On dynamic programming-like recursive gradient formula for alleviating MLP hidden-node saturation in the parity problem

Eiji Mizutani [1], *Stuart E. Dreyfus* [2], *and Jyh-Shing Roger Jang* [1]

eiji@wayne.cs.nthu.edu.tw, dreyfus@ieor.berkeley.edu, jang@cs.nthu.edu.tw

1) Dept. of Computer Science, National Tsing Hua University, Hsinchu 300 Taiwan

2) Dept. of Industrial Engineering and Operations Research, Univ. of California at Berkeley, CA 94720, USA

Keywords

Hidden-node teaching, parity problem, dynamic programming-like recursive gradient formula

Abstract

This paper addresses saturation phenomena at hidden nodes during the learning phase of neural networks. The hidden-node saturation tends to cause a "plateau," a region of very little or no change in a graphic representation of the error learning curve. We investigate the saturation phenomena in multilayer perceptrons (MLP) with the well-known neural-network benchmark problem: the parity problem, describing how to augment their learning capacity to solve it perfectly by avoiding the hidden-node saturation in conjunction with a dynamic programming-like recursive gradient formula. To make the parity problem especially challenging, we first show that the seven-bit parity problem can in principle be solved using only four hidden nodes and we then compare various learning algorithms using this MLP architecture with emphasis on whether or not each algorithm can solve the problem "perfectly" regardless of its learning speed. In particular, we highlight an online-mode steepest descent-type learning with **hidden-node teaching**, as well as a batch-mode direct **dog-leg trust-region** algorithm.

1 Introduction

The **neural networks nonlinear least squares problem** * is an optimization problem to find the n weight parameters of *neural networks* (NN) [e.g., *multilayer perceptrons* (MLP)], denoted by an n-dimensional vector $\boldsymbol{\theta}$, that minimize the sum of squared errors. In *standard* MLP-learning, the hidden-node activations are usually produced by sigmoidal "squashing" functions (e.g., hyperbolic tangent functions), which act as *implicit constraints* because the net inputs to hidden nodes may get driven to their limits: i.e., **saturation** (e.g., +1.0 or -1.0). Therefore, even with a "sophisticated" *unconstrained* optimization technique (e.g., dogleg algorithms), NN learning might fail due to saturation (implicit constraint). In consequence, the aforementioned *neural networks nonlinear least squares problem* is not a totally *unconstrained*, but rather an **implicitly constrained optimization** problem.

To circumvent hidden-node saturation, two different approaches can be considered: **indirect approach** and **direct approach**. Many existing NN-learning algorithms can be viewed as the *indirect* approach, † which attempts to avoid hidden-node saturation "indirectly" by using the information of an approximate Hessian matrix (e.g., quasi-Newton algorithms [1]). In contrast, our *direct* approach is to *present certain features or characteristics associated with the desired target outputs to (a subset of) hidden nodes* , which is what we call **hidden-node teaching** [2]. For this purpose, the MLP parameter optimization is viewed as a multi-stage decision making problem (see Figure 1) with the following *nonlinear dynamics*, ordinary MLP *activations*:

$$\begin{aligned} a_j^s &= f_j^s\left(\textstyle\sum_{i=1}^{n_{s-1}} a_i^{s-1}\theta_{ij}^{s-1}\right) \\ &= f_j^s(net_j^s), \end{aligned}$$

where a_j^s is the activation of node j at layer s; n_{s-1} is the # of nodes of layer s-1; θ_{ij}^{s-1} is the weight parameter of node i at layer s-1 to node j at layer s; $f_j^s(.)$ is the neuron function of node j at layer s; and net_j^s denotes the **net input** to node j at layer s. Furthermore, the objective cost function has the following **discrete optimal control** form:

$$E(\boldsymbol{\theta}) = \sum_{s=1}^{N-1} g^s(\mathbf{a}^s, \boldsymbol{\theta}^s) + h(\mathbf{a}^N), \qquad (1)$$

*We focus on "nonlinear least squares algorithms" that exploit the special structure of the sum of squared error measure; hence, the other objective functions (e.g., logarithmic error functions) are outside the scope of this paper.

†There are other indirect approaches; such as, heuristic parameter initialization, modifications of NN structures and node functions. These are outside the scope of this paper.

