Multi-objective Precision Optimization of Deep Neural Networks for Edge Devices

Nhut-Minh Ho, Ramesh Vaddi, Weng-Fai Wong
School of Computing, National University of Singapore — {minhhn,ramesh,wongwf}@comp.nus.edu.sg

***Abstract*—Precision tuning post-training is often needed for efficient implementation of deep neural networks especially when the inference implementation platform is resource constrained. While previous works have proposed many ad hoc strategies for this task, this paper describes a general method for allocating precision to trained deep neural networks data based on a property relating errors in a network. We demonstrate that the precision results of previous works for hardware accelerator or understanding cross layer precision requirement is subsumed by the proposed general method. It has achieved a 29% and 46% energy saving over the state-of-the-art search-based method for GoogleNet and VGG-19 respectively. Proposed precision allocation method can be used to optimize for different criteria based on hardware design constraints, allocating precision at the granularity of layers for very deep networks such as Resnet-152, which hitherto was not achievable.**

I. Introduction

There are many obvious value propositions and use cases for the implementation of deep neural networks on resource constrained platforms. Many works have considered the use of low precision, fixed-point arithmetic, or quantization methods to achieve better inference speeds and/or low energy consumption. This is the central problem of this paper: how to deploy a trained deep neural network in the most efficient manner based on precision tuning subject to user's constrains on chosen hardware. Recent works on this problem [1]–[3], focused on hardware design and relied on empirically searching that repeatedly assigns a combination of bitwidths to different layers followed by testing to try to ensure a certain quality (e.g. less than 1% dropping in classification accuracy), failing which the assignment is tweaked and retried. This approach will likely over-fit the precision result to the testing data set. Furthermore, because it is very time-consuming, this approach can only assign precision at a coarse granularity. The method presented in this paper subsumes these. [4] presented a theoretical method that finds the relationship of bitwidth allocation between the different layers of a given neural network. Unfortunately, it only works for convolutional layers in a shallow network. Other works on providing a theoretical bound on classification error of quantized network [5], [6] improved the fidelity of the bitwidth result, but assign all layer to have the same data type, which is not suitable for developing hardware accelerators or only works for weights. In short, dynamic search will give much better result with reasonable granularity but suffers a heavy computation load since it relies on running the actual network repeatedly. Theoretical bounds, on the other hand, are usually too conservative, and impractical at finer granularities. In this work, we will introduce a method that strikes a middle path that efficiently allocates bitwidth at the level of layers, and take into consideration constraints on resources. This paper makes the following contributions to the problem of precision tuning for hardware design:

- We found an important relationship between fixed point rounding errors of the different layers in deep neural networks.
- We operationalized this insight in an open source precision optimization framework that subsumes previous works, and demonstrated it effectiveness on some of the largest deep neural networks.
- Our method can be used to optimize for different criteria.

II. A motivating example

Suppose we have a simple dot product operation $y = w_1x_1 + w_2x_2$ where y is the output, w_1, w_2 are the constant weights, and x_1, x_2, are the inputs. Let δ_y, δ_{x_i} and δ_{w_i} be the numerical errors of y, x_i, and w_i, respectively.

$$\begin{aligned} &y + \delta_y = (w_1 + \delta_{w_1})(x_1 + \delta_{x_1}) + (w_2 + \delta_{w_2})(x_2 + \delta_{x_2}) \\ &\Rightarrow \delta_y = x_1\delta_{w_1} + w_1\delta_{x_1} + x_2\delta_{w_2} + w_2\delta_{x_2} + \delta_{w_1}\delta_{x_1} + \delta_{w_2}\delta_{x_2} \end{aligned} \quad (1)$$

If we assume $w_i \gg \delta_{w_i}$ and $x_i \gg \delta_{x_i}$, we have:

$$\delta_y \approx x_1\delta_{w_1} + w_1\delta_{x_1} + x_2\delta_{w_2} + w_2\delta_{x_2} \quad (2)$$

Variants of Eq. 2 were used in neural network quantization [4] and simple programs [7]. It also suggests that if we are given an output error δ_y (i.e., an output quality threshold), we can '*distribute*' it to the numerical error of the data (δ_{x_i}) and the weights (δ_{w_i}). In the above example, to get one solution for δ_{x_i} and δ_{w_i} we can simply divide δ_y into four equal portions, calculating δ_{x_i} and δ_{w_i} accordingly: $\delta_{w_i} = \delta_y/(4x_i), \delta_{x_i} = \delta_y/(4w_i), i = 1, 2$.

