present. This effect is approximated in artificial neural nets by the multiplication of a synapse weight (which models a combination of type of neurotransmitter and amount present) by the output from the sending neuron.
There are approximately five thousand synapses on a mammalian motor cell and ninety thousand on a Purkinje cell (a type of neuron) in the human cerebellar cortex. In all, there are about $10^{15}$ synaptic connections in the human central nervous system, give or take an order of magnitude. The effect of a signal arriving at a synapse will be to either excite or inhibit the receiving neuron, depending on the type of synapse.

**Connectivity Patterns**

Neurons and synapses in the flatworm and in Man are essentially the same. What makes a human immensely superior in intelligence to a flatworm is the manner in which its neurons are connected. The principles underlying neural networks are blindingly simple—at least conceptually. Yet from these simple processing elements, connected via amplifying/attenuating contact points, can be produced remarkable abilities. The power exists in the patterns of connectivity. As yet, there is no systematic method for deriving a neural connection pattern to implement a given function. It is this problem that the research described in the following pages addresses.
There are about twenty neural 'circuits' that have been intensively studied and are well understood. For example, the pattern of connectivity which causes lateral inhibition between sensory neurons – thus giving rise to edge-detection in mammalian retinas – is well understood. Similarly, the 'circuits' enabling reflex actions in the sea hare, processing of sonar return signals by bats and location of prey by owls are well documented.

Hierarchy

Biological neural nets are not 'flat'. There is hierarchy present in even the most simple brains. The human brain is composed of several major components such as the cerebral cortex, the thalamus, the cerebellum, and so on, which seem to have distinct roles. Within these main components, clusters of neurons are observable and, within these clusters, other clusters. Similarly with synapses, there exist groupings – for example, in
the optic nerve. It seems clear that hierarchy does exist in the brain, a principle whose application to artificial neural nets is a central thrust of this work. Not only are modules observable in the brain but distinct functions seem to be associated with these modules. For example, reasonably precise maps of cerebral cortex function have been made. At present, however, there is little notion of hierarchy in artificial neural nets. *Neural Darwinism* (see section 2.2.6) is one model containing implicit hierarchy, yet then only at two levels. Current neural analysis seems to be focused almost entirely at the primitive (i.e. synapse-neuron-synapse) level. This thesis suggests that the theory — and hence understanding — of neural nets may be advanced by the introduction of hierarchy.

Some work has attempted the creation of modules within nets but, in general, this ‘modularity’ seems restricted to the integration of flat networks of differing models.
Biological neural nets are formed by two methods that seem to work closely together:

**A. Genetic Specification:** to some (unknown) extent the structure of the brain is derived from the genetic code. This explicit method of construction is all but entirely lacking in artificial neural networks and is the primary novelty of this research. Hitherto, researchers may have assumed that the amount of information required to specify brain structure to a significant extent must be enormous and indeed, if the specification is made at the primitive level then this is true. However, a hierarchical specification, as seems a sensible description from which to form a hierarchical brain, is orders of magnitude more compact than a primitive specification. Research has shown that connection patterns are, at least partly, genetically determined.

Use of explicit hierarchical specifications from which to generate artificial nets has not been performed. However, what does constitute a specification in the formation of artificial neural nets is the network topology. This topology is highly specialised not only to the neural model but also to the application of the model. For example, the number of layers and the number of neurons in each layer in a multilayer perceptron (see section 2.2.2) seem to be chosen in an ad hoc manner guided by the constraints of the application. This is in spite of the fact that it is this initial network structure which enables the learning process to succeed. Indeed, the choice of a feedforward net itself is a form of specification. Unfortunately, this choice of topology is not guided by any theory and thus is often made without any understanding of what will best enable learning. A theme of this work is that *a priori* knowledge should be imparted to a model in a more intelligent and meaningful way.

**B. Empirical Derivation:** neural plasticity with training has been shown to occur in the somatosensory mammalian cortex and in the hippocampus. As it would be a mistake to claim function is entirely learnt so it is a mistake to claim that function is entirely specified. *Hebbian learning*, the increase in size (weighting) of a synapse when both sending and receiving neurons are active, has been shown to occur in some areas of the

---

Please note that the use of the word 'genetic' throughout this thesis bears no reference to *genetic algorithms.*
In artificial neural nets, empirical data is used to derive desired behavior in one of three ways:

A. Supervised Training: an external ‘teacher’ knows the desired response and inputs an appropriate error signal into the net.

B. Unsupervised Training: internal clusters/categories of the input data are formed which typically compress the amount of input data that must be processed at some higher level.

C. Self-Supervised Training: the network monitors its own performance and, on that basis, feeds an error signal back to itself.

The net effect in each case is to attempt to modify the synaptic weights and, more rarely, the network topology or neuron thresholds so as to develop the desired behavior. This ability is probably considered the most important aspect of neural nets and it is here that most attention seems to have been focused.

**Very Artificial Neural Networks**

The brain is the inspiration for artificial neural nets and yet the analogy is inevitably crude and some artificial models cannot be considered worthy of the term *neural*. Here follows a list of some of the more obvious limitations of artificial models:

The difference in size is immense. The brain is $10^9$ times larger than typical artificial neural nets. A 100 MIPS computer would take somewhere between ten and a hundred years to simulate, even crudely, the processing that takes place in the brain during one second.

Artificial neurons are gross approximations of biological neurons. There are five known distinct types of biological neurons: Purkinje, Golgi, Basket, Granule and Stellate cells. Presumably each of these has a distinct role to play.

At least forty different chemicals are known to be active in the brain. Eleven of these have been shown to be neurotransmitters; the rest are probably active in performing some more or less global control. Given that there is normally a good
reason for the way nature does things, it seems probable that each of these chemicals acts in its own unique way for a reason. Most artificial neural models cannot remotely approach the emulation of these effects.

The multiplication of synapse weight by cell output is unlikely to be an entirely accurate model of the chemical synapse.

There seems to be little modelling of electrical synapses in artificial nets, or indeed any understanding of what part these might play.

The volume of empirical data presented to a human during development is unimaginably vast compared to the restricted data set on which artificial neural nets are trained.

There is currently no meaningful formation of neural nets from genetic specifications.

Learning processes that have been shown to 'work' are almost certainly not approximations of biological dynamics.\(^{24}\)

Changes in synaptic strength in biological nets seem to be dependent on factors such as location, chemical environment, cell type, neurotransmitter used, in addition to activity which is the only one of these factors of which most artificial learning algorithms are a function.

The brain is, at least to some extent, modular; artificial models have little inherent concept of hierarchy.

The continuous-valued neuron output used in neural simulations is but an approximation of the pulse-firing of biological neurons.

The list of gross approximations seems endless! With all these shortcomings in mind, the 'neuralness' of artificial neural nets seems decidedly weak. However, artificial neural nets do capture the essential features of biological nets and it is perhaps necessary to assume that these additional factors are but refinements to a basic theory which may be derived from known neural principles. Recently, serious attempts have been made to overcome
some of these shortcomings and, in particular, to improve the temporal characteristics in some models.4,16,25-27

2. A Brief History of Neural Nets

This section sketches the development of the field of neural nets. It considers models in the following categories: Early Work, Perceptrons, Associative Memories, Pattern Classifiers, Recurrent Models and Hybrid Models.

Early Work

The study of the human central nervous system dates back to antiquity 28 but some of the first work in artificial neural nets was published in 1943 by McCulloch and Pitts. 29 In this, they proposed their *threshold logic units* and showed that any effective procedure 30 (that is, any functioning that can be precisely described) could be implemented by a network of these units. Threshold logic units are somewhat different from today's artificial neurons and at that time no learning algorithms existed for them. However, as was noted by von Neumann their result concerning the computational power of neural networks is significant. 2 It shows that a neural net has at least the computational power of a von Neumann (or Turing) machine 31 and, since Farley and Clark showed in 1954 that a von Neumann machine has at least the computational power of a neural net, it must be deduced that neural nets and von Neumann machines are equivalent in computational power. The experimental results of this thesis support this conclusion and even suggest that modern programming representations, traditionally implemented on von Neumann machines, in fact map more naturally onto neural architectures. Symbolic and neural architectures are equivalent in computational power, though neural representations are probably more natural models of the world, as is suggested in section 4.5.

Von Neumann's paper *The General and Logical Theory of Automata* 2 discusses the results of McCulloch and Pitts and, in spite of its title, actually laments the lack of any theory of neural nets. His paper seems as relevant today as it was then. Other work by von Neumann showed that a *threshold logic*-type network could be self-reproducing. 32 Shortly after this, threshold logic units became of interest as a potential means of
constructing computers, even resulting in the implementation of one small machine.\textsuperscript{33-35} Also at this time, Hebb published his landmark book on biological neural learning, now known as \textit{Hebbian learning}\textsuperscript{22} which has since been shown to generate models that perform visual feature detection.\textsuperscript{36-39}

**Single and Multilayer Perceptrons**

The single layer perceptron, a more ‘biological’ neural model than that of McCulloch and Pitts, was first proposed by Rosenblatt in 1958.\textsuperscript{40} It consists of a single layer of processing units similar to the artificial neuron presented in figure 2.2. In neither the single layer nor the multilayer perceptron are there any connections between neurons in the same layer.\textsuperscript{41} The computational power of the single layer perceptron has been extensively analysed and shown to be limited to the computation of linear separable functions.\textsuperscript{42} This, after overexaggerated claims for the perceptron’s power, led to an unwarranted period of disenchantment with neural nets. This was in spite of the fact that Minsky and Papert, in demonstrating the single layer perceptron’s limitations, themselves pointed out that a multilayer perceptron with feedback as well as feedforward connections had the computational power of a von Neumann machine.\textsuperscript{42} The initial popularity of the single layer perceptron was due to the existence of learning algorithms, the perceptron convergence algorithm\textsuperscript{41,43} and the LMS algorithm,\textsuperscript{43} which were proved to converge to a correct linear classifier if such a classifier could exist.

The multilayer perceptron gained in popularity with the rediscovery of \textit{backpropagation},\textsuperscript{24,44,45} first reported in 1974 \textsuperscript{46} and still probably the most popular learning algorithm. This is a form of supervised learning and basically computes the difference between the actual output and the desired output and, working backwards from the output nodes to the input nodes, modifies synaptic weights and neuron thresholds appropriately. It has not been possible to prove that a multilayer perceptron converges to correct classification using backpropagation. Indeed, the algorithm will often become trapped in \textit{local minima} in the classification space. Largely because of this, many variants on backpropagation have been proposed.\textsuperscript{47-49} This thesis advocates placing the initial network in the region of the global minimum by means of specification, and allowing learning to advance the net to the exact global minima (see figure 2.5).
Recent work has demonstrated the problems that occur when multilayer perceptrons are scaled up in size.\textsuperscript{50,51} It has been proved that the time taken for a single layer perceptron to learn an arbitrary, linearly-separable function grows exponentially with the number of inputs. Similarly, the search space for a multilayer perceptron grows exponentially with number of inputs, and thus the problem of learning an arbitrary classification is NP-complete.\textsuperscript{52} This supports this thesis's argument that a stronger element of specification is needed in initial network configurations before it will be possible to scale them up. Steps in this direction have been made by specifying and fixing weights in initial layers.\textsuperscript{18,53} In spite of these limitations, however, backpropagation has proved useful for many applications.\textsuperscript{18,54-56} Several alternative training algorithms have been proposed for multilayer perceptrons.\textsuperscript{18,50,57,58}
Associative Memories

Early work in this area was performed by Kohonen amongst others.\textsuperscript{59-61} The best-known neural model was proposed by Hopfield\textsuperscript{62} and achieves energy minimisation based on the outer product rule. This model has been exhaustively analysed\textsuperscript{63-66} and shown to be of limited capacity and inefficient in its use of hardware. Other associative memory models are the Hamming or Unary Net\textsuperscript{67,68} and Sparsely-Distributed models.\textsuperscript{69} Neither of these models suffer from the efficiency and capacity limitations of the Hopfield net. It could be argued that the reason for this is that the element of specification is far stronger in these models than in the Hopfield model, which has an unconstrained topology. More \textit{a priori} knowledge concerning associative memory is built into the Hamming and Sparsely-Distributed models in the form of network structure.

Classification and Clustering Models

Classification is an essential function for real-time response in, for example, vision – the experimental area for this research. In theory, a three-layer multilayer perceptron will perform any classification though in practice, as already described, learning algorithms cannot guarantee to converge to such a classifier. An excellent review of classifiers has been performed by Lippmann\textsuperscript{70} and analyses of their relative merits have been made.\textsuperscript{18,50} Various taxonomies of neural classifiers have been attempted.\textsuperscript{71} These typically divide classifiers into those that take binary and those that take continuous-valued input. Beneath that, classifiers may be further subdivided into those that are trained under supervision and those that are unsupervised. In most classifiers connections are predominantly feedforward; exceptions to this are the ART, Hopfield and Darwin II models. In most cases, the model implements a classical algorithm (see table 2.1).\textsuperscript{72} This provides further evidence that \textit{a priori} knowledge is available in most situations which could be incorporated in the model in the form of more explicit specification.

\textbf{Supervised Classifiers} include single and multilayer perceptrons, Hopfield, Hamming, RCE,\textsuperscript{73} Feature Map\textsuperscript{18} and High-Order networks.\textsuperscript{74} Of most interest to this research are High-Order network models. These contain more complex operations at each node than the conventional summation and averaging. Since it has been shown that these higher
level functions may themselves be implemented as neural nets, a High-Order network may be considered as a network of clusters of neurons; in other words, a network model containing hierarchy (though this is not in fact how the model was conceived). It is the presence of hierarchy in neural models which is a central concern of this thesis.

**Unsupervised Classifiers** include ART\(^\text{75,76}\) (Adaptive Resonance Theorem), Kohonen\(^\text{61}\) and Darwin II\(^\text{4,16}\) models. Within the ART and Kohonen models clusters of neurons are formed during training; this perhaps indicates the emergence of two-level hierarchy in these models. Darwin II is probably the model most closely allied to the neural methodology advocated in this thesis. It derives from a theory called *Neural Darwinism*\(^\text{4}\) which takes a more biological approach to the generation of network structure. It does this during 'embryogenesis' by generating many clusters (typically of 100-1000 neurons) which then constitute a pool of candidate components, or a *repertoire*, from which the final network is drawn. A neural version of natural selection then takes effect with the broad principle that those clusters that are active, survive, and are strongly connected to other active clusters. Clusters that are very rarely active – and thus, it is deemed, not serving any useful function – die, as seems to occur in normal child brain development. This model shows undoubted emergence of hierarchy, though still only at two levels: clusters are formed from neurons, not from other clusters of neurons. Natural selection does seem likely to be a guiding principle in brain development, but still doesn’t explain how complex functioning occurs – how inputs are transformed to outputs, at any non-primitive level. It may explain how functions develop but doesn’t explain how those functions are neurally implemented or what those functions actually are. The theory is of especial interest to this work since the initial network and cluster structures are generated
as if from a genetic code. In fact, however, this ‘genetic code’ is not a meaningful specification describing actual structures and, as I understand it, acts so as to generate a repertoire of randomly-connected structures.

Recurrence Networks

These are models which do not contain predominantly feedforward connections. In general, they can perform time dependent tasks in addition to having the capabilities of feedforward nets, though training is more difficult owing to their (normally) less-constrained architectures. Backpropagation has recently been generalised for recurrent nets, though it still suffers from the same limitations as with feedforward nets.\(^47\),\(^77\) Hopfield and Tank have applied the Hopfield model to global optimisation problems such as the Travelling Salesman Problem, results showing yet again the problems in scaling up.\(^78\),\(^79\) Here again, specification might have a role to play in imparting knowledge obvious to a human – such as which cities the salesman should definitely visit in a certain sequence – and thus initialising and making resistant to change relevant synaptic weights.\(^80\),\(^81\) Boltzmann machines use the technique of simulated annealing in order to avoid being trapped in local minima in the search space during training.\(^82\)-\(^84\) The model suffers from training times that are too long for it to be of practical use.

Cellular Automata\(^32\) also fall into the class of recurrent nets, as do Winner-Take-All nets.\(^85\) ART, developed by Grossberg and Carpenter, is also highly recurrent.\(^75\) Training seems to be enabled in ART because its recurrent connections are accurately specified and thus highly constrained. It seems that many neural models have avoided recurrence because the added lack of constraint limits the learning procedure. It has not been possible to generate any realistic general purpose learning algorithm for recurrent nets owing to the unconstrained topology. This only supports the argument advanced by this thesis that specification is necessary in order for learning to succeed. This thesis advances a methodology for incorporating meaningful specification (as opposed to just a particular type of topology) in recurrent nets. A newcomer to the field of neural nets might be struck by the artificial constraint of layered feedforward nets, observing that biological nets do not seem to impose such a constraint. What is lacking is a technique for mapping
meaningful specification into network structure, thus enabling learning. Layering with feedforward connections does seem to be an arbitrary and artificial constraint.

Hybrid Models

These are systems that integrate symbolic and ‘subsymbolic’ (i.e. neural) methods of computation. Ideally, such systems combine the strengths of each method so as to compensate for the weaknesses of the other.\textsuperscript{86,87} It has been claimed that symbolic and subsymbolic methods of computation are complementary: tasks not suited to symbolic models, for example low level perceptual processes, are suited to neural methods; tasks such as \textit{planning}, whilst apparently not suited to neural models, are suited to symbolic modelling. \textsuperscript{86} Hybrid systems should, it is claimed, enable a cross-fertilisation between these two radically different methods of computation.

Problems to which hybrid systems have been applied include classification,\textsuperscript{88} speech recognition,\textsuperscript{89} noun-phrase understanding,\textsuperscript{90} diagnosis of back-pain\textsuperscript{91} and optimisation of knowledge-based inference.\textsuperscript{92,93} Recent work has also performed the translation of symbolic representations into functionally equivalent subsymbolic representations at a primitive, one-to-one level; and vice versa.\textsuperscript{94,95} This has implications for this thesis in that it shows that in principle it is possible to construct a neural net from a specification.

3. State of the Art

A Multitude of Models

A striking feature of the current scene is the number of independent and apparently-unrelated models. About thirty distinct models can be identified, each suited to a small area of tasks, each with its own \textit{ad hoc} ‘theory’. This ‘theory’ explains how the model works in its own relatively small domain: how it learns, its limitations, its computational power, its efficiency, and so on. The less computationally-powerful models normally guarantee convergence during training – for example, with the single layer perceptron and the Hamming net. The more computationally-powerful models have less constrained architectures and hence the potential to perform more powerful tasks but are limited in
their convergence during training. The exceptions to this are those recurrent architectures containing closely-specified connections – for example, Grossberg’s ART. It is this added element of specification which enables successful learning. Arriving at an appropriate network topology, however, seems at best to be a ‘black art’. In practice, the topology seems to be derived through the designer’s intuitive incorporation of a classical algorithm into the model’s architecture. The ART model, for instance, seems to have been constructed with adaptive filters in mind.

Almost none of these models scale up well. For most, the training problem is NP-complete. Remarkably, virtually no work has attempted construction of an initial network architecture from a meaningful specification. Since it is known that about seventy percent of human DNA is devoted to specifying generation of the central nervous system it seems absurd that no work has been spent on applying this method to constructing artificial neural nets. The reason for this must be simply that it is not known how networks could be specified. The key concept that is missing is hierarchy.

Most results delivered by neural nets are only new in that they are learnt. The vast majority of neural applications are functionally equivalent to well-understood, classical algorithms. What is novel, is that these functions have been learnt. It has been argued that the only reason these models do work, however, is because they were designed, however intuitive the design process, to implement known classical algorithms. That design process was effectively the incorporation of specification into the model. This thesis advocates making that specification explicit and meaningful.

The Lack of Theory

Perhaps the most striking thing about the state of the art is the absence of any unifying theory. What theory that exists is piecemeal explanation here and there, not overarching principle that would unify the diversity of models, so enabling a common understanding. Theory would unify neural applications to classification, clustering, optimisation, association, competition, control, planning and world representation. Von Neumann’s paper bemoaning lack of neural theory is as relevant today as it was when written.
What a Neural Theory might look like

The individual components of the biological nervous system are starting to be understood. Artificial neurons can approximate biological ones, however crudely. What is not known is how ensembles of neurons produce remarkable functions. Not only is it not known how a particular architecture enables, for example, an owl to intercept a zigzagging mouse, but neither is there any framework for deriving such an architecture. Current research seems to be unguided by any quest for theory, which tends to make it necessarily random and directionless. Conversely, theory-testing would guide why an experiment should be performed.

What is needed is a framework within which the role of neurons, clusters of neurons, synapses, groups of synapses, systems and subsystems can be understood: not an explanation of how the brain works but a framework within which such an explanation could be formulated. This framework would provide a methodology for deriving a neural architecture to satisfy an arbitrary specification. Such a framework would necessarily require a means of meaningfully describing and thus understanding the architecture. It would explain how neural processes operate to transform inputs to outputs and how those processes are realised by patterns of connectivity.

The evidence such a theory has to build on may yet be insufficient. What is known is the basic characteristics of primitive processors and primitive interconnections. It is known that clusters of neurons and groups of synapses exist in biological nets and that that clustering presumably serves some purpose. It is known that biological nets are divided into subsystems and modules. It is known that processes should ideally be modeled so as to satisfy the 100-step rule so that, for example, visual perception can take place in about 100 msec using biological-rate neurons.

Where this thesis fits in

This work proposes a framework for understanding neural nets by the introduction of hierarchy and modularity. With this framework in place, it is possible to generate initial network architectures that incorporate specifications described to an arbitrary degree of precision. It is thought that taking this approach to the construction of neural models will:
(a) reduce training time by meaningful topology constraint, thus placing the network in the neighbourhood of a solution within the search space (see figure 2.5); this should also allow the scaling-up of neural systems.

(b) enable understanding at arbitrary levels of representation as to how a neural net implements a given function.¹⁰¹

Constructing networks from specifications may seem to be going against the spirit of neural networks, in that their greatest virtue is that they learn. However, throughout this chapter it has been extensively argued from the literature that not only are biological nets largely constructed by specification but that existing artificial models incorporate implicit specification.
This chapter reviews work in disciplines related to Neural Nets. The understanding of a new field such as neural nets seems most unlikely to come about by spontaneous generation. Theory can emerge from new insights or by the systematic investigation of specific problems. In the absence of these, however, new theory can be formed as a result of the cross-fertilisation of related disciplines. Thus, this chapter seeks to deduce a framework for understanding neural models from theory in the related fields of: connectionist expert systems, distributed systems, object-oriented modeling and miscellaneous non-neural connectionist approaches. The purpose is to find metaphors of neural computation that may enable the proposal of a framework within which neural processing may be understood. If the metaphor is well-studied, then that should allow the application of that understanding to neural processing. This could enable answers to be given to questions such as: What is the role of a neuron within the network? What function does a cluster of neurons perform? What is the role of a cluster within the network? How is function distributed? What internal representations are natural models of the world? And so on. The preceding chapter sought to place this thesis in context within the field of neural nets. This chapter places the thesis in the context of the wider field of computation and modeling.

