On complexity analysis of supervised MLP-learning for algorithmic comparisons

Eiji Mizutani [1] and Stuart E. Dreyfus [2]

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

1) Dept. of Computer Science, National Tsing Hua University, Hsinchu, 300, TAIWAN R.O.C.

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

Abstract

This paper presents complexity analysis of a standard supervised MLP-learning algorithm in conjunction with the well-known **backpropagation***, an efficient method for evaluation of derivatives, in either batch or incremental learning mode. In particular, we detail the cost per epoch (i.e., operations required for processing one sweep of all the training data) using "approximate" flops (floating point operations) in a typical backpropagation for solving* **neural networks nonlinear least squares problems***. Furthermore, we shall identify erroneous complexity analyses found in the past NN literature. Our operation-count formula would be very useful for a given MLP architecture to compare learning algorithms.*

1 Introduction

In the standard MLP-learning, the fundamental concepts of two important processes, **forward propagation/pass (FP)** [1] and **backpropagation (BP)**, have been discussed in a variety of contexts since the first BP concept arose historically in connection with the optimal control theory (see ref. [1] and the bibliography therein). In order to eliminate any possibility of confusion, we define certain key words below:

Backpropagation (BP) Algorithm
Forward pass (FP)
Process 1: Node net-input computation
Process 2: Node activation (or output) evaluation
Process 3: Error (or objective-function) evaluation
Backward pass
Process 4: Node sensitivity (or delta) evaluation
Process 5: Gradient computation
Process 6: Parameter updating (using gradient)

In descriptions of MLP-learning, some use FP merely for Processes 1 and 2, and some include Process 3 also. Likewise, the term BP inevitably depends on a context, implying the **backward pass** alone[2], or the entire BP algorithm as a *steepest descent*-type method [3]. In this paper, as shown in the above table, we distinguish "backward pass" and "BP." That is, we define BP as an efficient algorithm for computing derivatives of the objective function, which uses both "forward pass" and "backward pass." Each pass is split into three stages for our later discussion purposes.

In the past NN literature, some researchers describe complexity of FP and BP using flops (floating point operations), but several descriptions contain some ambiguity in a sense that their cost analysis was described by using *the number of parameters* alone without involving *the number of data* (see, for instance, page 532 in Moller [4]; page 117 in Shepherd [5]); yet, the dominant costs in MLP-learning must depend on *both*. Perhaps, more important, some individuals estimate the complexity of their own algorithms using Yoshida's very rough estimate [6] on BP complexity; for instance, Moller (see page 532 in ref. [4]) wrote "Backward pass needs *three* times as many operations as FP" (see also page 1,782 in Magoulas et al. [7]). Here, a question arises: Is the proportionality factor, specified as *three*, independent of the MLP architecture? The answer is "No" to be detailed in the remainder of this paper. In other words, such a very rough estimate may result in a misleading conclusion for algorithmic speed comparisons. In fact, FP might take more time than "backward pass" on modern computer systems due to Process 2; in this case, the proportionality factor becomes less than *one*. Hence, algorithms that need FPs many more repetitions per epoch than backward passes typically with line-search methods (see, for instance ref. [7]) might work much slower per epoch than expected. On the other hand, the proportionality factor might become larger than three in another setting. Those situ-

[1] In this paper, "forward propagation" and "forward pass" are employed interchangeably, both denoted by FP.

[2] In Algorithm 3.2(b) in ref. [2], the term *backpropagation* was used as the "backward pass."

ations are totally dependent upon an MLP architecture and a user's computer platform. In the following, we shall attempt to clarify this fact, showing "flops complexity per epoch" in the standard MLP learning.

2 MLP Backpropagation

In this section, we derive quite general formulas for evaluating flops (per epoch) of FP and backward pass using several set-up parameters (e.g., the number of nodes per layer) specified for a given MLP architecture. Here are some notations we use in this paper.