After computing all the numerical errors of w_i and x_i, we can select a limited precision number format that will ensure the output error of our new reduced precision will be less than or equal the given δ_y. The details for doing so are as follow:

A. Fixed point format and uniform quantization

Let's assume a fixed point format '**I.F**' where **I** and **F** is the number of bits in integer part and the fraction part of the fixed point format, respectively. The worst case rounding error when using the format **I.F** with correct rounding is $\pm 2^{-(\mathbf{F+1})}$, it is also called the *quantization error* because the fixed point format is a special case of uniform quantization when the step size is a power-of-two. After computing δ_{x_i}, we can assign

TABLE I
NOTATION AND ABBREVIATION

s.d.	standard deviation
Within a specific layer, each input location has index i	
δ_y, σ_y	the error at output node y and its s.d.
w_i	the constant learned weight at index i at a specific layer
δ_{x_i}, σ_{x_i}	the rounding errors of input at index i and its s.d.
Within a network, consider each layer as a whole	
Ł	the index of the last layer in the network (before softmax)
σ_{X_K}	s.d. of the rounding errors of all inputs X of layer K
σ_{Y_K}	s.d. of the output error at layer K
Δ_{X_K}	the boundary of uniform distribution with s.d. equals σ_{X_K}

We use capital character X, Y with index K to denote the collective input and output, respectively, of layer K in the network, and non-capital character x, y with index i, for indexing a specific element within the respective tensor.

$\lceil -\log_2(2\delta_{x_i}) \rceil$ as the **F** for x_i. For the integer part, we need to measure the range of the value of x_i to ensure x_i's integer part will not cause arithmetic overflow. In other words, **I**= $\lceil \log_2(|x_i|) \rceil + 1$ for a signed format. Furthermore, if we apply the same format **I.F** to represent each value in a set of large enough values X, after we obtained $\hat{X}$ in limited precision, the quantization error $(\hat{X} - X)$ will form an approximate uniform distribution in range $[-\Delta; \Delta]$ with $\Delta = 2^{-(\mathbf{F+1})}$. According to [8], in general, when the quantization step is much smaller than the actual value, the quantization error can be modelled as an additive white noise that is not correlated with the actual value. The white noise is a symmetric uniform distribution with mean 0, and variance : $\sigma^2 = (2\Delta)^2/12$.

Recent hardware implementation for fixed point arithmetic for neural network [1], [2] explored dropping some of the less significant bits from integer part when the value is high error-tolerant beyond integer arithmetic. In this paper, we will also consider saving the integer bitwidth when Δ is greater than 1 as it renders the fraction part and '**F**' less significant bits in '**I**' are useless. By using the appropriate scaling factor (for example, a mandatory implicit shift), it is possible to use a fixed point arithmetic with a bitwidth of **I + F** where $\mathbf{F} < 0$.

B. Fixed point convolutional neural network

Any convolutional neural network (CNN) in inference mode is simply a chain of dot product operations between large tensors of inputs and weights where the output of one layer becomes the input of the next layer, with some simpler computation components in between. At the last layer, there is usually a Softmax function that classifies an image into one of the possible classes [9]. Figure 1 shows the typical structure of a neural network comprises $1, \ldots,$ Ł layers of computation, other components (pooling, ReLU) are omitted for simplicity. The error behavior is simple when weights are kept exact (floating point): the rounding error of the inputs to layer K is uniformly distributed with a mean ≈ 0 and is symmetric. The error at its output will be approximately Gaussian $\sim \mathcal{N}(0, \sigma^2_{Y_K})$. This error will propagate through the dot product, intermediate layers ending at the output layer which will also approximately a normal distribution $\sim \mathcal{N}(0, \sigma^2_{Y_Ł})$. This insight led us to focus on the relationships between the

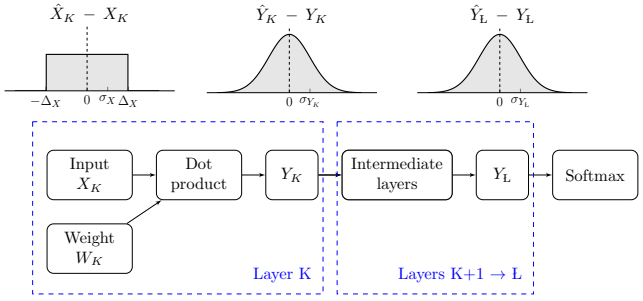


Fig. 1. The typical error propagation behavior of fixed point neural network when using fixed point format on input at layer K and keeping other components exact. Zero values at X_K are always accurately represented in fixed point and hence not included. Error histograms were drawn with smooth lines for simplicity.

standard deviations (s.d.) of these error distributions, and use these to allocate bitwidth for each layer of a neural network.

