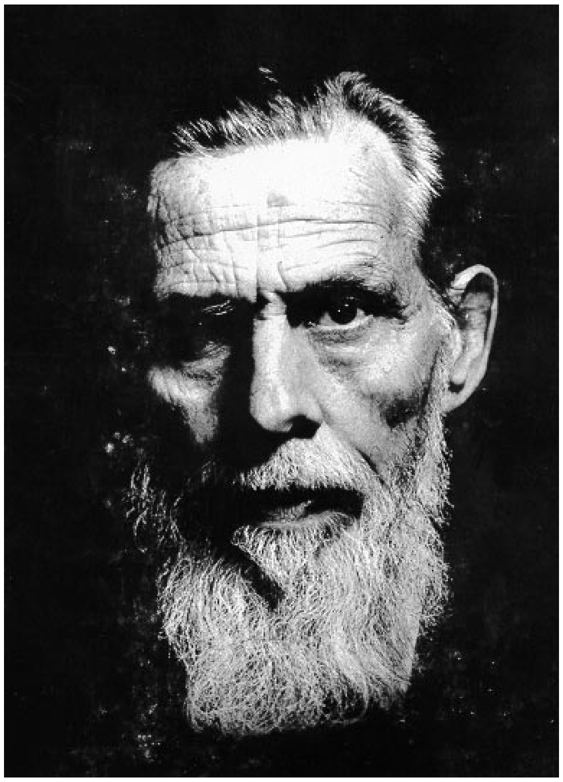

NET PROPHET: McCulloch and developments from his neural net model

Paul Cull

96-20-01

February, 1996

Department of Computer Science
Oregon State University
Corvallis, Oregon 97330 USA

NET PROPHET: McCulloch and developments from his neural net model

Paul Cull
Department of Computer Science
Oregon State University
Corvallis, Oregon USA
97331–3202
pc@cs.orst.edu

Technical Report 96–20–1

1 Introduction

Warren McCulloch died about 25 years ago. Almost 50 years ago, he and Walter Pitts published one of the most cited papers in history. I refer, of course, to the now classic paper, *A Logical Calculus of the Ideas Immanent in Nervous Activity* [30].

As befits a groundbreaking paper, there is a simply stated underlying idea, which can be cast in the following syllogistic form:

1. the brain carries out logical thinking
2. logic describes logical thinking
3. therefore, the brain's functioning can be described by logic.

This idea has behind it the weight of two thousand years of Western philosophy, making it at once both believable and banal. How can such an obvious idea have any interesting consequences? Here McCulloch and Pitts make a great leap from philosophy to physiology. They identify the abstract elements which compute logical functions with the neurons of the brain. Suddenly abstract words like "AND" and "OR" become concrete, viewable, measurable, biological entities.

Why hadn't this identification been made before? I don't know, but at one place, at one time, the people with the right knowledge and viewpoints came together. One should at least mention that models of the neuron and neural nets predate McCulloch and Pitts. For example, Rashevsky [36] and Hill [22] proposed models of the neuron, but their models were based on differential equations. Further, Rashevsky and his group [37, 23] created networks of their model neurons and used these networks to explain experimental results from physiology and psychology.

How were McCulloch and Pitts different from their counterparts? Walter Pitts never completed a degree, but according to legend he is the only person, other than Bertrand Russell, to have read and understood all of Whitehead and Russell's *Principia Mathematica.* Where Russell had attempted to reduce all of mathematics to logic, Pitts would try to reduce all of life to logic. Warren McCulloch, on the other hand, had completed degrees in philosophy, theology, and medicine, and mathematics was not his strongest suit. McCulloch was interested in the Big Picture. He wanted to create a new philosophy that was appropriate for a scientific age. In this new philosophy, vague metaphysical concepts had to be replaced by concrete elements which could be measured and manipulated by the scientist. McCulloch's attitude can be seen from a little joke he used to tell. For many years, he was associated with the EE department at MIT. For most people, he explained, EE stood for electrical engineering, but for him, EE stood for experimental epistemology.

Perhaps such backgrounds were necessary to create the logical model of the neuron, because in creating a model one must pick out exactly the relevant part of a phenomenon and ignore all of the other aspects. Newton did not record whether the apple was red or green, only that it fell down. Galileo demonstrated that weight was irrelevant by dropping spheres of different weight, but he did not drop bodies with different shapes. So McCulloch and Pitts used physiological evidence in support of their model, but ignored details like the biochemistry which were irrelevant to their model.

Of course, a model, as well as describing its phenomenon, must be in concert with the spirit of its age. A model not in concert with its time will probably have zero impact. Luckily the logical model of McCulloch and Pitts found acceptance because it was only slightly ahead of its time. In 1943, there were no digital computers, and the locution "electronic brain" was still a decade in the future. As someone has remarked, we try to compare the brain with the most complicated artifact we have. In the early 40's, this most complicated artifact was the telephone exchange, and the brain had been compared to a very large telephone exchange. The idea of describing an exchange as a Boolean contact network was in its infancy, and so the McCulloch-Pitts model did reflect the technology of its age. But the McCulloch-Pitts model was a gate network rather than a contact network. The importance and technological relevance of gate networks was described a few years later by John von Neumann [33]. In this most widely cited unpublished paper, von Neumann described how a slightly modified version of the McCulloch-Pitts model could be used to describe the logical design of a digital computer. In creating the logical description, von Neumann was pointing out that from the perspective of what one wants a computer to do, the underlying electric or electronic properties are irrelevant. A computer could be designed using physical properties, other than electronics, without changing the logical design.

I hope this brief background indicates that the McCulloch-Pitts model was the right model at the right time. It would take several volumes to even outline the developments in theory and application of neural nets in the past 50 years. So in this brief paper, I will have to content myself with mentioning a few subareas that my colleagues and I have been working on recently.

2 Classical Results

The brilliance of the McCulloch-Pitts model lies in the simplicity of its individual elements, neurons, and the complexity which can arise out of an interconnection of these simple elements. As usually presented, the model neuron has a set of weights $w_1, w_2, ...w_k$, and a threshold h. These parameters are allowed to be any real number, so, in particular weights may be negative or positive. The neuron has k input lines, each of which can carry a 0 or a 1. These lines will be called $x_1, x_2, ..., x_k$, and the signal on the i^{th} line at time t will be represented by $x_i(t)$. A time unit is chosen so that the output of the neuron at time $t+1$ will be a function of its inputs at time t. The function used is usually a linear threshold function; that is, the neuron takes a weighted sum of its inputs, compares this sum with the threshold, and outputs 0 or 1 depending on the result of the comparison. In equation form,

$$Out(t+1) = H(\sum_{i=1}^{n} w_i x_i(t) - h)$$

where H is the nonlinear Heaviside operator which takes positive numbers to 1 and nonpositive numbers to 0. Actually this is one of the two model neurons discussed in the classic McCulloch-Pitts paper. The other model treats positive and negative weights separately. If an input line corresponding to a negative weight carries a 1 at time t, then the neuron will output a 0 at time $t+1$. If none of the negative weight input lines carries a 1, then the output at $t+1$ is computed according to the above equation.

Behavior impossible for a single neuron can be obtained by creating neural nets; that is, by connecting the ouputs of some neurons to the inputs of some neurons. Some of the inputs may still be from external sources. McCulloch and Pitts showed that even though their two model neurons were different, nets of either type could be simulated by nets of the other type. Further, their simulations were carried out by local replacements, so the number of neurons in a net of one type could be bounded by a constant multiple of the number of neurons in a net of the other type.

McCulloch and Pitts recognized two major problems for neural nets: the analysis problem, and the synthesis problem. The *analysis* problem is: Given a neuron net, determine its dynamical behavior. The *synthesis* problem is: Given a desired behavior, either build a neural net with that behavior or show that no such net is possible. McCulloch and Pitts gave simple solutions for both the analysis and synthesis problems for loop-free nets. A net is loop-free if it is impossible to find a neuron and a path from ouput to input to output to ... which starts at a neuron and returns to the same neuron.

LOOP-FREE SYNTHESIS: Let F be any function from $\{0,1\}^m$ to $\{0,1\}$; then there is a loop-free net of n neurons so that

$$Out(t+2) = F(Y_t)$$

and $n \leq 2^m$.

COROLLARY: Let G be a function of input sequences of length d; then there is a loop-free net so that

$$Out(t+2) = G(Y_t, Y_{t-1}, ..., Y_{t-d+1})$$

LOOP-FREE ANALYSIS: For a loop-free net of depth d with m input lines and k output neurons, there is a function F from $\{0,1\}^m$ to $\{0,1\}^k$ so that

$$Out(t+1) = F(Y_t, Y_{t-1}, ..., Y_{t-d+1})$$

.

McCulloch and Pitts also give theorems about nets with loops. Unfortunately, this part of their paper is very difficult to read. According to legend, only Walter Pitts really understood what this part of their paper meant. A clear statement about nets with loops was given by Kleene [26].

KLEENE'S THEOREM: The following three statements are equivalent:

1. L is the set of input sequences which takes a neural net from a designated start state to some state in a set of designated on-states.
2. L is a language recognized by a finite automaton.
3. L is a language represented by a regular expression.

In essence, this theorem solves both the analysis and synthesis problems. To analyze a neural net, one can construct the corresponding finite automaton. A net with n neurons has 2^n states. For each state and each setting of the inputs, one can determine the unique next state. Hence, by drawing a state diagram with labeled arrows for each input setting, one can obtain a full description of the behavior of the neural net. For synthesis, if one can show that no finite automaton exists for a desired behavior, then one knows that no neural net can display the desired behavior. On the other hand, if a finite automaton with n states $q_1, q_2, ...q_n$ and m input symbols $s_1, s_2, ...s_m$ exists, then one can construct a corresponding neural net as shown in Figure 2. The circled ANDs in the figure represent neurons which turn on if at least two of their input lines are on. In the simulation of the automaton, exactly one AND cell will be on at a time. For example, if the automaton is in state q_n receiving symbol s_1, then the upper right AND cell is the only one which is on. If the automaton should change state to state q_2, the output line from this AND cell is connected to the input of each cell in the column for q_2. Now, if exactly one line for an s_i is on, a single cell in column 2 representing state q_2 and symbol s_i will go on.

These classical results say what can and can't be done with neural nets. As we will see in subsequent sections, these results also raise questions which we are still trying to answer.

