AN ABSTRACT OF THE THESIS OF

Martin S. Perlot for the degree of Master of Science in Mathematics presented on May 15, 1989.

Title: The Suppression of Learning at the Hidden Units of Neural Networks

Abstract approved: Redacted for privacy

Robert Burton

Under certain conditions, a neural network may be trained to perform a specific task by altering the weights of only a portion of the synapses. Specifically, it has been noted that certain three layer feed-forward networks may be trained to certain tasks by adjusting only the synapses to the output unit. This paper investigates the conditions under which this hobbling of the process may be possible.

The investigation assumes that the existence of a set of weights which perform a task with low error implies that the task is learnable. Thus an algorithm is developed which attempts to find such a set, given the weights at the hidden units as fixed. Success of the method is equated with the ability to suppress learning at the hidden units.

The result is a classification of tasks for which suppression is possible. In general, classification problems require learning at the hidden units, but approximation of simple continuous functions does not.

The Suppression of Learning at the Hidden Units of Neural Networks

by

Martin Perlot

A THESIS

submitted to

Oregon State University

in partial fulfillment of the requirements for the degree of

Master of Science

Completed May 15, 1989

Commencement June 9, 1990

APPROVED:

Redacted for privacy

Professor of Mathematics in charge of major

Redacted for privacy

Head of Department of Mathematics

Redacted for privacy

Dean of Graduate School

Date thesis is presented May 15, 1989

TABLE OF CONTENTS

	Page
I. INTRODUCTION	1
II. DESCRIPTION	17
III. FIRST RESULT	19
IV. SECOND RESULT	27
V. CONCLUSIONS	32
BIBLIOGRAPHY	40

LIST OF FIGURES

		Page
Figure 1.	A neural network	38
Figure 2.	A 1-2-1 neural network	39
Figure 3.	A 2-3-1 neural network	39

THE SUPPRESION OF LEARNING AT THE HIDDEN UNITS OF NEURAL NETWORKS

I. INTRODUCTION

In recent years interest in neural networks has exploded, and the subject has been studied avidly by psychologists, computer scientists, engineers, and even mathematicians. The reason for this is straight-forward: neural networks utilyze a philosophically satisfying technique to successfully solve a variety of difficult problems. They have theoretical, philosophical, and practical importance.

UNDERLYING CONCEPTS

A neural network is an example of a parallel processing system. Whereas most computing systems used today utilyze a single central processor, a parallel processing system uses several processors working simultaneously on separate parts of a problem. In a neural network, each of these processors, called a neuron or node, is exceedingly simple - capable only of a single kind of calculation. These nodes are connected by synapses to eachother according to some architecture. Each synapse is assigned a weight, and directs the output of one node to the input of a second node. A node may be linked to other nodes, or to itself, both as input and as output, and the weights represent the importance and nature (positive or negative

enhancing or inhibiting) of the connection. Figure 1 illustrates a simple network.

Each node performs the same kind of calculation. Thus the distinction between networks is the architecture - the way the nodes are connected - and the values of the weights. To build a network to perform a specific task, these two parameters must be specifically set. This corresponds to programming a conventional computer, but the method is drastically different.

First, the architecture must be chosen. The most important considerations are that the network be large enough and deep enough to provide the complexity demanded by the task. There are techniques to help optimize the size and structure of networks, but they have important limitations, as we'll see later. The power of a network to solve a problem is related to the size and depth in an ill-understood way, and it is crucial that the network be large enough for the appointed task. It is also important that the network not be too large [1].

Second, the synapses must be assigned weights. This is where neural networks diverge most strongly from conventional computers, even parallel processing machines. For the weights are not pre-determined, but are arbitrarily set, then adjusted during a training period until the network performs the task adequately. In this manner a network can be trained to perform a task which may be intractable to program conventionally. Indeed, networks may be trained to classify pattern set that may be too complicated or vague for programmers to provide a useful heuristic. For example, Soviet researchers used a neural network to successfully predict the outcome of the 1988 American presidential election, using the outcomes of previous elections to train the system [2]. Such a network is merely a toy, but it illustrates that the programmer does not need to provide an algorithm for the task, merely a reasonably large set of inputs to train with.

Neural networks have been used primarily for pattern recognition thus far. Since they don't require an explicit method or heuristic to distinguish patterns - just a training set of identified patterns, they can easily be created to solve a variety of problems. Furthermore, they deal with ambiguous or non-ideal inputs effectively. They are rarely perfectly accurate, but their performance in usually adequate, and sometimes spectacularly so.

DEFINITIONS

A neural network is a set of nodes and synapses. A node is a simple calculator - in most networks it sums the values of its input synapses, evaluates a transfer function at the sum, and the result is its output. A synapse transmits the output of one node to become the input of another node, after multiplying by a weight. A unit is a node together with the synapses which provide its input.

The transfer function is usually chosen to be an increasing function with range [0,1], or to be a Heaviside function - i.e. $f_a(x)=1$ if $x \geq a$, 0 if $x<a$. The value of the threshold parameter, a, may be fixed, or it may be adjusted in a manner similar to the weights. If a continuous function is chosen, it is usually the logistic function, $f_\theta(x)=\frac{1}{1+e^{\theta-x}}$, where θ is again either fixed (usually $\theta=1$) or adjustable during training. One advantage this function has is that its derivative is easily calculated from the relation $f'(x)=f(x)(1-f(x))$. All of the experimental work done in this paper used this function, with $\theta=0$.

Every network requires two special kinds of nodes. An output node generally provides no information to any other node. It is observed by the user, and provides the results of the networks attempt to perform the task. A network may have several such nodes. An input node receives no information from any

other node, and it performs no calculations. Instead it is assigned a value by the user, the input for the task. Again, there may be several input nodes. Performing a task once thus requires the user to set the values of the input nodes, allow the system to calculate, and then observe the values of the output nodes. Nodes which are neither input nor output nodes are called hidden nodes.

In some situations, another specialized node is introduced. A global node is a node which always keeps the same value, usually negative one. The use of a global node is formally equivalent to the threshold parameters in the transfer functions, and may be desirable in certain instances. The parameter is replaced by a synapse of the same weight from a global node with value one. The synapse weight may then be adjusted using the same algorithm as the other synapse weights.

The synapses and nodes form a directed graph. If this graph has no loops, i.e. if the output of a node does not affect its own input directly or indirectly, the network is called feed-forward. A feed-forward network is easily modeled on an ordinary computer, since the values of each node may be calculated sequentially, starting at the nodes which receive input only from the input nodes, and finishing with the output nodes. Furthermore, there are well-established methods for training these networks, and feed-forward networks are capable of a number of important tasks. Thus, although non-feed-forward networks are an exciting concept, this paper will consider only feed-forward networks, which have been the object of most of the recent research and applications.

If a feed-forward network is arranged in layers, with the output of the nodes of one layer becoming the input of the next layer, it is often denoted by listing the number of nodes in each layer. It is assumed that the nodes in

consecutive layers are completely connected (every node in a layer is connected by a synapse to every node in the next layer), and nodes in non-consecutive layers are not connected. The input nodes are considered the first layer, and the output nodes are the final layer. Thus a 2-5-3-1 network has two input nodes, eight hidden units divided into two layers, and one output unit.

The inputs and outputs of a network may be treated as vectors, and the task of a network is to approximate a specific vector function. In a classification problem, the function is discrete-valued, and the utility of the network is given by the accuracy with which it assigns the proper values to the input vectors - i.e. the accuracy with which it correctly partitions the input vectors into classes. In a general problem, the utility of the network is measured by the norm of the difference between the desired function and the actual output. For ease of calculation, and in analogy to physical and statistical measurements, the absolute error is usually squared. There is a strong theoretical difference between the two methods, and although the latter is used almost universally, even for classification problems, it does not guarantee best performance as measured by the former method.

