[7] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, Inc., New Jersey, 1984.

[8] Stephen M. Pizer with Victor L. Wallace, *To Compute Numerically, Concepts and Strategies*, Little, Brown and Company, Boston•Toronto, 1983.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing (PDP): Exploration in the Microstructure of Cognition (Vol. 1)*, Chapter 8, MIT Press, Cambridge, Massachusetts, 1986.

[10] P. J. Werbos. Beyond regression: New tools for prediction and analysis in the behavior science. Ph.D. Thesis, Harvard University, Cambridge, 1974.

computation. On the other hand, for network learning, at least 14-16 bits of precision must be used for the weights to avoid having the training process divert too much from the trajectory of the high precision computation.

References

[1] D. Hammerstrom. A VLSI architecture for high-performance, low cost, on-chip learning. In Proc. *Int'l Joint Conf. on Neural Networks,* pp. II:537-543, San Diego, CA, June 1990.

[2] T. E. Baker. Implementation limits for artificial neural networks. Master Thesis, Dept. of Computer Science and Engr., Oregon Graduate Institute of Science and Technology, 1990.

[3] J. N. Hwang and P. S. Lewis. From nonlinear optimization to neural network learning. In Proc. *24th Asilomar Conf. on Signals, Systems, & Computers*, Pacific Grove, CA, November 1990.

[4] S. Y. Kung and J. N. Hwang. Parallel architectures for artificial neural nets. In Proc. *Int'l Conf. on Neural Networks,* Volume 2, pp. 165-172, San Diego, California, July 1988.

[5] S.Y. Kung, J.N. Hwang, A unified architecture for artificial neural networks. *Journal of Parallel and Distributed Computing*, pp. 358-387, April, 1989.

[6] P. S. Lewis and J. N. Hwang. Recursive least squares learning algorithms for neural networks. In Proc. *SPIE's Int'l Symposium on Optical and Optoelectronic Applied Science and Engr.*, pp. 28-39, San Diego, July 1990.

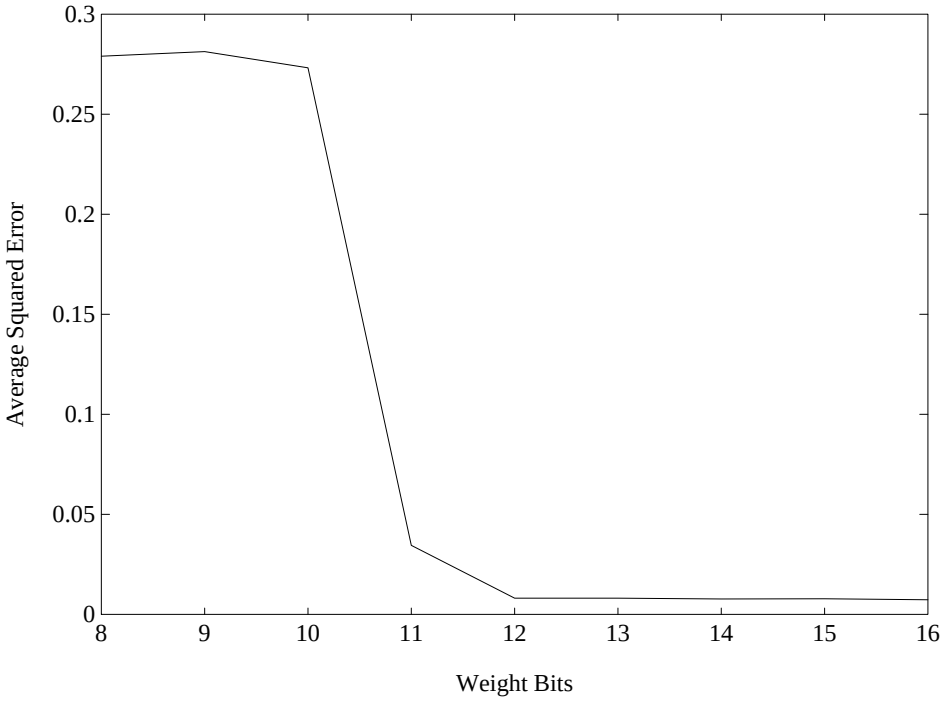

Figure 10: The average squared differences between the desired and actual outputs of the XOR problem after the network converges.

5 Concluding Remarks

The paper is devoted to the derivation of the finite precision error analysis techniques for neural network implementations, especially analysis of the back-propagation learning of MLP's. This analysis technique is proposed to be more versatile and to prepare the ground for a wider variety of neural network algorithms: recurrent neural networks, competitive learning networks, and etc. All these networks share similar computational mechanisms as those used in back-propagation learning. For the forward retrieving operations, it is shown that 8-bit weights are sufficient to maintain the same performance as using high precision

ture of the XOR problem, a soft convergence is good enough for the termination of training. Therefore, at the predicted point of 12-13 bits of weights, the squared difference curve dives.

Another interesting observation is worthwhile to mention: the total finite precision error in a single iteration of weight updating is mainly generated in the final jamming operators in the computation of the output delta, hidden delta, and weight update. Therefore, even though it is required to have at least 13 to 16 bits assigned to the computation of the weight update and stored as the total weight value, the number of weight bits in the computation of forward retrieving and hidden delta steps of learning can be as low as 8 bits without excessive degradation of learning convergence and accuracy.

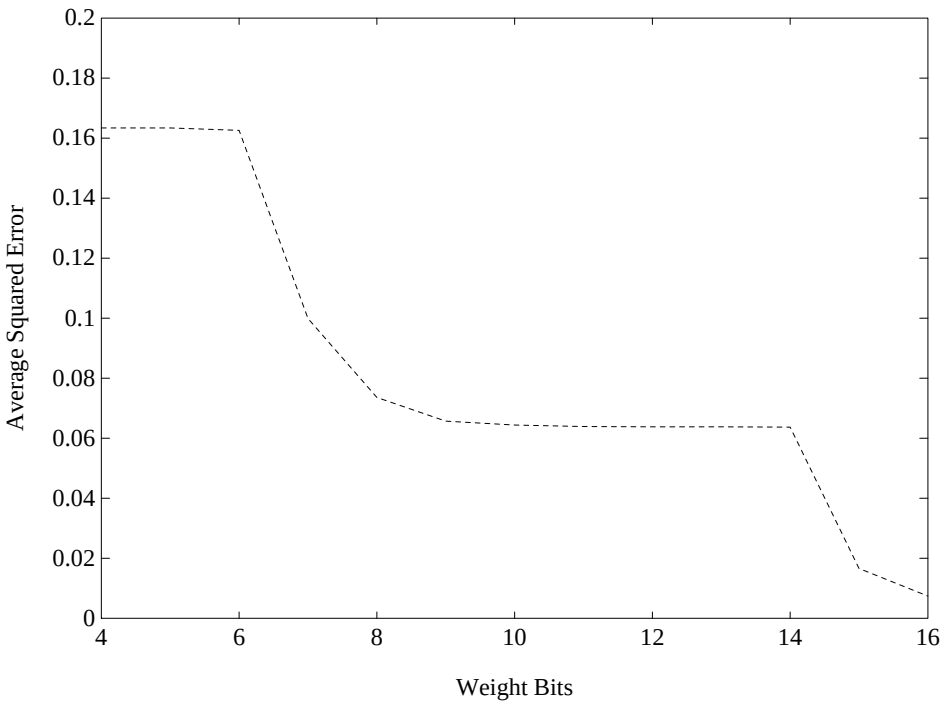

Figure 9: The average squared differences between the desired and actual outputs of the 2-D regression problem after the network converges.

