Task Decomposition Through Competition in a Modular Connectionist Architecture

Robert A. Jacobs

COINS Technical Report 90–44

May 1990

Department of Computer & Information Science

University of Massachusetts, Amherst, MA 01003

Task Decomposition Through Competition in a Modular Connectionist Architecture

Robert A. Jacobs

Department of Computer & Information Science

University of Massachusetts, Amherst, MA 01003

COINS Technical Report 90-44

May 1990

TASK DECOMPOSITION THROUGH COMPETITION IN A MODULAR CONNECTIONIST ARCHITECTURE

A Dissertation Presented by

ROBERT A. JACOBS

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1990

Department of Computer and Information Science

TASK DECOMPOSITION THROUGH COMPETITION IN A MODULAR CONNECTIONIST ARCHITECTURE

A Dissertation Presented by

ROBERT A. JACOBS

Approved as to style and content by:

Andrew G. Barto, Chair of Committee

Roderic A. Grupen, Member

Christopher V. Hollot, Member

Michael I. Jordan, Member

W. Richards Adrion, Department Chair
Computer and Information Science

ACKNOWLEDGEMENTS

The influence of Andrew Barto and Michael Jordan on my thinking is evidenced throughout this dissertation. They provided technical training, constructive criticism, guidance, and inspiration throughout my tenure as a graduate student. Andy helped me to discriminate between vital research topics and those of secondary import, and urged me to focus my energies on those issues that should be central to learning theory in the future. Andy also taught me to organize my thoughts and communicate them in a coherent manner. Mike showed me the ingredients that go into scientific research at the day to day level. By allowing me to observe his own activities, Mike attempted to teach me how to properly define a research question and how to evolve and refine one's thinking so that steady progress is made towards answering that question. Mike stressed the importance of translating my abstract, heuristic notions into concrete, mathematical statements. Furthermore, Mike was always willing to study my research at a detailed level, to commend those aspects that were of interest, and to challenge those aspects that required further thinking.

In addition to Andy and Mike, I wish to thank the other members of my committee, Roderic Grupen and Christopher Hollot. Their questions forced me to re-evaluate and clarify several points in this thesis.

This research has benefited from conversations with many people who have been willing to share their thoughts and suggestions. I am particularly grateful to Jonathan Bachrach, Kyle Cave, Vijaykumar Gullapalli, Geoffrey Hinton, Stephen Kosslyn, Michael Mozer, Steven Nowlan, and Richard Sutton.

Much of the research reported here was conducted during a nine month stay at the Massachusetts Institute of Technology where I was a visiting graduate student in the Department of Brain & Cognitive Sciences. I thank the members of that department for welcoming me into their community and for allowing me to use their facilities.

Portions of this thesis appeared in a paper that I co-authored with Michael Jordan and Andrew Barto (Jacobs, Jordan, and Barto [32]). Consequently, some of the thoughts and writing contained in this thesis result from a collaboration between myself, Jordan, and Barto. Some of the ideas in Section 3.5 were contributed by Steven Nowlan and Geoffrey Hinton who convinced me of the soundness of their arguments by conducting a number of computer simulations using a modified version of the modular architecture that I propose here. Charles Anderson, Judy Franklin, and Stephen Judd contributed LaTeXmacros that I used to format this thesis.

This research was supported by funding provided to Andrew Barto by the Air Force Office of Scientific Research, Bolling AFB, under Grant AFOSR-89-0526, and

the National Science Foundation under Grant ECS-8912623. My stay at MIT was supported in part by funding provided to Michael Jordan from the Siemens Corporation.

ABSTRACT

TASK DECOMPOSITION THROUGH COMPETITION IN A MODULAR CONNECTIONIST ARCHITECTURE

SEPTEMBER 1990

ROBERT A. JACOBS, B.A., UNIVERSITY OF PENNSYLVANIA

M.S., UNIVERSITY OF MASSACHUSETTS

PH.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Andrew G. Barto

A novel modular connectionist architecture is presented in which the networks composing the architecture compete to learn the training patterns. As a result of the competition, different networks learn different training patterns and, thus, learn to compute different functions. The architecture performs task decomposition in the sense that it learns to partition a task into two or more functionally independent

tasks and allocates distinct networks to learn each task. In addition, the architecture tends to allocate to each task the network whose topology is most appropriate to that task, and tends to allocate the same network to similar tasks and distinct networks to dissimilar tasks. Furthermore, it can be easily modified so as to learn to perform a family of tasks by using one network to learn a shared strategy that is used in all contexts along with other networks that learn modifications to this strategy that are applied in a context sensitive manner. These properties are demonstrated by training the architecture to perform object recognition and spatial localization from simulated retinal images, and to control a simulated robot arm to move a variety of payloads, each of a different mass, along a specified trajectory. Finally, it is noted that function decomposition is an underconstrained problem and, thus, different modular architectures may decompose a function in different ways. A desirable decomposition can be achieved if the architecture is suitably restricted in the types of functions that it can compute. Finding appropriate restrictions is possible through the application of domain knowledge. A strength of the modular architecture is that its structure is well–suited for incorporating domain knowledge.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT vii

LIST OF TABLES xii

LIST OF FIGURES xiii

CHAPTER

1 INTRODUCTION 1

1.1 Outline of the Thesis 6

2 ADVANTAGES OF MODULAR CONNECTIONIST ARCHITECTURES 8

2.1 Learning Speed 9

2.2 Generalization 13

2.3 Representation 14

2.4 Hardware Constraints 16

3 A Modular Connectionist Architecture 20

3.1 Output of the Architecture 20

3.2 Training of the Architecture 21

3.3 Discussion 25

3.4 Relationship to Previous Research 27

3.5 An Alternative Error Function 34

4 The What and Where Vision Tasks 42

4.1 Simulation Experiments 45

4.1.1 Temporal Crosstalk 46

4.1.2 Spatial Crosstalk 54

5 Multi–Payload Robotics Task 63

5.1 Training Procedure 64

5.2 Comparison of Architectures 70

5.3 Utilization of Domain Knowledge 86

6 Task Decomposition and Network Architectures 92

6.1 Simulations 93

6.2 Design of Modular Architectures 101

7 Conclusions 103

7.1 Future Work 104

Appendices

A Simulation Details for Chapter 4 110

B Simulation Details for Chapter 5 113

C Simulation Details for Chapter 6 116

References 118

LIST OF TABLES

4.1 Systems studied by Rueckl, Cave, and Kosslyn on the "what" and "where" tasks. 44

4.2 Systems used in the temporal crosstalk experiments. 48

4.3 The modular architecture used in the spatial crosstalk experiment. . . 59

5.1 Parameters of the robot arm. 69

5.2 f_i and g_i are the transformations of the robot arm's joint positions, velocities, and accelerations used in the input vectors of sets L and NL respectively. The subscripts on $\theta, \dot{\theta}$, and $\ddot{\theta}$ are the joint number. . 71

5.3 Single networks, modular architectures, and modular architectures with share networks used in the multi–payload robotics experiments. . . . 73

5.4 Three simulations that make use of a priori knowledge of the domain. 87

6.1 Systems used in the sine wave experiments. 95

A.1 Parameter values used in the temporal crosstalk experiments. 111

B.1 Parameter values used in the multi–payload robotics experiments. . . 115

LIST OF FIGURES

2.1 A: A network that is susceptible to spatial crosstalk. B: A modular architecture that is not susceptible to spatial crosstalk. 11

3.1 A modular connectionist architecture. 21

3.2 Multiplicative connection—The input to unit C is the product of the activations of units A and B. 31

3.3 Arrangement of networks used by the modular and cooperative architectures. 39

4.1 Learning curves for the 26 → 18 → 9 network on the "what" and "where" tasks using random and blocked training. 49

4.2 Learning curves for the 26 → 36 → 9 network on the "what" and "where" tasks using random and blocked training. 51

4.3 The modular architecture simulated in the temporal crosstalk experiments. 52

4.4 Learning curves for the modular architecture on the "what" and "where" tasks using random and blocked training. 55

4.5 A modular architecture with multiple gating networks ($y_1^{[1\cdots m]}$ denotes the vector whose components are the first m components of expert network 1's output vector, and the other expressions similarly denote subvectors). 57

5.1 A: During the performance stage, both a feedforward controller and feedback controller compute torques used to control the robot arm. B: During the training stage, the feedforward controller receives the actual joint variables. Its desired output is the torque applied to the arm. 67

5.2 Two–joint planar arm. 69

5.3 A modular architecture with a share network. 75

5.4 Learning curves for the systems trained with the input vectors from set L. 78

5.5 Allocation of modular architecture MA–L's expert networks to payloads. 79

5.6 Allocation of modular architecture with a share network MAS–L's expert networks to payloads. 80

5.7 Learning curves for the systems trained with the input vectors from set NL. 83

5.8 Allocation of modular architecture MA–NL's expert networks to payloads. 84

5.9 Allocation of modular architecture with a share network MAS–NL's expert networks to payloads. 85

5.10 Learning curves for the three simulations during the last stage of training. 88

6.1 Graph of f. 94

6.2 Task decomposition discovered by System 1. 96

6.3 Task decomposition discovered by System 2. 99

6.4 Task decomposition discovered by System 3. 100

CHAPTER 1

INTRODUCTION

Although many biologists and psychologists share the view that the brain has a modular architecture, there is no general agreement on the number of modules, the function of the modules, the nature of the interaction between modules, or the manner in which the modules develop. One reason for this diversity of opinion is that answering questions about the modular nature of the brain involves the difficult task of reasoning about a system with a large number of interacting components. Even systems of interacting components with a small fraction of the brain's complexity present formidable conceptual and analytical difficulties. In many cases, mathematical and computer models provide essential tools for understanding aspects of these systems. One class of models that has the potential for helping to answer questions about modular systems is the class of connectionist models, also known as artificial neural network models.

A hierarchical classification of the components of connectionist models may be defined in which a unit is the finest level of classification, a layer is a coarser level,

and a network is a still coarser level. Connectionist researchers typically design systems that are modular at the level of units or layers. In this thesis we argue that there are significant practical and theoretical advantages to be realized by considering modularity at the level of networks. In other words, we argue that connectionist architectures should consist of multiple networks, and that connectionist learning algorithms should be designed to take advantage of this modular structure.

Although terms such as *layer* or *network* are imprecise, it is generally agreed that they provide a convenient language for discussing connectionist architectures. An analogous situation occurs in the neurosciences where researchers debate whether nervous systems are best conceptualized at the level of the neuron or at coarser levels such as the column, hypercolumn, or area (Kaas [36]). This debate persists despite the lack of agreement on the precise definitions of the coarser taxonomic levels. For our purposes, we rely on the intuition that portions of an architecture larger than a unit or layer that learn and perform different functions constitute different networks.

This thesis introduces a novel modular connectionist architecture in which the networks composing the architecture compete to learn the training patterns. An outcome of this competition is that different networks learn different training patterns and, thus, learn to compute different functions. The architecture performs task decomposition in the sense that it learns to partition a task into two or more functionally independent tasks and allocates distinct networks to learn each task. In addition, it tends to allocate to each task the network whose topology is most appropriate to that task, and tends to allocate the same network to similar tasks and distinct networks

to dissimilar tasks. Furthermore, it can be easily modified so as to learn to perform a family of tasks by using one network to learn a shared strategy that is used in all contexts along with other networks that learn modifications to this strategy that are applied in a context sensitive manner. These properties are highlighted in this introductory chapter and detailed in the remainder of the thesis.

Task decomposition can take at least two different forms. A complex task may be decomposed into a set of simpler tasks that must be performed in sequential order (Sutton [68], Barto, Sutton, and Watkins [11]). Alternatively, a complex task may be decomposed into a set of simpler tasks such that different tasks need to be performed in different contexts. Mathematically, this is equivalent to saying that a complex function may be decomposed into a set of simpler functions such that different functions must be performed when the inputs come from different regions of the input space. Although we consider the first form of task decomposition to be very important, our study focuses on the second form.

If a complex task is sufficiently well understood then an investigator can use domain knowledge to decompose this task into a set of simpler tasks. In this case, an advantage of our modular architecture is that the decomposition can be embedded in the architecture by individually training distinct networks to perform distinct tasks. Unfortunately, such extensive domain knowledge is rarely available. Therefore, our research emphasizes the need for a connectionist system that learns to perform task decomposition. Specifically, our objective is to develop a connectionist architecture

that learns when it is required to perform two or more functionally independent tasks and allocates distinct networks to learn each task.

The first major property of our architecture is that, by allowing networks to compete to learn the training patterns, this objective can be achieved. Consider training an architecture that consists of two networks, possibly with different topologies, to perform the function

$$f(x) = \begin{cases} x & \text{if } x < 0 \\ \sin x & \text{if } x \geq 0. \end{cases} \tag{1.1}$$

At each time step, both networks receive an input x and each computes an output. The network whose output is closest to the desired output wins the opportunity to learn about the given training pattern. The losing network does not adjust its weights. Using a slightly more sophisticated competitive mechanism, our architecture allocates its networks such that, after many time steps, one network learns to compute $f(x) = x$ for $x < 0$ and the other network learns to compute $f(x) = \sin x$ for $x \geq 0$. Thus, the architecture is said to perform function decomposition in the sense that it learns that the training patterns are generated by two distinct functions and allocates different networks to learn each function.

Suppose that in the above example, the architecture consists of one single–layer linear network and two multi–layer nonlinear networks. As is discussed in Chapter 2, the architecture may achieve the best generalization if it allocates the linear network to learn the linear portion of f and one of the nonlinear networks to learn the nonlinear portion of f. The second major property of our modular architecture is that it tends

to allocate to each function the network whose topology is most appropriate to that function. Once again, this ability results from the competition among networks to learn the training patterns.

Above we assumed that when a complex task is decomposed into a set of simpler tasks, then a different network should learn each of the simpler tasks. However, this strategy fails to take advantage of beneficial generalization across tasks. Several researchers (e.g., Selfridge, Sutton, and Barto [66], Jacobs [31]) have noted that training a system to compute one task may facilitate the system's ability to learn a second task, a situation referred to as positive transfer of training, or it may retard the system's ability to learn a second task, a situation referred to as negative transfer of training. The third major property of our modular architecture is that, in many circumstances, similar tasks are learned by the same network, resulting in the benefits of positive transfer of training, whereas dissimilar tasks are learned by different networks, thereby avoiding the detrimental effects of negative transfer of training.

The last major property of our modular architecture concerns the assumption that a complex task should be decomposed into a set of distinct simpler tasks. Minsky [48] noted that a disadvantage of modular systems is that they hinder the sharing of knowledge across modules. This problem arises when a complex task is best approached by utilizing a shared strategy in all contexts plus modifications to this strategy that are context dependent.

For example, consider the task of controlling a robot arm to move a variety of payloads, each of a different mass, along a specified trajectory. One decomposition of this task is to create separate controllers for different mass categories (e.g., light, medium, or heavy payloads). However, a better decomposition is to utilize a shared controller that controls the arm when it contains no payload along with other controllers, one for each mass category. The shared controller provides torques to the arm at all times. The other controllers add extra torques in a context sensitive manner. Thus, one controller adds extra torques when the object is light and a second controller adds extra torques when the object is heavy.

As is demonstrated in Chapter 5, a slight modification to our modular architecture allows it to allocate one network to learn to be the shared controller and the other networks to learn to be the controllers for the different mass categories. Therefore, the fourth major property of our architecture is that it can learn to solve a task by learning a shared strategy that is used in all contexts along with a set of modifications to this strategy that are applied in a context sensitive manner.

1.1 Outline of the Thesis

This thesis is organized as follows. Chapter 2 lists several advantages that suitably designed modular connectionist architectures have over single networks. Chapter 3 presents the modular architecture that we have developed and details the equations that govern the architecture's behavior. Chapter 4 reports the results of training

single networks and modular architectures on the "what" and "where" vision tasks studied by Rueckl, Cave, and Kosslyn [62]. Chapter 5 reports the results of training single networks and modular architectures to control a simulated robot arm to move a variety of payloads, each of a different mass, along a specified trajectory. Chapter 6 discusses the relationship between the structure of a modular architecture and the task decomposition discovered by the architecture. The last chapter, Chapter 7, draws final conclusions and suggests extensions to the research described here.

CHAPTER 2

ADVANTAGES OF MODULAR CONNECTIONIST ARCHITECTURES

Theorists have shown that connectionist networks are universal approximators, meaning that for any given function there is a connectionist network capable of approximating it arbitrarily closely (e.g., Hornik, Stinchcombe, and White [30]). Given this result, what advantages might a modular architecture consisting of several connectionist networks have over a single network? In this chapter we answer this question by arguing that modular architectures have advantages in terms of learning speed, generalization capabilities, representation capabilities, and their ability to satisfy constraints imposed by hardware limitations. This discussion is not specific to the architecture that we have developed; it applies to the general class of multi-network systems of which ours is an example.

2.1 Learning Speed

Several characteristics of modular architectures suggest that they should learn faster than single connectionist networks. One such characteristic is that modular architectures can take advantage of *function decomposition*. If there is a natural way to decompose a complex function into a set of simpler functions, then a modular architecture should be able to learn the set of simpler functions faster than a single network can learn the undecomposed complex function. For example, consider the absolute value function

$$f(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0. \end{cases} \tag{2.1}$$

This nonlinear function can be learned by a single network with at least one layer of hidden units. Alternatively, it can be learned by a modular architecture consisting of two networks, each a single linear unit, and a mechanism for switching on the appropriate network in the appropriate context. One network can learn the function $f(x) = x$, and the other network can learn the function $f(x) = -x$. Assuming that it is relatively easy to learn the switching mechanism, the modular architecture should be able to learn faster than the single network because it does not use hidden units and each module only needs to learn a linear function.

In addition to being able to take advantage of function decomposition, modular architectures can be designed to reduce the presence of conflicting training information that tends to retard learning. We refer to conflicts in training information as

crosstalk and distinguish between spatial and temporal crosstalk. Spatial crosstalk occurs when the output units of a network provide conflicting error information to a hidden unit. Jordan [34] and Plaut and Hinton [57] noted that this occurs when the backpropagation algorithm is applied to a single network containing a hidden unit that projects to two or more output units. For example, suppose that a hidden unit projects via positive weights to two output units and that when compared to the desired output values, the output level of the first output unit is too small, whereas the output level of the second output unit is too large. Using the backpropagation algorithm, the first output unit provides derivative information specifying that the hidden unit should have a larger output. However, the second output unit provides derivative information specifying that this same hidden unit should have a smaller output. This conflict in derivative information is an instance of spatial crosstalk.[1]

Plaut and Hinton [57] noted that a modular architecture consisting of a separate network for each output unit is immune to spatial crosstalk. For example, consider the systems shown in Figure 2.1. Panel A shows a single network and Panel B shows a modular architecture consisting of three separate networks, one for each output unit. Although these systems can be applied to the same tasks, the modular architecture

[1]Although spatial crosstalk is clearly seen in terms of the backpropagation algorithm, it is not limited to networks trained using this algorithm. A network trained using any algorithm that approximates gradient descent (e.g., the A_{R-P} algorithm of Barto and Anandan [9] and Barto and Jordan [10]) can suffer from spatial crosstalk.

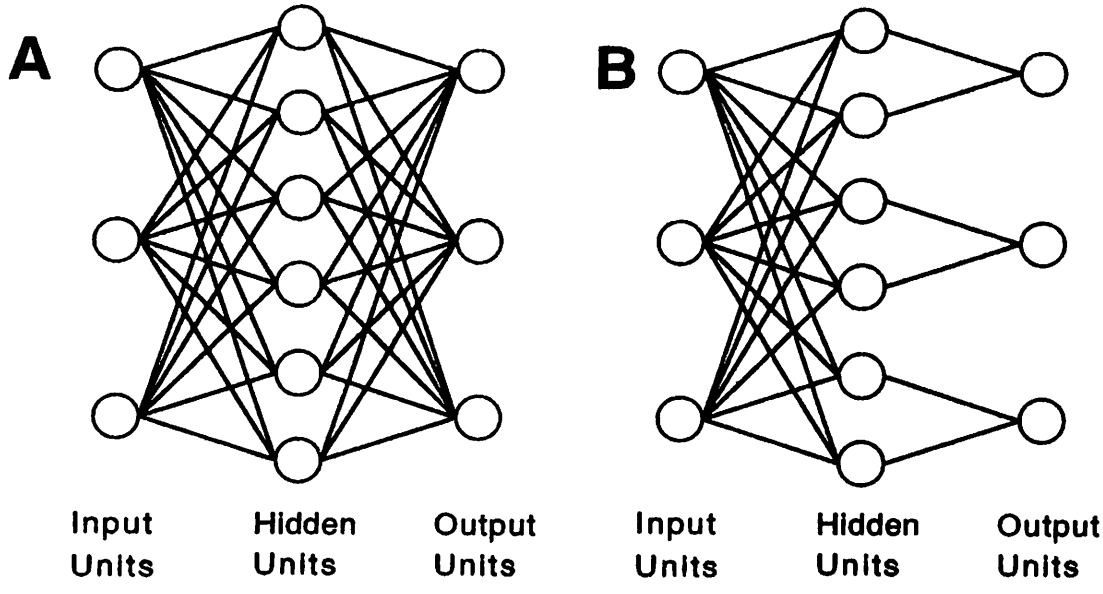

Figure 2.1: A: A network that is susceptible to spatial crosstalk. B: A modular architecture that is not susceptible to spatial crosstalk.

is immune to spatial crosstalk because each hidden unit projects to a single output unit.

In contrast to spatial crosstalk—where a unit receives inconsistent training information at a single instant in time—a unit might receive inconsistent training information at different times, a situation resulting in temporal crosstalk. One form of temporal crosstalk occurs when a network is trained to perform different functions at different times. For example, suppose that when a network is trained to perform one function, some of its hidden units become particularly useful in performing that function. When this same network is later trained to perform a second, different function,

one would like it to learn the second function without unnecessarily degrading its performance on the first function. This can be accomplished if the hidden units that participate in implementing the second function are not the ones that implement the first function. However, as Sutton [67] observed, gradient descent algorithms produce the opposite behavior because they preferentially modify the weights of the hidden units that are already useful. Therefore, after training on the second function, the network's performance on the first function will tend to be significantly degraded.

Temporal crosstalk arises in another situation closely related to that just described: when training patterns come from one region of the input space for many consecutive trials and later come from a different region for many consecutive trials. [2] If the function has different properties in these different regions, the network would receive conflicting training information. Training with patterns in the second region of the input space can result in degradation of the network's ability to perform correctly for patterns in the first region.

The conflicts in training information producing spatial and temporal crosstalk can be thought of in terms of transfer of training. Training a system to compute one function may facilitate the system's ability to learn a second function, a situation referred

[2] Although this is not how one would typically prefer to train a network, it may be unavoidable. For example, in training a network to control a dynamical system, the state of the dynamical system is likely to change slowly relative to the rate at which the system's variables are observed. Hence, the network is likely to receive inputs from the same region of the input space for many consecutive time steps.

to as positive transfer of training, or it may retard the system's ability to learn a second function, a situation referred to as negative transfer of training. Suitably designed modular architectures should learn faster than single networks because similar functions can be learned by the same network of the modular architecture, resulting in the benefits of positive transfer of training, whereas dissimilar functions can be learned by different networks, thereby avoiding the detrimental effects of negative transfer of training.

