On the Complexity of Learning with a Small Number of Nodes

Ian Parberry*
Center for Research in Parallel
and Distributed Computing
Department of Computer Sciences
University of North Texas

Abstract

It is shown that the loading problem for a 6 node neural network with node function set $\mathcal{AC}_1^0$ (that is, the conjunction or disjunction of a subset of the inputs or their complements) is $\mathcal{NP}$ complete. It can be deduced from this observation that the loading problem for a 6 node analog neural network is $\mathcal{NP}$ hard.

1 Introduction

Judd [5, 6, 7, 8] has shown that the problem of loading simple tasks onto neural networks with a fixed architecture is $\mathcal{NP}$ complete, which implies that there is quite likely to be no fast learning algorithms even for quite simple architectures and node function sets. Surprisingly, the loading problem is $\mathcal{NP}$ complete even for networks of size 3 if the node function set is the set of linear threshold functions (Blum and Rivest [1, 2]).

One weakness of the result of Blum and Rivest is that it holds only for node function sets that are *exactly* weighted linear threshold functions. Judd's techniques work for any node function set that includes $\mathcal{AC}_1^0$ and hence apply to analog neural networks, but he has no results for a fixed number of nodes. We show a result that is as general as Judd's, but has a fixed node bound. Specifically, we show that the loading problem for a 6 node neural network with node function set $\mathcal{AC}_1^0$ is $\mathcal{NP}$ complete. We deduce from this that the loading problem for a 6 node analog neural network is $\mathcal{NP}$ hard.

To simplify the presentation, we first show that the loading problem for a 4 node neural network with node function set equal to $\mathcal{AC}_1^0$ plus the three-input equality function (the Boolean function that outputs 1 iff its three inputs are identical) is $\mathcal{NP}$ complete, and then indicate how the required results can be derived in a similar, but less convenient fashion.

*Author's address: Department of Computer Sciences, University of North Texas, P.O. Box 13886, Denton, TX 76203-3886, U.S.A. Electronic mail: ian@ponder.csci.unt.edu.

We assume that the reader is familiar with the rudiments of the theory of $\mathcal{NP}$ completeness. The interested reader can consult a standard text such as Garey and Johnson [4].

The remainder of this paper is divided into three short sections. The first describes the loading problem in more technical detail, the second contains the main results, and the third concludes and gives open problems.

2 The Loading Problem

An *architecture* is a 4-tuple $A = (V, X, Y, E)$, where

V is a finite ordered set
X is a finite ordered set such that $X \cap V = \emptyset$
$Y \subseteq V$
$(V \cup X, E)$ is a directed, acyclic graph.

The set V represents a set of *nodes*, X represents a set of *inputs*, and Y represents a set of *outputs*. E represents the connections between the nodes, the inputs, and the outputs. The graph $(V \cup X, E)$ is called the *interconnection pattern* of C.

A *neural network* is a 3-tuple $C = (A, \mathcal{F}, \ell)$, where A is an architecture, $\mathcal{F}$ is a set of functions with domain $\mathbb{B}^*$ and range $\mathbb{B}$, and $\ell : V \to \mathcal{F}$. Each node $v \in V$ computes a Boolean function from the *node function set* $\mathcal{F}$ determined by the *node function assignment* ℓ.

Let $C = (A, \mathcal{F}, \ell)$ be a neural network, where $A = (V, X, Y, E)$, $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$. For each input $b_1, \ldots, b_n \in \mathbb{B}$, and each $g \in V \cup X$, define the *value* of node g on input $b_1, \ldots, b_n$, denoted $v(g)$, as follows. If $g = x_i$ for some $1 \leq i \leq n$, then $v(g) = b_i$. If $g \in V$, g has in-degree m, $(g_i, g) \in E$ for $1 \leq i \leq m$, and $\ell(g) = f$, then $v(g) = f(g_1, \ldots, g_m)$. The *output* of C on inputs $b_1, \ldots, b_n \in \mathbb{B}$ is defined to be $v(y_1), \ldots, v(y_m)$.