1. Connectionist Expert Systems

In the last two years, a large amount of attention has been focused on the relationship between neural networks and rule based systems. Results of this work have shown that some neural models approximate rule based systems, not just in what they compute but also in how they compute it, at each stage in the processing. This ‘equivalence’ has given rise to the term connectionist expert system, which is the subject of this section.
Chapter Three – Related Disciplines

Expert System Concepts

An expert system consists essentially of three elements: a rule base, an inference engine and a user interface. A rule base normally consists of Horn Clause predicates\(^{105,106}\) of the form:

\[ \text{conclusion if condition}_0 \& \text{condition}_1 \& \ldots \& \text{condition}_n. \]

The inference engine, as its name implies, performs logical inferencing from these rules. The user interface enables the user to query the system. Thus, if the rule base contained the following rules:

- \(\text{loves}(X, Y)\) if \(\text{rich}(Y)\) \& \(\text{beautiful}(Y)\) \& \(\text{male}(X)\) \& \(\text{female}(Y)\).
- \(\text{male}(\text{Jason})\).
- \(\text{female}(\text{Kylie})\).
- \(\text{female}(\text{Jane})\).
- \(\text{beautiful}(\text{Jane})\).
- \(\text{rich}(\text{Kylie})\).
- \(\text{rich}(\text{Jane})\).

– and the system is queried with:

\(\text{loves}(\text{Jason}, Y)?\)

The inference engine will try to deduce who Jason loves, on the basis of the knowledge contained in the rules. The answer obtained is that Jason loves Jane because she is female, beautiful and rich, but not Kylie because, though female and rich, she is not beautiful.

In reality, Jason would almost certainly not love someone on the basis of true/false predicates. Any real world expert system should be able to handle uncertainty.\(^{107-109}\) Hence, most contemporary expert system models compute the likelihood of various competing hypotheses, on the basis of unreliable data. Such systems seem to have been most successful in their application to medical diagnosis.\(^{110,111}\) Here, the system is asked to decide between various hypothetical illnesses, given an array of unproven symptoms.
There are four areas of uncertainty involved in calculating the likelihood of a hypothesis:\textsuperscript{107,112}

(i) **Data** – How rich is Kylie?

(ii) **Importance of data to conclusion** – Is it more important that she is rich than beautiful?

(iii) **Inference** – How reliable is this rule?

(iv) **Conclusion** – How likely is the hypothesis?

Strictly speaking, however, since the conclusion from one inference will often form the input data to another, (i) and (iv) may be considered the same area of uncertainty.

Various theories have been proposed to model one or more of these areas of uncertainty. Most widely used amongst these is a model which uses Bayesian techniques and is based on *Probability Theory*.\textsuperscript{108,113-115} In spite of its popularity,\textsuperscript{116,117} which is probably due to its solid base of theory, the Probability model has significant shortcomings.\textsuperscript{118-120} Ignorance is not modeled, and the inference step contains no analogue of neural inhibition.\textsuperscript{121} The model is restricted to binary data only and thus does not model uncertainty in data (i) above).\textsuperscript{107} Hierarchical models exist,\textsuperscript{122,123} but this ‘hierarchy’ seems restricted to partitions within ‘flat’ networks that evaluate simple predicates. The Dempster-Shafer model, a variant on the Bayesian model, was developed in order to overcome some of these limitations.\textsuperscript{124-126} Unfortunately, a side effect of this was to make the model computationally intractable.\textsuperscript{107,127}

Possibility theory, also known as *fuzzy logic*, was developed primarily by Zadeh and is based on fuzzy set theory.\textsuperscript{128,129} Unlike probability theory, possibility theory does model uncertainty in data (i) above) but not uncertainty in importance of data (ii) above). Unlike Dempster-Shafer models, it is computationally efficient and has resulted in a number of applications.\textsuperscript{110,130} Various other uncertainty management methods such as *certainty factors* have been proposed, which seem to be largely *ad hoc* techniques not grounded in theory.\textsuperscript{107}
The relevance of these models will become apparent later in the chapter. Suffice it for now to say that each model is limited in some significant respect. Probability models do not model uncertainty in data; Dempster-Shafer models seem theoretically sound but are computationally unrealistic; Possibility models do not model weighting of data; and the ‘ad hoc techniques’, though empirically derived, can give inconsistent answers.

**Structure**

The fundamental similarity between neural nets and rule bases is in their structure. A Horn Clause has the same structure as a neuron in that several (continuous-valued) inputs combine, in some way, to produce one (continuous-valued) output.

![Figure 3.1 Structural Similarity between Neuron and Rule](image)

In the same way that neurons are structurally equivalent to rules, some neural models are structurally equivalent to rule bases.\(^{131,132}\)
Dynamics

By dynamics is meant the processing of information within the model – as opposed to its static topology. It has been seen that neural and rule based models are structurally equivalent. The present concern is to determine whether they are dynamically equivalent: do neural models approximate rule based models in the way they process information? A neural model has a specific topology, weights and thresholds: are there analogues of these components within rule based models which enable us to understand their role in neural models? If so, a metaphor for neural processing may exist which illumines neural theory.
First Gallant, and then others, have shown empirically the ‘equivalence’ of some neural and rule based models. *Connectionist Expert Systems* perform evaluations in an event-driven control strategy, as do neural models. Traditionally, expert systems have employed a goal-directed order of evaluation as a control strategy. This, however, is simply a difference in *order* of evaluation, not in *what* is computed at each node. Within the connectionist expert system model, *synaptic weights* may represent uncertainty in importance of data (ii) above), *neuron thresholds* may represent the validity of a rule (iii) above: its predisposition to fire) and *cell outputs* may represent certainty of data and hypotheses (i & iv) above).

More recently, it has been rigorously proven that a multilayer perceptron approximates a Bayes optimal discriminant function. This shows that some, if not all, neural models approximate various rule based uncertainty management models. Building on this work, Lacher et al. have applied the neural learning algorithm *backpropagation* to connectionist expert systems, which they have dubbed *Expert Networks*. Other work has observed various aspects of this ‘equivalence’ between neural and rule based learning. It is beyond the scope of this thesis to enter into a detailed analysis of this work. Of interest to us, by way of results, is the role played by each component of a neural model – topology, weights, thresholds – in the expert system metaphor.

Neurons may be understood as performing primitive inferencing. Their thresholds contain some concept of the prior probability of a hypothesis being correct: the cell’s predisposition to fire. Synaptic weights perform (as they were perhaps unwittingly named) weighting of evidence. The network topology interrelates inferences so as to realise the desired global function.

A feature of expert systems is that they automatically generate explanations of how they arrived at an answer. This capability has been incorporated in neural modeling software produced by the Hecht-Neilsen Corporation and in *Neuralworks Professional*,† which give ‘explanations’ by tracing through heavily-weighted synapses back to inputs. A tool designed explicitly for the development of connectionist expert systems has also been produced.†

---

Produced by Neuralware Inc.
Primitive vs. High Level

Much work relating expert systems to neural nets has compared neural nets and expert systems containing first order predicate rules only. First order predicates are boolean functions that take no arguments, as opposed to second order predicates, which do take
arguments. Comparisons between second order predicate rule bases and neural nets have been performed\(^\text{137}\) but these models assume the existence of processes more complex than the simple summation-and-thresholding of a neuron at each node. In Gallant’s model, \(^\text{131}\) for instance, the neural part of the system implements uncertainty management; the higher level capabilities of expert systems are not implemented neurally. Higher level features of expert systems include: variable binding, variable instantiation, the use of a stack to perform recursive inference, list processing and explicit control by use of operations such \textit{cut}.\(^\text{105}\) This in no way invalidates this ‘metaphor’ of neural computation since first order equivalence alone enables the opening up of the neural ‘black box’ to see how, at a primitive (neuron/synapse) level, functions are computed. However, it is relevant to this thesis to consider how these higher level functions might be implemented by neural models. As will be seen, a more natural analogy of these capabilities is the object-oriented model. Thus, the consideration of higher level functioning will be left to one side for now and it is simply noted that connectionist expert systems provide understanding of \textit{primitive} neural processing.

\section*{Specification vs. Learning}

Expert systems have traditionally been constructed by specification. It is this element which gives insight on ‘equivalent’ neural models. However, much work has been spent in getting expert systems to learn.\(^\text{132,142-146}\) Some of these methods, arrived at in isolation from neural learning techniques, may lend insight on neural learning. For example, one skill refinement model is modeled on market forces within an economy. Each rule is viewed as a buyer and seller within a ‘knowledge market’. It buys proof of its \textit{conditions} from other rules, and sells its conclusion to other rules on the basis of how well it can be established. Thus, rules which find it ‘cheap’ to prove their hypotheses are used often by other rules since they are inexpensive. When asked to support a hypothesis a rule will be given a certain amount of credit (based on how vital the proof of the hypothesis is) with which to buy proof of its own conditions, which are themselves hypotheses belonging to other rules. The rule with the most credit will be evaluated first. In this way, the best method of establishing a hypothesis is determined. This seems to be some parallel of \textit{competitive learning}.\(^\text{147}\) It also suggests an analogue of Hebbian learning.\(^\text{22}\)
Other expert system refinement models appear to be applications of neural learning.\textsuperscript{131,137} Backpropagation may be used to determine the optimum conditional (threshold) and prior probabilities (weights) in a Bayesian-type model. An empirical comparison of connectionist and symbolic learning in general\textsuperscript{140} and comparisons of ID3 (a symbolic learning algorithm) and backpropagation\textsuperscript{148} have been performed.

In summary, the virtue of the expert system metaphor of neural computation is that it enables the opening up of the neural ‘black box’ so that what is occurring inside may be understood.\textsuperscript{103,104,149} Unfortunately, this understanding is limited to the primitive level of representation. Thus, we now turn to another related discipline, in which hierarchy is more explicit: distributed systems.

2. Distributed Systems

The Motivation for Distribution

Undoubtedly the most successful computation-engine ever to have been built is the von Neumann machine. It is \textit{universally powerful},\textsuperscript{30,150} flexible, ‘easily’ programmed and, above all, well understood. Compare this with neural nets which, though universally powerful,\textsuperscript{29} are certainly \textbf{not} well understood, thus \textbf{not} easily programmed and thus \textbf{not} flexible. Indeed, so many resources have been invested in the von Neumann machine that it is hard to imagine an alternative method of computation becoming predominant, at least in the near future. In spite of this fact, however, the von Neumann machine does have a major flaw: the \textit{von Neumann bottleneck}. The von Neumann model of computation is basically sequential which, though vast efforts have been made to widen and speedup this bottleneck, does fundamentally restrict the speed of computation. This has driven huge research into unifying the power of multiple processors and a good understanding of the problems involved in this has emerged.\textsuperscript{151,152}

One method of combining processors to solve a particular function is to \textit{farm out} subordinate functions – in a hierarchical manner – to \textit{slave} processors. This model leaves the \textit{master} processor with absolute control: lower level functions/processes do not exist until invoked by a higher level function. This technique is not of interest to us since it
seems unlikely that functions are farmed out to vacant areas of a neural network.

The Neural Metaphor

What is relevant by way of distributed theory, however, is the method by which a process/function may be divided into autonomous, cooperating processes which, by working together, compute the same function. This theory seems of direct relevance to neural models since it indicates how function may be distributed across neurons and across clusters of neurons and how communication may be distributed across synapses and groups of synapses. Thus, by ‘distributed system’ this thesis means: *a model of computation in which one or more functions are divided, in some manner, so that each subdivision is performed as an autonomous process executing on its own dedicated hardware and is interrelated in such a way as to implement those functions.*

A significant factor determining the way in which function is distributed is the target hardware. It is likely that a function will be distributed in a different way if each node is a von Neumann machine than from if each node is a neuron. The experimental results of this thesis (chapters Five and Six) show that function can be recursively distributed in such a way that each bottom level, atomic function is implemented by a neuron. If function is not distributed to this very primitive level then higher processing powers are needed at each node. It seems clear, therefore, that nature has chosen to use a completely distributed model of computation: it is not sensible to subdivide a neuron function. Intuitively, the reason for the choice of the neuron as primitive processor and of the synapse as primitive connection must be that these are the commonest ‘denominators’ of distributed function. It must be most ‘natural’ to realise higher level (human) functions in terms of interconnected neurons. Functions such as perception are presumably more naturally expressed in terms of interconnected neurons than in terms of interconnected logic gates, for example. The experimental results of this research indicate that the converse is also true: functions unnatural to humans, such as long division, are better represented in terms of logic gates than neurons. Neural nets thus will not take over from conventional machines in most tasks to which computers are currently put.

A key feature which distributed and neural systems have in common is their redundancy and fault-tolerance. Of the order of 100 neurons die out in the human brain each day, yet
it continues to function. Similarly, distributed systems are defined with built-in redundancy so that if, for example, one node fails, other nodes will detect this and take over its functions.

This thesis argues that some aspects of neural models can best be understood within the field of distributed systems. The methodology by which function is distributed implies a framework within which neural distributed processing may be understood: clusters of neurons perform describable functions; their computation is distributed across other appropriately-connected clusters of neurons; ultimately, all processing is distributed to primitive (neuron) level; and in the same way, high level messages between functions are distributed as lower level messages and, ultimately, as synaptic connections.

**Autonomy and Control**

A feature of distributed models is that each component has autonomy. Each component does not depend for its existence/creation on another component. It may receive requests to perform tasks but chooses whether or not to perform these at its own discretion. Within itself it has complete control, but outside itself it is powerless. All it can do if it requires some resource is to request it.

A distributed function cannot be understood in terms of its isolated components. What defines that function is the way in which its parts are interrelated by communication. Still less may a function be understood in terms of the components of its components. This is precisely the scenario encountered with neural nets. It is not possible to understand the function of a network by observation of the interconnection of individual neurons. In conventional distributed systems there exists hierarchy such that the role of each function can only be understood at a higher level. In the same way, it seems likely that neural distributed function will only be understood through hierarchy. Thus, the role of each cluster of neurons – that is, of each function – will only be determined through its interrelationship with other clusters.
Chapter Three - Related Disciplines

Communication

Autonomous processes that cooperate to perform some higher level function must, in order to achieve this, communicate in some way. This is performed in traditional distributed systems by message passing. Each module/function typically has a well-defined interface by means of which it can send and receive messages.

In most distributed systems, functions are not completely distributed to their most primitive, atomic form. Thus, in each function a degree of sequentiality is retained. Typically, each autonomous process is an imperative program executing on a von Neumann processor. In this case, it is necessary to have sophisticated message passing facilities, with queueing of incoming messages and blocking of process execution when waiting for certain messages. The metaphor that best fits neural models, however, is the completely distributed system, in which every component of every function (except the atomic function) is itself distributed. Thus, every function executes in parallel – and thus, incidentally, such communications complications as queueing do not arise.

The most significant result of the distribution of computation is increase in communication. Inevitably, the more that computation is distributed, the more intercommunication is required. Indeed, if it were possible to obtain unlimited, high speed communication between distributed processes, there would be few problems in distributing systems. The problem then would be how to distribute function, whereas at present the problem is how to distribute function whilst minimising and localising intercommunication. Too much distribution, using conventional hardware, replaces a processing (von Neumann) bottleneck with a communications bottleneck.

Neural nets are, it seems, completely distributed systems and thus, as discussed in Chapter Two, have massive intercommunication. It is hard to conceive a manner in which the amount of connectionism in biological neural nets will ever be replicated artificially in electronic hardware. This is due to the two-dimensional nature of electronic circuits as compared with the three dimensions within which the brain works, though electronics does have a third dimension of time. Optical implementations of artificial nets seem, apart from chemical implementations - which would effectively replicate biological nets, to be the only viable method of implementing massive connectionism artificially.
Incidentally, completely distributed functions communicating and processing optically would seem to be the fastest possible execution of that function that could ever exist. This is not to say that the function could not be represented in a different way (see section 4.5) so that it executed faster or slower. However, each particular expression of a function could not be executed faster than its completely distributed optical implementation.

3. Object-Oriented Modeling

Motivation

The development of programming languages and modeling methods was strongly influenced by the von Neumann machine. Modeling techniques developed so as to best utilise the sequential nature of this method of computation. Most high level languages are thus fundamentally sequential and the incorporation of parallel features within them seems unnatural. The framework presented in this thesis (see Chapter Four) is a more natural method of modeling entirely parallel systems and requires a different approach to ‘programming’ within it. This framework does allow sequential flow of control but is most suited to completely parallel/distributed representations. The use of sequential control is as unnatural to it as is the use of distributed control within conventional languages.

The motivation for object-oriented methods is to create supposedly more natural models of a world. (For an explanation of how all computing may be regarded as world-modeling, see Weizenbaum.) An object-oriented model contains a set of processes, each of which represents, and behaves in the same way as, an object in the world being modeled. The behavior of these objects is typically described in a conventional, sequential language.

Object-Oriented Concepts

The object-oriented methodology has developed its own terminology, much of which is still in a state of flux and not yet standardised. Here follow some of the basic concepts involved:
**Object** – an autonomous process representing and behaving like some entity in the world being modeled. For example, a process might represent a Porsche.

**Attribute** – a characteristic of an object. For example, size, colour, etc.

**Class** – the definition of a type of object. For example, *Car* or *Porsche*.

**Instance** – an object of a particular class. For example, Toby’s Porsche.

**Inheritance** – the means by which one object can be defined to be a special case of another, more general, class/type. For example, ‘Porsche’ would inherit the attributes of ‘Car’.

**Subclass** – a class that inherits the attributes of another class. ‘Porsche’ is a subclass of ‘Car’.

**Method** – an operation that can be performed by an object and which is typically invoked by the receipt of the appropriate message from another object.

**‘Pure’ Object-Oriented Modeling**

The framework for neural theory presented within this thesis is inspired largely by concepts in the field of object-oriented modeling. This field has moved away from sequential techniques in the modeling of a world. Attempts are made to create more natural models of a world by creating within the model a process to represent each object in that world. The definition of that object’s behavior is still, however, described in conventional – basically sequential – code. The framework presented in this thesis differs from this in that each object is defined exclusively in terms of other objects, much as function is subdivided in completely distributed systems (section 3.2). Thus, the model is completely parallel and constitutes a natural framework within which to describe neural structures. This is what is meant by *pure* object-oriented modeling.

The object-oriented paradigm seems a good metaphor for neural computation owing to its natural representation of a world. It is conceivable that the brain models the world in the same way. Objects are ‘understood’ in terms of more primitive objects in the same way that clusters of neurons are defined in terms of lower level clusters. This argument is
advanced in the next chapter (section 4.5).

4. Non-Neural Connectionism

Besides the three major metaphors for neural computation already discussed, there are various other non-neural network formalisms, each of which may offer its own insights on neural modeling. The relationship between \textit{semantic nets} and neural nets has been explored.\textsuperscript{155,156} \textit{Fuzzy petri nets} are networks similar to expert networks\textsuperscript{137} and perform primitive knowledge-based processing.\textsuperscript{157} \textit{Configurable hardware} is a class of target architectures for completely distributed implementations.\textsuperscript{158} Neural models could perhaps be included in this class even though these architectures traditionally contain logic gate functions at primitive nodes.

5. Summary

In this chapter various disciplines related to neural nets have been discussed. Expert systems, distributed systems and object-oriented paradigms are all disciplines in their own right and, to some extent, seem to be variations on a common theme. Each discipline is, as has been argued, intimately related to neural modeling and each offers unique insights on neural theory.

From connectionist expert systems we have gained a potential understanding of primitive neural functioning — a well-understood metaphor for cell function, synapses and network topology. From distributed systems we have gleaned a framework containing \textit{hierarchy} for the distribution of computation to a target architecture containing primitive processors such as the neuron and primitive connections such as the synapse. From object-oriented modeling we have deduced a potential framework for neural representations that is, above all, a natural method of modeling the world. The unification of these three 'theories' forms the foundation for the next chapter, which proposes a framework for neural theory.
Chapter Four

A Framework for Neural Theory

1. Introduction

The key concept underlying this thesis is hierarchy. This concept is all but entirely absent from current analysis and construction of neural nets. At present, attention is focused almost exclusively at the primitive level. When faced with the questions: What does a network mean? and How does a network compute its function? current ‘theory’ is powerless to respond. It simply is not possible to understand a complex network in terms of individual primitive neurons and synapses. This problem is analogous to trying to deduce the function of a one million-transistor digital integrated circuit solely from a netlist of transistors. The physical layout is a clue to its various components (as seems likely to be the case in biological neural nets) and an experienced chip-designer may be able to deduce some understanding of its function from this. It may be possible to group transistors into D-type flip-flops, group these into shift registers, and so on. However, the very principle underlying this process is hierarchy. Though hierarchy is not readily apparent in a flat netlist of transistors (or, for that matter, a netlist of neurons) it is present and actually underpins the correct implementation and testing of such a system. The problem of reverse-engineering a neural implementation to a hierarchical representation of its function is yet more complex than for an integrated circuit: neural nets don’t process discrete values; they interconnect massively; they have never been constructed from hierarchical specifications so it is not known what cues to look for in discerning which structures implement which functions.

To take another example, one could try reading this thesis by selecting characters at random from its pages. Hierarchy underpins the framework by means of which we comprehend text. Characters, which have meaning at a low level, are related to form words, which have meaning at a higher level. Words are related to form phrases, phrases to form sentences, sentences to form paragraphs, and so on through subsections and sections, to chapters and thesis. It is not possible to either write or understand this thesis
without a concept of hierarchy, however subconscious that may be.

Yet another example of how essential hierarchy is to our understanding is in the field of physics. It is not sensible to try to understand the replication of DNA in terms of subatomic particles. What is needed is intermediate levels of representation which bridge this gap. Each of these levels has its own ‘theory’ describing how it relates to lower levels (a capability required for the neural framework). Atoms are formed from subatomic particles, base molecules from atoms, proteins and polymer chains from base molecules. Levels of representation are essential to our understanding of this.

An example which is closer to the neural problem, in that it too is concerned with computation, is hierarchy within software systems. A complex program cannot be understood in terms of its compiled binary machine code. Its function becomes only vaguely-discernable if the binary is transformed to mnemonic machine codes. These in turn need to be abstracted to programming constructs such as if...then...else, then to functions, higher level functions, and so on up to module, subsystem and system levels.

It is helpful to consider these examples of hierarchy in order to enable us to appreciate its virtues. A final example, especially relevant to this thesis (see Chapter Six), is in image processing. It is not possible to read a numberplate, or recognise a face, or match two fingerprints, or any other non-trivial image processing task, simply by consideration of a two-dimensional array of intensity values. All the necessary information to perform any of these tasks may be present in this array but it cannot be directly transformed into the representation we are seeking. Instead, it must pass through a hierarchy of levels of representation: typically, pixels must be transformed to edges, edges to boundaries, boundaries to segments, segments to measurements and measurements to classifications. The theory by which these transformations are made is at the very heart of image processing, as is the problem of finding the best sequence of representations through which to pass in arriving at the goal.