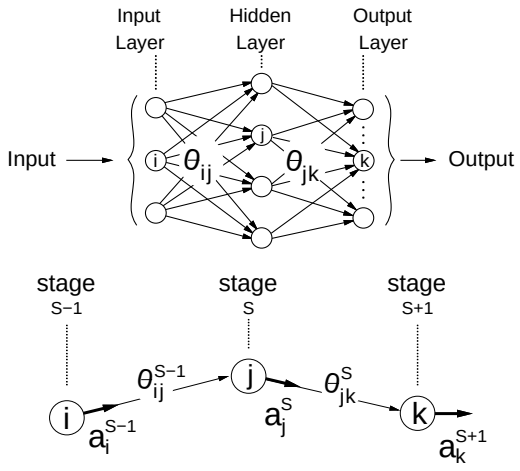

Figure 1: *A typical MLP structure (top) with anatomical view (bottom) of connections of neurons i, j, and k at different layers;* θ^s_{jk} *denotes the weight of neuron j at layer s to neuron k at layer (s+1).*

where, in conformity with the optimal control theory, $\mathbf{a}^s$ and $\boldsymbol{\theta}^s$ signify the *state vector* and *decision vector* at stage s, and $g^s(.,.)$ and $h(.)$ denote the *immediate cost per stage s* and the *terminal cost*, respectively.

In the MLP-learning, $g^s(.,.)$ is usually omitted in Equation (1), and the sum of squared errors is often employed as the terminal cost function $h(.)$ alone at the final layer N, which consists of n_N neurons:i.e.,

$$\begin{aligned} h(\mathbf{a}^N) &= \tfrac{1}{2}\sum_{p=1}^{m}\sum_{k=1}^{n_N}(t^N_{k,p}-a^N_{k,p})^2 \\ &= \tfrac{1}{2}\sum_{k=1}^{n_N}(\mathbf{t}^N_k-\mathbf{a}^N_k)^2 \\ &= \tfrac{1}{2}\mathbf{r}^T_N(\boldsymbol{\theta})\mathbf{r}_N(\boldsymbol{\theta}), \end{aligned} \tag{2}$$

where $t^N_{k,p}$ is the kth desired output for the pth training data pattern at the terminal layer N; $a^N_{k,p}$ is the kth activation (i.e., output of the kth neuron function f) for the pth data; $\mathbf{t}^N_k$ is the desired output vector; $\mathbf{a}^N_k$ is the activation vector; and $\mathbf{r}_N(\boldsymbol{\theta})$ denotes the residual vector composed of $r_i(\boldsymbol{\theta})$, $i = 1, \ldots, mn_N$.

In this paper, we consider a case where the immediate cost g at stage s is also the squared-error cost measure:

$$\begin{aligned} g^s(\mathbf{a}^s,-) &= \tfrac{1}{2}\sum_{p=1}^{m}\sum_{k=1}^{n_s}(t^s_{k,p}-a^s_{k,p})^2 \\ &= \tfrac{1}{2}\sum_{k=1}^{n_s}(\mathbf{t}^s_k-\mathbf{a}^s_k)^2. \end{aligned} \tag{3}$$

Note in this setup that $g^s(\mathbf{a}^s,\boldsymbol{\theta}^s)$ becomes independent of $\boldsymbol{\theta}^s$; hence, denoted by $g^s(\mathbf{a}^s,-)$. In words, $g^s(\mathbf{a}^s,-)$ is the immediate cost at stage s, or at the sth (hidden) layer, between the (hidden) activations $\mathbf{a}^s$ and the desired values $\mathbf{t}^s$. Those desired values can be chosen so as to pull hidden node activations away from saturation values (e.g., +1.0 or -1.0) for such a classification problem as the parity problem. The n_s neurons at the sth hidden layer are supervised and expected to produce specific target values $\mathbf{t}^s$; that is, *hidden-node teaching* [‡].

In the next section, we employ **dynamic programming** (**DP**)-like recursive gradient formula to derive the **backpropagation** "generalized delta rule" for the objective function in Equation (1). We then describe the parity problem, in which hidden-node saturation often occurs, and demonstrate "hidden-node teaching" for solving a seven-bit parity problem perfectly using the standard MLP with only four hidden nodes.