Suppose $A = (V, X, Y, E)$ is an architecture with $\|X\| = n$ and $\|Y\| = m$. A *task* for A is an element of $\mathbb{B}^n \times (\mathbb{B} \cup \{*\})^m$. A *task set* is a set of tasks. Intuitively, the first component of a task is an input string and the second component is the desired output string, where the starred values are "don't-cares". If $\ell : V \to \mathcal{F}$, the neural network corresponding to A and ℓ is given by $A(\ell) = (A, \mathcal{F}, \ell)$. A neural network $A(\ell)$ is said to *support* task $(x_1 \cdots x_n, y_1 \cdots y_m)$ if the output of $A(\ell)$ on input $(x_1, \ldots, x_n)$ is $(z_1, \ldots, z_m)$, where for all $1 \leq i \leq m$, if $y_i \in \mathbb{B}$, then $z_i = y_i$. A neural network $A(\ell)$ is said to *support* a task set if it supports every task in it. The *loading problem* for node function set $\mathcal{F}$ is the following problem: given an architecture A and a task set T, find a node function assignment $\ell : V \to \mathcal{F}$ such that $A(\ell)$ supports T.

The loading problem is clearly computable (simply try all combinations of functions from the node function set), but is it feasibly computable? That is, is there a polynomial time algorithm that solves it? If there were, then there would be one for the following decision problem:

The Loading Problem
INSTANCE: An architecture A and a task set T
QUESTION: Is there a node function assignment ℓ such that $A(\ell)$ supports T?

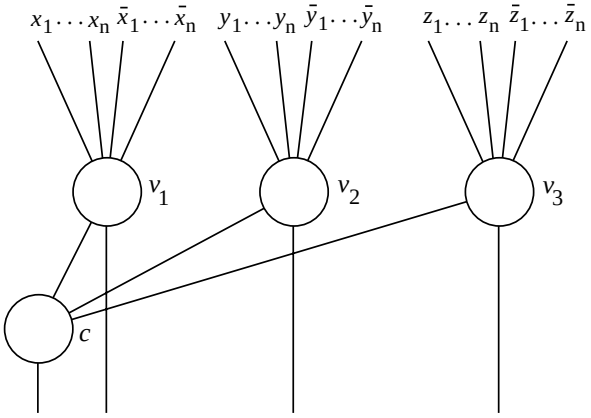

Figure 1: The architecture corresponding to an instance of not-all-equal 3SAT with n variables and m clauses in Theorem 3.1.

3 The Main Result

Theorem 3.1 *The loading problem for neural networks of 4 nodes is $\mathcal{NP}$ complete.*

PROOF: It is clear that the loading problem for neural networks of 4 nodes is a member of $\mathcal{NP}$. It remains to show that it is $\mathcal{NP}$ hard. We will show that not-all-equal 3SAT reduces in polynomial time to the loading problem for neural networks of 4 nodes. Since not-all-equal 3SAT is $\mathcal{NP}$ complete (Schaefer [10]), this is sufficient to show that our loading problem is $\mathcal{NP}$ hard. Not-all-equal 3SAT is defined as follows:

> **Not-all-equal 3SAT**
> INSTANCE: A set of clauses with 3 literals per clause.
> QUESTION: Is there an assignment of values to the variables such that no clause has all of its literals assigned the same value?

For each variable x, let $x[0]$ denote $\overline{x}$, $x[1]$ denote x, $\overline{x}[0]$ denote x, and $\overline{x}[1]$ denote $\overline{x}$. Suppose we are given an instance of not-all-equal 3SAT, $C = \{C_i \mid 1 \leq i \leq m\}$ for some $m \in \mathbb{N}$, where each C_i is a set of three literals over the variables $x_1, \ldots, x_n$. Suppose for all $1 \leq j \leq m$, that clause C_j uses variables x_{j_k} for $1 \leq k \leq 3$.