III. STATISTICAL PROPERTIES OF ROUNDING ERRORS IN NEURAL NETWORKS

In this section, we describe a simple model for rounding error propagation in real neural networks that have a mix of convolution, fully connected, pooling and ReLU layers, running in its inference mode. Convolution and fully connected layers use the same dot product operation, the only difference is the way inputs or weights are shared.

A. Single layer error model

Let's start with a single layer. The model described in Section II is applicable for simple fixed inputs. Let's assume that the weights are perfect, i.e., $\delta_{w_i} = 0$ in Eq. 3. We can derive the quantization error model from Eq. 2 for S different input images with N pixels each as follows:

$$\delta_y \approx \sum_{i=1}^{N} w_i \delta_{x_i} \tag{3}$$

Since we are dealing with inference, w_i in Eq. 3 are constant as they are learned weights. For a single input image, we can get a discrete value for each δ_{x_i}. To generalize over all input images, we will consider each δ_{x_i} to be a random variable drawn from a distribution of rounding error on x_i with a sample size S, the number of input images. Specifically, as described in II-A, it will be a a uniform distribution with a mean of 0 and s.d. equals σ_{x_i}. From Eq. 3, if we assume that the δ_{x_i}'s are mutual independent, and have the same s.d. of σ_{X_K} (this assumes that we will use the same fixed point format for all x_i in layer K), the relationship between the variances is:

$$\sigma_y^2 \approx \sum_{i=1}^{N} w_i^2 \sigma_{X_K}^2 \Rightarrow \sigma_{X_K} \approx \lambda \sigma_y \tag{4}$$

B. Correlated input errors and grouping error at output

The relationship in Eq. 4 could be sufficient to model the propagation of error in a single dot product operation that produces one output element. Because the output of each layer is an array with M elements, there will be a small variation in

the s.d. of each $\sigma_{y_j}, j \in [1, M]$ as they will be computed on different set of weights. Thus, if we measure the s.d. of errors over the whole tensor output at layer K as σ_{Y_K}, it will be a mixture of many normal distributions [10]. Furthermore, as the output of one layer is the input of the next layer with some degree of input sharing in the computation, the correlation between rounding errors on different input locations needs to be included when we consider a dot product performed on an intermediate layer. After considering possible correlations when grouping all the outputs of the same layer into a single error distribution, we found that the relationship needs to be adjusted by one more (additive) constant θ giving us $\sigma_{X_K} \approx \lambda\sigma_{Y_K} + \theta$. This will be justified in Section IV.

C. Nonlinear activation and Pooling layers

In the previous sections, we modeled the change in s.d. of input passing through the dot product at a single layer. Likewise, we can model the change in the s.d. of rounding error passing through other types of layer as well. The popular nonlinear activation function ReLU(x) = $\max\{0, x\}$ does not break the linear relationship between s.d. of value passing through it. Consider the output $y = \text{ReLU}(x)$. Since the rounding error of exact 0 values when using fixed-point is 0, regardless of the precision we choose, having more zero values after ReLU will scale down the s.d. while keeping the mean at 0. Thus, $\sigma_y = \alpha \cdot \sigma_x$, with some constant α.

For pooling layers, max pooling does not affect the output rounding error, i.e. if $y = \text{max\_pool}(x)$ then $\sigma_y = \sigma_x$, since the error of y is a sub-sample of the error in x. For average pooling with filter size N, we can consider it as a dot product operation in Eq. 3 with constant weights of $1/N$.