Each of these examples illustrates the crucial role of hierarchy in comprehending a complex system. Each example contains relatively few levels of hierarchy – though perhaps intermediate levels could be inserted which we have not mentioned. Where these examples differ from the framework of hierarchy advocated in this thesis is in their lack
Chapter Four - The Framework

of homogeneity. A different set of rules exists at each level (barring silicon compilation and software definition) for mapping one level of representation to another. The framework of hierarchy for neural nets, however, is not restricted to a certain number of levels and is homogeneous throughout the levels.

The only level that is common to all models constructed under this framework is the primitive level, that which contains neurons and synapses. Though the framework is applicable to all neural models, the way in which these models form higher levels of representation is not constrained by the framework and is instead determined by the designer or (perhaps) the learning process. The hierarchy presented in this thesis is not the same as modularity of networks, which has been described in previous work as hierarchy. The scenario where several subnets or modules produce results which form input to a 'higher level' module is not taken to be true hierarchy. Representations are only analysed above the primitive level in a very restricted sense. There is no hierarchy of data.

It has been argued that a multilevel representation, in addition to a method of interrelating levels, is essential to the understanding of neural systems. Thus, the next section (4.2) presents a framework of hierarchy for understanding neural systems at arbitrary levels of abstraction. The subsequent section (4.3) presents a method for relating levels by means of state-sequence analysis.\(^\text{101}\) Section 4.4 considers whether it is plausible that specifications made within this framework can be biologically encoded within the genetic code. In addition, the implications of the presence of hierarchy for learning are explored. Finally, the importance of adopting a good representation at each level is discussed.

The Concept of Levels

The purpose here is to represent functions and data, and to perform transformations between these representations. A framework is required which enables the description of distributed functions and data at arbitrary levels of abstraction and which enables the interrelation of those levels.\(^\text{159}\) As discussed in the previous section, the idea of levels is crucial to this framework. Perhaps the best-known analysis of levels is that given by Marr.\(^\text{160}\) He identifies three main levels of representation, at which understanding is essential:
Computational Theory - what is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out.

Representation and Algorithm - how can this computational theory be implemented? In particular, what is the representation for input and output, and what is the algorithm for the transformation?

Hardware Implementation - How can the representation and algorithm be realised physically?

Marr cites these as three levels at which any machine carrying out an information processing task must be understood. He also sketches the main levels of representation involved in image processing; these levels are representations of data – not function, as above.

I prefer to think of the levels of representation of function as in Figure 4.1. The pyramid illustrates not the increased complexity as a function is implemented but the concept that a function at any particular level of abstraction can be implemented in a usually large, and sometimes enormous, number of ways. Similarly, there exists a common abstraction for many different implementations of a function.  

Abstraction and Implementation

Abstraction contains the idea of capturing the essence of something described at a greater level of detail. It involves saying less about how something is done and more about what is done. Abstraction contains the concept of summarising (not modifying) some description from a more to a less concrete form.

Implementation is the inverse operation to abstraction. It involves putting a description of a function into effect. It involves making a function more concrete, saying the same thing but in more detail, transforming what a function is into how it should be performed.

There is considerable debate over what is the best view of levels. The concept of levels now developed is sufficient to describe the framework for neural theory.
Levels for Neural Representations

Within the framework the neuron/synapse level will be defined as the ‘primitive’/realisation level, the base of the pyramid in Figure 4.1. It is conceivable that there are yet more primitive implementations of this level but, for the purposes of understanding neural systems, neurons and synapses will be treated as primitive representations. The contention of this thesis is that a neural network is a realisation of functioning that can be meaningfully described and understood at higher levels of abstraction. As already discussed in the introduction to this chapter, that function cannot be understood at the primitive/realisation level alone. The framework must enable the abstraction and implementation of functioning in a completely distributed manner. This is
achieved by the use of three basic concepts (see also figure 4.2):

(i) a function - which transforms inputs to outputs in some way.

(ii) a connection - which provides a means of integrating functions.

(iii) an interface - by means of which a function communicates with other functions - the 'outside world'.

![Figure 4.2 Concepts of the Framework: Definition of a Function](image-url)
2. The Framework

Functions

Use of the term function can be misleading since our functions are not restricted to returning a single, or even composite, value. Instead, they are allowed to take many inputs and produce many outputs, simultaneously. Our use of the term is more closely allied to the idea of an object, as used in object-oriented models of computation (see section 3.3). The difference here is that objects in these models are typically defined in terms of (sequential) imperative code, and thus cannot naturally respond to simultaneous inputs with simultaneous outputs. In this sense, our use of the concept function is closer to the way in which a distributed system is defined. Here, a distributed system (function) is defined in a completely distributed manner such that the distributed system (function) consists of the appropriate interconnection of lower level distributed systems (functions). This analogy is a better parallel of the inherent distribution in neural systems, though the valuable concepts in object-oriented modelling are not explicit. For a discussion of these issues see sections 3.2 and 3.3.

Figure 4.3 Definition of if...then...else function

Broadly speaking, a function at one level of abstraction is implemented at a lower level (and in a multiplicity of ways) by interrelating lower level functions in such a way that together they produce the desired behavior. (See figure 4.3 for an example of the way in
which an if...then...else function may be implemented in terms of primitive functions and connections.) This interrelation is performed by message passing between functions (see section 3.3). Where messages come from and go to is defined by interconnecting functions to form the appropriate topology. This style of definition is more declarative than most classical techniques (e.g. imperative algorithms) for describing functions.

Connections

Just as levels of abstraction exist in representation of function, so connections represent levels of abstraction in the representation of data. If the synapses transmitting visual information from the eye to the part of brain that processes visual information were to take random paths through the rest of the brain it would be very difficult indeed to deduce what was going on. In practice, however, these nerves are tightly grouped into a 'higher level' connection, the optic nerve. It makes sense to understand the role these synapses play by grouping them together: the grouping transmits an ‘image’ (actually a combination of intensity values and primitive objects such as edges) to another module within the brain.

As described in Chapter Six (section 6.6) a connection of type ‘image’ may be defined in terms of more primitive types of connection. For example, an image may be defined as a row of columns; or as a column of rows; or as a row of columns of blocks; and so on. A row may be defined in terms of pixels, which may themselves be defined in terms of primitive synapses. (See Figure 4.4.)

This hierarchy in connections is necessary to facilitate high level message passing. Though at implementation level an image is sent along, say, a million primitive paths, at the conceptual level an image is sent, period. This abstraction of data must go hand in hand with the abstraction of function.

Interfaces

Each function, at each level of abstraction, has a typed interface. This consists of one or more ports, of particular connection-types, at which input is received and from which output is sent. It is by means of this interface that each function communicates with the
outside world. Thus, when a function is defined – by interconnecting lower level functions – these interconnections are made to/from individual ports on those functions, not directly to components of those functions. Thus, each function has no control over its role in defining higher level functions; all it ‘knows about’ and can do is to perform its own function, transforming inputs received at its interface to outputs which it transmits via its interface. In this way, as in distributed and object-oriented models (see sections 3.2 and 3.3), functions are autonomous. This use of typed interfaces allows the definition of a function to be restricted to one level at a time.

Each non-primitive type of connection is defined in terms of lower level types. Thus, each port in the interface of function X itself contains ports – of lower level types. Connections external to X must be of the same type as the port on X to which they connect. Internal connections, however, may connect to one of the port’s lower level ports which represent the types in terms of which the port is defined. Using this latter method of connection enables the function to decompose a high level connection into its constituent types. Thus, for a function to perform edge detection on input received as type
image it must first decompose this image type to pixel level. Composition of higher level connection types is achieved in the same manner.

Instances

If it is necessary to define several functions in terms of one common lower level function, an instance of that function is required. For example, functions to perform object detection and object classification might both be defined in terms of a function which detects edges at a particular point in an image. Instead of creating two instances of this edge-detection function it makes sense to use a common instance, in terms of which both higher level functions are defined. This is analogous to the concepts of class and instance in object-oriented modelling (see section 3.3).

This capability permits compact implementation of higher level functions; two functions are not required to do the same thing. Most neurons, or clusters of neurons, will typically be components of more than one higher level function. Thus, the implementations of multiple high level functions – which ultimately consists of primitive interconnections between primitive processors – will normally be closely intertwined. Several high level functions will typically be implemented in terms of common neurons, or common clusters of neurons, each function interconnecting these in different ways. In the same
way, instances of connections may be created so that disparate functions may communicate via the same communication path. This, of course, may not make sense without the use of multiplexing though such connection instances may be a feature of biological systems.

Summary

A framework of hierarchy has been described within which representations may be transformed between levels of abstraction. Neural Compilation, the process by which a hierarchical specification of a neural system is implemented, is facilitated by ANNECS, a software tool described in Chapter Five. What is significant about this framework is that it enables the understanding of neural systems at arbitrary levels of abstraction. As has been discussed, this ability is essential for the understanding of the operation of any non-trivial system and should aid analysis of neural systems by raising representations above the primitive level. The next section considers how levels may be formally related to each other and how transformations may be made between one level of representation and another.

3. A State-Oriented Analysis of the Framework

A function can be represented in many different ways. It could be described in a high level language such as Prolog or Pascal; it could be represented in machine code for a 68000 microprocessor; it might be described in terms of logic gates, or a state transition table; it might be represented as a Turing Machine; it might be realised by a neural network. How can these representations be compared? When are two implementations of a function computationally equivalent? When is a function a common abstraction of other functions?

Recent work\textsuperscript{10} has presented a method for characterising functions so that their relationship to each other can be analysed. This theory is also of use in analysing the neural framework of hierarchy. Thus, a description of the basic principles of Foster's approach is made and this approach is then applied to the relationship between levels of abstraction of neural function.
Foster’s State-Sequence Characterisation of Function

A function (i.e. algorithm, neural network, digital integrated circuit, etc) is characterised by a set of state-sequences. A state-sequence is, obviously, a sequence of states through which the function passes. A state consists of all the variables contained in the function — which may include, for example, instructions or network topology, as well as data. This is best illustrated by Foster’s example of a Pascal-style representation of an exclusive-or function:

```
program xor;
    var x, y, z: integer;
begin
    readln(x);
    readln(y);
    if (x=y) then
        z := 0;
    else
        z := 1;
    writeln(z);
end.
```

This function will start off in the following state:

- x: U
- y: U
- z: U

'x', 'y', 'z' and 'next instruction' are labels (or variable names) to which are attached states. Initially, x, y and z are all undefined: U. As the function executes, these states will change in the following sequence:
As Foster shows, a neural realisation of the exclusive-or function can be characterised using the same method. Here, however, variables are continuous-valued, not discrete, and must be represented to some arbitrary degree of precision. The labels (0, 1, 2, 3, 4) correspond to the neurons in Figure 4.6:
These examples convey the method of the approach and illustrate how classical and connectionist implementations of a function can be compared. A function is characterised by a set of state-sequences because it can only be exhaustively described in
terms of input and output by producing a state-sequence for each input/output combination. Obviously, this method of characterisation explodes with increase in complexity of function or data but this is not of concern. What is required is not a practical but a theoretical means of characterising an algorithm and interrelating it with its abstraction and implementations.

**Hierarchical State-Sequences**

Implicit in the previous examples was the concept of *detail*, which may be viewed as a form of hierarchy. The Pascal representation would typically be compiled and assembled to a machine code representation for execution on a von Neumann architecture. That machine code level of representation will contain several other variables used in computing intermediate results. Thus, if we were to take our state-sequence analysis to machine code level, intermediate sequences would typically be required to transform between each of the major states listed in the example. For the neural implementation, however, it doesn’t make sense to insert intermediate state-sequences because the neural realisation is ‘primitive’. (As previously described, it is possible to subdivide neuron function but, for the purpose of this analysis, the neuron/synapse level is taken as the primitive level.)

The process of ‘filling in’ more detailed state-sequences corresponds to *implementation*, whereas the process of removing intermediate state-sequences corresponds to *abstraction*. For example, the state-sequence description of the neural exclusive-or could be abstracted to omit the states of neurons two and three. Effectively, the function would then be described entirely in terms of its input and output states through time.

**A State-Oriented Description of the Framework**

A method that interrelates levels within the framework is required. This will determine how a function is implemented and how it is abstracted. A hierarchy of state-sequences, in addition to some simple constraints, enables the study of this relationship.

For illustrative purposes, take the hypothetical function – expressed within the framework – as shown in Figure 4.8. All that is shown is the interface to the function. By
defining state-sequences for inputs supplied to, and outputs received from, this interface, the function can be comprehensively characterised. This is an implementation-independent method of specifying what the function does. The class of correct implementations of this function is surprisingly large. Informally, any configuration of lower level functions which obeys certain simple constraints is a valid implementation. (See Appendix A for a formal treatment of this.) To describe these
Each lower level function used to implement the hypothetical function has its own state-sequence. It is not of importance, at this stage, how these functions are themselves implemented. What is required is that they satisfy constraints imposed on them by the topology in which they have been interrelated so as to implement the hypothetical function. The constraint which will ensure that the topology is a correct implementation...
of the function is as follows: where interface ports in two functions/interfaces are connected, the state on each port must be the same (or undefined) at each point in time. Thus, for the implementation of the hypothetical function in Figure 4.9, the state-sequences for the functions in terms of which it is implemented are as shown in Figure 4.10 ($a_{it}$ are algebraic variables denoting the state on a port at time $t$).

**State-Oriented Abstraction and Implementation**

If all functions within a hierarchical specification of a neural system satisfy these constraints then, by induction on state-sequences, the neural realisation satisfies the top-level function specification. The base case here, is the state-sequence of a neuron, which approximates some mathematical model. Note that, for the purpose of this analysis, the synaptic multiplication is incorporated in the state-sequence of a neuron. ANNECS is a software tool which performs this implementation of a high level function as a neural network (see Chapter Five). When a function is successively implemented in terms of its
components such that each stage in the implementation satisfies the constraints outlined above, the resultant realisation must be a valid implementation of the top level specification.

<table>
<thead>
<tr>
<th>Port</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>n</th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.29</td>
<td>0.30</td>
<td>0.29</td>
<td>0.30</td>
<td>0.92</td>
<td>0.25</td>
<td>0.50</td>
<td></td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>1</td>
<td>0.65</td>
<td>0.10</td>
<td>0.00</td>
<td>0.55</td>
<td>0.29</td>
<td>0.74</td>
<td>0.51</td>
<td></td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>0.78</td>
<td>0.99</td>
<td>0.95</td>
<td>0.81</td>
<td>0.36</td>
<td>0.86</td>
<td>0.73</td>
<td></td>
<td>0.79</td>
<td>0.28</td>
</tr>
<tr>
<td>3</td>
<td>0.49</td>
<td>0.96</td>
<td>0.68</td>
<td>0.72</td>
<td>0.43</td>
<td>0.91</td>
<td>0.44</td>
<td></td>
<td>0.01</td>
<td>0.52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Port</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.29</td>
<td>0.30</td>
<td>0.29</td>
<td>0.30</td>
<td>0.92</td>
<td>0.25</td>
<td>0.50</td>
<td></td>
<td>0.99</td>
</tr>
<tr>
<td>1</td>
<td>0.65</td>
<td>0.10</td>
<td>0.00</td>
<td>0.55</td>
<td>0.29</td>
<td>0.74</td>
<td>0.51</td>
<td></td>
<td>0.02</td>
</tr>
<tr>
<td>2</td>
<td>0.78</td>
<td>0.99</td>
<td>0.95</td>
<td>0.81</td>
<td>0.36</td>
<td>0.86</td>
<td>0.73</td>
<td></td>
<td>0.79</td>
</tr>
<tr>
<td>3</td>
<td>0.49</td>
<td>0.96</td>
<td>0.68</td>
<td>0.72</td>
<td>0.43</td>
<td>0.91</td>
<td>0.44</td>
<td></td>
<td>0.01</td>
</tr>
</tbody>
</table>

A

<table>
<thead>
<tr>
<th>Port</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>n</th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a_00</td>
<td>a_01</td>
<td>a_02</td>
<td>a_03</td>
<td>a_04</td>
<td>a_05</td>
<td>a_06</td>
<td></td>
<td>a_0n</td>
<td>a_{0n+1}</td>
</tr>
<tr>
<td>1</td>
<td>a_10</td>
<td>a_11</td>
<td>a_12</td>
<td>a_13</td>
<td>a_14</td>
<td>a_15</td>
<td>a_16</td>
<td></td>
<td>a_{1n}</td>
<td>a_{1n+1}</td>
</tr>
<tr>
<td>2</td>
<td>a_20</td>
<td>a_21</td>
<td>a_22</td>
<td>a_23</td>
<td>a_24</td>
<td>a_25</td>
<td>a_26</td>
<td></td>
<td>a_{2n}</td>
<td>a_{2n+1}</td>
</tr>
<tr>
<td>3</td>
<td>a_30</td>
<td>a_31</td>
<td>a_32</td>
<td>a_33</td>
<td>a_34</td>
<td>a_35</td>
<td>a_36</td>
<td></td>
<td>a_{3n}</td>
<td>a_{3n+1}</td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>Port</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a_00</td>
<td>a_01</td>
<td>a_02</td>
<td>a_03</td>
<td>a_04</td>
<td>a_05</td>
<td>a_06</td>
<td></td>
<td>a_0n</td>
</tr>
<tr>
<td>1</td>
<td>a_10</td>
<td>a_11</td>
<td>a_12</td>
<td>a_13</td>
<td>a_14</td>
<td>a_15</td>
<td>a_16</td>
<td></td>
<td>a_{1n}</td>
</tr>
<tr>
<td>2</td>
<td>a_20</td>
<td>a_21</td>
<td>a_22</td>
<td>a_23</td>
<td>a_24</td>
<td>a_25</td>
<td>a_26</td>
<td></td>
<td>a_{2n}</td>
</tr>
<tr>
<td>3</td>
<td>a_30</td>
<td>a_31</td>
<td>a_32</td>
<td>a_33</td>
<td>a_34</td>
<td>a_35</td>
<td>a_36</td>
<td></td>
<td>a_{3n}</td>
</tr>
</tbody>
</table>

C

<table>
<thead>
<tr>
<th>Port</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>n</th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.29</td>
<td>0.30</td>
<td>0.29</td>
<td>0.30</td>
<td>0.92</td>
<td>0.25</td>
<td>0.50</td>
<td></td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>1</td>
<td>0.65</td>
<td>0.10</td>
<td>0.00</td>
<td>0.55</td>
<td>0.29</td>
<td>0.74</td>
<td>0.51</td>
<td></td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>0.78</td>
<td>0.99</td>
<td>0.95</td>
<td>0.81</td>
<td>0.36</td>
<td>0.86</td>
<td>0.73</td>
<td></td>
<td>0.79</td>
<td>0.28</td>
</tr>
<tr>
<td>3</td>
<td>0.49</td>
<td>0.96</td>
<td>0.68</td>
<td>0.72</td>
<td>0.43</td>
<td>0.91</td>
<td>0.44</td>
<td></td>
<td>0.01</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Figure 4.10 State-sequence characterisation of functions which form a valid implementation of the hypothetical function.
The representation of a function in terms of its defining functions’ state-sequences instead of its own state-sequence corresponds to implementation as described by Foster.\textsuperscript{101} The reverse process corresponds to Foster’s definition of abstraction. This hierarchical state-oriented method of analysis offers a means of relating levels in a formal way.

**4. Biological Considerations**

This section considers the ease with which this framework may be embodied by biological processes. This involves exploring how a hierarchical specification may be represented in a genetic code and how such a representation may be interpreted during growth. It is also interesting to consider what role hierarchy might play during learning. It should be emphasised that this section (alone) is purely speculative and not central to the thesis. It is included for the sake of interest alone.

**Genetic Encoding of Hierarchical Specifications**

It is beyond the scope of this thesis to consider whether what is known about DNA and cell replication is sufficient to say whether hierarchical specifications of neural nets may be encoded and interpreted during development. What we can say is what level of biological functionality is required in order to achieve these objectives.

A complete specification of a neural system may be viewed as a netlist of netlists (see section 5.3). This structure may be genetically encoded and thus guide brain-generation, given the following capabilities:

1. It must be possible to point to a certain point in the DNA chain.

2. It must be possible to move the pointer to another point in the chain, dependent on what it previously pointed to.

3. It must be possible to give a neuron a label (in order to specify connections).

A method for genetically encoding a specification is illustrated in Figure 4.11.

Brain development, under this scenario, basically consists of moving pointers along the chain. When a cell replicates, one cell (a) contains the pointer moved along to the next
unit in the chain; the other cell (b) contains the pointer moved to a new cluster definition. Thus, generation will propagate down through the hierarchy to the primitive level. As cell (a) moves its pointer through all the clusters/functions which make up a certain level of definition, it will eventually come to some ‘stop’ code, whereupon the cell should die out. What that cell has represented is a particular implementation of a function at a non-primitive level in the hierarchy. When it has spawned the generation of the functions in terms of which it is defined, it has no role left and so dies out. This may explain why many neurons die out during biological development in a way that appears to be programmed in (see section 2.1). Presumably, neurons could be created to form connections to neighbouring neurons which have a common label. These labels could be determined by the position of the pointer within the chain.

**Hierarchical Learning**

It seems likely that levels of hierarchy in connections exist in biological systems. The optic nerve is an obvious example. Since these hierarchies are probably represented by the spatial alignment of connections it seems likely that some biological effects may cause ‘high level learning’. Artificial learning algorithms that ‘work’ (section 2.2) relate modification of one synapse to modifications of other synapses by, for example, backpropagation of errors. This relationship between modifications is necessary in order to converge on a solution. However, the method of relating synapse modifications is almost certainly non-biological. Abstracting neural functions to higher levels of
representation may supply an alternative method for relating weight-changes. This would come about as a result of the hierarchy inherent in connection patterns. There may be biological evidence that the strengths of a group of synapses between two clusters of neurons increase and decrease largely in unison (see also section 7.3). Again, it is stressed that these thoughts are entirely speculatory.

5. Discussion on Representations

What is a Representation?

Marr describes a representation as "a formal system for making explicit certain entities or types of information, together with a specification of how the system does this." For example, a model of the solar system is a representation of it. The Arabic, Roman and binary numeral systems are representations of numeric values. Arabic representations model a number by a string of symbols drawn from the set \{0,1,2,3,4,5,6,7,8,9\}. A representation is formed by decomposing the number into a sum of multiples of powers of 10 and concatenating these values into a string with higher powers to the left and lower
powers to the right. For example: \(37 = 3 \times 10^1 + 7 \times 10^0\). Binary representations, on the other hand, decompose the number into a sum of multiples of powers of two. Thus, 37 in Arabic representation becomes \(100101\) in binary. Roman representations use rules not directly based on powers of any number and thus the representation is not suited to performing arithmetic.