2.2 Generalization

Unless the structure of a network is well–matched to the function on which it is trained, it is unreasonable to expect the network to generalize well (e.g., Denker et al. [16]). For example, although both a single–layer linear network and a multi–layer nonlinear network can accurately learn a set of training pairs generated by a linear function, one would expect the single–layer network to generalize better because its structure is closer to the structure of the function being learned. If a modular architecture were able to decompose a complex function into a set of simpler functions and allocate an appropriately structured network to each simpler function, then one would expect good generalization. In this case, the mechanism responsible for allocating training patterns to the networks of the modular architecture would be automating part of the process of matching network structures to tasks, the other part of the process being played by the initial choice of the modular architecture's repertoire of networks.

A second reason that a modular architecture should generalize better than a single network involves the difference between local and global generalization. Global generalization occurs when the learning of a training pattern from one region of the input space influences the network's performance on patterns from a much wider region of the input space. As suggested above in the discussion of temporal crosstalk, when the function generating the training patterns possesses different characteristics in different regions, global generalization is an undesirable property due to negative transfer of training. In contrast, modular architectures perform local generalization in the sense that each network of the architecture only learns patterns from a limited region of the input space. Therefore, training a modular architecture on a training pattern from one of these regions does not affect the architecture's performance on patterns from the other regions.

2.3 Representation

Modular architectures tend to develop representations that are more easily interpreted than the representations developed by single networks. By this we mean that it tends to be easier to understand how a modular architecture implements a function than it is to understand how a single network implements the same function. This property was demonstrated by Rueckl et al. [62] who trained two connectionist systems to perform object recognition and spatial localization from simulated retinal images. As a result of learning, the hidden units of the system using separate networks

for the recognition and localization tasks contribute to the solutions of these tasks in more understandable ways than the hidden units of the single network applied to both tasks. In the former system, a different set of hidden units is used to represent information about the different tasks. In the latter system, on the other hand, the same set of hidden units is forced to represent information about both tasks despite the fact that the two tasks are relatively independent.

In addition to making it easier for experimenters to understand their systems, interpretable representations may make it easier for one portion of a system to understand the tasks performed by other portions. Minsky [48] argued that if one module can determine the task that is performed by a second module, then the former module can request the latter module to perform that task in the appropriate situation. This view emphasizes that the functions learned by the modules can be thought of as "building blocks" to be used on other occasions in the performance of more complex tasks.

Another advantage of using different networks to perform different portions of a task is that this representation facilitates the use of attentional mechanisms. Cowey [14] speculated that properties of a visual image may be attended to or unattended to through the operation of a relatively coarse mechanism that enhances or suppresses the activations of entire populations of processing units. Only if different modules of an architecture represent different portions of a task would we generally expect such mechanisms to have coherent and sensible effects.

Other reasons that modular architectures are superior to single networks in their representational capabilities are related to familiar arguments advanced in the computer science and psychology literatures: because modular architectures localize functions and develop more interpretable representations, they are easier for researchers to debug; because much of human knowledge is modularized, it is easier for researchers to embed domain knowledge into a connectionist system when the system is organized in a modular fashion; and because modules can be added one at a time, modular architectures facilitate the development of connectionist systems in an incremental manner.

2.4 Hardware Constraints

Another property of modular architectures suggesting advantages over single networks is that modular architectures can more closely satisfy several types of constraints imposed at the level of hardware implementation. In particular, suitably designed modular architectures can reduce the number of units and the lengths of connections.

In a discussion of representations employed by the brain, Ballard [5] suggested that a limitation on the number of neurons compels the brain to adopt a modular architecture. He hypothesized that the brain uses a coarse–code (Albus [1], Hinton [28]) to represent multi–dimensional spaces. Using this kind of representation, the number of neurons required to represent a space is $\frac{N^k}{D^{k-1}}$, where k is the dimension of the space,

N is the number of just–noticeable differences in each dimension, and D is the diameter of the receptive field of each neuron (Hinton [28], Ballard [5]). Because this rapid growth in the number of neurons limits the number of dimensions that can be represented in a cortical area, high–dimensional spaces must be represented in such a way that different dimensions are represented in different areas. Because different areas represent different dimensions, these areas must compute different functions. Analogously, in order to reduce the number of units required by a connectionist system, one may distribute the representation of multi–dimensional spaces among multiple networks.

Other speculations as to why the brain contains multiple cortical areas are also relevant to connectionist systems. Cowey [13] suggested that if the cortex uses lateral inhibition to sharpen various visual attributes, such as edges, orientation, color, disparity, spatial frequency, size, and movement, then a retinotopic representation of these attributes allows the use of relatively short connections between neurons since this representation places the interneurons necessary for receptive–field tuning relatively close together (Kohonen [39], Durbin and Mitchison [18]). If neurons for all the different attributes are represented in a single retinotopic map, however, the connections needed to sharpen the tuning of individual neurons for one of these attributes would be unnecessarily long. The existence of multiple retinotopic maps, each in a separate cortical area, allows the cortex to highlight several attributes using short local connections.

Similarly, Barlow [7] suggested that in order to detect various visual attributes using neurons whose connections are of minimal length, the cortex must employ multiple representations. He argued that because the brain primarily contains local connections, it is difficult, using a retinotopic representation, to detect similarities among non-contiguous locations of the visual field. In order to detect such similarities, it is necessary to map the information in the retinal image so that similar events are represented close to each other independently of the retinal coordinates of the events. For example, using a retinotopic representation and neurons with local connections, it is difficult to detect the co-linearity of line segments located at different places in the retinal image. However, in a different, non-retinotopic representation (e.g., the Hough transform (Duda and Hart [17], Ballard [4])) nearly co-linear line segments can be represented by neighboring neurons. In general, the existence of multiple representations, each in a separate area, allows the brain to detect several attributes using short connections.

The hypotheses of Ballard [5], Cowey [13], and Barlow [7] outlined above about why the cortex employs multiple representations located in different cortical areas are also relevant to connectionist architectures. These hypotheses suggest that in order to evaluate several attributes using units with local connections, connectionist systems should employ multiple representations located in different networks.

This completes our discussion of the advantages of modular architectures in relation to single networks. Based on this discussion, one might be tempted to conclude that it is best to use a modular architecture consisting of networks that each compute

a highly restricted domain-specific function. However, as noted by Fodor [22], certain portions of the human mind appear to be modular (for our purposes, we can take Fodor to mean that they compute domain-specific functions) whereas other portions seem to be non-modular. This suggests that there may be advantages to a connectionist architecture that includes both networks that compute fairly restricted domain-specific functions and networks that compute more general domain-independent functions. In addition, Marr [43] emphasized that modularity (in terms of computing domain-specific versus domain-independent functions) is a continuous and not a binary attribute. To this, we add that a network that learns a function in the context of one domain may later be found useful in other domains. Thus, a network that at first is thought to compute a domain-specific function may, due to transfer of training, later be thought to compute a domain-independent function.

CHAPTER 3

A MODULAR CONNECTIONIST ARCHITECTURE

In this chapter we introduce a modular connectionist architecture that learns to partition a task into two or more functionally independent tasks and allocates distinct networks to learn each task.

3.1 Output of the Architecture

The architecture illustrated in Figure 3.1 consists of two types of networks: *expert networks* and a *gating network*. The expert networks compete to learn the training patterns, and the gating network mediates this competition. After training, expert networks 1 and 2 compute different functions that are useful in different regions of the input space. Let the vectors $\mathbf{y}_1$ and $\mathbf{y}_2$ denote the outputs of the two expert networks. The gating network is an administrative agency that decides whether expert network 1 or 2 is currently applicable. Let the scalars g_1 and g_2 denote the two output units of

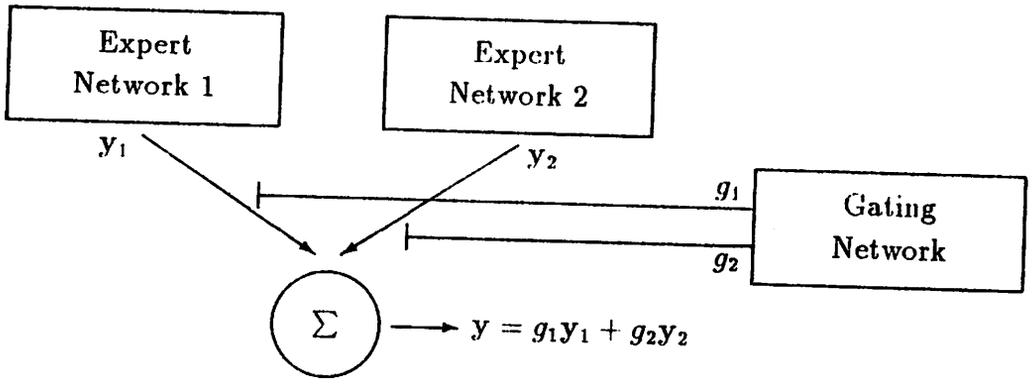

Figure 3.1: A modular connectionist architecture.

the gating network. The output of the entire architecture, $\mathbf{y}$, is $g_1\mathbf{y}_1 + g_2\mathbf{y}_2$. Therefore, when $g_1 = 1$ and $g_2 = 0$, expert network 1 determines the output of the architecture, whereas when $g_1 = 0$ and $g_2 = 1$, expert network 2 determines the output. In general, the architecture may contain any number of expert networks. If there are n expert networks, then the gating network has n output units, and the architecture's output is

$$\mathbf{y} = \sum_{i=1}^{n} g_i\mathbf{y}_i. \tag{3.1}$$

3.2 Training of the Architecture

During training, the weights of all networks are modified simultaneously using the backpropagation algorithm. However, the learning rules used to train the expert and gating networks are based on the minimization of different error functions. At each

time step, the weights of the expert networks are modified so as to reduce the sum of squared error between the output of the system, $\mathbf{y}$, and the desired output, $\mathbf{y}^*$. This error, denoted $J_{\mathbf{y}}$, is given by

$$J_{\mathbf{y}} = \frac{1}{2}(\mathbf{y}^* - \mathbf{y})^T(\mathbf{y}^* - \mathbf{y}). \tag{3.2}$$

The weights of the gating network are modified so as to reduce a more complicated error function. The intuition behind this function is as follows. For each training pattern, one expert network comes closer to producing the desired output than the other expert networks. In the competition among networks, this one is called the 'winner' and all others are called 'losers'. If on a given training pattern, the system's performance is significantly better than it has been in the past, then the weights of the gating network are adjusted to make the output corresponding to the winning expert network increase towards one and the outputs corresponding to the losing expert networks decrease towards zero. Alternatively, if the system's performance has not improved, then the gating network's weights are adjusted to move all of its outputs towards some neutral value.

This intuition is expressed mathematically as follows. First, it is necessary to specify what it means for the system's performance to be significantly better than it was in the past. If t is the current time step, then the error $J_{\mathbf{y}}(t)$ is a measure of the current performance. We measure the system's past performance by forming

an exponentially weighted average of $J_{\mathbf{y}}$ over time steps earlier than t. This value, denoted $\overline{J_{\mathbf{y}}}$, is computed iteratively by the following difference equation:

$$\overline{J_{\mathbf{y}}}(t) = \alpha J_{\mathbf{y}}(t) + (1 - \alpha)\overline{J_{\mathbf{y}}}(t-1), \tag{3.3}$$

where α, $0 \leq \alpha < 1$, determines how rapidly past values of $J_{\mathbf{y}}$ are forgotten. We use binary variables λ_{WTA} (WTA stands for 'winner–take–all') and λ_{NT} (NT stands for 'neutral') to indicate whether the system's performance has significantly improved. Specifically,

$$\begin{aligned} \text{If} \qquad & J_{\mathbf{y}}(t) < \gamma\overline{J_{\mathbf{y}}}(t-1), \\ & \text{Then } \lambda_{WTA} = 1 \text{ and } \lambda_{NT} = 0 \\ & \text{Else } \lambda_{WTA} = 0 \text{ and } \lambda_{NT} = 1, \end{aligned} \tag{3.4}$$

where γ is a multiplicative factor that determines how much less the current error must be than the measure of past errors in order for the system's performance to be considered significantly improved.

If the architecture's performance has significantly improved ($\lambda_{WTA} = 1$), we determine which expert network's output is closest to the desired output. Define the error for expert network i to be the sum of squared error between the expert network's output, $\mathbf{y}_i$, and the desired output, $\mathbf{y}^*$. This value, denoted $J_{\mathbf{y}_i}$, is

$$J_{\mathbf{y}_i} = \frac{1}{2}(\mathbf{y}^* - \mathbf{y}_i)^T(\mathbf{y}^* - \mathbf{y}_i). \tag{3.5}$$

The winning expert network is the network with the smallest error. If expert network i is the winner, then the desired value of the i^{th} output unit of the gating network, denoted g_i^*, is set to one. Otherwise, if expert network i is a loser, g_i^* is set to zero. If the architecture's performance has not significantly improved ($\lambda_{NT} = 1$), then the weights of the gating network are adjusted so that all the outputs of the gating network are moved towards a neutral value. This value is $\frac{1}{n}$, where n is the number of expert networks.

Using the quantities defined above, it is possible to write the gating network's error function, J_G, as:

$$\begin{aligned} J_G &= \lambda_{WTA}\frac{1}{2}\sum_{i=1}^{n}(g_i^* - g_i)^2 + \\ &\lambda_{WTA}\frac{1}{2}(1 - \sum_{i=1}^{n} g_i)^2 + \\ &\lambda_{WTA}\sum_{i=1}^{n} g_i(1 - g_i) + \\ &\lambda_{NT}\frac{1}{2}\sum_{i=1}^{n}(\frac{1}{n} - g_i)^2. \end{aligned} \tag{3.6}$$

Due to the definition of λ_{WTA} and λ_{NT} (Equation 3.4), the first three terms of Equation 3.6 contribute to the error when the architecture's performance has significantly improved, whereas only the fourth term contributes to the error when the system's performance has not significantly improved. The first term is the sum of squared error between the desired outputs and the actual outputs of the gating network. The

second term takes its smallest value when the outputs of the gating network sum to one. The third term takes its smallest value when the outputs of the gating network are binary valued. Therefore, the effect of changing the gating network's weights to reduce the second and third terms is that, in response to each input pattern, one output unit approaches one and all others approach zero. The fourth term is the sum of squared error between the neutral value and the actual outputs of the gating network. Reducing this term, which occurs only when the architecture's performance has not significantly improved, causes all of the outputs of the gating network to approach the neutral value $\frac{1}{n}$.

3.3 Discussion

The equations given above imply that there are three types of interactions among the networks of the modular architecture. The first type of interaction is that the gating network determines how much each expert network contributes to the output of the system (Equation 3.1); the second is that the performances of the expert networks determine the desired outputs of the gating network; and the third is that the gating network determines how much each expert network learns about each training pattern.

Referring to Figure 3.1, the error vector backpropagated into expert network 1 is $g_1(\mathbf{y}^* - \mathbf{y})$, and the error vector backpropagated into expert network 2 is $g_2(\mathbf{y}^* - \mathbf{y})$. This means that, in addition to determining how much each expert network contributes to the output of the architecture, the gating network also determines the

magnitudes of the expert networks' error vectors, and therefore determines how much each expert network learns about each training pattern. This interaction between the expert networks and the gating network implements a kind of credit assignment policy whose ramifications can be clarified by the following two examples.

Suppose that the gating network responds to an input pattern with $g_1 = 1$ and $g_2 = 0$. This implies that the output of the architecture, $\mathbf{y}$, is the output of expert network 1. In this case, the error vector backpropagated to expert network 1 is $\mathbf{y}^* - \mathbf{y}$, and the error vector backpropagated to expert network 2 is the zero vector. Thus, the first expert network is the one that learns about the function that generated the current training pattern, and the second expert network does not adjust its weights at all. This assignment of credit is logical because expert network 1 is solely responsible for the output of the architecture.

As a second example, suppose that during training of the architecture, expert network 1 is more closely approximating a given training pattern than is expert network 2, and g_1 is slightly larger than g_2. In this case, the first expert network receives a larger error and learns more about the function that generated the training pattern than the second expert network. Consequently, expert network 1 learns to perform this function even better than expert network 2, which causes g_1 to grow even larger than g_2. Thus, this credit assignment policy produces a positive feedback effect in the sense that it enhances the performance advantage of the expert network that is already most closely approximating the current target vector.

This credit assignment policy causes the modular architecture to allocate different expert networks to different tasks. Due to the positive feedback effect, one expert network learns the training patterns that compose a task. However, when later presented with the patterns that compose a second task, the network that won the competition to learn the patterns from the first task is unlikely to also win the competition to learn the new training patterns (unless the two tasks are very similar). Therefore, a different expert network wins the competition to learn the new training patterns. A consequence of the competition to learn the training patterns is that different expert networks learn to perform different tasks.

3.4 Relationship to Previous Research

Competitive Learning—There are many similarities between the competitive learning performed by conventional competitive learning systems (e.g., Kohonen [39], Rumelhart and Zipser [64], Grossberg [25], Reggia [61], Durbin and Willshaw [19], Yuille and Grzywacz [73]) and that performed by the modular architecture. Both types of systems utilize a competition among its components. In conventional competitive learning systems, the competition is between the units of a single network. In the modular architecture, the competition is between the different expert networks. Both types of systems have a method for selecting the winner of the competition. In conventional competitive learning systems, the competition is unsupervised. The winner of the competition at each time step is the unit whose weight vector most

closely matches the input vector. In the modular architecture, the competition is based on supervised errors. The winner of the competition at each time step is the expert network whose output vector most closely matches the desired output vector (Equation 3.5). Both types of systems utilize a competition to learn to respond to the input patterns. In conventional competitive learning systems, the units of a network compete for the right to respond maximally to each input pattern. In the modular architecture, the expert networks compete for the right to learn to produce the desired output pattern for each input pattern. Both types of systems attempt to discover natural groupings of patterns. A goal of conventional competitive learning systems is to cluster the input patterns into natural groupings. Different units learn to respond maximally to input patterns from different groupings. A goal of the modular architecture is to cluster the training patterns, which are input patterns together with desired output patterns, into natural groupings. Different expert networks learn training patterns from different groupings.

Despite the similarities between conventional competitive learning systems and the modular architecture, analogies between these two types of systems are imperfect. Such analogies fail for both implementational and theoretical reasons. At the implementational level, the training of the expert networks is closer to "soft" competitive learning than conventional winner-take-all or "hard" competitive learning. [1]

[1] Whereas "hard" competitive learning processes modify the parameters of only the winning competitor at each time step, "soft" competitive learning processes modify the parameters of all competitors in proportion to how well each competitor did in the competition for the current input

As discussed above, all expert networks, not just the winning expert network, modify their weights at each time step. The magnitude of an expert network's error vector is proportional to the output of the gating network corresponding to that expert network. This output is, in turn, related to how the expert network has fared in the competition for the current training pattern. Consequently, each expert network modifies its weights in proportion to how well it has fared in the competition for the current training pattern. The training of the gating network is also not done in a conventional winner–take–all manner. As explained above, the modification of the weights of the gating network is based on the selection of the winning expert network as well as on whether the architecture's performance has significantly improved.

At the theoretical level, analogies between conventional competitive learning systems and the modular architecture are also inexact. The clustering of input patterns discovered by many competitive learning systems can be characterized as having minimal within–group variance and maximal between–group variance. Such a characterization is possible because of the existence of metrics that allow one to compute the "nearness" or "similarity" of input patterns. In contrast, the clustering of training patterns discovered by the modular architecture cannot be similarly characterized. Such a characterization is not possible because of the difficulty of defining a metric that allows us to compute the "nearness" or "similarity" of training patterns. The problem of defining a metric for training patterns has been discussed in the con-

pattern (see Bridle [12], Nowlan [52, 53, 54], and Nowlan and Hinton [55] for a more thorough discussion of soft and hard competitive learning).

nectionist literature in terms of the difficulty of defining a formal theory of what it means to generalize correctly (Hinton [29]). The inability to characterize a clustering of training patterns, like the inability to characterize a system's ability to generalize, is due to the absence of a suitable distance metric for training patterns.

Multiplicative Connections—A second feature our modular architecture has in common with systems previously proposed is the use of multiplicative connections. Because these connections allow one set of inputs to determine the mapping from the remaining inputs to the outputs of the system, they are particularly useful when there is a natural distinction between information to be processed and information that sets the context for processing. Specifically, multiplicative connections allow the activation of one unit of a network to act like the weight on the connection between two other units. Referring to Figure 3.2, if the activation of unit A is used as the weight on the connection between units B and C then the value of this weight is dependent on the input to A. An interesting special case occurs when the activation of A is binary. If the activation of A is zero then B has no influence on C. Alternatively, if the activation of A is one then the activation of B is the input to C. Many investigators have incorporated multiplicative connections in the design of their systems so that the system can compute different functions in different contexts (e.g., Hinton [27], Sejnowski [65], Feldman [20], Feldman and Ballard [21], McClelland [45], Maxwell, Giles, Lee, and Chen [44], Pomerleau [60], Yeung and Bekey [72]). Most relevant to our use of multiplicative connections is the work of Pollack [59] and Hampshire and Waibel [26].

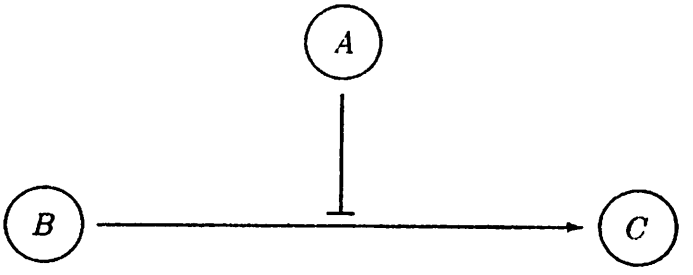

Figure 3.2: Multiplicative connection—The input to unit C is the product of the activations of units A and B.

The cascaded backpropagation (CBP) architecture of Pollack [59] consists of two networks called the context network and the function network. The output of the system is the output of the function network. However, the function network does not have a permanent set of weights. Instead, the outputs of the context network are used as the weights of the function network. Because the context network sets the weights of the function network, the mapping performed by the function network is said to be programmable by the context network. Using the chain rule, the backpropagation algorithm can be extended so that the appropriate weights of the context network can be learned. While the CBP architecture appears to consist of two networks, a simple change in notation shows that the architecture is equivalent to a single network with multiplicative connections. To the best of our knowledge, the CBP architecture was the first system to learn the appropriate weights on the multiplicative connections. Pollack has shown that the CBP architecture is capable of decomposing a task into a set of subtasks and of learning the set of subtasks faster than a single network can learn the undecomposed task. A disadvantage of this system is that the number of

output units of the context network must equal the number of weights of the function network and, thus, the number of units required by the system scales poorly with the size of the task. A second disadvantage is the the system uses the same network topology to perform all tasks.

Independently of the research described here, Hampshire and Waibel [26] developed an architecture similar to the one presented in Figure 3.1. They refer to this architecture as a "meta-pi" network. However, their system does not perform task decomposition. It is trained in two stages. In the first stage, a useful task decomposition is provided to the system, and each expert network is separately trained to perform one of the subtasks. In the second stage, the gating network is trained to switch in the appropriate expert network in the appropriate context. Because the meta-pi network does not itself perform task decomposition, the issues studied by Hampshire and Waibel are different from the issues we are addressing.