The architecture corresponding to an instance of not-all-equal 3SAT with n inputs and m clauses is $A = (V, X, Y, E)$, where (see Figure 1)

$$\begin{aligned}
V &= \{v_1, v_2, v_3, c\} \\
X &= \{x_j \mid 1 \leq i \leq n\} \cup \{y_j \mid 1 \leq i \leq n\} \cup \{z_j \mid 1 \leq i \leq n\} \cup \\
&\quad \{\overline{x}_j \mid 1 \leq i \leq n\} \cup \{\overline{y}_j \mid 1 \leq i \leq n\} \cup \{\overline{z}_j \mid 1 \leq i \leq n\} \\
Y &= V \\
E &= \{(x_i, v_1), (\overline{x}_i, v_1), (y_i, v_2), (\overline{y}_i, v_2), (z_i, v_3), (\overline{z}_i, v_3) \mid 1 \leq i \leq n\} \cup \\
&\quad \{(v_i, c) \mid 1 \leq i \leq 3\}.
\end{aligned}$$

We will construct a set of tasks that force A to behave as follows. Inputs $(x_1, \ldots, x_n)$, $(y_1, \ldots, y_n)$, and $(z_1, \ldots, z_n)$ will each encode a variable number in unary. Node v_i will be forced to output 0 when all of its inputs are 0, and to output 1 when all of its inputs are 1. Node c will be forced to output 1 iff the three values output by v_1, v_2, and v_3 are identical. Note that since node c does not have direct access to the inputs, it will be forced to compute that function regardless of the value of the inputs. Nodes v_1, v_2, and v_3 will also be forced to output a value for each literal when the index of that literal is encoded in their inputs. A set of tasks will ensure consistency of variables between the nodes (that is, when v_1, v_2, and v_3 are asked about the same literal, they will output the same value). Another set of tasks will ensure consistency of literals within the nodes (that is, when each one of v_1, v_2, and v_3 is asked about a variable and its complement, it will output different values). Additional tasks will force the neural network to solve not-all-equal 3SAT.

For convenience, we will use 0^n as a shorthand for the string of n zeros, and 1^n as a shorthand for the string of n ones. For all $n \in \mathbb{N}$, let $\delta_n^0, \delta_n^1 : \mathbb{N} \to \mathbb{B}^{2n}$ be defined as follows:

$$\begin{aligned} \delta_n^0(i) &= 0^n 0^{n-i} 1 0^{i-1} \\ \delta_n^1(i) &= 0^{n-i} 1 0^{i-1} 0^n. \end{aligned}$$

The first set of tasks are called *equality enforcers*. These force v_i to output 0 when all of its inputs are 0 and to output 1 when all of its inputs are 1, for $1 \leq i \leq 3$, and force c to output 1 iff the three values it sees are identical. Equality enforcers have the form $T_1(a, b, c, d) = (a^n a^n b^n b^n c^n c^n, dabc)$, where $a, b, c, d \in \mathbb{B}$. We use equality enforcers:

$$\begin{array}{ll} T_1(0,0,0,1) & T_1(1,0,0,0) \\ T_1(0,0,1,0) & T_1(1,0,1,0) \\ T_1(0,1,0,0) & T_1(1,1,0,0) \\ T_1(0,1,1,0) & T_1(1,1,1,1). \end{array}$$

The second class of tasks are called *consistency enforcers*. These tasks ensure that the value chosen for each literal by v_1, v_2, and v_3 is the same for each node. This is achieved with tasks of the following form, for $1 \leq i \leq n$:

$$(\delta_n^0(i)\delta_n^0(i)\delta_n^0(i), 1***) \quad (\delta_n^1(i)\delta_n^1(i)\delta_n^1(i), 1***).$$