As mentioned before, neural networks are not programmed, but instead are trained. This is done using a learning rule to adjust the weights of each synapse after entering an input vector and comparing the actual output to the desired output. The input vector is chosen from a training set, usually randomly, and the process is repeated until the system is either performing within acceptable tolerances, or it becomes apparent that the system will not learn the task.

If the training set is finite, it is common to use each element as input once until the set is exhausted. The process is then repeated, and each cycle is called

an epoch. Under this nomenclature [3], each input vector is called an environment, and the corresponding output is termed a response. One of the fundamental problems of neural network training is the distinction between learning the correct response to a given environment, in contrast to all the environments which it will be expected to respond to. For example, if the training set is too small, or has insufficient variation, after training the resulting network may perform poorly on environments which were poorly represented. Conversely, if the training set contains "outliers", rare environments requiring unusual responses, the network may be less effective for more common environments. And importantly, though application of the learning rule will improve performance for a specific environment, it may not improve overall performance. This phenomenon of "dynamic mislearning" is important to understanding many of the limitations of neural networks.

The most common learning rule is back-propagation of error, [1,4] which is a minor misnomer, since all common learning rules utilyze a recursive process of weight adjustment, starting at the output units and propagating backwards, in response to the error. More precise would be gradient descent. The theory is based on a common method of optimization. If the error of the system's response to a given input is treated as a function of the synapse weights, we can consider the surface thus defined. The current weights define a point on that surface, and the gradient at that point indicates how the weights may be adjusted to yield a smaller error. Thus the weights are adjusted by a small amount in the appropriate direction, and the process is repeated for a new input vector.

The error function used is almost always the square of the difference, since it is continuous and differentiable, thus easy to calculate. Thus, consider the

following equations and the resulting formulae for back-propagation.

Define

e = error,

o_i = value of output node i,

t_i = desired value of output node i,

v_i = value of (non-output) node i,

z_i^j = weight of synapse from node i to node j,

w_i^j = weight of synapse from node i to output node j,

δv_i, δo_i = delta value at node v_i, o_i,

$f(x)$ = transfer function.

Then

(1.1) $$e = \sum_i (t_i - o_i)^2$$ summation over all output nodes.

(1.2) $$o_j = f(\Sigma v_i w_j^i)$$ summation over all nodes which output to o_j.

(1.3) $$v_j = f(\Sigma v_i z_j^i)$$ summation over all nodes which output to v_j.

We wish to calculate $\frac{\partial e}{\partial w_i^j}$ and $\frac{\partial e}{\partial z_i^j}$ for every synapse. Thus note

(1.4) $$\frac{\partial e}{\partial o_j} = -2(t_j - o_j)$$

(1.5) $$\frac{\partial e}{\partial w_i^j} = \frac{\partial e}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_i^j} = -2(t_j - o_j) \cdot f'(\Sigma v_i w_i^j) \cdot v_i$$

Define

(1.6) $$\delta o_j = -2(t_j - o_j) \cdot f'(\Sigma v_i w_i^j)$$

So

(1.7) $$\frac{\partial e}{\partial w_i^j} = \delta o_j \cdot v_i$$

To find $\frac{\partial e}{\partial z_i^j}$, define

(1.8) $$\delta v_j = [\Sigma(\delta v_i \cdot z_j^i) + \Sigma(\delta o_i \cdot w_j^i)] \cdot f'(\Sigma v_k z_k^j)$$ where the first summations are over all nodes v_j outputs to, and the last summation is over the nodes v_j receives input from. Note that every δv_i and δo_i for nodes receiving input from v_j must be calculated first. Thus the term back-propagation - calculation begins with the output units, then proceeds backwards through the network. Thus without modification, the method only works for feed-forward networks.

Finally, $\frac{\partial e}{\partial z_i^j} = \delta v_j \cdot v_i$. This may be easily verified for small networks using the chain rule. For large networks, the additivity of the derivative preserves the awkward summations.

As an example, consider the "1-2-1" network illustrated in Figure 2. Assume it has the logistic transfer function $f(x)=(1+e^{-x})^{-1}$, and assume that the desired task is simply for the output to echo the input, i.e. $t_1 = v_1$. Note

(1.9) $$\frac{\partial e}{\partial w_2^1} = -2(t_1-o_1) \cdot f'(\sum_{i=2}^{3} v_i w_i^1) \cdot v_2$$

(1.10) $$\frac{\partial e}{\partial z_1^2} = -2(t_1-o_1) \cdot f'(\sum_{i=2}^{3} v_i w_i^1) \cdot w_2^1 \cdot f'(v_1 \cdot z_1^2) \cdot v_1$$

and similarly for w_3^1 and z_1^3.

To illustrate the learning rule, suppose $z_1^2 = z_1^3 = 1$, $w_2^1 = 1$, and $w_3^1 = -2$. If the input v_1 is 0.5, then $v_2 = v_3 = f(1 \cdot 0.5) = 0.622$. $o_1 = f(0.622 \cdot 1 + 0.622 \cdot -2) = f(-0.622) = 0.349$. The error, $e = (0.5 - 0.349)^2 = 0.0229$.

Back-propagating yields $\delta o_1 = -2(0.151) \cdot f'(-0.622) = -0.0686$, $\delta v_2 = (-0.0686 \cdot 1) \cdot f'(0.5) = -0.0161$, $\delta v_3 = (-0.0686 \cdot -2) \cdot f'(0.5) = -0.0322$. Thus $\frac{\partial e}{\partial w_2^1} = -0.0686 \cdot 0.622 = -0.0427$, $\frac{\partial e}{\partial w_3^1} = -0.0427$, $\frac{\partial e}{\partial z_1^2} = -0.0161 \cdot 0.5 = -0.0081$, $\frac{\partial e}{\partial z_1^3} = -0.0322 \cdot 0.5 = -0.0161$.

Each weight should now be adjusted in proportion to its corresponding partial derivative, e.g. $w_2^1 = w_2^1 - \epsilon \cdot \frac{\partial e}{\partial w_2^1}$, where ϵ is some pre-determined small positive number.

A second learning rule, useful for classification problems, is the <u>desired states</u> method [3]. It also propagates backwards, but rather than utilyze gradient information, each unit has attached to it three pieces of information: its actual value, desired value, and criticality. The desired state is either 1 or 0, thus the system is only used for classification problems, where the desired responses are also combinations of 1's and 0's. This does not limit the method to systems using Heaviside transfer functions, however. In fact, it is independent of the transfer function chosen.

For the output units, the criticality is a constant which is specified when the task is defined. The criticality is chosen in [0,1], and represents the importance of that unit to the performance of the system. For hidden units, the criticality represents the outcome of a "voting" process: a high criticality implies that adjustment of that unit will have the the desired effect on most of the units to which it outputs.

The desired states and criticality of each unit are calculated as follows. For output units, the desired state is simply the target, or desired output, and the criticality is given in advance. For hidden units, if the actual state, desired state, and criticality is calculated for all units to which it outputs, the values are given by the following formulae:

Define

v_i = value of node i.

d_i = desired state of node i.

c_i = criticality of node i.

w^i_j = weight of synapse from node j to node i. No distinction will be made between hidden and output units.

$s(x) = 1$ if $x=1$, -1 if $x=0$

$u(x) = 1$ if $x>0$, 0 if $x\leq 0$

Then

$$d_i = u\left(\sum_j w^j_i \cdot s(d_j) \cdot c_j\right) \tag{1.11}$$

$$c_i = \frac{\left|\sum_j w^i_j \cdot s(d_j) \cdot c_j\right|}{\sum_j \left|w^i_j \cdot s(d_j) \cdot c_j\right|} \tag{1.12}$$

The summations are over the nodes to which node i sends output. Note that the criticality is 1 if all of the nodes "desire" the same adjustment (positive or negative), and decreases according to how much the "desires" cancel eachother.