Stochastic Learning Automata—The gating network uses a learning procedure related to stochastic learning automata (Narendra and Thathachar [51]). Stochastic learning automata maintain a probability distribution over a set of actions. At each time step, an action is selected according to this distribution. If the environment provides the automaton with a reward, then the probability of performing the selected action is increased, whereas if the environment provides the automaton with a penalty, the probability of performing the selected action is decreased. After modifying the probability of performing the selected action, all the action probabilities are modified so that they sum to one. The learning procedure of the gating network is similar to

this, where the expert networks correspond to actions and the outputs of the gating network correspond to action probabilities. According to this view, if $J_y(t) < \gamma\overline{J_y}$, the gating network is rewarded; otherwise, it is penalized.

Brain Lateralization—The idea of competition between networks has appeared in the cognitive neuroscience literature in the form of the hypothesis that hemispheric specialization in humans is due to competition between neural subsystems. For example, Kosslyn [40] proposed that the brain contains many processing subsystems, each a neural network, which compete to learn about inputs. If the output of a network is used in subsequent computational processing, then the weights among connections in that network are altered so that the network produces the output faster and with less noise when the input recurs in the future. The weights of the networks whose outputs were not used in subsequent processing remain unchanged. Consequently, the networks compete to have their outputs used, and the strength of the training information received by a network is directly related to how that network fares in the competition. Kosslyn's credit assignment policy is thus nearly identical to the credit assignment policy of the modular architecture described above, and his hypothesis about why different subsystems of the brain learn to perform different tasks is consistent with our reasoning about why different expert networks of the modular architecture learn to perform different tasks.

Theories of brain lateralization also include the hypothesis that asymmetries in the cerebral hemispheres may influence the lateralization of brain functions (Geschwind

and Galaburda [24]). For example, if the left and right hemispheres compete for the ability to process language, then anatomical differences between the two hemispheres may bias the competition so that the left hemisphere usually wins. Similarly, a property of the modular architecture presented here is that the expert networks' architectures influence the competition between these networks. The architecture tends to allocate to each task the expert network whose topology is most appropriate for that task. For example, when required to perform a linear task and a nonlinear task, a linear expert network tends to win the competition to learn the linear task, whereas a nonlinear network tends to win the competition to learn the nonlinear task. A demonstration of this property is provided in Chapter 4 where we compare the performances of the modular architecture with that of two other networks trained with the backpropagation algorithm on the "what" and "where" vision tasks studied by Rueckl et al. [62].

3.5 An Alternative Error Function

The modular architecture consists of an arrangement of networks (Figure 3.1), an error function used in training the expert networks (Equation 3.2), and an error function used in training the gating network (Equation 3.6). Before reporting the results of training the modular architecture to perform various tasks, it is important to consider why the architecture was designed in this manner. In particular, we address the question of why the sum of squared error (SSE) between the desired

and actual outputs of the architecture (Equation 3.2) is used in training the expert networks but not in training the gating network. To answer this question, we compare the behavior of a modular architecture with that of a *cooperative architecture*, which is an architecture with the same arrangement of networks as the modular architecture but which uses the SSE in training both the expert and gating networks.

A cooperative architecture approximates a desired output pattern by linearly combining the outputs of all its expert networks. In contrast, a modular architecture tends to approximate a desired output pattern by "switching on" one expert network and "switching off" all other expert networks. The modular architecture may be considered a special case of the cooperative architecture in which the linear combination of the expert networks is restricted to a coefficient of one on the output of one of the expert networks and a coefficient of zero on the outputs of all other expert networks. Consequently, if cooperative and modular architectures have expert and gating networks with identical structures, then there may exist functions which the cooperative architecture can learn to approximate but which cannot be closely approximated by the modular architecture. Furthermore, if a solution is defined to be the set of functions computed by the expert networks at a single instant in time, then there may exist a large number of possible solutions that would allow a cooperative architecture to approximate a given function, whereas the set of solutions that would allow a modular architecture to approximate the same function may be relatively small. Because cooperative architectures have a larger number of potential solutions available

to them, one might be tempted to predict that these architectures learn faster than modular architectures.

Unfortunately, the question of which architecture learns faster cannot be answered in general, but rather its answer depends on such factors as the nature of the task that the architectures are required to perform, the structure of the expert and gating networks that compose the architectures, and the optimization procedures that are used to update the weights of the architectures. Consider a task in which different subtasks must be performed in different contexts. If the subtasks share many common properties, then an expert network that is useful in one context may also be useful in other contexts, albeit to a varying degree. In this case, a cooperative architecture may have an advantage over a modular architecture because it linearly combines the outputs of its gating networks using real–valued coefficients (however, see Chapter 5 for a modification to the modular architecture that allows it to explicitly model the common properties of a set of subtasks). Alternatively, if the subtasks do not share common properties or if the knowledge gained about one subtask interferes with the performance of other subtasks (e.g., negative transfer of training), then an expert network that is useful in one context is likely to be useless in other contexts or even diminish the performance of subtasks that must be performed in other contexts. In this case, a modular architecture may have an advantage over a cooperative architecture because it is constrained to combine the outputs of its expert networks using binary coefficients in which one coefficient is equal to one and all other coefficients

are equal to zero. Thus, it can quickly "switch off" an expert network that is not useful in a particular context.

The learning speeds of the modular and cooperative architectures are also dependent on the architectures' structures. In particular, whether the knowledge gained by an expert network during training on one subtask influences an architecture's ability to perform a second subtask is dependent on the structure of the architecture's expert and gating networks as well as the nature of the subtasks. If the expert and gating networks use relatively sparse representations (e.g., coarse–codes), then there is likely to be little generalization from one subtask to another. In contrast, if the networks use relatively compact representations, then there is likely to be alot of generalization from one subtask to another.

Whether the modular or cooperative architecture learns faster may also depend on the optimization procedure used to modify the architectures' weights. For example, one architecture may learn faster if the weights are modified using a conjugate gradient procedure with a line search, whereas the other architecture may learn faster if the weights are modified using a steepest descent procedure without a line search.

Although it is not possible to claim that one architecture learns faster than the other, we believe that many of the advantages of modular architectures over single networks (see Chapter 2) are also advantages of modular architectures over cooperative architectures. Specifically, suitably designed modular architectures are superior in terms of representation capabilities, generalization capabilities, and their ability to

satisfy constraints imposed by hardware limitations. These advantages stem from the fact that, due to a lack of computational constraints, a cooperative architecture's expert networks tend to learn functions that are arbitrarily related to the task that the architecture is trained to perform. In contrast, a modular architecture is sufficiently constrained so that its expert networks tend to learn functions that are meaningfully related to the target task.

In order to demonstrate this difference between the modular and cooperative architectures, we trained both architectures to approximate the function

$$f(x) = \begin{cases} \sin x & \text{if } x \in [-2\pi, 0] \\ \sin x + \sin 2x & \text{if } x \in (0, 2\pi]. \end{cases} \tag{3.7}$$

At each time step, x is selected from a uniform distribution over the interval $[-2\pi, 2\pi]$. The arrangement of networks used by the modular and cooperative architectures is shown in Figure 3.3. Each expert network is a linear unit with inputs set to $\sin x$ and $\sin 2x$. The gating network has two output units whose activations are between 0 and 1. The input to the gating network is a coarse-code of x (see the description given in Chapter 6 for more simulation details).

Because the modular architecture is constrained to "switch on" one expert network and "switch off" the other expert network at each time step, it can only compute f in two ways. The first expert network may compute $f(x) = \sin x$ and the second expert network may compute $f(x) = \sin x + \sin 2x$, or vice versa. Simulation results show that the modular architecture consistently finds one of these two possible solutions.

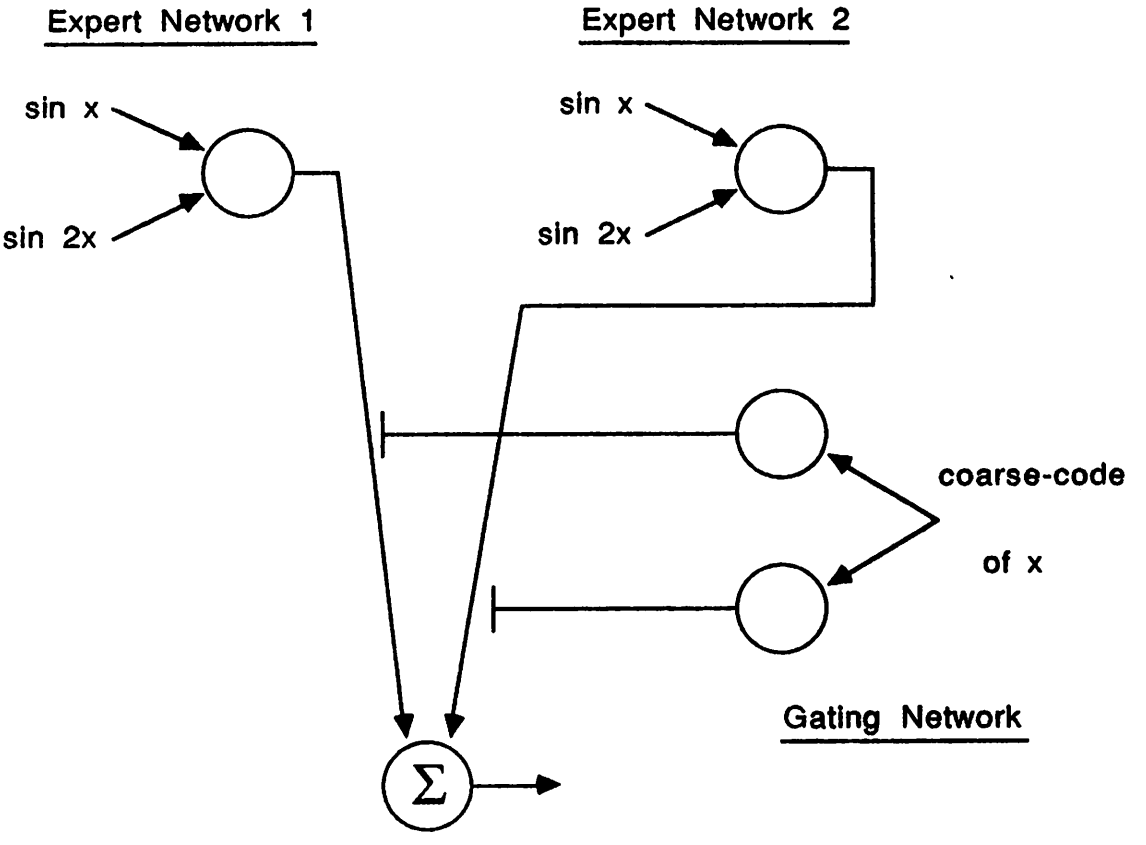

Figure 3.3: Arrangement of networks used by the modular and cooperative architectures.

The output units of the cooperative architecture's gating network may attain any value between 0 and 1. Consequently, there are an infinite number of ways in which the cooperative architecture can compute f. Not surprisingly, simulation results show that this architecture typically does not approximate f by computing $f(x) = \sin x$ with one expert network and $f(x) = \sin x + \sin 2x$ with the other expert network. Instead, the functions learned by the expert networks tend to have an arbitrary relationship with f, and both expert networks contribute to the output of the architecture for all values of x. Furthermore, each time the cooperative architecture is trained, the expert networks tend to learn to compute functions that are different from the functions that they learned during previous training sessions.

The expert networks of modular architectures, but not cooperative architectures, tend to learn identifiable and independent subtasks of the task that the architectures are trained to perform. Thus, modular architectures have advantages over cooperative architectures in terms of representation capabilities because they develop more interpretable representations. As argued in Chapter 2, systems with interpretable components are easier to understand, easier to extend by using the components as "building blocks" in the performance of more complex tasks, and can facilitate the use of attentional mechanisms. In addition, suitably designed modular architectures have advantages over cooperative architectures in terms of generalization capabilities. Modular architectures, but not cooperative architectures, can allocate expert networks with suitable structures to subtasks. Furthermore, for reasons analogous to those listed in Chapter 2, suitably designed modular architectures can more closely

satisfy several types of constraints imposed from the level of hardware than cooperative architectures. In conclusion, based on our discussion in Chapter 2 and in this section, we believe that the constraints placed upon the modular architecture give it significant advantages over single networks as well as other connectionist systems that use multiplicative connections, such as the cooperative architecture.

CHAPTER 4

THE WHAT AND WHERE VISION TASKS

Despite the fact that a variety of images are produced on the retina of a person watching a rotating or translating object, people recognize that the same object is depicted in each of the images. The ability to perform object recognition is said to be orientation and translation invariant. One hypothesis about how this invariance is achieved is that a canonical representation of each familiar object is stored, and the retinal image of an object is transformed so that the image and the representations can be compared. As a result of this transformation, information relevant to determining an object's spatial location is lost. This suggests that the process performing object recognition does not also perform spatial localization. Neuroscientists speculate that distinct cortical pathways of the primate visual system compute object recognition and spatial localization. Mishkin, Ungerleider, and Macko [49] reviewed evidence that a pathway running ventrally, interconnecting the striate, prestriate, and inferior temporal areas, computes object recognition, whereas a pathway running dorsally,

interconnecting the striate, prestriate and inferior parietal areas, computes spatial localization.

To investigate the computational advantages of employing distinct systems to perform these two tasks, Rueckl, Cave, and Kosslyn [62] compared the performance of two connectionist systems on an object recognition task (henceforth referred to as the "what" task) and a spatial localization task (henceforth referred to as the "where" task). The first system consisted of a single network that was required to perform both tasks. The second system consisted of two networks, one for each task. The retina was represented as a $5\,X\,5$ binary matrix. Each object was a specific pattern of binary entries in a $3\,X\,3$ matrix. At each time step of the training period, one of nine object matrices was centered at one of nine locations on the retinal matrix. The entries of the retinal matrix that lie outside the object matrix were set to zero. The "what" task is to identify the object; the "where" task is to identify its location.

The single–network system that Rueckl et al. [62] applied to these tasks was a network with two layers of modifiable weights. It had 25 input units, 18 hidden units, and 18 output units (see Table 4.1) and was strictly layered, meaning that all input units were connected to all hidden units, which in turn were connected to all output units. The input units encoded the $5\,X\,5$ retinal matrix, and the 18 output units corresponded to the nine possible objects and nine possible locations. The second system studied by Rueckl et al. [62] was a modification of this single–network system. Whereas the single–network system had connections from all hidden

Table 4.1: Systems studied by Rueckl, Cave, and Kosslyn on the "what" and "where" tasks.

System	Networks	Input	Output
1	$25 \rightarrow 18 \rightarrow 18$	retinal matrix	"what" and "where"
2	$25 \rightarrow 14 \rightarrow 9$	retinal matrix	"what"
	$25 \rightarrow 4 \rightarrow 9$	retinal matrix	"where"

units to all output units, the second system only had connections from the first 14 hidden units to the first 9 output units and from the remaining 4 hidden units to the remaining 9 output units (see Table 4.1). Thus, this system consisted of two separate networks, one for the "what" task and the other for the "where" task. Both the single–network system and the two–network system learned by means of the backpropagation algorithm.

Simulations showed that the two–network system learned the tasks faster and developed more interpretable representations than the single–network system. According to Rueckl et al. [62], both of these advantages resulted from the fact that the hidden units of the single–network system received inconsistent training information because they were connected to the output units for both the "what" and "where" tasks. Thus, the single–network system suffered from spatial crosstalk. In contrast, the hidden units of the two–network system did not receive inconsistent training information because they were connected to the output units for only one task. Based on these results, Rueckl et al. [62] concluded that it is better for a connectionist

system—and, by analogy, the primate visual system—to perform the "what" and "where" tasks in distinct networks.

An issue that Rueckl et al. [62] did not address, and the issue with which we are primarily concerned, is the development of a system that can learn for itself if it is better to decompose a learning task into two or more simpler tasks, and if so can allocate distinct networks for learning each simpler task. The next section presents simulation experiments using the "what" and "where" tasks to demonstrate that our modular architecture has this ability.

4.1 Simulation Experiments

Simulations of our modular architecture applied to the "what" and "where" tasks were conducted to investigate the architecture's ability to decompose learning tasks and the advantages that this ability may provide. For comparative purposes, several single networks were also simulated as applied to these tasks. Two sets of simulations were conducted. One set was designed to examine the modular architecture's behavior in the presence of temporal crosstalk; the other set of simulations addresses spatial crosstalk.

In describing the simulations, we refer to three different time periods: at each *time step*, a system is presented with a single input–output pair; during each *epoch*, a system is presented with every input–output pair in the training set exactly once; and a *run* consists of 100 epochs. In all simulations, the measure of a system's performance

is the percentage of input–output pairs that the system performs correctly during each epoch. A training pair is considered to be performed correctly when each output unit of the system has an activation greater than 0.6 when its desired activation is 1 and an activation less than 0.4 when its desired activation is 0.[1] The results for each system were averaged over 25 runs. For many parameters of each system (e.g., step size and momentum), we used the values that appeared to give the best performance. These values and additional details of the simulations are provided in Appendix A.

4.1.1 Temporal Crosstalk

Recall that temporal crosstalk occurs when a system is trained to perform different tasks at different times. We trained a modular architecture and single networks to perform the "what" and "where" tasks. At any given time step, each system was trained to perform only one task. In addition to receiving 25 input values corresponding to the entries of the 5 X 5 retinal matrix, each system received an input indicating whether it should perform the "what" or the "where" task. We call this input the *task bit*. There are 162 different input–output pairs (9 objects times 9 locations times 2 tasks).

[1]The more common sum of squared error measure was also computed and yielded results qualitatively similar to those produced using the percent correct. Thus, the sum of squared error is not reported here.

Two training procedures were employed to test each system's robustness in the presence of temporal crosstalk. One procedure avoids temporal crosstalk while the other does not. In the training procedure that avoids temporal crosstalk, called *random training*, at each time step an input pattern, consisting of a retinal matrix and a task bit, is randomly selected according to a uniform distribution over the set of 162 possible input patterns. The system's desired response is the object identity or location depending on the task bit. In the training procedure that is susceptible to temporal crosstalk, called *blocked training*, the task bit changes only once during each epoch. At the start of each epoch, one of the two tasks is randomly selected (with probability 0.5), and the input patterns presented during the first 81 time steps of the epoch have the task bit set to indicate the selected task. The input vectors presented during the last 81 time steps of the epoch have the task bit set to the other task. For example, during an epoch, a system may be required to perform the "what" task for 81 consecutive time steps followed by the "where" task for 81 consecutive time steps. At each time step within an epoch, the object and its location are randomly selected, and the desired system response depends on the task bit.

The first system trained to perform the "what" and "where" tasks is a single network trained with the backpropagation algorithm. It is strictly layered and contains 26 input units, 18 hidden units, and 9 output units (see Table 4.2). The input units encode the $5 X 5$ retinal matrix and the task bit. Figure 4.1 shows the learning curves for this network using the random training and blocked training procedures. The horizontal axis gives the number of epochs; the vertical axis gives the percent

Table 4.2: Systems used in the temporal crosstalk experiments.

Single Networks		
System	Networks	Input
1	26 → 18 → 9	retinal matrix and task bit
2	26 → 36 → 9	retinal matrix and task bit

Modular Architecture				
System	Expert Networks	Expert Networks' Input	Gating Network	Gating Network's Input
3	26 → 36 → 9	retinal matrix and task bit	1 → 3	task bit
	26 → 18 → 9	retinal matrix and task bit		
	26 → 9	retinal matrix and task bit		

of input-output pairs performed correctly. Clearly, this network learned faster with random training than with blocked training (at epoch 50, the difference between the performance with random and blocked training is statistically significant at the $p < 0.01$ level ($t = 4.84$)). This difference can be explained by the influence of temporal crosstalk in the case of blocked training. Temporal crosstalk attenuates the network's rate of learning in blocked training.

The second system trained to perform the "what" and "where" tasks is identical to the first system except that it has 36 hidden units instead of 18. Figure 4.2 shows the learning curves for this larger network. For the first 25 epochs, this network learned slightly faster with blocked training than with random training. However, for epochs 25–85, it learned significantly faster with random training than with

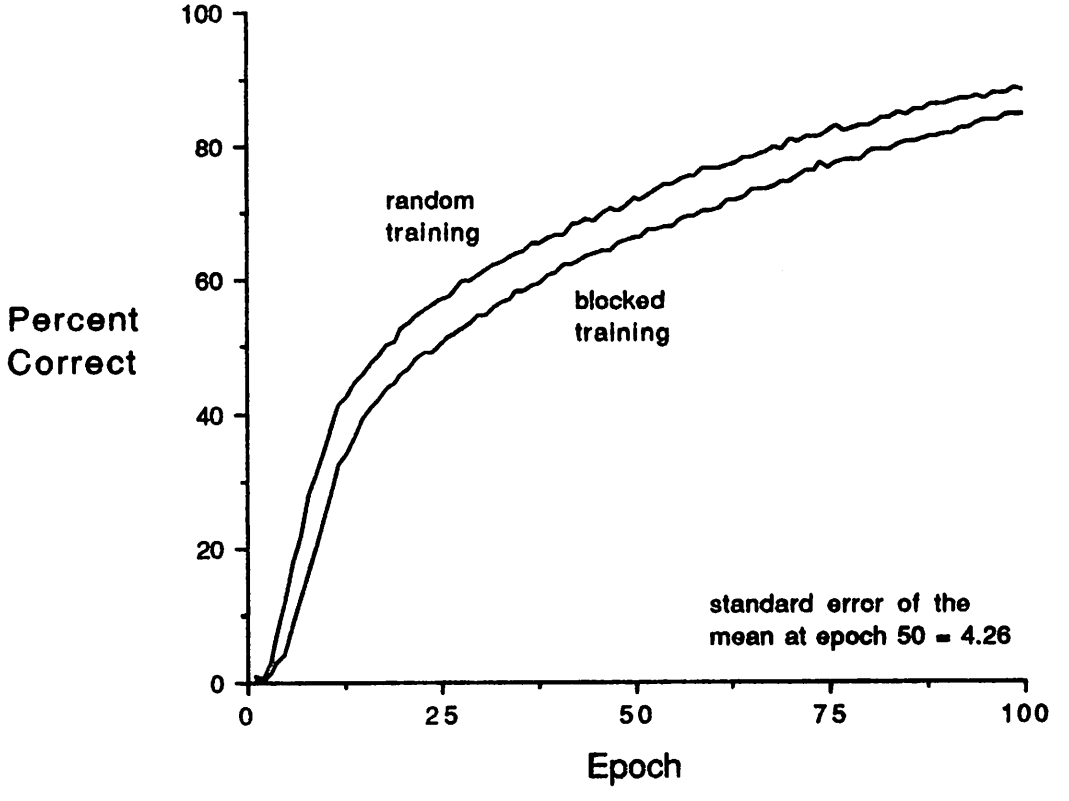

Figure 4.1: Learning curves for the 26 → 18 → 9 network on the "what" and "where" tasks using random and blocked training.

blocked training (at epoch 50, the difference between the performance with random and blocked training is statistically significant at the $p < 0.01$ level ($t = 7.89$)). Similar to the $26 \rightarrow 18 \rightarrow 9$ network described above, temporal crosstalk attenuates the $26 \rightarrow 36 \rightarrow 9$ network's rate of learning in blocked training. Therefore, we cannot conclude that an abundance of hidden units makes a network more robust in the presence of temporal crosstalk.

