TRAINING A 3-NODE NEURAL NETWORK IS NP-COMPLETE

Avrim L. Blum*
MIT Laboratory for Computer Science
Cambridge, Mass. 02139 USA

Ronald L. Rivest†
MIT Laboratory for Computer Science
Cambridge, Mass. 02139 USA

Appears in: *Neural Networks*, Vol. 5, pp.117-127, 1992.

Copyright 1992 Pergamon Press plc

Running title: Training a 3-Node Network

*This material is based upon work supported under a National Science Foundation graduate fellowship.
†This paper was prepared with support from NSF grant DCR-8607494, ARO Grant DAAL03-86-K-0171, and the Siemens Corporation.

TRAINING A 3-NODE NEURAL NETWORK IS NP-COMPLETE

Abstract: We consider a 2-layer, 3-node, n-input neural network whose nodes compute linear threshold functions of their inputs. We show that it is NP-complete to decide whether there exist weights and thresholds for this network so that it produces output consistent with a given set of training examples. We extend the result to other simple networks. We also present a network for which training is hard but where switching to a more powerful representation makes training easier. These results suggest that those looking for perfect training algorithms cannot escape inherent computational difficulties just by considering only simple or very regular networks. They also suggest the importance, given a training problem, of finding an appropriate network and input encoding for that problem. It is left as an open problem to extend our result to nodes with non-linear functions such as sigmoids.

Keywords: Neural networks, computational complexity, NP-completeness, intractability, learning, training, multilayer perceptron, representation.

1 INTRODUCTION

One reason for the recent surge in interest in feed-forward neural networks is the development of the "back-propagation" training algorithm (Rummelhart, Hinton and Williams, 1986). The ability to train large multi-layer networks is essential for utilizing neural networks in practice (eg. Sejnowski and Rosenberg (1987)), and the back-propagation algorithm promises just that. In practice, however, the back-propagation algorithm often runs very slowly (Tesauro and Janssens, 1988), and the question naturally arises as to whether there are necessarily intrinsic computational difficulties associated with training neural networks, or whether better training algorithms might exist. This paper shows that in a certain worst-case sense, there are intrinsic difficulties in training even some very simple 2-layer networks.

A common paradigm for the use of neural networks is that a sample of data is divided into a training set and a test set; the network is trained for some time on the training set until it makes few mistakes, and its performance is then measured on the test set. Two important theoretical issues arise in this framework. One is a *sample complexity* question which we do *not* deal with here, but see Baum and Haussler (1989) and Haussler (1989), which asks: how large should the training set be so that one can expect good performance in the training phase to translate to good performance in the testing phase? The other issue is the *computational complexity* question: how much computational effort is required to achieve good performance in the training phase in the first place? This paper addresses the latter issue.

For the single-layer, n-input perceptron, if there *exist* edge weights so that the network correctly classifies a given training set, then such weights can be found in time guaranteed to be polynomial in n, using linear programming. The question arises: is there an algorithm with the same guarantees for larger multi-layer networks? This paper shows that no such training algorithm exists for a very simple 2-layer network with only two hidden nodes and a single output node, unless a widely believed complexity-theoretic assumption proves false. Specifically, we show that unless P = NP, for any polynomial-time training algorithm there will be some sets of training data on which the algorithm fails to correctly train the network, even though there *exist* edge weights so the network *could* correctly classify the data.

1.1 Previous work

A common method of demonstrating a problem to be intrinsically hard is to show the problem to be NP-complete. NP is the class of decision problems for which an affirmative answer can be verified in polynomial time, and NP-complete problems are the hardest problems of this class; they are hardest in the sense that a polynomial time algorithm to solve one NP-complete problem could be used to solve any problem in NP in polynomial time. (NP-hard problems are like NP-complete problems, but need not belong to the class NP.) Also, P is the class of those decision problems solvable in polynomial time. Although no proof is known that no polynomial-time algorithm exists for NP-complete problems (that is, that P $\neq$ NP), many infamous hard problems—such as the traveling salesman problem—are now known to be NP-complete. A good discussion of the theory of NP-completeness, as well as a description of several hundreds of NP-complete problems, is given by Garey and Johnson (1979). While NP-completeness does not render a problem totally inapproachable in practice, and does not address the specific instances one might wish to solve, it often implies that only small instances of the problem can be solved exactly, and that large instances can at best only be solved approximately, even with large amounts of computer time.

The work in this paper is inspired by Judd (1990) who shows the following problem to be NP-complete:

> "Given a neural network and a set of training examples, does there exist a set of edge weights for the network so that the network produces the correct output for all the training examples?"

Judd shows that the problem remains NP-complete even if the network is only required to produce the correct output for two-thirds of the training examples, which implies that even approximately training a neural network is intrinsically difficult in the worst case (Judd, 1988). Judd produces a class of networks and training examples for those networks such that any training algorithm will perform poorly on some networks and training examples in that class. The results, however, do not specify any particular "hard network"—that is, any single network hard to train for all algorithms. Also, the networks produced have a number of hidden nodes that grows with the number of inputs and outputs, as well as a quite irregular connection pattern.

The work in this paper is also inspired by Megiddo (1986) who shows that if input features are allowed to be arbitrary rational values, then training a variant of the main network we consider here is NP-complete. If inputs are restricted to, say, binary or ternary values, then his proof techniques break down. The proofs we present here for our more general results are of a very different style.

1.2 Our results

We extend the results of Judd and Megiddo by showing that it is NP-complete to train a *specific* very simple network, that has only two hidden nodes, a regular interconnection pattern, and binary input features. We also present classes of regular 2-layer networks such that for *all* networks in these classes, the training problem is NP-complete. In addition, we relate certain problems in approximate network training to other difficult (but not known to be NP-hard) approximation problems. In particular, we consider the problem of finding approximation algorithms that make only one-sided error and the problem of approximating the minimum number of hidden-layer nodes needed for correct classification of a given training set.

Our results, like Judd's, are described in terms of "batch"-style learning algorithms that are given all the training examples at once. It is worth noting that training is at least as hard with an "incremental" algorithm, such as back-propagation, that sees the examples one at a time.