The way in which visual information can be transformed from one representation to another has already been discussed (section 4.1). Pixel intensity values can be transformed to edges, edges to boundaries, boundaries to segments, segments to measurements and measurements to high level representations of objects. There are many other ways in which representations of visual information can be transformed. For example, pixel intensity values can be represented as a histogram. A scan line (a horizontal or vertical line of pixels in the image) can be represented by a graph which plots intensity against pixel index. An image can be inverted by taking: \(\text{pixel}[i][j] = \text{max-intensity} - \text{pixel}[i][j]\). An image can be transformed into a representation within the frequency domain by performing a Fourier Transform.

Each of these examples serves to illustrate what is meant by a representation. Computation itself, in its most basic definition, may be viewed as the transformation of representations, by means of some method of combination, to other forms of representation. Thus, the number 37 can be represented as \(30+7\), or \(29+8\), or \(28+9\)... or \(3\times(5\times(5-3))+7\)... An image can be represented, at one extreme level, as an array of intensity values or, at another, as a description of the objects pictured, such as the name of the person whose face is visible. We are back to levels again.

This discussion is of relevance to us for two reasons. First, it is worth again stating that the framework for neural theory is not a particular representation. It is a formalism within which representations are made. Thus, this thesis is not primarily about representations. It does not attempt to answer what is or is not a ‘good’ representation. In particular, it does not concern itself with how the brain represents the world. A suggestion of this is given later in this section, but a mere suggestion it remains. In fact, there is no such thing as a fundamentally good representation. The question should rather be: what is a representation good for? Binary representation of numbers is good for determining whether a number is a power of two; it is bad for deciding whether it is a power of ten.
The binary representation of data suited to von Neumann computation is almost certainly not how numeric values are represented in the brain. The instruction-oriented representation of functions within von Neumann machines is almost certainly not the way in which functions are represented in the brain.

Just as there are usually many possible implementations of a function, so there are many possible ways of representing a function neurally. This thesis does not say which way is best but it does provide a method for implementing and comparing those representations.

**What representations are natural to neural realisation?**

In Chapter Three the virtues of a relatively recent methodology for creating natural models of a world were discussed. Object-oriented modelling seems to be the most 'natural' method of modeling a world that exists. Each entity in the world is realised by an entity in the model; its interactions with other entities are realised - in a 'natural' way - by passing messages. What is interesting about this is that these representations are elegantly implemented within the neural framework. Each object in a definition is autonomous; each cluster of neurons in a net is autonomous. Each object in a model is thought of as a continuously executing process; each cluster of neurons that implements an object is continuously existant and active. Objects communicate by message passing; clusters of neurons communicate by passing messages along multiple synapses. Objects are specified in terms of other objects and ultimately in terms of one or more primitive objects; each cluster of neurons may be perceived as interconnections of other clusters of neurons and ultimately as interconnections of primitive neurons.

As noted in Chapter Two (section 2.2) a von Neumann machine may be implemented in terms of neurons. This is a very unnatural use of neural hardware since it uses a parallel method of computation in a sequential manner, just as we do, slowly, when performing mental arithmetic. As the adoption of the Arabic representation of numbers was essential for the development of mathematics so the use of representations which are natural to neural implementation is essential for the advancement of neural computation. The limit to which this thesis can go is to say that the framework presented is suited to the expression of distributed, object-oriented-style implementations, and that neural systems are supremely distributed and are concerned with forming natural (i.e. perhaps object-
oriented) models of the world. Thus, it seems likely that representations that are easy to describe within the framework will make good use of neural hardware.

6. Summary

A framework whose basis is hierarchy has been presented which enables neural systems to be understood at arbitrary levels of abstraction. The concept of levels is applicable to both function and data. The formal relationship between levels in this hierarchy has been analysed and simple constraints for the correct implementation of a function have been identified. This framework, whilst facilitating the creation of distributed representations, does not identify what constitutes a ‘good’ representation.

The next chapter describes an embodiment of the framework in the form of a software tool. The following chapter then describes an application of the framework to a real-world problem.
Chapter Five

ANNECS : A Neural NEtwork Compiler and Simulator

1. Introduction

ANNECS is a software tool which embodies the methodology for constructing neural nets proposed in Chapter Four.\textsuperscript{161,162} It enables the formation – compilation – of a neural network from a hierarchical specification. It then enables learning of that net – simulation – by applying one of a number of learning algorithms. During compilation the high level information contained in the hierarchy of the specification is retained such that learning that occurs can be understood. The software that performs a function closest to ANNECS’ is probably the CONIC toolkit for constructing distributed systems.\textsuperscript{163} ANNECS however, whilst sharing some principles of operation with CONIC, is oriented exclusively towards neural implementations.

Basically, ANNECS enables the user to define functions in terms of appropriately interconnected lower level functions. The only primitive function is the neuron and the only primitive connection is the synapse, though the model upon which each of these is based can be selected by the user. Thus, all functions are defined, ultimately, in terms of neurons interconnected by synapses. The compilation component of ANNECS performs this translation between a high level, hierarchical specification and its functionally equivalent neural implementation.

The development of this software was undertaken to provide experimental support for the framework for neural theory advanced in this thesis. Thus, ANNECS integrates genetic and empirical methods of construction, the compilation and simulation components, respectively. The key element which enables this to be carried out in a meaningful way is the presence of hierarchy. The experimental results obtained from this work – the development of ANNECS and its application to numberplate recognition – endorse the methodology proposed by this thesis. Within a framework of the nature described in Chapter Four, the functions computed by neural systems can be comprehended at
ANNECS consists of circa 5000 lines of ‘C’ code and makes extensive use of the SunView graphics software. It was developed over a period of about eighteen months and forms approximately half the experimental work of this research. ANNECS is simple in design and easy to use. It is largely menu-driven and thus performs most functions by use of the mouse. The only typing required of the user is in order to name functions, connections and interfaces, and to supply initial weight and threshold values.

2. Features of ANNECS

This section reviews the significant features of ANNECS and describes why their implementation was necessary in order to substantiate this thesis.

Visualisation

Within the framework described in Chapter Four, description of a neural architecture consists of a hierarchy of netlists. Written in language-form, a netlist can be fairly meaningless. Text is inherently sequential in the way in which it lies on the page, even if what it expresses is something fundamentally parallel. A netlist is above all a structure, and structures are perhaps best conceived visually. Thus, an essential requirement of ANNECS is that it visualises specifications. Each component of a function is a real entity, continually existant in the target neural implementation, and thus it makes sense to have it represented by a real object at a particular place on the screen. This is not to say that the same specifications could not be described linguistically, but that the style of specification lends itself to, and is best understood by means of, visual representation. ANNECS uses visualisation for the same reason that schematic capture tools use it.

Each type of function and each type of interface is represented by a user-defined icon. This icon is used to capture function visually (see figure 5.1).

Similarly, the type of each connection is represented by a uniquely-patterned line. Cubic splines are used to generate curved interconnections between primitive functions and interfaces, in order to make structures look more biological! Arrows on connections indicate the direction of the message path: connections are unidirectional.
Libraries

As is customary in interactive editors (as opposed to entirely language-based methods of specification) it is necessary to categorise function/connection types hierarchically so as to be able to access them efficiently. ANNECS achieves this by the use of hierarchical libraries, one hierarchy for functions and another for connections. To carry out maintenance on these libraries, a number of housekeeping functions are provided.

**Edit Object** - used to load an object definition. An object is a function or a connection type.

**Create Object** - used to create a new function or connection type.

**Create Library** - creates a new library as a member of another library.

**Copy Item** - will copy a member of a library, or a library and all its dependents, to another library.

**Move Item** - the same as *copy* except that the source is deleted after copying.

**Rename Item** - self-explanatory.
Delete Item - if item is a library, deletes all dependents as well, after prompting for confirmation.

Store Library - ‘snapshots’ of a library can be stored under user control and reverted to later in the development process, if so desired.

Load Library - to reinstate a previously stored ‘snapshot’ of a library.

Macro Expansion

Associated with each function are a number of user-defined macros. By use of these ANNECS will generate a textual description of a function. This description is derived from the macros of the lower level functions in terms of which that function is defined. At the primitive level, ANNECS generates a list of Horn Clause, PROLOG-style predicates with ‘conditional probabilities’ (weights) and ‘prior probabilities’ (thresholds) incorporated.

The purpose of this automatic generation of textual descriptions is:

(a) to show the similarity between rule bases and neural nets at the primitive level, and

(b) to show that hierarchical linguistic descriptions of neural architectures can be made.

Macros are expanded in the order in which they are defined. The expansion of one macro can invoke the expansion of other functions to which it is connected. Thus, ‘sequential’ code for functions can be generated, though this ‘sequential’ function is in effect pipelined. For an example, see figure 5.3.

There is a reason for there being a number of macros associated with each function. The user, when generating text, can specify that the nth macro be used for each function, so as to enable generation in distinct languages. In addition to this, there is another parameter for text generation which is depth of expansion. If the depth is greater than one, the text for the components (in terms of which the top level function is defined) is generated recursively to the specified depth, so as to generate modular ‘source code’.
Chapter Five – Applying the Framework

Specification

The specification of a neural system is made by the hierarchical description of functions. The specification of each of these neural functions is made up of a netlist of lower level functions and interfaces. Thus, the specification process consists of:

![Diagram showing example hierarchical menu layout: Functions]
Function 2_to_4_decoder:
{
  if (Bit 0) {
    if (Bit 1) {
      Bit A;
    } else {
      Bit B;
    }
  } else {
    if (Bit 1) {
      Bit C;
    } else {
      Bit D;
    }
  }
}

Figure 5.3
Generation of textual description for function containing nested if...then...else

macro for if...then...else function:

\[
\text{if (@1) } \{\text{\indent }\text{$2$\indent \}\text{ else } \{\text{\indent }\text{$3$\indent \}\}\}
\]

\n start a new line
\t indent subsequent new lines
\u cancel last indentation
@n insert label(s) of object attached to port n
$\text{n}$ recursively expand macros for objects connected to port n
(i) creating instances of interfaces

(ii) creating instances of functions

(iii) interconnecting these functions and interfaces in the appropriate manner so as to implement the desired function

The entire specification process is carried out by use of the mouse. The function or interface type to be included in the function being defined is selected from its hierarchical library. Its icon then joins a menu of ‘current functions/interfaces’ from which it is again selected to become ‘current function/interface’, before inclusion in the function being defined. Input and output ports to the function are created by placing instances of connection types, represented by icons. To read input received at a particular port a connection is made from that interface port to the appropriate lower level function required to deal with that input. Output from the function is sent to an output port in the same way.

When creating connections between functions it is necessary, unless the function is primitive, to specify the input and output ports on the destination and source functions. This is done by the use of pop-up menus which indicate the ports of each function and their connection-types by use of labels and icons, respectively. This is an elegant and highly visual means of creating interconnections.

Each function is given a \textit{threshold} and each connection a \textit{weighting} that is continuous-valued and user-defined. By default these are both \textbf{1.0}. At the primitive level these values are used as initial thresholds and weights in the compiled neural implementation, prior to learning, though they may equally well be left random and undefined. At a higher level, it is possible to use these values as \textit{high level} thresholds and weights. It may be that clusters of neurons in biological nets have an overall, \textit{high level} threshold. Also, there may be biological evidence that the weights of a group of synapses between two clusters increase and decrease their weights largely in unison. These high level connections may perhaps behave as if they have an aggregate weight.

There is no queueing of messages at input/output ports. If multiple messages are received at the same port in the same time-step, they are simply passed in unison along
3. Compilation: Formation by Specification

The term *Compilation* is usually applied in a computing context to mean: generation of machine code from a high level language. In the context of this thesis, however, it means the generation of a neural architecture from a high level specification. Chapter Four presents the framework within which this generation occurs. Two quite different methods of performing this compilation were implemented in ANNECS.

The first method attempted seemed at first consideration to be the most sensible. It was simply to flatten out the hierarchical specification, from the top down, whilst resolving multiple references to common instances of functions, until no non-primitives exist; that is, until the definition consists of a neural architecture. There are non-trivial problems involved in doing this which will be described later. After implementing and testing this approach it was seen, from preliminary experiments, that it was fundamentally limited.

Firstly, high level information had been discarded during the compilation process such that it was not possible to understand, at a non-primitive level, learning that subsequently took place. Secondly, learning algorithms could not exploit the hierarchy that was inherent in the compiled network. The grouping between neurons, clusters of neurons, synapses and bundles of synapses in biological nets is to some extent contained in their relative positioning in three-dimensional space. No analogue of this exists in traditional artificial nets and thus it is necessary to retain this hierarchical information during compilation. Hence, the second compilation method which was explored and eventually adopted retained high level structure. It formed a netlist of netlists, which was used by a non-primitive simulation model different from that implemented for the first compilation method. As it turned out, the second method was easier to code, resulting in 1300 lines as opposed to 1600 lines of code.

Compilation Method #1: Flattening

The key data structure underlying this method was a *cactus stack*. This is a vertical stack from which horizontal stacks grow outwards. Incidentally, this data structure is at the
core of the compilation process of other object-oriented languages. A major problem involved in flattening is to resolve references to a function in terms of which more than one other function has been defined. The compilation method is basically as follows:

1. Push components of top level function to vertical stack.

2. For each non-primitive interface or function (not connection) push the components in terms of which it is defined to a stack in the horizontal dimension (a spine).

3. For each non-primitive interface or function on vertical stack, replace it with its definition and resolve all connections to common instances, in terms of which more than one function is defined. This collapses the horizontal stacks.

4. Repeat steps 2 & 3 until no non-primitive functions or interfaces exist on vertical stack.

5. Take high level connections and flatten them into the lower level connections in terms of which they are defined.

6. Repeat step 5 until no non-primitive connections between functions, and thus no non-primitive interfaces, exist.

Figure 5.4 illustrates this process for the implementation of a specification of if...then...else in terms of logic gates. It should be emphasised that, whilst the resultant neural implementation is typically very large, the specification from which that implementation is derived is extremely concise and compact. This is a very powerful feature of this methodology. The primitive, and apparently structureless, compiled network does in fact contain hierarchy of function and can only be understood by reference to its specification, in conjunction with the compilation process by which it was
Figure 5.4  Stages in compilation of if...then...else by flattening

constructed (see section 5.5 for an example).
Chapter Five - Applying the Framework

Compilation Method #2: Resolution for High Level Simulation

In this method, the key data structure is a netlist of netlists. This structure is created on a one-dimensional stack by the following stages:

1. Push components of top level function to stack.

2. For each component of this function, if it is not an instance, or if it is an instance and has not already been loaded, then load its definition to the top of the stack.

3. Resolve all references to this component.

4. Repeat steps 2 & 3 until all function definitions are loaded: the stack then contains a netlist of netlists.

Netlist-of-Netlist Formation

It is necessary, during this process, to maintain a table of instances. An instance might be, for example, a function which detects an edge of certain length and position in an image and which is used by more than one higher level function. Connections must be created to/from the cluster of neurons that recognises the edge to each of the higher level functions, rather than creating two identical clusters of neurons which perform the same function.
The loading of functions and connection types is recursive and, at each level of recursion, the total amount of space required by the function or interface during simulation is computed. For example, an interface representing a five-by-six retina would require a vector of size thirty during simulation.

4. Simulation: Formation by Learning

ANNECS enables the simulation of a compiled network according to one of a number of models. Thus the same initial architecture can be made to learn according to different models without changing the specification. The model for neurons and synapses is selected separately.

Two different methods of simulation were explored, corresponding to the two methods of compiling.

(i) Flat Simulation – as in conventional neural simulators

(ii) High Level Simulation – composing and decomposing high level messages at simulation time, according to the hierarchical specification. (see figure 5.6)

The high level simulator is most worthy of comment. Input data is read from input/output files to interfaces with which those particular files have been associated. This data is timestamped before sending it along connections from that source interface to destination functions and interfaces, after which the timestamp for the source interface is increased by one. When data is sent to a function, it is placed on the correct i/o port and that port’s timestamp is set equal to the timestamp of the port from which the data came. Any functions to which data has been sent are also simulated. Thus, data propagates downwards through various levels of hierarchy to the primitive level, where primitive cell functions are performed. Simulation continues until all timestamps have been incremented. The processing that is performed is the same as that performed on flat nets in conventional simulators, except that there is hierarchy in messages and functions in the nets.
5. Example: A Simple Robot Controller

In order to demonstrate the principles of ANNECS it is useful to consider an example. The robot moves around in a world containing stairs, objects and holes. When it finds an object it should pick it up and carry it until it finds a hole, into which the object should be dropped. Every other time the robot meets a stair, it should climb it; when not due to climb a stair it should instead turn left. Our aim is to formulate a specification describing this behavior and have ANNECS implement this as a functionally equivalent net. This will enable us to understand the part played by each neuron in achieving the overall function of the net.

The robot controller has been defined as one high level object in order to observe its entirety (see figure 5.7). It could, of course, have been divided into smaller modules.
This specification is compiled to the network shown in figure 5.8.

Section 4.5 discussed the issues concerning representations, the fact that ANNECS is a framework and that it does not constrain specifications to one particular representation. The specification of the robot controller given in Figure 5.7, for example, contains a single line of control. It could be redefined to an alternative - though functionally equivalent - representation as in Figure 5.9. Here, control is distributed and the compiled network, though behaviorally equivalent, is slightly different in structure.
This representation makes better use of its target architecture, a neural network, in that control is more distributed. The first specification was effectively a completely pipelined sequential implementation. This illustrates the fact that certain styles of representation are more suited to neural realisation than others.

6. Improvements to ANNECS

ANNECS constitutes a major piece of software development, perhaps comparable to the implementation of a conventional high level language compiler. A problem with the implementation of ANNECS has been that the problems involved in neural specification are all but unstudied and thus no body of experience is available to guide development. This means that many improvements could be made to the software which, though not essential to the experimental results of this thesis, would enhance it.
Both methods of compilation could be provided so that (a) high level learning can be studied and (b) flat neural architectures can be downloaded for implementation on neural hardware or simulation by conventional simulators. More learning and/or neuron models could be implemented, and not all those that have been implemented have been tested. Alternatively, neuron and synapse models could be made user-definable. The automatic expansion of text using function macros is not fully functional.
Chapter Five – Applying the Framework

Having said this, the basic functionality of ANNECS is well-debugged and it is this that is required for the substantiation of the methodology espoused in Chapter Four. ANNECS is an embodiment of the framework put forward by this thesis and shows that neural architectures can be generated to implement any hierarchically-described distributed specification. Thus, ANNECS offers a potential means of combining genetic (construction by specification) and empirical (construction by learning) methods of construction.

The next chapter presents a case study which applies the ANNECS methodology to a real world problem. The application of ANNECS to numberplate recognition is compared to a conventional implementation of a numberplate recognition system which was developed alongside the main line of research.
Chapter Six

Case Study: Automatic Numberplate Recognition

1. Introduction

This chapter applies the methodology developed in the preceding chapters to a difficult real world problem, the problem of automatic numberplate (character) recognition. This task involves locating and then reading the numberplate, given a picture of the vehicle – and is an extremely difficult function to perform to high accuracy. Commercially available numberplate recognition systems typically achieve recognition rates of only 60-80%.\textsuperscript{165-169}

Chapter Five showed that the framework presented in this thesis can be applied to constructing neural networks. The methodology ‘works’ but whether or not it is useful will only be determined by its application to genuine engineering problems. Thus, the purpose of the work described in this chapter is to substantiate, by way of experiment, the effectiveness of the methodology.

In more general terms, this chapter describes work which explores the application of this methodology within the field of image processing. Image processing is concerned with deducing the three-dimensional representation of objects which produces a two-dimensional image.\textsuperscript{170,171} The way in which one might go about specifying neural implementations of standard image processing tasks such as thresholding, edge detection and segmentation is explored. Applications within the field include: security and surveillance; target detection and tracking;\textsuperscript{172} assembly line monitoring; reading printed or handwritten text for computer input;\textsuperscript{173-180} aids for the blind; analysis of medical images; and many more. A whole new market in these areas seems to be opening up due to the introduction of enabling technologies such as very cheap yet high-quality cameras.\textsuperscript{181} However, the problems yet to be solved are far from trivial. Tasks which humans find easy, such as recognising a face, are very difficult to perform artificially. To some extent the reverse is also true. Playing chess is quite taxing to most humans, yet it
can be performed to a high standard by computer. This thesis, confirming conclusions which might be drawn from the 60-80% recognition rates quoted earlier, testifies to the difficult nature of the problem of numberplate recognition.

This chapter first reviews some basic techniques in image processing and then describes some applications of neural techniques to the field. An overview of the numberplate recognition problem is then presented, followed by a description of a conventional (non-neural) approach to the problem, performed for comparative purposes as part of this research. Finally, the methodology advocated in this thesis is applied to the problem.

2. Basic Image Processing Techniques

Image Capture

It is not possible to perform any image processing unless there is some good means of obtaining images — that is, a sensor. Biological neural nets have two image sensors *par excellence*: eyes. Each of these sensors transmits its output down approximately one million parallel data paths — the optic nerve — to the image processor *par excellence*: the brain. These sensors have a non-linear resolution; the fovea, the area at the centre of the retina and thus at the centre of the visual field, contains orders of magnitude more ‘pixel sensors’ — *rods* and *cones* — than other parts of the retina. Biological sensors perform neither grayscale nor colour sensing exclusively but combine both. Grayscale sensing is used for certain functions to which it is best suited, such as edge detection and object recognition. Colour enhances classification and recognition functions. In addition to this, primitive processing such as edge detection is performed actually within the sensor. The output transmitted down the optic nerve to the brain consists of edges and perhaps other primitive data such as texture, as well as colour and grayscale.

Artificial sensors, on the other hand, typically output composite video at fifty frames a second, with a resolution of the order of a million pixels. The video output is normally sampled and digitised by a *frame grabber* which outputs a digitised representation of the picture suitable for computer storage and analysis. Constraints imposed on the image processor by the sensor include: the horizontal and vertical resolution, the dynamic range and response profile of each pixel sensor, contrast and exposure control. If the image
capture is performed badly, the subsequent image processing is constrained by this. This principle also holds true for stages within the image processing process. The results of each stage can only be as good as the results from preceding stages. In numberplate recognition, for example, if the initial thresholding is performed poorly all subsequent processing will inevitably suffer as a result.

Edge Detection

An edge may be defined to be an area of pixels where the rate of change of intensity is greater than some threshold. If an image is ‘differentiated’ in the horizontal dimension, vertical edges in the image correspond to peaks and troughs within the derivative. There are many different methods for performing edge detection and the underlying theory is well understood.\(^\text{182}\) Perhaps the best-known and computationally most useful method is the one proposed by Canny.\(^\text{183}\)

The human eye performs edge detection by means of lateral inhibition.\(^\text{5}\) Linsker has shown that multilayer perceptron-style architectures using Hebbian learning produce edge detection functions, even with random training data.\(^\text{36}\) Edge detection is probably the most basic image processing operation carried out in the visual cortex and is certainly essential for all higher level operations such as determining shapes and hence recognising objects. What is of interest to this thesis is whether or not edge detection functions can be specified and compiled to neural structures which approximate those found in biological nets.