The modular architecture that we applied to the "what" and "where" tasks, shown in Figure 4.3, consists of three expert networks and a gating network (see Table 4.2). Each expert network has 26 input units and 9 output units. The input units encode the $5X5$ retinal matrix and the task bit. Two of the expert networks are multi-layered networks with 36 and 18 hidden units respectively. The third expert network has a single layer, i.e., it has no hidden units. The gating network has 1 input unit which encodes the task bit and 3 output units corresponding to the three expert networks.

There are at least three ways that this modular architecture might successfully learn the "what" and "where" tasks. One of the multi-layer expert networks could learn to perform both tasks. In this case, the gating network must always gate on the network that learns both tasks and gate off the remaining expert networks. Although this is a possible solution, the results of Rueckl et al. [62] described above suggest that it is not the best solution in terms of learning speed and clarity of the resulting representation. A second possibility is that one of the multi-layer expert networks could learn the "what" task, and the other multi-layer expert network could learn the

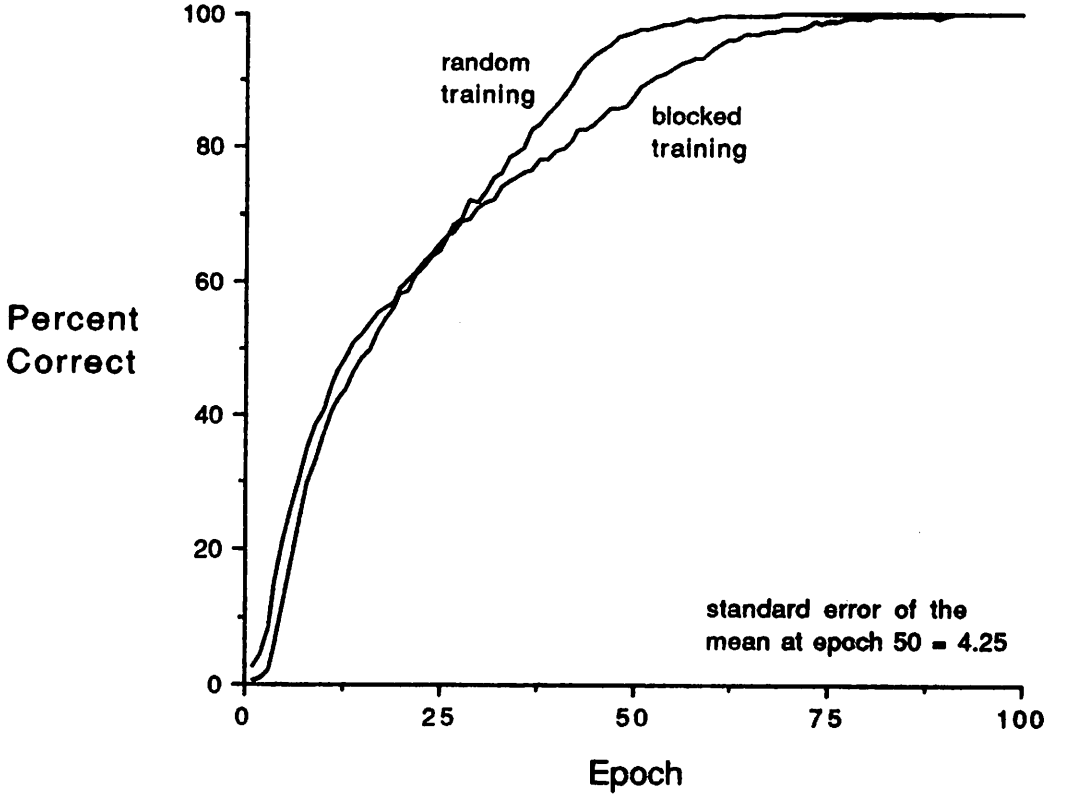

Figure 4.2: Learning curves for the 26 → 36 → 9 network on the "what" and "where" tasks using random and blocked training.

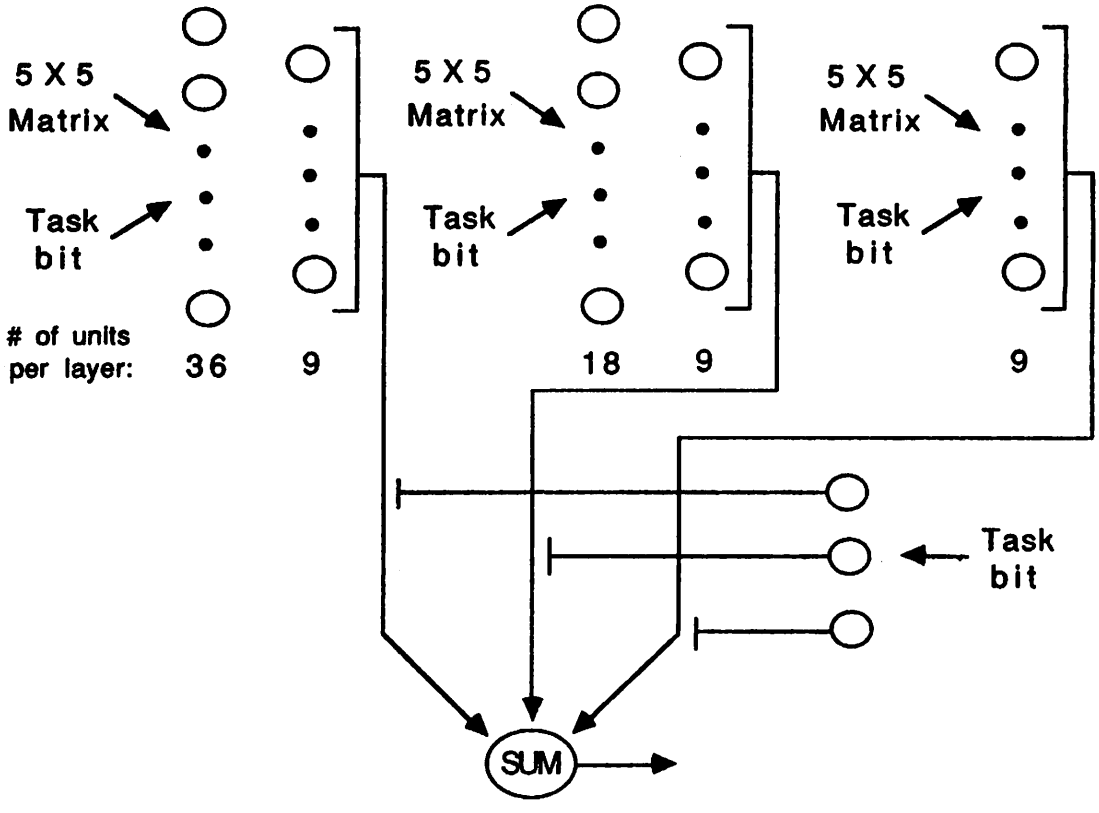

Figure 4.3: The modular architecture simulated in the temporal crosstalk experiments.

"where" task. In this case, the gating network must gate on the appropriate expert network based on the value of the task bit. This solution would indicate that the modular architecture had learned that it was required to perform two independent tasks and had allocated distinct networks to each task. However, a shortcoming of this solution is apparent when it is noted that, using the retinal images designed by Rueckl et al. [62], the "where" task is linearly separable. This means that the structure of the single–layer expert network most closely matches the "where" task. Consequently, a third and possibly best solution would be one in which one of the multi–layer expert networks learned the "what" task and the single–layer expert network learned the "where" task. This solution would not only show task decomposition but also the appropriate allocation of tasks to expert networks.

The simulation experiments using the modular architecture show that it produces this third possible solution. It always allocates the first multi–layer expert network to the "what" task and the single–layer expert network to the "where" task. This result suggests that, at least in some circumstances, the modular architecture is capable of performing function decomposition and that it tends to allocate to each function a network with a structure appropriate to that function. Learning curves for the modular architecture on the "what" and "where" tasks using random and blocked training are shown in Figure 4.4. Because little difference exists between performance with random and blocked training, these results suggest that the modular architecture is robust in the presence of temporal crosstalk (at epoch 50, the difference between the performance with random and blocked training is not statistically significant at

the $p < 0.01$ level ($t = 0.08$)). This robustness is due to the architecture's ability to allocate distinct networks to learn the different tasks.

A comparison of the learning curves in Figures 4.1, 4.2, and 4.4 shows that both the $26 \rightarrow 36 \rightarrow 9$ network and the modular architecture learn the "what" and "where" tasks faster than the $26 \rightarrow 18 \rightarrow 9$ network. In addition, the results suggest that of the three systems studied, only the modular architecture is capable of showing robust performance in the presence of temporal crosstalk.

4.1.2 Spatial Crosstalk

As presented so far, a limitation of the modular architecture is that only one expert network determines the output of the architecture at any one time step. When different tasks must be performed simultaneously, this system is unable to allocate different expert networks to learn the different tasks. Hence, the modular architecture is vulnerable to the detrimental effects of spatial crosstalk. Fortunately, a simple modification of the architecture overcomes this problem.

Consider the architecture illustrated in Figure 4.5. It consists of two expert networks and two gating networks. One gating network gates the first m components of each expert network's output vector, and the other gating network gates the remaining components. Suppose one function determines the desired values for the first m components of the architecture's output vector, $\mathbf{y}$, and a second function determines the desired values for the remaining components. The gating networks may allocate

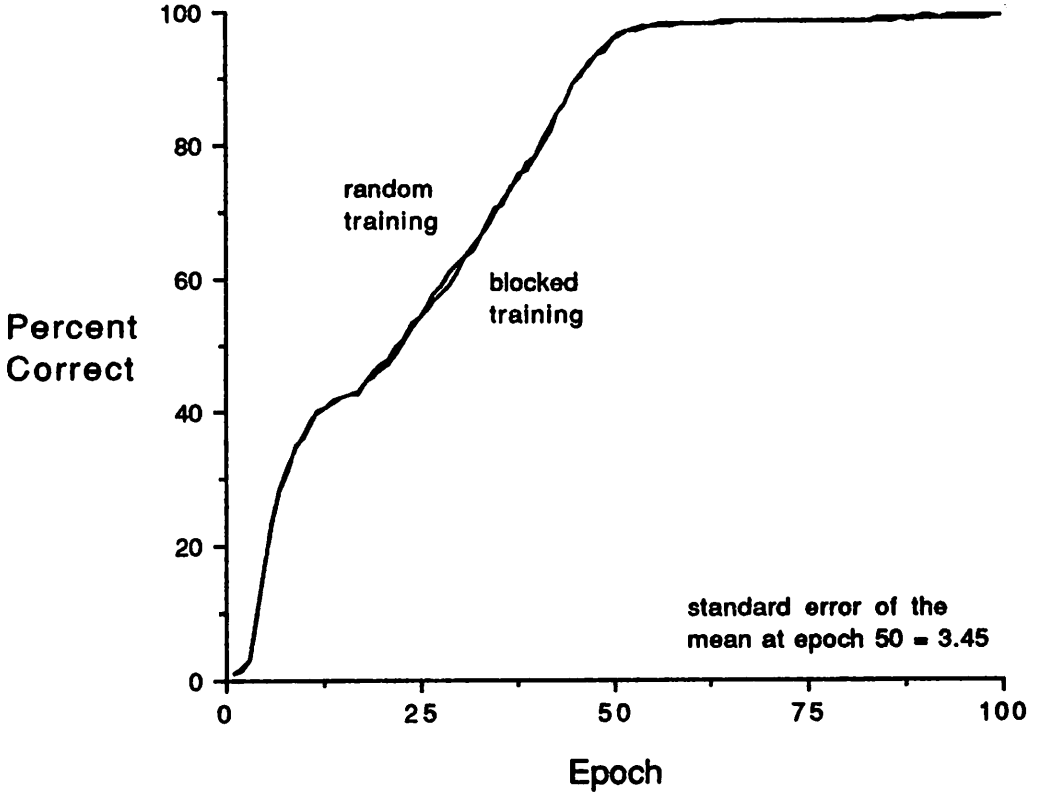

Figure 4.4: Learning curves for the modular architecture on the "what" and "where" tasks using random and blocked training.

the same expert network to both functions, or they may allocate different expert networks to each function.[2]

As in the case of the modular architecture with a single gating network, competition among the expert networks of the architecture with multiple gating networks determines how training patterns are allocated during learning. The learning algorithm for the modular architecture with multiple gating networks is identical to that described in Chapter 3 for an architecture with one gating network, with the proviso that the weight modifications are determined independently for each gating network.

Specifically, a modular architecture with multiple gating networks can be thought of as multiple separate modular architectures each having a single gating network. The separate architectures share expert networks although their gating networks gate

[2]Assuming that the output units of the gating networks produce binary outputs, the training of each expert network is identical to the case where an external teacher provides desired values for some output units of the expert network and places "don't care" conditions on other output units. At any given time step, let X denote the set of an expert network's output units whose outputs are multiplied by a gating network output of one, and let Y denote the set of output units whose outputs are multiplied by a gating network output of zero. During training, a non–zero error vector is backpropagated to the units in X, and the zero vector is backpropagated to the units in Y. Therefore, the output units in X, but not in Y, modify their weights and send error information to the hidden units of the network. In this sense, the training of this expert network is identical to the case where desired values are provided for the units in X and "don't care" conditions are placed on the units in Y (see Jordan [34] for a discussion of "don't care" conditions and spatial crosstalk).

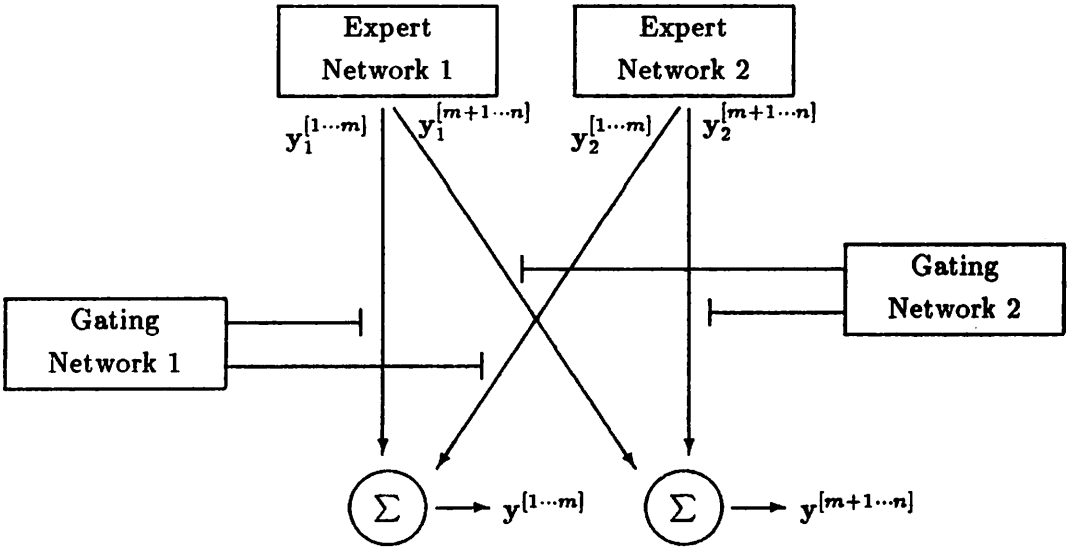

Figure 4.5: A modular architecture with multiple gating networks ($y_1^{[1\cdots m]}$ denotes the vector whose components are the first m components of expert network 1's output vector, and the other expressions similarly denote subvectors).

different sets of the expert networks' output components. Consequently, the output components of the expert networks that are not gated by a gating network do not participate in the weight modification process for that gating network. This implies that for each gating network there is a separate process for determining when the performance of the architecture has significantly improved, and this process is identical to that used in architectures with single gating networks (Equation 3.4) except that it only depends on the error over the output components gated by the gating network. This results in different values λ_{WTA} and λ_{NT} for each gating network and therefore different error functions for each gating network. The error functions are given by Equation 3.6 with the substitution of the appropriate values for λ_{WTA} and λ_{NT}.

We trained a specific instance of the modular architecture shown in Figure 4.5 to perform simultaneously the "what" and "where" tasks. This means that at each time step the architecture's output pattern should correctly identify both the object and its location in the current input pattern. The specifications of this modular architecture are summarized in Table 4.3. It has two expert networks and two gating networks. Both expert networks have 25 input units and 18 output units. The input units encode the $5 X 5$ retinal matrix. Because both tasks are to be performed simultaneously, the architecture does not receive a task bit. The first 9 output units of each expert network encode the 9 possible objects, and the second 9 output units encode the 9 possible locations. The first expert network is a two–layer network with 36 hidden units, and the second expert network has a single layer of modifiable weights, i.e., it

Table 4.3: The modular architecture used in the spatial crosstalk experiment.

Expert Networks	Expert Networks' Input	Gating Networks	Gating Networks' Input
$25 \rightarrow 36 \rightarrow 18$	retinal matrix	$0 \rightarrow 2$	none
$25 \rightarrow 18$	retinal matrix	$0 \rightarrow 2$	none

has no hidden units. The two gating networks are extremely simplified: each has two output units but no hidden units and no input units. Each gating network therefore just consists of two output units, each with a single bias weight. The first gating network gates the first 9 output components of the expert networks, and the second gating network gates the second 9 output components of the expert networks.

There are two ways that this modular architecture might successfully learn to perform the "what" and "where" tasks. One possibility is that the multi–layer expert network learns to perform both tasks, and the second possibility is that the multi–layer expert network learns to perform the "what" task whereas the single–layer expert network learns to perform the "where" task. For the reasons described above, the second possibility is the better solution.

Our first simulations of the architecture on the two tasks produced disappointing results. The architecture consistently allocated the single–layer expert network to both tasks. Because the "what" task is not linearly separable, the architecture did not correctly perform this task. Further analysis of the modular architecture's learning

rule reveals the reasons for this behavior. Recall from Section 3 that the architecture's current performance is compared with its past performance at each time step. If the performance has significantly improved, then the weights of a gating network are modified so that the gating network output corresponding to the winning expert network increases toward one, and the outputs corresponding to the losing expert networks decrease toward zero. If the current performance does not show significant improvement, then a gating network's weights are modified so that all of its outputs are moved towards a neutral value.

In the initial training runs, it turned out that the single–layer expert network approximated the "what" function more quickly than did the multi–layer expert network. Therefore, the output of the first gating network corresponding to the single–layer expert network approached one, and the output corresponding to the multi–layer expert network approached zero. However, because the "what" task is not linearly separable, the performance of the single–layer expert network on the "what" task could not improve past a certain low level.[3] Consequently, we expected the outputs of the gating network to eventually approach the neutral value and then change so as to appropriately reallocate tasks to expert networks, but this did not happen. The architecture failed to reallocate tasks because even the low level of performance achieved by performing both tasks with the single–layer expert network was

[3]At best, the single–layer expert network can correctly perform 54 percent of the "what" task's input–output pairs.

better than the performance attainable by nearby weight values. The architecture consistently became trapped in this local error minimum.

Rather than devising a mechanism to allow the architecture to escape from this kind of local minimum, we modified the learning process in order to make the architecture less likely to become trapped in this way. During training we varied the contribution to the gating network error function (Equation 3.6) of the term responsible for making the outputs of the gating network approach the neutral value $\frac{1}{n}$, where n is the number of expert networks. This term, the fourth term in Equation 3.6, contains the factor λ_{NT} which is non-zero only when the architecture's performance has not significantly improved.[4] Instead of setting λ_{NT} for a gating network to one when performance has significantly improved, we initialized it to a value greater than one at the start of training and slowly decreased it to one during training.[5] Larger values of λ_{NT} increase the tendency of the gating network outputs to remain near the neutral value and therefore prolong the period of training before the expert networks specialize. This gives expert networks unable to compete in terms of initial rates of learning—but which may be better in terms of eventual performance—the chance to exert their superiority. As a result, an expert network that learns a task quickly at

[4]Recall that when there are multiple gating networks, the case we are discussing, there is a different λ_{NT} for each gating network.

[5]The effect of altering λ_{NT} is to alter the step size of changes in gating network weights that occur when the architecture's performance has not significantly improved.

first, such as the single–layer network in the "what" task, does not necessarily become allocated to that task.

When modified in this manner, the modular architecture with two gating networks that we simulated consistently learned to allocate the multi–layer expert network to the "what" task and the single–layer expert network to the "where" task. As the outputs of the expert networks came to be gated in an appropriate manner by the gating networks, the degree of conflict in the training information received by the output units of the expert networks decreased, that is, spatial crosstalk decreased. However, we cannot report that as a result of decreased spatial crosstalk the modular architecture learned the "what" and "where" tasks more rapidly than did the single–network system studied by Rueckl et al. [62]. It is not surprising that prolonging the period before the expert networks specialize considerably slows learning, but this may be necessary for the modular architecture to selectively allocate networks to tasks based on the suitability of the networks' topologies.

In summary, we have studied the performances of single networks and modular architectures on the "what" and "where" tasks of Rueckl et al. [62]. Our major goal has been to demonstrate the modular architecture's ability to perform function decomposition in the sense of learning to allocate different networks to learn different functions. A second goal has been to suggest that the architecture tends to allocate to each function a network with a topology that is appropriate to that function. We have also shown that, at least in the case presented here, the modular architecture is robust in the presence of temporal crosstalk.

CHAPTER 5

MULTI–PAYLOAD ROBOTICS TASK

The previous chapter showed that the modular architecture's rate of learning may be relatively unaffected by the inconsistent training information that characterizes temporal crosstalk. This chapter shows that this robustness leads to superior performance on at least one interesting "real world" task. This task is to control a simulated robot arm to move a variety of payloads, each of a different mass, along a specified trajectory. Two important properties of the modular architecture are demonstrated in this chapter. A suitably designed modular architecture tends to allocate the same expert network to similar tasks, resulting in the benefits of positive transfer of training, and different expert networks to dissimilar tasks, thereby avoiding the detrimental effects of negative transfer of training. In addition, the modular architecture can be easily modified so as to learn to perform a family of tasks by using one network to learn a shared strategy that is used in all contexts along with other networks that learn modifications to this strategy that are applied in a context sensitive manner.

The chapter is organized as follows. Section 5.1 describes the procedure used to train several connectionist systems to perform the multi-payload robotics task. Section 5.2 compares the results of training these systems. Section 5.3 presents a demonstration of the ease with which a priori knowledge of the domain may be embedded in the modular architecture.

5.1 Training Procedure

Frequently, controllers of robot arms employ fixed models of the dynamics of the arm. Unfortunately, the use of such models has several drawbacks. For many robot systems, accurate models are hard to formulate and model parameters are difficult to determine. Additionally, such models may be inflexible in the sense that they are closely dependent on the parameters of the system being modeled and require significant modification if the system is altered or interacts with a new tool or payload. Consequently, many researchers study techniques to allow controllers to learn desired control laws without detailed knowledge of the robot system.

One such technique uses an adaptive feedforward controller and a fixed feedback controller to control a robot arm. The feedback controller aids in generating training data that the feedforward controller uses to learn a model of the arm's inverse dynamics. To the best of our knowledge, this training procedure was developed independently by Atkeson and his colleagues (Atkeson and McIntyre [2], Atkeson and Reinkensmeyer [3]), Kawato and his colleagues (Kawato [37], Kawato, Furukawa, and

Suzuki [38]), and Miller and his colleagues (Miller [46], Miller, Glanz, and Kraft [47]). The connectionist systems reported in this chapter were trained using this procedure and, thus, it is now described in detail.