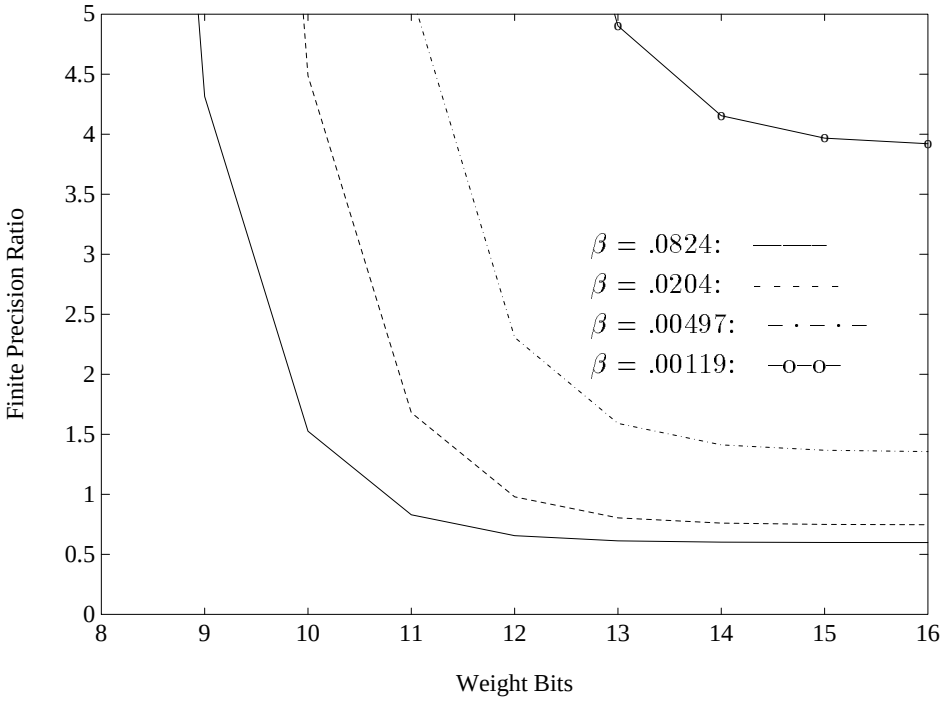

Figure 8: The statistical evaluation values of the finite precision ratio for the hidden-layer weights in a 2-layer MLP. Four different stages of learning are evaluated.

lations were performed based on the same choices of bit sizes for each component given in Section 3.6 vs. different number of bits assigned to $\{w_{ij}\}$ and $\{\Delta w_{ij}\}$. Figure 9 shows the average (of 256 training data) squared difference between the desired and actual outputs of the 2-D regression problem after the network converges (a hard convergence is usually required in this kind nonlinear regression problem). Note that, at the predicted point (around 15-16 bits of weights), the squared difference curve dives. That implies the inability to converge to the desired mapping when the number of bits for the weights is less than 16 bits. Similar supporting results are also observed in the XOR classification problem using an MLP with 2 inputs, 3 hiddens, and 1 output (see Figure 10). Due to the classification na-

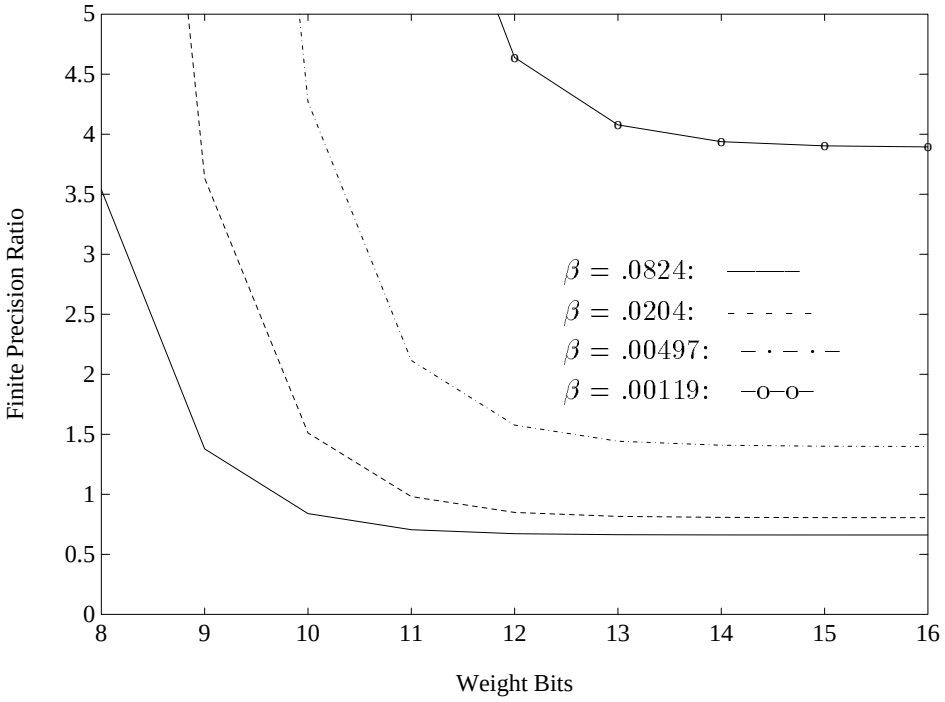

Figure 7: The statistical evaluation values of the finite precision ratio for the top-layer weights in a 2-layer MLP. Four different stages of learning are evaluated.

4.2 Simulation Results for Iterative Learning

To verify the theoretical evaluation of the impact caused by the finite precision computation, a simple regression problem is designed, which maps 2-dimensional inputs, $\{x_1,\ x_2\}$, into 1-dimensional outputs, $\{y\}$:

$$y = \frac{1}{2}(x_1 + x_2)^2 \cdot e^{-x_1^2 - x_2^2 + 1} \tag{52}$$

An MLP containing 2 input neurons, 8 hidden neurons, and 1 output neuron, is adopted. There are 256 pairs of randomly selected data for training. Finite precision learning simu-

Based on the same practical choices of finite precision bit size given in Section 3.6 vs. the number of bits (say k bits) assigned to the weights $\{w_{ij}\}$ and weight updates $\{\Delta w_{ij}\}$, we can statistically evaluate this ratio at several different stages of learning. Figure 7 shows the statistical evaluation values of the finite precision ratio for the weights connecting the neurons between the hidden and the output layers in a 2-layer MLP. 4 different values of β are used, which represent four different stages of learning: early stage ($\beta = 0.0824$), middle stage ($\beta = 0.0204$), soft convergence stage ($\beta = 0.00497$), and hard convergence stage ($\beta = 0.00119$). Figure 8 shows the statistical evaluation values of the finite precision ratio for the weights connecting the neurons between the hidden and the input layers for these four different values of β.