Once these values have been calculated for each unit, the synapse weights

are adjusted in proportion to the error (difference between actual and desired values), criticality of the receiving node, and the value of the sending node. Thus

$$w_i^j = w_i^j + \epsilon \cdot (d_j - v_j) \cdot c_j \cdot v_i \tag{1.13}$$

with ϵ being a small positive number. Often the changes are not made immediately, but are accumulated and adjustments are made at the end of an epoch.

As an example, consider the 2-3-2 network illustrated in Figure 3. Let the transfer function be the Heaviside function with threshold 0.5, the task be to evaluate the "AND" function on the inputs, i.e. $d_6 = v_1 \cdot v_2$, with $v_1, v_2 \in \{0,1\}$, and the criticality $c_6 = 1$. Suppose $w_1^3 = w_2^5 = w_3^6 = w_4^6 = w_5^6 = 1$, $w_1^4 = w_2^4 = -1$. We shall calculate the appropriate adjustments resulting from entering $v_1 = 1$, $v_2 = 0$.

First, note $v_3 = 1$, $v_4 = 0$, $v_5 = 0$, $v_6 = 1$. Since $d_6 = 1 \cdot 0 = 0$, the system must be adjusted. Starting with $d_6 = 0$, $c_6 = 1$, calculate $d_5 = u(w_5^6 \cdot s(d_6) \cdot c_6) = u(1 \cdot -1 \cdot 1) = 0$. $c_5 = |1 \cdot -1 \cdot 1| / |1 \cdot -1 \cdot 1| = 1$. Similarly $d_4 = u(1 \cdot -1 \cdot 1) = 0$, $c_4 = 1$, $d_3 = 0$, and $c_3 = 1$.

Thus, if $\epsilon = 0.1$, for example, $w_1^3 = 1 + 0.1 \cdot (0-1) \cdot 1 \cdot 1 = 1 - 0.1 = 0.9$. Since $d_4 = v_4$ and $d_5 = v_5$, the weights at those nodes are not adjusted. Since $v_4 = 0$ and $v_5 = 0$, the synapses from those nodes are not adjusted. The only other adjustment is $w_3^6 = 1 + 0.1 \cdot (0-1) \cdot 1 \cdot 1 = 1 - 0.1 = 0.9$.

Note that the method never refers to the transfer function. It merely assumes that the function is non-decreasing with range in [0,1].

LIMITATIONS OF NEURAL NETWORKS

Although neural networks have demonstrable practical value, they also have important limits, both theoretical and practical. These limitations stem from the method of learning. A network is likely to be theoretically capable of a task, but in practice, such a network may be intractable to train. Difficulties may arise in a number of ways, and a network designer must often devise schemes to overcome them.

For example, the method of error calculation may be inappropriate. Gradient descent back-propagation utilyzes a least-squares definition of minimum error. In a classification problem, the set of weights which provides the minimum error may not necessarily correctly classify each member of the training set, even when there is a set of weights which can perform the task [5]. This may happen when there are unusual members in the training set, or the training set has an inappropriate distribution. In general, there is no simple fix for this problem. The situation is analogous to the presence of outliers in other optimization problems.

Another deficiency can strike any learning rule. The learning corresponding to a specific environment may not improve the system's performance overall. This phenomenon shall be refered to as dynamic mislearning. Research is being conducted which indicates that for suitably small adjustments, dynamic mislearning can be avoided, but the practicality of the result has not yet been demonstrated, nor has a rigorous proof been given.

These first two limitations are mainly of theoretical importance. They underscore the difficulty of establishing any general theorems of practical value, but they detract little from the utility of neural networks, since they arise only in

unusual cases. But there are other deficiencies which are more practical, which depend less on mathematical theory.

Neural networks, regardless of learning rule, may fail to learn a task which theoretically they should be able to perform. For any task that networks are theoretically capable of, there is some minimal network that can perform that task arbitrarily well. Larger networks generally perform the task only marginally better. But often the small network will not be able to learn the task starting with randomly selected weights. Thus a larger network must be trained for the job. And although it is often possible to determine the size of the minimum network capable of the task, there is no known method for determining the smallest network which will reliably learn the task.

So for most applications, networks significantly larger than necessary are used. After training, a unit may be removed if it has little effect on the output, or it is duplicated by another unit. An entire layer of units may be removed if it is similarly not important to the performance of the system. But although well-defined rules exist to determine how a network may be "pruned" and retrained [6], the method is severely limited. For example, a network may seem unprunable - that is, every node may appear essential to the system - even if it much larger than necessary. Units, or even groups of units, may be removable or replaceable by smaller groups without being apparent to the pruner. Thus the current heuristics for pruning are incomplete, and have only been tested on a limited range of tasks.

Furthermore, pruning is inconvenient and complicated to automate. Pruning can degrade a system, and occasionally render it incapable of the task. The results are unpredictable, and often unsatisfying.

Networks may fail to learn a task they are theoretically capable of, regardless of size. Units often tend to "couple", to act as a pair, their values being highly correlated, either positively or negatively. This has the effect of making the network one unit smaller in function. In the worst case, two units may have identical weights, creating a symmetry which robs the system of computing power. Coupling may affect entire groups of units, and may not be immediately obvious to the designer.

Methods like gradient descent often suffer from local minima. A set of weights may perform a task better than any nearby set of weights, but still be unacceptable. An effective way of bypassing such local minima and obtaining a useful set of weights is simulated annealing. At each iteration, in addition to the adjustment made by the learning rule, a small random term is added. In ordinary simulated annealing the random term is proportional to a factor which decreases exponentially over time, thus simulating annealing in physical systems. In TINA, or Time-Invariant Noise Algorithm, the random term is proprtional to the error. Other schemes are also used, with similar effects [7].

One of the most debilitating problems encountered with neural networks is slow learning. Even within the basin of attraction of a good minimum, learning may be slowed by dynamic mislearning, or by local flat regions. In a flat region, the adjustments calculated by the learning rule are correspondingly small, and thus learning is slow. The former case occurs when the weight adjustments are poorly correlated to the adjustments which would improve overall performance, and may be nearly the opposite the adjustments for another environment. Thus there may be cancellation and competition between environments, slowing the learning process. Annealing is of doubtful utility at this point, and may even

hinder learning, as the weights may "bounce" out of a good minimum. Another technique is the inclusion of a momentum term. A constant, α, in (0,1) is chosen, and to each weight adjustment is added α times the previous adjustment. Thus the weights will move more smoothly across the error surface, and presumably settle into a minimum more reliably.

These methods apply similarly to any learning rule. However, the choice of learning rule may introduce other sources of slow learning. For example, gradient descent suffers when used with "deep" systems - systems with several layers of hidden units, each layer having only a few units. The problem arises from the calculation of δv_i as $\Sigma \delta v_j \cdot z_i^j$ multiplied by $f'(\Sigma v_k z_k^i)$. This last factor may be very small, for the logistic function is is never greater than 1/4. Thus if the summation factor is not large, δv_i will be small relative to the δv_j it is derived from, and adjustments at that unit will be small. Fortunately, systems with many layers but few nodes have little application.

In summary, there are a number of problems neural networks may suffer from, both theoretically and in practice. They stem from the method of calculating error, the tendency for units to become correlated, and dynamic mislearning. There are a number of techniques used to bypass these problems, but there are few general principles to guide the designer or analyst.

MATHEMATICAL RESULTS

Most of the work with neural networks has been done by either computer scientists and engineers interested in applications, or biologists and psychologists interested in the analogies with actual nervous systems and learning. Their results are largely empirical, and hence inexact. However, even when

mathematicians or other scientists attempt to produce analysis, the results are disappointing - neural networks have not yet lent themselves well to mathematical techniques.

Where mathematical results do exist, they are usually of one of two types. Either the result is a broad positive theorem, or a special case where the common wisdom fails. The first kind of result is reassuring, but rarely useful - the existing theorems tend to be fairly obvious and without significant practical application. The second kind shows why the first kind is so rare - virtually every useful conjecture already has a counter-example attached to it. Thus it is difficult to produce any general results of value.

