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:

<table>
<thead>
<tr>
<th>Backpropagation (BP) Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward pass (FP)</strong></td>
</tr>
<tr>
<td>Process 1: Node net-input computation</td>
</tr>
<tr>
<td>Process 2: Node activation (or output) evaluation</td>
</tr>
<tr>
<td>Process 3: Error (or objective-function) evaluation</td>
</tr>
<tr>
<td><strong>Backward pass</strong></td>
</tr>
<tr>
<td>Process 4: Node sensitivity (or delta) evaluation</td>
</tr>
<tr>
<td>Process 5: Gradient computation</td>
</tr>
<tr>
<td>Process 6: Parameter updating (using gradient)</td>
</tr>
</tbody>
</table>

1In this paper, “forward propagation” and “forward pass” are employed interchangeably, both denoted by FP.

2In 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.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>Total no. of training data</td>
</tr>
<tr>
<td>$q$</td>
<td>A particular datum</td>
</tr>
<tr>
<td>$N$</td>
<td>Total no. of layers (including the first input layer)</td>
</tr>
<tr>
<td>$s$</td>
<td>A particular layer (or stage)</td>
</tr>
<tr>
<td>$P_s$</td>
<td>No. of nodes at layer $s$ excluding bias node</td>
</tr>
<tr>
<td>$P_N$</td>
<td>No. of output nodes at the terminal layer $N$</td>
</tr>
<tr>
<td>$n$</td>
<td>Total no. of weight parameters</td>
</tr>
<tr>
<td>$m$</td>
<td>$P_s \times d$ (no. of rows of the residual Jacobian matrix)</td>
</tr>
<tr>
<td>$\theta_{k, q}$</td>
<td>Parameter between nodes $q$ (layer $s$) and $k$ (layer $s+1$)</td>
</tr>
<tr>
<td>$a_{k, q}^s$</td>
<td>Activation (or output) of node $j$ at layer $s$ for datum $q$</td>
</tr>
<tr>
<td>$E_{k, q}$</td>
<td>Sensitivity (defined as $\frac{\partial E}{\partial a}$) for datum $q$ at node $j$, layer $s$</td>
</tr>
<tr>
<td>$f_r^s(.)$</td>
<td>An activation function at node $j$ in layer $s$</td>
</tr>
<tr>
<td>$T_r$</td>
<td>Operations for evaluating $f_r^s(.)$, node activation</td>
</tr>
<tr>
<td>$V_r$</td>
<td>Operations for evaluating $f_r^s(.)$, function derivative</td>
</tr>
<tr>
<td>$\theta^s$</td>
<td>A $[[P_s \times 1] \times [P_{s+1}]]$ matrix of parameters between layers $s$ and $s+1$, including $P_{s+1}$ threshold parameters. Its $(j, k)$-element is $\theta^s_{k, q} [1 \leq j \leq P_s; 1 \leq k \leq P_{s+1})$</td>
</tr>
<tr>
<td>$\theta^s_2$</td>
<td>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 $\theta^s_{k, q} [1 \leq j \leq P_s; 1 \leq k \leq P_{s+1})$.</td>
</tr>
</tbody>
</table>

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 $s$ ($2 \leq s \leq N$) for datum $q$ is computed as:

$$\text{net}_q^j = \sum_{a=1}^{P_{s-1}+1} \theta_{ij}^s a_{q}^{s-1}, \quad j = 1, \ldots, P_s$$

which has an interpretation that an incoming-signal vector $a_{q}^{s-1}$ acts on $\Theta^{s-1}$ to produce $\text{net}_q^s$. The required operations are associated with a matrix-vector product of a transposed “weight-parameter” matrix $(\Theta^{s-1})^T$ with a $(P_{s-1} + 1)$-dimensional vector $a_{q}^{s-1}$ 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 $s$ ($2 \leq s \leq N$) for datum $q$ is expressed as:

$$a_{q}^s = f^s_j(\text{net}_q^s).$$

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

$$f(x) \equiv \tanh(x) = \frac{2}{1 + \exp(-Kx)} - 1,$$

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 $T_r$, 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 $T_r P_s$, 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:

$$E(\theta) = \sum_{q=1}^{d} E_{q}^s = \frac{1}{2} \sum_{q=1}^{d} \sum_{k=1}^{P_N} (a_{q}^{N} - t_{k, q}^{N})^2 = \frac{1}{2} \sum_{q=1}^{d} \sum_{k=1}^{P_N} r_{q, k}^{N} r_{q, k}^{N} = \frac{1}{2} r^T r,$$

where $t_{k, q}^{N}$ is the $k$th desired output for the $q$th training data at the terminal layer $N$; $r_{q}^{N}(\equiv a_{q}^{N} - t_{q}^{N})$ is the residual vector for datum $q$ between the $q$th output vector $a_{q}^{N}$ and its desired output vector $t_{q}^{N}$; and, for our convenience, $r$ is defined as the residual vector composed of $r_{q}$ for $i = 1, \ldots, m(\equiv P_N \times d)$. Computing the $P_N$-dimensional residual vector $r_{q}$ for datum $q$ needs $P_N$ subtractions; so, the total cost associated with $r_{q}^{N} r_{q}$ 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 $\xi$ at node $k$\(^3\) 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:

\[
\xi_{k,q} = \frac{\partial E_q}{\partial a_{k,q}} = a_{k,q}^N - t_{k,q}, \quad k = 1, \ldots, P_N,
\]

which requires just \( P_N \) subtractions for computing the \( P_N \)-dimensional residual vector \( r_{\xi}(= a_{k,q}^N - t_{k,q}) \). (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 \( \xi \) is propagated/computed backward one layer at a time (i.e. starting from the terminal layer \( N \)) by the following recurrence relation:

\[
\xi_{j,q} = \frac{\partial E_q}{\partial a_{j,q}} = \sum_{k=1}^{P_s} \theta_{j,k}^s \delta_{k+1,q}, \quad j = 1, \ldots, P_s,
\]

\[
\Rightarrow \xi_j^s = \Theta_j^s \delta_{j+1}^s,
\]

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

\[
\delta_{k+1,q} = \frac{\partial E_q}{\partial \text{net}_{k,q}} = \xi_{k+1}^{s+1} f_k^{s+1}(\text{net}_{k,q}), \quad k = 1, \ldots, P_{s+1}.
\]

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 \( \xi_{k,q}^{s+1} \) of the \( P_{s+1} \)-dimensional vector \( \xi_k^{s+1} \) with its corresponding node derivative \( f_k^{s+1}(\cdot) \) to generate the \( P_{s+1} \)-dimensional vector \( \delta_{k,q}^{s+1} \) in Eq. (7);

(C) \( 2P_s P_{s+1} \) operations for a matrix-vector product \[\text{i.e., a } P_s \times P_{s+1} \text{ parameter matrix } \Theta_j^s \text{ times a } P_{s+1} \text{-dimensional vector } \delta_{j+1}^s, \text{ a result of the previous product (B)], which yields a } P_s \text{-dimensional sensitivity vector } \xi_j^s.

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 \( \theta_{j,k}^s \) with a given datum \( q \) at layer \( s \) (\( 1 \leq s \leq N - 1 \)) can be calculated by:

\[
g_{j,k,q}^s = \frac{\partial E_q}{\partial \theta_{j,k,q}} = a_{j,q}^s [\xi_{k,q}^{s+1} f_k^{s+1}(\text{net}_{k,q})], \quad j = 1, \ldots, P_s, \quad k = 1, \ldots, P_{s+1} + 1,
\]

\[
\Rightarrow g_{j,k,q}^s = (\text{reshape}) \leftarrow G_{j,q}^s = a_{j,q}^s \chi(\delta_{k,q}^{s+1}) T.
\]