3 Complexity

One of the most attractive properties of neural nets is that their finite size and finite number of states allow algorithmic methods for solving questions about them. But, in the past two decades, the study of computational complexity has lead to the classification of problems as "easy" and "hard". Those problems which possess an algorithm with run time bounded by a polynomial in the size of the input are easy problems. I will call such algorithms *fast* algorithms. Problems without such an algorithm are hard. Even when one has no proof of the non-existence of a fast algorithm, one can still call the problem hard if one can show that the problem is the hardest problem within some well-defined class of problems. For example, NP is the class of problems which have fast algorithms, if one augments the usual computer instructions with a magical "guess" instruction, which always chooses the correct alternative from a set of possibilities. A hardest problem for NP is called NP-Complete, and NP-Complete problems are considered too hard to be practically algorithmically solvable.

The point, of course, is that many of the questions one would like to answer about neural nets are NP-Complete problems. Consider a single neuron with n inputs, and some Boolean function which specifies the output as a function of the inputs. If one allows a general Boolean expression, and asks if there is some setting of the inputs that will turn the neuron on, one is faced with an NP-Complete problem. In fact, this is the well-known SAT problem (see Garey and Johnson [15] for more details). For McCulloch and Pitts neurons, on the other hand, this

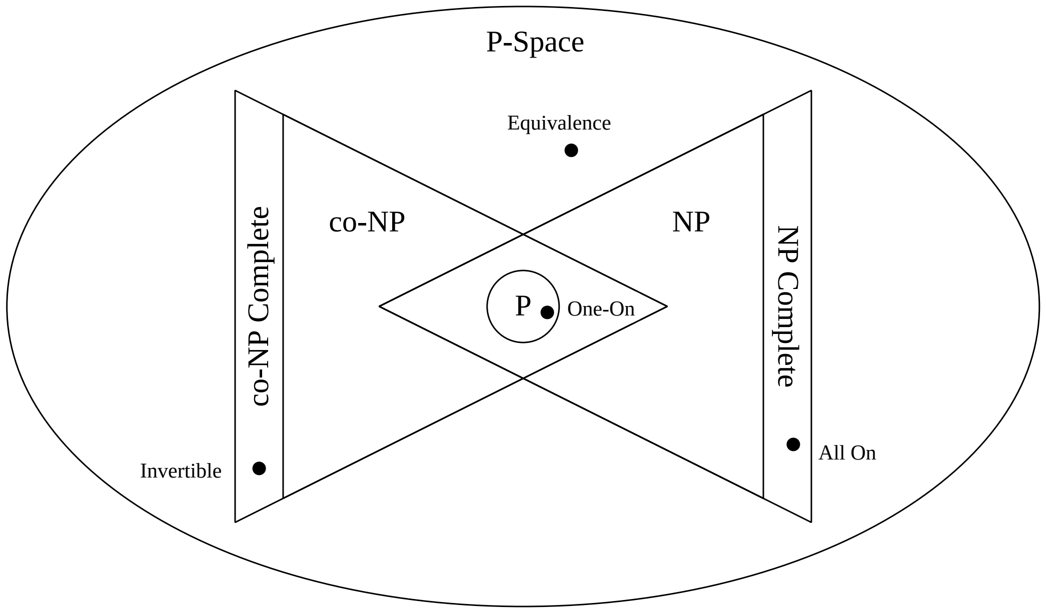

Figure 1: Relationship between complexity classes and some problems.

problem is easy. To determine if there is an input setting which turns the neuron on, one only has to check to see if the sum of the positive weights is greater than threshold. If so, there will be one or more input settings which turn the neuron on.

On might hope that the special McCulloch-Pitts form would allow one to avoid NP-Completeness, but NP-Completeness quickly appears in problems which seem only slightly more complicated. The following problems about McCulloch-Pitts neural nets are NP-Complete:

1. Is there a setting of the inputs which simultaneously turns on all the neurons in a (single-level) neural net?

2. Given an autonomous neural net and a specified state, is there a state such that if the net is started in this state, then the net will enter the specified state?

3. Given a loop-free net with n inputs and n outputs, does the net compute a noninvertible function from the inputs to the outputs?

4. Given an autonomous net, are there some transient states?

5. Given two loop-free nets, is the function computed by one net different from the function computed by the other net?

This is just a sample of some of the questions about neural nets which turn out to be NP-Complete, but there are questions about neural nets which seem to be even harder. For example, the equivalence problem for neural nets is PSPACE-Complete. By PSPACE we mean the class of all problems which can be solved by algorithms which use memory space bounded by a polynomial in the size of the input. As usual, complete means the hardest problem in this class. The equivalence problem is determining if the two neural nets recognize the same set of input sequences. A neural net functions as a recognizer when there is a specified initial state and a specified output neuron. The neural net recognizes an input sequence when the neural net is started in the specified initial state, is fed the input sequence, and turns on the output neuron when the input sequence is finished.

The point is that even though there are algorithms that answer questions about neural nets, these algorithms may be practically useless because they use unreasonably large amounts of time. Figure 1 displays the relationship between the mentioned complexity classes and the locations of several problems.

4 Size

McCulloch and Pitts [30] and Kleene [26] exactly characterized the abilities of neural nets. In particular, they showed that every Boolean function can be computed by a loop-free neural net, and that every finite state machine can be simulated by a neural net with loops. These demonstrations were constructive and therefore give upper bounds on the number of neurons required.

These constructions suggest two natural questions:

1. Are these constructions worst case optimal? That is, do the constructions use the fewest neurons possible for some situations? Of course, we expect there to be specific functions or machines which can be computed by smaller neural nets than those given by the constructions.

2. Can one name specific functions or machines for which the constructions are optimal? This question seems to presuppose that the answer to (1) is yes, but if one finds a better construction, then (2) can be asked about the new construction.

McCulloch and Pitts showed that every Boolean function of n variables can be computed by a loop-free neural net with at most 2^n neurons. Their construction made use of the disjunctive canonical form which states that any Boolean function can be computed as a disjunction (ORing together) of terms, where each term is a conjunction (ANDing together) of literals, and a literal is a variable or the negation of a variable. This form is canonical when each variable appears exactly once in each term. When all variables are not required to appear in every term,

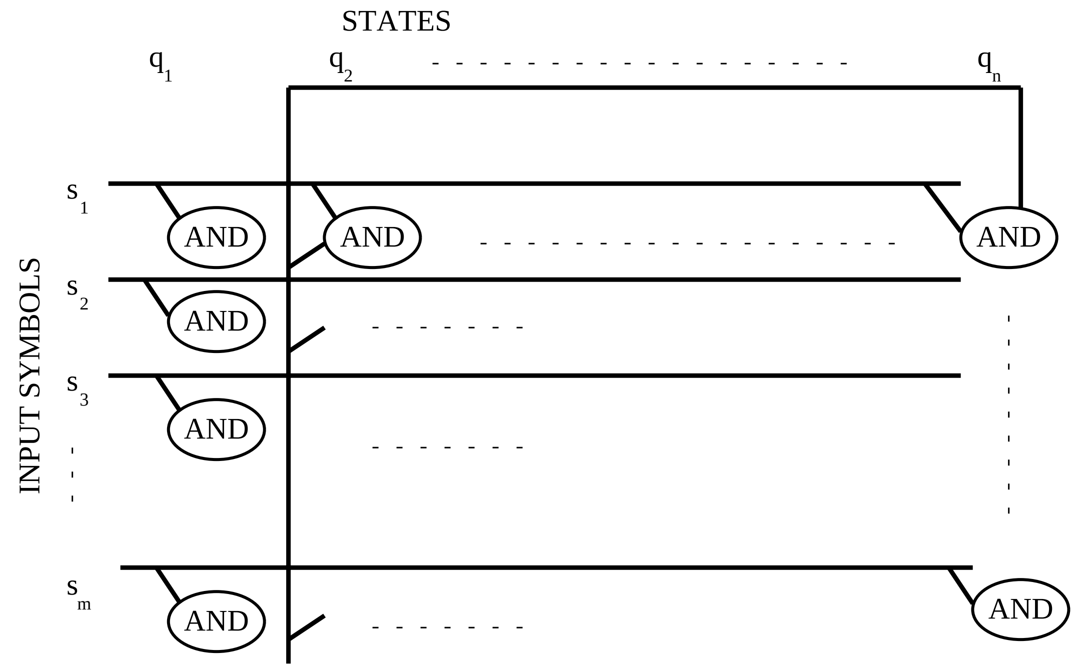

Figure 2: Every n state, m symbol finite automaton can be represented by an O(nm) neuron net.

the form is called a disjunctive normal form. A function may have a number of different normal forms, but the canonical form is unique up to the order in which the terms appear, and the order in which the variables appear in a term.

The McCulloch-Pitts construction is based on the observation that each term can be computed by a single neuron, and the ORing of terms can also be computed by a single neuron. This construction may not be practical because the term neurons will have n inputs and the OR neurons may have about 2^n inputs. The number of inputs is usually called the "fan-in", and for practical nets there should be some fixed constant upper bound on the fan-in. Further, this construction can badly overestimate the number of neurons needed to compute a function. For example, the "majority" function which ouputs 1 if at least half of the inputs are on, can be computed by a single neuron, but any method based on disjunctive normal form will use about $2^n/\sqrt{n}$ neurons.

In one sense the size problem has been solved. Shannon [42] showed that with fan-in 2 almost all functions require $2^n/n$ neurons, and Lupanov [28] gave a construction which built a net with about $2^n/n$ neurons. On the other hand, these results are disappointing because they don't specify a function which requires this number of neurons. In fact, Wegener [45] states that no one knows how to prove that any specific function requires more than a constant times n neurons.