For example, there are task which neural networks can perform, but for which given a specific learning rule, it cannot learn, or only has a small probability of learning [5]. Thus general conjectures about the learning of tasks are almost certainly false, unless they are seriously weakened by the placing of strong conditions on the predicate, or contain a probabilistic conclusion. The latter is common in applications - most workers are satisfied if a system *usually* learns a task.

This paper intends to avoid obscure counter-examples, and provide positive results which are as practical and testable as possible. The result will be dependent on the existence of a network which can perform the task, and the strength of the result will depend upon the aptness of the parameters. The result will be an existence statement - no reference will be made to learning. Thus the result will have some modest practical value, but be strictly limited theoretically. Of particular interest will be the contrast with the negative results arising from a similar conjecture for larger systems.

II. DESCRIPTION

The purpose of this paper is to analyze a phenomenon first noted by Burton, Mpitsos, and their colleagues in their investigation of annealling [7]. While working with a 1-4-1 feed-forward network using gradient-descent back-propagation, they observed similar results whether learning was allowed at the hidden units or only at the output units. In the latter case, synapse weights were arbitrarily set at the hidden units, and they were never adjusted in response to error. Yet despite this hobbling, the systems almost always learned the tasks comparably quickly and well.

This phenomenon was convenient for analyzing the effects of annealling, and it was during my own research on annealling that I became aware of it. Assuming learning is only necessary at one layer simplifies analysis considerably, and thus it became important to determine when this assumption can be made. Furthermore, if this phenomenon is specific to a certain kind of network, it may call into question empirical results based on work with those networks.

Briefly, this paper attempts to show that in a 1-n-1 feed-forward network, given a task, if a set of weights exist so the network performs the task within a certain error, then if the weights at the hidden units are set arbitrarily, a set of weights at the output unit exists so that the task is performed comparably well. A maximum error may be calculated which is dependent on the appropriateness of the chosen hidden weights. The maximum error is unfortunately uselessly large, but it gives a qualitative indication of the goodness of the weights. Further,

evidence is given that the conjecture is not true for most multi-input networks.

Note that the proposition only claims existence of a set of weights, not that a particular learning rule will ensure that those weights will be found. Thus the proposition is independent of the algorithm chosen. Also, the success of the new network is dependent upon the success of an objective network. Thus if the task cannot be performed accurately by a network in which all the weights are adjustable, the proposition is trivial.

In the single-input (1-n-1) case, the proof consists of an algorithm for determining the weights at the output unit, given the weights of an objective network and the arbitrarily determined hidden weights. Although the error calculations provide no assurance that the new network will be useful, experiments verify that the algorithm works surprisingly well.

In the multi-input (m-n-1) case, a similar algorithm is developed, but it fails experimentally. An explanation is given why the method fails for the system it was tested on. Also, there will be a discussion of the difficulties of producing any effective algorithm.

The major limitation of the method is that the transfer function must be C^n, i.e. continuously n-times differentiable. This paper assumes the logistic transfer function as standard, although the result is valid and the calculations similar for other functions.

In the conclusion, several important limitations and objections will be noted and discussed. What is described here are the results of the application of a method to a specific approximation task. The full ramifications of this will be determined after the results are shown.

III. FIRST RESULT

Consider a 1-n-1 feed-forward neural network, with transfer function $f:\mathbb{R}\rightarrow(0,1)$ which is C^n and bijective, a task represented by a target function $t:I\rightarrow[0,1]$, where I is a set of possible input values, i.e. a training set. Then the output of the network may be considered as a function $o:I\rightarrow(0,1)$. We will assume $I=(0,1)$, and inputs are chosen uniformly. Then the mean-square error is given by $E=\int_0^1 (t(x)-o(x))^2\,dx$.

In the following analysis, hatted variables refer to values for an objective network. Non-starred variables refer to the values for a second network which is intended to approximate the first. The transfer function is the same for both. Define

$n, \hat{n}$ = number of hidden units in each system.

$z_i, \hat{z}_i$ = weight of synapse from input unit to hidden unit i.

$w_i, \hat{w}_i$ = weight of synapse from hidden unit i to output unit.

x = an arbitrary input.

The goal of the algorithm is to approximate $\hat{o}(x)$ by $o(x)$ over all $x \in I$, where

$$o(x) = f\left(\sum_{i=1}^{n} (w_i \cdot f(z_i \cdot x))\right) \tag{3.1}$$

and all z_i are fixed, distinct, and non-zero. Since f is a continuous bijection we will develop a method for approximating $f^{-1}(\hat{o}(x))$ by $\sum_{i=1}^{n} (w_i \cdot f(z_i \cdot x))$, and

consider the effects of f in our error calculations.

Thus the task is to approximate

$$\sum_{i=1}^{n} (\hat{w}_i \cdot f(\hat{z}_i \cdot x)) \tag{3.2}$$

by

$$\sum_{i=1}^{n} (w_i \cdot f(z_i \cdot x)) \tag{3.3}$$

given the z_i as fixed. We accomplish this by approximating each term in the sum in expression (3.2) by a term like expression (3.3), then adding the resulting values for each w_i. That is if

$$\hat{w}_j \cdot f(\hat{z}_j \cdot x) \doteq \sum_{i=1}^{n} (w_i^j \cdot f(z_i \cdot x)) \tag{3.4}$$

then

$$\sum_{j=1}^{\hat{n}} (\hat{w}_j \cdot f(\hat{z}_j \cdot x)) \doteq \sum_{i=1}^{n} ((\sum_{i=1}^{\hat{n}} w_i^j) \cdot f(z_i \cdot x)) \tag{3.5}$$

To accomplish the approximation in equation (3.4), we build an augmented matrix using the Taylor expansions of $\hat{w}_i \cdot f(\hat{z}_i \cdot x)$ and $f(z_i \cdot x)$, taking the coefficients of the first n terms, and solving to find the w_i values.

Specifically, let $b \in [0,1]$ be the base point for the Taylor expansions. Let

$$g_i(x) = \hat{w}_i \cdot f(\hat{z}_i \cdot x). \tag{3.6}$$

$$\vec{v}_i = \left[g_i(b), g_i'(x), \tfrac{1}{2} g_i''(x), \cdots, \tfrac{1}{(n-1)!} g_i^{(n-1)}(x) \right]^T \tag{3.7}$$

$$A = \begin{bmatrix} f(z_1 b) & f(z_2 b) & \cdots & f(z_n b) \\ f'(z_1 b) z_1 & f'(z_2 b) z_2 & \cdots & f'(z_n b) z_n \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{(n-1)!} f^{(n-1)}(z_1 b) z_1^{n-1} & \cdots & \cdots & \frac{1}{(n-1)!} f^{(n-1)}(z_n b) z_n^{n-1} \end{bmatrix} \tag{3.8}$$

If rank(A)=n, then let $\vec{s}_i$ be the solution to $A\vec{s}_i = \vec{v}_i$. Let $\vec{w} = \sum_{i=1}^{n} \vec{s}_i$. Then $\vec{w} = [w_1, w_2, ..., w_n]^T$ is the desired set of weights. Note that for b=0, A reduces to a Vandermond matrix, and hence has rank n, as the z_i are distinct and non-zero.

If the rank of A is less than n, then either the base point or the transfer function were poorly chosen. For most transfer functions, given the set of z_i, the set of points such that rank(a)<n is small. If the set is large, then it indicates that some nodes are redundant, and the function is inappropriate for the task. For most transfer functions, simply changing the base point arbitrarily is sufficient remedy to the problem.

The maximum error of the new system can be calculated as follows:

Let R[f(x)] = remainder of the Taylor expansion of f to n-1 terms.

Let $f_a(x) = f(a \cdot x)$. Note $f_a^{(m)}(x) = a^m \cdot f^{(m)}(a \cdot x)$

Let h = the maximum possible difference between the systems before the application of the transfer function at the output unit.

