

Neural Network Design for Switching Network Control

Thesis by

Timothy X Brown

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California
1991
(Submitted June 29, 1990)

©1991
Timothy X Brown
All rights reserved

Acknowledgements

Many more people than the one named on the cover influenced the content of this thesis and even the fact that you are reading this now. Thanks goes to each of these individuals. First and foremost honor goes to my advisor Edward C. Posner whose knowledge, seemingly endless in breadth, and insights appear throughout this thesis. He also suggested and guided the development of the thesis topic. Many fellow graduate students added ideas and were sounding boards to my own ideas, as well as being friends. These include Kathleen Kramer, Zorana Popović, Ivan Onyszchuk, Gabriel Rabeiz, Rajaram Ramesh, John Miller, and Kumar Sivarajan. Special thanks goes to Dr. Kuo-Hui Liu of Pacific Bell. He introduced me to the ATM switching problem and suggested several avenues of approach. Pacific Bell provided generous grants to the EE systems group that not only supported me throughout my research at Caltech, but also provided extensive computing facilities.

On a more personal level I must thank: my parents for their support; my brother and sister for spurring me on to higher education; and my grandparents who each in their own way positively influenced my academic career. To my wife I reserve my greatest gratitude. All my experiences at Caltech are intimately intertwined with her.

Abstract

A neural network is a highly interconnected set of simple processors. The many connections allow information to travel rapidly through the network, and due to their simplicity, many processors in one network are feasible. Together these properties imply that we can build efficient massively parallel machines using neural networks. The primary problem is how do we specify the interconnections in a neural network. The various approaches developed so far such as outer product, learning algorithm, or energy function suffer from the following deficiencies: long training/specification times; not guaranteed to work on all inputs; requires full connectivity.

Alternatively we discuss methods of using the topology and constraints of the problems themselves to design the topology and connections of the neural solution. We define several useful circuits—generalizations of the Winner-Take-All circuit—that allows us to incorporate constraints using feedback in a controlled manner. These circuits are proven to be stable, and to only converge on valid states. We use the Hopfield electronic model since this is close to an actual implementation. We also discuss methods for incorporating these circuits into larger systems, neural and non-neural. By exploiting regularities in our definition, we can construct efficient networks.

To demonstrate the methods, we look to three problems from communications. We first discuss two applications to problems from circuit switching; finding routes in large multistage switches, and the call rearrangement problem. These show both, how we can use many neurons to build massively parallel machines, and how the Winner-Take-All circuits can simplify our designs.

Next we develop a solution to the contention arbitration problem of high-speed packet switches. We define a useful class of switching networks and then design a neural network to solve the contention arbitration problem for this class. Various aspects of the neural network/switch system are analyzed to measure the queueing performance of this method. Using the basic design, a feasible architecture for a large (1024-input) ATM packet switch is presented. Using the massive parallelism of neural networks, we can consider algorithms that were previously computationally

unattainable. These now viable algorithms lead us to new perspectives on switch design.

Contents

1	Introduction:	1
1.1	What is a Neural Network and What Will We Show?	1
1.2	Neural Networks and Parallel Machines	2
1.3	Neural Networks	4
1.4	Designing Neural Networks	5
1.5	Saying What We Said	7
1.6	Thesis Outline	8
2	Designing with Neural Networks	11
2.1	Introduction	11
2.2	Neural Network Model	11
2.3	The Winner-Take-All Circuit	14
2.4	The Multiple Overlapping Winner-Take-All Circuit	18
2.5	The Multiple Overlapping K-Winner-Take-All Circuit	20
2.6	Neuron Gain	22
2.7	Designing with Winner-Take-All Circuits	25
2.8	Conclusions	26
2.A	Winner-Take-All Dynamics	27
3	Controlling Circuit Switching Networks	31
3.1	Introduction	31
3.2	Background on Switches	31
3.3	Large Multistage Switching Networks	34

3.4	Finding Routes using Neural Networks	37
3.5	Rearrangeable Switches	39
3.6	The Neural Network Solution to the Rearrangement Problem	44
3.7	Conclusion	57
4	Banyan Network Controller	60
4.1	Introduction	60
4.2	ATM switching networks	60
4.3	Banyan Networks	63
4.4	Blocking Constraints and Deterministic Switches	65
4.5	Queueing	67
4.6	The Neural Network Solution	72
4.7	Network Prompting	73
4.8	Simulation Results	75
4.9	Implementation Considerations	83
4.10	Buffered Memory Switches and Large Switch Designs	87
4.11	Conclusions	90
4.A	Estimating the Tails	91
5	Epilogue	98

List of Figures

1.1	Loss in computational efficiency.	3
1.2	Solution to the parity problem.	7
2.1	The electronic neural model.	12
2.2	Winner-Take-All mutual inhibition circuit.	15
2.3	The Multiple Overlapping Winner-Take-All concept.	18
2.4	A set matrix showing the equivalence of the threshold definitions. . .	19
3.1	The $N \times N$ crossbar switch concept.	32
3.2	A general three-stage switch.	34
3.3	Multistage switch.	35
3.4	Multistage switch with five calls already put up.	36
3.5	Neural network for switch path search.	38
3.6	Neural network searching for a path.	40
3.7	Paull matrix.	42
3.8	A blocked call request at $(2, 3)$	43
3.9	Unblocking rearrangement for call request.	44
3.10	Three-dimensional neuron grid.	46
3.11	Additional memory neuron.	50
3.12	Neural network solving blocking situation.	52
3.13	Additional circuitry to force the network to follow Paull's algorithm. .	53
3.14	Solving the rearrangement problem for $n = 3$	55
4.1	An 8×8 Banyan switching network.	64
4.2	Equivalence classes.	68

4.3	The queueing model.	69
4.4	Mapping from the queue state to a cell matrix.	71
4.5	Demonstration of smoothing.	77
4.6	Average queue size vs. average number of arrivals (Bernoulli).	79
4.7	Average queue size vs. average number of arrivals (batch).	80
4.8	Performance range.	81
4.9	Control architecture comparison.	82
4.10	A schematic of the queue architecture for one of the input queues.	85
4.11	The buffered-memory Banyan hybrid switch.	88
4.12	A sample of a queue size distribution.	92

List of Tables

2.1	Comparison of necessary neuron gains	24
3.1	Tally of the neural rearranger size.	56
4.1	Buffer size estimates.	84
4.2	Buffer size necessary for the hybrid switch.	90

Chapter 1

Introduction:

Has parallel computing failed us again?

—W. L. Miranker [1]

1.1 What is a Neural Network and What Will We Show?

Neural networks are a class of systems that have many simple processors —“neurons”— that are highly interconnected. The function of each neuron is simple, and the behavior is determined predominantly by the set of interconnections. Thus, a neural network is a special form of parallel computer. Although a major impetus for using neural networks is that they may be able to “learn” the solution to the problem that they are to solve, we argue that another perhaps stronger impetus is that they provide a framework for designing massively parallel machines. As a step in this direction, we develop techniques which will aid us in our designs.

The highly interconnected architecture of switching networks suggests similarities to neural networks, and indeed, we present three applications in switching in which neural networks can solve the problems efficiently. The first two problems come from circuit switching: finding routes through large multistage switches and calculating a rearrangement that allows a new call to be placed through a rearrangeable switch. In this latter problem we show that a computational advantage can be gained by using nonuniform time delays in the network. The last application is to high-speed

interconnection networks, of relevance to packet switching. Using the computational speed of many neural processors working in parallel, we are able to resolve contention for paths through the network in the necessary time. We analyze this problem in detail to show the applicability of neural systems to real applications.

1.2 Neural Networks and Parallel Machines

For applications requiring computational speed beyond what a single processor is capable of, increasing the number of processors can decrease the computation time. Standard parallel computing models are all fundamentally equivalent to the Turing model of computation. While, in principle, the programming of the multiple nodes is a straightforward extension of the programming of a single node, unfortunately, complications arise since the processors must spend time communicating intermediate results and waiting for other processors to send needed data. The programming and even the way that the multiple processors must be connected so that the machine isn't bogged down in this interprocessor-communication and scheduling overhead is not so well understood. As a result, the increase in speed as a function of the number of processors is significantly sublinear.

We illustrate this phenomenon using data from reference [2]. The time using one processor for a given task is less than N times the time spent with N processors. Putting this in a comparable form, we define:

$$\text{Loss in Efficiency} \triangleq \frac{N \times (\text{time using } N \text{ processors}) - (\text{time using } 1 \text{ processor})}{N \times (\text{time using } N \text{ processors})}$$

Figure 1.1 shows examples of the *loss in efficiency* as a function of the number of processors for some common parallel machines.

This was from a performance test that allowed the manufacturers to use the fastest possible algorithm that they could develop to solve a system of 1000 equations and 1000 unknowns. This comparison is interesting because these machines have a variable number of processors. By comparing only within a single architecture, we can control for the differences between machines. The loss in efficiency is significant for

Figure 1.1: Loss in computational efficiency.

the architectures in the graph. For example, with just seven processors, the Alliant computer spends almost 25% of each processor's time on this communications overhead. Over the domain of the data given, the loss in efficiency grows linearly with the number of processors. The linear increase in the loss in efficiency implies a decreasing amount of speedup that, if extended, would ultimately lead to an absolute decrease in the computing speed. This does not bode well for systems with many processors.

Even if the design and programming problems could be solved, a microprocessor with its associated memory and support components is a complex and relatively expensive computational element, limiting the number of processors. Certainly the idea of 1,000,000 microprocessors in a system does not yet seem feasible.

Alternatively one can consider neural networks. While they don't fit into the von Neumann/Turing framework (although some researchers have been able to formulate special cases within this framework as feedforward threshold-logic circuits), much

work has gone into developing their theory. General theorems on their behavior have been derived [3][4]. In digital systems, we typically speak in terms of “0” or “1,” despite the fact that these symbols only represent real valued voltages and currents that exist inside a real system. Furthermore, the logical operation of elements, such as the AND gate or the flip flop, also represents more complex underlying behavior.

There exist similar abstractions that simplify the analysis of neural network systems. Thus, we often can treat the neuron as a simple threshold element whose output is either “−1” or “+1.” We will also introduce neural network elements that will allow us to introduce feedback in a controlled manner. Using these elements as building blocks, similar to the use of flip flops, etc., in digital systems, releases us from the details of the underlying dynamics.

Because of the simplicity of the individual neurons and because the connections themselves are an integral part of the system’s computation, large neural networks are feasible and useful. Biological neural systems, such as the brains of mammals, show that systems with as many as 10^{12} neurons are possible. The problem that arises is how to configure a network to solve a particular problem.

1.3 Neural Networks

Before discussing the details of the neural network model that we use, we describe some general principles. The basic unit of an artificial neural network is the *neuron*. Its function is extremely simple, usually just a threshold function on the weighted sum of its inputs. The functionality of the network is not derived from the operation of the individual neurons, but from the collective properties derived from the many connections between the neurons. The problem in any neural network is to define these connections so that the network operates as desired. There are three issues that arise:

- Can we determine these connections in a simple manner? It has been shown theoretically that in general this problem belongs to a class of “hard” problems [5]. This is true even for simple neural architectures [6]. Empirically it is

known that the number of iterations needed to converge on a set of connections grows quickly as the size of network grows [7][8]. Because of this, we cannot guarantee that learning algorithms such as “Back-Propagation,” or “The Boltzmann Machine,” will find a working set of connections in a reasonable amount of time. Furthermore, if we must modify the problem (e.g., go to a larger size), does the procedure for specifying the weights have to be repeated all over again.

- If we have x neurons, then in general a network may have x^2 connections, one from every neuron to every other neuron. Most methods of specifying the connections require such a so-called *fully interconnected* network. For large networks, this can become prohibitively complex.
- Does the set of connections found actually find a valid solution for all possible inputs? Methods proposed in the literature have been shown to work statistically when the number of desired stable states is small compared to the number of neurons [9], or empirically [8] for small problems. These conditions are unreasonable for most applications. For this reason we can never be sure that for the networks found by these methods there isn't some input that will be problematic.

The method that we propose for the applications in this thesis is to simply *design* the set of connections for a problem using direct methods, with sub-networks that are already available.

1.4 Designing Neural Networks

This idea is analogous to the work of a digital designer. Given a problem, the designer doesn't look directly at desired input and output signals and then solder a circuit using transistors, resistors, and other components; rather they analyze what they know about the problem to formulate a solution using the already available AND gates, flip-flops, etc., leading to a circuit design. A good designer will often try to

produce a solution to the general class of problems, not just the specific instance that is at hand.

Our approach with neural networks will be similar. Given a problem, analysis can yield much information about the problem. We often know the constraints and the direct causes of particular elements in the response. It is therefore to our advantage to incorporate this knowledge into the neural solution. As research continues on neural networks, the number of classes of neural networks which are well understood increases. These can help us to incorporate the knowledge we have about the problem. The fact that a neural network can assume quite arbitrary topologies is extremely useful. The topology of the neural network can be matched to the topology of the problem. This obviates the correspondence between the problem and the neural solution. Finally, since we know the underlying structure which is producing the neural network, we can exploit this structure to simplify the construction. These principles will be very important in guiding us to our design solutions.

To introduce some of the basic ideas, consider the infamous (in neural circles) *parity check problem*: Given N inputs of “ -1 ” or “ $+1$,” output a “ $+1$ ” if the number of input “ 1 ’s” is odd, otherwise output a “ -1 .” This is often quoted as an unnatural and difficult problem for neural networks to solve [10]. General learning algorithms have great difficulty in finding a solution to the parity check problem. But it is known that there exists a straightforward neural network to solve this for any N ; we present this network below.

The way to solve this problem here is to count the number of “ $+1$ ’s” in the input, and then to see if this number is odd or even. So we first create N neurons numbered 1 through N . We connect neuron i so that it only turns on if there are at least i “ $+1$ ’s” in the input. We then create an output neuron that uses this regularity to only turn on if the number of “ $+1$ ’s” is odd. This clearly works. The network is shown in Figure 1.2 [8, p. 334].

Once a solution exist for a class of problems, then as the size of the problem becomes larger, the many processors in the corresponding network can reduce the time of the computation. In the example of the parity problem, all the elements are

Figure 1.2: Solution to the parity problem.

feedforward, and the output is only two stages from the input, independent of N . This implies—to a first order approximation—that the time to solve the problem is constant for all N .

One must be a little careful in the solution, though. First, the number of neurons and connections must not grow too fast with the problem. Second, the time for the network to produce the output must not grow too quickly either. Thus a solution to the parity problem using 2^N neurons and connections, or in which the number of stages between the input and the output is N^2 , for example, would not be acceptable. In all of our solutions we will be careful to show that both the component and time complexity is reasonable.

1.5 Saying What We Said

We argue that neural networks are a significant break from conventional parallel computing. Whereas the fundamental problem with conventional parallelism is how to program the multiprocessors; with neural networks it is how to determine the connections between processors. Most current methods assume that the problem

is unstructured, unnecessarily ignoring available information. By designing solutions using available information, we can produce efficient general neural solutions to whole classes of problems.

1.6 Thesis Outline

Chapter 2 will present the core analytical results. We present the basic neural model and define three generalizations to the Winner-Take-All neural circuit. Each of these generalizations is shown rigorously to be stable and only converge on stable states. A certain flexibility is incorporated into these designs. We develop methods for utilizing this flexibility for particular applications. By exploiting the regularities in the design, we present methods for significantly reducing the connectivity, while still maintaining identical functionality.

Chapters 3 and 4 discuss specific applications of these methods. Chapter 3 contains two applications to problems from circuit switching; finding routes in large multistage switches, and the call rearrangement problem. These will show both, how we can use many neurons to build massively parallel machines, and how the Winner-Take-All circuits can simplify our designs.

Chapter 4 is a complete development of an application to high-speed packet switches. We define a useful class of switching networks and then design a neural network to solve the contention arbitration problem for this class. Various aspects of the neural network/switch system are analyzed to measure the queueing performance of this method. Using the basic design, a feasible architecture for a large (1024-input) packet switch is presented.

Bibliography

- [1] Miranker, W. L., *Has Parallel Computing Failed Again? A Manifesto for Parallel Computation*, EE Systems Seminar presented at Caltech, March 12, 1990.
- [2] Dongara, J. J., *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*. Argonne Nat. Lab. Technical Memorandum No. 23, August 13, 1988.
- [3] Hopfield, J. J., Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons, *Proceedings of the National Academy of Sciences USA*, Vol. 81, May 1984, pp. 3088–3092.
- [4] Marcus, C. M., Westervelt, R. M., Stability of Analog Neural Networks with Delay, *Physical Review A*, Vol. 39, 1989, pp. 347–359.
- [5] Judd, J. S., *Neural Network Design and the Complexity of Learning*, MIT Press, Cambridge, MA, 1990.
- [6] Blum, A., Rivest, R. L., Training a 3-Node Neural Network is NP-Complete, ed. Touretzky, D. S., *Advances in Neural Information Processing Systems I*, Morgan Kaufman Pub., San Mateo, CA, 1989, pp. 494–501.
- [7] Hinton, G. E., Sejnowski, T. J., Learning and Relearning in Boltzmann Machines, ed. Rumelhart, D. E., McClelland, J. L., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, MIT Press, Cambridge, MA, 1986, Chapter 7.

- [8] Rumelhart, D. E., Hinton, G. E., Williams, R. J., Learning Internal Representations by Error Propagation, ed. Rumelhart, D. E., McClelland, J. L., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, MIT Press, Cambridge, MA, 1986, Chapter 8.
- [9] McEliece, R. J., Posner, E. C., Rodemich, E. R., Venkatesh, S. S., The Capacity of the Hopfield Associative Memory, *IEEE Transactions on Information Theory*, Vol. 33, No. 4, pp. 461–482, Jul. 1987.
- [10] Minsky, M. L., Papert, S. A., *Perceptrons: An Introduction to Computational Geometry*, Expanded Edition, MIT Press, Cambridge, MA, 1988, p. 153.

Chapter 2

Designing with Neural Networks

It could be that the nonlinear nature of inhibitory feedback is the key to building a complex analog processing system with a great deal of gain and time delay, and to keeping the entire mess stable. We will know that we understand these matters only when we can build such a system—and can keep it stable.