Let the state of a robot arm at time t be represented by its joint positions, $\theta(t)$, and joint velocities, $\dot{\theta}(t)$. In order to achieve joint accelerations $\ddot{\theta}(t)$, torques $\tau(t)$ must be applied to the arm. This relationship is known as the inverse dynamics of the arm and is written:

$$\tau(t) = f^{-1}(\theta(t), \dot{\theta}(t), \ddot{\theta}(t)). \tag{5.1}$$

The goal of the procedure is to train a feedforward controller to accurately model this relationship.

Each time step of the procedure contains a performance stage and a training stage. During the performance stage, both a feedforward controller and a feedback controller compute torques used to control the arm (see Panel A of Figure 5.1). The inputs to the feedforward controller are the desired joint positions, velocities, and accelerations for the current time step as specified by the desired trajectory. The outputs are the feedforward torques. The inputs to the feedback controller are the desired and actual joint positions and velocities. The outputs are the feedback torques. The sum of the feedforward and feedback torques are applied to the arm and the resulting joint accelerations are observed. During the training stage, the feedforward controller receives new inputs, namely the actual joint positions, velocities, and accelerations, and computes new outputs (see Panel B of Figure 5.1). The desired outputs of this

controller are the actual torques applied to the arm. In this manner, the feedforward controller learns a model of the arm's inverse dynamics. During the initial time steps of the procedure, most of the torques are generated by the feedback controller and the arm imprecisely follows the desired trajectory. However, after the feedforward controller learns an accurate model of the arm's inverse dynamics, it generates most of the torques and the arm faithfully follows the desired trajectory.

Our interest is in training connectionist systems to serve as feedforward controllers for a robot arm when a variety of payloads, each of a different mass, must be moved along a specified trajectory. In comparison with the case where the payload is fixed, controlling an arm with a time–varying payload is more difficult. Assuming that the payload is modeled as part of the last link of the arm, this difficulty arises because the inverse dynamics of the arm are strongly dependent on the mass of the payload. [1] In our simulations, the multi–payload task is particularly difficult because we assumed that the connectionist systems could detect the identity of a payload (e.g., payload A or payload B) but not the mass of the payload. Six payloads were used with masses of 0, 2, 10, 15, 22, and 27 kg respectively.

One complete simulation of the training of a connectionist system is called a run and is described using four time frames. Each run was divided into ten epochs. Each epoch was divided into six bins corresponding to the six payloads. At the start

[1] The inverse dynamics of the arm are also dependent on the moments of inertia and center of mass of the payload. However, as mentioned below, we modeled the links of the robot arm and the payloads as point masses and, thus, these factors did not play a role in our simulations.

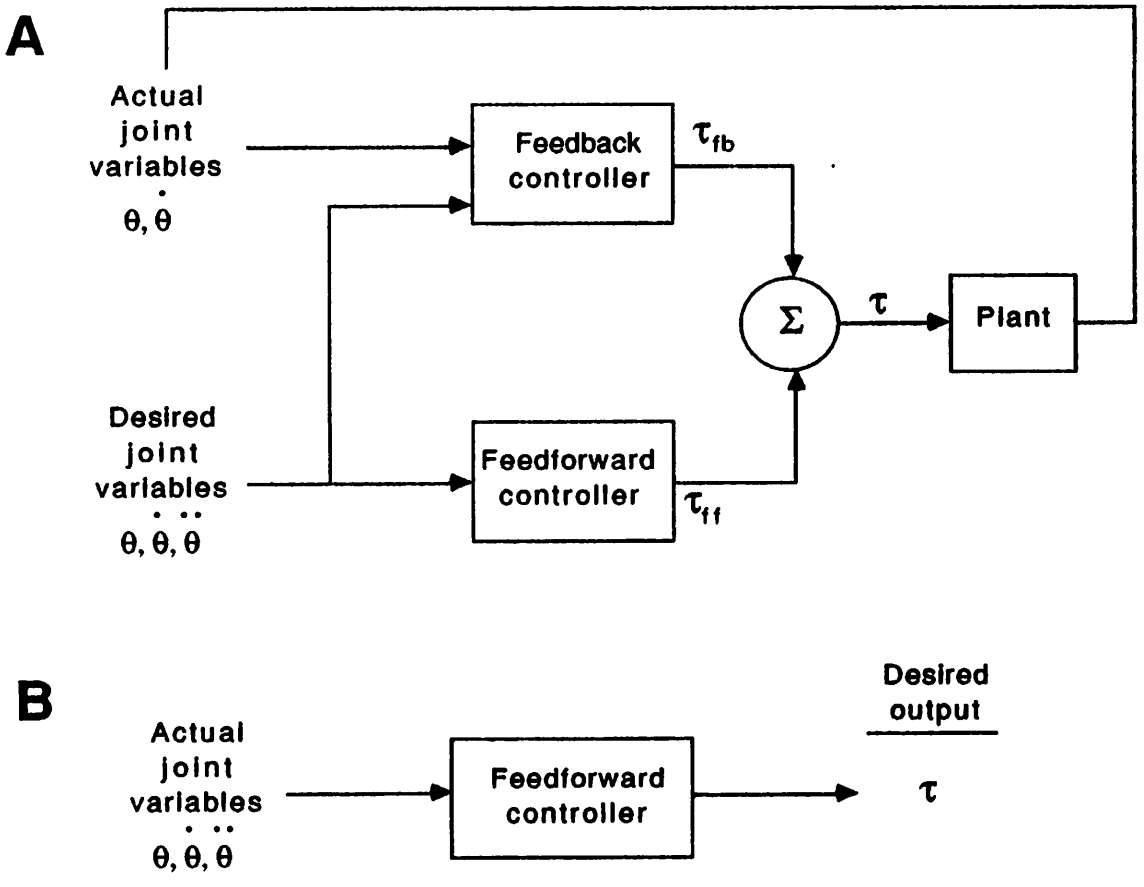

Figure 5.1: A: During the performance stage, both a feedforward controller and feedback controller compute torques used to control the robot arm. B: During the training stage, the feedforward controller receives the actual joint variables. Its desired output is the torque applied to the arm.

of each bin, a payload was randomly selected from a uniform distribution such that each payload was selected exactly once in each epoch. During a bin, the connectionist system attempted to direct the robot arm with the selected payload along the desired trajectory five times. Torques were applied at a rate of 100 Hz and the desired movement has a duration of 2 seconds. Therefore, each traversal of the trajectory lasted 200 time steps. In summary, a run comprised 10 epochs; an epoch comprised 6 bins; a bin comprised 5 traversals of the desired trajectory; a traversal comprised 200 time steps.

The robot arm is the two–joint planar manipulator shown in Figure 5.2. Each link of the arm was modeled as a point mass located at its distal end. The payload was modeled as part of the second link. The dynamic equations were obtained from Craig [15]. These equations were integrated using a fourth–order Runge–Kutta method with a time step of 0.01 seconds. The parameters of the arm are listed in Table 5.1. The feedback controller was a position controller with a gain of 1000 kg $\cdot$ m^2/s^2/radian. The parameters of the arm and of the feedback controller were obtained from Miller, Glanz, and Kraft [47]. The desired trajectory is a straight line horizontal movement with a velocity of zero at the endpoints and the maximum velocity at the center of the trajectory. If the origin of a Cartesian coordinate system lies at the axis of joint 1, then the coordinates of the desired trajectory, quantized in meters, are

$$x(t) = -\cos(\frac{t\,\pi}{2}) \tag{5.2}$$

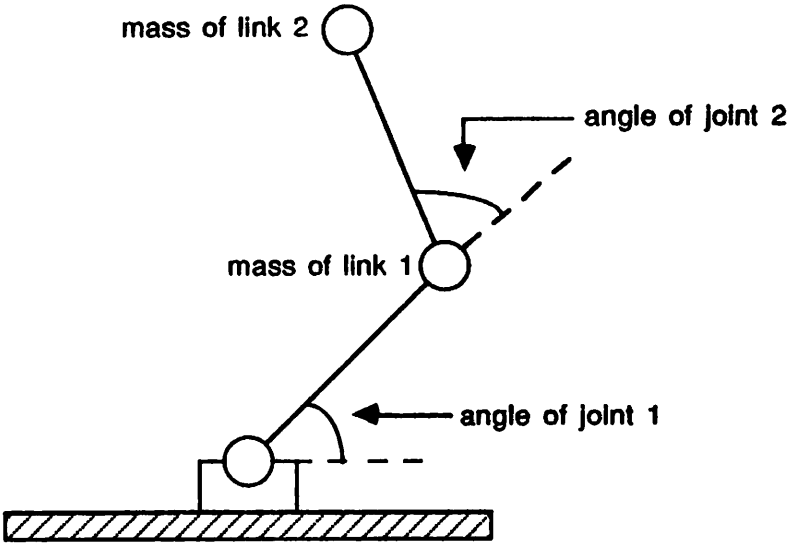

Figure 5.2: Two–joint planar arm.

and

$$y(t) = 1, \tag{5.3}$$

where $t \in [0, 2]$.

Table 5.1: Parameters of the robot arm.

Parameter	Link 1	Link 2
Length	1.0 m	0.8 m
Mass	10 kg	10 kg
Viscous friction coefficient	30 kg · m^2/s/radian	20 kg · m^2/s/radian

5.2 Comparison of Architectures

Six connectionist systems were trained to serve as feedforward controllers. These six systems consist of two systems from each of three classes of architectures. The two systems from each class were trained using different sets of input vectors. These sets of input vectors are labeled L and NL (L stands for linear; NL stands for nonlinear). The input vectors in both sets contain 19 components. Ten components are binary and represent ten possible payloads (only six of the payloads were actually used in the simulations). The remaining nine components are real–valued and represent the robot arm's joint positions, velocities, and accelerations. It is in the representation of the robot arm's joint variables that the two sets of input vectors differ. In the input vectors of set L, the nine components are the transformations (labeled $f_i(\theta, \dot{\theta}, \ddot{\theta})$) of the joint variables shown on the left of Table 5.2. These transformations were conveniently chosen so as to facilitate learning of the robot arm's inverse dynamics. More precisely, these transformations were selected so that, for a fixed payload, the inverse dynamics of the robot arm are linear in the transformations (hence, the input vectors using these transformations are members of set L). This practice was adopted from Kawato et al. [38] who argued that such useful transformations can be prepared despite very incomplete knowledge of the arm's dynamics. In the input vectors of set NL, the nine components are the transformations (labeled $g_i(\theta, \dot{\theta}, \ddot{\theta})$) of the joint variables shown on the right of Table 5.2. Even if the payload is fixed, the inverse

Table 5.2: f_i and g_i are the transformations of the robot arm's joint positions, velocities, and accelerations used in the input vectors of sets L and NL respectively. The subscripts on $\theta, \dot{\theta}$, and $\ddot{\theta}$ are the joint number.

Set L					Set NL			
i	$f_i(\theta, \dot{\theta}, \ddot{\theta})$	i	$f_i(\theta, \dot{\theta}, \ddot{\theta})$		i	$g_i(\theta, \dot{\theta}, \ddot{\theta})$	i	$g_i(\theta, \dot{\theta}, \ddot{\theta})$
1	$\ddot{\theta}_1$	6	$\dot{\theta}_2^2 \sin\theta_2$		1	θ_1	6	$\ddot{\theta}_2$
2	$\ddot{\theta}_2$	7	$\dot{\theta}_1 \dot{\theta}_2 \sin\theta_2$		2	θ_2	7	$\cos\theta_1$
3	$\ddot{\theta}_1 \cos\theta_2$	8	$\cos\theta_1$		3	$\dot{\theta}_1$	8	$\cos\theta_2$
4	$\ddot{\theta}_2 \cos\theta_2$	9	$\cos(\theta_1 + \theta_2)$		4	$\dot{\theta}_2$	9	$\cos(\theta_1 + \theta_2)$
5	$\dot{\theta}_1^2 \sin\theta_2$				5	$\ddot{\theta}_1$		

dynamics of the robot arm are nonlinear in these transformations (hence, the input vectors using these transformations are members of set NL).

The specifications of the six connectionist systems trained to serve as feedforward controllers are listed in Table 5.3. The six systems are grouped into three classes of architectures. For each system, the system label indicates the class of architectures to which the system belongs and the set of input vectors used in training the system. Systems N–L and N–NL are single networks (N stands for network). System N–L was trained with the input vectors from set L and system N–NL was trained with the input vectors from set NL. Systems MA–L and MA–NL are modular architectures with three expert networks and one gating network. Systems MAS–L and MAS–NL are referred to as *modular architectures with share networks.* This class of architectures

is formed by modifying the standard modular architecture, and its use requires some explanation.

Throughout this thesis, the importance of learning to allocate different networks to learn different tasks has been emphasized. However, in some circumstances, this strategy has an important drawback. Consider two tasks that possess many similarities but also some dissimilarities. Since the tasks possess similarities, the features learned about one task may also be useful for learning and performing the second task. Therefore, allocating distinct networks to learn each task is a poor strategy. Alternatively, since the tasks possess dissimilarities, a network trained to perform both tasks may learn slowly and develop a relatively uninterpretable representation. Therefore, allocating one network to learn both tasks is a poor strategy.

An ideal strategy may be to allocate one network to learn the features that are unique to the first task, one network to learn the features that are unique to the second task, and one network to learn the features that are common to the two tasks. When performing a task, a system must combine the outputs of the network that learns the common features with the outputs of the network that learns the features particular to the task. In an attempt to design a modular architecture that can discover this ideal strategy, we have modified the standard modular architecture. This modified modular architecture, called a modular architecture with a share network, was inspired by an architecture that was proposed, but never implemented, by Kawato et al. [38].

Table 5.3: Single networks, modular architectures, and modular architectures with share networks used in the multi-payload robotics experiments.

Single Networks			
System	Network	Input	Output
N-L	$19 \rightarrow 10 \rightarrow 2$	payload identity and $f_i(\theta, \dot{\theta}, \ddot{\theta})$	feedforward torques
N-NL	$19 \rightarrow 10 \rightarrow 2$	payload identity and $g_i(\theta, \dot{\theta}, \ddot{\theta})$	feedforward torques

Modular Architectures				
System	Expert Networks	Expert Networks' Input	Gating Network	Gating Network's Input
MA-L	$9 \rightarrow 2$	$f_i(\theta, \dot{\theta}, \ddot{\theta})$	$10 \rightarrow 3$	payload
	$9 \rightarrow 2$	$f_i(\theta, \dot{\theta}, \ddot{\theta})$		identity
	$9 \rightarrow 2$	$f_i(\theta, \dot{\theta}, \ddot{\theta})$		
MA-NL	$9 \rightarrow 10 \rightarrow 2$	$g_i(\theta, \dot{\theta}, \ddot{\theta})$	$10 \rightarrow 3$	payload
	$9 \rightarrow 10 \rightarrow 2$	$g_i(\theta, \dot{\theta}, \ddot{\theta})$		identity
	$9 \rightarrow 10 \rightarrow 2$	$g_i(\theta, \dot{\theta}, \ddot{\theta})$		

Modular Architectures With Share Networks						
System	Share Network	Share Network's Input	Expert Networks	Expert Networks' Input	Gating Network	Gating Network's Input
MAS-L	$9 \rightarrow 2$	$f_i(\theta, \dot{\theta}, \ddot{\theta})$	$0 \rightarrow 2$	none	$10 \rightarrow 3$	payload
			$9 \rightarrow 2$	$f_i(\theta, \dot{\theta}, \ddot{\theta})$		identity
			$9 \rightarrow 2$	$f_i(\theta, \dot{\theta}, \ddot{\theta})$		
MAS-NL	$9 \rightarrow 10 \rightarrow 2$	$g_i(\theta, \dot{\theta}, \ddot{\theta})$	$0 \rightarrow 2$	none	$10 \rightarrow 3$	payload
			$9 \rightarrow 10 \rightarrow 2$	$g_i(\theta, \dot{\theta}, \ddot{\theta})$		identity
			$9 \rightarrow 10 \rightarrow 2$	$g_i(\theta, \dot{\theta}, \ddot{\theta})$		

The modular architecture with a share network, which is illustrated in Figure 5.3, consists of a share network as well as a set of expert networks and a gating network. During training, the share network learns a strategy that is useful for performing all tasks. This strategy is referred to as a shared strategy. The expert networks learn modifications to the shared strategy that are particular to individual tasks. The output of the architecture, $\mathbf{y}$, equals the output of the share network, $\mathbf{y}_s$, plus the gated outputs of the expert networks:

$$\mathbf{y} = \mathbf{y}_s + \sum_{i=1}^{n} g_i \mathbf{y}_i. \tag{5.4}$$

The training procedure for this architecture is nearly identical to the training procedure for the standard modular architecture. The weights of the share network, like the weights of the expert networks, are modified so to reduce the sum of squared error between the desired and actual outputs of the architecture (Equation 3.2). The training of the gating network is slightly different in the two varieties of modular architectures. Recall that for the standard modular architecture, the winner of the competition among the expert networks is defined as the network whose output most closely approximates the desired output of the system. For the modular architecture with a share network, the winning expert network is defined as the network whose output most closely approximates the difference between the desired output of the system and the output of the share network. Define the error for expert network i to be

$$J_{\mathbf{y}_i} = \frac{1}{2}(\mathbf{y}^* - \mathbf{y}_s - \mathbf{y}_i)^T(\mathbf{y}^* - \mathbf{y}_s - \mathbf{y}_i). \tag{5.5}$$

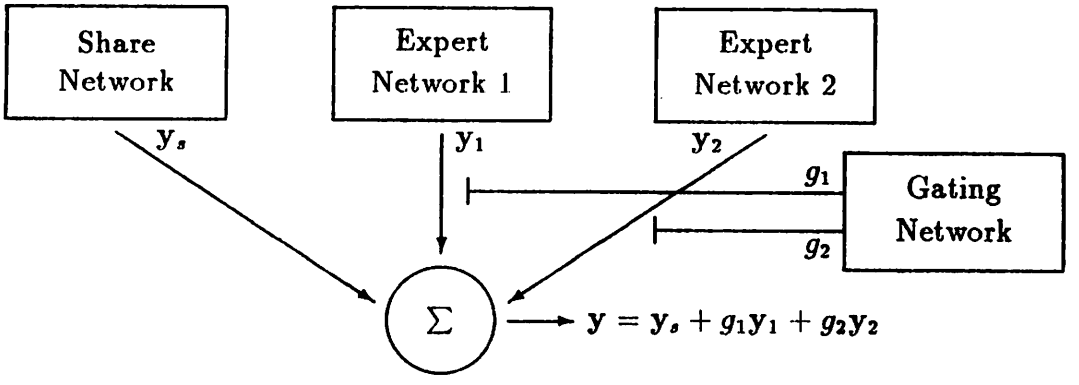

Figure 5.3: A modular architecture with a share network.

The expert network with the smallest error is the winner.

In regard to the problem of controlling a robot arm to move a variety of payloads along a specified trajectory, there are many possible decompositions of this problem into a shared strategy and a set of modifications to this strategy. The decomposition that we are interested in defines the shared strategy to be the application of the correct feedforward torques to control the arm with no payload along the desired trajectory. The modifications to this strategy are to add the extra torques required to compensate for the mass of the payload.

Systems MAS-L and MAS-NL have one share network, three expert networks, and one gating network (see Table 5.3). In each system, the first expert network doesn't contain any input units and contains two output units whose activations are always zero. For this reason, this expert network is referred to as the null expert network. When designing these systems, it was hoped that the systems are suitably structured so that the share network and the null expert network would learn to control the

arm with no payload, the share network and another expert network would learn to control the arm with a light payload, and the share network and the remaining expert network would learn to control the arm with a heavy payload. The null expert network is needed because we want the share network alone to control the arm with no payload, but the gating network's error function (Equation 3.6) specifies that, in response to each input pattern, one of the expert networks should be "switched on" and all other expert networks should be "switched off". Therefore, we provided the null expert network in the hope that this expert network would be "switched on" in the absence of a payload and, thus, the share network alone would learn to control the arm in this situation.

The learning curves for the systems trained with the input vectors from set L are shown in Figure 5.4. The horizontal axis gives the number of epochs. The vertical axis gives the joint root mean squared error (RMSE) in radians averaged over 25 runs. [2] For many parameters of each system (e.g., step size and momentum), we used the values that appeared to give the best performance. These values and

[2]This joint RMSE was computed as follows. At each time step, the squared differences between the desired and actual joint positions was determined. The joint RMSE was determined for each traversal of the trajectory by averaging the squared differences over all 200 time steps of the traversal, and then taking the square root of this average. The joint RMSE for each epoch was determined by averaging the joint RMSE of the 30 traversals (6 payloads times 5 traversals per payload) that occurred in each epoch. The joint RMSE shown in Figure 5.4 was determined by averaging the joint RMSE of each epoch over the 25 runs.

additional details of the simulations are provided in Appendix B. Clearly, the two modular architectures MA–L and MAS–L performed better than the single network N–L. This superior performance is due to the fact that the multi–payload robotics task is characterized by a high degree of temporal crosstalk. Each system attempted to model the inverse dynamics of the robot arm with a payload that is fixed within each bin of 1000 consecutive time steps, but varied between bins. As shown in Chapter 4, the rate of learning of single networks is often diminished by temporal crosstalk, whereas modular architectures tend to be robust in the presence of temporal crosstalk. The performances of the two modular architectures were approximately equal.

Figures 5.5 and 5.6 show how the modular architecture MA–L and the modular architecture with a share network MAS–L allocated their expert networks to the various payloads. Corresponding to each allocation is a bar, and the horizontal axis numbers the different allocations. For example, 25 runs of system MA–L resulted in 5 different allocations of expert networks to payloads and, thus, there are five different bars in the bar graph shown in Figure 5.5. The allocation that each bar corresponds to is given by the arrangement of the six numbers above the bar where the numbers are the masses in kilograms of the six payloads. Payloads that were allocated to the same expert network have their masses bracketed together. The vertical axis gives the number of runs in which an architecture allocated its expert networks in the corresponding manner. For example, the leftmost bar of Figure 5.5 shows that on 12 of the 25 runs, one expert network of system MA–L won the competition to control the arm when the payload was 0 or 2 kg, another expert network won the

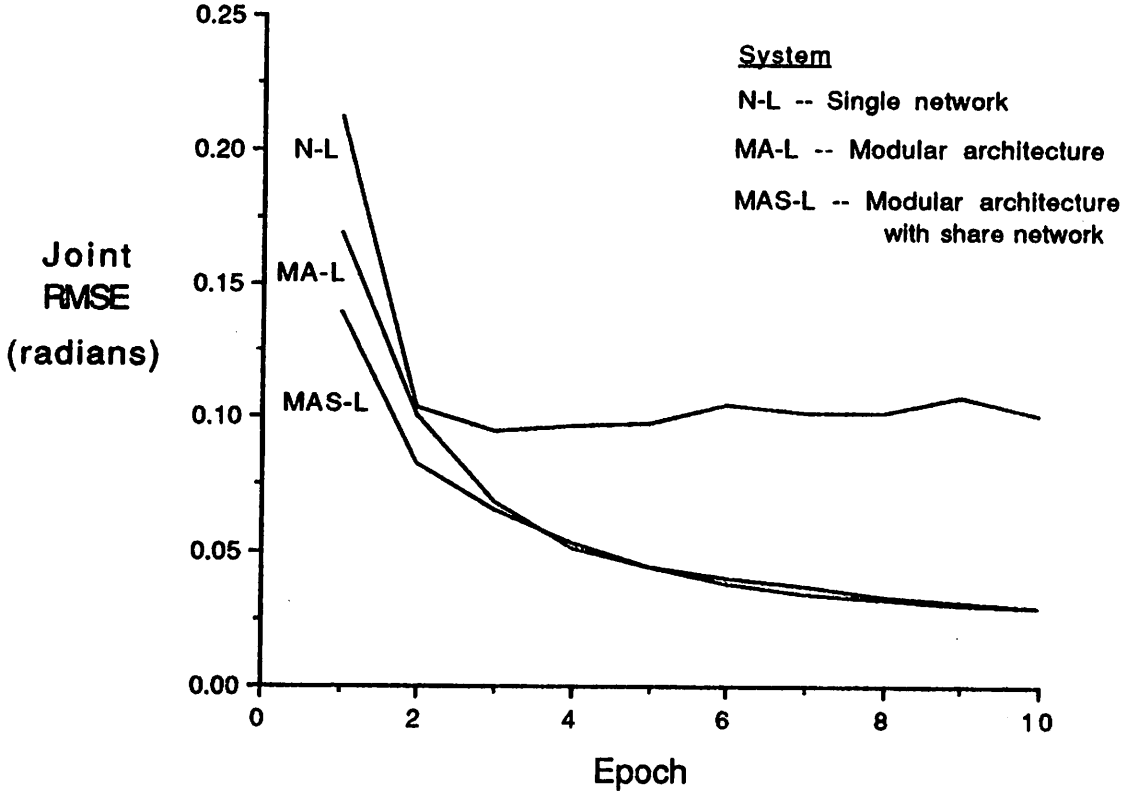

Figure 5.4: Learning curves for the systems trained with the input vectors from set L.