Then

$$h = \max_{x \epsilon I} \left| \sum_{i=1}^{\hat{n}} (\hat{w}_i \cdot f(\hat{z}_i \cdot x)) - \sum_{i=1}^{n} (w_i \cdot f(z_i \cdot x)) \right| \tag{3.9}$$

$$= \max_{x \epsilon I} \left| \sum_{i=1}^{\hat{n}} \hat{w}_i \cdot R[f(\hat{z}_i x)] - \sum_{j=1}^{n} w_j \cdot R[f(z_j x)] \right|$$

$$= \max_{x \epsilon I} \left| \sum_{i=1}^{\hat{n}} \hat{w}_i \frac{1}{n!} f_{\hat{z}_i}^{(n)}(\alpha_i)(x-b)^n - \sum_{j=1}^{n} w_j \frac{1}{n!} f_{z_j}^{(n)}(\beta_j)(x-b)^n \right|$$

for some $\alpha_1, \alpha_2, ..., \alpha_{\hat{n}}, \beta_1, \beta_2, ..., \beta_n$.

$$= \max_{x\epsilon I} \left| \sum_{i=1}^{\hat{n}} \frac{1}{n!} \hat{w}_i \cdot z_i^n \cdot f^{(n)}(\alpha_i \cdot z_i)(x-b)^n - \sum_{j=1}^{n} \frac{1}{n!} w_j \cdot z_j^n \cdot f^{(n)}(\beta_j \cdot z_j)(x-b)^n \right|$$

$$h \leq \frac{1}{n!} \cdot \max_{x\epsilon I} \left[\left| \sum_{i=1}^{\hat{n}} \hat{w}_i \cdot z_i^n \cdot f^{(n)}(\alpha_i \cdot z_i)(x-b)^n \right| + \left| \sum_{j=1}^{n} w_j \cdot z_j^n \cdot f^{(n)}(\beta_j \cdot z_j)(x-b)^n \right| \right]$$

Let $M = \max_{x\epsilon\mathbb{R}} \left| f^{(n)}(x) \right|$. If $b=\frac{1}{2}$, note $\max_{x\epsilon I} \left| (x-b)^n \right| = 2^{-n}$.

$$\leq \frac{1}{n!} \sum_{i=1}^{\hat{n}} \left| \hat{w}_i \cdot z_i^n \cdot M \cdot 2^{-n} \right| + \frac{1}{n!} \sum_{j=1}^{n} \left| w_j \cdot z_j^n \cdot M \cdot 2^{-n} \right|$$

finally yielding

$$h \leq \frac{1}{n!} \cdot M \cdot 2^{-n} \cdot \left[\sum_{i=1}^{\hat{n}} \left| \hat{w}_i \cdot z_i^n \right| + \frac{1}{n!} \sum_{j=1}^{n} \left| w_j \cdot z_j^n \right| \right] \tag{3.10}$$

Remembering that h represents the maximum of the difference between the two systems before calculating the transfer function at the output node. Then let o(x) and $\hat{o}(x)$ be the outputs of the two networks, and E and $\hat{E}$ be the mean square error of the two networks. Then

$$E = \int_0^1 (t(x)-o(x))^2 \, dx \tag{3.11}$$

$$= \int_0^1 t^2(x) - 2 \cdot t(x)o(x) + o^2(x) - 2 \cdot t(x)\hat{o}(x) + \hat{o}^2(x) + 2 \cdot t(x)\hat{o}(x) - \hat{o}^2(x) \, dx$$

$$= \int_0^1 (t(x)-\hat{o}(x))^2 \, dx + \int_0^1 2 \cdot t(x)\hat{o}(x) - 2 \cdot t(x)o(x) \, dx + \int_0^1 o^2(x) - \hat{o}^2(x) \, dx$$

$$= \hat{E} + 2\int_0^1 t(x)(\hat{o}(x) - o(x))\,dx + \int_0^1 (o(x) - \hat{o}(x))(o(x) + \hat{o}(x))\,dx$$

Let $m = \max_{x\in\mathbb{R}} |f'(x)|$. Note $\max_{x\in I} |t(x)| \leq 1$.

$$E \leq \hat{E} + 2hm + 2hm$$

$$E \leq \hat{E} + 4hm \tag{3.12}$$

Thus the error bound is dependent upon the values of the synapse weights, on the number of hidden units of the new system, and on M and m, the maximum of the derivatives of the transfer function. The difficult factor to predict is the values of the w_j. Clearly we want moderate values, which require that the z_j be "appropriate" for the application. A thorough analysis of this is far beyond the scope of this paper. Intuitively, though, the z_j values should be well spread and adequately ranged. Importantly, though, they should not be too large, since the error bound grows with z_j^n.

The values of M and m depend upon the transfer function. For the logistic function $f(x) = (1 + \exp(-x))^{-1}$, m = .25, and M is given by the following table:

n	$M = \max \left\| f^{(n)}(x) \right\|$
1	0.25
2	0.0962
3	0.125
4	0.1277
5	0.25
7	1.0625
9	7.75

For large n, the logistic function has large derivatives. This diminishes the advantage of using a large number of hidden units.

EXPERIMENTAL VERIFICATION

Of course, the actual error is generally much less than the maximum bound. In practice, the maximum bound is uselessly large, but the actual error is closely comparable to the original system. The results of my first attempt to apply the algorithm illustrate this clearly.

A 1-2-1 system was trained by gradient descent back-propagation to approximate the target function t(x)=x, with learning occurring at all units, and inputs chosen uniformly over (0,1). Learning was halted when no improvement was noted in a span of 2000 iterations. The weights were recorded, and the system became the objective system for the experiment.

The new system was also a 1-2-1 network, with $z_1 = 1$, $z_2 = -1$. For the objective function, $\hat{z}_1 = 1.769$, $\hat{z}_2 = -1.708$, $\hat{w}_1 = 4.057$, $\hat{w}_2 = -9.451$. The base point chosen was $b = \frac{1}{2}$. Thus

$$\vec{v}_1 = [\ 4.057 \cdot f(1.769 \cdot 0.5),\ 4.057 \cdot f'(1.769 \cdot 0.5)\ ]$$

$$= [\ 2.872,\ 1.484\ ]$$

$$\vec{v}_2 = [\ -2.822,\ 3.381\ ]$$

$$A = \begin{bmatrix} f(0.5) & f(-0.5) \\ f'(0.5) & -f'(-0.5) \end{bmatrix} = \begin{bmatrix} 0.6225 & 0.3775 \\ 0.2350 & -0.2350 \end{bmatrix}$$

From this was calculated

$\vec{w}$ = [7.865, -12.838]

E and $\hat{E}$ were estimated numerically using a rectangular approximation with 100 partitions over the integral. $\hat{E}$ equalled approximately 0.001204. Thus we can calculate an upper bound on E, and compare it with the actual value.

From equation (3.10), $h \leq \frac{1}{2}\cdot M\cdot\frac{1}{4}\cdot\left[\sum_{i=1}^{2} \left|\hat{w}_i\cdot z_i^n\right| + \frac{1}{2}\sum_{j=1}^{2} \left|w_j\cdot z_j^n\right| \right]$. From the table, M = 0.0962, thus

$$h = \frac{0.0962}{8}\left[12.69 + 27.57 + 7.865 + 12.838 \right] = 0.733$$

From equation (3.12),

$$E \leq \hat{E} + 4hm = 0.001204 + 4\cdot 0.25\cdot 0.733 = 0.734$$

which is disappointingly large, to the point of irrelevancy. However, the actual value of E is 0.00089, which is less than $\hat{E}$. This is partly a matter of luck, but it also indicates an amount of error in the objective system at the extremes of the interval, which is removed by the approximation process. The fact that the target function is easily approximated by this method allows this improvement. Of course, the difference in performance of the two systems is practically negligible.