For machines, it is well known [31] that any n state machine can be simulated

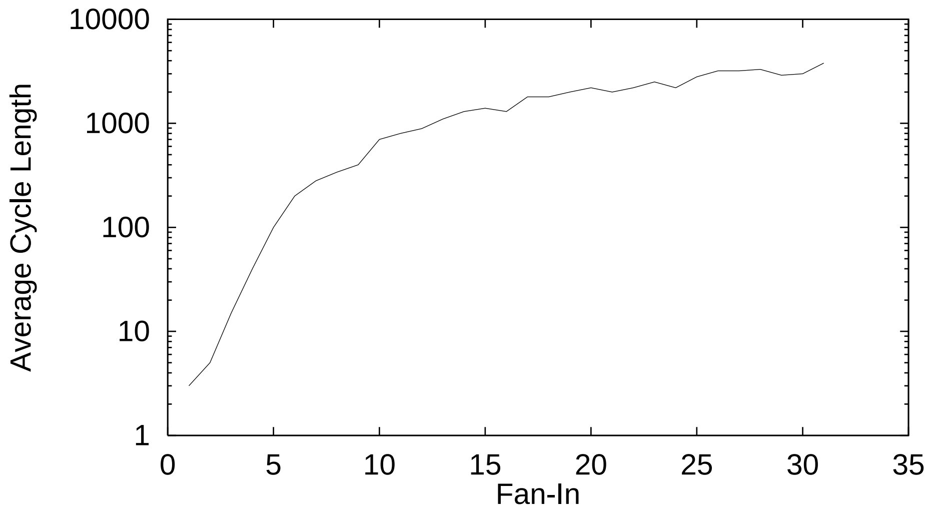

Figure 3: Simulation of randomly chosen 31 neuron nets.

by a net with about n neurons. Recently Alon et al. [1] have given a construction which uses about $n^{3/4}$ neurons. They also show that at least $(n\, log\, n)^{1/3}$ neurons are required. More satisfactorily, they show that with constant fan-in, or with simultaneous limits on fan-out, weights and thresholds, then about n neurons are required. These results present the seeming paradox that the 2^n states of an n neuron neural net are required to simulate the n states of a machine. A possible explanation is that the Minsky construction works for both deterministic and nondeterministic machines, and it is known that some n state nondeterministic machines require about 2^n deterministic states. So it may be that the number of neurons required to simulate a machine is related to the minimum number of nondeterministic states rather than to the number of deterministic states.

5 Random Nets

Some years ago, Kauffman [24] proposed that certain biological phenomenon could be explained if one could understand the state cycles generated by autonomous neural nets (nets with no inputs). By simulating randomly chosen nets, Kauffman showed that nets with low fan-in had short state cycles, while nets with high fan-in had very long state cycles. Figure 3 shows some results of our simulations confirming Kauffman's results [10].

Since random nets in which every neuron receives an input from every neuron correspond to state mappings that are random mappings, reasonable probabilistic techniques can be used to study the expected behavior of such nets. Such results were computed long ago by Rubin and Sitgreaves [41], and summarized more recently by Gelfand [16]. To study nets with other than complete fan-in, other techniques seem to be needed.

Back in 1971, I developed a linearization technique for studying neural nets [8]. Because the linearization was over the finite field GF(2), it did not allow the calculation of expected values. Subsequently, Caianiello [6, 7] developed a rational field linearization. In [9, 11], I have shown how the results for nets with complete fan-in can be computed using Caianiello's rational linearization. In the future, I hope to be able to use the linearization to compute the expected behavior of nets with other fan-ins.

On the other hand, perhaps the complicated calculations based on the linearization will be unnecessary, because cycle length may obey a 0-1 law. One version of a theorem of Fagin [13] says that if a property of a structure can be stated in first order logic, then, as the number of elements in the structure goes to infinity, the probability that the structure has the property goes exponentially to 0 or to 1. For example, this theorem can be used to show that there is a $\hat{c}$, so that if every vertex in a graph with n vertices has degree greater than $\hat{c} \ log\, n$, then the probability that the graph has a Hamiltonian cycle goes to 1 as n goes to infinity. Similarly, if the degree is less than $\hat{c}\, log\, n$, then the probability that the graph has a Hamiltonian cycle goes to 0. Such a 0-1 law might apply to neural nets. There may be a critical value of the fan-in, so that nets which have fan-in greater than the critical value will have long state cycles with high probability, while nets with fan-in less than the critical value will have short state cycles with high probability.

We have run into some difficulties trying to apply Fagin's theorem to random neural nets. First, we had difficulty trying to state the property that the net has a long state cycle as a first order property of a net. The difficulty is that the property occurs in the state space, which is somewhat removed from the connections and functions used to describe the neural net. In fact, it can be shown that even the seemingly simple statement that there is a path between two specified states cannot be expressed in first order logic (see Papadimitriou [34]). Second, if there is a critical value of fan-in, the critical value should occur between 2 and 3. We have carried out detailed simulations in this range and found a gradual increase in average cycle length rather than a step-like transition from low to high cycle length. Further, when we looked at the increase in cycle length as a function of the number of neurons, there seemed to be a slow increase rather than the exponential increase predicted by Fagin's theorem.

Another concept closely related to randomness is "chaos". Experimentalists have been measuring brain waves for years, and having great difficulty explaining these waves in terms of linear systems theory. Recently some researchers, for example Freeman [14], have suggested that brains are highly nonlinear systems and so chaotic behavior should be expected in normally functioning brains.

We reasoned that, if chaos is typical, then we should also expect to find it in simulated networks. We attempted to find chaos by looking at the correlation dimension of the eventual trajectory found by picking a random starting state, and running a net for 100 time units to get rid of transients. As expected, for low fan-in we got cycles that were too short for our method to measure their dimension. Also as expected for high fan-in nets, the method reported that the trajectories were essentially random and did not measure dimension. We expected that medium fan-in nets would have measurable dimension, but the method also reported that these trajectories were essentially random. More details can be found in [10].

Perhaps another method would allow us to find chaos, or perhaps there is a 0-1 law saying that a net has either small cycles or essentially random cycles.

6 Learning

The explosive rebirth of interest in neural nets is largely based on the ability of neural nets to learn. Clearly, the ability to learn is a property of the nervous system, and since neural nets were meant to model nervous systems, the ability of neural nets to learn was at least implicit in the original McCulloch-Pitts formulation. How neural nets were meant to learn is a little unclear. Since neural nets are specified by the interconnection weights between neurons, and by the thresholds of the neurons, a learning mechanism must involve some sort of rule for changing these quantities. There are two obvious sorts of rules: one for supervised learning, and one for unsupervised learning. In supervised learning, one can look at a neural nets' behavior and tell the net when its behavior does not match the desired behavior. Thus learning rules for supervised learning make the neural net adjust its weights so as to reduce the difference between its behavior and the desired behavior. In unsupervised learning, the neural net must notice relationships within its environment. Thus, for unsupervised learning, the learning rule should depend on some correlation between the firing of neurons. This rule may be as simple as: if at the time t neuron i and neuron j both fire, then increase the strengths of connections between i and j. Both of these types of learning rules were proposed and used many years ago. For example, Rosenblatt's perceptron learning rule [39] is an error decreasing rule, while Caianiello [5] gives a correlation based learning rule.

The currently most popular learning rule is back propagation. It is based on making changes in the weights proportional to the partial derivatives of the error with respect to the weight. To make this process of taking derivatives possible, one replaces the threshold nonlinearity of the McCulloch-Pitts model with an S-shaped nonlinearity. By choosing this S-shaped function as a simple differentiable function, the back propagation changes can be easily calculated. Because of this seemingly easy algorithm, all sorts of wild claims have appeared. For example, some have claimed that with trainable neural nets, programmers will be obsolete. Instead of writing programs, one can simply train a neural net by presenting it with a few examples of inputs and desired outputs.

Even if neural nets can learn is that enough? One is reminded of an old fisherman's story. When someone asked the fisherman, "Can fish learn?" He replied, "Sure!" Then as he pulled up another fish, he added, "Luckily they don't learn fast enough." In a similar sense we want learning rules, so that a neural net can learn a desired behavior, but we also want the learning to be fast. As in most algorithmic situations, one equates fast with the existence of a polynomial time bound. Different definitions will allow the polynomial to be a function of different parameters.

Valiant [44] came up with the currently most widely accepted definition of efficiently learnable. This is usually called the PAC model, where PAC abbreviates "probably almost correct." "Almost correct" means that there is an error parameter, and the behavior of the learned program is only guaranteed to be within this error parameter of the behavior of the intended program. The "probably" introduces another parameter, so that the guarantee of almost correct is only with within the probability given by this parameter. This model is distribution-free because it requires learning within these parameters regardless of the distribution of learning examples, but the distribution is taken into account when one calculates almost and probably. To be efficient, the run time of the learning algorithm must be bounded by a polynomial in the size of its inputs and outputs, and in the reciprocals of the error and probability parameters. These reciprocals are used because the parameters go to 0 for guaranteed error free learning, and we expect the algorithm to have to work harder as these parameters go to 0.

One would be pleased to state that a large variety of problems are PAC-learnable. Unfortunately, this may not be the case. Except for a few very simple examples which can be shown to be PAC-learnable, almost all results say that such and such is not PAC learnable unless some expected complexity result is false.

An example of a class of Boolean functions which are PAC-learnable is the class of functions which can be represented as the ANDing together of complemented and uncomplemented Boolean variables. Since these functions can be represented by a single McCulloch-Pitts neuron, it is not surprising that this class can be learned. In fact, the classical perceptron learning rule will learn this class. The perceptron rule is guaranteed to converge in a finite number of steps, but there is no obvious bound on how many steps the perceptron rule will use. While a PAC-algorithm may produce an incorrect answer, with high probability it will produce the correct function in time bounded by a polynomial which is almost linear in all the parameters. In this example, one can trade off a guarantee of correctness against a guarantee of speed.

Unfortunately, most of the recent theoretical results on learning are negative rather than positive. For example, Blum and Rivest [4] showed that the question: Given a desired behavior and a 3-neuron neural net, is there a learning algorithm so that the net will learn the desired behavior? is an NP-complete problem. Pitt and Valiant [35] have shown that under a reasonable assumption about complexity classes, a disjunctive normal form formula with three disjuncts cannot be learned from examples in polynomial time. Again, under a reasonable complexity assumption, Reif [38] has shown that there is a class of functions computed by fixed depth, loop-free, neural nets, so that these functions cannot be PAC-learned in polynomial time.