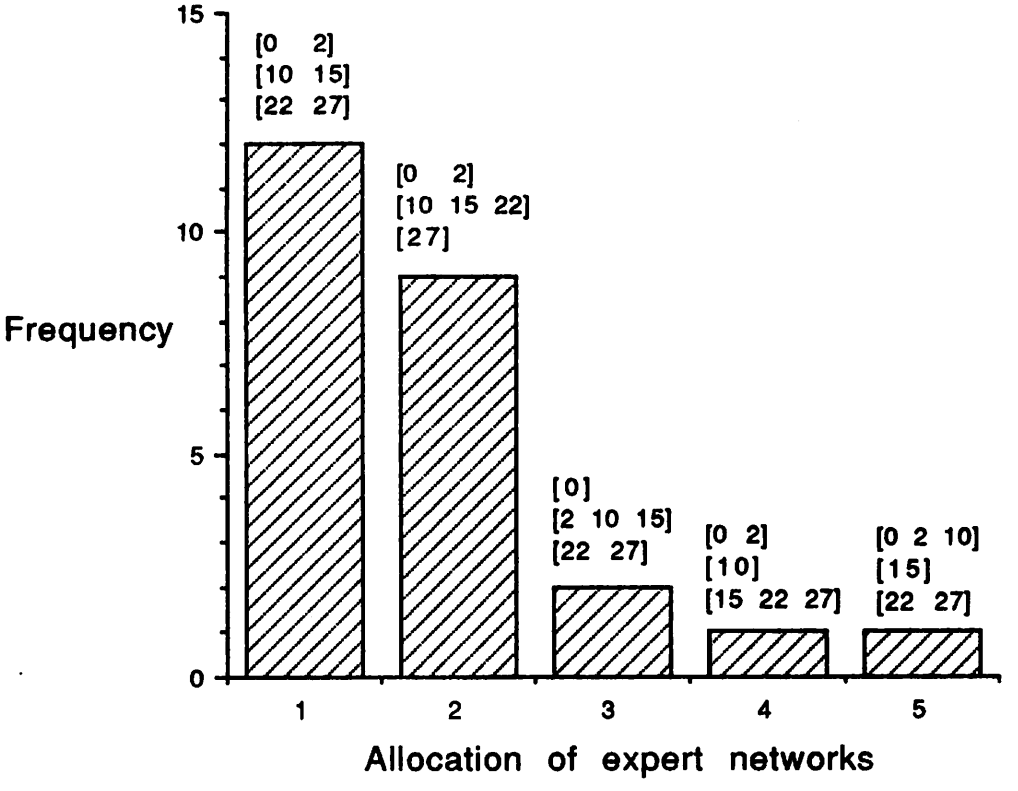

Figure 5.5: Allocation of modular architecture MA–L's expert networks to payloads.

competition when the payload was 10 or 15 kg, and the remaining expert network won the competition when the payload was 22 or 27 kg.

Figure 5.5 shows that the modular architecture MA–L learned to control the robot arm with different payloads by allocating different expert networks to control the arm with payloads from different mass categories. One expert network tended to win the competition to learn to control the arm when there was no payload or a very light payload (0 or 2 kg), another expert network tended to win the competition when

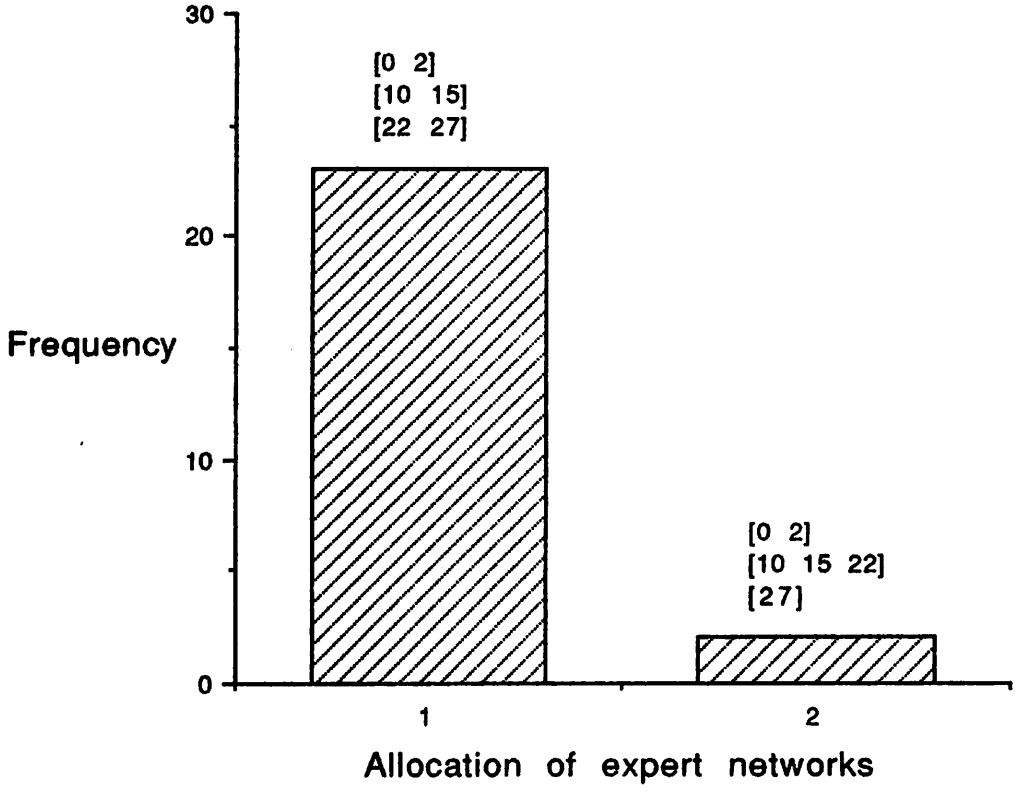

Figure 5.6: Allocation of modular architecture with a share network MAS–L's expert networks to payloads.

the payload was light (10 or 15 kg), and a third expert network tended to win the competition when the payload was heavy (22 or 27 kg). This occurred despite the fact that the system MA–L was only provided with the identity of each payload, not the mass of a payload. The tendency to allocate the same expert network to control the arm with payloads of similar masses, and to allocate different expert networks to control the arm with payloads of dissimilar masses is an outcome of the competitive process. If, for example, an expert network wins the competition to learn to control the arm with no payload, then it is likely to also win the competition when the payload is 2 kg, but lose the competition when the payload is 27 kg. This result suggests that a suitably designed modular architecture tends to allocate the same expert network to similar tasks, resulting in the benefits of positive transfer of training, and different expert networks to dissimilar tasks, thereby avoiding the detrimental effects of negative transfer of training.

As is shown in Figure 5.6, the modular architecture with a share network MAS–L allocated its expert networks to payloads in a manner similar to that of system MA–L. In addition to the torques specified by the share network, one expert network tended to specify extra torques when there was no payload or a very light payload (0 or 2 kg), another expert network tended to specify extra torques when the payload was light (10 or 15 kg), and a third expert network tended to specify extra torques when the payload was heavy (22 or 27 kg). This occurred despite the fact that the system MAS–L was only provided with the identity of each payload, not the mass of a payload. A fact not shown in Figure 5.6 is that the modular architecture with a share network

always discovered the desired task decomposition as defined above. Specifically, this architecture always learned to use the share network and the null expert network to control the robot arm with no payload or a very light payload. The remaining expert networks learned to specify the extra torques required to compensate for the mass of the payload. One expert network specified extra torques when the payload was light, and another expert network specified extra torques when the payload was heavy. This result suggests that a suitably designed modular architecture with a share network tends to learn to perform a task by learning a shared strategy that is used in all contexts along with a set of modifications to this strategy that are applied in a context sensitive manner.

Figure 5.7 shows the learning curves for the systems trained with the input vectors from set NL. The single network N–NL learned faster than the modular architecture MA–NL and the modular architecture with a share network MAS–NL. The relatively slow rates of learning of systems MA–NL and MAS–NL are due to their poor allocation of expert networks to payloads. These allocations are shown in Figures 5.8 and 5.9. Systems MA–NL and MAS–NL did not always allocate different expert networks to control the arm with payloads from different mass categories. These systems often used the control law learned by one of the expert networks to control the arm with payloads of very different masses. Learning is slow because this expert network received conflicting training information when the arm carried payloads of different masses. Because the expert network was not provided with the identity of a payload, the knowledge that it gained about the inverse–dynamics of the arm with one pay-

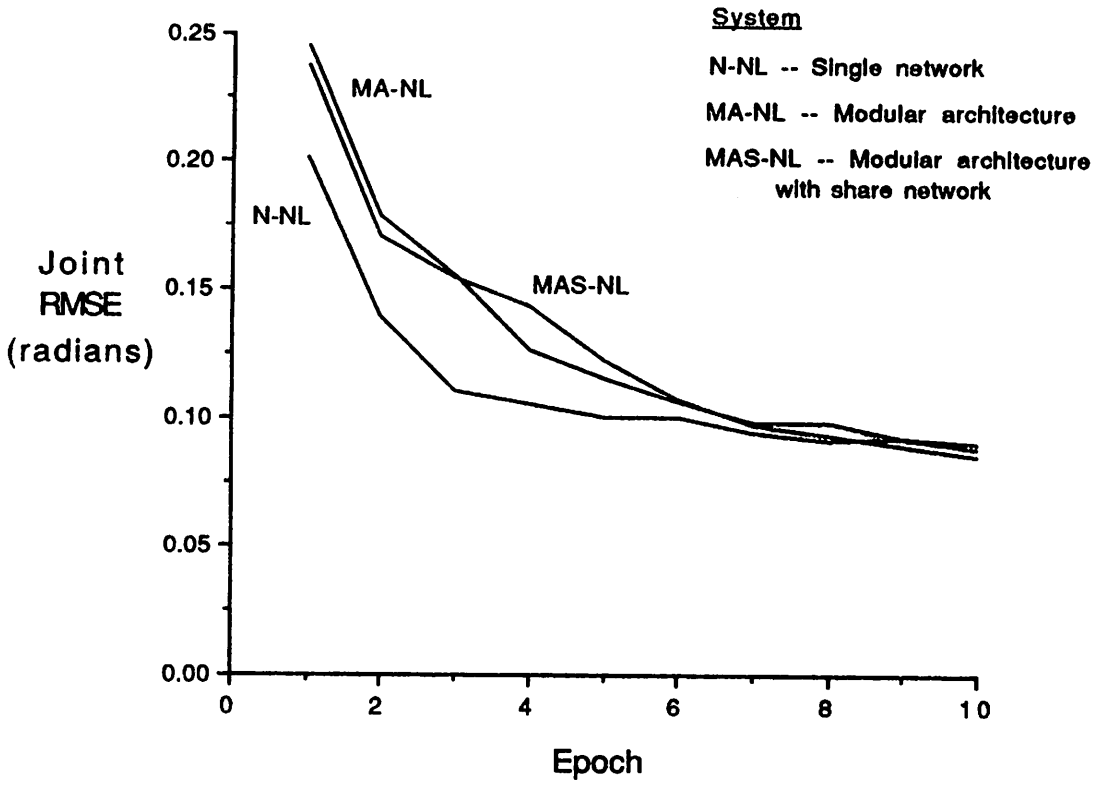

Figure 5.7: Learning curves for the systems trained with the input vectors from set NL.

load was applied without modification to the control of the arm with a payload of a significantly different mass.

Figures 5.5, 5.6, 5.8, and 5.9 show that, in contrast to the modular architectures MA–L and MAS–L, the modular architectures MA–NL and MAS–NL did not allocate their expert networks to the payloads in an appropriate manner. Due to the input vectors they received and the fact that their expert networks do not contain hidden

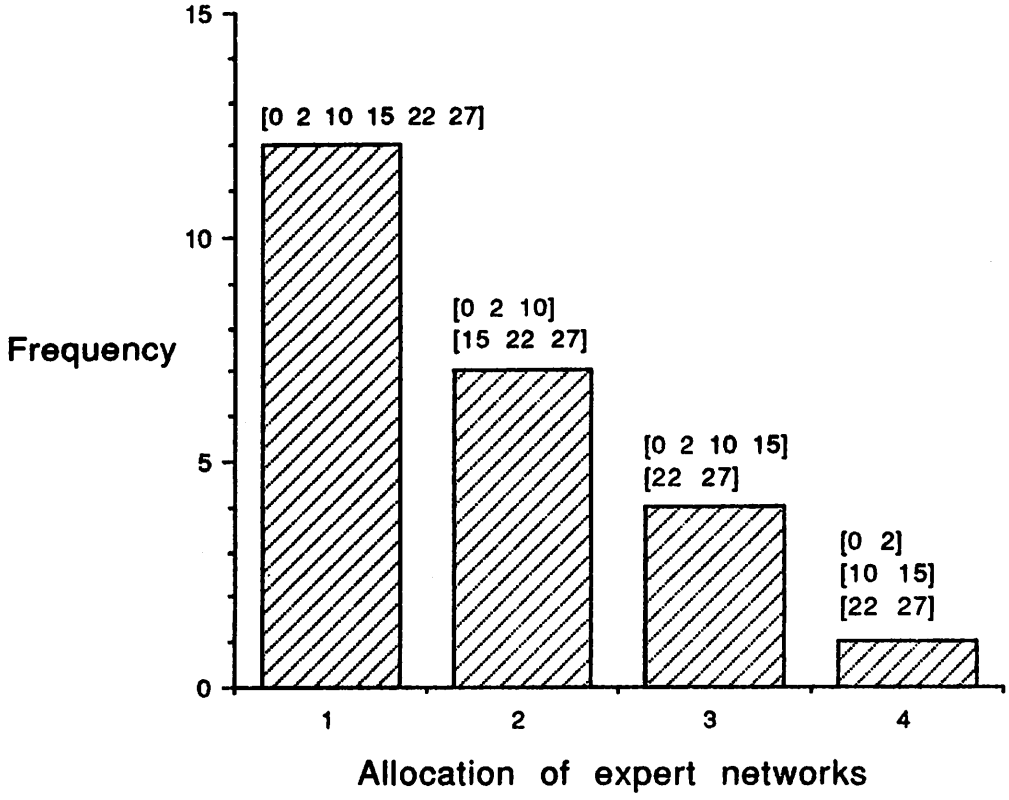

Figure 5.8: Allocation of modular architecture MA–NL's expert networks to payloads.

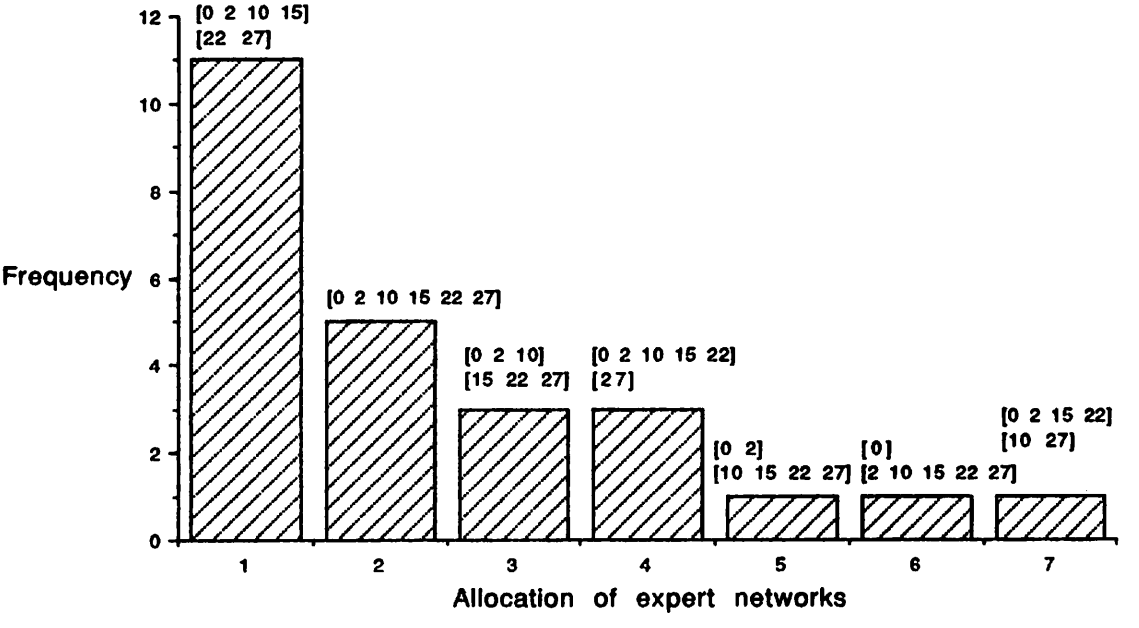

Figure 5.9: Allocation of modular architecture with a share network MAS–NL's expert networks to payloads.

units, the former systems are more constrained in the types of functions that they can compute than the latter systems and, consequently, are more constrained in the types of task decompositions that they can discover. These simulations show that there are many possible decompositions of a task into simpler subtasks. If there are reasons to prefer one decomposition over another, then it is necessary to use domain knowledge in order to design a modular architecture that is appropriately restricted in the types of functions it can compute. The relationship between the design of a modular architecture and the task decomposition that it discovers is an important topic and a complete discussion is postponed until Chapter 6.

5.3 Utilization of Domain Knowledge

In the simulations reported in Section 5.2, the connectionist systems were trained to control the robot arm with various payloads when the systems were only provided with the identity of each payload. In this section, we report simulations in which a system was provided with more detailed information concerning the payload. The system is the modular architecture with a share network MAS–L. Our goal is to demonstrate the ease with which a priori knowledge of the domain may be embedded in modular connectionist architectures.

Three simulations are reported corresponding to three uses of domain knowledge in the training of system MAS–L. In all cases, the modular architecture with a share network learned to allocate its networks in an identical fashion. After training, the

Table 5.4: Three simulations that make use of a priori knowledge of the domain.

Simulation	Stage	Training
1		Assume can detect the absence of a payload.
	1	Train share network using no payload.
	2	Train expert and gating networks as in Section 5.2.
2		Assume can detect the absence of a payload. Assume known sample light and heavy payloads.
	1	Train share network using no payload.
	2	Train second expert network using light payload.
	3	Train third expert network using heavy payload.
	4	Train gating network as in Section 5.2.
3		Assume can measure the mass of a payload.
	1	Set weights of gating network.
	2	Train share and expert networks as in Section 5.2.

share network and the null expert network controlled the arm with no payload or with an extremely light payload; the share network and either the second or third expert network controlled the arm with a light payload; and the share network and the remaining expert network controlled the arm with a heavy payload. A description of the three simulations is given below and a summary is provided in Table 5.4. Each simulation is divided into a number of stages. The learning curve for each simulation during the last stage of training is shown in Figure 5.10.

In simulation 1, we assumed that it is possible to detect when the arm has no payload (e.g., the gripper of the arm may be open). The training of the architecture

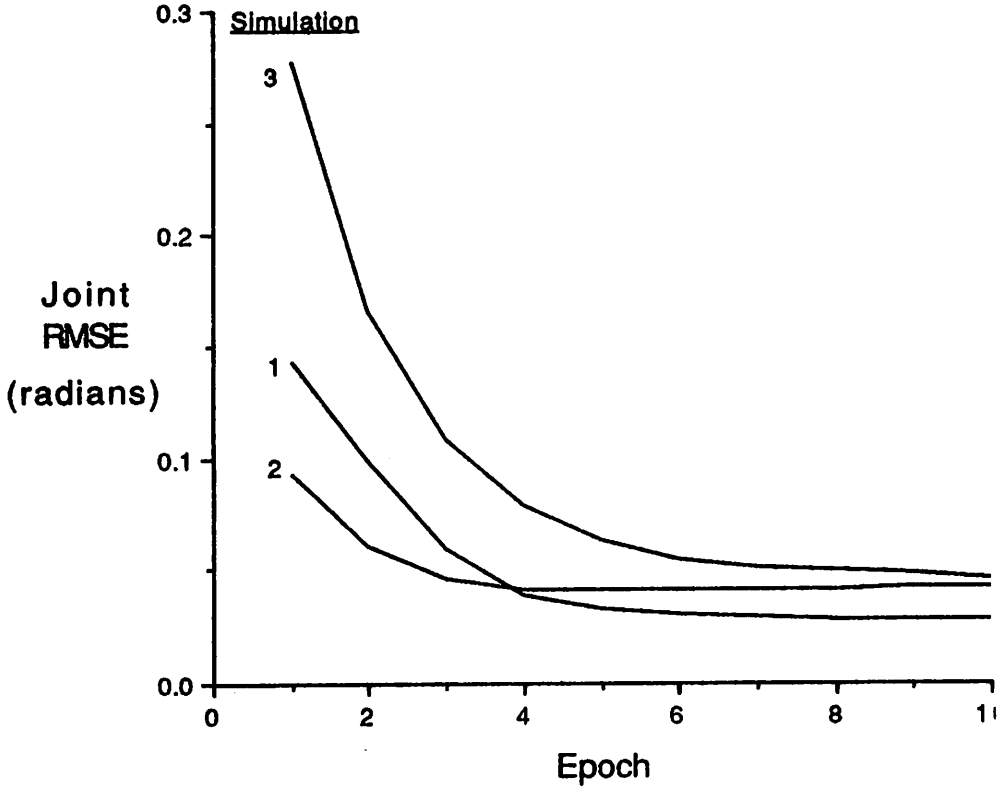

Figure 5.10: Learning curves for the three simulations during the last stage of training.

is divided into two stages. In the first stage, we trained only the share network by requiring it to direct the arm with no payload 50 times along the desired trajectory. After training, the step size of the share network was set to zero. In the second stage, the expert and gating networks were trained as in Section 5.2 to move the arm with a variety of payloads, each of a different mass, along the desired trajectory.