Using the same objective system, new networks were calculated using other base points for the expansion and values for the hidden weights. The results are summarized here:

base point	z_1	z_2	w_1	w_2	E	Error bound
0.5	1.0	-1.0	7.865	-12.838	0.00089	0.734
0.5	2.0	-2.0	3.377	-8.996	0.00147	1.079
1.0	1.0	-1.0	6.105	-9.100	0.00963	2.668
1.0	2.0	-2.0	3.713	-10.521	0.00301	4.675
0.0	1.0	-1.0	8.963	-14.357	0.00166	3.058
0.0	2.0	-2.0	3.133	-8.527	0.00203	4.180

The results conform loosely to our expectations from the error bound calculations. The error bound predicted the relative magnitude of the error reasonably well in every case but one, the third in the table. The error bound correctly predicted that even though the larger z_i values were closer to the $\hat{z}_i$ values, and the resulting w_i values were smaller, the error was generally larger. The error bound also correctly indicated the advantage of using $\frac{1}{2}$ as the base point. The degree of the importance of these phenomenon are exagerrated in the error bound calculations, just as the maximum error is rediculously larger than the actual error. Thus, although the error bound formula is useless as a rigorous test of the efficacy of the method, it has some value in analyzing the results.

In summary, the method works better than anticipated. The error bound formula is useful mainly as an indicator of some general trends, and provides little support for the proposition of the thesis. However, the empirical results support the thesis admirably.

IV. SECOND RESULT

Single input networks are a small sub-class of feed-forward networks, and an extension of the method to multi-input networks would be much more useful. The method does generalize easily to the larger networks, but unfortunately it fails to produce the desired results.

The method for multi-input networks is substantially the same as for single-input networks. In the latter, the new system is calculated to have the same output for a certain basepoint input, and the first several derivatives at that point are also duplicated. Specifically, $\hat{o}(b) = o(b)$, and $\hat{o}^{(k)}(b) = o^{(k)}(b)$ for $1 \leq k \leq n-1$, $b \in I$. For multi-input networks, the new system is designed to have the same partial derivatives as the objective network. Thus if $I = \{ \vec{x} \mid \vec{x} = (x_1, x_2, ..., x_m), x_i \in I_i \text{ for all } i \}$, $\vec{b} \in I$, then the system will have the properties $\hat{o}(\vec{b}) = o(\vec{b})$, and $\frac{\partial \hat{o}}{\partial x_i} = \frac{\partial o}{\partial x_i}$ for $1 \leq i \leq m$, and perhaps similarly for higher order partial derivatives.

For single input networks, the size of the new network was arbitrary, and corresponded to the highest derivative for which the new system matches the objective system. Specifically, for $\hat{o}^{(k)}(b) = o^{(k)}(b)$ for all $k \leq n$ requires $n+1$ neurons at the hidden layer. But for multi-input networks, the required number of neurons at the hidden layer is much larger. If the system has m inputs, then a first order approximation (where all first partial derivatives match at the base vector) requires $m+1$ hidden units, and a second order approximation requires $1 + m + m \cdot (m+1)/2$ hidden units, since each different partial derivative of first and

second order must be calculated and solved for. The number of units required is a polynomial in the number of inputs, with the degree of the polynomial matching the order of the approximation. Thus higher order approximations require an awkward number of hidden units.

The execution of the method again relies upon the additivity of the inputs to the output unit, and the approximation is calculated at the output synapse before the transfer function is applied. Thus each neuron of the objective network may be approximated individually, and the resulting values summed to get the final values.

Specifically, let $\hat{o}$: I $\rightarrow$ (0,1) be the output of a m - $\hat{n}$ - 1 feed-forward neural network, with transfer function f: $\mathbb{R} \rightarrow$ (0,1) which is k times differentiable. Define the following symbols:

ζ_b^a = weight of synapse from input unit b to hidden unit a.

ω_a = weight of synapse from hidden unit a to the output unit.

Let o: I $\rightarrow$ (0,1) be the output of the new m - n - 1 network, with the same transfer function. Define

z_b^a = weight of synapse from input unit b to hidden unit a.

w_a = weight of synapse from hidden unit a to the output unit.

Represent the values of the inputs for both networks by $\vec{x} = (x_1, x_2, ..., x_m)$, $\vec{x} \in I$. Let $\vec{b} \in I$ be the base point for the approximation.

For illustration, consider the method for a first order approximation. The procedure for higher order approximations is a straightforward extension. Thus let n = m + 1. The problem then is to find a set of w_a given arbitrary z_a^b such

that $\hat{o}(\vec{b}) = o(\vec{b})$, and $\frac{\partial \hat{o}}{\partial x_i} = \frac{\partial o}{\partial x_i}$ for $1 \le i \le m$.

Since $\hat{o}(\vec{x}) = f(\sum_{i=1}^{\hat{n}}(\omega_i \cdot f(\sum_{j=1}^{m}\zeta_j^i \cdot x_j))$ and $o(\vec{x}) = f(\sum_{i=1}^{n}(w_i \cdot f(\sum_{j=1}^{m} z_j^i \cdot x_j))$, we apply f^{-1} to both to simplify the task. Then for $1 \le k \le \hat{n}$, we approximate $\omega_k \cdot f(\sum_{j=1}^{m}\zeta_j^k \cdot x_j))$ by $\sum_{i=1}^{n}(w_i \cdot f(\sum_{j=1}^{m} z_j^i \cdot x_j))$. We find w_i such that

$$\omega_k \cdot f(\sum_{j=1}^{m}\zeta_j^k \cdot b_j)) = \sum_{i=1}^{n}(w_i \cdot f(\sum_{j=1}^{m} z_j^i \cdot b_j) \tag{4.1}$$

and for $1 \le a \le m$

$$\frac{\partial}{\partial x_a}\ \omega_k \cdot f(\sum_{j=1}^{m}\zeta_j^k \cdot b_j)) = \frac{\partial}{\partial x_a}\ \sum_{i=1}^{n}(w_i \cdot f(\sum_{j=1}^{m} z_j^i \cdot b_j). \tag{4.2}$$

Simplifying (4.2) yields

$$\omega_k \cdot \zeta_a^k \cdot f'(\sum_{j=1}^{m}\zeta_j^k \cdot b_j)) = \sum_{i=1}^{n}(w_i \cdot z_a^i \cdot f'(\sum_{j=1}^{m} z_j^i \cdot b_j) \tag{4.3}$$

Equation (4.1) and the set of m equations from (4.3) provide a set of n equations in n variables. Denote the solution, if it exists, $\vec{w}_k$. After completing this operation for k from 1 to $\hat{n}$, then $\vec{w} = \sum_{i=1}^{\hat{n}} \vec{w}_i = (\ w_1, w_2, ..., w_n\ )$ is the desired set of weights.

No attempt has been made to provide an error bound on this method, although it may be done in a manner analogous to the earlier calculation.

EXPERIMENTAL RESULTS

To test the method, a 3 - 7 - 1 network was trained to perform integer addition mod 2, using uniformly random values from $\{0,1\}\times\{0,1\}\times\{0,1\}$. This task may be considered a classification problem, with the criterion for classification being the parity of the input. Both first order and second order approximations were calculated, using (0,0,0) and $(\frac{1}{2},\frac{1}{2},\frac{1}{2})$ as base points. The first order approximations formed 3 - 4 - 1 networks, and the z values were chosen specifically. The second order approximations included some redundancy, thus a 3 - 13 - 1 was created, rather than a 3 - 10 - 1 network. The z values were chosen randomly from several different ranges. All attempts had one thing in common: the resulting network failed to perform the task. The results were comparable to networks with random weights.

Admittedly, the task chosen was a particularly difficult task for this method to approximate. The target function is not statistically correlated to any of the individual inputs, or even to any pair. The training of the objective network using the extreme values of $[0,1]^m$ is irrelevant, but measuring the error of the new network using those values is "tough". However, if the error is measured using inputs uniformly chosen on $(0,1)^m$, the error is still large, but now the training of the original network on the extrema contributes to the error. Experimentally, a 3 - 7 - 1 network appears to be incapable of learning to calculate the decimal part of the sum of three real numbers, anyway, so it is hardly surprising that the new network also fails.