In words, computing the \((P_s + 1)P_{s+1}\)-dimensional gradient vector \( g_{j,q}^s \) for datum \( q \) costs \((P_s + 1)P_{s+1}\) operations for performing the outer product, denoted by “\( \times \)” of a \([P_s + 1] \times 1\) incoming-signal column vector \( a_{\cdot,q}^s \) with a \([1 \times P_{s+1}]\) sensitivity row vector \((\delta_{k,q}^{s+1})^T\) already computed in Process 4, yielding gradients in a \((P_s + 1) \times P_{s+1}\) matrix \( G_{j,q}^s \) for the associated weight matrix \( \Theta_j^s \). 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:

\[
g_{j,k,q}^s = \sum_{q=1}^d g_{j,k,q}^s.
\]

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) \( \eta \) in either of the next two forms:

- On-line incremental (pattern-by-pattern) mode learning (or incremental gradient) method:
  \( \theta_{j,k}^s \leftarrow \theta_{j,k}^s - \eta \delta_{j,k,q}^s \). Hence, it costs \( 2(P_s + 1)P_{s+1} \) per datum.

- Off-line batch-mode learning (or steepest descent) method:
  \( \theta_{j,k}^s \leftarrow \theta_{j,k}^s - \eta \delta_{j,k,q}^s, \text{ all} \). 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 \( g \) can be expressed as:

\[
g = J^T r,
\]

where \( J \) denotes the \( m \times n \) Jacobian matrix of the residual vector. Now, an alternative popular method, a Newton-type method, can be written as:

\[
H \Delta \theta = -J^T r,
\]

where \( H \) denotes a Hessian matrix. If we simply consider a Gauss-Newton model (i.e., \( J^T J \)), then the method with Equation (9), known as the Gauss-Newton method, is the normal equations approach in a least-squares sense. When \( J \) is ill-conditioned or nearly rank-deficient, we can proceed with the Jacobian system:

\[
J \Delta \theta = -r
\]

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 \((m > n)\), forming the

\footnote{Our definition of “sensitivity” \( \xi \) (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 125 and 326 in ref. [8], which corresponds to \( \delta \) (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 \( N \)).}
Gauss-Newton model Hessian $J^TJ$ ($mn^2$ 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, $J^TJ$ can be formed at a smaller cost than $mn^2$ (see ref. [10] for more details). For a large-scale problem, instead of these 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 $d$ for $\Delta \theta = \eta d$ with:

$$d = -g_{\text{now}} + \beta \Delta \theta_{\text{old}}, \quad (10)$$

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

$$\beta = \frac{g_T^T(g_{\text{now}} - g_{\text{old}})}{g_{\text{old}}^T g_{\text{old}}} \quad (11)$$

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

1. Search direction determination:
   - Processes 1 to 5
     $\Rightarrow \beta$-computation (costs $5n$) in Eq. (11)
     $\Rightarrow d$-computation (costs $2n$) in Eq. (10);

2. Step size determination:
   - Repeat function evaluations (i.e., Processes 1 through 3) until we find $\eta_1$ and $\eta_2$ that satisfy $E(\theta_{\text{now}}) > E(\theta_{\text{now}} + \eta_1 d) < E(\theta_{\text{now}} + \eta_2 d)$, and then compute $\eta$ (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:

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

plus extra (less important) costs $O(n)$ independent of $d$, which include operations necessary for computing $\beta, d$, and $\eta$ as well as parameter-updating with $\theta_{\text{next}} = \theta_{\text{now}} + \eta d$. 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.