Entry	Descriptions
d	Total no. of training data
q	A particular datum
N	Total no. of layers (including the first input layer)
s	A particular layer (or stage)
P_s	No. of nodes at layer s excluding bias node
P_N	No. of output nodes at the terminal layer N
n	Total no. of weight parameters
m	$P_N \times d$ (no. of rows of the residual Jacobian matrix)
θ^s_{jk}	Parameter between nodes j (layer s) and k (layer s+1)
$a^s_{j,q}$	Activation (or output) of node j, layer s for datum q
$\xi^s_{j,q}$	Sensitivity ($\stackrel{\text{def}}{=} \frac{\partial E_q}{\partial a^s_{j,q}}$) for datum q at node j, layer s
$f^s_j(.)$	An activation function at node j in layer s
T_s	Operations for evaluating $f^s(.)$, node activation
V_s	Operations for evaluating $f^{s'}(.)$, function derivative
$\mathbf{\Theta}^s$	A $[(P_s+1)\times(P_{s+1})]$ matrix of parameters between layers s and s+1, including P_{s+1} threshold parameters. Its (j,k)-element is $\theta^s_{jk}(1 \le j \le P_s+1; 1 \le k \le P_{s+1})$.
$\mathbf{\Theta}^s_-$	A $[P_s \times P_{s+1}]$ matrix of parameters between layers s and s+1, *excluding* P_{s+1} threshold parameters. Its (j,k)-element is $\theta^s_{jk}(1 \le j \le P_s; 1 \le k \le P_{s+1})$.

Using these notations, we first show operation counts for each of the six processes in forward and backward passes with a given datum q. We then summarize the total cost for each process. Note in this paper that all vectors are considered to be *column* vectors unless otherwise specified, and vectors and matrices are represented as *bold* lowercase and uppercase letters, respectively.

2.1 Forward pass

Process 1: Node net-input computation

The **net input** to node j at layer s ($2 \le s \le N$) for datum q is computed as:

$$\begin{array}{ll} & net^s_{j,q} = \sum_{i=1}^{P_{s-1}+1} \theta^{s-1}_{ij} a^{s-1}_{i,q},\ j = 1, ..., P_s \\ \Longleftrightarrow & \mathbf{net}^s_q = (\mathbf{\Theta}^{s-1})^T \mathbf{a}^{s-1}_q, \end{array} \quad (1)$$

which has an interpretation that an incoming-signal vector $\mathbf{a}^{s-1}_q$ acts on $\mathbf{\Theta}^{s-1}$ to produce $\mathbf{net}^s_q$. The required operations are associated with a *matrix-vector product* of a transposed "weight-parameter" matrix $(\mathbf{\Theta}^{s-1})^T$ with a $(P_{s-1}+1)$-dimensional vector $\mathbf{a}^{s-1}_q$ of incoming signals that include a constant activation (e.g., 1.0) from one bias node[3] at layer s-1, which requires $2(P_{s-1}+1)P_s$ operations in total.

Process 2: Node activation (or output) evaluation

The **activation** (or output) of node j at layer s ($2 \le s \le N$) for datum q is expressed as:

$$a^s_{j,q} = f^s_j(net^s_{j,q}). \quad (2)$$

A node function $f(.)$ is often a sigmoidal "hyperbolic tangent (tanh)" function given next:

$$f(x) \stackrel{\text{def}}{=} tanh(x) = \frac{1-exp(-Kx)}{1+exp(-Kx)} = \frac{2}{1+exp(-Kx)} - 1, \quad (3)$$

where a parameter K is set equal to *two* in the standard "tanh" function. As indicated in the notation table, we denote the cost of evaluating $f^s(.)$ by T_s, which requires at least four flops plus some operations for evaluating $exp(.)$ when Equation (3) is implemented directly. Hence, the total cost of Process 2 is given by $T_s P_s$, assuming each layer s has the same type of node functions.