In conclusion, for this task, the method fails. The behavior at the base point is a poor indicator of the behavior of the system over the entire domain of inputs. The behavior of the objective system over the range $(0,1)^m$ is not a

simple interpolation of the behavior at the endpoints, and any method which assumes it is is likely doomed to failure.

But one example of failure does not mean that the method always fails. Special cases may be devised where the method succeeds. A sufficiently well-behaved function, such as a planar function, or a function which is dependent mainly on one input, is approximable by this method. For example, a 2 - 3 - 1 network trained to average the two input values was the objective function of a first order approximation, with base point ($\frac{1}{2}$, $\frac{1}{2}$), and z values ± 1 assigned in a manner which avoided duplication. The objective network had an approximate mean-square error of 0.000991. The resulting 2 - 3 - 1 network had a mean-square error of 0.000747. Once again, the new system had a lower error than the objective system, just as happened for the analogous example of a single input system. This can again be attributed to the simplicity of the target function, and some inefficiency in the learning process.

However, tasks like the averaging problem are not a very large class. Most neural network applications are more akin to the parity classification task, where the method fails. Classification problems usually utilyze discrete inputs, and the outputs associated with inputs in the interior of $(0,1)^m$, besides being irrelevant to the application, are not easily approximable. Thus the method is unsuccessful in approximating them.

V. CONCLUSIONS

The method evidently works for networks trained to approximate relatively simple functions over the entire interval $(0,1)^m$, but not for classification problems, which generally only use the endpoints of or certain discrete values in the interval. Single-input networks are rarely used for the latter type of problem, and multi-input networks are rarely used for the former type. Thus there is a strong motivation to state the result in terms of the number of inputs, though technically that would be incorrect. The statements made in the description of the method thus are false, but they are a useful generalization for most applications.

Note that the situations where the method works are precisely those where the network may be constructed from scratch to perform the task, without recourse to learning. First, the task must be performable, and second, the task must be simple and utilyze the entire range of inputs, thus enabling the use of the Taylor expansions. If this is the case, a set of weights could be determined using a method analogous to this paper's, without using any objective function or any learning. There are also functions which are not approximable using Taylor expansions but are still approximable by neural networks. For these, other approximation schemes, such as Bernstein polynomials, can be applied to similar effect. One weakness is that the transfer function at the output unit may limit the accuracy of the approximation. Thus the method applies almost precisely to those situations where learning can be superceded.

Returning to the original question - whether learning can be suppressed at the hidden layer of a three-layered network - we must consider three conjectures. First, whether the tests of the method described in this paper are sufficient to generalize to all tasks. Second, whether the method used is capable of providing sets of weights in all situations when weights exist. And third, whether the existence of output weights given specific hidden weights implies that learning can be suppressed at the hidden units during the learning process. If all three statements are true, then we have a simple criterion for determining when learning can be suppressed at the hidden layers.

The first conjecture is certainly supportable. The three-input parity classification task is a very simple classification problem, and failure there is certainly a strong indication of general failure on classification problems. The examples of success are also very simple, and there likely are intermediate problems where the success is marginal. This area - the approximation of functions over an interval by neural networks - has received relatively little attention, and more research needs to be done to support the thesis in this grey area. The positive tests are sufficient for at least a qualified affirmative of the conjecture.

Whether another technique exists which may succeed where this method fails is a more difficult question. Any such method must utilyze information about the behavior of the system at the input points, rather than a single base point, and thus if there are a large number of such points, a large number of hidden units must be present in the new system.. If the new method utilyzes systems of equations, as this method did, one hidden unit must be provided for each equation. Thus if each of the eight elements of the training set of the 3-

input parity classifier is used to determine a system, then the new network must have at least eight hidden units. The number of hidden units grows exponentially with the number of inputs, thus such a method would become impractical with large networks. Furthermore, the result would be diluted considerably - it is of less interest to note that learning at the hidden units may be suppressed in networks with a very large number of hidden units relative to the number of inputs. Thus, even if an alternative method exists, it is scant contradiction to the second conjecture.

As for the third conjecture, there are few guarantees where the application of a learning rule is considered, but since the positive results are only for simple tasks, where learning is less likely to encounter problems like local minima, the conjecture is strong. The appropriateness of the chosen hidden weights is important, and is included in the analysis of the method. Since they are assumed to be appropriate - specifically, well spread and adequately ranged - some learning problems, like coupling, are eliminated. Hence, the statement about learning here is about as reliable as any positive statement one can make about learning. Lastly, the conjecture was verified by experiment. Every system for which the method applied also could learn the task with learning suppressed at the hidden layer, and the learning was comparably fast, and yielded similarly small error.

Thus we have a thesis: learning may be suppressed at the hidden units of three-layered neural networks, where the hidden weights are well-spread and ranged, and the task is a simple function with domain equal to $(0,1)^m$. To be considered simple, the function must be approximable by a neural network using the same transfer function. Learning may not be reliably suppressed in other systems, particularly classification tasks, assuming a moderate number of hidden

units.

And with the thesis, rather than a rigorous demonstration, we have three sources of objection. The objections are valid and important to the understanding of the thesis, and they should not be discounted. On the other hand, the grey areas they introduce are far enough from the normal regions of interest in neural networks to give confidence in the thesis for any "normal" application.

Identifying the grey areas where the thesis is weakest provides grist for the researcher's mill. The objections raised above, and the conditionals in the thesis, all point to important problems.

The first and second objections can be considered a challenge to find a method of assigning weights to a network without utilyzing a learning rule. This is probably impossible or at least impractical, but a modified version may be considered, though it too is obviously very difficult. Given a set of weights which does perform a task, find another distinct set of weights which also performs the task. The z values evidently cannot be arbitrarily chosen, but perhaps it may be possible to find full sets of weights. If so, headway may be made in research on pruning, analysis of minima, and optimal architecture.

The third objection leads to questions about learning and reliability of learning rules. Although general results are impossible, simplifying by accepting as a task the approximation of a second, objective network, and suppressing learning at the hidden units, results may likely be obtained which, if their scope is not large, at least they fill a gap in the present thesis.

The conditionals in the thesis are that the z values be appropriate, and that an objective function exist. The error bounds and the algorithm provide a

means for analyzing the appropriateness of z values, and a further examination of them may provide insight into optimizing them for faster learning. The question of existence of an objective function is unanswered, particularly for continuous functions, which have received less attention than classification problems, but represent a large grey area. The number of units required for the system is also unknown. For functions over the full interval, some progress seems attainable here, which if produced would further reduce the uncertainty in this thesis.

This thesis provides some avenues for further research. The methods used here also apply to some of these other problems. By using an objective function which is assumed to already approximate the task, by approximating before the final application of the tranfer function, by considering only a single output (since multiple outputs are an obvious generalization for this particular problem), and by utilying the additivity of inputs in conjunction with the additivity of differentiation to allow consideration of each hidden unit individually, the problem was considerably reduced. Investigating the internal workings of networks is difficult. Simplifications such as these are valuable.

And the thesis also provides a basis for commentary on other research. The results demonstrate that there is a clear qualitative and quantitative difference between classification problems and approximation problems over the full interval. Since, for the most part, learning in the latter may be suppressed at the hidden layer, the interaction of the hidden units must be different, or at least more complicated, in the former. Results for learning in one class do not automatically apply to the other.

In sum, we have a thesis which is not unexpected, which has several important limitations and objections. The thesis is sufficient to allow application

to actual systems, and is well verified by experiment. The limitations and objectives are not crippling, but further investigation of them would be appropriate. The methods used in the thesis may also be useful in other research.