The third class of tasks are called *complement enforcers*. These tasks ensure that nodes v_1, v_2, and v_3 correctly complement their variables when called upon to do so. These tasks have the following form, for $1 \leq i \leq n$:

$$\begin{array}{ll} (\delta_n^0(i)\delta_n^0(i)\delta_n^1(i), 0***) & (\delta_n^0(i)\delta_n^1(i)\delta_n^0(i), 0***) \\ (\delta_n^0(i)\delta_n^1(i)\delta_n^1(i), 0***) & (\delta_n^1(i)\delta_n^0(i)\delta_n^0(i), 0***) \\ (\delta_n^1(i)\delta_n^0(i)\delta_n^1(i), 0***) & (\delta_n^1(i)\delta_n^1(i)\delta_n^0(i), 0***). \end{array}$$

The fourth set of tasks are called *computation tasks*. These tasks ensure that C is satisfiable. For each clause $C_j = \{x_{j_1}[\alpha_j], x_{j_2}[\beta_j], x_{j_3}[\gamma_j]\}$ (where $\alpha_j, \beta_j, \gamma_j \in \mathbb{B}$), we have a task

$$(\delta_n^{\alpha_j}(j_1)\delta_n^{\beta_j}(j_2)\delta_n^{\beta_j}(j_3), 0***).$$

Task Type	*Tasks*	
Equality	(0000 0000 0000 0000 0000 0000 , 1 000)	(0000 0000 0000 0000 1111 1111 , 0 001)
	(0000 0000 1111 1111 0000 0000 , 0 010)	(0000 0000 1111 1111 1111 1111 , 0 011)
	(1111 1111 0000 0000 0000 0000 , 0 100)	(1111 1111 0000 0000 1111 1111 , 0 101)
	(1111 1111 1111 1111 0000 0000 , 0 110)	(1111 1111 1111 1111 1111 1111 , 1 111)
Consistency	(0000 0001 0000 0001 0000 0001 , 1 ∗∗∗)	(0001 0000 0001 0000 0001 0000 , 1 ∗∗∗)
	(0000 0010 0000 0010 0000 0010 , 1 ∗∗∗)	(0010 0000 0010 0000 0010 0000 , 1 ∗∗∗)
	(0000 0100 0000 0100 0000 0100 , 1 ∗∗∗)	(0100 0000 0100 0000 0100 0000 , 1 ∗∗∗)
	(0000 1000 0000 1000 0000 1000 , 1 ∗∗∗)	(1000 0000 1000 0000 1000 0000 , 1 ∗∗∗)
Complement	(0000 0001 0000 0001 0001 0000 , 0 ∗∗∗)	(0000 0001 0001 0000 0000 0001 , 0 ∗∗∗)
	(0000 0001 0001 0000 0001 0000 , 0 ∗∗∗)	(0001 0000 0000 0001 0000 0001 , 0 ∗∗∗)
	(0001 0000 0000 0001 0001 0000 , 0 ∗∗∗)	(0001 0000 0001 0000 0000 0001 , 0 ∗∗∗)
	(0000 0010 0000 0010 0010 0000 , 0 ∗∗∗)	(0000 0010 0010 0000 0000 0010 , 0 ∗∗∗)
	(0000 0010 0010 0000 0010 0000 , 0 ∗∗∗)	(0010 0000 0000 0010 0000 0010 , 0 ∗∗∗)
	(0010 0000 0000 0010 0010 0000 , 0 ∗∗∗)	(0010 0000 0010 0000 0000 0010 , 0 ∗∗∗)
	(0000 0100 0000 0100 0100 0000 , 0 ∗∗∗)	(0000 0100 0100 0000 0000 0100 , 0 ∗∗∗)
	(0000 0100 0100 0000 0100 0000 , 0 ∗∗∗)	(0100 0000 0000 0100 0000 0100 , 0 ∗∗∗)
	(0100 0000 0000 0100 0100 0000 , 0 ∗∗∗)	(0100 0000 0100 0000 0000 0100 , 0 ∗∗∗)
	(0000 1000 0000 1000 1000 0000 , 0 ∗∗∗)	(0000 1000 1000 0000 0000 1000 , 0 ∗∗∗)
	(0000 1000 1000 0000 1000 0000 , 0 ∗∗∗)	(1000 0000 0000 1000 0000 1000 , 0 ∗∗∗)
	(1000 0000 0000 1000 1000 0000 , 0 ∗∗∗)	(1000 0000 1000 0000 0000 1000 , 0 ∗∗∗)
Computation	(0001 0000 0010 0000 0000 0100 , 0 000)	(0001 0000 0010 0000 0000 1000 , 0 000)
	(0000 0001 0000 0010 0100 0000 , 0 000)	(0010 0000 0000 0100 1000 0000 , 0 000)
	(0000 0001 0100 0000 1000 0000 , 0 000)	