Our results state that given a network of the classes considered, for any training algorithm there will be some types of training problems such that the algorithm will perform poorly as the problem size

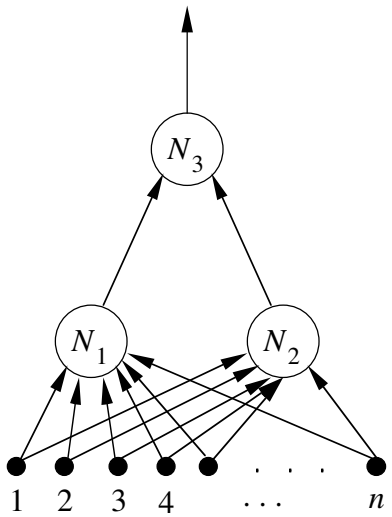

Figure 1: The 3-Node Network.

increases. The results leave open the possibility that given a training problem that is hard for some network, there might exist a different network and encoding of the input that make training easy. In fact, we present an example of two networks, the second more powerful than the first, such that training the first is NP-complete but the second can be trained in polynomial time. So, in particular, those sets of examples hard to train on the first network can be trained easily on the other. Kearns and Valiant (1989) show, however, that there exist more complicated networks for which this approach will not help in the worst case. Preliminary versions of this paper have appeared in Blum and Rivest (1988), (1989) and Blum (1989).

2 THE TRAINING PROBLEM AND NETWORKS CONSIDERED

Definition 1 *Given a neural network $\mathcal{N}$, let the* training problem for $\mathcal{N}$ *be the question:*

> *"Given a set of training examples, do there exist edge weights and thresholds for the nodes of $\mathcal{N}$ so that it produces output consistent with the training set?"*

Note that we have stated the training problem as a decision ("yes" or "no") problem, but that the search problem (finding the weights) is at least as hard.

For most of this paper, we will focus on a multilayer network with n binary inputs and three nodes labeled N_1, N_2, N_3. All inputs are connected to nodes N_1 and N_2. The outputs of hidden nodes N_1 and N_2 are connected to output node N_3 which gives the output of the network (see Figure 1).

Each node N_i computes a linear threshold function (also called N_i) on its inputs. If N_i has input $x = (x_1, \ldots, x_m)$, then for some values $a_0, \ldots, a_m$,

$$N_i(x) = \begin{cases} +1 & \text{if } a_1x_1 + a_2x_2 + \cdots + a_mx_m > a_0 \\ -1 & \text{otherwise.} \end{cases}$$

The a_j's ($j \geq 1$) are typically viewed as weights on the incoming edges and a_0 as the threshold. We will call the network as described above the 3-*Node Network*.

A training algorithm for this network is given a set of training examples. Each is either a positive example (an input for which the desired network output is $+1$) or a negative example (an input for which the desired output is -1). The main result of this paper is that the training problem for the 3-Node Network is NP-complete. That is, unless P = NP there is no polynomial-time algorithm that given a collection of training examples on n Boolean inputs, can always correctly decide whether there exist linear threshold functions for nodes N_1, N_2, and N_3 so that the 3-Node Network produces output consistent with the training examples.

Since it is NP-complete to train, the 3-Node Network differs greatly in a computational sense from the single-node perceptron which can be trained in polynomial time using linear programming. Note the 3-Node Network training problem is in NP since the maximum number of bits needed for each weight is the same as that needed for the weights of a perceptron. Raghavan (Raghavan, 1988) shows that in fact one needs at most $O(n \log n)$ bits per weight (and threshold) and therefore one can certainly write down all the weights and thresholds, and then verify that the network so produced classifies all examples correctly, in polynomial time.

We also show the training problem for the following networks to be NP-complete:

1. The 3-Node Network restricted so that any or all of the weights for one hidden node are required to equal the corresponding weights of the other (so possibly only the thresholds differ) and any or all of the weights are required to belong to $\{+1, -1\}$.
2. Any k-hidden node, for $k \geq 2$ and bounded above by some polynomial in n (eg: $k = n^2$), two-layer fully-connected network with linear threshold function nodes where the output node is required to compute the AND function of its inputs.
3. The 2-layer, 3-node n-input network with an XOR output node, if ternary features are allowed.

In addition we show that any set of positive and negative training examples classifiable by the 3-node network with XOR output node (for which training is NP-complete) can be correctly classified by a perceptron with $O(n^2)$ inputs which consist of the original n inputs and all products of pairs of the original n inputs (for which training can be done in polynomial-time using linear programming techniques).

3 TRAINING THE 3-NODE NETWORK IS NP-COMPLETE

In this section, we prove the following theorem.

Theorem 1 *Training the 3-Node Network is NP-complete.*

First, we provide some intuition. To see why training such a simple network might be hard, imagine that the output node were required to compute the AND function of its inputs—that is, output $+1$ when it receives inputs $(+1, +1)$ from nodes N_1 and N_2, and output -1 on all other pairs of inputs. When the network is presented with a positive example, we know that both hidden nodes N_1 and N_2 must output $+1$. Therefore, we know in some sense in what direction we should modify the weights of these nodes. When the network is presented with a negative example, however, all we know is that either N_1 *or* N_2 (or both) should output -1. We might, perhaps, just try to make both nodes output -1, but unless the positive and negative examples are linearly separable—implying that we could have solved the training problem on a perceptron—this will not work. For some negative examples, we will have to make a choice: should N_1 output -1 or should N_2 output -1? It may be that we must make the correct combination of choices over all or at least a large number of the negative examples in order to correctly train the network, and there are an exponential number of such combinations. NP-completeness tells us that in the worst case, we will not be able to do much better than just blindly trying all combinations and seeing if one happens to work, which clearly would take exponential time. So, regardless of the linear programming problem of finding a good set of weights for a node given that we know what it

should output, what makes the training problem hard is that we must decide what the outputs for the hidden nodes should be in the first place.

The proof of Theorem 1 involves reducing the known NP-complete problem "Set-Splitting" to the network training problem. In order to more clearly understand the reduction, we begin by viewing network training as a geometrical problem.

3.1 The geometric point of view

A training example can be thought of as a point in n-dimensional Boolean space $\{0,1\}^n$, labeled '+' or '$-$' depending on whether it is a positive or negative example. The zeros of the linear functions that are thresholded by nodes N_1 and N_2 can be thought of as $(n-1)$-dimensional hyperplanes in this space. These hyperplanes divide the space into four quadrants according to the four possible pairs of outputs for nodes N_1 and N_2. If the hyperplanes are parallel, then one or two of the quadrants is degenerate (non-existent). In this paper, the words "plane" and "hyperplane" will be used interchangeably.