Note that the finite precision ratio curves gradually (along various stages of learning) dive around the region where the number of bits for weights are 12-14 bits for soft convergence stage, 14-16 for hard convergence stage of learning, and then get almost steady when the number of bits is increased further. This dive point indicates the potential strong disturbance to the convergence and accuracy of back-propagation leaning in the long run. Therefore, it gives a good guideline to which number of bits are required for the weights in the learning so as to have similar learning convergence and accuracy as that attained using high precision computation. It is also interesting to note that at the later stage of the learning the impact of the finite precision error is getting larger due to the smaller values of β when the network is fine-tuning its weights.

search approximation, the real effect to the learning convergence and accuracy due to finite precision computation will be difficult to measure. Therefore, the statistically evaluated average sum of the squares of $\epsilon_{\Delta w}$, by itself, can not determine a network's propensity to learn.

4.1 Ratio of Finite Precision Errors

A more meaningful measure, ρ, which potentially indicates the effect of finite precision computation to the weight updating, can be defined as the ratio of the statistical average sum of the squares of finite precision weight updating error $\epsilon_{\Delta w}$ and that of full precision weight updating magnitude Δw_{ij}^2:

$$\rho \equiv \frac{E\left[\epsilon_{\Delta w}^2\right]}{E\left[\Delta w_{ij}^2\right]} \tag{50}$$

This ratio serves as a useful indicator of the additional impact caused by finite precision computation on top of that caused by the gradient descent approximation of back-propagation learning. The ratio depends not only on the number of bits assigned to the finite precision computation, but on the current stage of learning progress, which can be specified by the distribution of the difference between the desired and actual output $\{t_j - x_{L,j}\}$. More specifically,

$$\beta = E\left[(t_j - x_{L,j})^2\right] \tag{51}$$

where we assume that $\epsilon_{x_{L,j}} = \epsilon_{t_j} = 0$, since the ability to learn should depend on the ability to learn these finite precision values.

the simulated evaluation of finite precision error introduced in neurons of the first hidden layer, and the upper dashed curve shows that of the output layer. Both curves match quite consistently with those produced by statistical evaluations.

4 Finite Precision Analysis for Iterative Learning

As discussed in Section 3.1, back-propagation learning involves all four consecutive steps of computations: forward retrieving, output delta computation, hidden delta computation, and weight updating. Therefore, for each weight updating iteration at the presentation of any training pattern, the finite precision errors $\epsilon_{\Delta w}$ introduced to $\{\Delta w_{l,i,j}\}$ given in Eq. (48) is in fact a propagated result of the error generated from the previous three steps. Therefore, the final mathematical expression of finite precision error for a *single weight updating iteration* can be formulated in a straightforward manner based on the existing derivations given in Eqs. (38), (44), (46), and (48). The statistical evaluation value for the average sum of the squares of weight updating error $\epsilon_{\Delta w}$ due to the finite precision computation of a single learning iteration can thus be computed.

The back-propagation learning discussed above is simply a nonlinear optimization problem based on simple gradient descent search, with an elegant way of computing the gradients using the chain rule on the layers of the network. This gradient descent search updates the weights based only on the first derivative approximation of the error surface with the updating of each individual weight being independent of the others [6]. Therefore, even if the approach is computationally efficient, it can behave very unwisely and converge very slowly for complex error surfaces. Due to the strong influence introduced in the gradient descent

of the decimal), and then download the well trained finite precision portion (8 bits total, or 4 bits to the right of the decimal) of the weights into the hardware. The performance degradations due to this finite precision conversion is almost negligible.

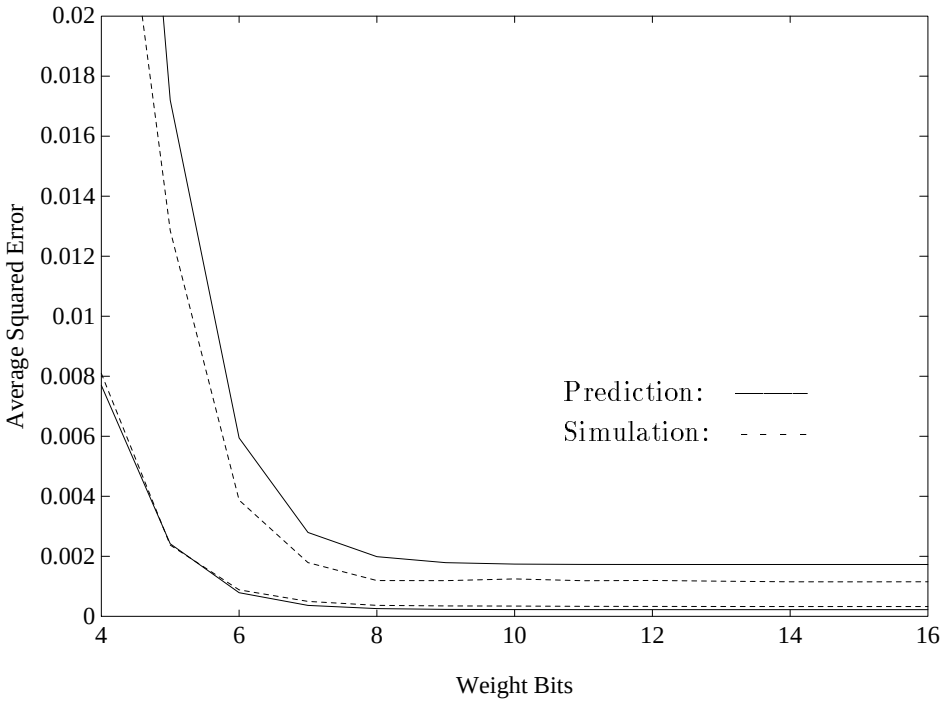

Figure 6: Statistically evaluated and simulation evaluated values of $E[\epsilon_y^2]$ introduced to neurons of the first and second hidden layers.

Simulations are also conducted to verify the statistical evaluations. A 2-layer perceptron with 100 inputs, 100 hidden neurons and 100 output neurons is simulated. 100 sets of randomly generated 100-dimensional input data are tested and averaged. All the weights of the network are also randomly generated. The average sum of the squares of the differences between the finite precision computation and the full precision (64 bits for all the contributing components) computation can thus be obtained. The lower dashed curve shows

place.