—Carver A. Mead [1]

2.1 Introduction

In this chapter we will discuss the neural model which will be the starting point of our work. We will then develop a series of increasingly complex neural network classes that will lead us to an understanding of how we can build stable complex analog processing systems that can be applied to specific problems.

2.2 Neural Network Model

We start with the Hopfield continuous model [2], since it is defined in terms of electronic components, and therefore close to an actual implementation. In this model the neurons are high-gain amplifiers that are interconnected through resistors as shown in Figure 2.1 for a four-neuron network. The amplifiers have both positive and negative outputs. A connection from a positive output is known as *excitatory*, and a connection from a negative output is known as *inhibitory*. We define $\mathbf{W} = \{w_{ij}\}$, the *connection matrix*. If R_{ij} is the value of the resistive connection from neuron j to neuron i , then we say that the strength of the connection, $|w_{ij}|$, is $1/R_{ij}$. The sign

Figure 2.1: The electrical model of a neural network showing topology and details of a neuron.

of w_{ij} is positive or negative depending on whether the connection is excitatory or inhibitory. Each neuron can have an external input, I_i . A threshold, t_i , is subtracted from all the inputs, and the result is the net input to the neuron. In a system of N neurons, given a neuron i , its state variable, u_i , is governed by the differential equation:

$$c_i \frac{du_i}{d\tau} = -\lambda_i u_i + \sum_{j=1}^N w_{ij} g(u_j) + I_i - t_i, \quad (2.1)$$

where

$$\lambda_i = \sum_{j=1}^N |w_{ij}|.$$

This equation is derived from simple circuit analysis of each neuron. Some researchers include the internal resistance of the amplifier. This only adds a positive constant to λ_i that does not significantly change the results, so we will disregard it here.

The response or output of the neuron is the “sigmoidal” function g . More precisely, $g(x) = f(\gamma x)$, where $\gamma > 0$ is the neuron *gain* and f satisfies: $-1 \leq f(x) \leq 1$; $\lim_{x \rightarrow \pm\infty} f(x) = \pm 1$; $0 < f'(x) \leq f'(0) = 1$; and $f'(0)$ is continuous. This allows for most definitions of a continuous neuron’s function. Neurons with outputs close to $+1$ and -1 are said to be *on* and *off* respectively. We define I_i^{tot} to be the sum of the last three terms in (2.1):

$$I_i^{tot} \triangleq \sum_{j=1}^N w_{ij} g(u_j) + I_i - t_i. \quad (2.2)$$

We see that u_i evolves as a negative exponential with time constant c_i/λ_i , and that it decays toward the value I_i^{tot}/λ_i . In general neuron i is on or off depending on whether I_i^{tot} is positive or negative.

Formally, for a fixed g , we define a *neural network* as a triple $(\mathbf{W}, \mathbf{t}, \mathbf{c})$ corresponding to the matrix of connections, (w_{ij}) , vector of thresholds, (t_i) , and vector of capacitances, (c_i) , respectively. The state of the system is the vector, $\mathbf{u}(\tau) = (u_i(\tau))$. The input to the system is $\mathbf{u}(\tau = 0)$ and the vector, $\mathbf{I} = (I_i)$, that could be a function of time, but we will assume that it is static.

A *neural computation* is defined as

$$\Psi_{(\mathbf{W}, \mathbf{t}, \mathbf{c})}(\mathbf{I}, \mathbf{u}(0)) \triangleq \lim_{\tau \rightarrow \infty} (g(u_1(\tau)), g(u_2(\tau)), \dots, g(u_N(\tau))) \triangleq G(\mathbf{u}^*)$$

Neural networks such that the above limit is defined for any input are known as *stable*. Reference [2] shows that a sufficient condition for a network to be stable is that \mathbf{W} be symmetric. All of the networks that we will define have symmetric \mathbf{W} and therefore are stable.

In equation (2.1), scaling c_i only changes the time scale of the calculation. We also note from (2.1) that for any scalar $\beta > 0$, $\Psi_{(\mathbf{w}, t, \mathbf{c})}(\mathbf{I}, \mathbf{u}(0)) \equiv \Psi_{(\beta\mathbf{w}, \beta t, \beta\mathbf{c})}(\beta\mathbf{I}, \mathbf{u}(0))$. This implies that we can scale the neural network to correspond to any particular implementation without affecting the calculation.

2.3 The Winner-Take-All Circuit

Now that the neural circuit is better defined, we discuss a neural circuit that we will use several times, the Winner-Take-All circuit. As its name implies, it has the property that given N neurons all with the same initial internal state (i.e., $u_i(0) = u_0$ for all i), only one (or a one, if more than one exists) neuron turns on, the one with the largest external input. This process of choosing the neuron with the largest input is known as a *competition*. By using external inputs, this circuit provides a method for selecting a neuron to turn on.

A generalization of this circuit is the K -Winners-Take-All circuit. This has the property that not just the neuron with the largest, but the K neurons with the K largest inputs, turn on. For an N -neuron K -Winners-Take-All, we define the network prototypically as:

$$\begin{aligned} w_{ij} &\triangleq \begin{cases} -1 & \text{if } i \neq j \\ a, |a| < 1 & \text{if } i = j \end{cases} \\ t_i &\triangleq N - 2K + 1, \\ c_i &\triangleq C, \end{aligned} \tag{2.3}$$

where C is a positive constant. The diagonal element, a , denotes a small amount of direct feedback which we allow. We limit the inputs by:

$$|a| < |I_i| < (2 - |a|). \tag{2.4}$$

Note that in general, if $I_i < 0$, neuron i will be off and out of the competition. The

Figure 2.2: Winner-Take-All mutual inhibition circuit.

circuit is shown in Figure 2.2a. Using definition (2.3) in (2.1):

$$C \frac{du_i}{d\tau} = -\lambda u_i + (a + 1)g(u_i) - \sum_{j=1}^N g(u_j) + I_i - (N - 2K + 1), \quad (2.5)$$

where $\lambda = N - 1 + |a|$. Analysis in reference [3] shows that with these thresholds and connections this circuit has the property that given different initial internal states, $u_i(0)$, and $I_i = 1$ for all i , the K neurons with the K largest initial internal states will be on and the rest will be off. As we shall see, this formulation is not as versatile as the one above, but in any case, we use the analysis in [3] as a basis for our results. The fundamental result appears in Appendix 2.A at the end of this chapter.

We want the neural network to make a decision based on the inputs. The problem is that given a neural calculation it is possible that the final state of one or more of the neurons is in the vicinity of zero and even with arbitrarily high gain the output cannot be forced to plus one or minus one. Such neurons we call *undetermined*, otherwise a neuron is *determined*. Appendix 2.A contains a proof of the following theorem:

Theorem 2.1 *Given the neural network defined by (2.3), the neural network is stable and there exist at most one undetermined neuron.*

For any neuron i , if we can bound u_i^* away from zero, then we know from the definition of g that with a large enough gain, γ , $g(u_i^*)$ can be made arbitrarily close to ± 1 and neuron i is determined. If I_i^{tot} defined in equation (2.2) can be bounded away from zero, then since in equilibrium $u_i^* = I_i^{tot}/\lambda$, this is sufficient to bound u_i^* away from zero. Further, this implies that the sign of I_i^{tot} determines whether neuron i is on or off.

We now show that the definition of the K -Winners-Take-All is valid. It is slightly complicated if fewer than K neurons have a positive input (i.e., $I_i > |a|$), but the only effect is that some of the other non-competing neurons may be forced on:

Theorem 2.2 *Given a neural network defined by (2.3), satisfying (2.4), and at time $t = 0$ with all of the neurons starting at the same initial state u_0 ; let P be the number of neurons satisfying $|a| < I_i < (2 - |a|)$. Given these conditions, the neural*

computation results in the K' neurons with the K' largest inputs on, where $K' = \begin{cases} K & \text{if } P \geq K \\ K - 1 & \text{if } P < K \end{cases}$; the rest of the neurons are off.

Proof: Throughout this proof we assume that the gain is large enough so that the output of any neuron that is on or off is arbitrarily close to ± 1 respectively. We define the quantity $\epsilon = \min_i(\{|I_i| - |a|\}, \{(2 - |a|) - I_i\})$. This is the closest that any of the I_i approach any of the limits in (2.4). By Theorem 2.1 we know that the network is stable, and that at most there is one undetermined neuron.

Suppose there exists such an undetermined neuron i . Except for this neuron, all of the other neurons are on or off. Let κ_i denote the number of *other* neurons which are on. In this case;

$$I_i^{tot} = (ag(u_i^*) + I_i) + 2(K - 1 - \kappa_i). \quad (2.6)$$

No matter what the integer κ_i is, we still have $|I_i^{tot}| > \epsilon$, contradicting that neuron i is undetermined. Thus all the neurons are determined.

Equation (2.6) applies equally well to other neurons. If $\kappa_i \geq K$ then $I_i^{tot} < -\epsilon$ and neuron i is off. This implies that $K' \leq K$. If instead $\kappa_i < K - 1$ then $I_i^{tot} > \epsilon$ and neuron i is on. This implies $K' \geq K - 1$. Finally, if $\kappa_i = K - 1$ then neuron i is on or off depending on whether $I_i > |a|$ or $I_i < -|a|$. This implies that $K' = K$ unless $P < K$.

To see that the neurons that are on are the ones with the K' largest inputs, we note:

$$C \frac{d(u_i - u_j)}{d\tau} = -\lambda(u_i - u_j) + (a + 1)(g(u_i) - g(u_j)) + (I_i - I_j). \quad (2.7)$$

If $u_i(\tau) = u_j(\tau)$ at some time τ , then the sign of (2.7) is determined by the sign of $(I_i - I_j)$. Since $u_i(0) = u_j(0)$ this implies that if $I_i > I_j$ then $u_i(\tau) > u_j(\tau)$ for all $\tau > 0$. Thus an ordering on the inputs translates to a corresponding ordering on the internal states of the neurons, and in particular on the final state of the neurons. ■

Figure 2.3: The Multiple Overlapping Winner-Take-All concept.

2.4 The Multiple Overlapping Winner-Take-All Circuit

We describe an extension to the Winner-Take-All circuit, the Multiple Overlapping Winner-Take-All. The idea is that we have a set of neurons, Θ , with subsets, S_i . These subsets represent constraints on the computation. Within each subset we restrict the neural computation to having at most one neuron on per subset. The idea is shown in Figure 2.3. Note that the subsets are not necessarily disjoint.

Intuitively, the neural network that will satisfy these constraints is the one in which the neurons in each subset are connected in a separate Winner-Take-All network. The rest of this section will develop this idea and prove its validity. For a given Θ and $\{S_i\}$, we define the network as follows: Let $J_i \triangleq \{k \mid \text{neuron } i \in S_k\}$.

$$\begin{aligned}
 w_{ij} &\triangleq \begin{cases} -|J_i \cap J_j| & \text{if } i \neq j \\ a, |a| < 1 & \text{if } i = j \end{cases} \\
 t_i &\triangleq \sum_{j \neq i} |J_i \cap J_j|, \\
 c_i &\triangleq C,
 \end{aligned} \tag{2.8}$$

where C is a positive constant. The connection defined in (2.8) between a neuron pair increases by -1 for each subset that the pair is jointly in, matching our intuition

		Subsets, S_i						
		1	2	...	m	m+1	...	
Neurons	1	1	1	1	1	1		
	2	1		1		1	1	1
	3			1		1		1
	\vdots					1		
	\vdots	1	1	1			1	
	N	1			1	1	1	1
	N	1		1	1	1		1

Figure 2.4: A set matrix showing the equivalence of the threshold definitions.

above. Note also that if there is only one subset, $S_1 = \Theta$, then we reduce to the Winner-Take-All. We limit the inputs as in the Winner-Take-All case by:

$$|a| < |I_i| < (2 - |a|). \quad (2.9)$$

It is sometimes easier to define t_i alternatively:

$$t_i = \sum_{j \in J_i} (|S_j| - 1). \quad (2.10)$$

These two definitions of t_i are equivalent by the following argument. Suppose that we create an *incidence matrix* where the rows are numbered 1 to N and the columns correspond to the subsets. We place a 1 in row i , column k if neuron i is in S_k . Assume without loss of generality that we are looking at neuron 1, and that it belongs to the first m subsets as shown in Figure 2.4. The definition in (2.8) sums the number of 1's in the first m columns of each row after the first, and the definition in (2.10) sums the number of 1's less one in each of the first m columns. These both count the same 1's, so they are equivalent.

In Appendix 2.A, we prove the following:

Theorem 2.3 *Given the neural network defined by (2.8), the neural network is stable; moreover, within each subset, S_i , there exist at most one undetermined neuron.*

We use this to show the following:

Theorem 2.4 *Given a neural network defined by (2.8), and satisfying (2.9), then the neural computation results in each subset, S_i , having at most one neuron on, the rest off.*

Proof: This proof is similar to the one for Theorem 2.2. We assume that the gain is sufficiently high so that the outputs of on or off neurons can be considered arbitrarily close to ± 1 respectively. We define $\epsilon = \min_i(\{|I_i| - |a|, \{(2 - |a|) - I_i\}\})$. From Theorem 2.3, we know that the network is stable, and that at most there is one undetermined neuron in each subset S_i . Note that this means that if neuron i is undetermined, then it is not connected to any other undetermined neuron, else there would be two undetermined neurons in the same subset. For neuron i , we define κ_i to be the total connection strength between neuron i and all other neurons that are on. Note $\kappa_i \geq 0$.

Using (2.8) in (2.2), we have

$$I_i^{tot} = ag(u_i^*) + I_i - 2\kappa_i. \quad (2.11)$$

Since κ_i is an integer, we have $|I_i^{tot}| > \epsilon$, and thus neuron i is determined. Neuron i is on or off depending on whether or not $\kappa_i = 0$. But $\kappa_i = 0$ —and neuron i is on—only if no other neurons in the same subset are on. Thus, all neurons are determined, and furthermore, they are on if and only if no other neurons belonging to the same subsets as neuron i are on, otherwise they are off. ■

Note from the proof that the neural network never chooses the trivial solution of all neurons off.

2.5 The Multiple Overlapping K-Winner-Take-All Circuit

We develop one final generalization to the Winner-Take-All Circuit, the Multiple Overlapping K-Winner-Take-All Circuit. This is identical to the Multiple Overlapping Winner-Take-All Circuit, except that in certain instances, we can define additional subsets, S'_i , where not at most one neuron, but at most K_i neurons are allowed to

turn on per S'_i . The only limitation on these subsets is that S'_i and S'_j are disjoint for all $i \neq j$, that is, each neuron can belong to no more than one set that allows more than one neuron to turn on.

For a given Θ and two sets of constraints $\{S_i\}$ and $\{(S'_i, K_i)\}$, we first define a Multiple Overlapping Winner-Take-All Circuit using Θ and $\{S_i\}$ in (2.8). Define $K_{max} \triangleq \max_i \{K_i\}$. We restrict the network slightly in that the diagonal elements, a , must satisfy $|a| < \frac{1}{2K_{max}}$. Using this as a basis, we incorporate the additional constraints.

$$\text{Let } J'_i \triangleq \begin{cases} k \text{ such that neuron } i \in S'_k & \text{if such a } k \text{ exists} \\ 0 & \text{otherwise.} \end{cases}$$

$$w'_{ij} \triangleq \begin{cases} w_{ij} - \frac{1}{2K_{J'_i}} & \text{if } i \neq j, J'_i = J'_j \neq 0, \\ a, |a| < \frac{1}{2K_{max}} & \text{if } i = j, \\ w_{ij} & \text{otherwise;} \end{cases}$$

$$t'_i \triangleq \begin{cases} t_i + \frac{|S_{J'_i}| - 2K_{J'_i} + 1}{2K_{J'_i}} & \text{if } J'_i \neq 0, \\ t_i & \text{otherwise;} \end{cases} \quad (2.12)$$

$$c'_i \triangleq c_i.$$

Comparing this definition with (2.3) we see that we are simply adding a K-Winner-Take-All circuit to each neuron set, S'_i , that is scaled by a factor of $\frac{1}{2K_i}$. We limit the inputs by:

$$|a| < |I_i| < \left(\frac{1}{K_{max}} - |a| \right), \quad (2.13)$$

This further reinforces the scaled K-Winner-Take-All concept. We now prove that this definition works in two theorems, as in the previous section.

In Appendix 2.A we prove the following:

Theorem 2.5 *Given the neural network defined by (2.12), the neural network is stable; moreover, within any subset S_i or S'_j , there exist at most one undetermined neuron.*

This is used to show the following:

Theorem 2.6 *Given a neural network defined by (2.12) and satisfying (2.13), then the neural computation results in each subset, S_i , having at most one neuron on, and each subset, S'_i , having at most K_i neurons on, the rest of the neurons being off.*

Proof: We make the assumption that the gains are all sufficiently large, and define $\epsilon = \min_i(\{|I_i| - |a|\}, \{(\frac{1}{K_{max}} - |a|) - I_i\})$. From Theorem 2.5, we know that the network is stable, and that there is at most one undetermined neuron in each constraint subset. For neuron i , we define κ_i to be the total connection strength between neuron i and all other neurons that are on, using only the weights w_{ij} . If $J'_i \neq 0$, we define κ'_i to be the number of neurons other than neuron i that are on in $S'_{J'_i}$; otherwise $\kappa'_i = 0$.

Suppose that neuron i is undetermined. By Theorem 2.5 we know that all of the other neurons that it is connected to are determined. Using (2.12) in (2.2) and defining $K_0 = 1$, we have

$$I_i^{tot} = ag(u_i^*) + I_i - 2\kappa_i + \frac{(K_{J'_i} - 1 - \kappa'_i)}{K_{J'_i}}. \quad (2.14)$$

Since κ_i and κ'_i are both integers, by (2.13), we have $|I_i^{tot}| > \epsilon$, contradicting that neuron i is undetermined. Thus all neurons are determined.