Since the output node receives as input only the outputs of the hidden nodes N_1 and N_2, it can only distinguish between points in different quadrants. The output node is also restricted to be a linear function. It may not, for example, output "+1" when its inputs are $(+1,+1)$ and $(-1,-1)$, and output "-1" when its inputs are $(+1,-1)$ and $(-1,+1)$.

So, the 3-Node Network training problem is equivalent to the following: given a collection of points in $\{0,1\}^n$, each point labeled '+' or '$-$', does there exist either

1. a single plane that separates the '+' points from the '$-$' points, or
2. two planes that partition the points so that either one quadrant contains all and only '+' points or one quadrant contains all and only '$-$' points.

We first look at a restricted version which we call the *Quadrant of Positive Boolean Examples* problem:

> "Given $O(n)$ points in $\{0,1\}^n$, each point labeled '+' or '$-$', do there exist two planes that partition the points so that one quadrant contains all '+' points and no '$-$' points?"

The Quadrant of Positive Boolean Examples problem corresponds to having an "AND" function at the output node. Once we have shown this to be NP-complete, we will extend the proof to the full problem by adding examples that disallow the other possibilities at the output node. Megiddo (Megiddo, 1986) has shown that for a collection of *arbitrary* '+' and '$-$' points in n-dimensional Euclidean space, the problem of whether there exist two hyperplanes that separate them is NP-complete. His proof breaks down, however, when one restricts the coordinate values to $\{0,1\}$ as we do here. Our proof turns out to be of a quite different style.

3.2 Set-splitting

The following problem, *Set-Splitting*, was proven to be NP-complete by Lovász (Garey and Johnson, 1979).

> "Given a finite set S and a collection C of subsets c_i of S, do there exist disjoint sets S_1, S_2 such that $S_1 \cup S_2 = S$ and $\forall i, c_i \not\subset S_1$ and $c_i \not\subset S_2$?"

The Set-Splitting problem is also known as 2-non-Monotone Colorability or Hypergraph 2-colorability. Our use of this problem is inspired by its use by Kearns, Li, Pitt, and Valiant to show that learning k-term DNF is NP-complete (Kearns et al., 1987) and the style of the reduction is similar.

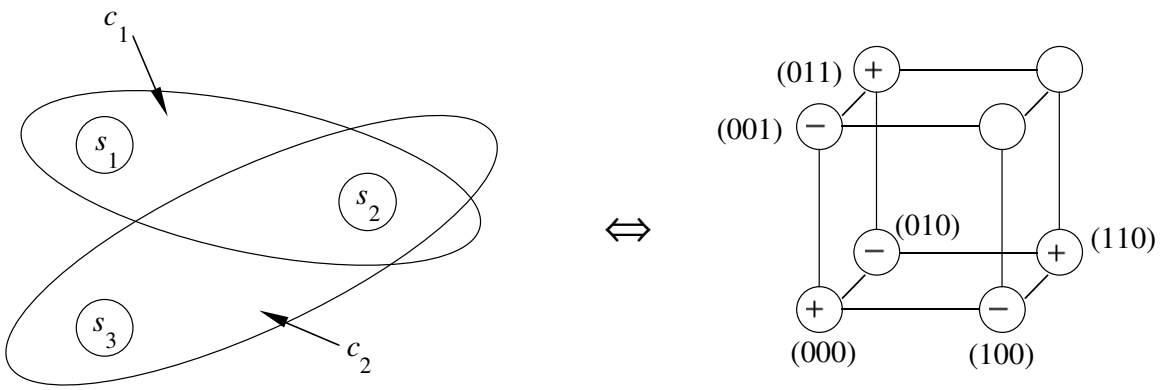

Figure 2: An example.

3.3 The proof

Theorem 2 *Quadrant of Positive Boolean Examples is NP-complete.*

Proof: The proof is by reduction from Set-Splitting. That is, given an instance of Set-Splitting, we convert it into an instance of Quadrant of Positive Boolean Examples, such that the constructed instance has a solution if and only if the Set-Splitting instance had a solution.

So, given an instance of the Set-Splitting problem:

$$S = \{s_i\},\ C = \{c_j\},\ c_j \subseteq S,\ |\ S\ | = n,$$

We create the following signed points on the n-dimensional hypercube $\{0,1\}^n$:

- Let the origin 0^n be labeled '+'.
- For each s_i, put a point labeled '−' at the neighbor to the origin that has a 1 in the ith bit—that is, at $(\overset{1}{0}\overset{2}{0}\cdots\overset{i}{1}\cdots\overset{n}{0})$. Call this point $\mathbf{p}_i$.
- For each $c_j = \{s_{j1}, \ldots, s_{jk_j}\}$, put a point labeled '+' at the location whose bits are 1 at exactly the positions $j_1, j_2, \ldots, j_{k_j}$—that is, at $\mathbf{p}_{j1} + \cdots + \mathbf{p}_{jk_j}$.

For example, let $S = \{s_1, s_2, s_3\},\ C = \{c_1, c_2\},\ c_1 = \{s_1, s_2\},\ c_2 = \{s_2, s_3\}$. We create '−' points at positions: (0 0 1), (0 1 0), (1 0 0) and '+' points at positions: (0 0 0), (1 1 0), (0 1 1) in this reduction (see Figure 2).

We now show that the given instance of the Set-Splitting problem has a solution *iff* the constructed instance of the Quadrant of Positive Boolean Examples problem has a solution.

($\Rightarrow$)

Given S_1, S_2 from the solution to the Set-Splitting instance, let P_1 be the plane $a_1x_1 + \ldots + a_nx_n = -\frac{1}{2}$, where $a_i = -1$ if $s_i \in S_1$, and $a_i = n$ if $s_i \notin S_1$. Similarly, let P_2 be the plane $b_1x_1 + \ldots + b_nx_n = -\frac{1}{2}$ where $b_i = -1$ if $s_i \in S_2$, and $b_i = n$ otherwise. Let $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$.