On the positive side, efficient learning is possible when the learner is not restricted to seeing a sequence of examples. As every mathematics teacher knows, a student often can correctly solve some problems, but fail to solve others, because the student has created a method which only works in special cases. By examining the student's method, the teacher can often show where the student's method fails and lead the student toward the correct method. A strategy like this can be used to learn a finite automaton. The learner starts with some automaton. The teacher then provides an example in which the learner's automaton gives different output than the desired automaton. The learner then asks questions about what output is desired on certain inputs. After a number of rounds like this, the learner will have learned the desired automaton. Further, all of the learner's work will take time polynomial in the size of the desired automaton and the size of the teacher's examples. Since a neural net with loops is exactly equivalent to a finite automaton, a neural net can be learned in the same way. This positive result was demonstrated by Angluin [2]. Notice that for this type of learning the teacher must be very powerful–much more powerful than the simple giver of examples in PAC learning.

To find out more about learning theory, a good place to start is Hearns and Vazirani [25]. Theoretical results on learning appear every year at the Conference on Learning Theory (COLT). Applications of learning neural nets often appear in the yearly conference of the International Neural Net Society (INNS).

At present, the field of learning in neural nets consists largely of negative theoretical results, and a myriad of papers claiming to show that neural nets can be practically used to learn a wide variety of functions. At this point, many of the practical papers are marred by poor experimental design, and by exaggerated claims that run counter to known theoretical results. We can hope that in the future, there will be a rapprochement between theory and practice, with the theoreticians analyzing models which more closely describe what practioners want to do, and with the practioners using better experimental design and theory to create better experiments.

7 Learning To Decode

As an example of a situation in which a trainable neural net may provide a reasonable solution to a problem, we consider the decoding of an error-correcting code. This example is based on Tallini and Cull [43].

In some sense, this problem has a trivial solution, but we want to argue that

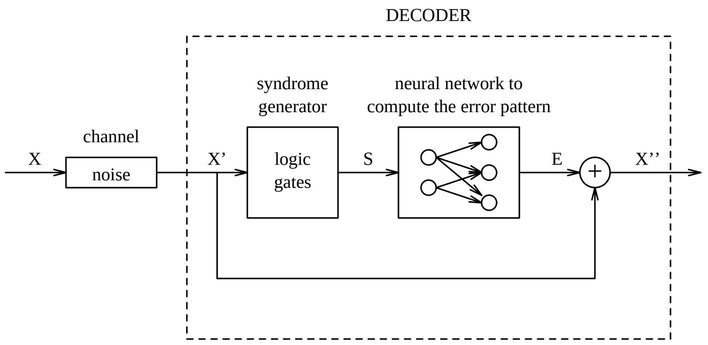

Figure 4: Proposed decoder architecture.

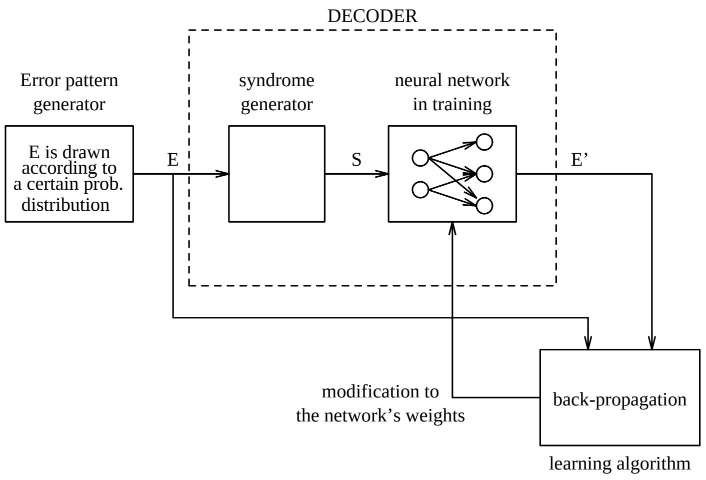
DECODER
Error pattern generator
E is drawn according to a certain prob. distribution
syndrome generator
neural network in training
E
S
E'
back-propagation
modification to the network's weights
learning algorithm

Figure 5: Proposed training procedure.

in the idealized solution a number of features of the realistic problem have been glossed over. When the realistic features are added to the problem, the ideal solution is no longer applicable, and the realistic problem is difficult. On the other hand, we will demonstrate that neural nets can provide a solution to this difficult realistic problem.

In an error-correcting code, one usually considers an n-dimensional Boolean vector space, in which certain vectors have been designated as codewords. When these codewords are transmitted over a noisy channel, the received vector may be a non-codeword. The general idea is that if only a few errors occur in the transmission then the received vector will still be close to only one codeword. Thus, decoding the received vector as the closest codeword will correct the errors made in transmission.

Since there are only 2^n possible vectors, one could construct a table which contains the closest codeword to each vector. With this table, decoding becomes the trivial task of table look-up. But clearly, even for moderate values of n, 2^n becomes too large to make this table reasonable. Hence codes are often constructed so that there is an efficient decoding algorithm. In particular, most known error-correcting codes are constructed based on the idea of linearity. For example, in the prototypic Hamming code, operating on the received vector with a matrix produces a vector which can be read as the binary number indicating which bit of the received vector is in error when there is exactly one error. When there is no error, the product of this matrix times the received vector produces the all zeroes vector. As usual, when we generalize, things become more complicated. When linear codes which can handle several errors are used, the product of the matrix with the received vector is not sufficient to locate the errors. One is faced instead with finding the solution to a set of linear equations. While solving a set of linear equations can be done quite readily by standard methods like Gaussian elimination, the linear system for decoding is under-determined and so has many solutions. The decoding problem then is to find the solution which corresponds to the fewest errors, and thereby locate the closest codeword. Berlekamp et al. [3] showed that this is an NP-complete problem. Hence decoding even for linear codes is a hard problem.

In realistic situations, things are even more complicated. For example, if each bit of a codeword is sent over a different wire, then the probability of an error may well be different for different bits. So the “closest” codeword to a received word may not be the one with the fewest changes, but the one with probabilistically fewest changes. For example, if wire 1 had a probability of error of .001, and wires 2 and 3 each had a .1 error probability, then having errors on both wire 2 and wire 3 would be more probable than having a single error on wire 1.

Assuming we are dealing with linear codes, there is a matrix H so that for every codeword X, $XH = 0$. If a codeword X is transmitted, then the received word X' can be written as $X + E$, where E is the vector of errors. Assuming linearity,

$X'H = (X + E)H = EH$. This vector EH is called the syndrome, and we will use the symbol S for syndrome. The decoding problem is to work from S and find a good estimate E' of the actual error vector E, and then output $X' + E'$ as the estimate of the sent codeword. Our plan is to have a neural net learn to calculate the mapping from syndrome S to most probable error E' .

In the following, by (n, k, d) we denote a linear code of length n, dimension k and minimum Hamming distance d. To test our method we chose four liner codes: the $(7, 4, 3)$ Hamming code (see Subsection 7.1), two shortened Hamming codes (see Subsection 7.2) and the $(32, 16, 8)$ 2^{nd} order Reed-Muller code [29] (see Subsection 7.3).

We implemented the neural decoders using the continuous neuron model. In this case, a neuron is the following computing device. It has $p+1$ input wires. For $i = 0, \dots, p$, the i-th wire carries a real number $x_i \in [0, 1]$ and has a real number w_i associated with it. The value x_0 is always set to 1. Further, the neuron has q output wires (feeding q different neurons), all carrying the output of the neuron, which is computed as follows:

$$x_{output} \overset{\text{def}}{=} \begin{cases} f\left(\sum_{i=0}^{p} x_i w_i\right), \\ \\ f(s) \overset{\text{def}}{=} \frac{1}{1+e^{-s}}. \end{cases}$$

Since $x_0 = 1$, w_0 represents the threshold of the neuron.

The neural decoder architecture is multi-layered: there is an input layer with r (= syndrome length) neurons, some internal layers, and an output layer with n (= error pattern length) neurons. Each neuron in the internal and output layers takes input from all the neurons of the previous layer. In order to digitalize the output from the neural network decoder, we set the output from a neuron of the output layer to 0, if the real output is less than or equal to 0.5, and to 1 otherwise.

To train the neural decoder we used a simple implementation of the back-propagation rule. Each connection weight is modified following the presentation of each training example, using changes given by:

$$w_{new} = w - \alpha \cdot \frac{\partial e}{\partial w}, \qquad \alpha \in [0.4, 0.45],$$

e being the total error on the network response to the example. Only the weights of wires which are input to internal or output neurons are changed. In the following, with $p_0 : p_1 : \dots : p_{d-1}$, we indicate a neural network with d layers containing p_i neurons at layer i, for $i = 0, \dots, d-1$.

We ran our computer programs on an HP-9000 workstation.

7.1 The $(7, 4, 3)$ Hamming code

In this subsection, we show that the best performances, in terms of degree and speed of learning, seem to be obtained when the training examples are uniformly

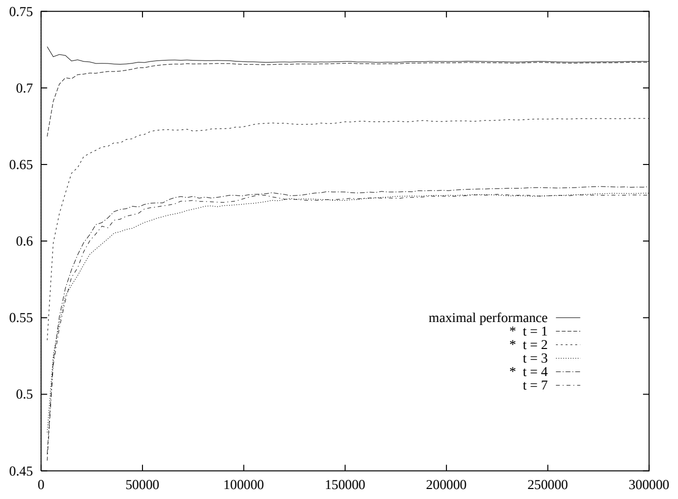

Figure 6: Learning behavior with disturbance.

distributed in the set of error patterns $\mathcal{E}'$ that we want the neural decoder to correct, provided that the function which associates an error pattern $E \in \mathcal{E}'$ with its syndrome is one-to-one. In particular, we compare the learning behavior of a neural decoder for a $(7, 4, 3)$ Hamming code $\mathcal{C}$ with respect to various learning schedules. The parity check matrix of $\mathcal{C}$ is the following:

$$H = \begin{pmatrix} 100 & 1101 \\ 010 & 1011 \\ 001 & 0111 \end{pmatrix}.$$

Note that $\mathcal{C}$ is a perfect code; i.e., $\mathbb{Z}_2^7$ is partitioned by the spheres of radius 1 centered at each codeword.