Note that if $\kappa_i \geq 1$ then $I_i^{tot} < -1$, and neuron i is off. If $\kappa_i = 0$ then neuron i is off whenever $\kappa'_i > (K_{J'_i} - 1)$. Thus all neurons are determined, and furthermore, no S_i has more than one neuron on, and no S'_i has more than $K_{J'_i}$ neurons on. ■

2.6 Neuron Gain

Each of the proofs in this chapter are true only with “high enough gain.” This section gives upper bounds on the minimum gain sufficient for any of the Winner-Take-All networks described. Unlike previous results, this will depend on the actual sigmoid function, f , that we use.

Theorem 2.8, Lemma 4 gives a lower bound. It requires $\gamma \geq \frac{\lambda}{a+1}$. Recall from (2.1) that λ_i is the total connection strength between neuron i and every neuron in the network, and that $\lambda = \max_i\{\lambda_i\}$. This lower bound assumes that all of the weights have been scaled so that the minimum non-zero weight is one, which for comparison

purposes we will assume is the case. In Section 2.5, this means multiplying all weights by $2K_{max}$.

Let ϵ be defined as in the proofs in the preceding as the closest that any I_i approaches the limits of (2.4), (2.9), or (2.13), as appropriate. Let us focus on the effect of the gain on a single neuron, i . Assume, as a worse case, that I_i is exactly ϵ away from the nearest limit. The limits on the inputs were set such that in a worst case $|I_i^{tot}| = \epsilon$. For concreteness, assume $I_i^{tot} > 0$ and that neuron i is on. These limits were set assuming that all of the neuron outputs were exactly ± 1 . In fact, since the total input to any neuron is finite, the outputs are all some non-zero amount away from ± 1 .

Suppose that every neuron is no more than δ away from the nearest of either $+1$ or -1 . In the worst case, they are all exactly δ away and each of these differences from the ideal value add up constructively. This implies that $I_i^{tot} = \epsilon - \lambda_i \delta$ and $u_i^* = \epsilon/\lambda_i - \delta$. By assumption neuron i is on, implying $u_i^* > 0$. Thus, we require $\epsilon/\lambda_i > \delta > 0$. If we put similar requirements on all of the neurons, we find that at worst $u_i^* = \epsilon/\lambda - \delta$, and $\epsilon/\lambda > \delta > 0$.

Intuitively, if δ is near zero, since the the inputs are finite and by construction bounded below ϵ , this implies that the gain is large enough to maintain the output within δ of ± 1 . As we decrease the gain, the smallest δ such that we can still maintain the output within δ of ± 1 increases. At the other extreme, if δ is near ϵ/λ , that is, the total input is near zero, then a large gain is necessary to drive the outputs toward ± 1 . As we decrease the gain, the largest δ such that we can still drive the outputs to within δ of ± 1 decreases. Eventually the gain will be reduced below a minimum so that the smallest δ of the first case is larger than the largest δ of the second case, and we cannot guarantee the network functionality in the worst case.

By definition, in this worst case $\delta = 1 - g(u_i^*) = 1 - g(\epsilon/\lambda - \delta)$. Using $g(x) = f(\gamma x)$ and solving for γ ,

$$\gamma = \frac{f^{-1}(1 - \delta)}{\epsilon/\lambda - \delta}. \quad (2.15)$$

As described in the previous paragraph, when $\delta \rightarrow 0$ or $\delta \rightarrow \epsilon/\lambda$ the necessary gain

Table 2.1: Comparison of necessary neuron gains

Sigmoid Name	$f(x)$	Upper Bound on γ_f^{min}
Piecewise Linear	$\begin{aligned} -1 & \text{ if } x < -1 \\ x & \text{ if } x \leq 1 \\ +1 & \text{ if } x > 1 \end{aligned}$	$\frac{\lambda}{\epsilon}$
Hyperbolic Tangent	$\frac{e^{2x} - 1}{e^{2x} + 1}$	$\frac{\lambda}{\epsilon} \log\left(\frac{4\lambda}{\epsilon}\right)$
Inverse Tangent	$\frac{2}{\pi} \tan^{-1}\left(\frac{\pi}{2}x\right)$	$\left(\frac{2\lambda}{\epsilon}\right)^2$
“Slow” Sigmoid	$\frac{x}{1 + x }$	$\left(\frac{2\lambda}{\epsilon}\right)^2$

approaches infinity. We could find the minimum of (2.15), but this leads to messy and non-instructive results. Since any positive $\delta < \epsilon/\lambda$ in (2.15) produces a large enough gain, for simplicity, we use $\delta = \frac{1}{2}\epsilon/\lambda$:

$$\gamma_f^{min} < \frac{2\lambda}{\epsilon} f^{-1}(1 - \epsilon/2\lambda). \quad (2.16)$$

We use Equation (2.16) to calculate a bound on the minimum necessary gain for some common sigmoid functions. We use approximations that are upper bounds on the right-hand side of (2.16) to put them in a comparable form. Table 2.1 tabulates the results.

The simplicity of the piecewise linear case allows us to find the minimum of (2.15) exactly. We include the piecewise linear function even though it technically does not satisfy our definition of a sigmoid (e.g., $f'(x) = 0, x > 1$). But, it does serve as a lower bound on the sufficient gain of any function. This follows since the piecewise linear function approaches the ± 1 limits at least as fast as any function constrained by $f'(x) \leq 1$. Thus, any $\gamma > \max\left(\frac{\lambda}{\epsilon}, \frac{\lambda}{a+1}, \gamma_f^{min}\left(\frac{\lambda}{\epsilon}\right)\right)$ is sufficient for the validity of

the theorems in this chapter when the node function is f . Note that of the sigmoid functions evaluated, the hyperbolic tangent function produces the smallest necessary gain.

2.7 Designing with Winner-Take-All Circuits

Having defined these Winner-Take-All Circuits, we show how to reduce the connectivity of the neural network, and how we can incorporate neural networks into larger systems. To reduce the connectivity, we note that two neurons, i and j , have the same input except that i does not connect to itself, but connects to j , and vice versa for j . Several researchers have noted that we can exploit this regularity [3, 4, 5]. They all fundamentally rely on the same principle. By making a weighted sum of all neurons only once for the whole circuit and providing a self connection in each neuron to negate the resulting feedback, we reduce the number of connections needed from $N(N - 1)$ to $3N$ for a single Winner-Take-All. This modified circuit is shown in Figure 2.2b. We will assume that all mutually inhibitory connections are made in this manner and will represent them schematically as shown in Figure 2.2c.

For the Multiple Overlapping Winner-Take-All, we could connect the network using the definition in (2.8). But every neuron is connected to every other neuron within each subset. This implies a total of $O(\sum_i |S_i|^2)$ connections. If instead we connect each subset in a separate Winner-Take-All as described above, we produce a network which is mathematically equivalent to (2.8), but now there are only $3|S_i|$ connections per subset S_i . This results in a total of $3\sum_i |S_i|$ connections in the entire network, yielding a significant savings.

The definitions and analysis of these Winner-Take-All circuits are all for a particular network scale. But as described in Section 2.2, we can scale \mathbf{c} arbitrarily, as well as \mathbf{W} , \mathbf{t} , and \mathbf{I} . Thus the network can be matched to the particular voltage and current levels appropriate for a particular implementation. From (2.1) we see that the threshold, t_i , and the external input, I_i , are fundamentally the same except for a change in sign. Since the only dependency on K in (2.5) occurs in the threshold,

by adjusting the external input to all of the neurons we can change the threshold, and so use the same circuit as a K -Winners-Take-All for any K . We also note that inputs from neurons outside of the Winner-Take-All are equivalent to the input and threshold, therefore these can be used to modify the value of either I_i or t_i .

The definition of the K -Winners-Take-All allows for a range of values for the external input, I_i , depending on the magnitude of the feedback, $|a|$. These can be used to indicate various levels of “priority” of the neurons. The neurons with the highest priority will then be the neurons which win the competition.

The analysis of Section 2.3 relies on initially identical internal states. This requires outside circuitry that can reset the network every time new winners must be selected. If we relax the requirement that the internal states are all initially identical, an inspection of the proof for Theorem 2.2 will show we only lose the ordering on the internal states, otherwise the result is the same. We summarize this in a separate theorem:

Theorem 2.7 *Given a neural network defined by (2.3), satisfying (2.4), let P be the number of neurons satisfying $|a| < I_i < (2 - |a|)$. Given these conditions, the neural computation results in K' neurons on, where $K' = \begin{cases} K & \text{if } P \geq K \\ K - 1 & \text{if } P < K \end{cases}$, the rest of the neurons off.*

This implies that we can use the K -Winners-Take-All in addition to the Multiple Overlapping Winner-Take-All in a completely asynchronous mode as a selector, or we can use the K -Winners-Take-All circuit as a discriminating selector. By using external inputs and neurons outside of the circuit as described previously, we can “program” the Winner-Take-All to compute particular functions. The details of how we can do this will be the subject of the next chapter.

2.8 Conclusions

This chapter defined several useful neural networks that embody constraints on the neuron outputs. The networks are all generalizations of the Winner-Take-All circuit.

We gave rigorous proofs that these definitions result in stable networks that satisfy all of the constraints. We described efficient methods for constructing these networks, as well as how they might be incorporated into larger neural systems.

2.A Appendix: Winner-Take-All Dynamics

Theorems 2.1, 2.3 and 2.5 follow directly from the following:

Theorem 2.8 *Given a neural network, $(\mathbf{W}, \mathbf{t}, \mathbf{c})$, with symmetric \mathbf{W} such that $w_{ii} = a \forall i$, and $|a| < \min_{i \neq j} \{w_{ij}\}$; given any subset, S , of neurons such that $\forall i \neq j \in S, w_{ij} \neq 0$; then the network is stable. Also, with large enough gain, the neural computation results in the internal state of at most one neuron in S not being bounded away from zero.*

Proof: By reference [2] and the hypothesis that \mathbf{W} is symmetric, we know that the network is stable. That at most one u_i cannot be bounded away from zero follows with a small modification to the proof given in [3] for a single Winner-Take-All network.

That proof is repeated here in detail for completeness and is obtained in four lemmas. We assume without loss of generality that $\min_{i \neq j} \{w_{ij}\} = 1$ for all w_{ij} . Also define $\lambda \triangleq \max_i(\lambda_i)$.

Lemma 1 *Given any asymptotically stable equilibrium state \mathbf{u}^* , we always have $\forall i \neq j \in S$:*

$$\lambda > a \frac{g'(u_i^*) + g'(u_j^*)}{2} + \frac{\sqrt{a^2(g'(u_i^*) - g'(u_j^*))^2 + 4g'(u_i^*)g'(u_j^*)}}{2}. \quad (2.17)$$

Proof: System (2.1) can be linearized around any equilibrium state \mathbf{u}^* :

$$\frac{d(\mathbf{u} - \mathbf{u}^*)}{d\tau} \approx L(\mathbf{u}^*)(\mathbf{u} - \mathbf{u}^*),$$

where

$$L(\mathbf{u}^*) = \mathbf{W} \cdot \text{diag}(g'(u_1^*), \dots, g'(u_N^*)) - \text{diag}(\lambda, \dots, \lambda).$$

A necessary and sufficient condition for the asymptotic stability of \mathbf{u}^* is for $L(\mathbf{u}^*)$ to be negative definite. A necessary condition for $L(\mathbf{u}^*)$ to be negative definite is for all 2×2 matrices $L_{ij}(\mathbf{u}^*)$ to be negative definite where

$$L_{ij}(\mathbf{u}^*) \triangleq \begin{pmatrix} ag'(u_i^*) - \lambda & w_{ij}g'(u_j^*) \\ w_{ji}g'(u_i^*) & ag'(u_j^*) - \lambda \end{pmatrix}, \quad (i \neq j).$$

This results from an infinitesimal perturbation of components i and j only.

Any of these matrices $L_{ij}(\mathbf{u}^*)$ has two real eigenvalues. Since the larger one has to be negative, we obtain:

$$\frac{1}{2} \left(ag'(u_i^*) - \lambda + ag'(u_j^*) - \lambda + \sqrt{a^2(g'(u_i^*) - g'(u_j^*))^2 + 4w_{ij}^2g'(u_i^*)g'(u_j^*)} \right) < 0,$$

where we use the symmetry $w_{ij} = w_{ji}$. The left side of the inequality is monotonically increasing with w_{ij}^2 . Since $w_{ij}^2 \geq 1 \forall i \neq j \in S$, it is also true when $w_{ij}^2 = 1$. Solving for λ proves (2.17). ■

Lemma 2 *If $a > -1$, and $x, y > 0$ then*

$$\frac{a}{2} + \frac{a^2(x-y) + 2y}{2\sqrt{a^2(x-y)^2 + 4yx}} > 0. \quad (2.18)$$

Proof: If $x \geq y$, then the lemma is true for $a \geq 0$. If $x < y$ and $a \geq 0$ then it is also true since from (2.18) we get:

$$\frac{a}{2} \left(1 + \frac{a(x-y) + 2y/a}{\sqrt{a^2(x-y)^2 + 4yx}} \right) > \frac{a}{2} \left(1 + \frac{a(x-y)}{\sqrt{|a(x-y)|^2}} \right) = 0.$$

Finally, let $a < 0$. For any a with $a^2 < 2$ (and now $-1 < a < 0$), the second term in (2.18) is positive. Therefore, when $a < 0$ the lemma is true if

$$\begin{aligned} \frac{a^2}{4} &< \left(\frac{a^2(x-y) + 2y}{2\sqrt{a^2(x-y)^2 + 4yx}} \right)^2 \\ &= \frac{a^2}{4} \left(1 + \frac{4y^2(\frac{1}{a^2} - 1)}{a^2(y-x)^2 + 4yx} \right), \end{aligned}$$

which is true for $|a| < 1$. Thus for all $a > -1$ (2.18) holds. ■

Lemma 3 Equation (2.17) implies $\forall i \neq j \in S$:

$$\min(g'(u_i^*), g'(u_j^*)) < \frac{\lambda}{a+1}. \quad (2.19)$$

Proof: Consider the function h of three variables:

$$h(a, g'(u_i^*), g'(u_j^*)) = a \frac{g'(u_i^*) + g'(u_j^*)}{2} + \frac{\sqrt{a^2(g'(u_i^*) - g'(u_j^*))^2 + 4g'(u_i^*)g'(u_j^*)}}{2}.$$

Differentiating h with respect to $g'(u_j^*)$, we obtain:

$$\frac{\partial h}{\partial g'(u_j^*)} = \frac{a}{2} + \frac{a^2 g'(u_j^*) + (2 - a^2)g'(u_i^*)}{2\sqrt{a^2(g'(u_i^*) - g'(u_j^*))^2 + 4g'(u_i^*)g'(u_j^*)}}$$

By Lemma 2, this is positive since $|a| < 1$. Thus, if $g'(u_i^*) \leq g'(u_j^*)$ (without loss of generality), we have:

$$h(a, g'(u_i^*), g'(u_j^*)) \geq h(a, g'(u_i^*), g'(u_i^*)) = (a+1)g'(u_i^*)$$

This, combined with (2.17), yields:

$$g'(u_i^*) < \frac{\lambda}{a+1},$$

which is Lemma 3. ■

Lemma 4 If the gain $\gamma \geq \frac{\lambda}{a+1}$, then at most one component in S can not be bounded away from zero.

Proof: At most one u_i^* in S satisfies:

$$g'(u_i^*) \geq \frac{\lambda}{a+1}, \quad (2.20)$$

since if two u_i^*, u_j^* both satisfy (2.20), then (2.19) would be violated.

Next, choose $\gamma > \frac{\lambda}{a+1}$. Since $g'(x) = \gamma f'(\gamma x)$ and $f'(x) \leq f'(0) = 1$, we have $g'(u_i^*) \leq g'(0) = \gamma$. Using the fact $f'(u)$ (and therefore $g'(u)$) is continuous at $u = 0$, any u_i^* with $g'(u_i^*) < \frac{\lambda}{a+1} < \gamma$ can be bounded away from zero. Thus, only the at most one u_i^* in S that satisfies (2.20) cannot be bounded away from zero. ■

Bibliography

- [1] Mead, C. A., *Analog VLSI and Neural Systems*. Addison-Wessley, Reading, Massachusetts, 1989, p. 192.
- [2] Hopfield, J. J., “Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons,” *Proc. Nat. Acad. Sci. USA*, Vol. 81, pp. 3088–3092, May 1984.
- [3] Majani, E., Erlanson, R., Abu-Mostafa, Y., *On the K-Winners-Take-All Network*. ed. Touretzky, D. S. *Advances in Neural Information Processing Systems I*. Morgan Kaufman Pub., San Mateo, CA. 1989. pp. 634–642.
- [4] Eberhardt, S. P., Daud, T., Thakoor, A. P., Brown T. X, *Limited-Connectivity VLSI Network for Resource Allocation* Submitted to IEEE Neural Information Processing Systems Conference, Denver, CO, November, 1990.
- [5] Lazzarro, J., Ryckebusch, S., Mahawold, M. A., Mead, C. A., Winner-Take-All Networks of $O(N)$ Complexity, ed. Touretzky, D. S., *Advances in Neural Information Processing Systems I*, Morgan Kaufman Pub., San Mateo, CA, 1989, pp. 703–711.

Chapter 3

Controlling Circuit Switching Networks

The price paid for this great increase in combinatory power is the current difficulty of controlling networks of many stages. This difficulty is technological, though, and will decrease as improved circuits are developed.

—V. E. Beneš [1, p. 121] on switching networks.

3.1 Introduction

This chapter will take the principles and theorems developed in the last chapter and apply them to problems in switching. In general, switches are composed of many highly interconnected simple elements. This topological similarity to neural networks will be exploited to design neural network controllers of switch functions. We first give relevant background on switching theory before designing two neural network solutions to problems from circuit switching.