IV. Cross-layer linear relationship between standard deviations

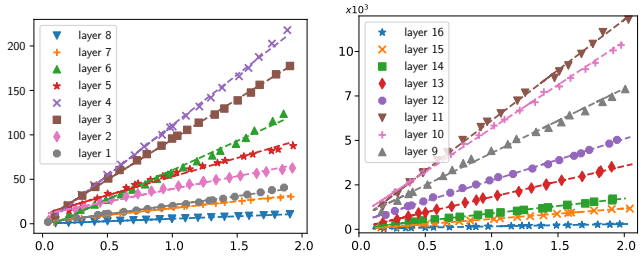
From the single layer model above, we shall now extend the model to the entire network. In particular, if we round all the inputs of layer K with some fixed point format that has s.d. of rounding error equals σ_{X_K} and do a forward pass to the last layer Ł, $\sigma_{Y_Ł}$ will still be linearly related to σ_{X_K}. As layer K is the origin of this error, we use the symbols $\delta_{Y_{K\to Ł}}$ and $\sigma_{Y_{K\to Ł}}$ to denote the error at layer Ł caused by layer K and its s.d., respectively. From Section II-A, we see that the boundary of the uniform distribution $\Delta_{X_K} = \sigma_{X_K}\sqrt{12}/2$. We postulate that there is a linear relationship between Δ_{X_K} of layer K, and $\sigma_{Y_{K\to Ł}}$ of the last layer that is as follows:

$$\Delta_{X_K} \approx \lambda_K \sigma_{Y_{K\to Ł}} + \theta_K \tag{5}$$

with λ_K and θ_K are measurable constants of layer K.

We validated Eq 5 by conducting measurements for all deep networks that we have access to. For each layer K in each neural network, we inject a random uniform error of $[-\Delta_{X_K}, \Delta_{X_K}]$ to X_K. Then we measured $\sigma_{Y_{K\to Ł}}$at the last layer after the inference. Figure 2 shows the linear relationship in Eq. 5 measured on different CNNs.

In all the CNNs we checked, including some that have complex modern structures and other types of computation



(a) AlexNet (b) VGG-19

Fig. 2. Relationship between Δ_{X_K} (y-axis) and $\sigma_{Y_{K\to Ł}}$ (x-axis) in two CNNs. Each line is a linear regression model connects all measurements of Δ_{X_K} and $\sigma_{Y_{K\to Ł}}$ for layer K. Half of the layers of VGG-19 were removed to avoid overlapping. Both were measured on 500 images.

beyond what we have modelled in Section III, Eq 5 was confirmed. In particular, it was able to predict the Δ_{X_K}'s mostly with a $< 5\%$ error by using the measured $\sigma_{Y_{K\to Ł}}$. In the worst case, it was about 10% of the actual value. As will be shown in our experiments, this linear regression derives safe fixed point bitwidth for a variety of CNNs.

V. Analyzing the whole neural network

There are three more pieces needed to complete our framework. To use Eq. 5, we first need to find the coefficients of the relationship. Next, we consider the effect when propagating all the rounding error of each layer towards the final layer, Ł's, numerical error. Finally, we need to find the numerical error at layer Ł that will satisfy the user's accuracy requirement.

A. Error injection and measurement

This section presents a linear regression based method to measure the two constants λ_K and θ_K in Eq. 5 for each layer, K in a CNN. Given a neural network with $1, \ldots,$ Ł layers to be analyzed (convolutional and fully connected layers), we need to run the forward pass of the CNN over a "large enough" dataset Ł times, each time doing the following:

1) Record down the exact value of layer Ł's output ($Y_Ł$).
2) Guess an initial value of Δ_{X_K}.
3) Inject an error from the uniform distribution $[-\Delta_{X_K}, \Delta_{X_K}]$ into input of layer K, and do a forward pass to layer output Ł to obtain $\hat{Y}_Ł$.
4) Measure the s.d. of $(\hat{Y}_Ł - Y_Ł)$ as $\sigma_{Y_{K\to Ł}}$.
5) Change the value of Δ_{X_K} and loop over (3-5) for a small number of times (we found 20 to be sufficient) to get enough points for a linear regression.

We found in our experiment that measurements from 50-200 images will produce stable regression results.

B. Analyzing the errors in the final layer

To see how errors propagate to the final layer, let's consider a CNN with Ł layers again. For each layer $K \in [1 \ldots Ł]$, we use a fixed point format that has uniformly distributed errors in the range $[-\Delta_{X_K}, \Delta_{X_K}]$ on input X_K. That error will propagate through the computation in neural network and result in a change in the output of layer Ł. We shall have the propagated error at the final layer that originates in layer