Process 3: Error (or objective-function) evaluation

As the *objective function*, we employ the usual "sum of squared errors" measure:

$$\begin{array}{l} E(\boldsymbol{\theta}) = \sum_{q=1}^d E_q = \frac{1}{2}\sum_{q=1}^d \sum_{k=1}^{P_N} (a^N_{k,q} - t^N_{k,q})^2 \\ \quad = \frac{1}{2}\sum_{q=1}^d \sum_{k=1}^{P_N} r^2_{k,q} = \frac{1}{2}\sum_{q=1}^d \mathbf{r}^T_q \mathbf{r}_q, = \frac{1}{2}\mathbf{r}^T\mathbf{r}, \end{array} \quad (4)$$

where $t^N_{k,q}$ is the kth desired output for the qth training data at the terminal layer N; $\mathbf{r}_q (\stackrel{\text{def}}{=} \mathbf{a}^N_q - \mathbf{t}^N_q)$ is the residual vector for datum q between the qth output vector $\mathbf{a}^N_q$ and its desired output vector $\mathbf{t}^N_q$; and, for our convenience, $\mathbf{r}$ is defined as the residual vector composed of r_i for $i = 1, \ldots, m (\stackrel{\text{def}}{=} P_N \times d)$. Computing the P_N-dimensional residual vector $\mathbf{r}_q$ for datum q needs P_N subtractions; so, the total cost associated with $\mathbf{r}^T_q \mathbf{r}_q$ for a given datum q is $3P_N$.

2.2 Backward pass

Process 4: Node sensitivity (or delta) evaluation

At the terminal layer N, the **sensitivity** ξ at node k

[3] It is well-known that the P_s threshold parameters in layer s can be viewed as the parameters connected to a bias node, located at layer s-1, that always produces a fixed output (e.g., 1.0).

for datum q is given by[4]

$$\xi^N_{k,q} \stackrel{\text{def}}{=} \frac{\partial E_q}{\partial a^N_{k,q}} = a^N_{k,q} - t^N_{k,q} = r_{k,q},\ k = 1, ..., P_N, \quad (5)$$

which requires just P_N subtractions for computing the P_N-dimensional residual vector $\mathbf{r}_q (= \mathbf{a}^N_q - \mathbf{t}^N_q)$. (Of course, this cost can be omitted if the residual vector is computed in Process 3.)

Next, at layer s $(1 < s < N)$, the sensitivity $\boldsymbol{\xi}$ is propagated/computed backward one layer after another (departing from the *terminal* layer N) by the following recurrence relation:

$$\begin{aligned} &\xi^s_{j,q} \stackrel{\text{def}}{=} \tfrac{\partial E_q}{\partial a^s_{j,q}} = \textstyle\sum_{k=1}^{P_{s+1}} \theta^s_{jk} \delta^{s+1}_{k,q},\ \ j = 1, ..., P_s, \\ &\Longleftrightarrow \boldsymbol{\xi}^s_q = \boldsymbol{\Theta}^s_- \ \boldsymbol{\delta}^{s+1}_q, \end{aligned} \quad (6)$$

where, for our convenience, another form of *sensitivity* is defined as:

$$\delta^{s+1}_{k,q} \stackrel{\text{def}}{=} \frac{\partial E_q}{\partial net^{s+1}_{k,q}} = \xi^{s+1}_{k,q} f^{s+1'}_k (net^{s+1}_{k,q}), \quad k = 1, ..., P_{s+1}. \quad (7)$$

The costs associated with Equations (6) and (7) are:

(A) $P_{s+1} V_{s+1}$ operations for evaluating the node-function derivative $f^{s+1'}(.)$ at layer $s+1$ in Eq. (7);

(B) P_{s+1} component-wise multiplications of each component $\xi^{s+1}_{k,q}$ of the P_{s+1}-dimensional vector $\boldsymbol{\xi}^{s+1}_q$ with its corresponding node derivative $f^{s+1'}_k(.)$ to generate the P_{s+1}-dimensional vector $\boldsymbol{\delta}^{s+1}_q$ in Eq. (7);

(C) $2P_s P_{s+1}$ operations for a matrix-vector product [i.e., a $P_s \times P_{s+1}$ parameter matrix $\boldsymbol{\Theta}^s_-$ times a P_{s+1}-dimensional vector $\boldsymbol{\delta}^{s+1}_q$, a result of the previous product (B)], which yields a P_s-dimensional sensitivity vector $\boldsymbol{\xi}^s_q$.

Hence, the total cost of Equation (6) is $P_{s+1}(V_{s+1} + 1) + 2P_s P_{s+1}$ for a given datum q.