3.2 Background on Switches

This section will introduce some basic concepts of switching theory relevant to our applications. More detailed development of switching theory can be found in any of several references, [1, 2, 3]. We abstract a switch as a device that takes a set of N signal inputs and reproduces them in any permuted order at the output. We restrict our discussion to square switches, where the number of inputs and outputs

Figure 3.1: The $N \times N$ crossbar switch concept.

is the same, although the concepts that we develop readily generalize to non-square switches. If the inputs and outputs of the switch are N distinct lines, then we refer to the switch as a *space* switch. Alternatively, the switch could have one line for the input and one line for the output. In this case, the inputs and outputs are N distinct blocks of data in which the order that the blocks are sent is permuted by the switch. Such a switch is referred to as a *time* switch. Since each time switch is equivalent to some space switch, we will restrict our discussion to space switches [2, pp. 114–117].

The basic switch is the $N \times N$ crossbar switch. Conceptually, it comprises a grid of N wires by N wires with a closable contact at each of the crosspoints as shown in Figure 3.1a (Figure 3.1b shows the schematic representation). A *legal call request* in a general switch is a request for a connection from one unused input to one unused output. A call request is *blocked* if the connection can not be put up through the switch. This occurs due to constraints from the architecture of the switch, or by available routes being used by calls already put up through the switch. Already put

up calls that block a call request are identified as *blocking calls*. A switch is *strictly non-blocking* if any sequence of call requests, with intervening call disconnects, can be put up as each request arrives, no matter what algorithm is used for routing calls. The $N \times N$ crossbar is the prototypical example of the strictly non-blocking class. While such a switch is desirable, unfortunately it is at the expense of N^2 crosspoints.

The crosspoint count of a switch is often used as a measure of its cost or complexity. We desire to reduce the number of crosspoints. Usually this is achieved by building larger switches from stages of smaller crossbar switches. Figure 3.2a shows a general three-stage example built from smaller $r \times r$ and $n \times n$ crossbars, while a specific example for $n = 2, r = 4$ is shown in Figure 3.2b. For an appropriate choice of n and r , such a multistage switch has a reduced number of crosspoints. Unfortunately, multistage switches are often not strictly non-blocking. Yet they may fall into the broader class of *non-blocking with call rearrangement*, or rearrangeable switches. A switch is *rearrangeable* if any sequence of legal call requests can be connected through the switch, as long as the routing of calls already in progress can be rearranged. Alternatively a switch is rearrangeable if simultaneously given any legal set of calls (each input and output used at most once), the switch can put up all of the calls.

A switch can be blocking in various senses. Yet switches can be designed so that statistically the probability that a call is blocked in whatever sense is small, even when the call traffic being routed through the switch is high. Typically this requires several stages, and the provision of many alternative routes through which a call can be put up [3, pp. 526–552]. In the case of rearrangeable switches, however, we are not relying on probability, but want to be able to rearrange with certainty.

Two questions therefore arise. First, given a large multistage switch, a set of calls already put up, and a legal call request, which of the many routes can accommodate the new call? Second, given a rearrangeable switch, a set of calls already put up, and a legal call request, how do we rearrange to accommodate the new call request? We address each of these questions separately, and in turn.

Figure 3.2: A general three-stage switch.

3.3 Large Multistage Switching Networks

A multistage switch is shown in Figure 3.3. As mentioned earlier, these switches are typically blocking. But, by having a large number of routes available between any input and output of the switch, the probability that all these routes are blocked simultaneously is very low. The switch shown in Figure 3.3 has 16 different routes between any inlet-outlet pair (16 *distributions*). A practical switch would have more and bigger crossbar switches in each stage. This implies many possible routes between each stage. The AT&T ESS #4 toll switch, for example, has 1,920 (16×120) possible

Figure 3.3: Multistage switch.

routes between any input and output pair. The problem is that, in such a switch, finding an unblocked route from the many possible routes is a time consuming task, especially as the switch becomes loaded with calls. The problem can be generalized to the case when the outlet stage is not unique, as for toll switches where any one of several trunks between two offices would be acceptable.

An easy way to specify a route is to specify the crossbar switch that it passes through at each stage. For example, the path highlighted in Figure 3.3 is (1, 4, 3, 4, 2). Most practical switching networks have at most one connection between any two crossbar switches, so that this specification is well defined.

The simplest parallel algorithm to find a route checks all possible routes at the same time, and then chooses one unblocked route if it exists. More specifically, we search forward to find all the crossbar switches that have an unblocked route to the call request's inlet stage. We then use this information to search backward to find all the crossbar switches that have an unblocked route to both the call request's outlet and inlet stages. If, as we search backward, we order our search so that we limit ourselves to choosing a single crossbar in each stage, and subsequently consider only routes through this crossbar, then our search will finish with a list of crossbars that denote a single unblocked path.

Figure 3.4: Multistage switch with five calls already put up.

Figure 3.4 shows the switch of Figure 3.3 with five calls already put up. A new call request arrives at the topmost inlet for the topmost outlet. To start our parallel search we mark all switches that have a free path to the first inlet switch. The easiest way to do this is to start at the first stage and mark the request's inlet, Switch 1. Then go to the second stage and mark all of the switches that have a free connection to a marked switch in the previous stage, Switch 1 and 3, continuing this procedure until the final stage is reached. This results in all of the shaded switches in Figure 3.4 being marked.

Now we do the backward search. Of all the switches that are marked in the last stage, Stage L , we choose the one at the call request's outlet, Switch 1, and use it as the last switch in our route. If the call request outlet was not marked, then we would know that no free connection exists from the outlet to the inlet, and would stop. Otherwise, as in this example, we know that there is some route among the marked switches; we must choose one. In Stage $L - 1$, we choose one of the switches that is already marked and has a connection to the switch chosen for the route through

the next (closer to the outlet) stage. The switch chosen, e.g., Switch 2, is used for our route through Stage $L - 1$. This procedure is continued at Stage $L - 2$, and so on, until the first stage is reached, resulting in an unblocked route through the crosshatched switches (1, 1, 1, 2, 1).

This method of choosing a route will always find one, if it exists. Since we only choose among the switches that were marked in the feedforward search, we know that every switch chosen for the route always has some connection among the marked switches in the remaining unspecified stages to the input. We only need to choose switches for our route that have a connection to the switch already chosen in the next stage. Simply choosing one of the marked switches in each stage does not work. For example, the route (1,1,1,1,1) contains only marked switches, but does not designate a free route.

The procedure described above is exactly the approach taken by our neural network algorithm.

3.4 Finding Routes using Neural Networks

Figure 3.5 shows the basic neural network topology. We establish a group of neurons for each of the crossbar switches that make up the switching network. If two crossbars are connected in the switching network, then the corresponding neuron groups are connected in the neural network. The connections in the neural network are gated so that if the corresponding connection is used in the switching network by an already put up call, then that connection is open-circuited in the neural network. The neural network has an external connection for each of the inlet and outlet stage switches. The neuron group consist of three neurons, a *feedforward*, a *feedback*, and a *path* neuron. The feedforward neuron turns on if it is connected to a feedforward neuron in the previous stage that is already on. In the first stage, the feedforward neuron turns on if there is an excitatory input at this inlet-stage neuron group. From this recursive definition, we see that an excitatory signal applied at the input corresponding to the desired inlet-stage switch will result in all feedforward neurons that have an unblocked

Figure 3.5: Neural network for switch path search.

route to this switch turning on.

The path and feedback neurons work together to do the backward search. They operate similar to the feedforward neurons. A feedback neuron turns on if it is connected to a path neuron in the subsequent stage that is already on. In the last stage, the feedback neuron turns on if there is an excitatory input at this outlet-stage neuron group. The path neurons in each stage are designed as a Winner-Take-All circuit. As described in Section 2.7, we use the inputs from the feedforward and feedback neurons in each group to “program” their associated path neuron. In addition to the standard Winner-Take-All definition, within each group we add a +1 connection from the feedforward to the path neuron and a +1 connection from the feedback to the path neuron. The external input we set as -1.5 for all path neurons. The effective external input (external input I_i plus connections from neurons outside of the Winner-Take-All network) for a path neuron is:

$$I_i^{effective} = s_{iff} + s_{ifb} - 1.5,$$

where s_{iff} and s_{ifb} are the states of the feedforward and feedback neurons, i.e., +1 if

on, -1 if off. Thus, assuming that $w_{ii} = 0$, a path neuron can only turn on if both the feedforward neuron is on (there is an unblocked path to the inlet), and the feedback neuron is on (there is an unblocked path to the inlet).

By simply applying an excitatory input to the desired inlet-outlet pair, the network follows the massively parallel algorithm described in the previous section: The feedforward neurons search forward; the feedback neurons search backward; and the path neurons guarantee that as we search backward we limit ourselves to a single crossbar in each stage and subsequently only consider routes through this crossbar.

Figure 3.6 shows the neural network finding a path for the switch of Figure 3.4. In addition to finding point to point routes, the problem of toll switches is solved also. An excitatory input can be applied not just to one outlet, but to each eligible outlet. The Winner-Take-All in the last stage will guarantee that the call is routed to only one outlet.

This network is a straightforward solution to the path search problem. A switch with N input/output lines will have N connections between each stage. The number of crossbars in each stage is not more than N . The network will not settle until the signals propagate from the input to the output and back. Therefore, if there are M stages in the switch, the number of neurons and connections are both $O(NM)$, and the time to produce an output is $O(M)$.

3.5 Rearrangeable Switches

The switch shown in Figure 3.2 is an example of a $N \times N$ Slepian rearrangeable switch [4]. This is characterized by three stages that we denote as the *inlet*, *center*, and *outlet* stages. The inlet stage comprises r $n \times n$ crossbars. Each of these connects to each of n $r \times r$ center-stage crossbars that in turn connect to each of the r $n \times n$ outlet-stage crossbars. Note that $N = rn$, and that the minimum number of crosspoints is obtained when $r \approx n \approx \sqrt{N}$. Conventionally, the inlet and outlet switches are numbered from 1 to r , and the center-stage switches labeled A, B, etc.

M. C. Paull analyzed this switch in detail [5]. He showed that this switch is

Figure 3.6: Neural network searching for a path.

blocking in the strong sense that, no matter how the calls are put up through the switch, there always exists a sequence of call requests with intervening call disconnects that can force a block. Although blocking can not be avoided, this switch can put up any legal call request if we allow calls-in-progress to be rerouted through the switch before we put up a new call request.

To aid in analyzing the behavior of these switches, we introduce *Paull matrices*. These provide a format for describing the state of the switch. An example is shown in Figure 3.7, with the correspondence to the switch given. A *symbol* Q in entry (i, j) of the matrix implies that a call originating from inlet switch i is routed through center switch Q to outlet switch j . The switches in each stage have exactly one connection to each of the switches in the next stage so that this state description is well defined. Since the inlet and outlet switches are composed of strictly non-blocking crossbars, we need, and do not distinguish between which of the n external connections are used on these switches. Calls and call request are therefore denoted simply by (i, j) .

The structure of the switch limits the Paull matrices that we can have. Each row of the matrix represents a specific inlet-stage switch. Since only n calls can be routed through this inlet switch, and each must be routed through a different center-stage switch, each of the n distinct symbols can appear in a row only once. For the same reason and because of the symmetry of the switch, this is also true for the columns of the matrix. Multiple symbols can appear in a single entry, representing calls that are connected to the same inlet- and outlet-stage switches. A call, (i, j) , is blocked if every symbol is used in row i and column j together. Figure 3.8 shows an example of a blocked call request.

Paull proved that the following algorithm will always find an unblocked rearrangement for a legal call request (i, j) :

1. Choose one symbol, Q , that does not appear in row i . Similarly, choose one symbol, R (possibly the same symbol as Q), that does not appear in column j . These two symbols must exist or else every connection to the inlet and/or outlet is used and (i, j) is not a legal call request. Having these two symbols, choose one, without loss of generality Q . Define $\tilde{r} = i$, and $\tilde{c} = j$ (the tildes

Figure 3.7: Paull matrix.

Figure 3.8: A blocked call request at $(2, 3)$.

are to distinguish these from the number of inlet/outlet stages r and from the neuron capacitances c_i).

2. If there is some symbol (possibly Q) that does not appear in either row \tilde{r} or column \tilde{c} , place that symbol in entry (\tilde{r}, \tilde{c}) and stop; no calls are blocked. Otherwise, there is another row, \tilde{r}' , such that (\tilde{r}', \tilde{c}) contains a Q . Place Q in entry (\tilde{r}, \tilde{c}) of the matrix and remove the Q from (\tilde{r}', \tilde{c}) . Go to step 3.
3. If there is some symbol (possibly R) that does not appear in either row \tilde{r}' or column \tilde{c} , place that symbol in entry (\tilde{r}', \tilde{c}) and stop; no calls are blocked. Otherwise, there is another column, \tilde{c}' , such that (\tilde{r}', \tilde{c}') contains an R . Place R in entry (\tilde{r}', \tilde{c}) of the matrix and remove the R from (\tilde{r}', \tilde{c}') . Define $\tilde{r} = \tilde{r}'$ and $\tilde{c} = \tilde{c}'$ and go to step 2.

We denote this as Paull's algorithm. Figure 3.9 shows an example of this algorithm finding an unblocking rearrangement for the situation in Figure 3.8. Note how a blocked call causes a chain of rearrangement steps, alternately replacing A for B , and vice versa. Slepian showed that this algorithm uses at most $(2r - 2)$ of these steps [5], while later Paull showed that the best algorithm uses at most $r - 1$. Furthermore, Paull showed that situations exist that, even with the best algorithm, require $(r - 1)$ rearrangement steps.

Figure 3.9: Unblocking rearrangement for call request.

3.6 The Neural Network Solution to the Rearrangement Problem

The key to utilizing the parallelism of a neural network is matching the network topology to the problem as closely as possible. We now show that Paull’s algorithm fits readily into a neural network solution. For each entry in Paull’s matrix we create n neurons. The neurons are labeled with A, B, etc., corresponding to the labels on the center-stage switches. The state of the neurons corresponds to the current state of the switch. If neuron Q is on in entry (i, j) , then a call from inlet i to outlet j is routed through center-stage switch Q , and vice versa.

A neural computation is initiated when a call request arrives. The inputs are changed to “program” the network for the new call, and the network is allowed to evolve. The new call assignments are determined from the final state of the network.

The state of the network is maintained until the next call arrives, at which time the inputs are changed again, and the network evolves to the next state. When a call is disconnected, the corresponding neuron in the network is turned off.

We will concentrate on what happens at call arrivals. The question is how to connect the neurons so that the network can properly compute the assignments. From the previous section, we see the following constraints are placed on these neurons:

- In each row of the matrix, no two neurons with the same label may both be on.
- In each column, no two neurons with the same label may both be on.
- In each entry, (i, j) , only as many neurons may be on as there are calls (i, j) .

These constraints imply that we have a three-dimensional grid of neurons, with each axis around a neuron having its own mutual inhibition as shown in Figure 3.10. We could in fact design this network according to a Multiple Overlapping K-Winner-Take-All design of Section 2.5. Although this would guarantee a network that is stable and never violates the first two constraints, it would not guarantee that every call is put up. To achieve this we take a slightly different approach. We view the network as a set of Winner-Take-All networks—one for each entry—that communicate their states to other Winner-Take-All networks in the same row and column.

As a convenience we define:

\tilde{r}_i = the row in Paull's matrix in which neuron i is located.

\tilde{c}_i = the column in Paull's matrix in which neuron i is located.

Q_i = the symbol in Paull's matrix that neuron i represents.

For a switch with r inlet and outlet stages and n center stages, the connection weights are;

Figure 3.10: Three-dimensional neuron grid.

$$\begin{aligned}
w_{ij} &= \begin{cases} -1 & \text{if } \tilde{r}_i = \tilde{r}_j, \tilde{c}_i = \tilde{c}_j, \text{ and } i \neq j, \text{ (same entry)} \\ -0.5 & \text{if } Q_i = Q_j, \tilde{r}_i = \tilde{r}_j, \text{ and } i \neq j, \text{ (same row)} \\ -0.5 & \text{if } Q_i = Q_j, \tilde{c}_i = \tilde{c}_j, \text{ and } i \neq j, \text{ (same column)} \\ 0 & \text{otherwise;} \end{cases} & (3.1) \\
t_i &= r + n - 1; \\
c_i &= C,
\end{aligned}$$

where C is some positive constant. This is, in fact, the Multiple Overlapping Winner-Take-All implied by Figure 3.10, except that a Winner-Take-All within an entry

has twice the connection strength of the Winner-Take-All connections along rows and columns. This implies that we can sum the inhibition along each axis once, as described in Section 2.7, with a resulting savings in connectivity.

Let K_{ij} be the number of calls from inlet i to outlet j including the new call request. Let s_i be the state of neuron i before the computation starts, that is, $+1$ if i is on, -1 if off. For each neuron i ,

$$I_i = 2K_{\tilde{r}_i\tilde{c}_i} - 0.5s_i + x_i, \quad (3.2)$$

where x_i is some control input such that $|x_i| < 0.25$.

Recall from Chapter 2 that a neuron is determined if its internal state can be bounded away from zero, and thus with high enough gain the neuron output is arbitrarily close to ± 1 .

Theorem 3.1 *The network defined in (3.1) with inputs (3.2) is stable and all neurons are determined.*

Proof: The results of Appendix 2.A can be applied to show that this network is stable and results in at most one undetermined neuron per Winner-Take-All. Let κ_i^e be the number of neurons, other than neuron i , that are on and in entry $(\tilde{r}_i, \tilde{c}_i)$. Let κ_i^{rc} be the number of neurons other than neuron i that are on and correspond to symbol Q_i in row \tilde{r}_i or column \tilde{c}_i . Using Equations (3.1) and (3.2) in (2.2):

$$\begin{aligned} I_i^{tot} &= (r - 1 - \kappa_i^e) - \kappa_i^e + (n - 1 - 0.5\kappa_i^{rc}) - 0.5\kappa_i^{rc} + I_i - t_i \\ &= 2K_{\tilde{r}_i\tilde{c}_i} - 2\kappa_i^e - \kappa_i^{rc} - 1 - 0.5s_i + x_i. \end{aligned} \quad (3.3)$$

Suppose that the state of neuron i can not be bounded away from zero. Since s_i is ± 1 , then for any integers $K_{\tilde{r}_i\tilde{c}_i}$, κ_i^e , and κ_i^{rc} ; we have $|I_i^{tot}| \geq 0.25$, contradicting that it is not determined. ■

This theorem allows us to treat all neurons as either on or off and will greatly simplify our subsequent discussions.