Figure 6 compares the learning behaviors of a $3:7$ neural decoder for $\mathcal{C}$, using five different learning schedules. We generated error patterns according to the probability distribution of a binary symmetric channel whose error probability is $\epsilon = 0.15$, but we trained the neural decoder only on those error patterns whose weight is less than or equal to t, for $t = 1, 2, 3, 4, 7$. In Figure 6, on the x-axes we represent the number of training examples generated so far. Each curve is the graph of the ratio between the number of training examples for which the network gives a correct answer and the total number of examples generated. The topmost curve is the ratio between the number of examples whose weight is less than or equal to 1 and the total number of examples generated. Since $\mathcal{C}$ is a perfect 1-error correcting code, this curve represents the maximal performance of $\mathcal{C}$ on the given binary symmetric channel. We attribute the learning behaviors represented in Figure 6 to a phenomenon which we call "disturbance." In this case, the neural decoder is faced during the training phase with two different error patterns having the same syndrome, causing an unstable output on those error

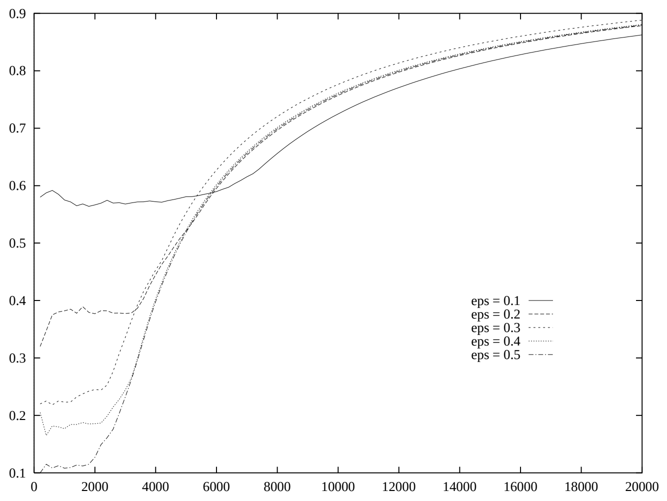

Figure 7: Learning behavior without disturbance.

patterns. Assume that the set of training examples contains a subset $\mathcal{E}(S)$ of error patterns having the same syndrome S. Then the neural decoder tends to learn the most likely element in $\mathcal{E}(S)$ with respect to the probability distribution defined by the learning schedule.

In Figure 7, we generated error patterns according to the probability distribution of five BSCs whose error probabilities were $\epsilon = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively. Further, we trained the neural decoder only on those error patterns whose weight is less than or equal to 1, to avoid disturbance. On the x-axes we represent the number of training examples generated so far. Each curve is the graph of the ratio between the number of training examples for which the network gives a correct answer and x. After $20,000$ training examples, all the neural decoders learned how to optimally decode $\mathcal{C}$; with the decoder which was trained with $\epsilon = 0.3$, the learning was fastest.

7.2 The shortened Hamming codes

In this subsection, we give some examples of how our approach can be used to design an optimal decoder of a (binary) linear code, for any given channel.

For example, assume we have a binary channel composed of $n = 8$ parallel wires, each one carrying a component of the transmitted binary vector X, where wire i carries the i-th component of X, for $i = 0, \ldots, 7$. Assume also that the i-th wire is a BSC with probability of error equal to ϵ_i, where:

$$(\epsilon_0, \epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4, \epsilon_5, \epsilon_6, \epsilon_7) = (0.05, 0.05, 0.05, 0.06, 0.16, 0.15, 0.15, 0.15). \quad (1)$$

Prob	Err. patt.
0.4157	$E_1 = 0000\,0000$
0.0792	$E_2 = 0000\,1000$
0.0734	$E_3 = 0000\,0100$
0.0734	$E_4 = 0000\,0010$
0.0734	$E_5 = 0000\,0001$
0.0265	$E_6 = 0001\,0000$
0.0219	$E_7 = 0010\,0000$
0.0219	$E_8 = 0100\,0000$
0.0219	$E_9 = 1000\,0000$
0.0140	$E_{10} = 0000\,1100$
0.0140	$E_{11} = 0000\,1010$
0.0140	$E_{12} = 0000\,1001$
0.0129	$E_{13} = 0000\,0110$
0.0129	$E_{14} = 0000\,0101$
0.0129	$E_{15} = 0000\,0011$
0.0050	$E_{16} = 0001\,1000$

Table 1: The first 16 most likely error patterns of the channel given in the example.

If statistical properties of channels don't change over time, every channel transmitting binary words of length n defines a probability distribution over the set of all error patterns $\mathcal{E} = \mathbb{Z}_2^n$: for all error patterns $E \in \mathcal{E}$, $Prob(E)$ is the probability that a word X is sent and $X + E$ is received. $Prob(E)$ is nothing but the probability of occurrence of the error pattern E during the transmission of a word. Table 1 shows the 16 most likely error patterns of our example channel (1), ordered in non-increasing order with respect to $Prob$.

Given $r, n \in \mathbb{N}$, let $\mathcal{C}$ be a binary linear code having, as parity check matrix, the $r \times n$ matrix H of maximum rank. The channel and $\mathcal{C}$ define the following relation in the set $\mathcal{E}$ of the 2^n error patterns. For $E_1, E_2 \in \mathcal{E}$, $E_1 \preceq E_2 \overset{\text{def}}{\Longleftrightarrow}$

$$Prob(E_1) \leq Prob(E_2) \quad \text{and} \quad E_1 \cdot H^T = E_2 \cdot H^T.$$

The above relation is reflexive and transitive and so defines a (pre-)ordering in $\mathcal{E}$. The code $\mathcal{C}$ is optimal for the channel in the sense that it minimizes the probability of error, iff there exists a set $\mathcal{E}' \subseteq \mathcal{E}$ composed of the 2^r most likely error patterns of the channel, such that:

$$\forall E_1, E_2 \in \mathcal{E}', \quad E_1 \not\preceq E_2 \quad \text{and} \quad E_2 \not\preceq E_1.$$

In other words, $\mathcal{C}$ is optimal iff $\mathcal{E}'$ is an antichain [40] of the (pre-)ordered set $(\mathcal{E}, \preceq)$. In order to make the relation $\preceq$ anti-symmetric, let $\mathcal{E}^*$ be the set $\mathcal{E}$ where

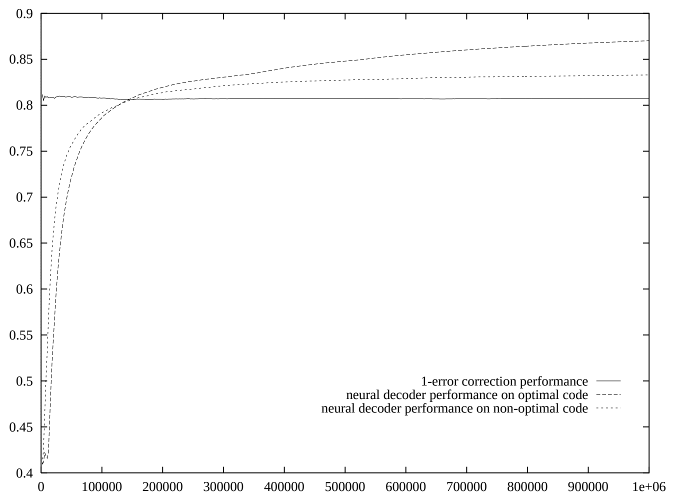

Figure 8: On the x-axes we represent the number of examples generated so far. Each curve is the graph of the ratio between x and the number of training examples for which the decoders give a correct answer.

we identify two error patterns having the same syndrome and probability of occurrence. The elements of $\mathcal{E}^*$ are classes, but we can identify each class with one of its elements. Now, a decoder for $\mathcal{C}$ is optimal iff it can correct the 2^r most likely maximal elements of the partially ordered set $(\mathcal{E}^*, \preceq)$.

In our example, the code $\mathcal{C}_1$ defined by the parity check matrix:

$$H_1 = \begin{pmatrix} 1000 & 1110 \\ 0100 & 1101 \\ 0010 & 1011 \\ 0001 & 0111 \end{pmatrix}$$

is optimal, whereas the code $\mathcal{C}_2$ defined by the parity check matrix:

$$H_2 = \begin{pmatrix} 1000 & 1110 \\ 0100 & 1100 \\ 0010 & 1011 \\ 0001 & 0111 \end{pmatrix}$$

is not optimal because $E_{10} \preceq E_5$ (see Table 1). Figure 8 shows the learning behavior of a 4 : 10 : 10 : 8 and a 4 : 14 : 14 : 8 neural decoder for $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. We generated error patterns according to the probability distribution defined by our channel (1), but we trained both the decoders on error patterns in the set:

$$\mathcal{E}_{training} = \{E = e_1 \dots e_8 \in \mathcal{E} :$$
$$w(E) \leq 2 \quad \text{and} \quad w(e_1 \dots e_4) = w(e_5 \dots e_8) = 1\},$$

$$H = \begin{pmatrix} 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111 \\ 1111\,1111\,1111\,1111\,0000\,0000\,0000\,0000 \\ 1111\,1111\,0000\,0000\,1111\,1111\,0000\,0000 \\ 1111\,0000\,1111\,0000\,1111\,0000\,1111\,0000 \\ 1100\,1100\,1100\,1100\,1100\,1100\,1100\,1100 \\ 1010\,1010\,1010\,1010\,1010\,1010\,1010\,1010 \\ 1111\,1111\,0000\,0000\,0000\,0000\,0000\,0000 \\ 1111\,0000\,1111\,0000\,0000\,0000\,0000\,0000 \\ 1100\,1100\,1100\,1100\,0000\,0000\,0000\,0000 \\ 1010\,1010\,1010\,1010\,0000\,0000\,0000\,0000 \\ 1111\,0000\,0000\,0000\,1111\,0000\,0000\,0000 \\ 1100\,1100\,0000\,0000\,1100\,1100\,0000\,0000 \\ 1010\,1010\,0000\,0000\,1010\,1010\,0000\,0000 \\ 1100\,0000\,1100\,0000\,1100\,0000\,1100\,0000 \\ 1010\,0000\,1010\,0000\,1010\,0000\,1010\,0000 \\ 1000\,1000\,1000\,1000\,1000\,1000\,1000\,1000 \end{pmatrix}.$$

Figure 9: Parity check matrix of the $(32, 16, 8)$ 2^{nd} order Reed-Muller code $\mathcal{C}$.