Process 5: Gradient computation

The gradient for θ^s_{jk} with a given datum q at layer s $(1 \le s \le N-1)$ can be calculated by:

$$\begin{aligned} &g^s_{jk,q} = a^s_{j,q} [\xi^{s+1}_{k,q}\ f^{s+1'}_k (net^{s+1}_{k,q})] = a^s_{j,q}\ \delta^{s+1}_{k,q}, \\ &\quad (j = 1, ..., (P_s + 1);\ k = 1, ..., P_{s+1}) \\ &\Longleftrightarrow \mathbf{g}^s_q \leftarrow (\text{reshape}) \leftarrow \mathbf{G}^s_q = \mathbf{a}^s_q\ \chi\ (\boldsymbol{\delta}^{s+1}_q)^T. \end{aligned} \quad (8)$$

[4]Our definition of "sensitivity" $\boldsymbol{\xi}$ (*after-node sensitivity*) is in the spirit of the optimal control theory [1], being slightly different from the one defined by Rumelhart et. al. (see pages 325 and 326 in ref. [3]), which corresponds to $\boldsymbol{\delta}$ (*before-node sensitivity*) in Equation (7). Those two forms of sensitivity become identical when node functions $f(.)$ are *linear identity* functions (usually employed only at the terminal layer N).

In words, computing the $(P_s+1)P_{s+1}$-dimensional *gradient* vector $\mathbf{g}^s_q$ for datum q costs $(P_s+1)P_{s+1}$ operations for performing the **outer product**, denoted by "χ," of a $[(P_s + 1) \times 1]$ incoming-signal *column* vector $\mathbf{a}^s_q$ with a $[1 \times P_{s+1}]$ sensitivity *row* vector $(\boldsymbol{\delta}^{s+1}_q)^T$ already computed in Process 4, yielding gradients in a $(P_s + 1) \times P_{s+1}$ matrix $\mathbf{G}^s_q$ for the associated weight matrix $\boldsymbol{\Theta}^s$. Hence, the total cost is $(P_s + 1)P_{s+1}$. In the batch mode, $(P_s+1)P_{s+1}$ operations for summation per datum are required for: $g^s_{jk,\text{all}} = \sum_{q=1}^d g_{jk,q}$.

Process 6: Parameter updating (using gradient)

We can consider a wide variety of parameter-updating formulas in light of the vast literature on *nonlinear optimization* (see, for instance, ref. [8] and references therein). The simplest ones, steepest descent-type methods, can be written with a step size (or learning rate) η in either of the next two forms:

- On-line incremental (pattern-by-pattern) mode learning (or **incremental gradient**) method: $\theta^s_{jk} \leftarrow \theta^s_{jk} - \eta^s g^s_{jk,q}$. Hence, it costs $2(P_s+1)P_{s+1}$ per datum.
- Off-line batch-mode learning (or **steepest descent**) method: $\theta^s_{jk} \leftarrow \theta^s_{jk} - \eta^s g^s_{jk,\text{all}}$. This final computation occurs only once per epoch and may be therefore negligible.

For the posed nonlinear least squares problem [in Equation (4)], the gradient vector $\mathbf{g}$ can be expressed as: $\mathbf{g} = \mathbf{J}^T \mathbf{r}$, where $\mathbf{J}$ denotes the $m \times n$ Jacobian matrix of the residual vector. Now, an alternative popular method, a **Newton**-type method, can be written as:

$$\mathbf{H} \Delta \boldsymbol{\theta} = -\mathbf{J}^T \mathbf{r}, \quad (9)$$

where $\mathbf{H}$ denotes a Hessian matrix. If we simply consider a Gauss-Newton model Hessian (i.e., $\mathbf{J}^T\mathbf{J}$), then the method with Equation (9), known as the **Gauss-Newton** method, is the *normal equations approach* in a least-squares sense. When $\mathbf{J}$ is ill-conditioned or nearly rank-deficient, we can proceed with the *Jacobian system*: $\mathbf{J}\Delta\boldsymbol{\theta} = -\mathbf{r}$ in conjunction with a suitable matrix decomposition; namely, QR-decomposition or SVD (singular value decomposition) approaches (see, for instance, Chapter 3 in Demmel [9]). Among those three standard **direct methods**, the *normal equations* approach is fastest per epoch and least memory-intensive: In a usual overdetermined case[5] $(m > n)$, forming the

[5]Shepherd's complexity analysis did not involve m; so, his computational cost $O(n^3)$ (see, for example, pages 63 and 117 in ref. [5]) was quite confusing, and not appropriate for a highly overdetermined problem $(m \gg n)$.