Paull's algorithm consists of a series rearrangement steps (i.e., Steps 2 and 3) where one symbol is placed in entry (i, j) and then possibly removed from another entry in the same row or column. To perform one of these steps at entry (i, j) , we require the following: symbols that are on in entry (i, j) at the beginning of the step remain on; the rearrangements are *forward*, that is, if Q was just removed from entry (i, j) , it is not immediately placed there again; finally, when possible, place a symbol that does not appear at all in row i or column j , otherwise place a symbol that only appears once in row i or column j , and remove the already placed symbol. We assume that at the time when we want to place a symbol at (i, j) , the matrix is in a *valid state*—all constraints are met except for one unplaced symbol at (i, j) .

To aid our discussion we identify four types of neurons, classed according to their state and the value of κ_i^{rc} defined in the proof of Theorem 3.1:

1. Neurons that are already on.
2. Neurons that are off and $\kappa_i^{rc} = 0$ (Q_i not in row \tilde{r}_i or column \tilde{c}_i).
3. Neurons that are off and $\kappa_i^{rc} = 1$ (Q_i in one of row \tilde{r}_i or column \tilde{c}_i).
4. Neurons that are off and $\kappa_i^{rc} = 2$ (Q_i in both row \tilde{r}_i and column \tilde{c}_i).

Note that $\kappa_i^{rc} \leq 2$, since $\kappa_i^{rc} > 2$ would imply more than one of the same symbol in row \tilde{r}_i or in column \tilde{c}_i , contradicting that the network is in a valid state.

Theorem 3.2 *Given a neural network in a valid state and that we want to place a symbol in entry (i, j) as part of Paull's algorithm, then:*

1. *The Type 1 neurons at (i, j) will remain on.*
2. *The chaining is forward, i.e., a symbol that was removed is not placed again.*
3. *If a Type 2 neuron exists, one will turn on, otherwise one Type 3 will turn on and the blocking neuron in the same row or column will turn off.*

Proof: By definition of a valid state, if neuron z is on, no other neuron labeled Q_z in row $\tilde{r}_z = i$ and in column $\tilde{c}_z = j$ is on. Therefore, $\kappa_z^{rc} = 0$. From Theorem 2.2 we know that at most K_{ij} neurons turn on. This implies that as long as neuron z is on, $\kappa_z^e < K_{ij}$. Thus $I_i^{tot} > 0.25$, neuron i will remain on, and the first conclusion is proved.

A symbol will not be removed unless the same symbol was already placed in the same row or the same column. Thus a Type 2 neuron can not have just turned off, so its turning on is consistent with forward chaining. A symbol is removed when only one of the same symbols was just placed in the same row or column, implying that no Type 4 neuron can have just been turned off. Looking at a Type 3 neuron z , we know that since none of the Type 1 neurons turn off and that there are at least $K_{ij} - 1$ of these, then $\kappa_z^e \geq K_{ij} - 1$. Using this data in Equation (3.3):

$$I_z^{tot} < -0.5s_z + x_i \quad (3.4)$$

This implies that if $s_z = 1$, then neuron z will remain off. But $s_z = 1$ only if neuron z started the rearrangement being on, that is, if it was turned off during the rearrangement. Thus no neuron that was just turned off will turn on and the second conclusion is shown.

Suppose neuron y is Type Y , neuron z is Type Z , and $1 < Y < Z$. Since in the period preceding the competition, one of the Type 3 could have been a Type 1 and $I_y \geq I_z$, we can assume that $u_y \geq u_z$ at the start of the competition. During the competition, $I_y > I_z$. It follows directly from the proof of Theorem 2.2 that, with this ordering on the external inputs and initial ordering on the internal states, a neuron of Type 2 will turn on if it exists.

Assume that no Type 2 neurons exist. We know from the first step of Paull's algorithm that if no Type 2 neurons exist there are at least two Type 3 neurons. In the chain of rearrangements, the steps alternately take place along rows, and columns. As part of his analysis, Paull showed that each column and each row could be used at most once. In particular, only one neuron can turn off per entry per rearrangement. This implies that at most one Type 3 in this entry started the rearrangement initially

Figure 3.11: Additional memory neuron.

on. Thus there is at least one Type 3 neuron which in the initial state was off. If none of these neurons turn on, then for every such neuron, z , $\kappa_z^e = K_{ij} - 1$, $\kappa_z^{rc} = 1$, $s_z = -1$, and $I_z^{tot} > 0.25$, contradicting that none are on. So, at least one Type 3 neuron will turn on. Suppose more than one of these neurons turns on, then since all of the $K_{ij} - 1$ Type 1 neurons will remain on, $\kappa_z^e \geq K_{ij}$ for all neurons, and $I_z^{tot} < 0.25$. This implies all neurons are off, a contradiction. Thus only one neuron turns on, if not a Type 2 then certainly a Type 3.

If the neuron that turned on is Type 3, then it is blocked by some neuron z in the same row or column. Since neuron z was on before, it is now a Type 3 neuron. Since $s_z = 1$, by (3.4) it will turn off. Thus, the proof of the theorem is complete. ■

We digress at this point to discuss an interesting modification of the neural network defined so far. One function that needs to be performed after every calculation is the inverting of s_i for all neurons corresponding to calls that were rearranged. One way to do this automatically is to add to each neuron i in the network a *memory* neuron. Figure 3.11 shows the modified circuit. The connections between a neuron and memory neuron pair are chosen so that if the neuron is on, then the memory neuron is off, and if the neuron is off, then the memory neuron is on. The memory neuron gets its name since $(C/\lambda)_{memory} \gg (C/\lambda)$, resulting in a much longer time constant. Because of this, the memory neuron remains in its state for a long time

after the primary neuron changes. By making the connection to neuron i from the memory $+0.5$, it serves exactly as s_i . This shows how we can use non-uniform time constants and exploit them to our advantage. We assume that all of these networks incorporate s_i in this way.

The example of a rearrangement from Figure 3.9 is redone using a neural network in Figure 3.12. This shows all the basic features of the circuit. The evolution of the neural circuit is virtually identical to the algorithm described by Paull.

Theorem 3.2 shows that the neural network, as it is defined, follows Paull's algorithm exactly, except that it does not alternately rearrange the same two symbols Q and R defined in step one of the algorithm. The previous example works, though, since there are only two symbols to arrange (i.e., $n = 2$). In general, for $n > 2$, this exception results in states that violate the constraints. Furthermore, it can result in rearrangements involving many more than $2r$ rearranged calls. The network needs to globally communicate the symbols Q and R to alternately rearrange. We are assuming that no Type 2 neurons exist at (i, j) , otherwise by Theorem 3.2 we know that the neural network will behave correctly.

Figure 3.13 shows the modifications necessary to force the network into following the algorithm. The additional circuitry consists of a set of detectors labeled D1, D2, and D3. Figure 3.13a shows the D1 detector added to each neuron-memory neuron pair. The connections to a D1 are such that it only turns on if both neurons in the pair are on. This occurs only if the neuron was off and recently turned on, thus the D1 detectors detect neurons that turn on during the course of the rearrangement.

Figure 3.13b shows the detectors located at the end of each row of the neuron grid. A similar set appears at the end of each column. For each symbol, Q , there are detectors denoted by $D2_Q$, and $D3_Q$. The $D2_Q$ detectors simply collect the outputs of all of the D1's associated with symbol Q in this row. It turns on if any of these neurons turn on. The output of $D2_Q$ is a small positive signal fed back to each of the neurons labeled Q in every other row. Thus a neuron i turning on causes its D1 neuron to turn on, which in turn causes the associated $D2_{Q_i}$ neuron in row \tilde{r}_i to turn on, which then feeds a small positive signal back to the neurons labeled Q_i in every

Figure 3.12: Neural network solving blocking situation.

Figure 3.13: Additional circuitry to force the network to follow Paull's algorithm.

other row. This means that once a symbol is placed, it will be favored over otherwise equal symbols in all subsequent rearrangement steps. As a result, the first symbol placed chooses the first symbol with which to rearrange.

We need to choose an appropriate second symbol to rearrange with. For a blocked call request at entry (i, j) (i.e., no Type 2 neurons), a sufficient condition for choosing symbols Q and R to rearrange is that they can not appear together in row i nor in column j . The neuron that turns on will force another neuron either in the same row or the same column to turn off. Assume that it is in the same row. Since there are no Type 2 neurons, any symbol not appearing in row i will be in column j . This implies that if the symbol placed in this second rearrangement is not in row i then it must be in column j , and will satisfy the conditions for the second symbol chosen to rearrange.

The $D3_Q$ neurons guarantee this result, as we now show. A $D3_Q$ neuron turns on if a neuron Q is on in the same row and one of the other $D2_R$, $R \neq Q$, are on, that is, if symbol R was placed in this row and symbol Q is also in this row. The output of $D3_Q$ is a small negative signal to each of the neurons labeled Q in this row. Thus, in the second rearrangement step, neurons corresponding to symbols already placed in this row are disfavored. Further, out of the remaining neurons, the neuron that turns on will, through its D1 and D2 detectors, favor neurons with the same symbol in all further competitions. Thus the network chooses two appropriate symbols to rearrange in the first two rearrangement steps, and subsequent steps will favor these two symbols.

Since the detector signals add to all of the neurons in the same row, an efficient way to distribute this signal is to add them once to the inhibition summers at the end of the row as shown in Figure 3.13. Each neuron receives control signals from up to $2(r - 1)$ D2 and two D3 detectors. To guarantee our previous results, we require that the total control signal, x_i , from the D2 and D3 detectors never exceeds the limit given by $|x_i| < 0.25$. Figure 3.14 shows an example of this network in action. Again the neural network follows the algorithm as stated by Paull.

We check that the solution is reasonable. Table 3.1 tallies the number of neurons

Figure 3.14: Solving the rearrangement problem for $n = 3$.

Table 3.1: Tally of the neural rearranger size.

Type	Total Number	Total Number of Connections from
Neuron	r^2n	$6r^2n$
Memory	r^2n	$2r^2n$
D1	r^2n	$2r^2n$
D2	$2rn$	$2r^2n + 2rn^2 - 4rn$
D3	$2rn$	$2rn$
Total	$3r^2n + 4rn$	$11r^2n + 2rn^2 - 2rn$

and connections as a function of the parameters n and r . Under the assumption that $r \approx n \approx \sqrt{N}$, the number of neurons and the number of connections are both $O(N^{3/2})$. This is equivalent to the number of crosspoints in the switch. The time to find the rearrangement is given by Paull's algorithm and is $O(r) = O(\sqrt{N})$. Therefore the neural network solution to the switch rearrangement problem is approximately as complex as the switch itself, but no more complex, and uses time as fast as the best algorithm.

As a conclusion to this section, we describe an interesting application of this controller. To reduce the number of crosspoints below the minimum possible in a three stage Slepian switch, we can construct each of the center stage switches out of a smaller three stage network. This recursive construction, if taken to its limit, leads to a switch constructed solely out of 2×2 crossbars: $N/2$ 2×2 inlet/outlet stages connected to two center stage switches that each have $N/4$ 2×2 inlet/outlet switches, and so on. This is known as a Beneš network [1]. It has a total of $O(N \log(N))$ crosspoints. How can we find routes through this switch?

We construct a hierarchical neural network. The first level is the neural network of Figure 3.10. The outputs of this network are the on neurons that indicate the necessary connections through center-stage switches A or B . But, these requests can be used to control the next level, the two smaller controllers for switches A and B . Each of these in turn control their own two center stage switches. Continuing in this manner until all the center most 2×2 switches are reached. Thus we can use the

neural network to control the many switches that comprise this efficient but difficult to control switch design.

3.7 Conclusion

This chapter took the design techniques from Chapter 2 and implemented two switching control algorithms using neural networks. In both cases we were able to take a circuit with known characteristics, the Winner-Take-All, and utilize its capabilities within a larger neural system. It was a primary means for communicating the many constraints of the problems between the neurons in our solutions.

In large multistage switches, determining the existence of a route through a switch and, if possible, determining the route itself is a time consuming problem with conventional computer architectures. We defined the simplest brute force search technique and designed a massively parallel neural algorithm that was able to implement this technique.

In a three-stage rearrangeable switch, previously placed calls may have to be rerouted before a new call can be put up through the switch. Using a known optimal algorithm, we designed a neural network solution that has complexity (in terms of the number of nodes and connections) on the same order as the complexity of the switch (in terms of the number of crosspoints). By assuming nonuniform time constants in the neurons, we showed that the consequent time delays could be used to computational advantage. We discussed a recursive extension to the three-stage rearrangeable switch that utilizes the ability of neurons in one network to program the neurons in another network.

In both solutions, one freedom is that the various “programming” neurons (e.g., the feedforward neurons in the first problem, and the detectors in the second) could easily be implemented with components other than neural networks. For instance, the feedforward neurons could be optical components that are ideal for the interconnect between stages. The detectors could be standard digital logic. In any case, as long as the signals that they introduce are within the limits defined in the neural solutions,

the system will still function correctly. The neural components can be concentrated where they can excel: as highly interconnected feedback elements.

The solutions described here show that neural networks can be applied to practical problems. We emphasize that although each instance of these problems requires a different neural network, the solution is well defined, even for large-size problems where the parallelism of the neural network is most advantageous.

Bibliography

- [1] Beneš, V. E., *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.
- [2] Inose, H., *An Introduction to Digital Integrated Communications Systems*, Univ. Tokyo Press, 1979.
- [3] Schwartz, M., *Telecommunication Networks: Protocols, Modeling, and Analysis*, Addison–Wesley Pub. Co., Reading, MA., 1987.
- [4] Beneš, V. E., On Rearrangeable Three-Stage Connecting Networks, *Bell System Technical Journal*, Vol. 41, 1962, pp. 1481–1492.
- [5] Paull, M. C., Reswitching of Connection Networks, *Bell System Technical Journal*, Vol. 41, 1962, pp. 833–855.

Chapter 4

Banyan Network Controller

The distant goal of “neural networkers” is to understand how to store, retrieve, and process data in neural networks; ultimately to characterize the types of data that need to be stored, to know best how to represent them, and to see how to design such machines that accomplish it with the greatest engineering ease.

—J. S. Judd [1]

4.1 Introduction

This chapter will develop a neural network design, starting from the original problem analysis and formulation, proceeding through the design stage, continuing on to implementation considerations and further applications of the networks developed. Along the way we will develop many interesting results related to switching and queueing theory, and will finish with a feasible design of a 1024×1024 packet switch controller based on neural network principles.

4.2 ATM switching networks

This section gives a general background to the ATM (Asynchronous Transfer Mode) switching environment, and introduces the fundamental problem of blocking and queueing. It also motivates why we need a new approach.

A Broadband Integrated Services Digital Network (B-ISDN) consists of two major parts: optical transmission with a Synchronous Optical Network (SONET) [2], and switching with an ATM [3] switch. Although current transmission rates are already

several gigabits per second, the ATM switching fabric at present can function at rates up to only a few hundred megabits per second. The broadband network research community has therefore spent much time and effort on the design of ATM switch fabrics.

One approach in designing ATM switch fabrics is to use interconnection networks [4] that were originally designed for multiprocessor parallel and distributed processing. There are several different interconnection networks; e.g., Banyan, Baseline, Buffered Memory, Delta, and Shuffle Exchange [4, 5]. Among them, the Banyan network is the most popular to be used as a basic building block in ATM switch fabric designs, although Buffered Memory switches are gaining in popularity due to their simplicity in concept.

In an ATM network, unlike the circuit switching of Chapter 3, bursty and continuous data are segmented into small fixed-length units called ATM *cells*. These cells will be our basic unit that the ATM switching network processes. The data rates are approximately 150Mbps and the cell sizes 53 bytes (424 bits) [3]. This implies that any cell processing must be completed in less than $2.8\mu\text{sec}$. We assume there exist some external interface modules that assign each cell entering the switching network an output destination address.

Within every ATM network, blocking must occur due to what is called *output blocking*. If two or more inputs to the network have cells with the same destination address, their cells will be routed to the same output at the same time and hence collide. This is output blocking. Output blocking cannot be avoided, so blocked cells must be buffered somewhere. The location of the buffers affects the queueing characteristics of the network. Although not optimal, we will assume that cells are buffered at each input since this produces the least complicated network. This will be discussed further in the section on queueing.

A controller is needed to choose which cells to send at an instant and guarantee that all the destination addresses are distinct at that instant. The choice of control algorithm is important. The simplest controller uses first in first out (FIFO) queues, and chooses a non-blocking set of cells from the head of the queue. Unfortunately,

this scheme suffers from a statistical impediment called “head-of-line” blocking [6] that ultimately results in each input transmitting significantly less than one cell per time slot. To achieve higher rates, more sophisticated controllers use *bypassing*. In this scheme, cells with different destination addresses further back in the queues are transmitted instead, if doing this avoids blocking. While this increases the maximum throughput rate over that of the FIFO scheme, many of the proposed bypass control schemes still have the problem that the maximum throughput per input is significantly less than one cell per time slot [7].

The controller is further complicated when the switch can not transmit every possible permutation of the inputs, that is, there are sets of cells with distinct inputs and outputs that can collide due to overlapping paths through the switch. This is *internal blocking*. ATM switches that have no internal blocking (but still the unavoidable output blocking) we denote as *non-blocking*, otherwise they are *blocking*. We note that implicitly most switching networks have a third form of blocking, *input blocking*: only one cell may enter each input per time slot.

Some control schemes treat the internal and output blocking separately. For example, to avoid internal blocking within the Banyan network, a so-called Batcher sorter network is used to preprocess the incoming cells of the ATM switch fabric [8, 7]. In this scheme, the cells are first sorted in such a way so that they arrive to the Banyan in an order that avoids internal blocking. But this sorting scheme suffers from several deficiencies. Foremost among these is that it is even more complex than the Banyan network itself. This is true not only in terms of the number of switching elements, but also, since these elements must make comparisons between the destination addresses to perform their sort, a precise cell bit alignment must be maintained through the sorting network. Furthermore, some output blocking still occurs anyway, and queueing and a controller are still required.