In addition to assuming that it is possible to detect when the arm has no payload, in simulation 2 it was also assumed that we know a sample light payload (12.5 kg) and a sample heavy payload (25 kg). The training of the architecture is divided into four stages corresponding to the training of the share network, the second expert network, the third expert network, and the gating network respectively. At the end of each stage, the step size of the network trained during that stage was set to zero. In the first stage, the share network was trained by requiring it to direct the arm with no payload 50 times along the desired trajectory. In the second stage, the second expert network was trained by requiring it and the share network to direct the arm with the sample light payload 50 times along the desired trajectory. In the third stage, the third expert network was trained by requiring it and the share network to direct the arm with the sample heavy payload 50 times along the desired trajectory. Finally, in the fourth stage, the gating network was trained as in Section 5.2 so that the system could control the arm with a variety of payloads.

In simulation 3, we assumed that the mass of a payload can be measured. The training of the architecture is divided into two stages. In the first stage, the weights of the gating network are hand set so that, in response to the input vector identifying the

payload, the output unit corresponding to the appropriate expert network assumed a value of one and all other output units assumed a value of zero. Specifically, when the robot arm didn't contain a payload or contained a very light payload, g_1 was set to one; when the payload was light, g_2 was set to one; and when the payload was heavy, g_3 was set to one. In the second stage of training, the share and expert networks were trained as in Section 5.2 so that the system could control the arm with a variety of payloads.

These simulations show that if there is sufficient knowledge of a task, some or all of the expert and gating networks can be individually trained independently of the rest of the modular architecture (Hampshire and Waibel [26]). This is one of many methods for embedding domain knowledge in a modular connectionist architecture. Chapter 6 lists other methods for utilizing domain knowledge and argues that a strength of the modular architecture is that its structure is well-suited for incorporating such knowledge.

In summary, this chapter has reported the results of training connectionist systems to control a robot arm to move a variety of payloads, each of a different mass, along a desired trajectory. These results suggest that a suitably designed modular architecture or modular architecture with a share network tends to allocate the same expert network to similar tasks, resulting in the benefits of positive transfer of training, and different expert networks to dissimilar tasks, thereby avoiding the detrimental effects of negative transfer of training. The results also suggest that a suitably designed modular architecture with a share network tends to learn to perform a task by learning a

shared strategy that is used in all contexts along with a set of modifications to this strategy that are applied in a context sensitive manner. Finally, the chapter reported simulations demonstrating that if there is sufficient knowledge of a task, some or all of the expert and gating networks can be individually trained independently of the rest of the modular architecture.

CHAPTER 6

TASK DECOMPOSITION AND NETWORK ARCHITECTURES

The previous two chapters reported the ability of the modular architecture to perform task decomposition on the "what" and "where" vision tasks and on the multi-payload robotics task. This chapter considers some domain-independent issues concerning task decomposition and discusses similarities between task decomposition and generalization as implemented in connectionist systems.

Because function approximation is an underconstrained problem, networks with too many degrees of freedom may not generalize as desired (e.g., Denker et al. [16], le Cun [42], Poggio and Girosi [58]). One approach to this problem is to use domain knowledge to design a network architecture that is appropriately restricted in the types of functions that it can implement. Such an architecture should generalize in the desired manner. Experience with the modular architecture described here has shown that a similar situation exists with regard to task decomposition. Because function decomposition is an underconstrained problem, there are many possible de-

compositions of a task into simpler tasks. Thus, modular architectures with too many degrees of freedom may not decompose a task as desired. Section 6.1 reports simulations demonstrating this idea. Section 6.2 argues that if there are reasons to prefer one decomposition over another, then it is necessary to use domain knowledge in order to design a modular architecture that is appropriately restricted in the types of functions it can compute. Such an architecture should decompose a task in the desired manner.

6.1 Simulations

Three systems were trained to approximate the function

$$f(x) = \begin{cases} \sin x & \text{if } x \in [-2\pi, 0] \\ \sin x + \sin 2x & \text{if } x \in (0, 2\pi]. \end{cases} \tag{6.1}$$

Figure 6.1 shows the graph of this function. At each time step, x was randomly selected from a uniform distribution over the interval $[-2\pi, 2\pi]$. As discussed below, the three systems are modular architectures, and are numbered such that higher-numbered systems are less restricted in the types of functions that they can compute than lower-numbered systems. All systems consist of two expert networks and a gating network (see Table 6.1). The gating network is a single-layer network with 71 input units and 2 output units. The input units provide a coarse-coded representation of x. Details concerning this representation and several other aspects of the simulations are provided in Appendix C.

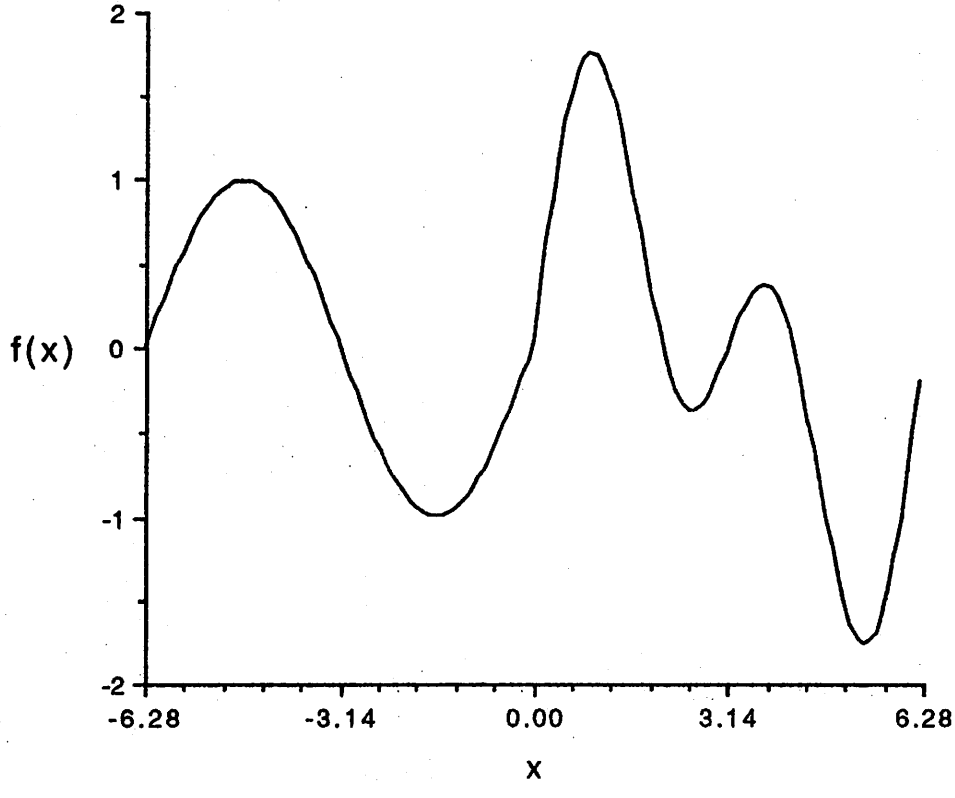

Figure 6.1: Graph of f.

Table 6.1: Systems used in the sine wave experiments.

Modular Architectures				
System	Expert Networks	Expert Networks' Input	Gating Network	Gating Network's Input
1	$2 \rightarrow 1$ $2 \rightarrow 1$	$\sin x$ and $\sin 2x$ $\sin x$ and $\sin 2x$	$71 \rightarrow 2$	coarse–code of x
2	$1 \rightarrow 10 \rightarrow 10 \rightarrow 1$ $2 \rightarrow 1$	x $\sin x$ and $\sin 2x$	$71 \rightarrow 2$	coarse–code of x
3	$1 \rightarrow 10 \rightarrow 10 \rightarrow 1$ $1 \rightarrow 10 \rightarrow 10 \rightarrow 1$	x x	$71 \rightarrow 2$	coarse–code of x

In system 1, each expert network is a single–layer network with 2 input units and 1 output unit. The activations of the input units are assigned the values $\sin x$ and $\sin 2x$ respectively. Figure 6.2 shows how system 1 allocated its expert networks at the end of a typical simulation. The horizontal axis gives the values of x. The vertical axis gives the activation values of the output units of the gating network. This graph shows that, roughly, the system allocated one expert network to approximate f when x was negative and the other expert network to approximate f when x was positive. Given the piecewise nature of f, the system discovered an appropriate decomposition. Because of the restrictions placed on the expert networks (they can only compute affine functions of $\sin x$ and $\sin 2x$), this was the only decomposition that the system could use to closely approximate f.

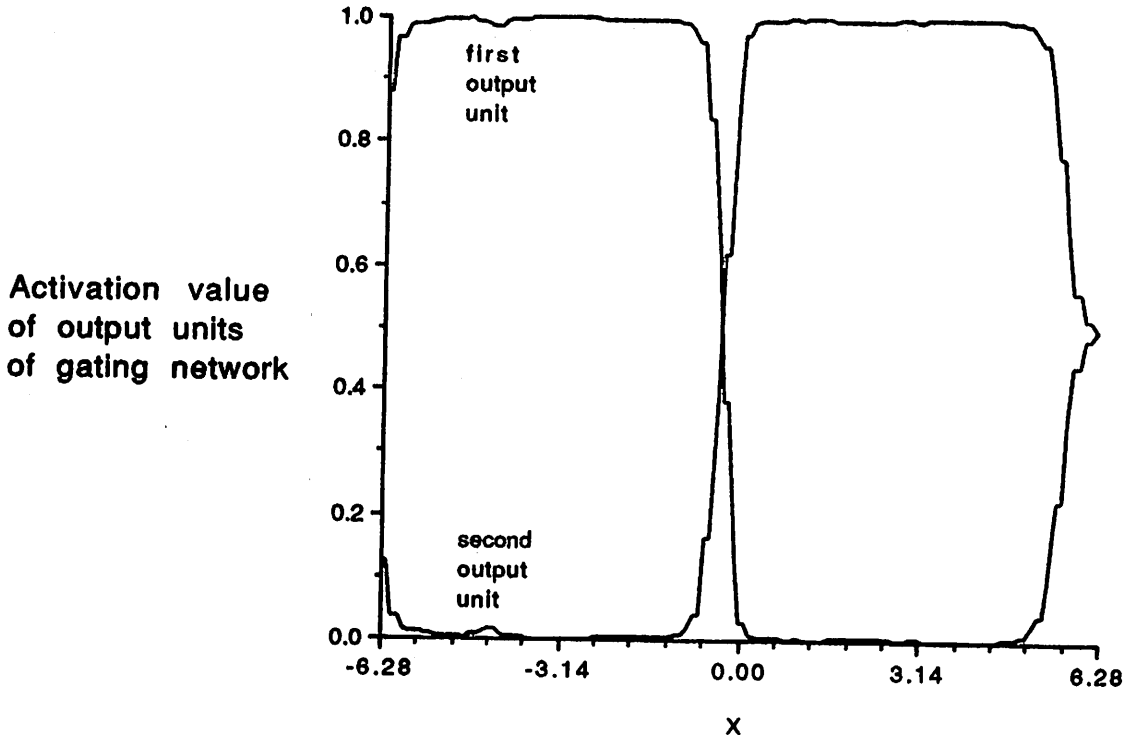

Figure 6.2: Task decomposition discovered by System 1.

Figure 6.2 contains two other noteworthy features. When $x \approx 0$, the outputs of the system were roughly the average of the outputs of the expert networks. This seems appropriate since $x \approx 0$ lies on the borderline that separates the "domains of expertise" of the expert networks. A second feature is that when x was near -2π or 2π, the activations of the output units of the gating network were near 0.5. This is a side–effect of our use of a coarse–coded representation of x and can be eliminated by further training with values of x near these endpoints.

In system 2, the first expert network is a strictly layered network with 1 input unit, 2 layers of 10 hidden units each, and 1 output unit. The input unit encodes the value of x. The second expert network is a single–layer network with 2 input units and 1 output unit. The input units encode the values $\sin x$ and $\sin 2x$. System 2 is less restricted than system 1 in the types of functions it can compute. Consequently, it is more difficult for system 2 to discover a decomposition that allows it to closely approximate f. We find that in order for this system to discover an appropriate decomposition, it is necessary to vary the contribution to the gating network error function (Equation 3.6) of the term responsible for making the outputs of the gating network approach the neutral value, $\frac{1}{n}$, where n is the number of expert networks. This term, the fourth in Equation 3.6, contains the factor λ_{NT} which is non–zero only when the architecture's performance has not significantly improved. Instead of setting λ_{NT} to one when performance has not significantly improved, we initialized it to a value larger than one at the start of training and slowly decreased it to one during training. Recall that this policy of varying λ_{NT} was introduced in

Section 4.1.2, and was found to allow expert networks unable to compete in terms of initial rates of learning—but which may be better in terms of eventual performance—the chance to exert their superiority. When λ_{NT} was modified in this manner, the system consistently discovered an appropriate decomposition.

Figure 6.3 shows how system 2 allocated its expert networks at the end of a typical simulation. This graph shows that the system allocated the multi–layer expert network to approximate f when x was negative and the single–layer expert network to approximate f when x was positive. Although there are an infinite number of task decompositions that would allow the system to closely approximate f, the combination of the following three properties ensured that the system always discovered the decomposition shown in Figure 6.3. These properties are that the single–layer expert network can only compute affine functions of $\sin x$ and $\sin 2x$; the single–layer expert network can learn such affine functions faster than the multi–layer expert network; and it is easier for the multi–layer expert network to learn the function $f(x) = \sin x$ than the function $f(x) = \sin x + \sin 2x$.

In system 3, each expert network is a strictly layered network with 1 input unit, 2 layers of 10 hidden units each, and 1 output unit. The activation of each input unit is assigned the value of x. System 3 is less restricted than either system 1 or 2 in the types of functions that it can compute. As a result, it never discovered the decomposition of f that was discovered by the first two systems. Figure 6.4 shows how system 3 allocated its expert networks at the end of a typical simulation. A comparison of Figures 6.1 and 6.4 shows that, roughly, system 3 allocated one

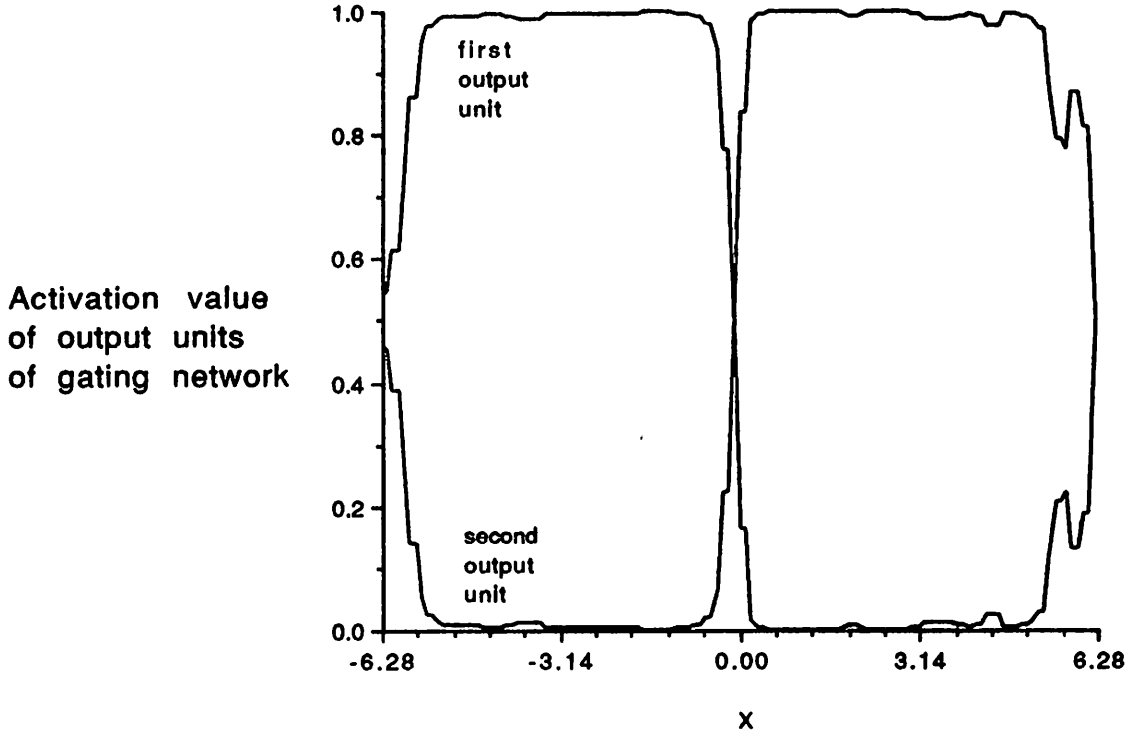

Figure 6.3: Task decomposition discovered by System 2.

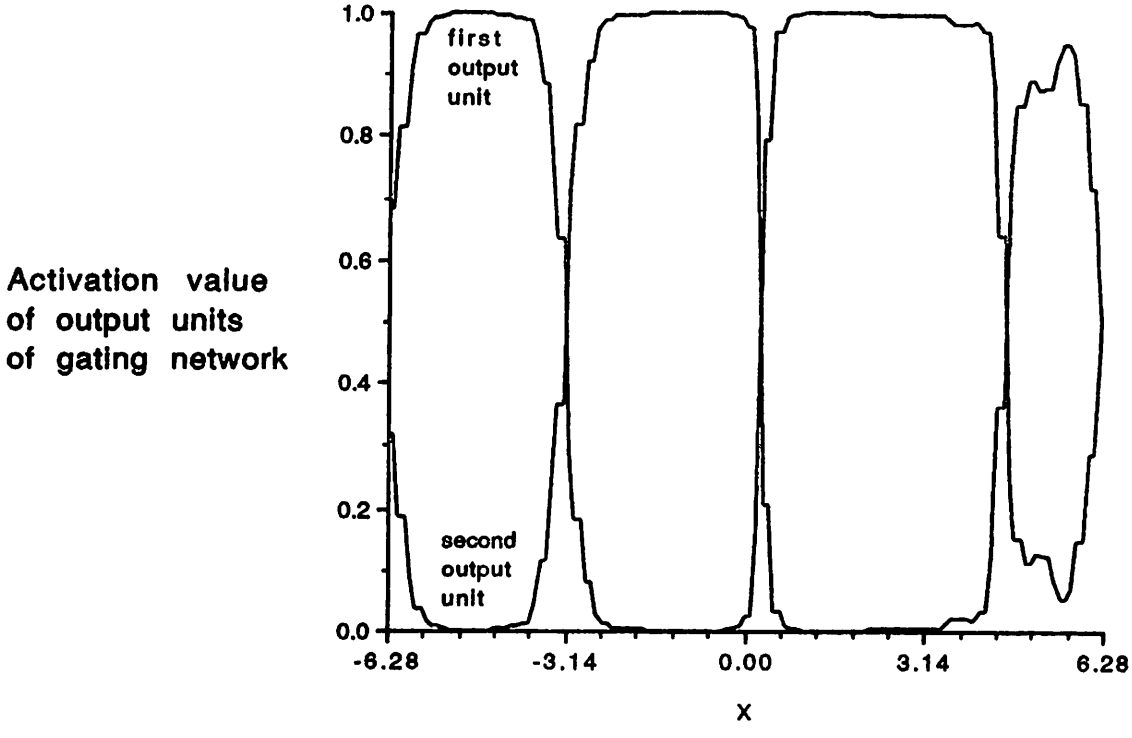

Figure 6.4: Task decomposition discovered by System 3.

expert network to approximate f where f is concave up and allocated the other expert network to approximate f where f is concave down. Although there are an infinite number of task decompositions that would allow the system to closely approximate f, we find that it frequently, though not always, decomposed f in this manner. Considering that there are no constraints on which decomposition the system may discover, this result is somewhat surprising.

As mentioned above, in order to obtain a network that generalizes in a desired manner, it is necessary to design a network architecture that is appropriately restricted in the types of functions it can compute. As the above experiments demon-

strate, a similar situation exists with regard to task decomposition. In order to obtain a modular architecture that decomposes a task in a desired manner, it is necessary to design the gating and expert networks so that the architecture is appropriately restricted in the types of functions it can compute.

6.2 Design of Modular Architectures

Designing an appropriate architecture, whether for generalization or task decomposition, requires prior knowledge about the task the system will be required to perform. A strength of modular architectures is that their structures are well-suited for inserting prior knowledge to bias the decompositions to be formed. The experiments reported in this thesis have highlighted several ways in which such knowledge can be utilized. For example, in many tasks there is a natural distinction between information to be processed and information that sets the context for processing. The experimenter may know that when one or more of the (context) inputs remain constant, the mapping from the remaining inputs to the desired outputs is relatively easy to compute. The distinction between the context inputs and the remaining inputs can form the basis for deciding how to divide input information between gating networks and expert networks.

Another way prior knowledge can be incorporated into the design of a modular architecture is that known properties of the function to be approximated can provide constraints on the design of the expert networks. The design of the expert networks, in

turn, biases the nature of the decomposition discovered by the modular architecture. This is particularly true when the repertoire of expert networks consists of networks with different characteristics. For example, the function to be approximated may be known to contain a linear portion and a nonlinear portion. In this case, suitable expert networks are easy to design. In general, different expert networks may be designed to possess different topologies, initial weights, activation functions, step sizes, error functions, etc. In addition, different expert networks may receive different input variables or perhaps different representations of the same input variables. Note that we do not advocate providing the modular architecture with a large number of different expert networks. Rather, the experimenter should judiciously design a small set of potentially useful expert networks where the potential utility of an expert network is evaluated using domain knowledge. Indeed, if there is sufficient knowledge of the task, some or all of the expert and gating networks can be individually trained independently of the rest of the architecture (Hampshire and Waibel [26]).

CHAPTER 7

CONCLUSIONS

A novel modular connectionist architecture has been presented in which the networks composing the architecture compete to learn the training patterns. As a result of the competition, different networks learn different training patterns and, thus, learn to compute different functions. The architecture performs task decomposition in the sense that it learns to partition a task into two or more functionally independent tasks and allocates distinct networks to learn each task. In addition, the architecture tends to allocate to each task the network whose topology is most appropriate to that task, and tends to allocate the same network to similar tasks and distinct networks to dissimilar tasks. Furthermore, it can be easily modified so as to learn to perform a family of tasks by using one network to learn a shared strategy that is used in all contexts along with other networks that learn modifications to this strategy that are applied in a context sensitive manner. These properties were demonstrated by applying the architecture to two sets of tasks. Chapter 4 reported the results of training the architecture to perform object recognition and spatial localization from simulated

retinal images. Chapter 5 reported the results of training the architecture to control a simulated robot arm to move a variety of payloads, each of a different mass, along a specified trajectory. Finally, in Chapter 6, it was noted that function decomposition is an underconstrained problem and, thus, different modular architectures may decompose a function in different ways. We argued that a desirable decomposition can be achieved if the architecture is suitably restricted in the types of functions that it can compute. Finding appropriate restrictions is possible through the application of domain knowledge. A strength of the modular architecture is that its structure is well–suited for incorporating domain knowledge.