Gauss-Newton model Hessian $\mathbf{J}^T\mathbf{J}$ (mn^2 operations) is dominant and more expensive than solving the linear system at a cost of $O(n^3)$. For a multiple-output ($P_N > 1$) MLP, $\mathbf{J}^T\mathbf{J}$ can be formed at a smaller cost than mn^2 (see ref. [10] for more details). For a large-scale problem, instead of those *direct batch-mode learning* schemes, we can employ *iterative batch-mode learning* schemes so as to reduce the dominant cost from $O(mn^2)$ to $O(mn)$ (see refs. [2, 11]). Further details of Newton-type algorithms are outside the scope of this paper due to space limitation.

Perhaps, another popular alternative is a batch-mode *nonlinear conjugate gradient* (N-CG) (see Section 6.6 in ref.[8]) that determines the descent search direction $\mathbf{d}$ for $\Delta\boldsymbol{\theta} = \eta\mathbf{d}$ with:

$$\mathbf{d} = -\mathbf{g}_{\text{now}} + \beta\Delta\boldsymbol{\theta}_{\text{old}}, \tag{10}$$

where various choices of scalar β are known; in a most popular *Polak-Rebiere* N-CG, β is chosen as:

$$\beta = \frac{\mathbf{g}_{\text{now}}^T(\mathbf{g}_{\text{now}} - \mathbf{g}_{\text{old}})}{\mathbf{g}_{\text{old}}^T\mathbf{g}_{\text{old}}}. \tag{11}$$

The N-CG algorithms are usually equipped with an *inexact line search* method; e.g., a *quadratic interpolation* method that finds η for the next step $\Delta\boldsymbol{\theta} = \eta\mathbf{d}$ to a unique minimum point of a quadratic curve passing through three points ($\boldsymbol{\theta}_{\text{now}}$, $\boldsymbol{\theta}_{\text{now}} + \eta_1\mathbf{d}$, and $\boldsymbol{\theta}_{\text{now}} + \eta_2\mathbf{d}$) that satisfy $E(\boldsymbol{\theta}_{\text{now}}) > E(\boldsymbol{\theta}_{\text{now}} + \eta_1\mathbf{d}) < E(\boldsymbol{\theta}_{\text{now}} + \eta_2\mathbf{d})$ (see pages 141 through 146 in ref. [8] for more details). Then, the usual steps are:

(1) Search direction determination:

Processes 1 to 5
$\Longrightarrow$ β-computation (costs $5n$) in Eq. (11)
$\Longrightarrow$ $\mathbf{d}$-computation (costs $2n$) in Eq. (10);

(2) Step size determination:

Repeat function evaluations (i.e., Processes 1 through 3) until we find η_1 and η_2 that satisfy $E(\boldsymbol{\theta}_{\text{now}}) > E(\boldsymbol{\theta}_{\text{now}} + \eta_1\mathbf{d}) < E(\boldsymbol{\theta}_{\text{now}} + \eta_2\mathbf{d})$, and then compute η (see page 145 in ref. [8]).

Hence, the total cost can be written with L [the number of function evaluations required at Step (2)] as:

$$\text{cost of Processes 1 to } 5 + L \cdot [\text{cost of Processes 1 to 3}], \tag{12}$$

plus extra (less important) costs $O(n)$ independent of d, which include operations necessary for computing $\beta, \mathbf{d}$, and η as well as parameter-updating with $\boldsymbol{\theta}_{\text{next}} = \boldsymbol{\theta}_{\text{now}} + \eta\mathbf{d}$. This (one-shot) *quadratic interpolation* is one of the least expensive "inexact" line-search methods. More *exact/precise* line-search methods could be used, but the cost associated with the N-CG algorithm would become much larger, although the situation might be problem-dependent.