As an alternative to the Batcher network, we combine the processing for both internal and output blocking into a single controller. This controller utilizes the state of all the input queues in a bypass scheme that increase the throughput of the ATM switch as a whole. We define a useful class of switching networks to which this method

applies. We also design a neural network implementation of the controller. By using the massive parallelism of the neural network, we can complete each computation quickly. Before we develop these ideas we digress to discuss a particular ATM switch architecture.

4.3 Banyan Networks

We base this section on the work of K. H. Liu [9, 10]. It is included for completeness, and to set up notation for future sections. The Banyan network considered consists of $n = \log_2 N$ stages composed of 2×2 non-blocking switching elements (see Figure 4.1). The topology of the Banyan network can be generated in a recursive way. An $N \times N$ Banyan network can be viewed as a first stage with $N/2$ switching elements, followed by two $(N/2) \times (N/2)$ Banyan sub-networks. The connection between the first stage and the following blocks is the “Banyan Exchange.” This will be defined later.

The Banyan network has the *routing* property. Rather than a global router, the individual switching elements themselves are able to route cells correctly much as in the old step-by-step telephone switches. The routing strategy for a switching element in stage k is to look at the k th most significant bit in the destination address of a cell and send the cell to the “top” outlet if the bit is 0, and to the “bottom” outlet if the bit is 1. Figure 4.1 shows an 8×8 Banyan network, and an example of the self routing. A cell arrives at input 010 and is destined for 101. In the first-stage switch it is routed to the bottom outlet since the first bit of the destination address is 1. In the second stage it is routed to the top outlet since the second bit is 0, and in the last stage it is routed to the bottom outlet since the last bit of the destination address is 1. Note how the successive address bits automatically direct the cell to the correct output destination, and how the cell would arrive at outlet 101 no matter what inlet it started at.

The topology of the Banyan network can be abstracted and described in terms of functions on the input and output addresses of the cells. We assign a label, (I, D) , to each cell at the input of the network, where $I = i_n i_{n-1} \dots i_1$ is the cell’s input address

Figure 4.1: An 8×8 Banyan switching network showing self routing of cell (010,101).

and $D = d_n d_{n-1} \dots d_1$ is the cell's output (destination) address; both are written in n -bit binary ($n = \log_2 N$). As shown in Figure 4.1, we number the inlets and outlets to each stage 0 to $N - 1$ sequentially in n -bit binary. Define L as the *locator label* n -tuple of a cell as it is transmitted through the network. L indicates the address of the input or output to the switching stage at which the cell is currently located.

The Banyan network can be represented as bitwise operations—one corresponding to each of the stages of switching elements and to each of the sets of links between stages—that transform L from $L = I$ at the input to $L = D$ at the output. This is effected by defining a set of *topology defining rules* [5]. The operation R_k of switching stage k is a binary operation, a function of a cell's address at the input to the stage and its destination address D :

$$R_k(a_n a_{n-1} \dots a_2 a_1) \setminus (d_n d_{n-1} \dots d_2 d_1) = (a_n a_{n-1} \dots a_2 d_{n-k+1}).$$

This “replacement operator” defines the self-routing of the cells. In Figure 4.1 we see how the cell at input 2 (010) of Stage 1 is sent to output 3 (011), since the MSB

of D is 1. The Banyan Exchange connection between the k th and $(k + 1)$ th stage switching elements is:

$$B_k(a_n a_{n-1} \dots a_2 a_1) = (a_n a_{n-1} \dots a_{n-k+2} a_1 a_{n-k} a_{n-k-1} \dots a_2 a_{n-k+1}).$$

This operator simply swaps the least and the k th most significant bits. (We use the usual convention that $a_x a_{x-1} \dots a_y$ is null when $x < y$.)

Continuing with our example, the Banyan exchange routes the output of the first stage at output 3 (011) to the input of the first stage at 6 (110). By concatenating these operators together, the Banyan network is represented as:

$$R_n B_{n-1} R_{n-1} \dots B_2 R_2 B_1 R_1(I) \setminus D.$$

Using these operations, we see that the locator label L for our example cell is transformed through the sequence (010), (011), (110), (110), (101), thus ending at the desired destination (101).

In general, for an n stage Banyan network, we define:

$$L^k(I, D) \triangleq d_n d_{n-1} \dots d_{n-k+2} i_{n-k} i_{n-k-1} \dots i_2 d_{n-k+1}. \quad (4.1)$$

This is the locator label at the outlet of stage k of a cell from inlet I to outlet D . For consistency, we define the input as Stage 0, with $L^0(I, D) = I$.

4.4 Blocking Constraints and Deterministic Switches

In this section, we define a general class of Banyan-like switches that we denote as *deterministic*. For a given switch architecture, let $S = \{(I, D)\}$ be a set of cells such that every cell in this set is mutually blocking, that is, given any two cells in S , these cells collide somewhere in the switch. Such a set we call a *constraint set*. For deterministic switches, these constraints are simple to define, and can be used to completely define blocking. We will later investigate the blocking for two switch architectures from within this class.

A switch is deterministic if:

- A. The switch is composed of non-blocking square switch nodes.
- B. The nodes can be arbitrarily interconnected, but any switch input or output has only a single link to one of these nodes.
- C. Between each input and output there is exactly one route (defined in terms of links and nodes).

A Banyan network is a deterministic switch composed of stages of 2×2 nodes. A non-blocking switch is a deterministic switch with a single node—the switch—that has links to the inputs and outputs. Omega networks, baseline networks, and flip networks are all topologically equivalent to the Banyan network and therefore also deterministic switches [5].

To define the constraint sets, we note that whenever two cells both attempt to use the same link between two switches, there is a collision, and thus blocking. These two cells will always collide, since they have only one choice for routes, and both routes are through this link. Each of the switching elements are non-blocking, as long as only one cell arrives per input and leaves per output. Therefore, none of the cells are blocked if and only if there is no link used by more than one cell.

Let $\mathcal{L}^A = \{l_j\}$, be the set of links in a switch with architecture A . Each link, l_j , in the switch defines a constraint set,

$$S_j = \{(I, D) \mid \text{the route from inlet } I \text{ to outlet } D \text{ uses link } l_j\}.$$

Using these constraint sets, our definition of non-blocking is clear:

Definition: A set of cells, $C = \{(I, D)\}$, is *non-blocking* if and only if each set, S_j , contains at most one $(I, D) \in C$.

For the Banyan switch, $\mathcal{L}^{\text{Banyan}} = \{l_j^k\}, 0 \leq k \leq n, 1 \leq j \leq N$, where l_j^k is the link connected to the j th outlet of the k th stage of switches. In the case of the non-blocking switch, the only links are the input and the output links. In terms of the Banyan definition of l_j^k , $\mathcal{L}^{\text{NB}} = \{l_j^k\}$, where $k \in \{0, n\}$, and $0 \leq j \leq N$.

From this labeling, we can define the constraint sets using (4.1):

$$S_k^j \triangleq \{(I, D) | L^k(I, D) = j\}.$$

These sets have an interesting interpretation. For each k , there are N different S_j^k , since there are N different links between each stage. At any stage, each cell uses only one link, so $S_j^k \cap S_i^k = \emptyset$ for all $i \neq j$. Since at any stage a cell must use some link, $\bigcup_{j=0}^{N-1} S_j^k$ is the set of all possible input-output pairs. Thus we have shown that each $L^k, 0 \leq k \leq n$, partitions the N^2 input-output pairs into N equivalence classes.

A more graphical interpretation can be gleaned from the following construction. Given a set of cells $C = \{(I, D)\}$, we create a binary $N \times N$ *cell matrix* by placing a 1 at entry $(I, D), \forall (I, D) \in C$. Using this matrix, Figure 4.2 shows the $n + 1 = 4$ different sets of equivalence classes for $N = 8$.

Let $\mathcal{C}^A = \{C : |C| = N \text{ and } C \text{ is non-blocking in switch architecture } A\}$. This is equivalent to the set of permutations realizable by the switch. From [9, 10], we find the number of these non-blocking sets for the Banyan (here as before $N = 2^n$):

$$|\mathcal{C}^{\text{Banyan}}| = 2^{\frac{nN}{2}} = N^{N/2}. \quad (4.2)$$

To compare the Banyan network to a non-blocking network, we note that the Banyan network has $n + 1$ levels of blocking, one for each set of links between stages. The non-blocking network has only two levels of blocking; input and output. Since the non-blocking network can transmit any permutation, $\mathcal{C}^{\text{NB}} = N!$. To get an idea of the size of $\mathcal{C}^{\text{Banyan}}$ relative to the set of all permutation matrices, we use Sterling's formula to show $\mathcal{C}^{\text{NB}} = N! \approx \sqrt{2\pi} 2^{nN-1.44N+n/2} = \sqrt{2\pi N} / 2^{1.44N} (\mathcal{C}^{\text{Banyan}})^2$.

Before investigating the implications of these differences further, we discuss some background on queueing architectures for switching networks.

4.5 Queueing

This section will introduce the queueing model for our system. We will investigate other queue control methods that have been proposed for this system, and then

Figure 4.2: The equivalence classes induced on the 8×8 input matrix by each stage of links in Figure 1. The numbers in the matrix indicate which link a cell uses at stage k .

Figure 4.3: The queueing model.

introduce our method, applicable to the class of deterministic switches. We show that the method has a throughput per inlet arbitrarily close to one cell per time slot.

As discussed in the introduction to this chapter, even if a preprocessor removes all internal blocking (if any exists) leaving only output blocking, cells must be queued to prevent losses. We assume a model as shown in Figure 4.3. The cells arrive at an input according to some random process, and each cell is destined for one of the outputs according to a second process. Cells that are blocked and can not be sent are queued at each input. We assume that the inputs have independent and identical arrival processes.

For the purposes of this chapter we consider two arrival processes. The simplest is a Bernoulli process, where one cell arrives at the beginning of each time slot with probability α . The second arrival process that we consider is a batch process where cells arrive in batches, every cell within a batch destined for the same output [11]. The number of cells within each batch is geometrically distributed, with mean batch size θ , while the number of batches that arrive each time slot is Poisson distributed, with mean number of arrivals α/θ . In either case, the destination addresses of the cells are uniformly distributed, with probability $1/N$ that a given output is chosen. We can view these two cases as extremes in the space of potential arrival traffic processes. For a given average number of arrivals, α , the Bernoulli case will spread the arriving traffic most evenly across the outputs and across time. The batch arrivals on the

other hand will tend to bring sharp concentrations of traffic both across the outputs and time.

Reference [11] analyzes a single deterministic server (a 1-input switch in our terminology). That analysis showed the sharp concentrations in time. As an example, for an average burst length of $\theta = 10$ and average arrival rate of $\alpha = 0.8$ cells per time slot, to have a buffer overflow probability of less than 10^{-6} requires a buffer length of about 600. For the Bernoulli case, this length is only 1 (since the buffer can always send one, and at most one arrives). But as we will show, even for a 32-input Banyan with the same average number of arrivals, the necessary buffer per input is only about 30. Note that in the approach we take, the entire batch enters the queue at once, unlike other simulations [12] that assume (with reason) an additional external buffer that governs the cell arrivals so that at most one cell is transferred per time slot from the external buffer to the system. By only counting the cells in the system, the external buffer substantially mitigates the effects of the batch concentrations.

For the case of a non-blocking switch, various queueing strategies are analyzed in [6] and [13]. One strategy discussed is output queueing. In this strategy, all cells are allowed to pass through the network, and multiple arrivals to an output are then queued. While this was found to be the best strategy statistically, it requires the switching network to operate at N times the transmit rate. This is undesirable in a network that is already operating at very high bit rates. Alternatively, the switching network can have multiple copies [12], but this adds undesirable complexity.

For queueing at the inputs, a strict FIFO and a so called *consecutive competition* scheme were also analyzed in [6]. The consecutive competition strategy starts with the first cell in each queue and determines which inputs will transmit, and which inputs are blocked. Then it repeats the procedure with the second cells among the blocked queues that compete for the remaining unused outputs. This procedure can continue up to a depth w into the queue. FIFO is merely the special case of $w = 1$. For finite w it was found that the length of the queues became unstable even for arrival rates, α , less than one. For instance, when $w = 1$, the maximum throughput per inlet is less than $(2 - \sqrt{2}) = 0.586$ cells per time slot as $N \rightarrow \infty$. While letting

Figure 4.4: The Mapping of the State of the Queues (a) to a Cell Matrix (b). A 1 is placed in (i, j) of the matrix if a cell is waiting at input queue i to be sent to output address j .

$w \rightarrow \infty$ could allow the maximum throughput to approach 1, this algorithm requires $O(Nw)$ steps to execute, and is therefore unsatisfactory for large N and w .

Here we propose a new approach applicable to the deterministic switches of the last section. Given a set C of cells queueing at the inputs of the switch, we map them to an $N \times N$ cell matrix as described in the previous section, with 1's in entry (I, D) if and only if there exists a cell $(I, D) \in C$ as shown in Figure 4.4. The controller then uses the blocking constraints on the subsets S_j to choose a subset C' that is non-blocking. Exactly how we use these constraints to choose a C' is the subject of the next section. Optimally the network controller will choose a C' that has the maximal overlap with C . This approach has a maximum throughput of 1, even if the controller only randomly chooses a $C' \in \mathcal{C}^A$, and sends $C' \cap C$. The only assumption that we make is that \mathcal{C}^A is *symmetric*, that is $|\{C' | (I, D) \in C'\} \cap \mathcal{C}^A| = |\{C' | (I, D') \in C'\} \cap \mathcal{C}^A|$ for all I, D , and D' . By the symmetry of the switch designs, both $\mathcal{C}^{\text{Banyan}}$ and \mathcal{C}^{NB} are both symmetric.

We define the idea of the *random queue manager* (RQM). Given a set C of cells waiting in the queue, a RQM randomly chooses a set $C' \in \mathcal{C}^A$ uniformly and indepen-

dent of C , and sends $C \cap C'$. We also define a controller to be *stable* if the expected size of the queues are finite. When the queues are finite, then the throughput equals the average number of cell arrivals. With these facts in mind we show:

Theorem 4.1 *Given a switch with architecture A and symmetric \mathcal{C}^A , a random queue manager is stable for any average arrival rate, α , that satisfies $0 \leq \alpha < 1$.*

Each input queue, I , can be decomposed into N separate queues, one for the cells waiting to be sent to each output. Focusing on the cells waiting for output D at input I , we show that this subqueue is stable under the RQM and thus so is the queue as a whole.

Because the destination addresses are chosen uniformly, an (I, D) arrives every time slot with probability α/N . The RQM chooses C' uniformly from \mathcal{C}^A . This and the symmetry of \mathcal{C}^A implies that $(I, D) \in C'$ with probability $1/N$. Given this probability of being sent, a cell (I, D) that arrives at the head of the subqueue has an expected service time of N time slots. The utilization factor in this case is

$$\begin{aligned} \rho &\triangleq (\text{average cell arrival rate}) \times (\text{average service time}) \\ &= (\alpha/N) \times N, \end{aligned}$$

which is stable for any $0 \leq \rho < 1$ [14, p. 19]. ■

Since the RQM is stable for any $\alpha < 1$, then any more sophisticated scheme that takes into account C is also stable with such α .

4.6 The Neural Network Solution

This section designs a neural network that chooses a set of non-blocking set of cells from the queue. The complexity of the design is calculated, and compared with another possible neural approach.

For our problem, we use N^2 neurons arranged in an $N \times N$ matrix that corresponds to the cell matrix of Figure 4.4. This is our neuron set Θ . The input to the problem is a particular cell matrix, that is, neuron (I, D) receives a +1 input if a 1 is in entry

(I, D) of the cell matrix, -1 otherwise. If at the end of the computation, neuron (I, D) is on, then send the cell (I, D) . If more than one cell has the same (I, D) , then send the first (I, D) that arrived in the queue.

The neural network design follows directly from the constraint sets, S_i , defined in Section 4.4. Using Θ and $\{S_i\}$, we design a Multiple Overlapping Winner-Take-All according to (2.8). By definition, this has the property that no more than one neuron will be on per constraint set. Therefore, the set of neurons that are on at the end of the computation will always correspond to a non-blocking set of cells.

To measure the complexity of this circuit, we count the number of interconnections. Using the construction method of Section 2.7, the number of interconnections in the network is $O(\sum_i |S_i|)$ connections. In the case of the Banyan network, there are $O(N)$ connections for each equivalence class, S_j^k . There are $(n+1)N$ of these classes, so the number of connections in the network is just $O(N^2 \log_2 N)$. For the non-blocking switch, there are $2N$ equivalence classes. The number of classes in this case is just $O(N^2)$. We can assume that the minimum circuit in either case would have at least N^2 connections, one for each neuron in the circuit. Thus, we have a circuit that is defined for any N , provably works, and is within a factor of $\log_2 N$ of the smallest possible network. We contrast this with the resulting networks from the energy function approach to optimization [15]. Using that approach on the problem of choosing a non-blocking set of cells for a non-blocking switch [16], it is seen that the resulting network has $O(N^3)$ connections instead of $O(N^2)$, a factor of N more connections, and further, the strength of the connections must be experimentally determined for every N .

4.7 Network Prompting

The neural network gives us some flexibility in processing the cells in the network, which allows us to tailor the network to specific applications. More importantly, it allows us to modify the network to improve the choice of non-blocking sets and improve upon the statistics.