Plane P_1 separates from the origin all '−' points corresponding to $s_i \in S_1$ and no '+' points. For each $s_i \in S_1,\ \mathbf{a} \cdot \mathbf{p}_i = -1$, which is less than $-\frac{1}{2}$. For each '+' point $\mathbf{p}$ we have $\mathbf{a} \cdot \mathbf{p} > -\frac{1}{2}$ since either $\mathbf{p}$ is the origin or else $\mathbf{p}$ has a 1 in a bit i such that $s_i \notin S_1$. Similarly, plane P_2 separates from the origin all

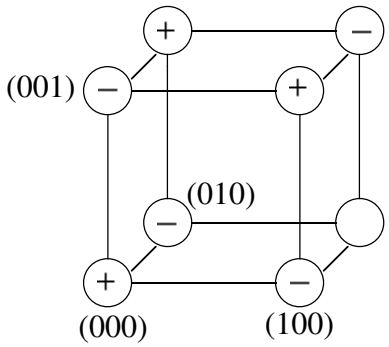

Figure 3: The gadget.

'$-$' points corresponding to $s_i \in S_2$ and no '$+$' points. Thus, the quadrant $\mathbf{a} \cdot \mathbf{x} > -\frac{1}{2}$ and $\mathbf{b} \cdot \mathbf{x} > -\frac{1}{2}$ contains all points labeled '$+$' and no points labeled '$-$'.

($\Leftarrow$)

Let S_1 be the set of points separated from the origin by P_1 and S_2 be those points separated by P_2. Place any points separated by both planes in either S_1 or S_2 arbitrarily. Sets S_1 and S_2 cover S since all '$-$' points are separated from the origin by at least one of the planes. Consider some $c_j = \{s_{j1} \cdots s_{jk_j}\}$ and the corresponding '$-$' points $\mathbf{p}_{j1}, \ldots, \mathbf{p}_{jk_j}$. If, say, $c_j \subset S_1$, then P_1 must separate all the $\mathbf{p}_{ji}$ from the origin. Therefore, P_1 must separate $\mathbf{p}_{j1} + \ldots + \mathbf{p}_{jk_j}$ from the origin. Since that point is the '$+$' point corresponding to c_j, the '$+$' points are not all confined to one quadrant, contradicting our assumptions. So, no c_j can be contained in S_1. Similarly, no c_j can be contained in S_2. ■

We have shown that the training problem for the 3-Node Network is NP-complete if the output node is required to compute the AND of its two inputs. In order to handle the other possibilities at the output node, we now add a "gadget" consisting of six new points in three new dimensions. The gadget forces that the only way in which two planes could separate the '$+$' points from the '$-$' points would be to confine the '$+$' points to one quadrant.

Proof of Theorem 1: Given an instance of Set-Splitting, create examples as in the proof of Theorem 2, except in addition we add three new dimensions, $x_{n+1}, x_{n+2},$ and x_{n+3}, and put '$+$' points in locations:

$$(0 \cdots 0\ 101),\ \ (0 \cdots 0\ 011)$$

and '$-$' points in locations:

$$(0 \cdots 0\ 100),\ \ (0 \cdots 0\ 010),\ \ (0 \cdots 0\ 001),\ \ (0 \cdots 0\ 111).$$

(See Figure 3.)

The '$+$' points of this cube can be separated from the '$-$' points by appropriate settings of the weights of planes P_1 and P_2 corresponding to the three new dimensions. Given planes $P_1' : a_1 x_1 + \cdots + a_n x_n = -\frac{1}{2}$ and $P_2' : b_1 x_1 + \cdots + b_n x_n = -\frac{1}{2}$ which solve a Quadrant of Positive Boolean Examples instance in n dimensions, expand the solution to handle the gadget by setting

$$\begin{aligned} P_1 \quad &\text{to} \quad a_1 x_1 + \cdots + a_n x_n + x_{n+1} + x_{n+2} - x_{n+3} = -\tfrac{1}{2} \\ P_2 \quad &\text{to} \quad b_1 x_1 + \cdots + b_n x_n - x_{n+1} - x_{n+2} + x_{n+3} = -\tfrac{1}{2} \end{aligned}$$

(P_1 separates '$-$' point $(0 \cdots 0\ 001)$ from the '$+$' points and P_2 separates the other three '$-$' points from the '$+$' points). On the other hand, notice that no single plane can separate the '$+$' points from the '$-$' points in the cube and there is no way for two planes to confine all the negative points in one

quadrant. Thus, any solution to the network training problem must have all '+' points in one quadrant and so as in the proof of Theorem 2, give a solution to the Set-Splitting instance. ■

4 CLASSES OF HARD NETWORKS

4.1 The Restricted 3-Node Network

In order to approach the dividing line between computational feasibility and infeasibility for neural network training, we now consider an even simpler network. If we require the two hidden nodes N_1 and N_2 of the 3-Node Network to compute exactly the same function, then the network would reduce to the simple perceptron and be trainable in polynomial time. However, suppose we allow only the thresholds used by N_1 and N_2 to differ; that is, we require just the weights on edges into node N_1 to equal the corresponding weights on edges into node N_2. We show that the training problem for such a network is NP-complete. Thus, adding the single extra free parameter of thresholds that may differ results in intractability. Another natural way we might simplify the network would be to require the edge weights to be either $+1$ or -1. This requirement forces nodes N_1 and N_2 to each separate out some Hamming ball in $\{0,1\}^n$—that is, all points on the hypercube differing in at most some fixed number of bits from some center—instead of just any linearly-separable region. Unfortunately, training for this type of network is also NP-complete as we will show.

Definition 2 *A* Restricted 3-Node Network *is a version of the 3-Node Network in which some or all of the weights of hidden node N_1 are required to equal the corresponding weights of hidden node N_2, with possibly only the thresholds allowed to differ, and in which some or all of the weights may be restricted to be from the set $\{-1,+1\}$.*

We prove that training the Restricted 3-Node Network is NP-complete. The proof uses a reduction from Set-Splitting slightly different from that in the last section and we use a form of the Set-Splitting problem in which the subsets c_j have at most three elements (this restricted version of Set-Splitting is still NP-complete). The reduction has the property that the following are equivalent:

- The instance of the Set-Splitting problem is solvable.
- The sets of '+' and '−' points created can be separated by two hyperplanes.
- The points can be separated by two parallel hyperplanes with coefficients in $\{+1,-1\}$.

That is, the reduction will also imply that training the 3-Node Network remains NP-hard even if we only look at training sets in which all the positive examples lie in two disjoint Hamming balls. Thus, restricting oneself to considering only sets of training data where the concept (set of positive examples) consists of two disjoint Hamming balls does not reduce the computational complexity in the worst case. The proof appears in appendix A.