Table 1: Tasks for the architecture shown in Figure 1 and not-all-equal 3SAT instance $\{(x_1, x_2, \overline{x}_3), (x_1, x_2, \overline{x}_4), (\overline{x}_1, \overline{x}_2, x_3), (x_2, \overline{x}_3, x_4), (\overline{x}_1, x_3, x_4).\}$

It is easy to devise a polynomial time algorithm circuit that constructs A and the task set from any given instance of not-all-equal 3SAT C. It remains to show that C is satisfiable iff A supports the tasks.

Suppose $b_1, \ldots, b_n$ is a satisfying assignment for C. It can be verified by inspection that A supports the equality, consistency, complement, and computation tasks with the following node function assignments. Let $\mathcal{X} = \{i \mid b_i = 1\}$. The node v_1 computes the disjunction of $\{x_i \mid i \in \mathcal{X}\} \cup \{\overline{x}_i \mid i \notin \mathcal{X}\}$. Nodes v_2 and v_3 compute similar functions, replacing x by y and z respectively. Node c computes the equality function.

Conversely, if A supports the tasks above, then by construction C is satisfied by setting x_i to the output of v_1 on input $\delta_n^1(i)$. □

For example, the task set for the instance of not-all-equal 3SAT

$$\{(x_1, x_2, \overline{x}_3), (x_1, x_2, \overline{x}_4), (\overline{x}_1, \overline{x}_2, x_3), (x_2, \overline{x}_3, x_4), (\overline{x}_1, x_3, x_4).\}$$

is shown in Table 1.

By inspection, Theorem 3.1 holds for any node function set that includes all functions in $\mathcal{AC}_1^0$ (that is, either the conjunction or disjunction of a subset of the inputs or their complements) and the 3-input equality function. Is this a reasonable node function set?

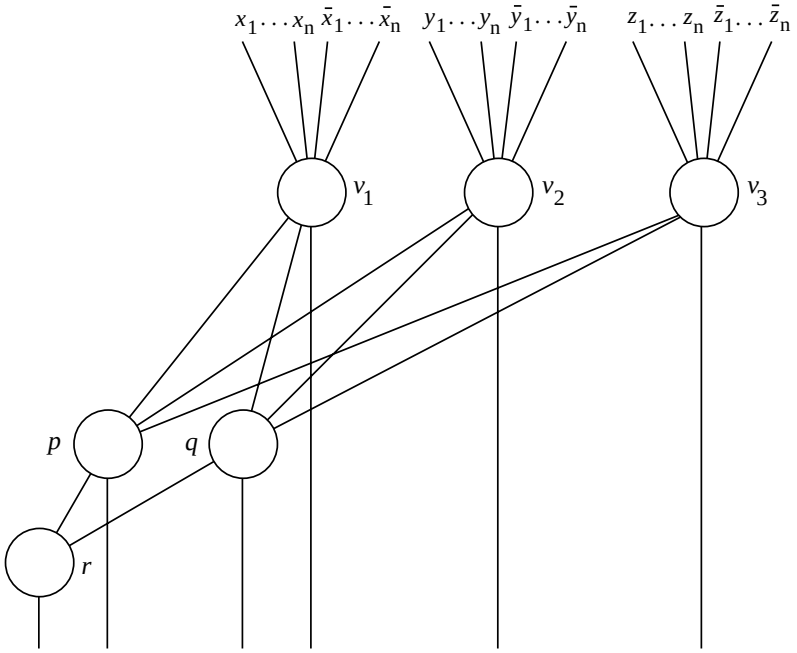

Figure 2: The architecture corresponding to an instance of not-all-equal 3SAT with n variables and m clauses in Theorem 3.2.