The definition of the network allows for inputs, I_i , other than ± 1 . In fact, if we set the common diagonal elements, $w_{ii} = a$, to zero, the network still works with positive inputs in the range $0 < I_i < 2$. If two neurons, i and j , with the same capacitance, C , have approximately the same inputs from other neurons and $u_i(t) \approx u_j(t)$ at some time t , then:

$$C \frac{d(u_i - u_j)}{dt} = \eta + (I_i - I_j), \quad (4.3)$$

where η is the sum of all the differences between the neurons. As long as $|I_i - I_j| > |\eta|$, we see from (4.3) that the neuron with the higher input will evolve to a higher value. For two such neurons in the same constraint set, this implies that the neuron with the greater external input will turn on. If a neuron i is in a position where other neurons are trying to turn on in each of its constraint sets, and neuron j has only one neuron (i.e., neuron i) trying to turn on out of all its constraint sets, then in this case η will not be small, but will be large and negative, favoring neuron j . This shows that the neuron with the largest input will turn on, unless doing so will block an inordinate number of other neurons from turning on. All neurons that are receiving a positive input are given a basic input I^0 . Under various conditions, a prompting signal π can be added such that $0 < I^0 + \pi < 2$. How can we use this neural network capability? We discuss two ideas, *priority* and *smoothing*.

Cells may have different priority levels. For simplicity we consider just two levels, *high* and *low*. Low priority is considered the normal case, with an input of I^0 . Neurons corresponding to cells with high priority, however, receive the prompt π_H . From the previous discussion, we see that this results in a soft priority, i.e., high priority cells will be sent first as long as they don't impair throughput significantly by preempting a large number of lower priority cells. This should prevent the deadlock behavior for low priority cells described in [6]. Further study is necessary to demonstrate this.

A queue containing one cell for each destination will, depending on the architecture, always send a cell. If the queue had N cells for only one destination, the queue would be extremely limited. Snapshots of the queue state from the simulations show that in fact the latter case is the more prevalent. A simple way to decrease the average

queue size is to give a prompting signal π_M to neurons corresponding to an (I, D) that has many cells waiting. This will smooth the number of cells waiting for each output destination in a queue, and can significantly reduce the expected queue size.

These are just two possibilities of prompting; further applications include flow control, giving cells that have been waiting in the system a long time a prompt to send them along sooner, and balancing for component irregularities or degradation in the system, for example, one neuron that is slower than the others receiving a prompt to compensate.

4.8 Simulation Results

To study the behavior of the queueing system under the neural network controller, we ran a series of simulations on a MIPS M/120 computer. We outline the simulation method and present graphs of the Banyan and non-blocking switch performance. These are used as a basis for comparing with other methods of choosing a non-blocking set of cells, and with other queue control methods. Finally we compare the Banyan and the non-blocking switches to each other to show that with a powerful controller such as the neural network, these switches have similar performance, i.e., the Banyan is almost as good as the non-blocking switches.

For a given switch architecture and size, N , the simulation set up N input queues, N^2 neurons, and defined all the constraint sets and resulting weights; $w_{ii} = 0$ was used. The simulation consisted of a sequence of time slots. Each time slot comprised the following steps:

1. Generate a set of arrivals for each input queue according to the chosen arrival process. Add the arrivals to the queues.
2. Create a cell matrix based on the current state of the queues.
3. Using the cell matrix, set up the inputs to all of the neurons.
4. Let the neural network settle on a solution.

5. Determine the neurons which are on, and remove the corresponding cells from the queue.

Statistics were recorded on the states of the queues after Step 1, as well as the time in the system of cells sent in the last step. Consistency checks were also made to guarantee that the constraints were always met (after debugging, they always were).

The simulation of the neurons deserves further discussion. Step 4 consisted of numerically integrating the differential equation in Equation (2.1). Several methods were tried for this; Euler's method, fourth order Runge-Kutta method with adaptive step size [17, Sec. 15.2]; and Richardson extrapolation with the Burlirsch-Stoer method [17, Sec. 15.4]. Euler's method suffered from always taking too big or too small steps. The Burlirsch-Stoer method suffered from trying to extrapolate too much information from the highly non-linear neuron functions. The Runge-Kutta method was the most satisfactory. At the start of the simulation, the capacitances were assigned the value e^x , where x was chosen uniformly from $(-0.05, 0.05)$. At the start of Step 4 of each time slot, the neuron states were initially set to a small number near zero chosen uniformly from $(-\frac{0.05}{\lambda}, \frac{0.05}{\lambda})$. The exact values here were not important; they were chosen only to add a small amount of noise to the system to prevent instabilities in the simulation.

Since the inputs, I_i , were chosen to be ± 1 , by (2.11), a neuron which turns on ($\kappa_i = 0$) will have a net input of 1. This implies that the internal state approaches $\frac{1}{\lambda_i}$. For the purposes of timing, the network was considered to have settled when, after an initial period, all of the neuron states were more than $\frac{1}{2\lambda_i}$ away from zero. For each simulation run, we recorded the maximum, minimum, and average settle times.

In calculating the derivative, we took advantage of the implementation techniques of Section 2.7. By summing the outputs of all of the neurons in each constraint set once and then for each neuron adding the resulting sums from the constraint sets to which it belongs, the complexity was reduced by a factor of $|S_j^k| = N$. The sigmoid function used was the hyperbolic tangent function. The gain was always chosen so

Figure 4.5: Demonstration of smoothing for simulations of Bernoulli traffic into an 8-input Banyan network with various controllers.

that it was larger than the value in Table 2.1 with $\epsilon = 0.5$.

In the previous section, we discussed the benefits of smoothing the input. Figure 4.5 shows such a comparison for an 8×8 Banyan network, using $I^0 = 1$ and $\pi_M = 0.3$ when two or more cells of the same (I, D) are waiting. The 8×8 size was chosen so that we could compare it to an *optimal controller*, i.e. one that exhaustively searches all of $\mathcal{C}^{\text{Banyan}}$ and sends the largest non-blocking set. Figure 4.5 shows that the prompting brings the neural network closer to the optimal controller in performance. Because of this benefit, we will assume that all neural controllers use this smoothing. The choice of $\pi_M = 0.3$ followed from testing of various values between 0.0 and 0.5 in increments of 0.1. Within this range 0.3 was the best, slightly better than 0.2 and 0.4 which were significantly better than values larger and smaller.

Using the computer simulator, we tested the neural network on Banyan and non-

blocking networks with 2, 4, 8, 16, and 32 inputs. A simulation consisted of 10,000 time steps of cell arrivals at a chosen arrival rate α . The average input queue sizes under Bernoulli arrivals are plotted as a function of α in Figure 4.6. These results show that high throughput rates can be maintained with only modest queue sizes, even for large switching networks.

Figure 4.7 show the same graphs, but now with batch arrivals (the extra plots on the figure will be discussed later in this section). The average batch size was $\theta = 10$. As expected, because of the more sporadic arrivals, the average queue sizes are much larger. The simulations were not as extensive in this case for two reasons. The first is that even though we simulated for over 10,000 time steps, the data contained a large variance, and was not as informative. This was seen by observing subsamples of the data. Increasing to 100,000 time steps only decreased the variance marginally. The second reason was that except for at the larger α , the plots were very close, often crossing each other several times. Since this is a work-conserving system the average wait time, \overline{W} (not graphed), of a cell can be calculated in terms of the average queue length, \overline{L} , using Little's formula: $\overline{W} = \overline{L}/\alpha$ [14, p. 17].

We compare the neural networks method with two other methods of choosing a non-blocking set of cells to send. The simplest method to compare with is a greedy algorithm. In this algorithm, we randomly choose one cell to send from each queue, throwing out any blocked cells one by one, until the remaining set of cells is non-blocking. The queueing behavior for this method we consider an upper bound. As a lower bound, we might consider a simple exhaustive search of all $C' \in \mathcal{C}^A$ to find the set with the largest $|C' \cap C|$, but this becomes computationally prohibitive for switches with more than 16 inputs, where even for the Banyan the possible number of non-blocking sets is $2^{\frac{Nn}{2}} \approx 4.3 \times 10^9$. In the case of the non-blocking switch, the problem of finding a maximal non-blocking set of cells to send reduces to the cardinality graph matching problem. This latter problem has a known polynomial time algorithm that solves it exactly [18].

Given a bipartite graph [19], a set M of edges is a *matching* if no two edges in M are incident to the same node. The cardinality matching problem is to find

Figure 4.6: Average queue size vs. average number arrivals for simulations of Bernoulli traffic into Banyan (a) and non-blocking (b) switches with 2 to 32 inputs.

Figure 4.7: Average queue size vs. average number arrivals for simulations of batch arrivals into various networks.

the matching with the largest number of edges. The reduction of the maximal non-blocking set problem to cardinality matching is simple and direct. Given an $N \times N$ cell matrix, C , construct a bipartite graph $B = (X, Y, E)$, where $X = \{x_1, x_2, \dots, x_N\}$ and $Y = \{y_1, y_2, \dots, y_N\}$ are two sets of vertices, and $E = \{(x_i, y_j)\}$ is the set of edges. If (I, D) is 1 in the cell matrix, then $(x_I, y_D) \in E$, and vice versa. If $(x_I, y_D) \in M$ then we send cell (I, D) . For a matching $M \subset E$ on B , $(x_I, y_D) \in M$ implies this is the only edge in the matching incident on node x_I and on node y_D . The constraints placed on a non-blocking set of cells for a non-blocking switch require no more than one cell per input and one cell per output. So the edges in a matching M correspond to the cells in a non-blocking subset of C .

Figure 4.8 shows graphs of the average queue size for the greedy, neural network, and optimal method on a 32-input non-blocking switches. The results show, we think

Figure 4.8: Performance range for simulations of Bernoulli traffic into a 32-input non-blocking switch.

as expected, that the neural network produces results intermediate between the two extremes. The range of values is not great, suggesting that we might consider the simple greedy algorithm. But even this has $O(N^2)$ time complexity. Assuming that the algorithm requires just N^2 machine cycles to find a non-blocking set on some serial machine, in order to complete this operation in less than the $2.8\mu\text{sec}$ between cell transmission times would require a clock speed of over 350MHz for a 32-input switch. This rules out this algorithm, and the even longer matching algorithm (time complexity $O(N^{5/2})$), for all but the smallest switches.

We now compare the neural network with alternative controlling schemes using a 32-input non-blocking switch. The first scheme is the sequential competition scheme described in Section 4.5 ($w = \infty$), while the second is an output queueing scheme where all cells are allowed to pass through the switch to the outputs [6, Equation

Figure 4.9: A comparison of different control architectures on 32-input switches with Bernoulli arrivals.

(21)]. These are all plotted on the same graph in Figure 4.9 (along with a plot for the neural network on the Banyan switch). Of the three, the sequential competition scheme, even with this optimal window size, is the worst. The output queueing is the best, as little as half the queue of the neural controller, but at the expense of a more complicated switch. From our discussion in the introduction, the cost of gaining this at best factor of two gain in performance is an N times more complicated switch. For large N this can be an expensive trade-off.

An interesting implication follows when we directly compare the performance of the Banyan network to that of a non-blocking network such as the Batcher Banyan combination. As discussed already, both will have queueing due to output blocking, but the Banyan network has additional internal blocking. For the Banyan we use the neural network with prompting as just described. For the Batcher Banyan we

use the sequential competition scheme and a neural network controller. We compare them using switches with 32 inputs. Results for these three are shown in Figure 4.9. This graph shows that the non-blocking network out-performs the Banyan network by a factor of two to three. This was for the case of Bernoulli arrivals. In the case of Batch arrivals, Figure 4.7 plots both the Banyan with neural network controller and the non-blocking switch with optimal matching controller. In this case we see that again the difference in performance is a factor of two to three.

From an engineering standpoint, a more relevant parameter than the average queue size is the length of the buffer, B , such that the probability of losing a cell due to a saturated buffer is less than a given level, P_L . Appendix 4.A derives a method for estimating an upper bound on B as a function of P_L using data on the distribution of the queue size recorded during the simulations. Using this method, we construct Table 4.1 of upper bounds on B . With this table as a measure, we see that the Banyan network is still within a factor of two to three of the non-blocking switch. Note, however, that the necessary buffer size in the case of batch arrivals is significantly larger, by a factor of 100 or more.

Although the non-blocking switches outperform the Banyan network, this better performance should be viewed in light of the extra complexity of a non-blocking switch. In many cases it would be simpler to use the longer queues with a Banyan network than to add the additional hardware necessary to make the network non-blocking.

4.9 Implementation Considerations

We show that each input queue can interact with the neural network in an independent and efficient manner. We also show that the constraint sets in the case of the Banyan network possess a regularity that can simplify the construction of the controller. Figure 4.10 shows a schematic of the architecture for a single input queue. It consists of the queue itself; the controller logic; the neural network; and a small amount of memory to hold a pointer to the first instance and the number of each type of cell.

Table 4.1: The Length of Buffer B Sufficient to Reduce Cell Loss Below P_L for Various Combinations of Offered Traffic, α , and Switch-Controller Pairs.

(a) Bernoulli traffic.

Switch Type:		Banyan			Non-Blocking					
Controller:		Neural Net			Matching			Neural Net		
	P_L	$\alpha = 0.50$	0.75	0.90	0.50	0.75	0.90	0.50	0.75	0.90
8 Input Switch	10^{-3}	5	11	30	4	7	16	4	8	21
	10^{-6}	8	20	57	6	12	26	7	14	32
	10^{-9}	12	29	84	9	17	37	10	19	44
32 Input Switch	10^{-3}	7	18	41	3	6	15	4	9	32
	10^{-6}	12	28	56	6	11	26	7	17	53
	10^{-9}	17	37	70	8	15	36	9	22	74

(b) Batch arrival traffic.

Switch Type:		Banyan			Non-Blocking		
Controller:		Neural Net			Matching		
	P_L	$\alpha = 0.50$	0.75	0.90	0.50	0.75	0.90
8 Input Switch	10^{-3}	210	280	1200	140	230	740
	10^{-6}	490	470	1500	280	390	1700
	10^{-9}	770	670	1700	440	540	2600
32 Input Switch	10^{-3}	160	560	870	130	260	680
	10^{-6}	300	1100	1200	270	510	1300
	10^{-9}	440	1700	1600	400	750	1900

Figure 4.10: A schematic of the queue architecture for one of the input queues.

Since the memory keeps track of the first instance of each type of cell, all cells sent to the same destination remain in arrival sequence. The following table lists the processing algorithm at time step τ .

Initialization	Queue Control reads the destination, D_{New} , of newly arriving cell. If $\#D_{New} = 0$ then: Set Pointer for D_{New} Toggle Latch D_{New} to +. Increment $\#D_{New}$	
Concurrent Processing	Release neural network. Read destination address D_τ .	If $D_{\tau-1}$ was sent last time, update pointer to next $D_{\tau-1}$.
Post Processing	Send cell at pointer for D_τ . Decrement $\#D_\tau$. If $\#D_\tau = 0$, change latch D_τ to -.	

In every time slot, at most one cell can leave, and cells for one destination can arrive. Since the latches retain their state from one time slot to the next, at most two latches are modified in any time step. For maximum throughput, the three steps in

this table must be completed in less than one time slot, 2.8 microseconds. We assume that the capacitance, C , is proportional to the number of connections per node, and the base capacitance is 2pF. The resistance of a connection we assume is $100\text{K}\Omega$. This results in a worst case settle time over all simulations of 250 nsec. This allows sufficient time for the other processing requirements. The limiting step in this process is the updating of the memory about $D_{\tau-1}$. This requires a search of the queue to find the next cell destined for $D_{\tau-1}$, which scales with the size of the queue but not with the number of inputs, N . This search-time limit can be removed if necessary by using a larger pointer memory that stores the location of each successive cell for each destination.

As a final point, we describe an efficient layout of the neurons for the Banyan controller. The equivalence classes in Figure 4.2 all occur in contiguous blocks of neurons. Note, though, that the numbering of the rows (inlets) is not in sequence. We claim that we can always reorder the rows or columns so that the equivalence classes are in blocks. Formalizing this notion of blocks, for an $N \times N$ matrix of neurons, (ν_{rc}) , a set S of neurons is said to be a *block* if there exist integers $r \leq r'$ and $c \leq c'$ such that $\nu_{lm} \in S$ if and only if $r \leq l \leq r'$, and $c \leq m \leq c'$.

Theorem 4.2 *For an $N = 2^n$ -input Banyan network, we can always find a single permutation of the labels on the rows and columns so that all of the constraint sets S_j^k are blocks.*

Proof: Recall Section 4.3. For a particular Banyan network configuration, the switches in stage k replace i_{x_k} by d_{y_k} . For a column labeled $D = d_n d_{n-1} \dots d_1$, rearrange the order of the bits and label the column instead $d_{y_1} d_{y_2} \dots d_{y_n}$. Similarly, relabel the rows from $I = i_n i_{n-1} \dots i_1$ to $i_{x_n} i_{x_{n-1}} \dots i_{x_1}$. We show that this is the desired permutation of the rows and columns.

After stage k , the locator label, L , of a cell is composed of the bits $i_{x_n} i_{x_{n-1}} \dots i_{x_{k+1}}$ from the input address and the bits $d_{y_1} d_{y_2} \dots d_{y_k}$ from the destination address. For simplicity we will assume that $L = i_{x_n} i_{x_{n-1}} \dots i_{x_{k+1}} d_{y_1} d_{y_2} \dots d_{y_k}$. In this case,

$$S_j^k = \{(I, D) | L = i_{x_n} i_{x_{n-1}} \dots i_{x_{k+1}} d_{y_1} d_{y_2} \dots d_{y_k} = j\}.$$

This set includes all neurons, ν_{ID} , such that

$$\begin{aligned} i_{x_n} i_{x_{n-1}} \dots i_{x_{k+1}} 00 \dots 0 &\leq I \leq i_{x_n} i_{x_{n-1}} \dots i_{x_{k+1}} 11 \dots 1, \\ d_{y_1} d_{y_2} \dots d_{y_k} 00 \dots 0 &\leq D \leq d_{y_1} d_{y_2} \dots d_{y_k} 11 \dots 1. \end{aligned}$$

But, for the permutations we defined, these correspond to contiguous rows and columns and by definition this is a block.■

The proof is general enough to include all switching networks that are topologically equivalent to the Banyan, such as the baseline network. In our case $y_k = n - k + 1$, implying that the columns are unpermuted. For the rows, $x_1 = 1$ with $x_k = n - k + 2$ for $k > 1$. For $N = 8$, this implies that the most significant two bits are simply permuted. The fact that the constraint sets for the Banyan network controller are blocks can greatly simplify the layout.