Where $w(X)$ indicates the weight of X. The neural decoder for $\mathcal{C}_1$ is optimal and the one for $\mathcal{C}_2$ is very close to optimal. It is interesting to note that when the neural decoder for $\mathcal{C}_2$ is faced with either the error pattern E_{10} or E_5, it gives as output E_5 because E_5 is about 5 times more probable than E_{10}. Figure 8 shows that the decoder for $\mathcal{C}_1$ performs better than the decoder for $\mathcal{C}_2$, which in turn performs better than a decoder performing 1 error correction.

7.3 $(32, 16, 8)$ Reed-Muller code

In this subsection, we discuss an example of a $16 : 96 : 96 : 32$ neural decoder for a $(32, 16, 8)$ 2^{nd} order Reed-Muller code $\mathcal{C}$. The parity check matrix of $\mathcal{C}$ is shown in Figure 9. It is well known that $\mathcal{C}$ is capable of correcting all error patterns of weight less than or equal to 3 and many error patterns of weight greater than 3. A priori, the set $\mathcal{E}'$ of correctable error patterns is a complex set. It might be very difficult to generate error patterns uniformly distributed in $\mathcal{E}'$. To approximate this uniform distribution, we generated error patterns according to the probability distribution of a BSC with probability of error $\epsilon = 0.3$, but we trained the decoder only on those error patterns of weight less than or equal to 3. Figure 10 shows the learning behavior of the neural decoder during the training phase on $4,000,000$ training examples. The training took approximately 90 hours to run. Table 2 shows the performance of the decoder. It can correct all error patterns of weight less than or equal to 2, and all but 11 error patterns of weight 3. It is also capable of correcting 47 error patterns of weight 4, although the decoder was not trained

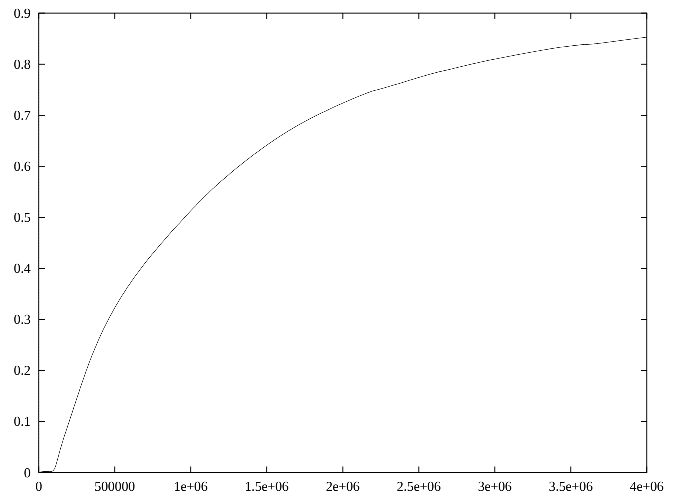

Figure 10: $16 : 96 : 96 : 32$ neural decoder learning behavior for a $(32, 16, 8)$ 2^{nd} order Reed-Muller code. On the x-axis we represent the number of training examples generated so far. The curve is the proportion of the training examples for which the decoder gives a correct answer.

on error patterns of weight greater than 3.

Table 2 shows that for three or more errors the neural net may not produce the correct error pattern. For example, if E is the correct error pattern of weight 3, then occasionally the net may produce E' which differs from E in exactly one bit. Now if E' is added to X', we have a word with one error, and by feeding $E' + X'$ back through the decoder, this single error will be corrected. Of course, we do not need to know that an incorrect error pattern was calculated for this double decoding trick to work. If the net correctly calculates E, then adding E to X' gives a word with no errors; feeding this to the decoder will produce an output of the 0 word since there was no error, and now $E + X' + 0$ will be the same word. This double decoding trick also often works when E has weight 4. From Table 2, when E has weight 4, then in about 26% of the cases E' differs from E in at most 3 bits. So decoding a second time will produce the correct sent word except for the 11 cases in which the decoder fails to correct three errors.

8 Analog Nets

The McCulloch-Pitts neuron is the prototypical digital device, but many feel that a “real” description of a neuron should be a continuous time, continuous state description, and that the digital neuron is some sort of an approximation to the real analog neuron. Recently, my colleague Rick Hangartner and I have been looking at the relationship between analog and digital neurons.

$w(E+E') \backslash w(E)$	0	1	2	3	4
0	1	32	496	4949	47
1	0	0	0	11	1618
2	0	0	0	0	3303
3	0	0	0	0	4330
4	0	0	0	0	5977
5	0	0	0	0	7146
6	0	0	0	0	7561
7	0	0	0	0	4641
8	0	0	0	0	1145
9	0	0	0	0	170
10	0	0	0	0	21
11	0	0	0	0	1
12	0	0	0	0	0
$\vdots$					
32	0	0	0	0	0
out of	1	32	496	4960	35960

Table 2: $16:96:96:32$ neural decoder performance for the $(32,16,8)$ 2^{nd} order Reed-Muller code $\mathcal{C}$. E' is the decoder response to the error pattern E. In the above table, the entry in row i and column j is the number of pairs (E,E') such that $w(E+E')=i$ and $w(E)=j$. The j-th entry in the last row is the number of error patterns E of weight j.

That analog devices behave digitally is the basis for a large part of electronics engineering and allows for the construction of electronic computers. It is part of the engineering folklore that when the gain is high enough, any circuit from a large class will eventually settle into one of two states, which can be used to represent Boolean 0 and 1. As far as we can tell, this theorem and its proof have never been published, but they probably appear in a now unobtainable MIT technical report of the 1950s.

We have worked on extending this analysis from a single analog neuron to a net of such neurons. While much more complicated phenomena are possible in such a net, we were able to show that under the assumption of high gain, if a consistent ternary (3-state) model of the net is possible, then many phenomena are not possible. In particular, neurons cannot get stuck except at their Boolean values. Our ternary analysis will allow one to decide whether and at which Boolean value the net will settle, or allow one to show that the net will continue oscillating.

We used our ternary analysis method to show that the pattern generator network of the sea slug *tritonia* must oscillate. This is the network observed by Getting [18]. We have also shown that an excitatory/inhibitory synapse proposed by Getting is unnecessary to make the network oscillate. While our ternary model can demonstrate oscillation, it cannot predict the detailed form of the oscillation. Using some information from Getting and some guesses for parameter values, we were able to simulate the networks and produce an oscillation that was reasonably similar to observed oscillations. Details of this work appear in [12] and [21].

Of course, we used a digital computer to run our simulations of analog neurons. But as we have mentioned, such a computer is based on using analog circuits which behave as Boolean gates. Thus, at a few levels removed we are using analog circuits to simulate analog nets. It would be more efficient to use analog circuits directly. Hangartner [20] has shown that it is possible to design an analog neuron which could be directly implemented in VLSI technology. Perhaps in the future such neurons will be fabricated, and it will be possible to study large scale neural nets by studying nets directly implemented in silicon. Such nets will give us the large scale parallelism found in actual brains, and the silicon implementation will allow speeds considerably greater than the speeds of the biological counterpart.

9 An Example: the Swim CPG of *Tritonia diomedea*

To amplify the preceeding, this section describes our analysis of the swim CPG of the sea slug *Tritonia diomedea* which has been extensively studied by Getting [17],[19]. In his work, Getting [17] found that each of *Tritonia*'s paired cerebral ganglia incorporated a swim CPG consisting of 6 neurons of three types: three dorsal swim interneurons (DSI), two ventral swim interneurons (VSIA, VSIB), and one central neuron (C2). All three DSIs are mutually coupled by excitatory synapses

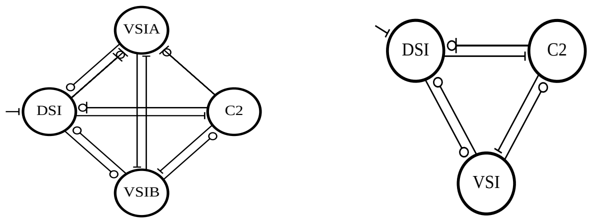

Figure 11: *Tritonia diomedea* swim central pattern generator (CPG): full network and simplified network.

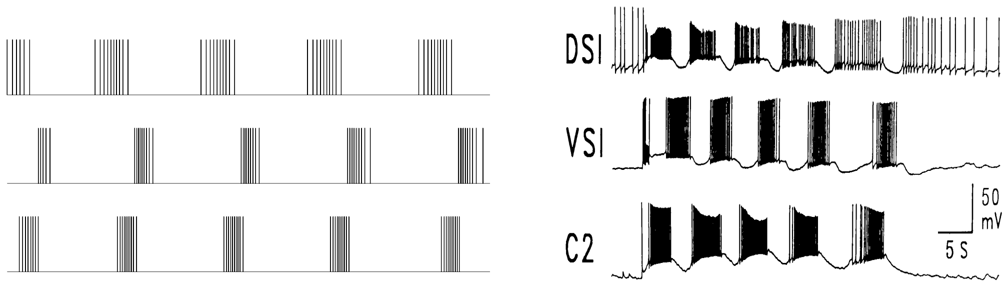

Figure 12: Simulated and actual *Tritonia* swim firing pattern.

while two of the three also seem to exhibit electrotonic coupling. Consequently, Getting represented the three DSIs by a single cell in the CPG schematic shown in Figure 11(a). Initially Getting identified only VSIA; he later identified VSIB and showed that although the action-potential sequences of VSIA and VSIB are distinct, they are highly correlated during rhythmic pattern generation. For qualitative modeling, therefore, they can be combined into a single VSI. One simplified schematic suggested by Getting [17] is shown in Figure 11(b).

The schematics use circles to represent inhibitory synapses and lines to represent excitatory synapses. Getting's study was surprising in that he identified synapses which generated multiple component post-synaptic potentials (PSPs) on varying time scales. Both he and Kleinfeld and Sompolinsky [27] claimed that these multi-action synapses are among the primary architectural elements responsible for pattern generation. The schematics represent these synapses by combinations of circles and lines in the order of the induced PSP; thus a single DSI action-potential generates an excitatory, then an inhibitory, and finally another excitatory PSP in VSIA. Significantly, the simplified schematic shown still includes an excitatory-inhibitory synapse between C2 and DSI in which the later inhibitory PSP dominates the initial excitatory PSP.