Judd [5, 6, 7, 8] concentrates on node function sets at least as powerful as $\mathcal{AC}_1^0$. The argument of Blum and Rivest [1] works *only* for node functions that are weighted linear threshold functions (which includes $\mathcal{AC}_1^0$). Our node function set is stronger then that of Judd, but weaker than that of Blum and Rivest in the sense that no constant depth, polynomial size circuit of functions drawn from our node function set can compute weighted linear threshold functions (this is a simple consequence of Furst, Saxe, and Sipser [3]). Nonetheless, one can prove the following:

Theorem 3.2 *The loading problem for neural networks of 6 nodes with node function set $\mathcal{AC}_1^0$ is $\mathcal{NP}$ complete.*

PROOF: The proof is similar to that of Theorem 3.1, replacing node c in architecture A by a 3-node subcircuit (see Figure 2). The tasks force this subcircuit to compute the same function as c. □

It is popular to study *analog neural networks*, in which the nodes compute functions of the form $f:(0,1)^k \to (0,1)$ (where $(0,1)$ denotes $\{r \in \mathbb{R} \mid 0 < r < 1\}$) defined by

$$f(x_1, \ldots, x_k) = g(\sum_{i=1}^{k} w_i x_i),$$

where $w_i \in \mathbb{R}$ for $1 \leq i \leq k$, and $g:\mathbb{R} \to (0,1)$ is a smooth, monotonic increasing function with the property that $\lim_{n\to\infty} g(n) = 1$ and $\lim_{n\to-\infty} g(n) = 0$. For example, Rumelhart, Hinton, and Williams [9] use $g(x) = 1/(1+e^{-x})$.

Theorem 3.3 *The loading problem for analog neural networks of 6 nodes is $\mathcal{NP}$ hard.*

PROOF: The proof is similar to that of Theorem 3.2, using techniques similar to those of Judd [8, Appendix B]. □

4 Conclusion and Open Problems

How does one interpret the meaning of $\mathcal{NP}$ completeness results for the loading problem? They imply that any learning algorithm that takes as input a set of tasks and a fixed architecture runs the risk of taking exponential time in the worst case even for architectures drawn from quite innocuous architecture classes. This can be avoided either by limiting the node function set and choosing specific architectures for which the loading problem is not intractable, by allowing the architecture to change during learning, or by only loading task sets that do not cunningly encode $\mathcal{NP}$ complete problems. Results such as those in this paper indicate that even very simple architectures and node function sets can have task sets that encode $\mathcal{NP}$ complete problems. This can be a major pitfall for the unwary neural network designer. Some open questions remaining are whether the three loading problems studied in this paper remain $\mathcal{NP}$ complete with fewer nodes.

References

[1] A. Blum and R. L. Rivest. Training a 3-node neural network is NP-complete. In *Neural Information Processing Systems 1*, pages 494–501. Morgan Kaufmann, 1989.

[2] A. Blum and R. L. Rivest. Training a 3-node neural network is NP-complete. *Neural Networks*, 5(1):117–127, 1992.

[3] M. Furst, J. B. Saxe, and M. Sipser. Parity, circuits and the polynomial time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[5] J. S. Judd. Learning in networks is hard. In *Proc. of the First International Conference on Neural Networks*, pages 685–692. IEEE Computer Society Press, 1987.

[6] J. S. Judd. *Neural Network Design and the Complexity of Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 1988.

[7] J. S. Judd. On the complexity of loading shallow neural networks. *Journal of Complexity*, 4:177–192, 1988.

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

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, 1986.

[10] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 216–226. ACM Press, 1978.