4.2 Networks with More Intermediate Nodes

We will now consider networks with more than two nodes in the hidden layer and present a large class of such networks for which training is NP-complete.

Definition 3 *Let Λ be the family of 2-layer, n-input, single-output networks in which there are $r \geq 2$ linear threshold function nodes in the hidden layer, each one connected to all n inputs, and in which the output node computes the AND function. That is, the output node outputs $+1$ if and only if all of its inputs are $+1$.*

The class Λ is just the straightforward generalization of the 3-Node Network to networks with more than two hidden nodes, with the restriction that the output node compute the AND of its inputs instead of an arbitrary linear threshold function.

Theorem 3 *For any network of the family Λ such that the number of hidden nodes, r, is bounded by some fixed polynomial in the number of inputs, n, the training problem is NP-complete.*

Essentially, to prove this result, for each of $r-2$ hidden nodes, we take an unused corner of the n-dimensional hypercube and label it '$-$' and all its neighbors '$+$'. This will force a hyperplane corresponding to a hidden node to have as its sole function separating the '$-$' point from the rest of the hypercube. There will be two hidden nodes left so we can then use the reduction from the proof of Theorem 1. The proof appears in appendix B.

4.3 The 3-Node Network with XOR Output

The last network for which we will show training to be NP-complete is a modification of the 3-Node Network in which the output node computes the XOR function. When the outputs of the two hidden nodes are $(+1,-1)$ or $(-1,+1)$, then the network output is "$+1$" and otherwise the network output is "-1". We will call this network the *3-Node Network with XOR Output*, or *3NX*. The motivation for considering this network is that in chapter 6 we will present a network that can both correctly classify any set of training examples that 3NX can, and be trained in polynomial time. This shows that worst-case hardness of training is not necessarily directly related to network power.

In the following discussion, we will suppose that the inputs to 3NX are from a ternary alphabet. Instead of each input being on or off, an input can be positive, negative or neutral.

Theorem 4 *Training 3NX is NP-complete if ternary input attributes are allowed.*

Proof: The ternary attributes used are $\{-1,0,1\}$ so every training example is a vector in $\{-1,0,1\}^n$ labeled '$+$' or '$-$'. Given an instance of Set-Splitting on n elements, create signed points in $\{-1,0,1\}^n$ as follows:

- Let the origin 0^n be labeled '$+$'.
- For each s_i, put a '$-$' point at $p_i = (\overset{1}{0}\,\overset{2}{0}\,\overset{\cdots}{\cdots}\,0\,\overset{i}{1}\,0\,\overset{\cdots}{\cdots}\,\overset{n}{0})$ and at $-p_i = (\overset{1}{0}\,\overset{2}{0}\,\overset{\cdots}{\cdots}\,0\,\overset{i}{-1}\,0\,\overset{\cdots}{\cdots}\,\overset{n}{0})$.
- For each $c_j = \{s_{j_1},\ldots,s_{j_k}\}$, put a '$+$' point at $p_{j_1}+\cdots+p_{j_k}$.

These points are the same as in the proof of Theorem 1 except the reflection of each '$-$' point through the origin is also given and there is no "gadget".

A solution S_1, S_2 to the Set-Splitting instance can be translated into the same plane equations P_1, P_2 as in the proof of Theorem 1. P_1 is $a_1x_1+\ldots+a_nx_n = -\frac{1}{2}$ where $a_i=-1$ for $s_i \in S_1$ and $a_i = n$ for $s_i \notin S_1$; P_2 is created from S_2 similarly. Notice that the '$-$' point p_i is separated from the '$+$' points by P_1 if $s_i \in S_1$ and by P_2 if $s_i \in S_2$. Conversely $-p_i$ is separated from the '$+$' points by P_2 if $s_i \in S_1$ and by P_1 if $s_i \in S_2$. Also, no '$-$' point is separated from the '$+$' points by both planes which implies that the network can correctly classify the training examples with an XOR output node.

A solution P_1, P_2 to the training problem can be translated into sets S_1, S_2, where $S_1 = \{s_i \mid P_1$ separates p_i from the origin $\}$ and $S_2 = \{s_i \mid P_2$ separates p_i from the origin $\}$. The following claim implies that these sets solve the Set-Splitting instance.

Claim: Given $c_j = \{s_{j_1},\ldots,s_{j_k}\}$, P_1 does *not* separate all of $p_{j_1},\ldots,p_{j_k}$ from the origin.

Proof of claim: If P_1 separates all of the p_{j_i} from the origin, it also separates the point $p = p_{j_1} + \ldots + p_{j_k}$ (the '+' point corresponding to c_j) from the origin and does not separate any of the $-p_{j_i}$ from the origin. Therefore, the other plane P_2 must separate all of the $-p_{j_i}$ from the origin and cannot separate p or any of the p_{j_i} from the origin. So, the point p and all the p_{j_i} are on the same side of both planes and the training problem is not correctly solved.

The claim implies that each c_j is split by S_1 and S_2, proving the theorem. ■

5 GETTING AROUND INTRACTABILITY

The results presented in the previous sections show several classes of networks such that for any training algorithm there will be some hard training problems. It is quite possible, however, that a problem hard for one network might be easier for another network. In this section, we describe two networks such that training the first is NP-complete, but the second can both be trained in polynomial time and is more powerful than the first in that it can be trained correctly on any set of examples the first is powerful enough to correctly classify. This phenomenon was discovered independently by Valiant and Warmuth (1989).

The first network is the network 3NX described earlier. The second is a perceptron with an expanded input representation. This perceptron has $2n + n(n-1)/2$ inputs, consisting of the original n inputs, their squares, and all $n(n-1)/2$ products of pairs of the original n inputs. We will call this network P^2 and the regular n-input perceptron, P. The number of weights in P^2 is $O(n^2)$, compared with $O(n)$ for 3NX. However, P^2 can be trained in polynomial time since it is just a perceptron with $O(n^2)$ inputs.