In contrast to this view, it is shown here that the net synaptic architecture represented by the interconnection matrices:

$$B = \begin{bmatrix} 0 & -b_{12} & -b_{13} \\ b_{21} & 0 & -b_{23} \\ -b_{31} & b_{32} & 0 \end{bmatrix} \qquad \vec{f} = \begin{bmatrix} f_1 \\ -f_2 \\ -f_3 \end{bmatrix}$$

where z_1, z_2, and z_3 are DSI, C2, and VSI respectively, qualitatively determines the firing pattern for the swim CPG if the C2-DSI synapse is regarded as solely inhibitory. (Although the negative entries in $\vec{f}$ could be interpreted as tonic inputs not appearing in the swim CPG, they also could have been absorbed into the activation function as a threshold shift. They have been explicitly included here to simplify the analysis.)

A ternary analysis of the model for the swim CPG which supports this claim is summarized next. The analysis shows that for a range of parameter values, the state transition mapping for the *Tritonia* swim CPG has no fixed points that a consistent ternary quantization maps into $\mathcal{B}^n$. Since the state transition mapping must have at least one fixed point, the analysis shows that the consistent ternary quantization maps any possible fixed point into regions such that the corresponding equilibria of the network are unstable. As a result, the net must oscillate in some fashion.

Choosing values for the various model parameters is the first and perhaps most difficult problem in the analysis. Obviously, the most desirable choices are parameters derived from extensive experimental observations. Unfortunately, this is a difficult task because our model represents an idealization of the biological network. Although Getting [19] faced this same problem and devised methods for estimating the parameters in his model, it is not clear how to convert his published parameter values into values for our model. Fortunately, the ternary analysis technique developed here does not require precisely measured parameters and one can chose parameters values which seem to be consistent with Getting's data.

Thus the analysis presented here uses the analysis tools summarized above to show that a range of parameter values exist such that the model oscillates in a fashion qualitatively similar to the swim CPG. The analysis begins with the assumption that only the most pessimistic bounds are possible for the state $\vec{z}$:

$$z_i^{[\ell]} = 0 \qquad z_i^{[u]} = \max_i \lim_{\zeta\to\infty} h_i \circ g_i(\zeta) = z^{[U]} \qquad i = 1,2,3$$

It is also necessary to make some assumptions on the entries of the B and $\vec{f}$. Various alternatives were investigated and although another set of assumptions best describes the biological network, the set used here has been chosen to better illustrate details of the analysis techniques:

- either C2 activity or VSI activity alone is sufficient to inhibit DSI, *i.e.*:

$$\begin{aligned} 0 &< f_1 \\ 0 &> -(1-\alpha)b_{12}z^{[U]} + f_1 \\ 0 &> -(1-\alpha)b_{13}z^{[U]} + f_1 \\ 0 &> -(1-\alpha)b_{12}z^{[U]} - (1-\alpha)b_{13}z^{[U]} + f_1 \end{aligned}$$

- VSI activity inhibits C2 even if C2 is excited by DSI, *i.e.*:

$$\begin{aligned} 0 &< (1-\alpha)b_{21}z^{[U]} - f_2 \\ 0 &> -(1-\alpha)b_{23}z^{[U]} - f_2 \\ 0 &> (1+\alpha)b_{21}z^{[U]} - (1-\alpha)b_{23}z^{[U]} - f_2 \end{aligned}$$

- C2 activity excites VSI even if VSI is inhibited by DSI, *i.e.*:

$$\begin{aligned} 0 &< (1-\alpha)b_{32}z^{[U]} - f_3 \\ 0 &< -(1+\alpha)b_{31}z^{[U]} + (1-\alpha)b_{32}z^{[U]} - f_3 \\ 0 &> -(1-\alpha)b_{31}z^{[U]} - f_3 \end{aligned}$$

These assumptions describe a network which has a consistent ternary quantization. Furthermore,

$$\vec{\phi}^{[0]} = \begin{bmatrix} -(1-\alpha)\min\{b_{12}, b_{13}\}z^{[U]} + f_1 \\ (1+\alpha)b_{21}z^{[U]} - (1-\alpha)b_{23}z^{[U]} - f_2 \\ -f_3 \end{bmatrix} \tag{2}$$

$$\vec{\phi}^{[1]} = \begin{bmatrix} f_1 \\ (1-\alpha)b_{21}z^{[U]} - f_2 \\ -(1+\alpha)b_{31}z^{[U]} + (1-\alpha)b_{32}z^{[U]} - f_3 \end{bmatrix} \tag{3}$$

so the network has a Boolean model for any $\xi > \xi_0$ where ξ_0 is the minimum value that satisfies

$$h_i \circ g_i(\xi_0 \phi_i^{[0]}) \;\leq\; \alpha z_i^{[U]} \qquad\qquad h_i \circ g_i(\xi_0 \phi_i^{[1]}) \;\geq\; (1-\alpha) z_i^{[U]}$$

for $i = 1, \ldots, n$. It should be clear from the assumptions that the Boolean model is

$$\Psi_1(\vec{\mathbf{z}}) = \overline{\mathbf{z}_2} \wedge \overline{\mathbf{z}_3} \qquad \Psi_2(\vec{\mathbf{z}}) = \mathbf{z}_1 \wedge \overline{\mathbf{z}_3} \qquad \Psi_3(\vec{\mathbf{z}}) = \mathbf{z}_2$$

To find the ternary model, first define the operators:

$$\lceil \mathbf{z}_i \rceil \;=\; \begin{cases} 0 & \text{if } \mathbf{z}_i \in \{0, \chi\} \\ 1 & \text{if } \mathbf{z}_i = 1 \end{cases} \qquad\qquad \lfloor \mathbf{z}_i \rfloor \;=\; \begin{cases} 0 & \text{if } \mathbf{z}_i = 0 \\ 1 & \text{if } \mathbf{z}_i \in \{\chi, 1\} \end{cases}$$

Then

$$\vec{\Phi}^{[0]} \;=\; \begin{bmatrix} -b_{12} z^{[U]} \lceil \mathbf{z}_2 \rceil - b_{13} z^{[U]} \lceil \mathbf{z}_3 \rceil + f_1 \\ b_{21} z^{[U]} \lfloor \mathbf{z}_1 \rfloor - b_{23} z^{[U]} \lceil \mathbf{z}_3 \rceil - f_2 \\ -b_{31} z^{[U]} \lceil \mathbf{z}_1 \rceil + b_{32} z^{[U]} \lfloor \mathbf{z}_2 \rfloor - f_3 \end{bmatrix} \tag{4}$$

$$\vec{\Phi}^{[1]} \;=\; \begin{bmatrix} -b_{12} z^{[U]} \lfloor \mathbf{z}_2 \rfloor - b_{13} z^{[U]} \lfloor \mathbf{z}_3 \rfloor + f_1 \\ b_{21} z^{[U]} \lceil \mathbf{z}_1 \rceil - b_{23} z^{[U]} \lfloor \mathbf{z}_3 \rfloor - f_2 \\ -b_{31} z^{[U]} \lfloor \mathbf{z}_1 \rfloor + b_{32} z^{[U]} \lceil \mathbf{z}_2 \rceil - f_3 \end{bmatrix} \tag{5}$$

Based on these assumptions, the ternary model $\widehat{\vec{\Psi}}(\vec{\mathbf{z}})$ has three potential fixed points:

$$\vec{\mathbf{z}}^{\infty} \;=\; \widehat{\vec{\Psi}}(\vec{\mathbf{z}}^{\infty}) \;\in\; \{\chi\chi\chi, \chi\chi 0, 1\chi\chi\}$$

The problem is to ascertain the actual nature of the corresponding equilibria and thereby deduce when the network must oscillate.

The only possible fixed points of the state transition mapping must be in the set:

$$\begin{aligned} \vec{z}^{\infty} \in \mathcal{Z} \;&=\; \mathcal{Z}_1 \cup \mathcal{Z}_2 \cup \mathcal{Z}_3 \\ &=\; [z^{[0]}, z^{[1]}] \times [z^{[0]}, z^{[1]}] \times [z^{[0]}, z^{[1]}] \;\cup\; [z^{[0]}, z^{[1]}] \times [z^{[0]}, z^{[1]}] \times [z^{[\ell]}, z^{[0]}] \\ &\quad\; \cup [z^{[1]}, z^{[u]}] \times [z^{[0]}, z^{[1]}] \times [z^{[0]}, z^{[1]}] \end{aligned}$$

Thus the compact region:

$$\begin{aligned} \mathcal{Y} \;&=\; \mathcal{Y}_1 \cup \mathcal{Y}_2 \cup \mathcal{Y}_3 \\ &=\; \left\{ \vec{y}^{\infty} \,\middle|\, \vec{h}(\vec{y}^{\infty}) \in \mathcal{Z}_1 \right\} \;\cup\; \left\{ \vec{y}^{\infty} \,\middle|\, \vec{h}(\vec{y}^{\infty}) \in \mathcal{Z}_2 \right\} \;\cup\; \left\{ \vec{y}^{\infty} \,\middle|\, \vec{h}(\vec{y}^{\infty}) \in \mathcal{Z}_3 \right\} \end{aligned}$$

contains all equilibria of the network. Furthermore, $\mathcal{Y}$ must contain at least one equilibrium. Since none of the potential stable states are binary, it can be inferred that the network must either settle to a non-binary stable equilibrium or oscillate.

The stability of the equilibria in $\mathcal{Y}$ can be ascertained by examining the roots of the characteristic equation:

$$|J(\vec{y}^{\infty},\xi) - \lambda I| \;=\; -\lambda^3 + \lambda(j_{13}j_{31} - j_{23}j_{32} - j_{12}j_{21}) - (j_{12}j_{23}j_{31} + j_{13}j_{32}j_{21}) \quad (6)$$

in each of the connected, compact regions $\mathcal{Y}_1$, $\mathcal{Y}_2$, and $\mathcal{Y}_3$ separately. In region $\mathcal{Y}_1$, the characteristic equation clearly has a non-zero root so that $J(\vec{y}^{\infty},\xi)$ is not nilpotent everywhere in $\mathcal{Y}_1$. Therefore, there exists a ξ_1 such that any equilibrium $\vec{y}^{\infty} \in \mathcal{Y}_1$ is unstable.