R.V.	Type	q	r	μ	σ^2
w_{ij}	uniform	16	3	$-1.22 \cdot 10^{-4}$	$2.13 \cdot 10^{1}$
$\epsilon_{w_{ij}}$	rounding	8	-12	$-4.77 \cdot 10^{-7}$	$5.03 \cdot 10^{-9}$
x_i	uniform	8	0	$4.98 \cdot 10^{-1}$	$8.24 \cdot 10^{-2}$
ϵ_{x_i}	truncation	16	-8	$-1.95 \cdot 10^{-3}$	$1.27 \cdot 10^{-6}$
f'_j	normal	10	5	$1.94 \cdot 10^{-2}$	$2.87 \cdot 10^{-3}$
ϵ_Λ	jamming	16	-4	0.00	$1.58 \cdot 10^{-4}$
ϵ_f	rounding	2	-5	$-3.91 \cdot 10^{-3}$	$1.72 \cdot 10^{-4}$

Table 1: Mean and variance of variables in the forward retrieving calculation.

Based on the precision assumptions given in Table (1) with various sizes of bit-allocation to the weights, Figure (6) shows the statistically evaluated average sum of the squares of ϵ_y, as defined in Eqs. (49) and (24), due to the finite precision computation of a single step forward retrieving. In these evaluations, for any weight bit number (say k-bits), the weight always contains one sign bit, 3 left-of-decimal bits, and $k-4$ right-of-decimal bits. The lower solid curve shows the statistical evaluation of finite precision error introduced in neurons of the first hidden layer and the upper solid curve shows that of the second hidden layer (or the output layer in a 2-layer perceptron). Note that the statistical evaluation of errors shows a dive-in at around 8-bit weights. This fact suggests that for the implementation of finite precision hardware for only the forward retrieving purpose, we can train the network using high precision computation (under the constraint of only sign plus 3 bits to the left

Finite Precision Error in Forward Retrieving: The expected forward retrieving error can be calculated for both a single layer of neurons and for multiple layers of neurons by propagating upward the finite precision errors of the lower layers. First, a simplification may be made to Equation (38). The multiply and accumulate steps can be computed without generating any error if enough bits (e.g., 24 bits) are used for each of the intermediate steps, y_{i*}, and y_{i+}. In this case, $\epsilon_{y_{i*}} = \epsilon_{y_{i+}} = 0$. This is practical since the expense of accumulator precision is very small.

The Λ operator now reduces the final 24-bit sum to an 8-bit (one sign bit, 3 bits to the left and 4 bits to the right of the decimal) value which is used as the input for the sigmoid look-up table. Equation (38) may now be rewritten as

$$\epsilon_y = f'_j \sum_{i=1}^{n} w_{ij}\epsilon_{x_i} + f'_j \sum_{i=1}^{n} x_i \epsilon_{w_{ij}} + f'_j \epsilon_\Lambda + \epsilon_{f_j} \tag{49}$$

Before invoking the central limit theorem on the sums, it is necessary to know the distributions of the random variables w_{ij}, $\epsilon_{w_{ij}}$, x_i, ϵ_{x_i}, $\epsilon_{f'_j}$, ϵ_Λ, and ϵ_{f_j}. Table 1 shows the the statistically evaluated values for each contributing component of Eq. (49) based on the assumptions of bit sizes given above. The evaluation starts with 16-bit weights which are uniformly distributed across the entire range, $[-8,\ 8)$, and weight error comes from truncating 24-bit weights to 16 bits. For an input neuron, x_i is an 8-bit value uniformly distributed over $[0.0,\ 1.0)$ which has been truncated from a 24-bit value. Therefore, ϵ_{x_i} is the truncation error with $r = -8$ and $q = 16$. The distribution of f'_j is approximated as a function of a normally distributed random variable. ϵ_Λ is the error generated when the accumulated 24-bit value is jammed to become an 8-bit value, and ϵ_{f_j} is the error generated in the lookup table which is approximated to be the same as rounding 2 bits at the $r = -5$

the finite precision error for weight update (see Eq. (9)):

$$\epsilon_y \equiv \epsilon_{\Delta w_{l,i,j}} = \delta_j x_i \epsilon_\eta + \eta x_i \epsilon_{\delta_j} + \eta \delta_j \epsilon_{x_i} + x_i \epsilon_{y_{j*}} + \epsilon_{*\Lambda} \tag{48}$$

3.6 Statistical Evaluation of the Finite Precision Errors

Given the analytical expressions of all the finite precision errors associated with the forward retrieving and back-propagation of an MLP, a statistical evaluation of these errors is undertaken. This evaluation is based on the mean and variance analysis using truncation, jamming, and rounding techniques, and also based on statistical properties of independent random variables, sums of independent random variables, and sums of products of independent random variables discussed in Sections 2.3, 2.4, and 2.5. The first step is to choose the precision of each component which will be employed at each step in the problem. A practical limited precision implementation of the MLP algorithm might use precisions as follows [2].

1. All neurons have 8-bit (8 bits to the right of the decimal) inputs, outputs, and targets with range $[0.0,\ 1.0)$.

2. All weights and biases use 16 bits (one sign bit, 3 bits to the left and 12 bits to the right of the decimal) with range $[-8.0,\ 8.0)$.

3. The output δ is 8 bits (one sign bit and 7 bits all to the right of the decimal) with range $[-0.5,\ 0.5)$.

4. Hidden δ uses 16 bits (one sign bit, 3 bits to the left and 12 bits to the right of the decimal) with range $[-8.0,\ 8.0)$.

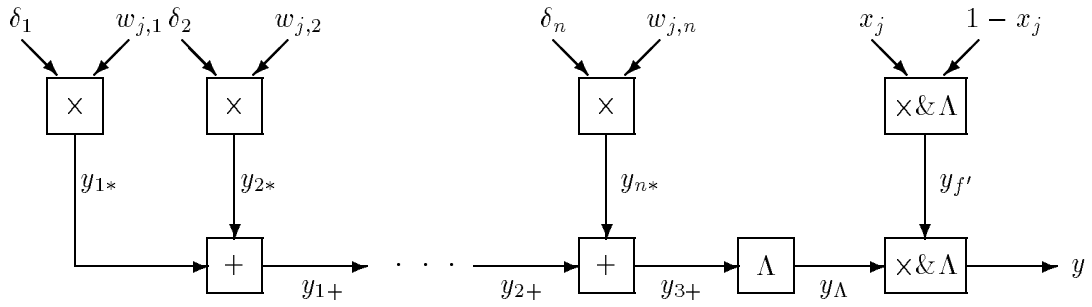

Figure 4: Calculation graph for $y = \delta_{l,j}$ in a hidden neuron.