Various parameters are usually supplied to ‘artificial’ edge detection algorithms. These include factors such as: the lateral distance over which to consider changes in intensity; the threshold over which the change in intensity must be before it constitutes an edge; the number of adjoining pixels which must be considered parts of an edge before an edge can be considered to be present. But these linguistically-described parameters are merely crude expressions of what is better mathematically expressed and analysed.\(^\text{112}\)

The conventional numberplate recognition algorithm presented in this chapter first performs thresholding, followed by edge detection on the resultant binary (black and white) image. This edge detection on a binary image is very simple to perform and is
described in section 6.5. In general, thresholding and edge detection are very closely related; if it is possible to edge detect, then it is usually possible to threshold, and vice versa. However, thresholding discards more information than edge detection. Thus, it is only suitable as the first processing stage for applications such as numberplate recognition where the objects to be recognised are originally binary in nature.

**Thresholding**

Thresholding is the transformation of a grayscale image to a binary (black and white) image. It is one of the hardest tasks to perform in the conventional numberplate recognition algorithm. It consumes half the total processing time and over half the total development time was required to achieve satisfactory results. If the thresholding is not done well, all subsequent stages are doomed!

Methods of thresholding may be divided into global and local techniques.\textsuperscript{184} Global methods choose, on some basis, a threshold to be applied to every pixel in the image. Conversely, local methods choose a different threshold for each local patch of the image. The threshold for each block is typically determined from the grayscale values of local pixels at run time, and thus the method is often called *local adaptive thresholding*. The main problem involved in this is to find the best grayscale (threshold) such that when all pixels with intensity greater than this are made white and all pixels with lower intensity are made black, the resultant patch of image is most useful to subsequent segmentation and recognition stages. Thus, in numberplate recognition it is desirable to select the best threshold such that the black of characters and the white of the background plate are clearly disambiguated. It is conceivable that the lower part of the plate will be in sunlight whilst the upper part will be in the shadow of the bumper. Thus, the ‘black’ of the bottom of the characters can be lighter than the ‘white’ of the background of the top of the plate. This example illustrates the necessity for choosing thresholds locally at run time. The threshold for the bottom of the plate should be higher than the threshold for the top of the plate.

Three main methods of deriving these thresholds were explored whilst developing the conventional recognition algorithm. These were:
Mean Thresholding – the mean of a block of pixels’ intensity values is used as the threshold. This method is simple and of use where the grayscale information is approximately equally distributed about the optimum threshold.

Median Thresholding – the median of a block of pixels’ intensity values is used as a threshold. This method is also simple and is of particular use in applications such as fingerprint recognition where it is desirable to have approximately half the image black and half white such that, for example, the bands of a fingerprint are of approximately equal width.\textsuperscript{185}

Histogram Thresholding – this includes a large class of techniques which examine the shape of the histogram of a block of pixels’ intensity values in order to derive a threshold.\textsuperscript{186} These methods are generally computationally more expensive but are also more versatile. Comprehensive mathematical analyses of these have been performed.\textsuperscript{184,187}

The mean and median methods of thresholding were found to give insufficient performance, and both for the same reason. If a block of pixels happens to overlap a character on the plate such that there is either more character than background or vice versa, the shape of the histogram will consist of two humps of unequal size. It is the trough between these humps where the threshold should ideally be placed (see figure 6.1) but both mean and median methods will shift the threshold from the trough towards the larger of the humps.

In practice, of course, smoothing must be performed on the raw histogram before anything else can be done. Ideally, two peaks will emerge from this process with a good intervening trough where the threshold may be placed. Adaptive smoothing is often required, however, since too much smoothing removes these peaks completely whilst too little leaves too noisy a histogram. This problem is returned to later, in the description of the conventional recognition algorithm in section 6.5.

Segmentation

At some point in the processing of an image it is necessary to take the results of low level, local operations such as thresholding and edge detection and to build more global
Figure 6.1 Histogram and potential thresholds for a block of an image

representations of objects and scenes. Segmentation is one stage in the transformation of local, low level results to global, high level representations. For most applications it is very difficult to segment an image on the basis of raw image data. It is more usual to perform segmentation on the basis of edges, blocks of thresholded images, texture and surfaces (normally deduced from edges), feature points, stereo maps, and so on.

For example, in conventional numberplate recognition, two stage segmentation is performed:

1. The thresholded image is divided into blocks of pixels which are separable from the background and could thus be characters.

2. Characters are divided into blocks and the parameters of those blocks are used as data for the classification process.
Segments are typically used as components from which higher level objects are formed, perhaps using a hierarchical representation. For example, a car consists of a body + wheels; a body consists of a bonnet + a middle + a boot; and so on. Figure 6.2 shows the main processing stages in the conventional numberplate recognition algorithm.

**Measurements/Classification**

Having segmented an image it is necessary, in order to build a three-dimensional representation of it, to relate these segments in some way so as to deduce the nature of higher level objects. This is done by the measurement of segment parameters such as: size, texture or mean intensity, shape, perspective. It is also done by the measurement of relationships between segments, such as: the two-dimensional distance between them, the three-dimensional distance between them, and so forth. For example, in face recognition it may be possible to characterise the face by factors such as distance between the eyes, nose and mouth, and the size and shape of the eyes or perhaps eyebrows. In numberplate recognition, the plate — a high level object — is formed by considering the distance between character segments. Characters are classified according to the parameters of the segments from which they are composed.

**Miscellaneous Techniques**

This section looks at various image processing techniques not of direct relevance to this research. A well known image processing operation is the **Fourier Transform** which extracts from an image information in the frequency domain. This technique was explored as a method of locating the numberplate in an image. Ideally, a horizontal Fourier Transform of an image should give high frequency components for scan lines containing the numberplate, owing to the sharp contrast between characters and plate. Unfortunately, however, other objects such as the grill on the front of a car give rise to conflicting results.

A general purpose image processing operation is a **convolution**. This involves passing a *mask* pixel by pixel over the image in order to transform the image in some way. Each pixel becomes the sum of the product of each element of the mask with the pixel it covers. This technique can be used to perform primitive operations such as edge
detection, smoothing and contrast enhancement. Most image processing accelerator boards include this function, owing to its versatility.
Optical techniques can be used to transform a ‘normal’ image in some way before capturing it by sensor. Headlights can make an adjacent numberplate unreadable using the visible spectrum. Thus, for some applications it may be desirable to use an infra-red or ultra-violet source and to filter out the visible spectrum.

**Stereo** capture and processing is necessary for true three-dimensional vision. This is of little relevance to numberplate recognition owing to the two-dimensional nature of characters. However, it is of interest to us in view of the fact that humans use stereo for three-dimensional interpretation of scenes. It is thought that biological nets probably compute the *depth* of relatively few points and infer the third dimension of other points from cues such as object characteristics.

3. Neural Techniques in Image Processing

There are too many neural models of image processing to allow a comprehensive review in this section but what follows is a representative sample of work from the field.

**Neocognitron**

This model was developed by Fukushima et al, primarily to perform character recognition.\(^{188,189}\) It consists of multiple layers, the higher layers containing successively fewer units than the lower layers. Each layer combines features produced by the preceding layer so as to produce higher and higher representations. Thus, the bottom layer performs primitive functions such as edge detection, whilst the output layer combines segments so as to classify characters. These functions are generated by the application of *competitive learning* (see section 2.2).\(^{147}\) Rotation and translation invariance is achieved by having identical feature detectors operating at multiple points in the image. Unfortunately, an effect of this is to make the number of units so large that the model is computationally inefficient.

**Silicon Retina**

This VLSI implementation, developed by Carver Mead, models the way in which primitive image processing is performed in the brain.\(^{190}\) The biological retina has been
closely studied and is thus understood well enough to attempt an artificial implementation of it. The aim was to implement retinal functions not merely functionally but also in the way in which they are performed. For example, the logarithmic response profile of rods and cones is performed by the sensor using analog circuits. The silicon retina produces an output signal which is invariant to size and rotation. Other analog implementations of retinal operations have been performed by Van der Spiegel et al.\(^1\)

Connectionist Models

Feldman and Ballard have carried out extensive analysis of the problems in applying connectionism to image processing.\(^ {99,192,193}\) Results have shown that the internal data representation is vitally important (see also section 4.5). As an aid to their work a neural network simulator called ISCON was developed, and this is now widely used.\(^ {164}\)

Self-Organisation in Primitive Vision

Linsker has achieved remarkable results by applying Hebbian learning to multilayer perceptron-type architectures.\(^ {36}\) He has shown that, even in the absence of any real world input data, primitive functions such as edge and texture detection are learnt. Structures that are generated from this learning process seem to parallel those found in the biological retina. What is unexpected in these results is that primitive image processing functions are learnt when \textit{random} data is used as a training set. This could explain how mammals are born with the ability to recognise edges in spite of the fact that it is very unlikely that the structures to perform edge detection are genetically specified.

Grossbergian Boundary and Feature Contour Systems

These models perform edge detection, join edges to form parts of boundaries, complete those partial boundaries to form complete boundaries, and then fill in the colour/intensity/texture for each segment contained by a boundary.\(^ {194}\) There is some parallel with cell structures found in primitive vision areas of mammalian brains. An explicit distinction is made between \textit{boundaries} and \textit{colour/intensity/texture} and two distinct but closely-interactive modules, the boundary contour system and the feature contour system, exist to handle each of these aspects.
Head-Centred Frame of Reference

This model consists of a multilayer perceptron trained using backpropagation. Input to the network consists of an image containing some object and a representation of the degree of extension of the eye muscles. The network is trained to translate the retinal input to a head-centred frame of reference. Thus, the object in the field of view is mapped to the same head-centred reference point, regardless of which way the eyes are turned.

Binocular Disparity

The brain computes depth information by combining output from two sensors separated by about 6.5 cm. The structures which perform this operation are to some extent observable and have motivated a model developed by Schwartz and Yeshurun. Their work emphasises the role of computational maps (c.f. Kohonen nets) in the visual cortex.

4. Overview of Numberplate Recognition

Requirements for Numberplate Recognition

Use of numberplate recognition systems has shown that if they are not highly accurate they are of no use at all. Current commercially available systems typically exhibit recognition rates of 60-80%. No highly accurate and cost effective numberplate recognition system yet exists. The reason for this is that the problems involved are extremely difficult to surmount, contrary to what one might at first think.

An application of numberplate recognition which springs to mind is in road pricing. Here, drivers are charged for use of a road perhaps according to its location, the level of congestion and the time of day. In fact, numberplate recognition is not, and never will be, a sufficiently reliable means of identifying a vehicle in order to charge its driver. Electronic tagging is a more dependable technique, though not without its problems, and has the added advantage of allowing transmission of information such as congestion maps to the vehicle. However, it is necessary to have some method of enforcing an electronic means of identification, at least in the short term – until tags are integrated into
car manufacture. In current road pricing systems numberplates are used as a means of identifying offenders of the system.

It is where 100% accuracy is not required that automatic numberplate recognition can be most useful. If numberplates can be identified and matched at key points in the road network it becomes possible to extensively analyse the speed and direction of traffic flows. Logging of numberplates passing these sites would also be of use to the police – for example, in control of terrorism. Automatic access to private car parks could also be controlled by numberplate recognition.

In general, numberplates are a relatively inaccurate method of identifying vehicles. Almost certainly, the results of automatic recognition could not be used in court, even though a picture of a speeding vehicle may perhaps constitute evidence in the future. In spite of this inherent inaccuracy, however, applications do exist.

**Problems involved in Numberplate Recognition**

Software to read printed document text that has been scanned into a PC is widely available and is often cheap. Such packages typically achieve accuracies of around 90-100%, depending on the textual quality of the source document. If this character recognition task can be performed with such high reliability then why cannot similar accuracy be obtained in numberplate recognition? Indeed, one would think numberplate characters are easier to recognise, owing to their block-like font which is specially designed for clarity.

The problem in numberplate recognition is not *reading* the characters but *finding* them. The classification of a numberplate character, once it has been located, is relatively easy and can be performed to high accuracy. This insight might lead us to try to recognise a plate by attempting classification of all segments in the image. To some extent, humans seem to recognise things in this way. We seem almost to locate characters by reading them. Certainly, the location and classification processes are closely related and affect each other intimately in the recognition process.

Finding by reading/classifying is too computationally expensive for non-biological methods of computation. In order to apply the classifier it is necessary to know the
height, width and rotation of the character. Information emerging from lower level processing, such as edges and segments, may be used as cues to this process to prune the search space, but even this approach is flawed (see figure 6.3). In order to generate all possible character locations it is necessary to use feature points, which are quite a low level representation. The number of ways in which these points could be related to form potential character locations is enormous.

The numberplate could be in bright sunlight. It could be in deep gloom. It could have blazing headlights next to it. It could be hanging at an angle. The camera might not be mounted head-on to the vehicle. It might not be mounted so that plates appear horizontally in its field of view. The characters could be in any one of about eight different fonts. The plate could be foreign. It might not obey British syntax. It could have a badge or a ‘smiley face’ in the middle of it. The characters could be black on white or they could be white on black. A towbar could stick up and partially obscure some characters. A bumper might obscure the top of the plate. The plate might be secured to the vehicle, usually on lorries, by means of a black band round it which lies across each of the characters. Snow or rain or even fog might obscure the plate. The characters within the plate could be irregularly spaced, such as:

24 BUS

Next to the plate there might be text in a similar font, such as:

WAYNE LUVS SHARON

or: RANK TAXIS

It is not known how far away the vehicle is. The numberplate could be at the top, bottom or side of the vehicle. It could be travelling fast, requiring fast real-time processing.
The problems involved in numberplate recognition are becoming plain!

<table>
<thead>
<tr>
<th>'W' separated into two 'I's</th>
<th>All characters joined to bumper</th>
</tr>
</thead>
<tbody>
<tr>
<td>White characters on black plate</td>
<td>SUZUKI, L, ALVINS</td>
</tr>
<tr>
<td>Part of plate missing/cracked</td>
<td>CSH joined to background; two rows</td>
</tr>
</tbody>
</table>

Figure 6.3 Illustration of difficulty in locating numberplate
Potential Solutions

There are two broad approaches which might be attempted:

A. Locate characters before attempting classification.

B. Attempt classification on several parts of the image based on cues such as edges and the location of characters which have already been recognised.

Method A was taken by the conventional implementation (section 6.5) whereas method B is more suited to neural implementation (section 6.6). With the neural implementation it is not the aim to produce a real-time system since, as described in section 2.1 – *Very Artificial Neural Networks*, the parallelism of biological neural nets is far beyond present artificial capabilities. Instead, what should be shown is that the methodology advocated in this thesis can be applied to produce a neural implementation that *would* recognise in real time, given hardware of the capability of biological hardware. Thus, the conventional implementation processes an image orders of magnitude faster than the neural implementation. This, of course, is because conventional algorithms are suited to running on conventional hardware whereas neural implementations have to be simulated sequentially.

An overall strategy for solving the recognition problem for each of these methods is given below. (See also figure 6.2.)

**Method A:**

1. Threshold
2. Edge detect
3. Segment
4. Sort segments to local groups and select group containing numberplate
5. Classify characters
Method B:

1. Edge detect
2. Classify edges to form segments
3. Classify segments to form characters
4. Group recognised characters

5. Conventional Numberplate Recognition with near-100% accuracy

Introduction and Results

This section describes a non-neural implementation of a numberplate recognition system which was developed in order to explore the problem prior to applying the neural methodology. It provides an excellent benchmark against which to compare the neural implementation. In addition to this, it allows the examination of the difference between the approach which is natural to a neural solution and the approach which is natural to a traditional solution, for a variety of image processing operations such as edge detection.

The conventional algorithm is implemented in ‘C’, circa 6000 lines and executes on a Sun4 workstation in approximately 15 sec. The algorithm was initially developed on a test set of thirty images (512 by 512) captured using a CCD camera and a framegrabber. The development environment was Unix on a SUN 3/80. The total development time was in the region of one year. After initial development, the algorithm was tested on a further set of 170 images (512 by 512), captured using a camcorder, of stationary vehicles in on-campus car parks.

Further development took place which gave rise to the following results:

- **99.43%** – vehicles for which at least part of the plate was correctly read.
- **98.86%** – vehicles for which the whole plate was correctly read.
- **99.94%** – percentage of characters which were correctly read.

† Framegrabber used: Data Translation – Model 1451
At first glance these are remarkable results. However, on a closer consideration it will be observed that the algorithm was developed on this image set and thus has been forced to function as well as possible for each individual numberplate. Hence, these results cannot be considered to be a fair trial of the system. It is anticipated that a full scale trial involving 1000+ images will be performed in the near future. The results of this trial, however, will not be of direct relevance to this research. What matters is that the development of both conventional and neural implementations was based on a substantial amount of test data.

It should also be noted that these figures are recognition rates as percentages of what was humanly-readable from the same image set. Requirements for a character to have been correctly read were reasonably tolerant: it was permitted for an ‘O’ to have been read as an ‘0’ or as a ‘D’. Syntax forcing can, in many cases, disambiguate these similar characters.

A ‘workbench’ has been implemented which allows manipulation of images intermediate to the various stages in the processing. Figure 6.4 illustrates the stages involved in conventional recognition, whilst figure 6.5 shows the relative execution times of the main stages.

Thresholding

This stage transforms the raw grayscale image derived from the camera into a black and white, binary image. Local adaptive thresholding based on histogram distribution analysis is used.\textsuperscript{184,186}

In order for the system to approach in accuracy what is humanly readable, the requirements for thresholding are rigorous. The algorithm should be able to determine a good threshold even if all the information is contained in only 20% of the dynamic range, and even if that 20% may be at any point in the range. It is also possible that one part of a character may be highly exposed (e.g. bright sunlight) whilst another part of it may be under exposed (e.g. in shadow of the vehicle bumper). Thus, it is necessary for thresholds to be chosen and applied locally and adaptively.
Figure 6.4 Images at main stages in conventional processing of a numberplate

The algorithm operates on non-overlapping 8 by 8 pixel blocks of a 512 by 512 image with 256 grayscales. The histogram for each block is formed with no subsampling, and
some smoothing is applied. Obviously, smoothing increases the chances of picking a
good threshold (within limits) but decreases sensitivity to thresholding ‘black’ and
‘white’ which differ in intensity by, say, only 10-20% of the dynamic range. After
smoothing, a ‘goodness’ function is applied to every $peak_1$-trough-$peak_2$ combination
where $peak_1$ and $peak_2$ are either side of trough but not necessarily direct neighbours of
trough. This goodness is computed on the basis of various factors such as peak height,
trough-to-peak height, trough-to-peak width, and so on. The trough with the greatest
goodness is chosen as a provisional threshold. This threshold is then examined against
thresholds derived for neighbouring blocks, and its validity on this basis determined.
Some rationalisation and smoothing of thresholds is performed before they are applied to
the grayscale image, resulting in a binary image.

At first sight the generation of all $peak$-trough-$peak$ combinations seems an absurdly
inefficient way of finding a good threshold. In practice, however, the average number of
combinations is around seven, and the worst recorded case for this test set was 127
combinations. Several alternative histogram analysis methods were investigated before this one was adopted.

Cleaning

This is a simple filter which removes pixels not strongly joined to a cluster. That is, a pixel is inverted if the number of its neighbours with the same intensity is less than some threshold.

Edge Detection

This process inverts each black pixel if all its immediate neighbours are also black. Edge detection on grayscale images can be extremely complex but since this process operates on a binary image it is comparatively simple and very quick to perform.

Object Detection

Object detection tracks edges within the image to derive top, bottom, left and right coordinates of distinct clusters. It also records coordinates of significant features such as corners, forks and extreme limits of curves in various directions.

The process operates on an edge-detected binary image. At this stage, edges are all that is needed to find distinct objects and pull out significant features. A distinct object is one that is separated from the rest of the image and thus can be detected by tracking along its boundary and recording the extreme limits reached in the x and y directions. An object whose boundary is continuous and thus joins up to its starting point is likely to be a character and is given a weighting to this effect. This weighting is taken into account, along with other factors, in the next stage, when objects are filtered out such that only characters remain. Since this process operates on edges within the image, black characters and white characters will be detected in the same way, because the edge-detected images for black and white characters are the same.

As the algorithm tracks along edges, it examines each pixel for significance as a feature. A number of factors, such as the location and direction in which a line is moving, are taken into account in order to enable recording of significant features such as corners,
forks, extreme points of curves, and so on. These are used in the next stage to locate characters that were non-separable from the background.

This method of finding objects was chosen after experimentation with various other methods such as the Fast Fourier Transform (FFT) and pixel thinning. With the FFT the problem was that other high-frequency parts of the vehicle, such as the radiator grill, could not be distinguished from the high-frequency components obtained from the plate. Thinning will, if carried to its limits, separate characters that are joined to the background, but in the process has the potential to change characters. For example, a ‘T’ whose top is joined to the background may be transformed into an ‘I’.

Object Filter

This process takes the coordinates of features and distinct objects and, by analysing these in conjunction with the thresholded image, produces the coordinates of the characters within the plate, plus a measure of confidence that a character is actually present. This is achieved in a number of stages:

1. Look for characters that are non-separable from the rest of the image, based on feature coordinates and the location of distinct objects.

2. Sort all objects (potential characters) by size and location into groups; these groups constitute potential parts of the plate.

3. Separate joined characters; this procedure is applied iteratively so that objects that consist of more than two characters that have been joined will be successfully separated.

4. Merge parts of the same plate; this involves matching groups to see if they are parts of the same plate. This enables plates which contain characters on more than one horizontal level to be identified.

5. Remove objects within objects: for example, the inside of an ‘O’ will be detected as an object, since it is a continuous edge, and should be removed.
6. By analysis of the distribution of black and white in each object compute a measure of confidence as to its likelihood of being a character.

7. Compute the likelihood of each group of objects being the plate, based on factors such as: number of objects in the group; likelihood of each object in the group being a character; relative heights of objects in the group; and so on.

9. Select the group that comes top and pass it, with a measure of confidence in each hypothesised character and the angle by which the plate must be rotated to make it horizontal, to the next stage in the processing.

Character Classification

This uses a trained decision tree $^{197,198}$ with breadth-first, fuzzy search to obtain the $n$ most-probable characters. It is this stage that normalises for size, translation, rotation, perspective and font.