Table 1: BP complexity for total operations per epoch.

Forward pass (FP)	
Process 1	$2d\sum_{s=2}^{N}(P_{s-1}+1)P_s = 2dn$
Process 2	$d\sum_{s=2}^{N} T_s P_s$
Process 3	dP_N for residual evaluation $+\ 2dP_N$ for error computation
Backward pass	
Process 4	dP_N for terminal sensitivity in Eq. (5) $+\ d\sum_{s=1}^{N-1} P_{s+1}(V_{s+1}+1)$ for $\boldsymbol{\delta}$ in Eq. (7) $+\ 2d\sum_{s=2}^{N-1} P_s P_{s+1}$ for sensitivity prop., Eq. (6)
Process 5	$d\sum_{s=1}^{N-1}(P_s+1)P_{s+1} = dn$ for incremental or $2dn$ (twice as many) for batch
Process 6	$2d\sum_{s=1}^{N-1}(P_s+1)P_{s+1} = 2dn$ for incremental or $2n$ (independent of d) for batch

2.3 BP Complexity

Table 1 summarizes the operation counts for processing one sweep of all the training data. The first cost dP_N in Process 4 can be omitted when the residual vector is computed in Process 3. Note also that $T_s = V_s = 0$ for linear identity node functions. If we employ a "tanh" node function [see Equation (3)] at all layers (except at layer 1), then each V_s turns out to be merely a constant, *four*, although T_s can vary because the operations required for evaluating $tanh(.)$ may depend on the magnitude of the net input to that function and on a computer platform. Actually, on most modern computers, evaluating a transcendental function $exp(.)$ [in Equation (3)] tends to take much more time[6] (e.g., more than ten times longer) than one addition/multiplication, because, for instance, it can typically take 60 clock cycles on emerging Intel PCs (see ref. [13]). Therefore, Process 2 can be most time-consuming in FP and thus *the time for FP can be greater than that for backward pass,* which might be

[6] Another evidence can be found in assembly codes, wherein the number of *instructions* associated with the transcendental function $exp(.)$ tends to be much greater than that with one addition or multiplication, which is typically implemented in hardware as a single instruction. Note also that "addition" and "multiplication" cost almost the same time on most modern computers (e.g., Pentium, UltraSparc). Even in a most recent cost analysis on NN-learning (e.g., page 138 in ref. [12]), additions and T (operations for evaluating node activations) are all ignored, which may not be appropriate for algorithmic comparisons on modern computer systems.

counterintuitive (see Section 3).

In a *single-output* MLP, we have $d = m$, and intuitively, the $2mn$ cost of the "batch-mode backward pass" (i.e., Process 5) corresponds to the operation counts (including both addition and multiplication) of the **matrix-vector multiplication** for computing the gradient $\mathbf{J}^T\mathbf{r}$ with neither the $m \times n$ Jacobian matrix $\mathbf{J}$ nor the m-dimensional residual vector $\mathbf{r}$ explicitly required. Note also in the backward pass of a single-output MLP that all the elements in the ith row of $\mathbf{J}$ are multiplied first by a single ith component (r_i) of the residual vector $\mathbf{r}$ to obtain the n-dimensional gradient vector $\mathbf{g}_i$ for datum i in a row-first-sweep manner, and then accumulate $\mathbf{g}_i$ to obtain the n-dimensional gradient vector $\mathbf{g} = \sum_{i=1}^{m} \mathbf{g}_i$ for all the data. On the other hand, in the straightforward *algebraic* matrix-vector multiplication, all the elements ($J_{ij}, i = 1, ..., m$) in the jth column of $\mathbf{J}$ are multiplied by the corresponding elements of $\mathbf{r}$ (for each parameter j; $1 \leq j \leq n$) in a column-first-sweep fashion. They are just different implementations of $\mathbf{J}^T\mathbf{r}$, *matrix-vector multiplication*.