The region $\mathcal{Y}_2$ requires only slightly more analysis. Let:

$$\mathcal{V}_1 \;=\; \left\{\vec{y} \in \mathcal{Y}_2 \,\middle|\, h_3(y_3) = z_3^{[\ell]}\right\}$$

and

$$\mathcal{V}_2 \;=\; \left\{\vec{y} \in \mathcal{Y}_2 \,\middle|\, z_3^{[\ell]} < h_3(y_3) \leq z_3^{[0]}\right\}$$

By definition, the third row of $J(\vec{y}^{\infty},\xi)$ is zero ($j_{i3} = 0$ for $i = 1,2,3$) everywhere in $\mathcal{V}_1$. As a result, the characteristic equation reduces to:

$$|J(\vec{y}^{\infty},\xi) - \lambda I| \;=\; -\lambda^3 - \lambda j_{23}j_{32} \quad (7)$$

which has a non-zero root, so $J(\vec{y}^{\infty},\xi)$ is not nilpotent everywhere in $\mathcal{V}_1$. The reasoning for $\mathcal{Y}_1$ shows that $J(\vec{y}^{\infty},\xi)$ is not nilpotent everywhere in $\mathcal{V}_2$. Therefore, there exists an ξ_2 such that any equilibrium:

$$\vec{y}^{\infty} \;\in\; \mathcal{V}_1 \cup \mathcal{V}_2 \;=\; \mathcal{Y}_1$$

is unstable.

A similar analysis shows that there exists a $\xi_3 > 0$ such that any equilibrium $\vec{y}^{\infty} \in \mathcal{Y}_3$ is unstable if $\xi > \xi_3$. Combining the results for all three regions leads to the result that any equilibrium $\vec{y}^{\infty} \in \mathcal{Y}$ must be unstable if:

$$\xi \;>\; \xi_0 \;>\; \max\{\xi_1, \xi_2, \xi_3\}$$

Of course, any network which has no asymptotically stable equilibria for any $\xi > \xi_0$ may have no asymptotically stable equilibrium for much smaller values of ξ.

Consequently, the class of network models for the *Tritonia* swim CPG includes instances which have no asymptotically stable equilibria. Those instances must exhibit either a stable periodic or a chaotic orbit, an interesting possibility currently under investigation. Figure 12 presents simulation results for one instance of an actual stochastic spiking network model with parameters chosen to meet the assumptions stated above. The action-potential pattern in the model duplicates the essential structure of the pattern generated by the *Tritonia* swim CPG as reported by Getting (Figure 12).

10 Conclusion

Fifty years after the pioneering work of McCulloch and Pitts, the study of neural nets is alive and active. In this paper, I have discussed some of the work that is of current interest to me and my co-workers. I would, perhaps, be remiss if I failed to mention some of the current hype about neural nets.

Can neural nets quickly solve NP-complete problems? No. A look at the proposed nets will show that the question of whether the net will converge, or where the net will converge to, are as difficult as the original NP-complete problem. This does not prevent the neural net from giving an approximate solution to a hard optimization problem, but no one has yet proven any approximation bounds. Hard problems are only hard in the worst case, so there may be many easy instances of a hard problem. Nothing prevents a neural net from solving these easy instances quickly.

Can analog neural nets compute things not computable a Turing machine? Yes. But any analog device with infinite precision has more computational power than a Turing machine, so a neural net with unlimited precision should be a very powerful device. But practically all devices are constructed with limited precision, and these limited precision devices have no more power than a Turing machine.

Can neural nets compute faster than other parallel models? No. Neural nets are in fact equivalent to the usual parallel models. The only difference that can occur is if the neural net has infinite precision which as mentioned above is highly unlikely.

Does learning in neural nets make programming unnecessary? No. As we saw in the discussion of learning, learning rules must be devised, and it seems that different learning tasks will require different learning rules. Further, the kind of net to use for a particular task will be an important decision. In our decoding example, some network topologies did not lead to good decoders, while other topologies did. Neural nets will not replace programmers, but give programmers another paradigm in which to program.

In spite of the hype, I believe that neural nets will be useful both as biological models and as programming paradigms.

Finally, according to an often-told tale, there was a golden age of neural nets which suddenly ended in 1970. Depending on the version of the tale, the golden age ended because of the Vietnam war, or Minsky and Papert's book on perceptions [32], or cuts in funding, or the rise of artificial intelligence. But I hope that the reader of this paper and the rest of this volume will see that the death of Warren McCulloch had a most profound effect on the field. We miss him as a brilliant scientist, as a warm human being, and as the greatest story-teller of our age.

11 Acknowledgment

I would like to thank the many people I have collaborated with over the years. Without their assistance and encouragement, I would have accomplished very little. In particular, this paper could not have been finished without the assistance of Rajeev Pandey, Luca Tallini, and Rick Hangartner. Thanks again.

References

[1] N. Alon, A.K. Dewdney, and T.J. Ott. Efficient simulation of finite automata by neural nets. *J. ACM*, 38:495–514, 1991.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[3] E. R. Berlekamp, R. J. McElice, and H. C. A. vanTilborg. On the inherent intractability of certain coding problems. *IEEE Trans. Inform. Theory*, pages 384–386, May 1978.

[4] A. Blum and R. L. Rivest. Training a 3-node neural network is NP-complete. In *Advances in Neural Information Processing Systems*, pages 494–501. Morgan Kaufmann, San Mateo, CA, 1989.

[5] E. R. Caianiello. Outline of a theory of thought-processes and thinking machines. *Journal of Theoretical Biology*, 1:204–235, 1961.

[6] E. R. Caianiello. Some remarks on the tensorial linearization of general and linearly separable Boolean functions. *Kybernetik*, 12:90–93, 1973.

[7] E. R. Caianiello. Problems connected with neuronic equations. In *Biomathematics and Related Computational Problems.* Kluwer, Netherlands, 1988.

[8] P. Cull. Linear analysis of switching nets. *Kybernetik*, 9:31–39, 1971.

[9] P. Cull. A matrix algebra for neural nets. In G. Klir, editor, *Applied General Systems Research*, pages 563–573. Plenum, New York, 1978.

[10] P. Cull. Dynamics of neural nets. *Trends in Biological Cybernetics*, 1:331–349, 1991.

[11] P. Cull. Dynamics of random neural nets. In *Cellular Automata and Cooperative Systems.* Kluwer, Netherlands, 1993.

[12] P. Cull and R. Hangartner. Oscillations in a central pattern generator. In *Cybernetics and Systems '94.* World Scientific, Singapore, 1994.

[13] R. Fagin. Probabilities on finite models. *J. Symbolic Logic*, 41:50–58, 1976.

[14] W. Freeman. If chaos in brains is for real, what is it for? *Proceedings of the Pacific Division, AAAS*, 7:28–29, 1988.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman, New York, 1979.

[16] A. E. Gelfand. A behavioral summary for completely random nets. *Bulletin of Mathematical Biology*, 44:309–320, 1982.

[17] P. A. Getting. Mechanisms of Pattern Generation Underlying Swimming In *Tritonia.* I. Neuronal Network Formed by Monosynaptic Connections. *Journal of Neurophysiology*, 46(1):65–79, 1981.

[18] P. A. Getting. Mechanisms of Pattern Generation Underlying Swimming in *Tritonia* III. *J. Neurophysiology*, 49:1036–1050, 1983.

[19] P. A. Getting. Reconstruction of Small Neural Networks. In C. Koch and I. Segev, editors, *Methods in Neuronal Modeling*, chapter 6, pages 171–194. MIT Press, Cambridge, 1989.

[20] R. Hangartner. An analog neuromine suitable for VLSI implementation. Technical Report 90–20–1, Department of Computer Science, Oregon State University, 1990.

[21] R. Hangartner and P. Cull. A ternary logic model for recurrent neuromime networks with delay. *Biological Cybernetics*, 73:177–188, 1995.

[22] A. V. Hill. Excitation and accommodation in nerve. *Proc. Roy. Soc. London, Series B*, 119:305, 1936.

[23] A. S. Householder and H. D. Landahl. *Mathematical biophysics of the central nervous system.* Principia Press, Bloomington, Indiana, 1945.

[24] S. A. Kauffman. Metabolic stability and epigenesis in randomly connected genetic nets. *Journal of Theoretical Biology*, 22:437–467, 1969.

[25] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, 1994.

[26] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, New Jersey, 1956.

[27] D. Kleinfeld and H. Sompolinsky. Associative Network Models for Central Pattern Generators. In C. Koch and I. Segev, editors, *Methods in Neuronal Modeling*, chapter 6, pages 195–246. MIT Press, Cambridge, 1989.

[28] O.B. Lupanov. On a method of circuit synthesis. *Izvestia VUZ (Radiofizika)*, 1:120–140, 1958.

[29] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[30] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[31] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice–Hall, Englewood Cliffs, New Jersey, 1967.

[32] M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.

[33] J. Von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[34] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[35] L. Pitt and L. Valient. Computational limitations on learning from examples. *Journal of the ACM*, 35:965–984, 1988.

[36] N. Rashevsky. Outline of a physico–mathematical theory of excitation and inhibition. *Protoplasma*, 20:42, 1933.

[37] N. Rashevsky. *Mathematical Biophysics*. Dover, New York, 1960.

[38] J. Reif. On threshold circuits and polynomial computations. In *2nd Conference on Structure in Complexity Theory*, pages 118–125, 1987.

[39] F. Rosenblatt. *Principles of Neurodynamics*. Spartan, New York, 1962.

[40] G. C. Rota. *Studies in Combinatorics*, volume 17 of *Studies in Mathematics*. The Mathematical Association of America, 1978.

[41] H. Rubin and R. Sitgreaves. Probability distributions related to random transformations on a finite set. Technical Report 19A, Applied Math. and Stat. Lab., Stanford University, 1954.

[42] C. Shannon. The synthesis of two-terminal switching circuits. *Bell Syst. Techn. J.*, 28:59–98, 1949.

[43] L. Tallini and P. Cull. Neural nets for decoding error-correcting codes. In *NORTHCON 95*, pages 89–94. IEEE Press, 1995.

[44] L. G. Valient. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[45] I. Wegener. *The Complexity of Boolean Functions.* Wiley, Chichester, 1987.