2 DP-like Recursive Gradient Formula

Backpropagation allows us to compute the partial-derivative form of **sensitivity** backward one stage after another from the terminal stage. This BP formulation dates back to a classical optimal control theory, although similar formulations were derived by several individuals independently (see [2] and references therein). The next table summerizes this BP history:

1985	D. Rumelhart et al.	Backpropagation
1974	P. Werbos	Sensitivity formula
1962	S. Dreyfus	DP-like recursive gradient formula
1961	A. Bryson	Multiplier Rule
1960	H. Kelley	Adjoint Equations

The next table compares terminologies between the optimal control literature and neural networks:

Notation	Optimal Control	Neural Network
θ	decision variable	weight parameter
a	state variable	(neuron) activation
s	stage	layer
δ	adjoint variable Lagrange multiplier costate	delta

Using these notations, we can derive BP for the objective cost function in Equation (1) in the spirit of classical *dynamic programming* [3]:

Value function:

$$T^s(\mathbf{a}^s,\boldsymbol{\theta}^s) \stackrel{\text{def}}{=} \text{cost-to-go, starting at state } \mathbf{a}^s \text{ at stage s, using a guessed policy } \boldsymbol{\theta}^s. \tag{4}$$

The initial guessed policy is often given by a randomly-initialized weight-parameter set.

[‡]It should be noted that this hidden-node teaching is different from what BPTT (backpropagation through time) algorithm does, because BPTT attempts to update the entire MLP at each discretized time tick, which corresponds to "stage," and thus it deals with a completely different model.

Recurrence relation:

$$\begin{aligned} T^s(\mathbf{a}^s, \boldsymbol{\theta}^s) &= T^{s+1}(\mathbf{a}^{s+1}, \boldsymbol{\theta}^{s+1}) + g^s(\mathbf{a}^s, -) \\ &= T^{s+1}(f^{s+1}(\mathbf{net}^{s+1}), \boldsymbol{\theta}^{s+1}) + g^s(\mathbf{a}^s, -), \end{aligned} \tag{5}$$

where $g(.,.)$ denotes the immediate cost, as shown in Equation (3).

Boundary condition:

$$T^N(\mathbf{a}^N, -) = h(\mathbf{a}^N) = \frac{1}{2}\sum_{k=1}^{n_N}(\mathbf{t}_k^N - \mathbf{a}_k^N)^2, \tag{6}$$

which corresponds to Equation (2).

In this setting, the **sensitivity** δ at the terminal stage is given by

$$\delta_k^N \stackrel{\text{def}}{=} \frac{\partial T^N}{\partial a_k^N} = -(t_k^N - a_k^N).$$

Departing from this *terminal* value at layer N, the sensitivity δ is backpropagated and computed backward one after another by the the following recurrence relation, our version of the **delta-rule**:

$$\begin{aligned} \delta_j^s &\stackrel{\text{def}}{=} \frac{\partial T^s}{\partial a_j^s} \\ &= \sum_{k=1}^{n_{s+1}} \delta_k^{s+1}\ f^{s+1'}(net_k^{s+1})\ \theta_{jk}^s - (t_j^s - a_j^s). \end{aligned}$$

Of course, the last term (for hidden-node teaching) would be omitted for any node not being supervised. For more details, refer to Mizutani, Dreyfus & Nishio [2].

It should be noted that the best-known BP formulation due to Rumelhart et al. was made by choosing the *net input* but not the *activation* as the *state* variable, resulting in the slightly different delta rule:

$$\delta_j^s = f_j^{s'}(net_j^s)\left[\sum_k^{n_{s+1}} \delta_k^{s+1}\ \theta_{jk}^s - (t_j^s - a_j^s)\right]. \tag{7}$$

3 The Parity Problem and Hidden-node Saturation

One of the most classical NN-benchmark problems is the *parity problem* that occupies an honored place in the history of NN-learning, because of the fact that a single-layer perceptron is unable to solve the XOR (two-bit parity) problem, which caused a hiatus in NN research. But later its multi-layer version (i.e., MLP) has been proved to have great learning capacity to solve the $N(>2)$-bit problem due to backpropagation, an efficient backward derivative-computing process.