Character reading is performed by segmenting each character according to the data received from the preceding processing stage, such as character height, width and rotation, and using the parameters of each segment as branching factors in the traversal of a decision tree. $^{199}$ This tree is formed by training on character sets of all standard fonts. Breadth-first, fuzzy search is employed to obtain the $n$ most-probable characters, and a measure of confidence in each result. $^{200,201}$ This confidence measure is combined with the likelihood passed from the preceding stage to give an overall confidence for each character and for the whole plate. As reading a character consists of descending the tree down at most $n$ branches, the time to read is $O(n \log_2 m)$, where $m$ is the number of segments into which the character is divided, equal to the tree height. Thus, the compute time to read a character is low. The tree is a sparse binary tree which, when well trained, consists of $O(300m)$ nodes. The criticism of the use of decision trees in pattern recognition has been the heavy accumulation of errors at each branch. The overall error is limited in this case by keeping the number of segments (and hence the number of branches) $m$ low and by using fuzzy search. The technique is surprisingly resilient to increase in noise and to breaks in character contours, unlike some other classification techniques.
The fact that this technique is based on training provides a good comparison with neural classification which is also derived through training. The amount of training required to generate a discriminative tree is low compared to traditional neural learning times, where the network contains no explicit specification of a priori knowledge. In the first 100 images presented to the system, only 11% of characters were required for training in order to give 100% accurate recognition.

The classifier defaults to trying to read black characters on a white background. If the confidences are extremely low for most characters, it assumes that the characters must be white on black and thus inverts the segments and reclassifies the characters.

6. Neural Numberplate Recognition

Introduction

This section describes the application of the framework of hierarchy, by use of ANNECS, to three stages in the numberplate recognition algorithm. The intention is to determine the usefulness – in engineering terms – of this method of implementation, and to substantiate the use of hierarchy as a method of constructing and understanding neural systems. The method is applied to local adaptive histogram-based thresholding, edge detection and character classification. These three stages were selected for detailed examination because each illustrates, in a different way, the importance of hierarchy. The histogram-based thresholding, in particular, is not an application suited to traditional neural models. Thus, it demonstrates the generality of this method of construction, the power of which arises from its integration of specification and learning. More specifically, the aims of these experiments are:

(a) to compare ease of neural implementation using the framework of hierarchy with ease of implementation using conventional programming;

(b) to determine whether the functioning of neural systems constructed with this method may be understood;

(c) to determine whether specification and learning may be integrated within the method;
(d) to determine whether the application of hierarchy facilitates the scaling-up of networks to 'system' level.

The character location stage in the conventional (non-neural) algorithm has not been implemented neurally as it is a fundamentally sequential method with much random access of pixels and thus is not naturally suited to neural implementation. This does not mean to say that the framework is not applicable to character location but that results from the experiments performed are sufficient to realise the above aims. A method of location more natural to neural implementation is discussed later, in section 6.6.4.

**Neural Local Adaptive Histogram-Based Thresholding**

The manually-generated hierarchical specification for a neural system that performs this task is given in Appendix B. The functionally equivalent 'C' implementation of this is given in Appendix E. The neural specification basically implements the same function as the 'C' specification, but in a fully distributed, parallel way. At each level of abstraction in this neural specification, the functioning of the system can be understood. The specification is similar to an object-oriented model in that each 'object' (i.e. cluster of neurons) is continually existant within the resultant implementation, rather like a process. Also, each 'object' is continually receiving and sending messages, which may be high level data representations. For example, the 'form histogram' function receives an 8x8 patch of an image (which is ultimately implemented by ANNECS as 64 synapses) and transmits the 32-bin histogram (implemented as 32 synapses) of the grayscales within this patch. A datatype '8x8 patch' has been defined as consisting of four 4x4 patches; a 4x4 patch is defined as four 2x2 patches; these are defined in terms of pixels; and a pixel is defined as a primitive connection, or synapse.

Similarly, the datatype 'histogram' has been defined in terms of 32 primitive connections. Thus, a histogram is represented as the activities along this number of synapses. This demonstrates the power of applying hierarchy to neural systems. At the highest level, an entire image is presented to the system by the simple creation of an interface of type '512x512 image'. This is compiled by ANNECS to 262144 primitive connections, but the designer treats these as one, high level data path. Thus, this image can be passed to any
lower level functions (as shown in Figure 6.7) simply by the creation of a connection (also of type ‘512x512 image’) to the appropriate function. This hierarchy of data representation, combined with the hierarchy of function, is what permits the neural system to be understood at all relevant levels of abstraction. This sort of high level
abstraction of function and data seems also to be present in biological neural systems. The optic nerve is an obvious analogue of the ‘image’ datatype defined in ANNECS. Primitive connections, which implement these high level representations, cannot be understood in isolation. It makes sense to abstract function, and data. It is this interrelation between levels of abstraction, that patently makes sense to the human designer, which provides understanding of neural systems.

![Image](image.png)

**Figure 6.7 Top Level Specification of Neural Numberplate Recognition System**

The neural implementation of histogram-based thresholding is basically the same as the conventional approach, except in the method of threshold selection. As before, the 32-bin histogram of each 8x8 patch of a 512x512 image is formed. This data is represented by the activities along 32 synapses, but is treated as one high level type. The method by which this histogram is formed is precisely specified, as shown in Appendix B. The function which then selects a good threshold from this histogram is realised as — effectively — a two layer multilayer perceptron, with one hidden layer containing 6 units. In actual fact, nearly all neurons in the compiled system are ‘hidden’; within the framework of hierarchy, however, no neurons are actually hidden: the role of every neuron can be identified.

Specification was used to determine the topology of the threshold-selection part of the system; learning was then applied in order to derive weights appropriate for the selection of a good threshold. This demonstrates the integration of explicit specification and empirical derivation within this methodology. The training data for the learning was obtained from the output of the conventional thresholding algorithm, using these thresholds as an oracle. It was specified that each of the candidate thresholds was to be defined in terms of six features. What these features were, and how each threshold was defined in terms of them, was entirely learnt.
The performance of the resultant – specified and trained – net, was as good as the performance of the conventional approach. 3168 neurons, interconnected by 14624 synapses, were required to threshold one 8x8 patch. Thus, for a completely distributed threshold of a 512x512 image, of the order of $10^7$ neurons and $6 \times 10^7$ connections are required. This implementation thresholds the image in seven update cycles (assuming a digitally-based simulation model). Alternatively, if the image is multiplexed onto the network that thresholds just one patch, $((512/8)^2 + 7) = 4103$ update cycles are required (using pipelining).

Figure 6.8 An image thresholded by the histogram-based neural implemention

The size of the training set was limited to 1000 samples, selected from regions surrounding and including the numberplate, within several images differing in exposure. When the training set was increased significantly in size, the network failed to converge
on a solution. If the training process is carried out for a significantly increased number of epochs, the model fits the training data too well and gives poorer results for the test data. If the training data was selected entirely from one image, test images with similar exposure conditions were thresholded well, whereas images with different exposures were not.

Finally, it should be noted that a reasonably large, yet highly efficient and sparse neural system has been constructed. This was due to the application of specification by the use of hierarchy, combined with learning. See figure 6.8 for an example of a thresholded image produced by the neural implementation.

**Neural Edge Detection**

The same principles as described above are employed in this task. Again, the method by which edges are detected is described hierarchically (see Appendix C). An edge is detected in the horizontal, and in the vertical, directions and in each direction a black-to-white and a white-to-black edge is detected. No training is necessary to realise satisfactory performance using this method (see figure 6.9 for a neurally edge-detected image). Weights are preinitialised to implement the desired function. Since it is easy to describe this function, there seems little point in trying to learn it. This is in contrast to the threshold selection problem described in the previous section, in which it was not known how to select a good threshold. In that instance it was appropriate to employ learning.

An edge is detected across a maximum width of five pixels. To edge-detect a 512x512 grayscale image, with no multiplexing, of the order of $2 \times 10^6$ neurons and $7 \times 10^6$ synapses are required. This edge-detects the image in three update cycles (see Appendix C). Unlike the conventional implementation of this function, the neural implementation will operate on either grayscale or binary images. The conventional implementation was only required to edge-detect binary images (see section 6.5.4).
Neural Character Location

The conventional character location algorithm achieves its task by tracking along edges, flagging feature points, and performing much sorting and grouping of objects. This algorithm is not suited to neural realisation — though that does not mean to say this cannot be done. A solution more natural to neural implementation is to locate by recognising, much as humans seem to do. If the character classification stage is sufficiently good, and is fast, characters may be located by scanning the image and attempting to read a character at each location. Since it is not known how large the characters are, it is also necessary to attempt to read characters of several different sizes, at each location.
This approach was investigated by use of a neural character classifier developed in the following section. It is necessary to apply a threshold to the results of the classifier, above which it is concluded a character is present, and below which it is concluded it is not. The classifier employed was trained on just one example (and thus only one font) of each character, and consists, essentially, of a two-layer, fully-interconnected MLP with 30 inputs, 15 hidden units and 32 output units. During recognition on the 170 images, and using a threshold of 0.3, 17% of locations not close to a character were falsely identified as being characters. 11% of locations where a character was actually present fell below this threshold. The average of the highest outputs (from the classifier) for correct character locations was 0.6915, whereas the same measure for locations where a character was not present was 0.2259. The erroneous location of characters would be virtually eliminated by the grouping of character location information at a higher level. If a character has been ‘strongly’ located in a neighbouring position to one that has been ‘weakly’ located – and in fact falls below the threshold – this information can be used to positively locate that character.

This approach was not exhaustively investigated because, in fact, it is neither supportive nor destructive of the thesis. It does, however, indicate a method of location that is natural to neural implementation.

Neural Character Classification

This task would traditionally be performed neurally by a multilayer perceptron (MLP) or a Kohonen Net, or some such classifier. Taking the hierarchical approach, however, the resultant implementation is not specific to any of these architectures. What is of concern, is that the neural system can be understood, at all levels of complexity. It so happens that the structure compiled by ANNECS from the specification supplied is a sparse MLP. However, if it had been decided to implement a different solution, the resultant implementation might have been similar to a Kohonen net. Which of these models the structure happens to be is not relevant. What is important is that, by the application of hierarchy, these systems are meaningful. Their construction is directed towards a solution according to the principles of the framework, not by picking almost at random a model just because it has ‘worked’ for similar problems. For the sparse MLP which ANNECS compiles from the specification, the number of hidden units is derived from a priori
knowledge about the problem – not by trial and error. In traditional approaches, however, this number is arrived at empirically, often by sheer guesswork.

Using ANNECS, the classification of characters was defined in terms of features from which characters are composed. These features are obvious to the human designer. For example, it is clear that an ‘E’ consists of a black column-1, row-1, row-3/4 and row-6. These columns and rows are patently features from which an ‘E’ may be recognised and, incidentally, from which many other characters such as ‘F’, ‘T’, ‘D’, ‘B’, ‘H’ may also be recognised. Because of this, it is appropriate to define the functions which detect these features as \textit{instances}, and then to define characters in terms of the same instance of each feature-detector.

This knowledge concerning how characters are written is imparted through specification, the hierarchical nature of which makes it meaningful to the designer. Learning can then be applied to optimise these ‘approximate’ classifications, and perhaps to learn classifications of those characters for which it was not easy to specify their features. Interestingly, when learning was applied from a random initial state, with no meaningful structure built into the ‘MLP’, the features that were learnt, from which classifications were made, were unlike the ‘obvious’ features first specified in ANNECS. Such a set of completely-learnt features is shown in figure 6.10. This does not mean to say either that a part-specified or that a learnt solution will be best. However, a solution that is part-specified has a greater chance of learning a ‘good’ classification than a system which is wholly empirically derived, even though its resultant performance will not necessarily be as good. The problems concerning convergence in, for example, multilayer perceptrons have already been discussed (see section 2.2.2). Such a system cannot be \textit{guaranteed} to learn a good classification, due to the existence of local minima in the solution space. Single layer perceptrons, however, have been proven to converge to a solution, if such a (linearly separable) solution exists. By the application of the framework of hierarchy, the numberplate character classifier can be guaranteed to converge to a solution (provided, of course, that such a solution as has been part-specified can actually exist). Because the classifier is part-specified, and thus the role of (most) neurons is known, training can be carried out layer by layer.
This process was performed based on a training set of one instance of each character (in only one font). 13 primitive features were specified and then trained. This training was enabled by specifying which features were present in which characters. The perceptron convergence algorithm was used, since the specification of features is effectively a sparse, single-layer perceptron. These part-specified, part-learnt features are shown in figure 6.11.

These features were then ‘frozen’ (their weights were locked) and character classifications were learnt, in terms of these primitive features. Two other primitive features were permitted to be learnt, at this stage, to allow the learning of any discriminative features not obvious to the designer. Again, this learning consisted essentially of the perceptron convergence procedure and was guaranteed to converge, given that a classifier could be learnt in terms of the already-learnt, primitive features.

The part-specified, part-learnt solution gave a performance of 73.262% correct.

† Some of these training experiments were carried out using the PDP Research Group simulation tools, described in EXPLORATIONS IN PARALLEL DISTRIBUTED PROCESSING: A Handbook of Models, Programs, and Exercises © 1987 by J. L. McClelland and D. E. Rumelhart.

† This relatively poor performance is due to the existence of multiple fonts in the character test set. The training set contained only one font.
classification when trained on just one example of each character (i.e. 32 training patterns†). When a solution was learnt by a fully interconnected MLP with randomly-initialised weights (and the same number of hidden units: 15), the performance was 85.562%. It is concluded that, for neuron-level operations, such as classification, a learnt solution out-performs a part-specified solution. The only advantage, it seems, of part-specifying at this low level is that the problem of local minima can be ‘avoided’. However, this does not mean to say that the designer’s specification will not place the network in a local minima – as actually happened.

The virtue of specification is more apparent at higher levels of complexity, at which learning abilities are more restricted. The neural implementation of histogram-based, local adaptive thresholding was completely reliant on specification. In more general terms, it is highly unlikely that unnormalised, grayscale images could be used to train an MLP to classify characters (see section 6.6.6 for a discussion of this); it is already difficult enough for an MLP to learn a good classification based on segmented binary images. What has been shown by the experiment described in this section is that – even at the primitive level – weights can be part-specified from a priori information, and ‘layered’ learning can then be applied. (See section 7.3 for an alternative to traditional learning methods, more suited to the framework of hierarchy.)

These experiments were performed on a test set of 1147 characters, drawn from the set of 170 numberplate images. Each character was located in the thresholded image (using the conventional location algorithm) and segmented into 6 rows and 5 columns. The proportion of each of these blocks that was black/white was used as input to the classifier as a continuous value between 0 and 1. Thus, the hierarchical specification, at the top level of abstraction, receives input through an interface of type ‘retina’ (see Appendix D). This datatype is defined as consisting of six ‘rows’, and a row is defined in terms of segments, each of which could be defined in terms of pixels but which, in this specification, was defined in terms of a synapse. This synapse represents the proportion of the block that is black/white.

† 32, not 36 (26 alpha plus 10 numeric), because some characters, such as Zero and Capital-O, are identical.
Features such as the presence of a particular row or column, or a diagonal feature, are defined ultimately in terms of these segments. Thus, a function that detects a horizontal row at the top of the retina reads in the retina and extracts from this the first two rows. The segments which make up these rows are then extracted and supplied, along synapses which are appropriately-weighted so as to recognise the presence of this row, to a neuron whose output will represent the presence of that feature. The function which recognises an ‘E’ may then be defined in terms of primitive features such as this row.

When unspecified MLP-type architectures are scaled up the learning time increases exponentially (see section 2.2.2). The use of specification places the model in an area within the search space which can be as precise as the designer cares to make it. Thus, this experiment again shows that hierarchical specification can be used to allow the scaling-up of neural systems and to prevent this exponential increase in training time. This method of constraining certain weights or connections has been used many times before, simply as a means of constraining the search space so that a solution is actually learnt. However, this constraint has not been undertaken within a formal framework which provides a method for deriving these weights/structure. This would not be possible without the application of hierarchy.
The relatively poor performance of these neural implementations compared to the decision tree classifier is due to the fact that the neural systems (owing mainly to compute-time limitations) were not trained on all the different fonts that exist in the test set; the decision tree was trained on these different fonts. When trained on the same data as was used for the development of the neural system, the decision tree gave a comparable level of performance (85.918%).

**Summary**

The framework of hierarchy has been applied to non-trivial image processing problems and implementations have been derived. These systems have been part-specified and part-learnt. The neural implementation of histogram-based local adaptive thresholding, in particular, demonstrates how an algorithm may be incorporated into a neural net, resulting in a scaling-up of the model and a widening of neural applications. The classification experiment has demonstrated the way in which specification can enable learning.

Given the availability of neural hardware, these implementations of common image processing operations will — in terms of speed — out-perform conventional implementations, which are fundamentally sequential. In addition to this, these implementations make extremely efficient use of hardware, owing to the high degree of specification in those parts of the system that can be specified. This results in very sparse networks, which are amenable to efficient simulation or realisation by dedicated neural hardware.

This work has attempted the recognition of a numberplate from a raw grayscale image by means of an unadulterated neural implementation. This is in marked contrast to traditional approaches which have performed most of the (pre-)processing using conventional techniques. The neural element in these systems has tended to consist of a simple classifier, tacked on the end of the conventional processing. It is in the preprocessing that most of the work is performed. It is virtually inconceivable that an MLP could be trained on unnormalised grayscale images in order to classify characters. The range in exposure, not only between different characters but even within the same character, is potentially extreme and renders the input data virtually without pattern. This
approach was briefly investigated. Two and three layer perceptrons, with various configurations of numbers of hidden units, were trained on (already-segmented) grayscale images. Under no conditions did the model converge to a solution. This, above all else, highlights the virtues of the framework of hierarchy. Its application to this ‘hard’ problem has achieved moderate results with an entirely neural implementation.

<table>
<thead>
<tr>
<th>test set</th>
<th>test data set (containing training data)</th>
<th>verification data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional</td>
<td>99.9%</td>
<td>85%</td>
</tr>
<tr>
<td>Specified Neural</td>
<td>unavailable due to compute-time limitations</td>
<td>73%</td>
</tr>
<tr>
<td>Unspecified Neural</td>
<td></td>
<td>85%</td>
</tr>
</tbody>
</table>

The above table summarises the results obtained. It shows that, when tested with a genuine verification data set (involving no data contained in the training set) the conventional and neural systems are of comparable performance. As previously stated, it was not possible to train the neural systems on multiple fonts owing to compute-time resource limitations. In summary, therefore, these results show a very good level of performance for a neural solution as compared with a more conventional technique.
Chapter Seven

Conclusions

This thesis claims that hierarchy is an essential concept for the understanding and application of neural systems. How was this conclusion arrived at? Is it valid? Are the supporting evidence and the experimental results sufficient to substantiate it? These questions are answered in this chapter.

1. A Brief Review

It was first observed, in Chapter One, that neural systems are capable of astoundingly complex function. When it was asked how this remarkable function emerges from the interaction of primitive neural hardware it was seen that there were no coherent principles by which this could be explained. Indeed, it was questioned whether such principles actually existed.

The field of artificial neural networks was then examined for clues to the direction in which a unifying neural theory might lie. In particular, it was argued that specification is a necessary complement to learning; empirical derivation of a system cannot succeed in the entire absence of specification. At present, however, neural specification is intuitive, not meaningful; no coherent method exists for describing a neural system. The relationship between specification and understanding was then explored: if the way in which a neural system realises a function can be described then it is understood - at that level of description. It was argued that lack of neural theory is stifling growth of the field and that the concentration on traditional models (MLP, Kohonen, Hopfield, etc) will probably not spawn unifying theory.

Chapter Three was concerned with deriving insights on neural theory from related disciplines. Various metaphors of neural computation were explored, in the hope that the theory of the metaphor would enrich neural theory. Connectionist expert systems were presented as a metaphor of primitive neural computation. Neurons perform an inferencing-style function, and MLPs in particular approximate the distribution of
uncertainty in MLP-structured expert systems. This provided understanding of neural systems at the primitive level of computation. At higher levels of computation it was suggested that it is possible to understand the distributed implementation of a high level function in terms of lower level functions. The object-oriented paradigm was also of relevance as it offered a natural method of modelling; it may be that the neural medium of computation is suited to employing a similar paradigm through which to model the world.

In Chapter Four, these metaphors were drawn together to formulate a framework within which neural computation could be understood. The basis of this framework is hierarchy and as such the framework offers a method for describing function as the appropriate interrelation of lower level function. This framework relates high level function, via an arbitrary number of intermediate levels of abstraction/implementation, to a realisation of that function in terms of neural hardware. It explains how, by the appropriate interaction of neural components, higher level function emerges.

Not content with merely academic propositions, a software tool was constructed which embodies the principles of the framework (Chapter Five). ANNECS demonstrates how hierarchical specifications, described in a fully distributed, object-oriented style can be used to understand a neural system. The compiled network can be understood at various levels of abstraction. Thus, it is possible to identify the role of each neuron, of each connection, of each cluster of neurons and of each group of connections in the system.

ANNECS showed that the framework could be applied. It was then important to determine the power of this method. This was assayed by applying the framework to the task of numberplate recognition. Chapter Six described these experiments and demonstrated several virtues of the method. It was seen that the method was extremely powerful in that it enabled the construction of part-specified, part-learnt solutions. Information that was ‘obvious’ to the designer, such as how a histogram should be formed, was incorporated in the specification. Information not apparent to the designer, such as how to select a threshold, was derived empirically. Using this method, it was shown how neural systems could be scaled-up. Perhaps most significantly, neural systems constructed by this method were not black boxes. It was possible to select any neuron or connection from a network of $10^6$ neurons and $10^7$ interconnections and
identify its role. These results demonstrated that the framework, far from being a purely theoretical approach, is useful in engineering terms.

What may be concluded from these observations?

2. Conclusion: Hierarchy is Foundational to Neural Theory

Three broad conclusions can be identified, each of which support the thesis that hierarchy is the basis of neural theory.

(i) A Framework of Hierarchy explains neural computation of complex function. It seems obvious that a complex system cannot be understood by consideration of its primitive components alone. The complexity is too much for human understanding. In order to understand such systems it is essential that function is abstracted to arbitrary levels of detail. This must be a step in the right direction with regard to neural systems. The case study has demonstrated the value of this approach in that a large neural system could be completely understood. This discards the ‘black-box’ syndrome traditionally accepted by neural researchers as a natural characteristic of neural systems. Neither is there any ‘black magic’ going on in such systems: the emergence of higher level function from lower level function can be explained according to the principles of the framework. Ultimately, a function at a high level of abstraction can be realised as the appropriate interconnection of primitive processing elements.

It has not been proved, however, that all neural systems can be understood in terms of this framework. It is conceivable that biological systems may not conform to these principles. However, this does seem highly unlikely. To some extent, as described in Chapter Two, hierarchy can be readily observed in brain structure both in modularity of neurons and groupings of connections. Another factor which implies hierarchy as the basis of biological neural systems is their method of generation (see section 4.4). A compact method of encoding is required to describe brain-sized systems within the DNA. A hierarchy of netlists provides such a compact encoding (see section 6.6.2 and Appendix B, and section 6.6.3 and Appendix C).