Theorem 5 *Any set of training data that 3NX can correctly classify, P^2 can also correctly classify.*

Proof: Let $w_1x_1 + \ldots + w_nx_n + w_0 \geq 0$ and $v_1x_1 + \ldots + v_nx_n + v_0 \geq 0$ be the linear threshold functions for the two hidden nodes of 3NX. (Notice we have moved the thresholds w_0 and v_0 to the left-hand sides of the inequalities.) We may assume that on all training examples, $w_1x_1 + \ldots + w_nx_n + w_0 \neq 0$ and $v_1x_1 + \ldots + v_nx_n + v_0 \neq 0$, since we can perturb the thresholds w_0 and v_0 by slight amounts if we wish and not affect the function computed by the network. Therefore, the network 3NX outputs "+1" exactly when

$$(w_1x_1 + \ldots + w_nx_n + w_0 > 0) \text{ and } (v_1x_1 + \ldots + v_nx_n + v_0 < 0)$$

or

$$(w_1x_1 + \ldots + w_nx_n + w_0 < 0) \text{ and } (v_1x_1 + \ldots + v_nx_n + v_0 > 0).$$

Equivalently, 3NX outputs "+1" exactly when

$$(w_1x_1 + \ldots + w_nx_n + w_0)(v_1x_1 + \ldots + v_nx_n + v_0) < 0$$

which implies

$$v_0w_0 + \sum_{i=1}^{n}(v_0w_i + w_0v_i)x_i + \sum_{i=1}^{n} v_iw_ix_i^2 + \sum_{i=2}^{n}\sum_{j=1}^{i-1}(w_iv_j + v_iw_j)x_ix_j > 0.$$

The left-hand side of this last formula is a linear function of the inputs to P^2. So, there exist edge weights for P^2 (those described by the above formula) such that P^2 classifies the examples in exactly the same way as does 3NX. ■

Theorem 5 shows that by increasing the power of a network, it is possible to remove as well as to introduce computational intractability. In terms of their representational power, we have:

$$P \subseteq \text{3NX} \subseteq P^2$$

where P can be trained in polynomial time, training 3NX is NP-complete, and P^2 can again be trained in polynomial time. Intuitively, the reason that network P^2 can be both more powerful than 3NX and easier to train is that we are giving it predefined non-linearities. The network P^2 does not have to start from scratch, but instead is given more powerful building blocks (the products of pairs of the inputs) to work with.

By using P^2 instead of 3NX, we gain in a worst-case computational sense, but lose in that the number of weights increases from $O(n)$ to $O(n^2)$. The increase in the number of weights implies that the number of training examples needed to constrain those weights so that the network can meaningfully generalize on new examples increases correspondingly (eg. see Baum and Haussler, 1989). Thus, there is a tradeoff. Theorem 5 can be extended in the obvious way to networks like 3NX with $k > 2$ hidden nodes; the number of inputs to the resulting perceptron will be n^k.

In practice, if one were to use the strategy of adding non-linear inputs to the perceptron, then instead of giving the perceptron all $O(n^2)$ products of pairs as inputs at once, one might just give the network those products that appear related to the training problem at hand. One could then test to see whether those products suffice by running a training algorithm and checking whether or not the network correctly classifies the training data. In addition, products of triples of inputs or other non-linear functions of the original inputs could be given as new inputs to the perceptron if the trainer has some prior knowledge of the particular training problem.

6 HARDNESS RESULTS FOR APPROXIMATION ALGORITHMS

We now state, but do not prove, two hardness results on approximate network training; the proofs appear in (Blum, 1989).

The first problem we consider is relaxing the restriction that the trained network output correctly on *all* the training examples, even if there exist edge weights so that the network would do so. Judd (1988) shows that there exist ⟨network, training set⟩ pairs for which outputting correctly on better than 2/3 of the training examples is NP-hard. He proves this result by showing training to be NP-complete for some such pair in which the training set has only 3 elements and therefore one cannot do better than 67% accuracy without achieving 100% accuracy. The networks he considers are quite complicated and contain many output nodes, however. Our results are weaker than his in that we cannot show that achieving such a high error rate is necessarily hard, but hold for the very simple networks discussed in the previous chapters.

Definition 4 *A* training algorithm with one-sided error *for a single-output network $\mathcal{N}$ is an algorithm that given a collection of positive and negative training examples that $\mathcal{N}$* can *correctly classify, will produce edge weights so that $\mathcal{N}$ outputs correctly on all of the positive examples and at least an ϵ fraction of the negative examples, for some constant $\epsilon > 0$.*

In this section we will use the problem *Graph k-Colorability*. An instance of this problem is a graph consisting of n vertices connected by some number of edges and k allowed colors. A solution is an assignment to each vertex of one of the k colors so that no edge has both endpoints given the same color. Graph k-Colorability is NP-complete for $k \geq 3$ and approximate graph coloring (approximating the minimum number of colors needed to color a graph) appears to be a hard problem in the worst case also for all $k \geq 3$.

Theorem 6 *For any network $\mathcal{N} \in \Lambda$ with n inputs and $k \geq 3$ hidden nodes, any training algorithm with one-sided error for $\mathcal{N}$ can be used to color any n-vertex k-colorable graph with $O(k \log n)$ colors.*

Theorem 6 implies, for instance, that training the network $\mathcal{N} \in \Lambda$ that has 3 hidden nodes so that $\mathcal{N}$ will output correctly on all the positive examples and on at least 10% of the negative examples ($\epsilon = 0.1$) on a collection of training data which M is powerful enough to correctly classify, is as hard in the worst case as $O(\log n)$-coloring a 3-colorable graph.

Finding $O(k \log n)$ approximations for the k-coloring problem is not known to be NP-complete, but $O(k \log n)$ is much lower than the bounds achieved by the current best approximation algorithms which all grow as n^α for a constant $\alpha < 1$. Thus, Theorem 6 suggests that one-sided error training in the worst case is "probably hard".

A second form of approximate training we consider is that given a set of training examples that is hard for a particular network, one might try to add power to the network in some way in order to make training easier. For the 2-layer networks of the kind discussed in this paper, one natural way to add power is to add more nodes to the hidden layer. We show that for networks of the class Λ, if one adds only relatively few nodes to the hidden layer, then there will be training sets that are hard for both the original and the enlarged network, so this approach will likely not help in the worst case.

Definition 5 *Given two networks $\mathcal{N}$ and $\mathcal{N}'$, an* $\mathcal{N}'/\mathcal{N}$-training algorithm *is one that given any set of training data that $\mathcal{N}$ is powerful enough to correctly classify, will correctly train $\mathcal{N}'$.*

Thus, for instance, in the last section we showed a $P^2/3NX$-training algorithm.