### 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 $\text{tanh}(\cdot)$ 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 $\exp(\cdot)$ [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

<table>
<thead>
<tr>
<th>Process</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Forward pass [FP]</td>
<td>$dP_N$ for terminal sensitivity in Eq. (5)</td>
</tr>
<tr>
<td>2</td>
<td>+ $d \sum_{k=1}^{N-1} P_{s+1} (V_{s+1} + 1)$ for $\delta$ in Eq. (7)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+ $2 \eta d$ for error computation</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$dP_N$ for incremental or $2n$ (twice as many) for batch</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$d \sum_{k=1}^{N-1} (P_s + 1) P_{s+1} = 2n$ for incremental or $2n$ (independent of $d$) for batch</td>
<td></td>
</tr>
</tbody>
</table>

\(^6\)Another evidence can be found in assembly codes, wherein the number of instructions associated with the transcendental function $\exp(\cdot)$ 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 338 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 \( j \)th row of \( \mathbf{J} \) are multiplied first by a single \( \mathbf{r} \) component \( (r_j) \) 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 \( (J_{ij}, i = 1, \ldots, m) \) in the \( j \)th 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 multi-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 \((P_N - 1)(P_N - 1)\) 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 \((d = 1,024 \text{ and } n = 100)\), 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:

<table>
<thead>
<tr>
<th>Batch 10-9-1 MLP</th>
<th>Approx. flops</th>
<th>Exec. time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward pass</strong></td>
<td>(618d)</td>
<td>(5.7 \times 10^{-3})</td>
</tr>
<tr>
<td>Process 1</td>
<td>(218d)</td>
<td>(1.8 \times 10^{-3})</td>
</tr>
<tr>
<td>Process 2</td>
<td>(400d)</td>
<td>(3.9 \times 10^{-3})</td>
</tr>
<tr>
<td>Process 3</td>
<td>(Omitted)</td>
<td>(Omitted)</td>
</tr>
<tr>
<td><strong>Backward pass</strong></td>
<td>(287d)</td>
<td>(2.6 \times 10^{-3})</td>
</tr>
<tr>
<td>Process 4</td>
<td>(69d)</td>
<td>(0.7 \times 10^{-3})</td>
</tr>
<tr>
<td>Process 5</td>
<td>(218d)</td>
<td>(1.9 \times 10^{-3})</td>
</tr>
<tr>
<td>Process 6</td>
<td>(128)</td>
<td>(Negligible)</td>
</tr>
</tbody>
</table>

The second column “Approx. flops” was obtained with \( V_i = 4 \) and \( T_s = 40 \), wherein \( T_s \) was approximated by a comparison between the execution time measured for \( \text{tanh}(\cdot) \) 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:

\[
\text{Ratio}_{\text{Approx. flops}} = \frac{\text{Cost(backward pass)}}{\text{Cost(forward pass)}} = \frac{287}{618} = 0.464
\]

\[
\text{Ratio}_{\text{Execution}} = \frac{\text{Cost(backward pass)}}{\text{Cost(forward pass)}} = \frac{2.6}{5.7} = 0.456
\]

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 \( N \).

<table>
<thead>
<tr>
<th>Incremental mode 10-3-18 MLP ((n = 305 \text{ and } d = 1,000))</th>
<th>Approx. flops</th>
<th>Exec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>(383d)</td>
<td>3.3 m/sec</td>
</tr>
<tr>
<td></td>
<td>1,040</td>
<td>9.7 m/sec</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>(443d)</td>
<td>4.1 m/sec</td>
</tr>
<tr>
<td></td>
<td>546d</td>
<td>4.2 m/sec</td>
</tr>
<tr>
<td><strong>Ratio</strong></td>
<td>1.23</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td>0.49</td>
<td>0.43</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incremental mode 20-10-10-20 MLP ((n = 540 \text{ and } d = 1,000))</th>
<th>Approx. flops</th>
<th>Exec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>(1,940d)</td>
<td>17.2 m/sec</td>
</tr>
<tr>
<td></td>
<td>2,740</td>
<td>25.4 m/sec</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>(2,300d)</td>
<td>22.1 m/sec</td>
</tr>
<tr>
<td></td>
<td>2,400</td>
<td>22.2 m/sec</td>
</tr>
<tr>
<td><strong>Ratio</strong></td>
<td>1.22</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td>0.89</td>
<td>0.87</td>
</tr>
</tbody>
</table>

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) $T_n$ 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 Electro-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