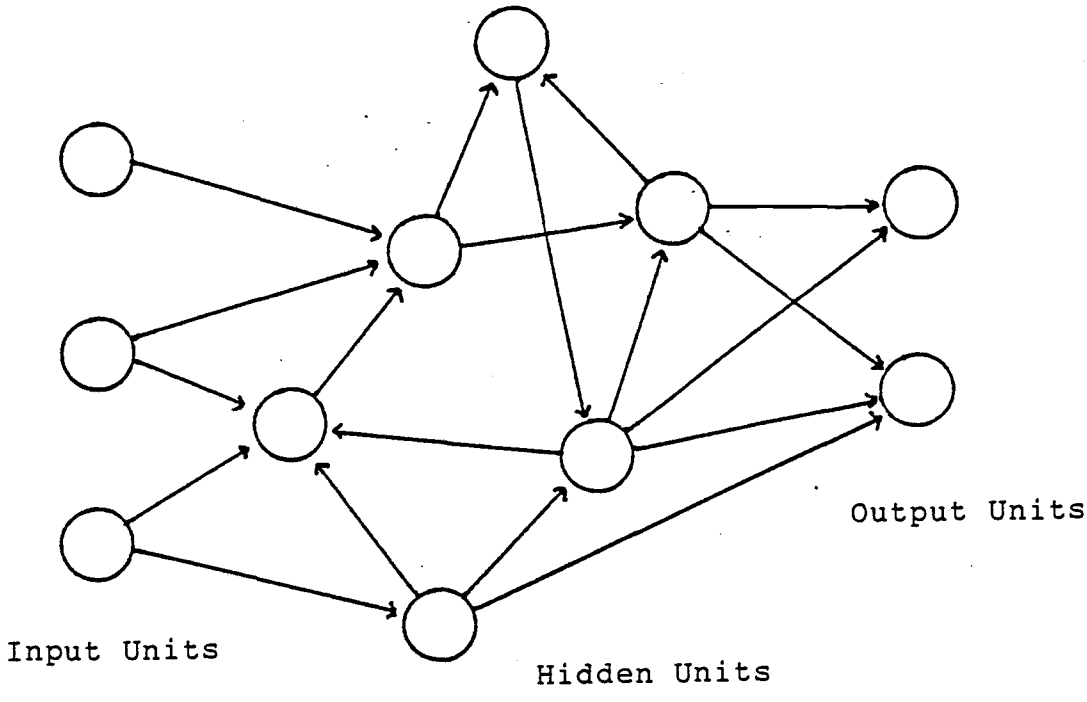

Figure 1. A neural network. The network has three input units, six hidden units, and two output units. Note that it is not "feed-forward".

Each circle represents a node, each arrow is a synapse. Although this diagram has no examples, it is acceptable for a synapse to have the same input and output node, thus forming a small loop, or for two nodes to be connected by two synapses, one directed each way.

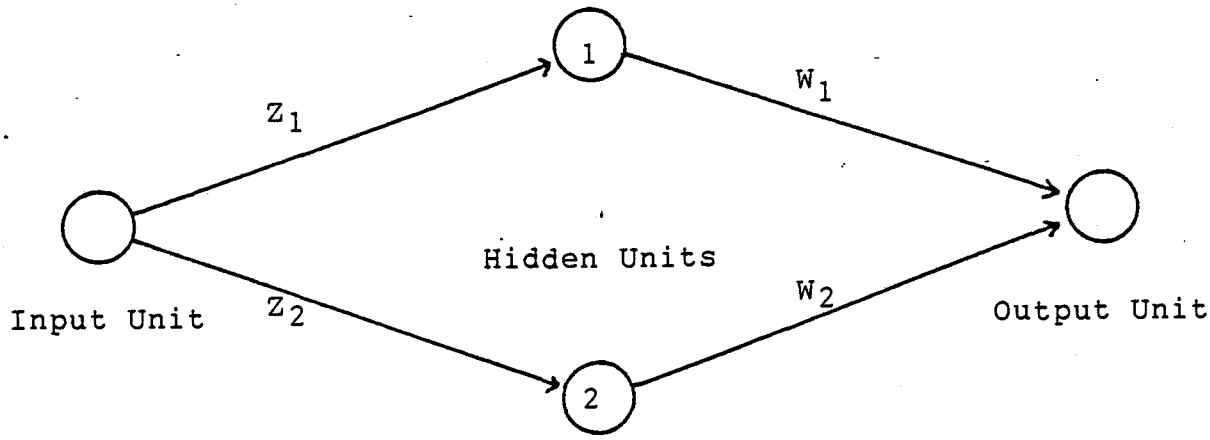

Figure 2. A 1 - 2 - 1 feed-forward neural network.

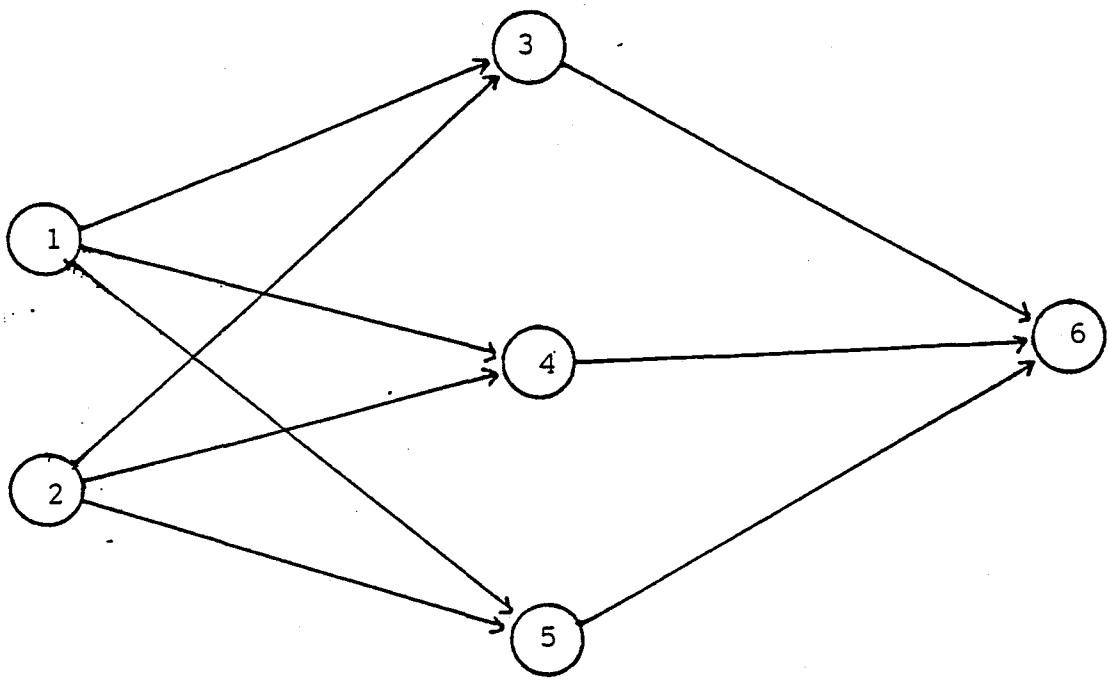

Figure 3. A 2 - 3 - 1 feed-forward neural network.

BIBLIOGRAPHY

1. Rumelhart, D.E., Hinton, G.E., Williams, R.J., Parallel Data Processing, Vol. 1, Cambridge MA, MIT Press, 1986.

2. "Soviet Model Picks Bush", Electrical Engineering Times, October 17, 1988.

3. Plaut, D.C., Nowlan, S.J., Hinton, G.E., "Experiments on Learning by Back Propagation", Pittsburgh, PA, Carnegie-Mellon University, 1986.

4. Werbos, P.J., Ph.D. Thesis, Harvard University, 1974.

5. Brady, M., Raghavan, R., "Gradient Descent Fails to Separate", International Conference on Neural Networks, San Diego, 1988.

6. Sietsma, J., Dow, R.J.F., "Neural Net Pruning, Why and How", IEEE ICNN Vol. 1, San Diego, 1988.

7. Burton, R.M., Mpitsos, G.J., "Event-Dependent Control of Noise Enhances Learning in Neural Networks: Generalization, Geometry, and Applicability to Biological Systems", publication pending.