We attacked the N-bit parity problem using a *standard single hidden-layer MLP* with "tanh" (hyperbolic tangent) node functions, yielding the intriguing findings:

(a) The **odd-number**-bit parity problem may require **fewer hidden nodes** for solution than the even-number-bit counterpart; for N odd, we shall demonstrate below a solution using only $\frac{N+1}{2}$ hidden nodes;

(b) The number of epochs required to solve the parity problem *perfectly* in light of our criterion [see Equation (8)] may not depend on N;

(c) The "standard" MLP can solve the N-bit parity problem with a smaller number of hidden nodes than N, even if N is even;

(d) Hidden-node saturation becomes more serious as the number of hidden nodes is decreased below N.

These findings are due to our empirical observations through investigation up to the 19-bit parity problem. Finding (a) was inspired by our numerical results that *parameters between the input and hidden layers connecting to the same hidden node tend to become the same value in magnitude*. To get more insight into this finding, consider the seven-bit parity problem and a single hidden-layer MLP with four "tanh" hidden nodes. For simplicity, we assume that all threshold parameters are omitted, and further assume that the parameters between the ith ($i = 1 \cdots 4$) hidden node and all the seven input nodes have the same value denoted by θ_i. The total 128 ($= 2^7$) input patterns where -1 represents "off" and $+1$ represents "on" can be split into eight cases, depending on the number of $+1$'s in the seven-dimensional binary input vector. For the first four cases, where the input vector has (1) seven $+1$'s; (2) six $+1$'s; (3) five $+1$'s; (4) four $+1$'s, we can obtain the following four equations which for arbitrary θ_i are linear in w_i:

$$\begin{cases} f(7\theta_1)w_1 + f(7\theta_2)w_2 + f(7\theta_3)w_3 + f(7\theta_4)w_4 = f^{-1}(a), \\ f(5\theta_1)w_1 + f(5\theta_2)w_2 + f(5\theta_3)w_3 + f(5\theta_4)w_4 = f^{-1}(-a), \\ f(3\theta_1)w_1 + f(3\theta_2)w_2 + f(3\theta_3)w_3 + f(3\theta_4)w_4 = f^{-1}(a), \\ f(\theta_1)w_1 + f(\theta_2)w_2 + f(\theta_3)w_3 + f(\theta_4)w_4 = f^{-1}(-a), \end{cases}$$

where a ($0 < a < 1$) is the required output [that passes the test in Equation (8)] for an odd number of $+1$'s in input and $-a$ for an even number, $f(.)$ denotes the "tanh" node function, and w_i are parameters between the ith hidden node and the final output node. Now we consider the rest of the four cases, where the input vector has (5) three $+1$'s; (6) two $+1$'s; (7) one $+1$; (8) zero $+1$'s. But again we end up the same set of four equations since, for $f(x) = tanh(x)$, $f(-a) = -f(a)$ and $f^{-1}(-a) = -f^{-1}(a)$. This implies that an MLP that correctly classifies the 64 patterns in cases (1) through (4) will classify correctly the 64 patterns in cases (5) through (8) *automatically*. This **symmetric property** holds for

the *odd-number-bit* parity problem, but not for the even-number counterpart. In general, for an N-bit parity problem, for N odd, $\frac{N+1}{2}$ hidden nodes suffice and $\theta_1, \cdots, \theta_{\frac{N+1}{2}}$ can be arbitrary as long as the $\frac{N+1}{2}$ linear equations of the type shown above are independent. Note in the above assumptions that we omitted threshold parameters, but in reality they help the MLP *learn the bit-counting rule*. (Due to the space limitation, the other findings are not detailed here, but for the 14 and 19-bit problems attacked by MLPs with N-1 hidden nodes, refer to Mizutani and Demmel [4].)

In MLP-learning, a **plateau**, *a region of very little or no change in a graphic representation of the error curve* is often encountered, especially when a steepest descent-type algorithm is employed for solving the **parity problem**. Many individuals attributed the *plateau* to **local minima** and/or *inappropriate initial parameters* [§]; however, it may not always be the case (see, for instance, Fukumizu & Amari [5]). In fact, our numerical experience reveals that a sophisticated **deterministic** optimization method that basically gravitates to the nearest local minimum often solves the problem perfectly in spite of starting at exactly the same initial point in the parameter space. Hence, it may be more likely due to **saddle point** issues, which are presumably related to hidden-node saturation.