3.5 Finite Precision Analysis of Weight Update

From Eq. (30), the calculation graph for the computation of the weight update (without the momentum term), with simplified notation, is shown in Figure (5).

$$y = \eta \cdot \delta_j \cdot x_i \tag{47}$$

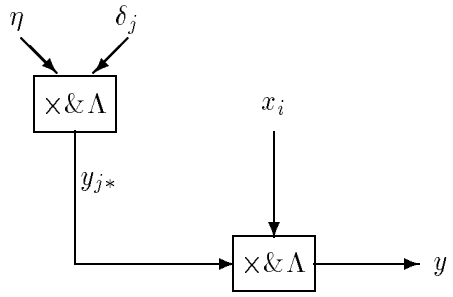

Figure 5: Calculation graph for $y = \Delta w_{l,i,j}$.

Following similar partial derivative evaluations using Eq. (47), we can again compute

$$\frac{\partial y}{\partial x_j} = -f_j' + (t_j - x_j)\frac{\partial f_j'}{\partial x_j} = -f_j' + (t_j - x_j)(1 - 2x_j) \quad (40)$$

$$\frac{\partial y}{\partial t_j} = f_j' \quad (41)$$

$$\frac{\partial y}{\partial y_{f'}} = t_j - x_j \quad (42)$$

$$\frac{\partial y}{\partial y_{j-}} = f_j' \quad (43)$$

Substituting for the partial derivatives and individual error terms, the overall finite precision error for the output delta computation is:

$$\epsilon_y \equiv \epsilon_{\delta_{L,j}} = \left(-f_j' + (t_j - x_j)(1 - 2x_j)\right)\epsilon_{x_j} + f_j'\epsilon_{t_j} + (t_j - x_j)\epsilon_{f_j'} + f_j'\epsilon_{y_{j-}} + \epsilon_{*\Lambda} \quad (44)$$

3.4 Finite Precision Analysis of Hidden Delta

From Eq. (31), the calculation graph for the computation of back-propagated error in a hidden layer neuron, with simplified notation, is shown in Figure (4).

$$y = f_j' \cdot \sum_i^n \delta_i \cdot w_{j,i} = x_j(1 - x_j) \cdot \sum_i^n \delta_i \cdot w_{j,i} \quad (45)$$

Following similar partial derivative evaluations using Eq. (45), we can again compute the finite precision error for the hidden delta (see Eq. (9)):

$$\begin{aligned} \epsilon_y \equiv \epsilon_{\delta_{l,j}} = {} & f_j' \sum_{k=1}^n w_{j,k}\epsilon_{\delta_k} + f_j' \sum_{k=1}^n \delta_k \epsilon_{w_{j,k}} + (1 - 2x_j)\epsilon_{x_j} \sum_{k=1}^n \delta_k w_{j,k} + \\ & f_j' \sum_{k=1}^n \epsilon_{y_{k*}} + \epsilon_{y_{f'}} \sum_{k=1}^n \delta_k w_{j,k} + f_j' \sum_{k=1}^{n-1} \epsilon_{y_{k+}} + f_j'\epsilon_\Lambda + \epsilon_{\times\Lambda} \end{aligned} \quad (46)$$

$$\frac{\partial y}{\partial y_{i+}} = \frac{\partial f(y_{i+} + \sum_{k=i+1}^{n} y_{k*})}{\partial y_{i+}} = f_j' \tag{36}$$

$$\frac{\partial y}{\partial y_{\Lambda}} = \frac{\partial f(y_{\Lambda})}{\partial y_{\Lambda}} = f_j' \tag{37}$$

By substituting the values for the partial derivatives in Eqs. (34) to (37) and the generated and propagated errors for variables and operators into Eq. (9),

$$\epsilon_y \equiv \epsilon_{x_{l+1,j}} = f_j' \sum_{i=1}^{n} w_{i,j}\epsilon_{x_i} + f_j' \sum_{i=1}^{n} x_i \epsilon_{w_{i,j}} + f_j' \sum_{i=1}^{n} \epsilon_{y_{i*}} + f_j' \sum_{i=1}^{n-1} \epsilon_{y_{i+}} + f_j' \epsilon_{y_{\Lambda}} + \epsilon_{f_j} \tag{38}$$

3.3 Finite Precision Analysis of Output Delta

From Eq. (32), the calculation graph for the computation of back-propagated error in an output neuron, with simplified notation, is shown in Figure (3).

$$y = f_j' \cdot (t_j - x_j) = x_j(1 - x_j) \cdot (t_j - x_j) \tag{39}$$

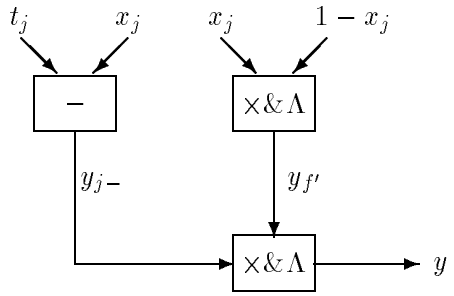

Figure 3: Calculation graph for $y = \delta_{L,j}$ in an output neuron.

Again, to carry out the analytical formula as given in Eq. (9) for the output delta computation of an MLP, the partial derivatives of Eq. (39) are evaluated.

3.2 Finite Precision Analysis of Forward Retrieving

Explicitly following the procedure discussed in Section 2.2, the calculation graph of the forward retrieving operation, with simplified notation (see Eq. (29)), in an MLP is shown in Figure (2):

$$y = f(\sum_i w_{i,j} \cdot x_i) \tag{33}$$

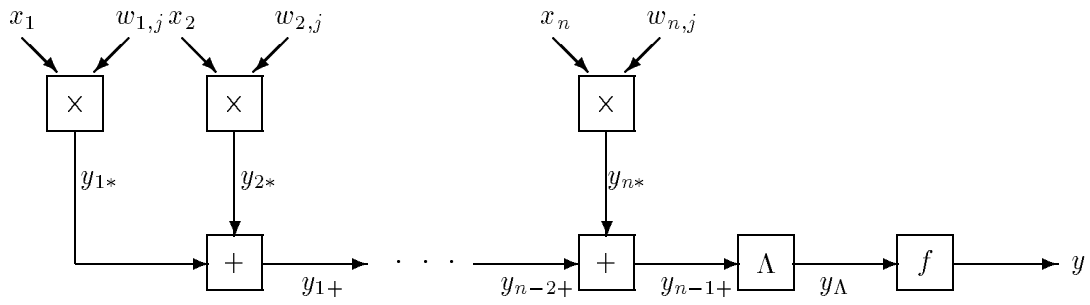

Figure 2: Calculation graph for the forward retrieving, $y = x_{l+1,j}$, of an MLP, where Λ denotes a truncation, jamming, or rounding operator.