Theorem 7 *Given network $\mathcal{N} \in \Lambda$ with k hidden nodes and $\mathcal{N}' \in \Lambda$ with k' hidden nodes ($k' > k$), then $\mathcal{N}'/\mathcal{N}$-training is as hard as coloring a k-colorable graph with k' colors.*

Theorem 7 implies that to avoid NP-completeness, one must in general at least double the number of hidden nodes, since it is NP-hard to color a k-colorable graph with $2k - \epsilon$ colors for general k. Current state-of-the-art coloring approximation algorithms (Wigderson, 1983; Blum, 1989b) suggest that one may wish to add at least n^α hidden nodes, ($0 < \alpha < 1$) for α depending on the original number of hidden nodes k. Of course there is no guarantee here that adding this number of hidden nodes will actually help, in a worst-case computational complexity sense.

7 CONCLUSIONS

We show for many simple two-layer networks whose nodes compute linear threshold functions of their inputs that training is NP-complete. For any training algorithm for one of these networks there will be some sets of training data on which it performs poorly, either by running for more than an amount of time polynomial in the input length, or by producing sub-optimal weights. Thus, these networks differ fundamentally from the perceptron in a worst-case computational sense.

The theorems and proofs are in a sense fragile; they do not imply that training is necessarily hard for networks other than those specifically mentioned. They do, however, suggest that one cannot escape computational difficulties simply by considering only very simple or very regular networks.

On a somewhat more positive note, we present two networks such that the second is both more powerful than the first and can be trained in polynomial time, even though the first is NP-complete to train. This shows that computational intractability does not depend directly on network power and provides theoretical support for the idea that finding an appropriate network and input encoding for one's training problem is an important part of the training process.

An open problem is whether the NP-completeness results can be extended to neural networks that use the differentiable logistic linear functions. We conjecture that training remains NP-complete when these functions are used since it does not seem their use should too greatly alter the expressive power of a neural network (though Sontag (1989) has demonstrated some important differences between such functions and thresholds). Note that Judd (1990), for the networks he considers, shows NP-completeness for a wide variety of node functions including logistic linear functions.

References

[1] Baum, E. B. and Haussler, D. (1989). What size net gives valid generalization? In *Advances in Neural Information Processing Systems I*, pages 81–90. Morgan Kaufmann.

[2] Blum, A. (1989). On the computational complexity of training simple neural networks. Master's thesis, MIT Department of Electrical Engineering and Computer Science. (Published as Laboratory for Computer Science Technical Report MIT/LCS/TR-445 (May, 1989).).

[3] Blum, A. (1989b). An $\tilde{O}(n^{0.4})$-approximation algorithm for 3-coloring (and improved approximation algorithms for k-coloring). In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, Seattle, pages 535–542.

[4] Blum, A. and Rivest, R.L. (1988). Training a 3-node neural network is NP-Complete. In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pages 9–18. Morgan Kaufmann.

[5] Blum, A. and Rivest, R. L. (1989) Training a 3-node neural net is NP-Complete. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems I*, pages 494–501. Morgan Kaufmann.

[6] Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco.

[7] Haussler, D. (1989). Generalizing the PAC model for neural net and other learning applications. Technical Report UCSC-CRL-89-30, University of California Santa Cruz.

[8] Judd, J. S. (1988). *Neural Network Design and the Complexity of Learning.* PhD thesis, University of Massachusetts at Amherst, Department of Computer and Information Science.

[9] Judd, J. S. (1990). *Neural Network Design and the Complexity of Learning.* MIT Press.

[10] Kearns, M., Li, M., Pitt, L., and Valiant, L. (1987). On the learnability of boolean formulae. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 285–295, New York.

[11] Kearns, M. and Valiant, L. (1989). Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 433–444, Seattle, Washington.

[12] Megiddo, N. (1986). On the complexity of polyhedral separability. Technical Report RJ 5252, IBM Almaden Research Center.

[13] Raghavan, P. (1988). Learning in threshold networks. In *First Workshop on Computational Learning Theory*, pages 19–27. Morgan-Kaufmann.

[14] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In David E. Rumelhart and James L. McClelland, editors, *Parallel Distributed Processing – Explorations in the Microstructure of Cognition*, chapter 8, pages 318–362. MIT Press.

[15] Sejnowski, T. J. and Rosenberg, C. R. (1987). Parallel networks that learn to pronounce English text. *Journal of Complex Systems*, 1(1):145–168.

[16] Sontag, E. D. (1989). Sigmoids distinguish better than Heavisides. *Neural Computation*, 1:470–472.

[17] Tesauro, G. and Janssens, B. (1988). Scaling relationships in back-propagation learning. *Complex Systems*, 2:39–44.

[18] L. Valiant and Warmuth M.K. (1989). Predicting symmetric differences of two halfspaces reduces to predicting halfspaces. Unpublished manuscript.

[19] Wigderson, A. (1983). Improving the performance guarantee for approximate graph coloring. *JACM*, 30(4):729–735.

APPENDIX A

Theorem 8 *The training problem for a Restricted 3-Node Network is NP-complete.*

Proof of Theorem 8: The reduction proceeds as follows. Given an instance of the Set-Splitting problem on $n/2$ elements in which each subset c_j has at most three elements:

$$S = \{s_i\},\ C = \{c_j\},\ c_j \subseteq S,\ |S| = n/2,\ |c_j| \leq 3,$$

create labeled points in n-dimensional space as follows.

- Label the origin '+' as before.
- For each s_i, put a '−' point at the location with 1's in bits $2i-1$ and $2i$ and 0's in all other bits. We will call the bits $2i-1$ and $2i$ the bits "corresponding" to s_i.
- For each subset c_j, there are two cases: $|c_j| = 2$ or $|c_j| = 3$. Create $2^2 = 4$ or $2^3 = 8$ '+' points respectively, such that for each '+' point, exactly one of the two bits corresponding to each $s_i \in c_j$ is 1.

For example, consider $S = \{s_1, s_2, s_3\}$, $C = \{c_1, c_2\}$, $c_1 = \{s_1, s_2\}$ and $c_2 = \{s_2, s_3\}$. Then the '−' points are: (110000), (001100), (000011) and the '+' points are: (000000), (101000), (100100), (011000), (010100), (001010), (001001), (000110), (000101).