In a *multiple-output* MLP, we have $m = P_N d$, and thus the $m \times n$ Jacobian matrix $\mathbf{J}$ becomes **sparse** because, in the i [$=P_N(p-1)+k$]th row of $\mathbf{J}$ (for datum q at node k in layer N), there are $(P_{N-1}+1)(P_N-1)$ *zeros* associated with the (weight) parameters connected to nodes $l(\neq k)$ at layer N (see ref. [10]). In this context, the *outer product* in Process 5 in the batch-mode backward pass can be viewed as an efficient method that exploits the *sparsity* structure of $\mathbf{J}$ to perform *matrix-vector multiplication* ($\mathbf{J}^T\mathbf{r}$) at a smaller cost than $2mn$.

3 Experiments and Discussion

In our experiments, we show analytically how many flops per epoch are necessary in terms of performing FP and backward pass. We then measure the actual execution time per epoch by simulation. After that, we compare these measured execution time with the flops-cost obtained with our derived complexity formula. All the simulations were implemented in double precision on a 700-MHz Pentium III laptop PC with FreeBSD 4.2 and gcc (2.95.1) compiler (with -O1 optimizer flag).

First, we consider a ten-input and single-output problem that consists of 1,024 data with a 10-9-1 MLP ($d = 1,024$ and $n = 109$), wherein totally ten "tanh (hyperbolic tangent)" node functions are used at the hidden and output layers. The next table presents the cost-per-epoch associated with this MLP when it is trained with a batch-mode steepest descent method:

Batch 10-9-1 MLP	Approx. flops	Exec. time (sec)
Forward pass	$618d$	5.7×10^{-3}
Process 1	$218d$	1.8×10^{-3}
Process 2	$400d$	3.9×10^{-3}
Process 3	(Omitted)	(Omitted)
Backward pass	$287d$	2.6×10^{-3}
Process 4	$69d$	0.7×10^{-3}
Process 5	$218d$	1.9×10^{-3}
Process 6	218	(Negligible)

The second column "Approx. flops" was obtained with $V_s = 4$ and $T_s = 40$, wherein T_s was approximated by a comparison between the execution time measured for $tanh(.)$ and the time for addition/multiplication operations; hence, the name "approximate" flops, different from the standard flops. The third column "Exec. time per epoch" was the execution time averaged over 1,000 epochs. Using the above information, we can compute a ratio between the costs of backward and forward passes with respect to both approximate flops and average execution time, as shown next:

$$\begin{aligned} \text{Ratio}_{Appx.flops} &= \frac{\text{Cost(backward pass)}}{\text{Cost(forward pass)}} = \frac{287}{618} = 0.464 \\ \text{Ratio}_{Exec.time} &= \frac{\text{Cost(backward pass)}}{\text{Cost(forward pass)}} = \frac{2.6}{5.7} = 0.456 \end{aligned} \tag{13}$$

Both ratios became fairly close mainly because we used an estimated T_s based on actually measured time.

Next, to construct a situation when backward pass uses more flops than FP, one needs to have very few "tanh" evaluations in FP. Here are two examples: a 10-3-18 MLP and a 20-10-10-20 MLP both in incremental mode with linear identity node functions used at terminal layer N (see the 2nd and 3rd columns in the next table). For comparison purpose, the 4th and 5th columns show a case where "tanh" functions were used at layer N.

	Approx. flops	Exec. time	Approx. flops	Exec. time
Incremental-mode 10-3-18 MLP ($n = 105$ and $d = 1,000$)				
	3 "tanh" nodes		21 "tanh" nodes	
Forward	$384d$	3.0 (m.sec)	$1,104d$	9.7 (m.sec)
Backward	$474d$	4.1 (m.sec)	$546d$	4.2 (m.sec)
Ratio	1.23	1.37	0.49	0.43
Incremental-mode 20-10-10-20 MLP ($n = 540$ and $d = 1,000$)				
	20 "tanh" nodes		40 "tanh" nodes	
Forward	$1,940d$	17.2 (m.sec)	$2,740d$	25.4 (m.sec)
Backward	$2,360d$	22.1 (m.sec)	$2,440d$	22.2 (m.sec)
Ratio	1.22	1.28	0.89	0.87

From this table, the ratios in Equation (13) were again fairly close. The average execution time (millisecond)

per epoch certainly included the cost of memory accesses; hence, the "absolute" time was not strictly proportional to the approximate flops measure; however, our cost formula in approximate flops can give a better estimate to "relative" speed when appropriate operation counts for (computer platform-dependent) T_s is first evaluated on a given simulation environment and then plugged into that formula.