To carry out the analytical formula as given in Eq. (9) for the forward retrieving of an MLP, several partial derivatives need to be computed from Eq. (33):

$$\frac{\partial y}{\partial x_i} = f'_j w_{i,j} \quad \text{and} \quad \frac{\partial y}{\partial w_{i,j}} = f'_j x_i \tag{34}$$

where $f'_j = y(1-y)$.

$$\frac{\partial y}{\partial y_{i*}} = \frac{\partial f(\sum_{i=1}^{n} y_{i*})}{\partial y_{i*}} = f'_j \tag{35}$$

The operations in the forward retrieving of an L-layer perceptron can be formulated as a *forward affine transformation* interleaved with a nonlinear scalar activation function:

$$x_{l+1,j} = f(\sum_i w_{l+1,i,j} \cdot x_{l,i}), \qquad \forall j, \ \ \forall l \tag{29}$$

where $x_{l,i}$ denotes the activation value of the i^{th} neuron at the l^{th} layer, $w_{l+1,i,j}$ denotes the synaptic weight interconnecting the i^{th} neuron at the l^{th} layer and the j^{th} neuron at the $l+1^{th}$ layer. The nonlinear activation function, $f(\cdot)$, is usually taken to be sigmoidal.

The learning of an MLP follows the iterative gradient descent approach with the following update at each presentation of a training data pair [10, 9]:

$$\Delta w_{l,i,j} = \eta \cdot \delta_{l,j} \cdot x_{l-1,i} \tag{30}$$

where the computation of the back-propagated error, $\delta_{l,j}$, can again be formulated as a *backward affine transformation* interleaved with the post-multiplication of the derivatives of the nonlinear activation function:

$$\delta_{l,j} = f'_{l,j} \cdot \sum_i \delta_{l+1,i} \cdot w_{l+1,j,i} \tag{31}$$

with initial output-layer propagated error being

$$\delta_{L,j} = f'_{L,j} \cdot (t_j - x_{L,j}) \,. \tag{32}$$

3 Application to Neural Network Retrieving and Learning

It has been shown that operations in both the retrieving and the learning phases of most of the neural network models can be formulated as a linear *affine transformation* interleaved with simple linear or nonlinear scalar operation. In terms of the hardware implementations, all these formulations call for a MAC (multiply and accumulation) processor hardware [4, 5, 1]. Without loss of generality, we will specifically discuss the multilayer perceptron neural network model and the back-propagation learning [10, 9].

3.1 Forward Retrieving and Back Propagation of an MLP

Given a trained (fixed-weight) MLP, the retrieving phase receives the input test pattern, $\{x_{0,i}\}$, and propagates forward through the network to compute the activation values of the output layer, $\{x_{L,j}\}$, which will be used as the indicator for classification or regression purposes. On the other hand, in the commonly used learning phase of an MLP, the input training pattern, $\{x_{0,i}\}$, is first propagated forward through the network and the activation values, $\{x_{l,j}\}$, are computed according to the same forward operations used in the retrieving phase. Then the output activation values, $\{x_{L,j}\}$, are compared with the target values $\{t_j\}$, and the value of the output delta, $\{\delta_{L,j}\}$, for each neuron in the output layer is derived. These error signals are propagated backward to allow the recursive computation of the hidden delta's, $\{\delta_{l,j}\}$, as well as the update values of the weights, $\{\Delta w_{l,i,j}\}$, at each layer. While many methods have been proposed to accelerate learning by estimating the local curvature of the training error surface using second order derivative information, the discussion of these methods are beyond the scope of this paper, and can be referred to [3].

a normal curve as $N \to \infty$. For discrete random variables, the probabilities tend to the samples of a normal curve. Normalization can be achieved in a couple of different ways. If $\{x_i\}$ are discrete random variables with mean μ and variance σ^2, then the sum $\overline{x}$;

$$\overline{x} = \sum_{i=1}^{N} x_i \quad \text{results in} \quad \mu_{\overline{x}} = N\mu, \quad \text{and} \quad \sigma_{\overline{x}}^2 = N\sigma^2 \tag{25}$$

Invoking the central limit theorem, the probability that the sum of the random variables, $\overline{x}$, is equal to the discrete value x_k which approaches the sample of a normal curve when N is large.

$$p_k \equiv Pr\{\overline{x} = \overline{x}_k\} \simeq \frac{1}{\sigma_{\overline{x}}\sqrt{2\pi}} e^{-(\overline{x}_k - \mu_{\overline{x}})^2/2\sigma_{\overline{x}}^2} \tag{26}$$

Sum of Products of Independent Random Variables: The central limit theorem can be extended to cover the case where the random variable being summed is a product of random variables. Say that for independent random variables $\{x_i\}$ and $\{y_i\}$,

$$\overline{xy} = \sum_{i=1}^{N} x_i y_i \tag{27}$$

then the probability density of the random variable $\overline{xy}$ approaches a normal curve for large N with mean and variance equal to

$$\mu_{\overline{xy}} = N\mu_{xy}, \qquad \text{and} \qquad \sigma_{\overline{xy}} = N\sigma_{xy}^2 \tag{28}$$

1. Multiplication by a constant.

$$\mu_{ax} = a\mu_x, \qquad \sigma^2_{ax} = a^2\sigma^2_x \tag{20}$$

2. Sum of two independent random variables.

$$\mu_{x+y} = \mu_x + \mu_y, \qquad \sigma^2_{x+y} = \sigma^2_x + \sigma^2_y \tag{21}$$

3. Product of two independent random variables.

$$\mu_{xy} = \mu_x\mu_y, \qquad \sigma^2_{xy} = \sigma^2_x\sigma^2_y + \sigma^2_x\mu^2_y + \sigma^2_y\mu^2_x \tag{22}$$

2.5 Statistical Properties of Sum of Independent Random Variables

Expected Squared Error: The expected squared error can be written in terms of the mean and variance of the error. Consider a set of errors which are independent random variables, $\{\epsilon_i\}$, with mean μ and variance σ^2. Then, the expected value of the average sum of the squares of $\{\epsilon_i\}$ can be written,

$$E\left[\frac{1}{N}\sum_{i=1}^{N}\epsilon_i^2\right] = \frac{1}{N}N\,E\left[\epsilon_i^2\right] = E\left[\epsilon_i^2\right] \tag{23}$$

Noting that $\sigma^2 = E\left[\epsilon_i^2\right] - E^2\left[\epsilon_i\right]$, the expected value of the average sum squared errors is equal to:

$$E\left[\epsilon_i^2\right] = \sigma^2 + \mu^2 \tag{24}$$

Central Limit Theorem: The central limit theorem [7] states that if $\{x_i\}$ are independent random variables, then the density of their sum, properly normalized, tends to

$$\mu = \sum_{i=-2^q+1}^{2^q-1} p_i \cdot i \cdot 2^{r-q} = 2^{-q-1} \cdot 2^{r-q} \sum_{i=-2^q+1}^{-1} i + 2^{-q-1} \cdot 2^{r-q} \sum_{i=1}^{2^q-1} i = 0 \quad (14)$$

$$\sigma^2 = \sum_{i=-2^q+1}^{2^q-1} p_i \cdot \left(i \cdot 2^{r-q}\right)^2 = 2^{2r-3q} \sum_{i=1}^{2^q-1} i^2 \quad (15)$$