4.10 Buffered Memory Switches and Large Switch Designs

In this section, we discuss buffered memory switches and show how we can use them to design a large ATM switch. A buffered memory switch is simply a high speed memory device that reads in all the arriving cells and reads out previously stored cells to the appropriate output lines. The memory can operate at ATM rates (≈ 150 Mbs) by first converting the cells from serial to parallel and then sending the cells one by one into the memory, gaining a data slow-down factor of

$$\frac{\text{cell size}}{\text{number of input lines}} = \frac{424}{n},$$

where 424 is the number of bits in a cell. So far the largest buffered memory switch proposed is $n = 32$ [20]. This results in a manageable internal data rate of 12Mbs.

The first advantage of these switches is that in effect they are utilizing output queueing with the improvement in queueing statistics shown in Figure 4.9. The second advantage is that, due to the trunking effect of the shared memory, they have a significant reduction in the buffer size per port for a given accepted level of cell loss. For $n = 32$, this is a reduction by a factor of 7 as shown in [20]. For larger

Figure 4.11: The buffered-memory Banyan hybrid switch.

n , this reduction factor is even larger. Unfortunately, due to the current state of technology, the possibility of going to larger n is restricted by memory speeds. One way to increase the size of the switch is to build larger switches from stages of the buffered memory switches as described in Chapter 2. The problem is that the buffered memory switch is an expensive component.

We propose a hybrid buffered-memory Banyan switch that would have the advantage of reduced buffer requirements and reduced blocking over the original Banyan, and allow for larger switch sizes than can easily be achieved by a single buffered memory switch. Figure 4.11 shows the architecture. It consists of two stages of switches, a first stage of buffered memory switches and a second stage of Banyan switches.

We show that this hybrid switch can be controlled. The buffered memory switch acts simultaneously as a queue and an initial switch stage. The control structure for the buffered memory switch is modified slightly. Each output of the buffered memory switch is directed to a different Banyan in the next layer. Instead of maintaining a queue for each of its outputs, a buffered memory switch must maintain a queue for

each of the outputs of the larger switch, that is, a queue for each of its outputs divided into subqueues, one for each output of the Banyan that it is connected to.

The Banyans use one neural network controller for each Banyan. The cells waiting at an output of a buffered memory switch are equivalent to the queue in Figure 4.10. Thus between each buffered memory switch and Banyan we have two lines: one line that transmits the cells and one to send messages about two things, changes in the state of the queue (buffered memory to Banyan) and second which cell to send (Banyan to buffered memory). Thus, instead of a single large and complex controller for the whole switch, we produce a distributed controller with elements that corresponds to each of the switches in each stage.

A 32-input Banyan and its neural controller can each be made on a single VLSI chip [21]. The projected 32-input buffered memory switch in [20] requires about 15 VLSI chips and one printed circuit board. Thus a feasible switch design would consist of a layer of thirty-two 32-input buffered memory switches, followed by a layer of thirty-two 32-input Banyans, resulting in a 1024-input ATM switch. The total hardware requirements would be approximately 550 VLSI chips and 40 circuit boards. Recall that this is a system processing information at a rate of 150 Gbs (=150Mbs \times 1024 lines) and it is equivalent in capacity to a small exchange office, so the complexity seems acceptable.

To evaluate the queueing behavior, we can use our previous simulation data. Each queue at the output corresponds to one of the input queues of a Banyan. The only difference is that now, instead of the independent arrival processes into each queue of Figure 4.3, each queue receives $1/N$ of the arrivals from each of the N inputs. For the Poisson arrival process of the batches, these two models are equivalent. For the Bernoulli arrivals we will treat the data from the model in Figure 4.3 as a lower bound on the performance expected for the actual hybrid switch.

The data for average queue size per buffered memory input port can be obtained from Figures 4.6 and 4.7. To find the size of the total buffer necessary for a given level of loss, we recall that the memory is shared by all of the queues. A given buffered memory switch has each output directed to a different Banyan switch. Assuming that

Table 4.2: Total buffer size necessary for a given P_L in a 32-input buffered memory switch when connected to a second stage of either 8 or 32-input Banyans (also buffer size per input port).

Traffic Type:		Bernoulli			batch arrivals		
	P_L	$\alpha = 0.50$	0.75	0.90	0.50	0.75	0.90
8 Input	10^{-3}	51(1.6)	140(4.4)	370(12)	1300(40)	2900 (91)	12000(360)
Banyan	10^{-6}	64(2.0)	170(5.2)	430(13)	1800(56)	3700(120)	16000(480)
Switch	10^{-9}	78(2.4)	190(6.1)	480(15)	2300(71)	4500(140)	19000(610)
32 Input	10^{-3}	74(2.3)	290(9.1)	830(26)	1300(40)	5000(160)	12000(370)
Banyan	10^{-6}	93(2.9)	340(11)	900(28)	1700(54)	6300(200)	14000(450)
Switch	10^{-9}	110(3.5)	380(12)	960(30)	2200(68)	7600(240)	17000(530)

the inputs to the large switch are uncorrelated, each of the buffers for cells waiting for an output are independent of the other buffers within the same switch. Therefore, if $g_r(s)$ is the generating function of the queue size distribution for an r -input Banyan, then $(g_r(s))^n$ is the distribution of the buffer size within one of the buffered memory switches when the second stage is composed of r -input Banyans. Using an empirical $g_r(s)$ recorded from the simulations we determine the buffer sizes necessary for various traffic and present the results in Table 4.2. These results show a significant reduction in the buffer size per input, compared with the results in Table 4.1.

4.11 Conclusions

ATM switches require a high-speed controller to prevent cell collisions. We analyzed this problem, and showed how it could readily be solved using a neural network with our previous results from Chapter 2. We showed how we could actually incorporate the neural design into the high-speed ATM switching network; the massively parallel neural network provided the required speed to solve the problem. Although we analyzed in detail the performance of the design on non-blocking and Banyan switches, the solution method is applicable to a whole class of switches that we denoted as *deterministic switches*. We showed that the controller results in a throughput arbi-

trarily close to 1. In the case of Bernoulli arrivals, the queue sizes were modest. In the case of burst traffic, the queue sizes were much larger, yet still reasonable.

Using the power of the neural network, an instructive comparison was made between the Banyan and non-blocking switch. This showed that the Banyan is within a factor of 2 of the non-blocking switch in performance. As a demonstration of these results, we presented a hybrid buffered-memory Banyan switch design, with which we were able to show the feasibility and performance of a large (1024-input) ATM switch.

Finally, we conclude by noting that although we concentrated on the stringent ATM switching environment, the results that we presented here certainly apply to modern packet switches in general.

4.A Appendix: Estimating the Tails

In this section we derive a method for producing a (non-rigorous) upper bound estimate on the tail of the distribution, and then derive an expression for using the parameters of the estimate to calculate an upper bound on the size of buffer B needed for a given cell loss rate P_L .

One approach would be to assume a discrete-time Markov chain [14, p. 21]. But this assumption is not true. A queue with 10 cells in it may have 10 cells all destined for the same output, or 10 cells each destined for a different output. Clearly the probability that a cell is sent is a function of the internal distribution of the cells among the subqueues. We more simply assume instead that the the probability that i cells are in the queue, p_i , can be described parametrically by a geometric distribution:

$$p_i(y, z) = ye^{iz}. \quad (4.4)$$

For a true geometric distribution, $y = 1 - e^{-z}$ so that the probabilities for $i \geq 1$ add to 1. But we will actually assume (4.4) only for the tails, so there really are two parameters. The simulations recorded the distribution of queue size such as in Figure 4.12. This plot is on a logarithmic scale. We see that the second derivative of

Figure 4.12: A sample of a queue size distribution.

this curve is negative. This is expected, since the probability that some cell is sent on average increases with the number of cells in the queue. It is apparent that by assuming a log-linear form, and fitting this form to these curves, we will certainly, as $i \rightarrow \infty$, achieve an upper bound on the probability distribution.

We want to use the data from our simulations to estimate y and z . The data from our simulations is just histogram data, i.e., $\{m_i\}$, where m_i is the number of times, out of M total possible times, that the queues had i cells waiting. To estimate the tail, we use the vector $\mathbf{m} = (m_{n_0}, m_{n_0+1}, \dots)$, that is, the histogram data recorded for all i greater than or equal to some integer n_0 . To get the estimates of y and z , we use a maximum likelihood method.

First, we introduce some notation. Let $\sigma_u = \sum_{i=u}^{\infty} m_i$. We let $F_u = \sum_{i=u}^{\infty} p_i = \frac{ye^{uz}}{1-e^z}$. We assume, reasonably, that the possible \mathbf{m} are multinomially distributed:

$$\begin{aligned}
L(y, z) &= \text{Prob}\{\text{the histogram data is } \mathbf{m}, \text{ when the probabilities are } \{p_i(y, z)\}\}, \\
&= \frac{M!}{(M - \sigma_{n_0})! \prod_{i=n_0}^{\infty} m_i!} (1 - F_{n_0})^{M - \sigma_{n_0}} \prod_{i=n_0}^{\infty} p_i^{m_i}.
\end{aligned}$$

(The middle factor accounts for the probability that is not in the tail.) Taking natural logs of both sides, and then, for purposes of finding a maximum, taking the partial derivatives with respect to y and z and setting the result to zero, we obtain:

$$\begin{aligned}
\frac{\partial \log(L(y, z))}{\partial y} &= \sum_{i=n_0}^{\infty} \left(\frac{m_i}{p_i} e^{iz} - \frac{M - \sigma_{n_0}}{1 - F_{n_0}} e^{iz} \right) = 0, \\
\frac{\partial \log(L(y, z))}{\partial z} &= \sum_{i=n_0}^{\infty} \left(\frac{m_i}{p_i} i e^{iz} - \frac{M - \sigma_{n_0}}{1 - F_{n_0}} i e^{iz} \right) = 0.
\end{aligned}$$

Multiplying both equations by y/M and using (4.4), the first is equation is zero when

$$\frac{\sigma_{n_0}}{M} = F_{n_0},$$

This restricts the sum of the first moments of the tail to being equal. Using this in the second equation, we find

$$\sum_{i=n_0}^{\infty} i \frac{m_i}{M} = \sum_{i=n_0}^{\infty} i p_i.$$

This restricts the sum of the second moments of the tail to being equal. Using (4.4) and solving for z and y , we have

$$\begin{aligned}
z &= \log \left(\frac{\frac{H'}{H} - n_0}{\frac{H'}{H} - n_0 + 1} \right), \\
y &= \frac{H(1 - e^z)}{e^{n_0 z}},
\end{aligned}$$

where $H = \sum_{i=n_0}^{\infty} \frac{m_i}{M}$ and $H' = \sum_{i=n_0}^{\infty} \frac{i m_i}{M}$ are easily obtained from the simulation data. Using these as estimates for y and z , we can calculate estimates for the probability that a cell is lost.

We can also derive a simple upper bound on the blocking when the buffer is length B . We first assume that the buffer is infinite. The total probability flow from all states beyond B to states B or below is at most equal to all the probability beyond B , that is, F_{B+1} . This must also bound the total flow in the opposite direction, i.e., the blocking. When we make the buffer finite, but with F_{B+1} small, then the probability flow will remain approximately the same. Thus, we must find the B so that $F_{B+1} = \frac{ye^{z(B+1)}}{1-e^z} < P_L$, that is;

$$B > \left(\frac{1}{z} \log(P_L(1 - e^z)/y) \right) - 1.$$

The only question that remains to be addressed is what n_0 to use to estimate y and z . We choose the largest n_0 so that $\sigma_{n_0}/M > 0.01$. To provide some indication of the robustness, we also did the estimation choosing the largest n_0 so that $\sigma_{n_0}/M > 0.02$, and so that $\sigma_{n_0}/M > 0.005$. In general, the effect of changing n_0 increased as we estimated further into the tails, and with noisier data (e.g., batch arrivals). At worst, the variation due to changing the cut-off was less than 20%; more typically less than 10%. Note that because the distribution tails off at least exponentially fast, dramatic refinements to the method only bring about small improvements in the upper bound estimates. For example, making certain stronger assumptions, resulting in a 10dB reduction in the blocking probability estimate, reduces the bound on the necessary buffer size by less than approximately 20%. So, despite our crude methods, the bounds that we produce are still useful.

Bibliography

- [1] Judd, J. S., *Neural Network Design and the Complexity of Learning*, MIT Press, Cambridge, MA, 1990, Introduction.
- [2] T1.105, "Digital Hierarchy Optical Interface Rates and Formats Specifications," *American National Standards for Telecommunications*, 1988.
- [3] Editor: Sinha, R., T1S1.1/90-001 R1, "T1S1 Technical Sub-Committee; Broadband Aspects of ISDN Baseline Document," April 1990.
- [4] Feng, T. Y., "A Survey of Interconnection Networks," *IEEE Computer Magazine*, Vol. 14, No. 12, pp. 12–27, Dec., 1981.
- [5] Wu, C. L., Feng, T. Y., "On a class of Multistage Interconnection Networks," *IEEE Transactions on Computer*, Vol. 29, No. 8, pp. 694–702, Aug., 1980.
- [6] Karol, M. J., Hluchyj, M. G., Morgan, S. P., "Input vs. Output Queueing on a Space-Division packet Switch," *IEEE Transactions on Communications*, Vol. 35, pp. 1347–1356, Dec. 1987.
- [7] Ahmadi, H., Denzel, W. E., "A Survey of High-Performance Switching Techniques," *IEEE Journal on Selected Areas in Communications*, Vol. 7, No. 7, Sept. 1989, pp. 1091–1103.
- [8] Batcher, K. E., "Sorting Networks and their Applications," *Proceedings AFIPS*, 1968, *SJCC*, 1968, pp. 307–314.
- [9] Liu, K. H., "Internal-Blocking-Free-Sequences and Banyan Networks," *Pacific Bell Advanced Technology Division Technical Memorandum*, September 14, 1989.

- [10] Brown, T. X., Liu, K. H., “Neural Network Design of a Banyan Network Controller,” *IEEE Journal of Selected Areas in Communications*, to appear Vol. 8, No. 7, Oct., 1990.
- [11] Chu, W. W., “Buffer Behavior for Batch Poisson Arrivals and Single Constant Output,” *IEEE Transactions on Communication*, Vol. 18, No. 5, pp. 613–618, Oct. 1970.
- [12] Giacomelli, J. N., Sincoskie, W. D., Littlewood, M., “Sunshine: A High Performance Self-Routing Broadband Packet Switch Architecture,” *International Switching Symposium '90*, Paper 138, pp. 1–6, June 1990.
- [13] Hluchyj, M. G., Karol, M. J., “Queueing in High-Performance Packet Switching,” *IEEE Journal of Selected Areas in Communications* Vol. 6, No. 9, pp. 1587–1597, Dec. 1988.
- [14] Kleinrock, L., *Queueing Systems Volume I: Theory*, John Wiley and Sons, New York, 1975.
- [15] Hopfield, J. J., Tank, D. W., “Neural Computation of Decisions in Optimization Problems,” *Biological Cybernetics*, Vol. 52, pp. 141–152. 1985.
- [16] Marrakchi, A., Troudet, T., “A Neural Net Arbitrator for Large Crossbar Packet-Switches,” *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 7, pp. 1039–1041, Jul. 1989.
- [17] Press, W. H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T, *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, 1988.
- [18] Lawler, E. L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976, p. 195.
- [19] Berge, C., *Graphs*, North-Holland, Amsterdam, 1985, p. 129.

- [20] Kuwahara, H., Endo, N., Ogino, M., Kozaki, T., Sakurai, Y., Gohara, S., “A Shared Buffer Memory Switch for an ATM Exchange,” *Proceedings ICC*, pp. 118–122, 1989.
- [21] Kramer, K. K., private communication, June 1990.

Chapter 5

Epilogue

In this thesis, we proposed that neural networks have an intrinsic value as a framework for designing massively parallel machines. To aid in such designs, we developed three generalizations to the Winner-Take-All network and proved that they were stable, and only stabilized on valid states. Although the basic model that we use is similar to other Winner-Take-all circuits proposed, the formulation that we presented can readily be incorporated into larger systems, neural and non-neural. The Multiple Overlapping Winner-Take-All generalizations are useful for including many types of constraints and restrictions on the possible solution states. It is interesting that other than a small issue of gain these results are independent of the details of the underlying neurons, reinforcing the folk theorem that the power of a neural network is “in the connections.”

These concepts were used to design two solutions to problems from circuit switching. The pathfinder-for-multistage-switches problem demonstrated how the many neurons in a network could realize a straightforward parallel algorithm. The neural network for rearranging calls provided insights into programming a network using external inputs to perform a particular task. The application to the Beneš network took this idea to an extreme. The external inputs initiated a program which the neural network used to compute inputs that programmed a second level of neurons, which in turn programmed a third level, and so on.

The last problem that we investigated was a neural network contention arbitrator for packets in a high-speed packet switch. We developed a useful class of switches for

which the neural network is a direct consequence of the switch architecture. Using the design techniques, we were able to design a neural network with over 1000 highly interconnected neurons that not only was stable but only converged on valid states. The basic design was extended to include various functionalities including packet priority and traffic balancing. Extensive simulations showed that the performance of the network compared favorably with that of other techniques, both in terms of maximum throughput and queueing performance. The simulations also showed that with the massive parallelism of a neural network behind it, the blocking Banyan network has queueing performance similar to that of a non-blocking switch. The controller was extended to design a large (1024-input) ATM switch and controller using a modest number of components.

Although the neural model that we used is far removed from the biology by which artificial neural networks were inspired, it is interesting to note that the interconnections in these circuits and designs are almost exclusively inhibitory, with excitation arriving in the form of external inputs. This has many gross similarities to some biological neural systems: many local inhibitory connections, with but a few long range, often sensory, excitatory connections. The implication is that we may have insight into the reason why large biological neural systems are in the form that they are. Conversely, the direction in which we are heading can lead to extremely large and complex artificial neural systems that are both stable and extremely powerful.

This biological issue aside, the work in this thesis certainly has implications beyond the three problems studied here. It is hoped that it will inspire neural solutions to problems from communications and elsewhere.