4 Experiments

Although we actually tested a variety of NN algorithms, this section presents several representative results obtained by the "direct" approach (i.e., the online-mode steepest descent (SD)-type learning with hidden-node teaching), as well as the selected "indirect" approaches; namely, the SD-type learning (without hidden-node teaching) and two representative batch-mode algorithms: Marquardt's algorithm (see Matlab NN toolbox [6]), and direct dogleg algorithms (see Mizutani [7, 8]). Tables 1 and 2 summarize all the results obtained by the "standard" single hidden-layer MLP with its parameters initialized randomly in the range $[-0.3, 0.3]$. Note in our experiments that we say that a "tanh" node is *saturated* if its activation is larger than 0.99, or smaller than -0.99. Likewise, an input pattern is said to be a *saturated pattern* if *all* the hidden-node activations are in the same extreme range. Note also that the classification accuracy is measured by the following criterion so that the MLP output for the pth pattern, a_p, can be regarded as either "on" (1.0) or "off" (-1.0):

$$\begin{array}{ll} \text{if } a_p \geq 0.8 & a_p \text{ is classified to "on,"} \\ \text{if } a_p \leq \text{-}0.8 & a_p \text{ is classified to "off,"} \\ \text{otherwise} & a_p \text{ is "undecided."} \end{array} \tag{8}$$

Indirect approaches tend to make some hidden nodes totally saturated, as shown in Table 1. This inspired us to apply hidden-node teaching to improve the performance of the simple SD method. For solving this *classification* problem, desired outputs "0.6" or "-0.6" [denoted by t_j^s in Equation (7)] were presented to a subset of hidden nodes in a deterministic fashion, so that *those nodes are prevented from being saturated*. The initial number of supervised hidden nodes was set equal to two, and until epoch 2,000, the hidden-node teaching was always applied to those two nodes. Then, after epoch 2,000, the hidden-node teaching was switched to the other two hidden-nodes and applied to them only when saturated. This is just our ad-hoc scheme, but interestingly it made the algorithm quite *insensitive* to the randomly-initialized parameters in the sense that the required epochs to converge were not very different for the ten trials, as shown in Table 2, and the problem was always solved as long as the range of randomly-initialized parameters was small.

Overall, the direct dogleg algorithm worked very well among the indirect approaches. Its robustness mainly comes from the "dogleg trust-region" mechanism, detailed in ref. [7, 8]. This dogleg strategy can be incorporated into other deflected gradient algorithms (that take a descent direction different from the steepest descent direction) in order to avoid "stagnation." Intriguingly, even with such a sophisticated dogleg method, which is a **deterministic** algorithm, the MLP-learning sometimes got "stuck" before the goal was achieved. This implies that the algorithm did not find any descent step, whatever infinitesimal steps were taken in the computed negative gradient direction. This *bizarre stagnation* may not be found in totally unconstrained optimization problem, but again the neural networks nonlinear least squares problem is an *implicitly constrained* problem. Presumably, the computed gradient might not be the true gradient due to hidden-node saturation; that is, if certain hidden nodes are saturated for some patterns, then the computed gradient may no longer be the true gradient, ensemble for all the training patterns; hence, losing a way in the *true* steepest descent direction.

In addition, it was observed that when the hidden-node saturation occurred (and thus the plateau appeared), the condition number of the Gauss-Newton model Hessian $\mathbf{J}^T\mathbf{J}$ soared up, since if *saturation* happens at some hidden nodes (say, for pattern i), then

[§] Initializing the first-layer weights with all zeros does not help avoid hidden-node saturation at all; interested readers are encouraged to test the $7 \times 4 \times 1$ MLP described in the text.