(ii) A Framework of Hierarchy allows the combination of specification and learning. This conclusion is of wider relevance than to the neural field alone. The traditional view
has been that conventional and neural models of computation are radically different in that one is specified and the other is empirically derived. This work prompts the conclusion that this is a false distinction: rather, conventional models have been primarily serial-based (and therefore unsuited to learning) whereas neural models are fundamentally distributed. The framework of hierarchy by which neural systems may be understood is essentially the same as that by which conventional distributed systems or object-oriented systems may be understood. The difference with neural systems is in the primitive components of the system: neurons and synapses, as opposed to ALUs and buses. Neural systems are not predominantly empirically derived. (Artificial neural nets are, though, and this probably explains their relative failure to achieve results.) Thus, applying the framework of hierarchy to neural systems allows the combination of specification and learning as methods of implementation. It might be that this medium of computation has been adopted by nature precisely because it integrates specification and learning. Whether or not this is so, the application of hierarchy to neural systems does allow a priori knowledge to be combined with information that is best derived empirically.

(iii) A Framework of Hierarchy facilitates the scaling-up of neural systems. The principles of the framework provide a meaningful method of constraining network topology such that learning can succeed (see section 6.6). By training locally (one layer at a time) it can be guaranteed that a solution will be arrived at, given the designer has chosen a sensible data representation and such a solution exists (see section 6.6.5). The framework may also perhaps be used to integrate various traditional neural models, to arbitrary levels of complexity. Indeed, it may be that a common understanding of the traditional neural models (MLP, Hamming, Kohonen, etc) may be found in the application of these principles.

3. Speculation: Hierarchy in Learning

Hitherto, ‘firm’ conclusions have been stated. Here, speculation regarding an area beyond the experimental scope of this work is indulged in. It is interesting to surmise what the full impact of hierarchy on learning might be.
Assume that, at an arbitrary level of abstraction, every function can be identified and its operation understood in terms of its implementation at a lower level of abstraction. (At the base level each function is a neuron and its operation is thus understood.) According to this thesis, learning consists of the formation of higher level functions (as opposed to merely optimising lower level functions) by the generation/modification of interconnections between objects at a particular level of abstraction. (In effect, of course, this consists of the modification of primitive synapse strengths.) If it is possible to identify those new connections that have interrelated functions at some level so as to realise a higher level function then learning can be understood within a hierarchical context. Low level functions, such as edge detection, are learnt first – in terms of primitive functions/neurons. These learnt functions are then identified so that their interrelation to form higher level functions such as boundary and segment detection can be identified, and so on, to ever higher levels of abstraction.

This understanding of learning is radically different from the traditional ‘flat’ methods of modifying weights. By understanding the hierarchy that is being formed during learning this structure can be used to interrelate the modification of weights. The hierarchy could indentify which weights should be modified in relation to which other weights. This relation has previously been enabled in an ‘unintelligent’ way by using the primitive network topology. For example, weight changes may be propagated back through layers in a multilayer perceptron; nearest neighbour connections are modified in Kohonen nets. Hierarchical learning, however, would relate the weight changes on two synapses at ‘opposite sides’ of a neural system – because they both belong to the same higher level connection. It must be stressed, however, that these thoughts are purely speculatory and do not support or detract from the substance of this thesis.

4. Directions for Future Research

Two exciting new areas of research arise out of this work. The first, hierarchical learning, has been sketched out in the previous section. The second is the unification of the traditional neural models within the framework of hierarchy. If it can be shown that a framework of hierarchy provides a common understanding of these models then the weight of evidence that hierarchy actually is the basis of neural theory will be greatly increased.
For a widespread application of this framework, including the integration of learning and specification, better support software is required. ANNECS is not robust enough for full scale systems development. Thus, another research area is the development of a software toolset which enables the application of hierarchy to neural systems.

What has been explored is a new and radically different approach to neural computation. The true power of hierarchical neural systems has yet to be demonstrated. Whether hierarchy is in fact the basis of neural theory will probably not be agreed for some time to come. However, several exciting and apparently-rewarding new avenues of research are opening up.
References


Intelligence, pp. 465-474 (August 1985).


Appendix A

Formal Presentation of Abstraction/Implementation

Define:

\[
P: \{\text{port}_{\text{object}}, \text{port}_{\text{ref}}\}
\]

\[
C: \{\text{connection}_{\text{source}}, \text{source}_{-\text{port}}\rightarrow \text{dest}_{-\text{port}}\}
\]

\[
O: \{O, P, C\} \cup \{\text{neuron}\}
\]

\[
f[P, t]: Z
\]

**O valid implementation:** (forall \( o_k \in O \)):

\[
((o_k = \text{neuron}) \lor (o_k \text{ valid implementation} \land (\text{there exists } P \in o_k) \land (\text{there exists } C \in o_k) \text{ s.t.}
\]

\[
(\text{forall } c_{a,b,c,d} \in C) :
\]

\[
((\text{there exists } p_{a,b} \in P) \land (\text{there exists } p_{c,d} \in P) \text{ s.t.}
\]

\[
f[p_{a,b}, t] = f[p_{c,d}, t]
\]

\[
)
\]

)
Addendum to Appendix B

The following appendix contains the neural specification of Local Adaptive Histogram-based Thresholding, as represented graphically within ANNECS. To understand how this specification describes a complete neural system an explanation is required. For example, consider the top level specification of thresholding, as shown below:

```
Here, a and b represent input and output ports in the interface to this function. c, d, e and f represent lower level functions, in terms of which this function is defined. c, d, e and f happen, in this instance, to be the same type of function, a 256x256 image threshold. This function type is not shown in the Appendix, but is defined in the same way as the above function, except that its interface ports are of type 256x256 image and it is defined in terms of 128x128 image threshold functions.

Each of the connections, for example g and h, represent data paths. Thus, the above function reads in a 512x512 image, separates this into its four constituent quadrants and passes these to c, d, e and f. When each of these 256x256 thresholding functions have processed this data, they produce 256x256 binary images which are reassembled into a 512x512 image at b.
```
Appendix B

Neural Specification of Local Adaptive Histogram-based Thresholding

Threshold 512x512 Image

Threshold 8x8 Patch

† These functions/datatypes are as defined in ANNECS. Not all are shown: for example, it is apparent that a 256x256 image threshold is defined similarly to a 512x512 threshold (above top).
Select Threshold

Form Histogram

Form Bin of Histogram from 8x8 Patch
Form Bin of Histogram from 2x2 Patch

Test A and B for Equality

Derive Threshold from Histogram
Get Optimum Threshold from Histogram

Select Bin

Form Feature

Combine Features to form Bin Goodness
One bin in this histogram - corresponding to the threshold - is active.

Each of these weights is set to generate an activity equal to the threshold to be applied.

- Get Grayscale of Bin
- Apply Threshold to Pixel
Addendum to Appendix C

The following appendix contains the neural specification of Edge Detection, as represented graphically within ANNECS. To understand how this specification describes a complete neural system an explanation is required. The top level function is not shown overleaf but is defined in the same manner as the top level function for neural thresholding, as shown in Appendix B. The functions shown overleaf implement primitive edge detection which is carried out for every pixel in the image. In the first function shown, detect edge, data is fed in through vertical and horizontal, and read out at ‘Edgeness’. These input and output points are ports within the interface to the function.

The second function shown overleaf implements functions Horizontal Edge and Vertical Edge in detect edge. This function reads in a line of pixels (either horizontal or vertical) and detects a black-to-white edge and a white-to-black edge and then combines the amount of edge of each of these types that is present. The function which actually detects a black-to-white edge is shown at the bottom of the page overleaf.

The function which combines the edge measure in the horizontal and vertical directions is not shown. It consists of one neuron with two equally weighted inputs.
Appendix C

Neural Specification of Edge Detection

Detect Edge in a Direction (Horizontal or Vertical)

Detect black-white edge
Addendum to Appendix D

The following appendix contains the neural specification of Character Classification, as represented graphically within ANNECS. To understand how this specification describes a complete neural system an explanation is required. The top level function, Read Character, assumes the character has been located and normalised and thus is fed in as a retina of pixels, of size six rows by five columns. Within Read Character, retina is passed to a separate function to recognise each possible character. Each of these functions then outputs a probability of that character being present and these probabilities are combined to give a complete table of probabilities for each character.

Each character is defined in terms of functions to recognise primitive features such as Recognise Column. Thus, the function to recognise an ‘E’ is defined in terms of functions to recognise Row 1, Row 3, Row 4, Row 6 and Column 1.
Appendix D

Neural Specification of Character Classification

Examples of the way in which datatypes are defined:
a row consists of five pixels; a pixel is represented by a synapse
Appendix D

1st half of row 1 (excitatory)

1st half of row 2 (inhibitory)

Recognise Half Row

Recognise Column

Transform Column-Ordered to Row-Ordered ‘Retina’ Datatype
Appendix E

‘C’ Implementation of Histogram-based Local Adaptive Thresholding ♦

local.c

/* Oliver Vellacott - 21/3/90 - local adaptive thresholding algorithm */

#include <stdio.h>
#include "local.h"

/******************************/

short write_image_ok(fname)

    char *fname;

    { 
        FILE *fp = fopen(fname, "w");

        if (!fp)
            return FALSE;
        else
            fwrite(out_image, 1, IMAGE_SIZE*IMAGE_SIZE, fp);
            return TRUE;
    }

/******************************/

short read_image_ok(fname)

    char *fname;

    { 
        FILE *fp = fopen(fname, "r");

        if (!fp)
            return FALSE;
        else
            fread(in_image, 1, IMAGE_SIZE*IMAGE_SIZE, fp);
            return TRUE;
    }

/******************************/

† This is a much-pruned version of the local adaptive thresholding algorithm developed as part of the conventional (non-neural) numberplate recognition system. It implements the same function as the hierarchical neural specification in Appendix B though, of course, in a different way.
void

clear_histogram()
{
    int i;

    for (i=0; i<DIVISIONS; i++)
        bars[i]=0;
}

/**********************************************************/

void

form_histogram(i, j)
{
    int i, j;

    int k, l;

    clear_histogram();

    for (k=i; k < i+PATCH_SIZE; k += SUBSAMPLE_SIZE)
        for (l=j; l < j+PATCH_SIZE; l += SUBSAMPLE_SIZE)
            bars[(int)in_image[k][l]/bin_size]++;
}

/**********************************************************/

int

common_goodness(low_peak, high_peak, trough)
{
    int low_peak, high_peak, trough;

    /* this computes a measure of the 'goodness' of the trough as a threshold
         - with respect to the two peaks */

    int width=trough-low_peak,
        height1=bars[low_peak]-bars[trough],
        height2=bars[high_peak]-bars[trough];

    if (height1 <= 0 || height2 <= 0)
        return 0;
    else
        return ((height1+width)*height2);
}

/**********************************************************/

int

find_next_physical_trough(next_to_trough, upwards)
{
    int next_to_trough;
    short upwards;

    /* find trough neighbouring next_to_trough in direction upwards;
            if trough is more than one bin wide - 'level' - return
            left of trough*/

int i, j, lowest=9999;

if (upwards) { /* look to right */
    for (i=next_to_trough; i<DIVISIONS; i++) {
        if (bars[i] < lowest) {
            lowest = bars[i];
            if (bars[i] > lowest) { /* must check that */
                if (bars[i] < lowest) {
                    return (i-1);
                }
                return i; /* reached edge of histogram */
            } else if (bars[i] > lowest) {
                return (i+1);
            }
        }
    }
    /* look to left */
    left_of_trough=0;
    for (i=next_to_trough; i>=0; i--) {
        if (bars[i] < lowest) {
            lowest = bars[i];
            if (bars[i] > lowest) {
                return (i+1);
            }
        }
    }
    /* must check that have found rightmost edge of trough */
    for (i=i-1; j=0; j--) {
        if (bars[j] > lowest) {
            left_of_trough+=j+1;
            return (i+1);
        }
    }
    if (bars[j] < lowest) {
        i-=1; j+=1;
        return (i+1);
    } else if (bars[j] > lowest) {
        i-=1; j+=1;
        return (i+1);
    }
    must check that
    have found rightmost edge of trough
    for (i=i-1; j=0; j--) {
        if (bars[j] > lowest) {
            left_of_trough+=j+1;
            return (i+1);
        }
    }
    if (bars[j] < lowest) {
        i-=1; j+=1;
        return (i+1);
    }
}
*/

int find_next_peak(next_to_trough, upwards)
{
    int i, highest=0;

    if (upwards) { /* look for peak to right */
        next_to_trough+=i;
        for (i=next_to_trough; i<DIVISIONS; i++) {
            if (i==DIVISIONS-1) { /* reached edge of histogram */
                return DIVISIONS-1;
            } else if (bars[i] >= highest) { /* are moving up a peak */
                highest = bars[i];
            } else if (bars[i] < highest) { /* peak is last 'bar' looked at */
                return (i+1);
            }
        }
    } else { /* look for peak to left */
        next_to_trough-=i;
        for (i=next_to_trough; i>=0; i--) {
            if (i) { /* reached edge of histogram */
                return i;
            } else if (bars[i] >= highest) { /* are moving up a peak */
                highest = bars[i];
            } else if (bars[i] < highest) { /* peak is last 'bar' looked at */
                return (i+1);
            }
        }
    }
    return 0;
}
int
find_best_lowest(low_peak, high_peak, *best_trough);
/* finds the trough between high_peak and low_peak with the best goodness */
{
  int goodness=-999, low_trough=0, temp_goodness, temp_low_peak;
  low_trough = find_next_physical_trough(low_peak, TRUE);
  while (low_trough < high_peak) {
    temp_goodness = common_goodness(low_peak, high_peak, low_trough);
    if (temp_goodness > goodness) {
      goodness = temp_goodness;
      *best_trough = low_trough;
    }
    temp_low_peak = find_next_peak(low_trough, TRUE);
    low_trough = find_next_physical_trough(temp_low_peak, TRUE);
  }
  return goodness;
}

int
find_best_middle(low_peak, high_peak, best_trough);
/* find best peak-trough-high_peak combination for low_peak <= peak < high_peak; return best threshold through *best_trough */
{
  int best_trough_perhaps, goodness=-999, temp_goodness, low_trough;
  while (low_peak < high_peak) {
    temp_goodness = find_best_lowest(low_peak, high_peak, &best_trough_perhaps);
    if (temp_goodness > goodness) {
      goodness = temp_goodness;
      *best_trough = best_trough_perhaps;
    }
    low_trough = find_next_physical_trough(low_peak, TRUE);
    low_peak = find_next_peak(low_trough, TRUE);
  }
  return goodness;
}

int
find_best_top(low_peak, high_peak, best_trough)
```c
int low_peak, high_peak, *best_trough;
/* find best low_peak-trough-peak combination for
   low_peak < peak \leq high_peak;
   return best threshold through *best_trough */

int best_trough_perhaps, goodness=-999, temp_goodness, high_trough;
while (low_peak < high_peak) { /* keep same leftwards peak and
   move through all peaks right of that */
   temp_goodness = find_best_middle(low_peak, high_peak,
       &best_trough_perhaps);
   if (temp_goodness > goodness) {
       goodness = temp_goodness;
       *best_trough = best_trough_perhaps;
   }
   high_trough = find_next_physical_trough(high_peak, FALSE);
   high_peak = find_next_peak(high_trough, FALSE);
}
return goodness;

byte select_threshold(i, j)
int i, j;
/* find a good threshold, based on the histogram for this patch */

int low_peak, high_peak, low_trough=1, high_trough=DIVISIONS, best_trough;

form_histogram(i, j);
low_peak = find_next_peak(low_trough, TRUE); /* find leftmost peak */
high_peak = find_next_peak(high_trough, FALSE); /* find rightmost peak */
if (low_peak >= high_peak-MIN_PEAK SEPARATION) { /* outermost peaks are v. close together, or there's only one peak */
    if (low_peak < 5) return 255; /* patch must be all black */
    else return 0; /* patch must be all white */
} else {
    find_best_top(low_peak, high_peak, &best_trough);
    /* find best peak1-trough-peak2 combination */
    return (byte)(((float)best_trough-0.5)*(float)bin_size);
}
}

void local_threshold()
/* threshold in_image to out_image */

int i, j, k, l, threshold;
for (i=0; i<IMAGE_SIZE; i+=PATCH_SIZE)
for (j=0; j<IMAGE_SIZE; j+=PATCH_SIZE) {
    threshold = select_threshold(i, j);
    for (k=i; k<i+PATCH_SIZE; k++)
    for (l=j; l<l+PATCH_SIZE; l++)
```
if (threshold < in_image[k][j])
    out_image[k][j] = 255;
else
    out_image[k][j] = 0;
}

/*----------------------------------------*/

void main(argc, argv)

    int argc;
    char **argv;
{

    if (argc < 3)
        printf("Not enough arguments : local inimage outimage0); 
    else 
        if (!read_image_ok(argv[1]))
            printf("local: failed to load image.0); 
        else 
            {
                printf("Thresholding %s.. argv[1]);
                local_threshold();
                if (!write_image_ok(argv[2]))
                    printf("local: failed to write image.0); 
            }

    /*----------------------------------------*/

local.h

#define TRUE 1
#define FALSE 0
#define IMAGE_SIZE 512
#define MAX_DIVISIONS 256
#define SUBSAMPLE_SIZE 1
#define DIVISIONS 32
#define MIN_PEAK_SEPARATION 1
#define PATCH_SIZE 8

typedef unsigned char byte;

byte in_image[IMAGE_SIZE][IMAGE_SIZE],
    out_image[IMAGE_SIZE][IMAGE_SIZE];

int left_of_trough,
    bin_size=MAX_DIVISIONS/DIVISIONS,
    bars[DIVISIONS];
Appendix F

Published Work


Vellacott, Oliver R., *ANNECS: A Neural NEtwork Compiler and Simulator*, International Joint Conference on Neural Nets 1991, July 8-12, Seattle, Vol II, p991

Vellacott, Oliver R., *Compilation of Neural Nets from High Level Specifications*, IEE Colloquium on Neural Networks: Design Techniques and Tools, March 1991, Savoy Place, London, p9/1-9/4
INTRODUCTION

Perhaps the most striking feature of the current state of the art in neural nets is the lack of unifying theory. A multitude of models and successful applications abound, each with its own piecemeal 'theory' explaining how it works. No overarching principles exist, however, for bringing together these diverse models and enabling a common understanding. This lack of theory was pointed out by von Neumann as far back as 1956 yet his paper seems as relevant today as it was then. More recently, Patricia Churchland has described her own search for neural theory, concluding that none is yet available, though much needed. Her book, *Neurophilosophy*, gives an excellent description of the role and requirements of a theory of neural networks.

To be more down to earth, we have no method for saying what the $i$th neuron in the $j$th layer of a multilayer perceptron actually does. Maybe it is not possible to answer this. Perhaps it is not possible to describe that neuron's role. Maybe there is no computational theory underlying neural cognition which explains what each neuron and each connection means. Maybe the only 'explanation' of neural processing is some principle such as *Neural Darwinism* which explains how a system evolves but not how it performs its function at any non-primitive level.

These objections seem unreasonable. Theory exists to explain the operation of most other areas of the universe: why should the field of neural nets be different? At the primitive level at least, it is possible to explain in computational terms how inputs are transformed to outputs. This can be understood in terms of connectivity, neuron states, and so forth. Understanding is at present limited to this level.

The Requirements for Neural Theory

Neural theory should provide a method for understanding how a network implements its function. Similarly, it should supply a method for explicitly constructing a network to implement any specified function.

Biological neural networks are not generated spontaneously. They are (initially) generated according to the genetic code. Many animals are born with certain abilities already existant. These functions do not arise by magic but are derived from a specification in the form of the genetic code.

It seems remarkable that this approach to the construction of artificial neural nets has been all but entirely ignored. The generation of initial network topology, upon which learning can build and upon which learning relies, is foundational to the formation of biological neural systems. A genuine theory of neural nets would provide a method by which artificial networks could be constructed from meaningful genetic specifications (note that these bear no relation to *genetic algorithms*).

With current models it is the network topology which enables learning to succeed. ART 'works' because connection patterns are precisely specified and highly constrained. Self-organisation relies on the nearest neighbour interconnects of Kohonen nets, et al. This topology is effectively a genetic constraint. The designer adopts a topology in a subjective manner, dependent on various factors such as number of inputs and outputs, likely number of features at each level of representation, and so on. This paper advocates making explicit the genetic (or predetermined) element in the construction of a neural system. This requires a method for generating an architecture which implements (to some arbitrary degree of precision) a described function.

Additionally, a neural theory would provide a unified explanation of learning. The learning problem may be viewed as the task of modifying each weight in relation to the modification of other relevant weights. Theory would explain what weights were 'relevant' to other weights and provide a method for relating the weight modifications such that convergence was achieved. Current neural models do not scale up for this reason. As the network size increases the search space increases exponentially and 'blind' learning breaks down. A neural theory would (a) enable the initial network topology to be constructed in a meaningful way such that the search space was dramatically reduced and (b) enable more constrained exploration of the search space.

Hierarchy

The key concept underlying the framework presented in this paper is *hierarchy*. This concept is missing from current analysis and construction of neural nets. At present, attention is focused almost exclusively at the primitive level. When faced with the questions: *What does a network mean?* and *How does a network compute its function?* current 'theory' is powerless to respond. It simply is not possible to understand a complex network in terms of individual primitive neurons and synapses. This
The only level that is common to all models constructed under the framework for neural theory is the primitive level, that which contains neurons and synapses. Though the framework is applicable to all neural models, the way in which these models form higher levels of representation is not constrained by the framework and is instead determined by the designer or (perhaps) the learning process. The hierarchy presented in this paper is not the same as modularity of networks, which has been described in previous work as hierarchy. The scenario where several subnets or modules produce results which form input to a ‘higher level’ module is not taken to be true hierarchy. Representations are only analysed above the primitive level in a very restricted sense and there is no hierarchy of data.

It has been argued that a multilevel representation, in addition to a method of interrelating levels, is essential to the understanding of neural systems. Thus, the next section presents a framework of hierarchy for understanding neural systems at arbitrary levels of abstraction. The subsequent section describes a software tool which embodies this framework.

THE FRAMEWORK

The Concept of Levels

What we are concerned with is representing functions and data, and performing transformations between representations. We require a framework which enables us to describe distributed functions and data at arbitrary levels of abstraction and which enables us to interrelate those levels. As discussed in the previous section, the idea of levels is crucial to this framework.

Abstraction contains the idea of capturing the essence of something described at a greater level of detail. It involves saying less about how something is done and more about what is done. Abstraction contains the concept of summarising (not modifying) some description from a more to a less concrete form. Implementation is the inverse operation to abstraction. It involves putting a description of a function into effect. It involves making a function more concrete, saying the same thing but in more detail, transforming what a function is into how it should be performed.