Through these observations, backward pass may take more time than FP when an MLP has very few (sigmoidal) node functions that require evaluations of transcendental functions, but FP often takes more time than backward pass. That is, the proportionality factor between FP and backward pass tends to stay smaller than *one*, although it varies, depending upon a given MLP architecture.

4 Conclusion

The flops measure is often a misleading time estimate in evaluating algorithmic costs because, for instance, addition/multiplication usually needs less time than division and some other mathematical library functions (e.g., square root). Also, any floating point operation takes most of the time to move data when a cache miss or page fault occurs in a modern computer memory hierarchy system (see, for instance, ref. [14]) rather than actually performs individual operations. Yet, our cost formula in Table 1 certainly gives a good estimate to the cost per epoch once T (cost of evaluating a node activation) is appropriately measured or guessed on each users' machine platform. Therefore, what we call "approximate" flops appears to be a useful touchstone in estimating the cost-per-epoch of various learning methods for algorithmic speed comparison purposes.

Acknowledgments

We would like to thank Professor James W. Demmel (CS/Math, UC Berkeley) and Dr. Rich Vuduc (CS, UC Berkeley) for their expertise on implementation issues on modern computer systems; of course, any erroneous statements and mistakes are purely ours. We also wish to acknowledge Prof. Toshinobu Yoshida (ICE, Univ. of Electoro-Communications, Japan) for useful information. Special thanks must go to Jones Chang (Ulead Systems, Inc., Taiwan) for much help in a friendly manner. The work was supported in part by "Program for Promoting Academic Excellence of Universities," grant 89-E-FA04-1-4, Ministry of Education, Taiwan.

References

[1] 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 (vol.2)*, pages 167–172, Como, Italy, July 2000.

[2] Eiji Mizutani and James W. Demmel. On iterative Krylov-dogleg trust-region steps for solving neural networks nonlinear least squares problems. In V. Tresp, editor, *Advances in Neural Information Processing Systems (NIPS 2000)*, volume 13, pages 605–611. MIT Press, 2001.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, volume 1. MIT press, Cambridge, MA., 1986.

[4] Martin Fodslette Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[5] Adrian J. Shepherd. *Second-Order Methods for Neural Networks: Fast and Reliable Training Methods for Multi-Layer Perceptrons*. Springer-Verlag, 1997.

[6] Toshinobu Yoshida. Rapid learning method for multilayered neural networks using two-dimensional conjugate gradient search. *Journal of Information Processing*, 15(1):79–86, 1992.

[7] G.D. Magoulas, M.N. Vrahatis, and G.S. Androulakis. Improving the convergence of the backpropagation algorithm using learning rate adaptation methods. *Neural Computation*, 11:1769–1796, 1999.

[8] 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.

[9] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[10] Eiji Mizutani and James W. Demmel. On structure of the residual Jacobian matrix arising in neural-network learning. To appear in the Proceedings of the Fifth International Conference on Knowledge-Based Intelligent Information Engineering Systems & Allied Technologies (KES'2001), Osaka, JAPAN. Sept., 2001.

[11] Eiji Mizutani and James W. Demmel. On generalized dogleg trust-region steps using the Krylov subspace for solving neural networks nonlinear least squares problems. Technical report, Dept. of Computer Science, University of California at Berkeley, 2001. (In preparation).

[12] Martin Bouchard. New recursive-least-squares algorithm for nonlinear active control of sound and vibration using neural networks. *IEEE Transactions on Neural Networks*, 12(1):135–147, January 2001.

[13] J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang. The computation of transcendental functions on the IA-64 architecture. Intel Technology Journal, 4th quarter, 1999. http://developer.intel.com/technology/itj/q41999/articles/art_5.htm.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996. (2nd Ed.).