Rounding: Rounding generates error which is uniformly distributed in the range $[-2^{r-1}, 2^{r-1} - 2^{r-q},]$ with each of the 2^q possible error values being of equal probability. Therefore, $p_i = 2^{-q}$ and $x_i = i \cdot 2^{r-q}$, for $i = -2^{q-1}$ to $2^{q-1} - 1$. The mean and variance are computed.

$$\mu = \sum_{i=-2^{q-1}}^{2^{q-1}-1} 2^{-q} \cdot i \cdot 2^{r-q} = 2^{r-2q} \sum_{i=-2^{q-1}}^{2^{q-1}-1} i = -2^{r-q-1} \quad (16)$$

$$\sigma^2 = \sum_{i=-2^{q-1}}^{2^{q-1}-1} 2^{-q} \left(i \cdot 2^{r-q} + 2^{r-q-1}\right)^2 = 2^{2r-3q} \sum_{i=-2^{q-1}}^{2^{q-1}-1} \left(i + 2^{-1}\right)^2 \quad (17)$$

Nonlinear Functions of Discrete Random Variables: For a nonlinear function of a discrete random variable, x, the mean and variance are given by:

$$\mu = E[f(x)] = \sum_i p_i \cdot f(x_i) \quad (18)$$

$$\sigma^2 = E[(f(x) - \mu)^2] = \sum_i p_i \cdot (f(x_i) - \mu)^2 \quad (19)$$

2.4 Statistical Properties of Independent Random Variables

For two independent random variables, x and y, with means μ_x, μ_y and variances σ_x^2, σ_y^2, and a constant a, the following properties of mean and variance can be shown [7].

$$\mu = E[x] = \sum_i p_i x_i \tag{10}$$

and the variance is:

$$\sigma^2 = E[(x-\mu)^2] = \sum_i p_i (x_i - \mu)^2 \tag{11}$$

where $p_i = Pr\{x = x_i\}$ is usually assumed to be uniformly distributed.

Truncation: Truncation generates error which is uniformly distributed in the range $[-2^r + 2^{r-q},\ 0]$ with each of the 2^q possible error values being of equal probability. Therefore, $p_i = 2^{-q}$ and $x_i = -i \cdot 2^{r-q}$, for $i = 0$ to $2^q - 1$. Mean and variance may then be computed.

$$\mu = -\sum_{i=0}^{2^q-1} 2^{-q} \cdot i \cdot 2^{r-q} = -2^{r-2q} \sum_{i=0}^{2^q-1} i = -\frac{2^r - 2^{r-q}}{2} \tag{12}$$

$$\sigma^2 = \sum_{i=0}^{2^q-1} 2^{-q} \left(-i \cdot 2^{r-q} + \frac{2^r - 2^{r-q}}{2} \right)^2 = 2^{2r-3q} \sum_{i=0}^{2^q-1} \left(i - \frac{2^q - 1}{2} \right)^2 \tag{13}$$

Jamming: The error generated by jamming is not uniformly distributed as the probability of the error being zero is twice the probability of the error holding any of its other possible values. The range of error is $[-2^r + 2^{r-q},\ 2^r - 2^{r-q}]$ with

$$p_i = \begin{cases} 2^{-q-1}, & i \neq 0 \\ 2^{-q}, & i = 0 \end{cases}$$

and $x_i = i \cdot 2^{r-q}$, for $i = -2^q + 1$ to $2^q - 1$. This results in $2^{q+1} - 1$ possible error values. The mean and variance of jamming error are then

4. Statistical methods discussed below are then used to evaluate the error in Eq. (9). Methods include the computation of mean and variance for various functions of random variables, as well as approximations using the central limit theorem.

2.3 Common Techniques of Finite Precision Computations

Three common techniques are used in finite precision computations: truncation, jamming, and rounding. The truncation operator simply chops the q lowest order bits off of a number and leaves the new lowest order bit, in the 2^r-th place, unchanged. The jamming operator chops off the q lowest order bits of a number and forces the new lowest order bit, in the 2^r-th place, to be a "1" if any of the q removed bits were a "1"; otherwise, the new lowest order bit retains its value. This operation is equivalent to replacing the 2^r-th bit with the logical OR of the 2^r-th bit and the q bits which are chopped off. The jamming operator has the advantage of generating error with zero mean, but generates error with a higher variance compared with the truncated one. The rounding operator also chops off the q lowest order bits of a number which will have its new lowest order bit in the 2^r-th place. If the q bit value chopped off is greater than or equal to 2^{r-1}, the resulting value is incremented by 2^r; otherwise, it remains unchanged [1].

The error generated by truncating, jamming, or rounding techniques may be considered to be a discrete random variable distributed over a range determined by the specific technique being employed. For a statistical view of the error, it is desirable to know the mean and variance of the error generated by each of these three techniques. For a discrete random variable, x, the mean is given by:

$$\prod_{k=i+1}^{n} \phi_k'\left(\phi_{k-1}\left(\cdots(\phi_{i+1}(\tilde{y}_i))\cdots\right)\right) \approx \prod_{k=i+1}^{n} \phi_k'\left(\phi_{k-1}\left(\cdots(\phi_{i+1}(y_i))\cdots\right)\right) = \frac{\partial y}{\partial y_i} \qquad (7)$$

Therefore,

$$\tilde{y} \equiv y + \epsilon_y \approx y + \epsilon_x \frac{\partial y}{\partial x} + \sum_{i=1}^{n} \epsilon_{\phi_i} \frac{\partial y}{\partial y_i} \qquad (8)$$

2.2 Error Generation and Propagation by General Compound Operators

The effects of finite precision error at the output of a general system of compound operators with multiple input variables can be calculated through an extension of the previous analysis for successive operators of a single variable [8]. The following steps are employed:

1. Break the computation into a *calculation graph* (see the example given in Figure (2)). The general *calculation graph* is made up of n operators, $\{\phi_i\}$, and m system inputs, $\{x_j\}$.

2. Number the operators, $\phi_1, \phi_2, ..., \phi_n$, such that the intermediate generated inputs, $\{y_k\}$, to an operator, ϕ_i, have lower indices than the operator output, y_i. By extending Eq. (8) to multiple inputs, the total finite precision error, ϵ_y, is given as [8]:

$$\epsilon_y = \sum_{j=1}^{m} \epsilon_{x_j} \frac{\partial y}{\partial x_j} + \sum_{i=1}^{n} \epsilon_{\phi_i} \frac{\partial y}{\partial y_i} \qquad (9)$$

3. Using the *calculation graph*, the partial derivatives, $\left\{\frac{\partial y}{\partial x_j}\right\}$ and $\left\{\frac{\partial y}{\partial y_i}\right\}$, are evaluated and substituted into Eq. (9) to give an equation for ϵ_y.

$$\tilde{y} = \phi_n(\cdots(\phi_2(\phi_1(x+\epsilon_x)+\epsilon_{\phi_1})+\epsilon_{\phi_2})\cdots)+\epsilon_{\phi_n} \tag{3}$$