7.1 Future Work

We conclude this thesis with a list of suggestions for future research using the the modular connectionist architecture. We warn the reader in advance that whereas some of the suggestions are well–formulated, others are highly speculative.

Applications—Although the modular architecture has potential utility in many application areas, we limit our discussion to its use in control tasks. Adaptive control engineers frequently consider control tasks for which it is difficult to design a continuous control law for all regions of a plant's state space. However, these tasks can often be performed by piecewise controllers that employ different control laws when the plant is operating in different regions of its state space (e.g., gain scheduling is a popular piecewise control technique). The success of these controllers in the control of

nonlinear systems suggests that task decomposition can be profitably utilized in the performance of these tasks. Control engineers typically design piecewise controllers by first identifying a suitable partition of the state space of the plant, and then designing a useful control law for each region of the state space. One proposed research program would be to go beyond the robotics experiments presented in Chapter 5 by using the modular architecture on more difficult control tasks and comparing the piecewise control laws learned by the architecture with piecewise control laws suggested by adaptive control theory. An advantage of the modular architecture over conventional piecewise controllers is that it can learn both a useful partition of a plant's state space as well as a useful control law for each region of the state space. Therefore, the modular architecture may be applicable to control tasks for which engineers do not possess sufficient a priori knowledge to design an effective piecewise control law.

Connectionist learning theory—A set of studies could investigate the relationship between the design of the modular architecture and the task decomposition discovered by the architecture. These studies would address a number of important issues.

- This thesis has emphasized that the task decomposition discovered by a modular architecture is, in part, determined by the design of the architecture. A second factor determining the task decomposition discovered by the architecture is the order in which training patterns are presented. A proposed research program would attempt to enhance the modular architecture's ability to take advantage of positive transfer of training from earlier to later training patterns. A suitably

designed modular architecture may discover task decompositions in which the subtasks identified by the decomposition are also subtasks of other tasks faced by the architecture. This objective emphasizes that the functions learned by the expert networks can be thought of as "building blocks" to be used on other occasions in the performance of more complex tasks.

- The modular architecture both allocates different expert networks to learn different functions as well as combines the expert networks' outputs to determine the output of the system. A proposed research program would decouple these two operations by performing them in different stages of processing. In the first stage, the modular architecture decomposes a task into a set of useful subtasks. A second stage of processing uses a different architecture to combine the expert networks' outputs. For example, many computer vision systems have an early stage of processing during which different modules compute different functions of the visual input. These modules may detect edges or estimate depth, orientation, optic flow, color, etc. In order to interpret the visual image, the modules' outputs are nonlinearly combined during subsequent stages of processing. Given that the modular architecture has similarities with conventional competitive learning systems (see Chapter 3), the use of the modular architecture as a preprocessor may be considered analogous to the use of competitive learning during preprocessing. For example, Moody and Darken [50] trained a network to perform a nonlinear mapping by initially training the hidden units of the network using a competitive learning technique, and subsequently training

the output units to combine the outputs of the hidden units using a supervised learning technique.

- In this thesis the expert and gating networks of the modular architecture are trained using different error functions. However, there may be practical and theoretical advantages to training both sets of networks using a single error function. Independently, Geoffrey Hinton (Hinton, personal communication) and Michael Jordan (Jordan, personal communication) developed similar proposals for error functions for training both the expert and gating networks. These error functions are based on the intuition that a competitive learning process analogous to "soft" competitive learning (Bridle [12], Nowlan [54]) may be useful in training all networks of the modular architecture. [1]

- This thesis has described the standard modular connectionist architecture (Figure 3.1) and two modifications of this architecture, namely the modular architecture with multiple gating networks (Figure 4.5) and the modular architecture with a share network (Figure 5.3). Different versions of the architecture may learn different decompositions of the same task. Consequently, different versions are most useful for different tasks. A proposed research program would

[1] The investigation of a single error function for all networks of the modular architecture is currently being pursued in a collaboration between Geoffrey Hinton, Michael Jordan, Steven Nowlan, and the author.

identify other architectural modifications along with the class of tasks for which each modification is most suited.

The notion of modularity has been found to be of considerable utility in cognitive science, particularly in the study of language and vision. Not only have modular theories been found to be more parsimonious and easier to understand than nonmodular theories, but also the predictions of modular theories have in many cases been verified (Freedman and Forster [23]). Modularity is also indispensable in the design and analysis of complex systems in engineering. We feel that the virtues of modular systems cited in the literature of these areas are also relevant to the problem of learning in connectionist networks. In particular, we have argued that if a task can be decomposed into subtasks, each of which has its own idiosyncratic properties, then the learner should itself be a decomposable system in which distinct system resources ("experts") are allocated to distinct subtasks. Such a learning system will in general be more robust, more efficient, and will generalize better than a nonmodular system.

Although domain knowledge may be useful in suggesting an a priori decomposition of a task, the boundaries between subtasks are rarely explicitly marked in the data presented to the learner. Moreover, the optimal allocation of experts to subtasks depends not only on the nature of the task but also on the nature of the learner. For these reasons we have argued that the problem of allocating experts to subtasks is itself part of the learning problem. Even if domain knowledge is used in designing the initial structure of the modular architecture, it is still necessary for the system to discover which experts to assign to which training instances.

The modular architecture presented in this thesis makes use of competition to induce a task decomposition. The competition allows experts to specialize as well as to extend their applicability. We feel that the competition between experts is the essential feature of the approach presented here and should serve as a useful point of departure for the further development of algorithms for learning in modular systems.

APPENDIX A

SIMULATION DETAILS FOR CHAPTER 4

This appendix provides details about the simulations reported in Chapter 4.

Input values—The task bit was set to -1 for the "what" task and to 1 for the "where" task.

Training—The weights of all systems were updated at each time step. Desired output values of 0.1 and 0.9 were used instead of 0 and 1.

Activation functions—The hidden and output units of all systems used activation functions that include the logistic function with asymptotes at 0 and 1.

Initial weights—The weights of the single networks and the expert networks were initialized with values randomly selected from a uniform distribution over the interval $[-\frac{1}{2}, \frac{1}{2}]$. The weights of the gating network were initialized to zero.

Table A.1: Parameter values used in the temporal crosstalk experiments.

	Random training			Blocked training		
System	Step size	Momentum	γ	Step size	Momentum	γ
1	3.00	0.00		0.44	0.75	
2	0.45	0.90		5.25	0.00	
3	5.75	0.00	3.50	6.00	0.00	2.75

Parameter values

- Temporal crosstalk experiments—For the single networks trained with the back-propagation algorithm, we used the step sizes and momentum that roughly give the best performance. These values are listed in Table A.1.

  For some of the parameters of the modular architecture, we searched for the values that give the best behavior, and for others, we didn't. Specifically, we did not attempt to optimize the step size and momentum of the gating network and α used in Equation 3.3. These parameters had the values 0.01, 0.0, and 0.2 respectively. In addition, although each expert network may have its own step size and momentum, the same values were used for all expert networks. Thus, the only parameter values that we attempted to optimize are the step size and momentum used by all expert networks and γ used in Equation 3.4. These values are listed in Table A.1.

- Spatial crosstalk experiment—Since no comparisons were performed in this set of experiments, we did not attempt to locate the optimal parameter values.

Thus, we provide the values that were used, not the values that are best. For the expert networks, the step size and momentum were 8.0 and 0.0. For the gating networks, the step size was 0.01, momentum was 0.0, α was 0.2, and γ was 0.75. When the system's current performance was significantly better than its past performance, λ_{NT} was set to zero. Otherwise, λ_{NT} was 10 for epochs 1–400, 8 for epochs 401–500, 6 for epochs 501–600, 4 for epochs 601–700, 2 for epochs 701–800, and 1 for epochs 801–900.

APPENDIX B

SIMULATION DETAILS FOR CHAPTER 5

This appendix provides details about the simulations reported in Chapter 5.

Training—The weights of all systems were updated at each time step.

Activation functions—The hidden units of all single networks, expert networks, and share networks as well as the output units of the gating networks used activation functions that include the logistic function with asymptotes at 0 and 1. The output units of all single networks, expert networks, and share networks used the identity function in their activation function. If a single network, expert network, or share network contained hidden units, then the output of that network was considered to be 250 times the activation of the output units. The desired outputs for such a network were similarly scaled. This was done so that the activations of the output units lied in the interval [-2, 2] which is close to the interval that contains the activations of the hidden units.

Initial weights—The weights of the single networks, expert networks, and share networks were initialized with values randomly selected from a uniform distribution over the interval $[-\frac{1}{2}, \frac{1}{2}]$. The weights of the gating networks were initialized to zero. The units of all networks except the gating networks contained a bias weight.

Parameter values—For the single networks, we used the step sizes and momentum that roughly give the best performance. For some of the parameters of the modular architectures and the modular architectures with share networks, we searched for the values that give the best behavior, and for others, we didn't. Specifically, we did not attempt to optimize the step size and momentum of the gating networks and α used in Equation 3.3. These parameters had the values 0.01, 0.0, and 0.2 respectively. In addition, although each expert and share network of a system may have its own step size and momentum, the same values were used for all expert and share networks of a system. Thus, the only parameter values that we attempted to optimize are the step size and momentum used by all expert and share networks of a system and γ used in Equation 3.4. The parameter values used are listed in Table B.1.

Table B.1: Parameter values used in the multi–payload robotics experiments.

System	Step size	Momentum	γ
N–L	0.00400	0.00	
N–NL	0.00225	0.25	
MA–L	0.00100	0.00	1.2
MA–NL	0.00165	0.25	2.2
MAS–L	0.00070	0.00	1.3
MAS–NL	0.00075	0.25	1.3

APPENDIX C

SIMULATION DETAILS FOR CHAPTER 6

This appendix provides details about the simulations reported in Chapter 6.

Input values—The input units of the gating network provided a coarse-coded representation of the value of x. The n^{th} input unit was set to 1 if

$$-2\pi + \frac{(n-1)\pi}{20} \leq x < -2\pi + \frac{(n-1)\pi}{20} + \frac{\pi}{2}. \tag{C.1}$$

Otherwise, it was set to 0.

Training—The weights of all systems were updated at each time step. The first system was run for 50,000 time steps. The second and third systems were run for 100,000 time steps.

Activation functions—The hidden units of all expert networks and the output units of the gating network used activation functions that include the logistic function with asymptotes at 0 and 1. The output units of all expert networks used the identity function in their activation function.

Initial weights—The weights of the expert networks were initialized with values randomly selected from a uniform distribution over the interval $[-\frac{1}{2}, \frac{1}{2}]$. The weights of the gating network were initialized to zero. The units of the expert networks, but not the gating network, contained a bias weight.

Parameter values

- Gating network—In all three systems, the values of the parameters of the gating network were: step size = 0.01; momentum = 0.0; $\alpha = 0.2$ (used in Equation 3.3); $\gamma = 0.6$ (used in Equation 3.4).

- System 1—Both expert networks used a step size of 0.005 and momentum of 0.0.

- System 2—The step size of the multi–layer expert network was 0.1 and the step size of the single–layer expert network was 0.005. Both expert networks used a momentum of 0.0. When the system's current performance was significantly better than its past performance, λ_{NT} was set to zero. Otherwise, λ_{NT} was 10 for time steps 1–15000, 8 for time steps 15001–30000, 6 for time steps 30001–45000, 4 for time steps 45001–60000, 2 for time steps 60001–75000, and 1 for time steps 75001-100000.

- System 3—Both expert networks used a step size of 0.1 and momentum of 0.0.

REFERENCES

[1] Albus, J.S. (1975) A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Journal of Dynamical Systems: Measurement and Control*, 97, 220–227.

[2] Atkeson, C.G. & McIntyre, J. (1986) Robot trajectory learning through practice. *Proceedings of the IEEE Conference on Robotics and Automation*, 1737–1742.

[3] Atkeson, C.G. & Reinkensmeyer, D.J. (1988) Using associative content–addressable memories to control robots. *Proceedings of the IEEE Conference on Decision and Control*, 792–797.

[4] Ballard, D.H. (1984) Parameter Nets. *Artificial Intelligence*, 22, 235–267.

[5] Ballard, D.H. (1986) Cortical connections and parallel processing: Structure and function. *The Behavioral and Brain Sciences*, 9, 67–120.

[6] Ballard, D.H. (1987) Modular learning in neural networks. *Proceedings of the Sixth National Conference on Artificial Intelligence*, 279–284.

[7] Barlow, H.B. (1986) Why have multiple cortical areas? *Vision Research*, 26, 81–90.

[8] Barlow, H. & Földiák, P. (1989) Adaptation and decorrelation in the cortex. In R. Durbin, C. Miall, & G. Mitchison (eds.), *The Computing Neuron*. Reading, MA: Addison–Wesley Publishing Company.

[9] Barto, A.G. & Anandan, P. (1985) Pattern–recognizing stochastic learning automata. *IEEE Transactions On Systems, Man, and Cybernetics*, 15, 360–375.

[10] Barto, A.G. & Jordan, M.I. (1987) Gradient following without back-propagation in layered networks. *Proceedings of the IEEE First Annual Conference on Neural Networks*, 2, 629–636.

[11] Barto, A.G., Sutton, R.S., & Watkins, C.J.C.H. (1989) Learning and sequential decision making. COINS Technical Report 89–95, University of Massachusetts, Amherst, MA. A version of this report will appear in M. Gabriel & J.W. Moore (eds.), *Learning and Computational Neuroscience*. Cambridge, MA: The MIT Press, in press.

[12] Bridle, J. (1989) Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In F. Fougelman-Soulie & J. Herault (eds.), *Neuro-computing: Algorithms, Architectures, and Applications*. New York: Springer-Verlag.

[13] Cowey, A. (1981) Why are there so many visual areas? In F.O. Schmidt, F.G. Warden, G. Adelman, & S.G. Dennis (eds.), *The Organization of the Cerebral Cortex*. Cambridge, MA: The MIT Press.

[14] Cowey, A. (1985) Aspects of cortical organization related to selective attention and selective impairments of visual perception: A tutorial review. In M.I. Posner & O.S.M. Marin (eds.), *Attention and Performance XI*. Hillsdale, NJ: Lawrence Erlbaum Associates.

[15] Craig, J.J. (1986) *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley Publishing Company.

[16] Denker, J., Schwartz, D., Wittner, B., Solla, S., Hopfield, J., Howard, R., & Jackel, L. (1987) Automatic learning, rule extraction, and generalization. *Complex Systems*, 1, 877–922.

[17] Duda, R.O. & Hart, P.E. (1973) *Pattern Classification and Scene Analysis*. New York: John Wiley & Sons.

[18] Durbin, R. & Mitchison, G. (1990) A dimension reduction framework for understanding cortical maps. *Nature*, 343, 644–647.

[19] Durbin, R. & Willshaw, D.J. (1987) An analogue approach to the traveling salesman problem using an elastic net method. *Nature*, 326, 689–691.

[20] Feldman, J.A. (1982) Dynamic connections in neural networks. *Biological Cybernetics*, 46, 27–39.

[21] Feldman, J.A. & Ballard, D.H. (1982) Connectionist models and their properties. *Cognitive Science*, 6, 205–254.

[22] Fodor, J.A. (1983) *The Modularity of Mind.* Cambridge, MA: The MIT Press.

[23] Freedman, S.A. & Forster, K.I. (1985) The psychological status of overgenerated sentences. *Cognition*, 19, 101–131.

[24] Geschwind, N. & Galaburda, A.M. (1987) *Cerebral Lateralization: Biological Mechanisms, Associations, and Pathology.* Cambridge, MA: The MIT Press.

[25] Grossberg, S. (1987) Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11, 23–63.

[26] Hampshire, J.B. & Waibel, A.H. (1989) The meta-pi network: Building distributed knowledge representations for robust pattern recognition. Technical Report CMU–CS–89–166, Carnegie–Mellon University, Pittsburgh, PA.

[27] Hinton, G.E. (1981) A parallel computation that assigns canonical object–based frames of reference. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 683–685.

[28] Hinton, G.E. (1981) Shape representation in parallel systems. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1088–1096.

[29] Hinton, G.E. (1989) Connectionist learning procedures. *Artificial Intelligence*, 40, 185–234.

[30] Hornik, K., Stinchcombe, M., & White, H. (1989) Multilayer feedforward networks are universal approximators. *Neural Networks*, 2, 359–366.

[31] Jacobs, R.A. (1989) Initial experiments on constructing domains of experience and hierarchies in connectionist systems. In D. Touretzky, G. Hinton, & T. Sejnowski (eds.), *Proceedings of the 1988 Connectionist Models Summer School.* San Mateo, CA: Morgan Kaufmann Publishers.

[32] Jacobs, R.A., Jordan, M.I., & Barto, A.G. (1990) Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. COINS Technical Report 90–27, University of Massachusetts, Amherst, MA.

[33] Jordan, M.I. (1986) Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 531–546.

[34] Jordan, M.I. (1986) Serial order: A parallel, distributed processing approach. Technical Report ICS–8604, University of California at San Diego, La Jolla, CA. A version of this report will appear in J.L. Elman & D.E. Rumelhart (eds.), *Advances in Connectionist Theory: Speech.* Hillsdale, NJ: Lawrence Erlbaum Associates, in press.

[35] Jordan, M.I. (1988) Supervised learning and systems with excess degrees of freedom. COINS Technical Report 88–27, University of Massachusetts, Amherst, MA.

[36] Kaas, J.H. (1982) The segregation of function in the nervous system: Why do sensory systems have so many subdivisions? In W.P. Neff (ed.), *Contributions to Sensory Physiology (Volume 7).* New York: Academic Press.

[37] Kawato, M. (in press) Computational schemes and neural network models for formation and control of multijoint arm trajectory. In T. Miller, R.S. Sutton, & P.J. Werbos (eds.), *Neural Networks for Control.* Cambridge, MA: The MIT Press.

[38] Kawato, M., Furukawa, K., & Suzuki, R. (1987) Hierarchical neural–network model for control and learning of voluntary movement. *Biological Cybernetics*, 57, 169–185.

[39] Kohonen, T. (1982) Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43, 56–69.

[40] Kosslyn, S.M. (1987) Seeing and imagining in the cerebral hemispheres: A computational approach. *Psychological Review*, 94, 148–175.

[41] le Cun, Y. (1985) Une procedure d'apprentissage pour reseau a sequil assymetrique [A learning procedure for asymmetric threshold network]. *Proceedings of Cognitiva*, 85, 599–604.

[42] le Cun, Y. (1989) Generalization and network design strategies. Technical Report CRG–TR–89–4, University of Toronto, Toronto, Ontario.

[43] Marr, D. (1982) *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.* New York: W.H. Freeman and Company.

[44] Maxwell, T., Giles, C.L., Lee, Y.C., & Chen, H.H. (1986) Nonlinear dynamics of artificial neural systems. In J.S. Denker (ed.), *Neural Networks for Computing, AIP Conference Proceedings 151.* New York: American Institute of Physics.

[45] McClelland, J.L. (1986) The programmable blackboard model of reading. In J.L. McClelland, D.E. Rumelhart, & the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 2: Psychological and Biological Models.* Cambridge, MA: The MIT Press.

[46] Miller, W.T. (1987) Sensor based control of robotic manipulators using a general learning algorithm. *IEEE Journal of Robotics and Automation*, 3, 157–165.

[47] Miller, W.T., Glanz, F.H., & Kraft, L.G. (1987) Application of a general learning algorithm to the control of robotic manipulators. *The International Journal of Robotics Research*, 6, 84–98.

[48] Minsky, M. (1986) *The Society of Mind.* New York: Simon and Schuster.

[49] Mishkin, M., Ungerleider, L.G., & Macko, K.A. (1983) Object vision and spatial vision: Two cortical pathways. *Trends In Neurosciences*, 6, 414–417.

[50] Moody, J. & Darken, C.J. (1989) Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1, 281–294.

[51] Narendra, K. & Thathachar, M.A.L. (1989) *Learning Automata: An Introduction.* Englewood Cliffs, NJ: Prentice Hall.

[52] Nowlan, S.J. (1989) The hard vs soft distinction in competitive adaptation. Thesis Proposal, Computer Science Department, Carnegie Mellon University.

[53] Nowlan, S.J. (1990) Max likelihood competition in RBF networks. Technical Report CRG–TR–90–2, Connectionist Research Group, University of Toronto, Toronto, Ontario.

[54] Nowlan, S.J. (1990) Maximum likelihood competitive learning. In D.S. Touretzky (ed.), *Advances in Neural Information Processing Systems 2.* San Mateo, CA: Morgan Kaufmann Publishers.

[55] Nowlan, S.J. & Hinton, G.E. (1989) Maximum likelihood decision–directed adaptive equalization. Technical Report CRG–TR–89–8, Connectionist Research Group, University of Toronto, Toronto, Ontario.

[56] Parker, D.B. (1985) Learning logic. Technical Report TR–47, Massachusetts Institute of Technology, Cambridge, MA.

[57] Plaut, D.C. & Hinton, G.E. (1987) Learning sets of filters using back–propagation. *Computer Speech and Language*, 2, 35–61.

[58] Poggio, T. & Girosi, F. (1989) A theory of networks for approximation and learning. A.I. Memo No. 1140, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

[59] Pollack, J.B. (1987) Cascaded back–propagation on dynamic connectionist networks. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, 391–404.

[60] Pomerleau, D.A. (1987) The meta–generalized delta rule: A new algorithm for learning in connectionist networks. Technical Report CMU–CS–87–165, Carnegie–Mellon University, Pittsburgh, PA.

[61] Reggia, J.A. (1987) Properties of a competition-based activation mechanism in neuromimetic network models. *IEEE First International Conference on Neural Networks*, 2, 131–138.

[62] Rueckl, J.G., Cave, K.R., & Kosslyn, S.M. (1989) Why are "what" and "where" processed by separate cortical visual systems? A computational investigation. *Journal of Cognitive Neuroscience*, 1, 171–186.

[63] Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986) Learning internal representations by error propagation. In D.E. Rumelhart, J.L. McClelland, & the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations.* Cambridge, MA: The MIT Press.

[64] Rumelhart, D.E. & Zipser, D. (1986) Feature discovery by competitive learning. In D.E. Rumelhart, J.L. McClelland, & the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations.* Cambridge, MA: The MIT Press.

[65] Sejnowski, T.J. (1981) Skeleton filters in the brain. In G.E. Hinton & J.A. Anderson (eds.), *Parallel Models of Associative Memory.* Hillsdale, NJ: Lawrence Erlbaum Associates.

[66] Selfridge, O., Sutton, R.S., & Barto, A.G. (1985) Training and tracking in robotics. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 670–672.

[67] Sutton, R.S. (1986) Two problems with backpropagation and other steepest-descent learning procedures for networks. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 823–831.

[68] Sutton, R.S. (1988) Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9–44.

[69] Waibel, A. (1989) Modular construction of time-delay neural networks for speech recognition. *Neural Computation*, 1, 39–46.

[70] Werbos, P.J. (1974) *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, Cambridge, MA.

[71] Werbos, P.J. (1988) Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions of Systems, Man, and Cybernetics*, 17, 7–20.

[72] Yeung, D.Y. & Bekey, G.A. (1989) Using a context–sensitive learning network for robot arm control. *Proceedings of the 1989 IEEE Conference on Robotics and Automation*, 3, 1441–1447.

[73] Yuille, A.L. & Grzywacz, N.M. (1989) A winner–take–all mechanism based on presynaptic inhibition feedback. *Neural Computation*, 1, 334–347.