Table 1: *Sample results of the seven-bit parity problem obtained with a $7 \times 4 \times 1$ MLP (37 parameters) for the number of incorrect patterns and the number of patterns for which all four hidden nodes were saturated at the stopped epoch. The last column shows the number of saturated patterns per hidden node. Five algorithms were compared; wherein, $diag(\mathbf{J}^T\mathbf{J})$-based algorithm used a small regularization constant with a dogleg strategy. All algorithms started at the same point in the parameter space.*

Algorithm	Stopped epoch	# of incor. patterns	# of satu. patterns	Hidden-node satu. patterns
SD with hid-teaching	18,708	0	0	114 / 98 / 0/ 58
SD w/o hid-teaching	20,000	38	58	128/ 98/ 88 / 98
Marquardt's algorithm	96	2	0	108/ 98/ 89 / 0
$diag(\mathbf{J}^T\mathbf{J})$-based algorithm	546	58	128	128/128/128/128
Dogleg algorithm	1,250	0	0	98 / 84 / 128 / 0

Table 2: *Comparison of the success trials and their required epochs among ten attempts for the five algorithms for solving the seven-bit parity problems by an MLP with four hidden nodes. The "average" of the required epochs was computed over the success trials.*

Algorithm	Required epoch			# of success trials
	min.	avg.	max.	
SD with hid-teaching	18,688	18,768.6	18,962	10
SD w/o hid-teaching	-	-	-	0
Marquardt's algorithm	103	112.5	122	4
$diag(\mathbf{J}^T\mathbf{J})$-based algorithm	-	-	-	0
Dogleg algorithm	63	403.62	1,250	8

the elements (associated with those hidden nodes) of the ith row vector of the Jacobian matrix $\mathbf{J}$ become very small, although the ith component of the residual vector is still large. This implies that the information of the full Gauss-Newton model Hessian $\mathbf{J}^T\mathbf{J}$ is quite useful, because it can sense such a bad situation; on the other hand, its diagonal approximation was not good enough in this seven-bit parity problem by a $7 \times 4 \times 1$ MLP, as shown in Tables 1 and 2.

5 Concluding Remarks

We have compared several MLP-learning algorithms using the parity problem. Although the simulations were small-scale, we have observed that hidden-node saturation is most likely to adversely affect MLP-learning, and can be avoided by employing hidden-node teaching (although our scheme is ad-hoc at this stage), or by using the information of the Gauss-Newton model Hessian in conjunction with the dogleg strategy. Our future work is to develop *effective* algorithms that deal with special but important features of the *implicitly constrained* neural networks nonlinear least squares problem.

References

[1] E. Mizutani and J.-S. Roger Jang. Chapter 6: Derivative-based Optimization. In *Neuro-Fuzzy and Soft Computing: a computational approach to learning and machine intelligence*, pages 129–172. J.-S. Roger Jang, C.-T. Sun and E. Mizutani. Prentice Hall, 1997.

[2] E. Mizutani, S.E. Dreyfus, and K. Nishio. On derivation of MLP backpropagation from the Kelley-Bryson optimal-control gradient formula and its application. In *Proceedings of the IEEE International Conference on Neural Networks*, Como, Italy, July 2000.

[3] R.E Bellman and S.E. Dreyfus. *Applied dynamic programming*. Princeton University Press, 1962.

[4] Eiji Mizutani and James W. Demmel. On iterative Krylov-dogleg trust-region steps for solving neural networks nonlinear least squares problems. *Advances in Neural Information Processing Systems (NIPS 2000)*, 2000. (To appear).

[5] K. Fukumizu and S.-I. Amari. Local minima and plateaus in hierarchical structures of multilayer perceptrons. *Neural Networks*, 13(3), 2000.

[6] H. Demuth and M. Beale. Neural Network Toolbox for Use with MATLAB. The MathWorks, Inc., Natick, Massachusetts, 1998. User's Guide (version 3.0).

[7] E. Mizutani. Computing Powell's dogleg steps for solving adaptive networks nonlinear least-squares problems. In *Proc. of the 8th Int'l Fuzzy Systems Association World Congress, vol.2*, pages 959–963, Hsinchu, Taiwan, August 1999.

[8] Eiji Mizutani. Powell's dogleg trust-region steps with the quasi-Newton augmented Hessian for neural nonlinear least-squares learning. In *Proceedings of the IEEE Int'l Conf. on Neural Networks*, Washington, D.C., July 1999.