If y_i is defined as the intermediate result after the first i successive operators,

$$y_i = \phi_i(\phi_{i-1}(\cdots(\phi_1(x))\cdots)) \tag{4}$$

and $y = y_n$, then

$$\begin{aligned}
\tilde{y}_i &= \phi_i(\phi_{i-1}(\cdots(\phi_2(\phi_1(x+\epsilon_x)+\epsilon_{\phi_1})+\epsilon_{\phi_2})\cdots)+\epsilon_{\phi_{i-1}})+\epsilon_{\phi_i} \\
&= \phi_i(\tilde{y}_{i-1})+\epsilon_{\phi_i} \\
&= \phi_i(\phi_{i-1}(\tilde{y}_{i-2})+\epsilon_{\phi_{i-1}})+\epsilon_{\phi_i} \\
&\approx \phi_i\left(\phi_{i-1}(\tilde{y}_{i-2})\right)+\phi_i'\left(\phi_{i-1}(\tilde{y}_{i-2})\right)\epsilon_{\phi_{i-1}}+\epsilon_{\phi_i} \\
&\approx \phi_i\left(\phi_{i-1}(\phi_{i-2}(\tilde{y}_{i-3}))\right)+\phi_i'\left(\phi_{i-1}(\phi_{i-2}(\tilde{y}_{i-3}))\right)\cdot\phi_{i-1}'(\phi_{i-2}(\tilde{y}_{i-3}))\epsilon_{\phi_{i-2}}+ \\
&\quad \phi_i'\left(\phi_{i-1}(\tilde{y}_{i-2})\right)\epsilon_{\phi_{i-1}}+\epsilon_{\phi_i} \\
&\approx \quad \vdots \quad \vdots
\end{aligned} \tag{5}$$

Carrying out similar expansion for all intermediate values, we can rewrite $\tilde{y}$ to be

$$\tilde{y} \equiv \tilde{y}_n \approx \phi_n\left(\phi_{n-1}\left(\cdots(\ \phi_2(\phi_1(x))\ )\cdots\right)\right)+\sum_{i=0}^{n}\epsilon_{\phi_i}\prod_{k=i+1}^{n}\phi_k'\left(\phi_{k-1}\left(\cdots(\phi_{i+1}(\tilde{y}_i))\cdots\right)\right) \tag{6}$$

where $\prod_{k=n+1}^{n}\phi_k'(\cdot)$ is defined to be 1.

The product shown is just the chain rule for derivative, $\frac{\partial y}{\partial \tilde{y}_i}$, which can be further approximated by the derivative without error, $\frac{\partial y}{\partial y_i}$. Note that this approximation is equivalent to the approximation already made in the first order Taylor series.

where we assume that the error product $\epsilon_w \epsilon_x$ is negligible, and a first order Taylor series approximation is used.

The input errors are propagated through the operators. For example, the multiplication of the two variables with finite precision errors propagates error, i.e., $w\epsilon_x + x\epsilon_w$. This propagated error along with the generated finite precision multiplication error, ϵ_*, further propagates through the nonlinear operator, resulting in the total finite precision error,

$$\epsilon_y = (w\epsilon_x + x\epsilon_w + \epsilon_*)\phi'(wx) + \epsilon_\phi. \tag{2}$$

The total finite precision error ϵ_y imposed on y will then become the input finite precision error of variable y for future operations.

2.1 Error Generation and Propagation by Successive Operators

A compound operator, $y = \Phi(x)$, which is produced by successive operators ϕ_1, ϕ_2, ..., ϕ_n, is shown in Figure (1). Error at the input, ϵ_x, and error generated in each operator, ϵ_{ϕ_i}, is propagated through the remaining operators to the output. We can approximate the output error, ϵ_y, in terms of ϵ_x, ϵ_{ϕ_i}, and ϕ_i [8]. From Figure (1),

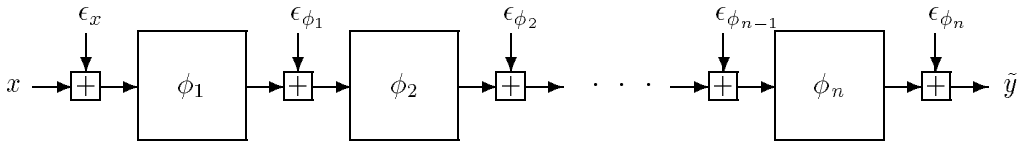

Figure 1: Successive operators generating and propagating error, where $y = \phi_n(\cdots(\phi_2(\phi_1(x)))\cdots)$.

error equations are derived in Section 3 for each of the operations required in the forward retrieving and error back-propagation steps of an MLP. Statistical analysis and simulation results of the resulting distribution of errors for each individual step of an MLP are also included in this section. These error equations are then integrated, in Section 4, to predict the influence of finite precision computation on several stages (early, middle, final stages) of back-propagation learning. Finally, concluding remarks are given in Section 5.

2 Sources of Error in Finite Precision Computation

For a finite precision computation of a nonlinear operation of multiple variables, several sources of error exist. For example, in the computation of $y = \phi(wx)$, the two input variables, w and x, have input errors ϵ_w and ϵ_x, respectively, whose sources are prior finite precision data manipulations. There are errors generated due to the finite precision computation of involved operators. More specifically, the finite precision *multiplication* of the two variables generates one error, ϵ_*. Similarly, the finite precision *nonlinear operator* ϕ generates the other error, ϵ_ϕ. Therefore, the resulting finite precision result $\tilde{y}$ is equal to

$$\begin{aligned}
\tilde{y} &\equiv \phi\left((w+\epsilon_w)(x+\epsilon_x)+\epsilon_*\right)+\epsilon_\phi \\
&= \phi\left(wx+w\epsilon_x+x\epsilon_w+\epsilon_w\epsilon_x+\epsilon_*\right)+\epsilon_\phi \\
&\approx \phi\left(wx+w\epsilon_x+x\epsilon_w+\epsilon_*\right)+\epsilon_\phi \\
&\approx \phi(wx)+(w\epsilon_x+x\epsilon_w+\epsilon_*)\phi'(wx)+\epsilon_\phi \qquad (1)
\end{aligned}$$

Finite Precision Error Analysis of Neural Network Hardware Implementations

Jordan L. Holt, Jenq-Neng Hwang

Department of Elect. Engr., FT-10

University of Washington

Seattle, WA 98195

1 Introduction

The high speed desired in the implementation of many neural network algorithms, such as back-propagation learning in a multilayer perceptron (MLP), may be attained through the use of finite precision hardware. This finite precision hardware; however, is prone to errors. A method of theoretically deriving and statistically evaluating this error is presented and could be used as a guide to the details of hardware design and algorithm implementation. The paper is devoted to the derivation of the techniques involved as well as the details of the back-propagation example. The intent is to provide a general framework by which most neural network algorithms under any set of hardware constraints may be evaluated.

Section 2 demonstrates the sources of error due to finite precision computation and their statistical properties. A general error model is also derived by which an equation for the error at the output of a general compound operator may be written. As an example,