We will also need a "gadget" as we did to prove Theorem 1, in order to force the planes to have all '+' points in one region and the '−' points in the others. This "gadget" is essentially the same as in the proof of Theorem 1. In six new dimensions,

put points labeled '+' at locations: $(0\cdots0\ 001111), (0\cdots0\ 110011)$
and points labeled '−' at locations: $(0\cdots0\ 110000), (0\cdots0\ 001100), (0\cdots0\ 000011), (0\cdots0\ 111111)$

where the bits in the n old dimensions are zero. That is, we replace each bit in the old gadget by two in the new gadget.

Claim 1: Given a solution for an instance of the Set-Splitting problem, we can find parallel hyperplanes with coefficients in $\{-1, +1\}$ that separate the '+' and '−' points.

Proof: Given S_1, create the plane P_1: $a_1x_1 + \cdots + a_nx_n = -1$, where $a_{2i-1} = a_{2i} = -1$ if $s_i \in S_1$ and $a_{2i-1} = a_{2i} = +1$ if $s_i \notin S_1$.

Note that for all '−' points corresponding to $s_i \in S_1$, $a_1x_1 + \cdots + a_nx_n = -2$ and for all other '−' points, $a_1x_1 + \cdots + a_nx_n = +2$. For all '+' points, $a_1x_1 + \cdots + a_nx_n \in \{-1, 0, +1\}$ since each c_j has at most three elements of which at least one contributes a "−1" and at least one contributes a "+1". Therefore, the plane P_1 separates exactly the '−' points derived from $s_i \in S_1$ from the '+' points since for all '+' points, $a_1x_1 + \cdots + a_nx_n \geq -1$ and for all '−' points corresponding to $s_i \in S_1$, we have $a_1x_1 + \cdots + a_nx_n < -1$. Define the second plane analogously.

To correctly "slice" the gadget, for one plane let the coefficients $a_{n+1}, \ldots, a_{n+6}$ in dimensions $n+1, \ldots, n+6$ respectively be $-1, -1, -1, -1, +1, +1$, and for the other plane, let the coefficients be $+1, +1, +1, +1, -1, -1$. One can just "plug in" the 6 gadget points to see that this works.

Planes P_1 and P_2 are parallel since the coefficients $a_1, \ldots, a_{n+6}$ of plane P_1 are just the negation of the corresponding coefficients of plane P_2. ■

Claim 2: Given splitting planes (not necessary parallel, any coefficients allowed) we can find a solution to the Set-Splitting instance.

Part 1: The gadget cannot be split with the '$-$' points all in one quadrant.

Proof: Exactly the same as for the reduction in the proof of Theorem 1.

Part 2: A single plane cannot have all '$-$' points corresponding to a subset c_j on one side and all '+'s on the other.

Proof: Suppose one did. Given a plane $a_1x_1 + \cdots + a_nx_n = a_0$, without loss of generality assume that for the '+' points, $a_1x_1 + \cdots + a_nx_n > a_0$, and that for all the '$-$' points corresponding to the elements of c_j, we have $a_1x_1 + \cdots + a_nx_n \leq a_0$. Since the origin is a '+' point, we know that a_0 must be negative.

For each $s_i \in c_j$, since s_i has 1's in bits $2i-1$ and $2i$, we have $a_{2i-1} + a_{2i} \leq a_0$ which implies that either $a_{2i-1} \leq \frac{1}{2}a_0$ or $a_{2i} \leq \frac{1}{2}a_0$ (or both). Therefore, if $|c_j| = 2$, then at least one of the '+' points corresponding to c_j will have 1's in bits i_1 and i_2 for which $a_{i_1}, a_{i_2} \leq \frac{1}{2}a_0$ and thus will force $a_1x_1 + \cdots + a_nx_n \leq 2 \times \frac{1}{2}a_0$. If $|c_j| = 3$, then at least one of the '+' points corresponding to c_j will force $a_1x_1 + \cdots + a_nx_n \leq 3 \times \frac{1}{2}a_0$. This presents a contradiction since we assumed that for all the '+' points, we had $a_1x_1 + \cdots + a_nx_n > a_0$ (recall, a_0 is negative). ■

APPENDIX B

Proof of Theorem 3: Given an instance of Set-Splitting on n elements, we create training examples of length $n+2$ (alternately '+' and '$-$' points in $(n+2)$-dimensional space) as follows.

1. Create labeled points as in the reduction in the proof of Theorem 1 (except we have added two extra dimensions):
   - Let the origin be labeled '+'.
   - For each $s_i \in S$, put a '$-$' point at $p_i = (\overset{1}{0}\overset{2}{0}\cdots\overset{i}{1}\cdots\overset{n+2}{0})$.
   - For each $c_j = \{s_{j_1}, \ldots, s_{j_k}\}$, put a '+' point at $p_{j_1} + \ldots + p_{j_k}$.

   Note that all these points created have zeros in bits $n+1$ and $n+2$.
2. For each of $r-2$ hidden nodes in the network, we will create labeled points as follows.
   - Choose any arbitrary empty (unlabeled) position in $\{0,1\}^{n+2}$ with 1's in bits $n+1$ and $n+2$ such that the total number of 1's in the vector for that position is odd and put a '$-$' point there. For example, we might pick position: 0110010011 (if n were 8).
   - Label all neighbors of (all positions differing in exactly one bit from) that '$-$' point as '+' points.

For each '$-$' point p created in created in step 2, there must be some plane that separates it from the '+' points. Since all the neighbors of p are labeled '+', a separating plane will have p on one side and the rest of the $(n+2)$-dimensional hypercube of the other. Thus, only two planes remain to separate the '$-$' points created in step 1 from the '+' points. The proof of Theorem 1 shows that two planes that separate these '$-$' points from the '+' points will yield a solution to the Set-Splitting instance.

Given a solution to the Set-Splitting instance, we can create r hyperplanes that separate the '+' and '$-$' points with all the '+' points in one region (which we want since the output node computes the AND function) by using $r-2$ hyperplanes to separate the '$-$' points created in step 2 and two planes to separate those from step 1. The two planes that separate the '$-$' points created in step 1 from the rest of the hypercube are formed exactly as in the proof of Theorem 1 except that the coefficients in dimensions $n+1$ and $n+2$ are large positive integers ($a_{n+1} = a_{n+2} = n$) so that all the '+' points from step 1 are in the same region as the '+' points from step 2.

We can handle up to $2+2^{n-1}$ hyperplanes (hidden nodes), and therefore certainly any fixed polynomial in n of them as n becomes large, using about as many labeled points (training examples) as the total number of weights in the network. ■