Levels for Neural Representations

Within our framework we shall define the neuron/synapse level to be the primitive level. It is conceivable that there are yet more primitive implementations of this level but, for the purposes of understanding neural systems, neurons and synapses may be treated as primitive representations. The contention of this paper is that a neural network is a realisation of functioning that
can be meaningfully described and understood at higher levels of abstraction. As already discussed in the introduction, that function cannot be understood at the primitive level alone. The framework must enable us to abstract and implement functioning (and communication) in a completely distributed manner. This is achieved by the use of three basic concepts (see Figure 1):

(i) a function — which transforms inputs to outputs in some way.

(ii) a connection — which provides a means of integrating functions.

(iii) an interface — by means of which a function communicates with other functions.

Fig 1. Concepts of the framework: Definition of a Function

Functions

Use of the term function can be misleading since our functions are not restricted to returning a single, or even composite, value. Instead, they are allowed to take many inputs and produce many outputs, simultaneously. Our use of the term is more closely allied to the idea of an object, as used in object-oriented models of computation. The difference here is that objects in these models are typically defined in terms of (sequential) imperative code, and thus cannot naturally respond to simultaneous inputs with simultaneous outputs. In this sense, our use of the concept function is closer to the way in which a distributed system is defined. Here, a distributed system (function) is defined in a completely distributed manner such that the distributed system (function) consists of the appropriate interconnection of lower level distributed systems (functions). This analogy is a better parallel of the inherent distribution in neural systems, though the valuable concepts in object-oriented modelling are not explicit.

Broadly speaking, a function at one level of abstraction is implemented at a lower level (and in a multiplicity of ways) by relating lower level functions in such a way that together they produce the desired behavior. This interrelation is performed by message passing between functions. Where messages come from and go to is defined by interconnecting functions to form the appropriate topology. This style of definition is more declarative than most classical techniques (e.g. imperative algorithms) for describing functions.

Connections

Just as levels of abstraction exist in representation of functions, so connections represent levels of abstraction in the representation of data. If the axons transmitting visual information from the eye to the visual cortex were to take random paths through the rest of the brain it would be very difficult indeed to deduce what was going on. In practice, however, these nerves are tightly grouped into a 'higher level' connection, the optic nerve. It makes sense to understand the role these axons play by grouping them together: the grouping transmits an 'image' (actually a combination of intensity values and primitive objects such as edges) to another module within the brain.

A connection of type 'image' may be defined in terms of more primitive types of connection. For example, an image may be defined as a row of columns; or as a column of rows; or as a row of columns of blocks; and so on. A row may be defined in terms of pixels, which may themselves be defined in terms of primitive synapses.

This hierarchy in connections is necessary to facilitate high level message passing. Though at implementation level an image is sent along, say, a million primitive paths, at the conceptual level an image is sent, period. This abstraction of data must go hand in hand with the abstraction of function.

Interfaces

Each function, at each level of abstraction, has a typed interface. This consists of one or more ports, of particular connection-types, at which input is received and from which output is sent. It is by means of this interface that each function communicates with the outside world. Thus, when a function is defined — by interconnecting lower level functions — these interconnections are made to/from individual ports on those functions, not directly to components of those functions. Thus, each function has no control over its role in defining higher level functions: all it 'knows about' and can do is to perform its own function, transforming inputs received at its interface to outputs which it
transmits via its interface. In this way, as in object-oriented and distributed models, functions are autonomous. This use of typed interfaces allows the definition of a function to be restricted to one level at a time.

Each non-primitive type of interface/connection is defined in terms of lower level types. Thus, each port in the interface of function X itself contains ports — of lower level types. Connections external to X must be of the same type as the port on X to which they connect. Internal connections, however, may connect to one of the port's lower level ports which represent the types in terms of which the port is defined. Using this latter method of connection enables the function to decompose a high level connection into its constituent types. Thus, for a function to perform edge detection on input received as type image it must first decompose this image type to pixel level. Composition of higher level connection types is achieved in the same manner.

Instances

If it is necessary to define several functions in terms of one common lower level function, an instance of that function is required. For example, functions to perform object detection and object classification might both be defined in terms of a function which detects edges at a particular point in an image. Instead of creating two instances of this edge-detection function it makes sense to use a common instance, in terms of which both higher level functions are defined. This is analogous to the concepts of class and instance in object-oriented modelling.

This capability permits compact implementation of higher level functions; two functions are not required to do the same thing. Most neurons, or clusters of neurons, will typically be components of more than one higher level function. Thus, the implementations of multiple high level functions — which consist ultimately of primitive interconnections between primitive processors — will normally be closely intertwined. In the same way, instances of connections may be created so that disparate functions may communicate via the same communication path. This, of course, may not make sense without the use of multiplexing, though such connection instances may be a feature of biological systems.

ANNECS: A NEURAL NETWORK COMPILER AND SIMULATOR

Introduction

ANNECS is a software tool which embodies the methodology for constructing neural nets proposed in this paper and has been described more fully elsewhere. It enables the formation — compilation — of a neural network from a hierarchical specification. It then enables learning of that net — simulation — by applying one of a number of learning algorithms. During compilation the high level information contained in the hierarchy of the specification is retained such that learning that occurs can be understood.

Basically, ANNECS enables the user to define functions in terms of appropriately interconnected lower level functions. The only primitive function is the neuron and the only primitive connection is the synapse, though the model upon which each of these is based can be selected by the user. Thus, all functions are defined, ultimately, in terms of neurons interconnected by synapses. The compilation component of ANNECS performs this translation between a high level, hierarchical specification and its functionally equivalent neural implementation.

The development of this software was undertaken to provide experimental support for the framework for neural theory advanced in this paper. Thus, ANNECS integrates genetic and empirical methods of construction, the compilation and simulation components, respectively. The key element which enables this to be carried out in a meaningful way is the presence of hierarchy. The experimental results obtained from this work — the development of ANNECS and its application to several image processing problems — endorse the methodology proposed in this paper. Within a framework of this nature neural architectures can be comprehended at arbitrary levels of representation.

Features of ANNECS

Visualisation. Within the framework, description of a neural architecture consists of a hierarchy of netlists. Written in language-form, a netlist can be fairly meaningless. Text is inherently sequential in the way in which it lies on the page, even if what it expresses is something fundamentally parallel. A netlist is above all a structure, and structures are perhaps best conceived visually. Thus, an essential feature of ANNECS is that it visualises specifications. Each component of a function is a real entity, continually existant in the target neural implementation, and thus it makes sense to have it represented by a real object at a particular place on the screen. This is not to say that the same specifications could not be described linguistically, but that the manner of specification lends itself to, and is best understood by means of, visual representation. ANNECS uses visualisation for the same reason that schematic capture tools use it. Each type of function and each type of interface is represented by a user-defined icon. This icon is used to capture function visually.

Specification. The specification of a neural system is made by the hierarchical description of functions. The specification of each of these neural functions is
made up of a netlist of lower level functions and interfaces. Thus, the specification process consists of:

(i) creating instances of interfaces
(ii) creating instances of functions
(iii) interconnecting these functions and interfaces in the appropriate manner so as to implement the desired function

Compilation: Formation by Specification. The term *Compilation* is usually applied in a computing context to mean: generation of machine code from a high level language. In the context of this paper, however, it means the generation of a neural architecture from a high level specification. All objects are defined ultimately in terms of just two primitives, the neuron and the synapse. Thus, the compilation method consists of recursively expanding each object into its constituent objects, until the definition consists of neurons and synapses only. Since clusters of neurons are embodiments of objects whose function is fully described within the specification, the functioning of the network may be understood.

Simulation: Formation by Learning. ANNECS enables the simulation of a compiled network according to one of a number of models. Thus the same initial architecture can be made to learn according to different models without changing the specification. The model for neurons and the model for synapses is selected separately.

**SUMMARY**

We have described a framework of hierarchy within which representations of neural functions and data may be transformed from level to level. Within this framework, the function of a neural system may be abstracted above the primitive level so that it may be understood at arbitrary higher conceptual levels. The framework is natural to neural systems in that representations are completely distributed at each level of abstraction. *Neural Compilation*, the process by which a hierarchical specification of a neural system is implemented, is facilitated by ANNECS, a software tool. What is significant about this framework is that it enables us to comprehend neural systems at arbitrary levels of abstraction. As we have discussed, this ability is essential for us to be able to understand the operation of any non-trivial system and should aid analysis of neural systems by raising representations above the primitive level.

**Acknowledgements**

My grateful thanks to Prof. P.B. Denyer, Dr. A.F. Murray and Dr. D. Renshaw for overseeing and guiding this work.

**References**


ANNECS: A Neural Network Compiler and Simulator

O.R. Vellacott

Department of Electrical Engineering,
University of Edinburgh,
Kings Buildings,
Edinburgh, UK EH9 3JL
email: oliver@uk.ac.ed.ee

Abstract

ANNECS is a software tool that compiles a high level, object-oriented specification to a functionally equivalent neural network. It does this by realising each object in the specification as a functionally equivalent cluster of neurons and synapses. All objects are defined ultimately in terms of just two primitives, the neuron and the synapse. Thus, the compilation method consists of recursively expanding each object into its constituent objects, until the definition consists of neurons and synapses only. Since clusters of neurons are embodiments of objects whose function is fully described within the specification, the functioning of the network may be completely understood. Moreover, since networks compiled in this way are functionally equivalent to their algorithmic specification, Computation Theory may be applied to these neural networks. An application which demonstrates these principles is described. This is a simple robot controller which picks up objects and drops them into holes as it moves around in a world containing stairs.

1. Motivation

ANNECS enables construction of an artificial neural net (ANN) from a high level specification. Perhaps the main virtue of ANNs is that they learn solutions which could not be specified by human designers. Therefore, why construct a neural net from a specification? One would think it would be easier to implement the specification on a von Neumann machine. That it is possible to construct an ANN that will implement any specification has already been shown.¹

The motivation behind ANNECS is twofold. Firstly, a net that has some built-in structure may have a better chance of learning and hence of deriving a complete solution. The designer may be able to sketch out an approximate strategy for solving a problem, upon which neural learning techniques can build.² It may be desirable to compile-in information regarding high level strategy and to leave the management of uncertainty and the adaptation to real-world data to neural learning techniques.³,⁴ The biological system itself is formed with some structure (specified by the genetic code) built in, before any learning takes place. Hence, the primary motivation for this approach is to enable construction of ANNs which incorporate high level a priori knowledge, so as to combine (i) what is known
by the designer and (ii) what can only be learnt from real-world data.

Secondly, this approach aids understanding of the realisation of high level functioning in ANNs. Using ANNECS, it is possible to implement various styles of specification and to observe the efficiency with which those styles utilise neural hardware. Preliminary results suggest that the object-oriented paradigm is the most natural framework within which to understand higher level neural processing. We may not be able to prove that any one style of specification maps into the best neural representation or, for that matter, the biological representation. We can, however, construct nets whose functioning is understood at all levels of abstraction. Thus, ANNECS enables the study of representation of high level processing in neural nets.

2. Specification

Specifications are expressed in an object-oriented style using graphical input in a manner similar to schematic capture. Each object resides in a library and is defined in terms of lower level objects, connected together in such a way as to produce the desired function. There is just one primitive object, the neuron. Thus, all objects are defined ultimately in terms of neurons. In a similar manner, datatypes are defined in terms of lower level datatypes. The one primitive datatype is the synapse.

A low level object is defined by connecting neurons together so as to realise the specification of that object's behavior. It is at this stage in the definition process that initial weights and thresholds are specified—which may then evolve during simulation, according to some learning algorithm. In addition to specifying the function of an object, by the way in which lower level objects are connected/related, the designer also defines a typed interface by which the object communicates with the outside world. For example, an object which performs an \texttt{if...then...else} function might be defined thus:

![Diagram of if...then...else object](image)

Figure 1. Definition of \texttt{if...then...else} object
Datatypes are formed by grouping together lower level datatypes. For example, an 8-bit representation of an integer may be defined by grouping eight synapses to form this higher level type. This may be an unnatural utilisation of neural hardware to represent numeric values, but serves to show how multiple synapses may be grouped together and thereafter treated as one, high level communication path of a particular type. When forming a communication path between two objects, the types of the source and destination interface slots must match the connection type. This is the only type-checking performed by the compiler.

ANNECS is supplied to the user with a comprehensive library of objects which perform functions ranging from control constructs to integer arithmetic. The user defines objects by use of the mouse, selecting predefined objects from the library and creating instantiations of these in a ‘definition area’ on the screen; he then relates those objects by connecting them with communication paths of appropriate types, specifying to which slot in each interface the connection joins. Associated with each object, he defines a macro. Using these macros, ANNECS generates textual descriptions of the specification with each object. Several macros may be associated with each object so as to enable generation of text in various languages: for example, C++ syntax, or Simula syntax.

3. Compilation

Compilation consists of recursively expanding each object into its constituent objects until the definition consists of neurons and synapses exclusively. Expanding objects until only primitive objects (neurons) exist is relatively easy. However, it is more complex to expand high level connections to their constituent synapses, and to determine how to expand these connections across interfaces. This task is achieved, as in the compilation process of other object-oriented languages, by the use of a cactus stack (a stack of stacks). Here, to compile object \( A \) we push its constituent objects \( B = \{ x : x \text{ constituent of } A \} \) onto the main stack (the trunk of the cactus). The definitions for the constituent objects of \( A \) \( C = \{ x : x \text{ is constituent of } B \} \) are then pushed onto stacks (the spines of the cactus) which grow outwards from the main stack. Connections between the constituents of the constituent objects of \( A \) (i.e. \( C \)), hitherto made via interfaces, are made direct. Thus, boundaries between constituents of \( A \) are removed and the spines shrink back to leave \( A \) defined, not in terms of its constituents \( (B) \), but in terms of its constituents' constituents \( (C) \). This process is repeated until no interfaces exist between objects. The definition then consists of a network of directly connected neurons.

Since the compiled net is ‘flat’ and consists of only neurons and synapses, the inherent structure with which the net was formed is not apparent. It is not possible, by observation of the net in isolation from its specification, to say which neurons cooperate to form which high level objects. Thus, it is only possible to understand the high level functioning of the net when viewed in relation to the specification.

The result of compilation is a netlist describing neurons (their initial thresholds) and their interconnection by synapses (their initial weights). This ‘flat’ net may then be simulated within ANNECS, in order to observe its behavior. Various learning algorithms can be applied to compiled nets during simulation,\(^7,8\) but their effectiveness has not yet been investigated. This investigation is now the primary aim of our work. Additionally, the neuron function may be globally specified at simulation time to be, e.g. perceptron-type threshold function, sigmoidal function, etc.
4. Application: Simple Robot Controller

In order to demonstrate the principles of ANNECS it is useful to consider an application. Our robot moves around in a world containing stairs, objects and holes. When it finds an object it should pick it up and carry it until it finds a hole, into which the object should be dropped. Every other time the robot meets a stair, it should climb it; when not due to climb a stair it should turn left instead. Our aim is to formulate a specification describing this behavior and have ANNECS realise this as a functionally equivalent net. This will enable us to understand the part played by each neuron in achieving the overall function of the net.

We have defined our robot controller as one high level object in order to observe its entirety. We could, of course, have split the functioning into smaller modules. Our robot controller is thus defined:

![Figure 2. Specification of Simple Robot Controller in ANNECS](image-url)
5. Implications and Conclusions

Using ANNECS we can guarantee to construct a neural net that implements any specification expressed in an object-oriented manner. It may be possible to apply learning to nets thus formed and thus this may offer a powerful means of combining construction-by-specification and construction-by-learning. That which may be specified by the designer is built into the net in the form of structure and initial weights and thresholds. That which is unknown by the designer—and may only be determined from real-world data—may be learnt, adding detail to the high level strategy imparted by the designer. This approach is biologically inspired and requires further investigation.

ANNECS thus implements a potential theory for understanding neural nets. It demonstrates how high level processing may be achieved by the structuring and weighting of neural interconnections and thresholds. It should enable Computation Theory to be
applied to neural processing, since nets formed by ANNECS are direct realisations of Effective Procedures. This should provide valuable insight on how the biological system represents, and reasons about, a world.

Acknowledgements

The author wishes to thank Prof P.B. Denyer, Dr A.F. Murray and Dr D. Renshaw for their support and helpful criticism.

References


Compilation of Neural Nets from High Level Specifications

Oliver R. Vellacott

This paper describes a software tool which compiles a high level, object oriented specification to a functionally equivalent neural net. It also explores the implications of this approach for the theory and construction of neural nets.

1. Specification of Function

The object oriented methodology\(^1\) was chosen as the most natural framework within which to specify neural systems for the following reasons. Each object in a definition is autonomous; each cluster of neurons in a net is autonomous. Each object in a model is thought of as a continuously executing process; each cluster of neurons that implements an object is continuously active. Objects communicate by message passing; clusters of neurons communicate by passing messages along multiple synapses. Objects are specified in terms of other objects and ultimately in terms of one or more primitive objects; each cluster of neurons may be perceived as interconnections of other clusters of neurons and ultimately as interconnections of primitive neurons.

Using the software tool, the specification of an object’s function is entered graphically by use of the mouse, in a manner similar to schematic capture. Each object is defined in terms of a relationship between other objects. This relationship is pictorially represented on the screen, with each object-type represented by an icon and each connection-type represented by a uniquely-patterned line. Thus, an object which performs an \textit{if...then...else} function might be defined:

![Diagram of if...then...else object](image)

In the same way that high level objects are defined, high level datatypes are also defined in terms of lower level datatypes. For example, a representation of an eight-bit integer may be formed by grouping eight synapses together and thereafter treating them as one, higher level connection. This may be an unnatural method of representing numeric values using neural hardware, but serves to show how high level message types may be created from one primitive message type. High level connections between objects will thus be compiled as multiple synaptic connections, passing direct from neuron to neuron. Using this graphical, hierarchically-structured method of specifying a neural net's function, the designer is able to clearly understand, at every level of abstraction, how each high level function is implemented in terms of lower level functions and ultimately in terms of primitive neural hardware.

\(^{1}\) Department of Electrical Engineering, University of Edinburgh, Kings Buildings, Edinburgh, UK. EH9 3JL email: oliver@uk.ac.ed.ee
2. Compilation of Specification

Compilation consists of recursively expanding each object into its constituent objects until the definition consists exclusively of neurons and synapses. Expanding objects until only primitive objects (neurons) exist is relatively easy. However, it is more complex to expand high level connections to their constituent synapses, and to determine how to expand these connections across interfaces. This task is achieved, as in the compilation process of other object oriented languages, by the use of a cactus stack (a stack of stacks). Here, to compile object $A$ we push its constituent objects ($B=\{x:x \text{ constituent of } A\}$) onto the main stack (the trunk of the cactus). The definitions for the constituent objects of $A$ ($C=\{x:x \text{ is constituent of } B\}$) are then pushed onto stacks (the spines of the cactus) which grow outwards from the main stack. Connections between the constituents of the constituent objects of $A$ (i.e. $C$), hitherto made via interfaces, are made direct. Thus, boundaries between constituents of $A$ are removed and the spines shrink back to leave $A$ defined, not in terms of its constituents ($B$), but in terms of its constituents' constituents ($C$). This process is repeated until no interfaces exist between objects. The definition then consists of a network of directly connected neurons.

3. Application : A Simple Robot Controller

A simple robot controller was built to illustrate the operation of the software tool. The robot moves around in a world containing stairs, objects and holes. When it finds an object it should pick it up and carry it until it finds a hole, into which the object should be dropped. Every other time the robot meets a stair, it should climb it; when not due to climb a stair it should turn left instead. A graphical representation of this specification is as follows:

![Figure 2. Definition of Simple Robot Controller](image-url)
This specification is compiled to the following net:

From the graphically-expressed specification the software automatically generates, in addition to this compiled net, a hierarchically-structured, textual description of the functioning of each object. This is achieved by the use of macros which the designer associates with each object.

4. Implications of this work

There are two major implications of this work. Firstly, we have a potentially powerful means of constructing neural nets, combining construction-by-specification and construction-by-learning. Secondly, we have a framework within which to understand how high level functions may be represented using neural hardware. These two areas are now considered.

4.1. Combining Genetic and Empirical methods of Construction

A primary motivation for the use of neural nets, as opposed to more conventional methods of computation, is that they learn. There are many problems whose solution is not amenable to specification by a human but which may be learnt by use of neural techniques.2,3 These solutions generally derive from problems which require adaptation to ‘fuzzy’, contradictory and often vast amounts of real-world data. What is amenable to specification, however, is high level strategy for solving a problem. This is usually difficult for an unstructured net to learn, but is what humans are generally very good at suggesting.

Thus, the combination of high level strategy (imparted by the designer) and adaptation to real-world, uncertain data (imparted by neural learning) should provide a very powerful means of constructing neural nets that must perform non-trivial tasks. That which cannot easily be specified is learnt and that which cannot easily be learnt is specified. It seems this is the method of construction used by the biological system: the brain is formed by a combination of specification, from the genetic code, and learning.

The work described in this paper has investigated how a-priori knowledge regarding a potential solution may be built-in to the structure of a net, in the form of initial weights and/or thresholds. It has not attempted to determine whether known neural learning techniques can be applied to nets constructed from a specification in the manner described. This latter investigation has now become the primary aim of our ongoing work.
4.2. Theory of Neural Nets: representation of high level functions

As yet, there exists no recognised theory for understanding how neural nets perform high level functions. Neural nets are often treated as black boxes with little concern as to how they achieve their function. A consequence of this is that it is then difficult to see how performance can be improved or modified in any particular direction and this may explain their relative failure to produce significant practical results. By observation of an apparently unstructured net it is very difficult to deduce any structure and hence to determine what part each neuron (or cluster of neurons) plays in the overall objective. This is true unless the network architecture is significantly constrained, e.g. to a multilayer perceptron. Even then, it is difficult to understand how the net achieves its function for all but the simplest tasks. This is a problem inherent in bottom up analysis. The structure present in a compiled net is not readily apparent by observation of 'flat' connection patterns, weights and thresholds.

If, however, we construct a net using a top down method of specification, it is possible to understand, at each level of abstraction, exactly how the net achieves its function. The role of each neuron and of each cluster of neurons may be understood. Thus it is possible to perceive how high level functions may be represented and computed by neural hardware. This observation may constitute part of a possible theory for neural nets. It offers a means of understanding how neural nets model a world, in terms of the object oriented paradigm. We cannot prove that this is the means used by the biological system to represent a world, except perhaps by interpreting how the genetic code specifies the brain structure, but it does seem a highly plausible explanation.

Neural nets constructed from object oriented specifications are direct realisations of those specifications. These specifications, however, are essentially Effective Procedures: they are unambiguous descriptions of functioning. Thus, by this route we can relate Computation Theory, in its entirety, to neural nets. In other words, the theory of what is and what is not computable applies equally to neural processing as to symbolic processing. This merely serves to confirm McCulloch and Pitts' earlier derivation of this result.

References