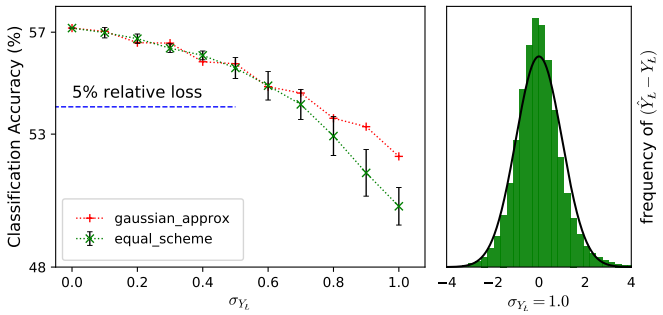


Fig. 3. The left shows the relationship between $\sigma_{Y_Ł}$ and accuracy under two different error injection modes for AlexNet. Each point is the average of 3 measurements, the variation is insignificant compared to the black error bars. In all the tests for *equal_scheme*, we also check the approximation of Eq. 7, the error is less than 5% of the target $\sigma_{Y_Ł}$ values. The green histogram on the right shows the actual error at layer Ł (green) against a perfect $\mathcal{N}(0,1)$ (black). This histogram has a s.d. = 0.99, mean = 7×10^{-5} measured on 5×10^5 output values.

K as $\delta_{Y_{K\to Ł}}$ with s.d. $\sigma_{Y_{K\to Ł}}$ as in Eq. 5. The output error at layer Ł, i.e., $\delta_{Y_Ł}$ which has s.d. $\sigma_{Y_Ł}$, consists of Ł error sources: $\delta_{Y_Ł} = \sum_{K=1}^{Ł} \delta_{Y_{K\to Ł}}$. According to [11], these noise sources can be assumed to be mutually independent when quantizing CNNs. Therefore we have this relationship between the variances from mutual independent sources $\delta_{Y_{K\to Ł}}$:

$$\sigma^2_{Y_Ł} = \sum_{K=1}^{Ł} \xi_K \cdot \sigma^2_{Y_Ł} \approx \sum_{K=1}^{Ł} \sigma^2_{Y_{K\to Ł}} \text{ where } \sum_{K=1}^{Ł} \xi_K = 1 \quad (6)$$

From Eq. 6, we have $\sigma^2_{Y_{K\to Ł}} = \xi_K \cdot \sigma^2_{Y_Ł}$. Putting $\sigma_{Y_{K\to Ł}}$ into Eq. 5, we can find a value of Δ_{X_K} that reflects the decomposition in Eq. 6. Hence for each layer, we have:

$$\Delta_{X_K} \approx \lambda_K \sigma_{Y_{K\to Ł}} + \theta_K = \lambda_K(\sigma_{Y_Ł}\sqrt{\xi_K}) + \theta_K \quad (7)$$

This is the main result of our paper: given a target $\sigma_{Y_Ł}$, and having found λ_k and θ_K, Eq. 7 gives us a way to distribute the error 'backward' to any layer K via ξ_K to yield Δ_{X_K}, the boundary of uniform distribution used to model the error of X_K. This is in turn used to select a bitwidth for that layer's input (X_K). We can then use different criteria to decide on ξ_K. This will be discussed further in Section V-D.

C. Relating classification accuracy to numerical error

The final piece to our framework is to relate the numerical error at the last layer, $\sigma_{Y_Ł}$, to the final classification accuracy. The error at layer Ł has s.d.'s around $\sigma_{Y_Ł}$ when using different values for each ξ_K according to Eq. 7. However, we observed that different ways of assigning ξ_K may affect classification accuracy. Fortunately, for all our experiments, we found that the differences were within the user accuracy criteria when using our optimization scheme. To further validate this, we estimated the worst possible variation to accuracy when changing the value of each ξ_K in range $[0.1/Ł, 0.8]$ which covers most of the possible results for bitwidth optimization. One can also impose this as a constraint for optimization. We define a baseline *equal_scheme* of error distribution as $\forall \xi_K \in \xi, \xi_K = 1/Ł$. The results of the study on AlexNet is shown in Figure 3. The maximum change in accuracy compared to the *equal_scheme* (in green series) is plotted as the black error bar at each point. Because the number of combinations is large, we tested the corner cases, which happens when $\xi_K = 0.8$ with the rest sharing the remaining 0.2 portion of the final error equally. For example, the first case for 3 layers would be $\xi = (0.8, 0.1, 0.1)$. As can be seen from Figure 3, the variation is tolerable when the accuracy loss is below 5%. We support the *equal_scheme* as 'Scheme 1' to estimate the accuracy degradation in our framework. We also tested a fast approximation that injects $\mathcal{N}(0, \sigma^2_{Y_Ł})$ to only layer Ł, labeled as the *gaussian_approx* series in Figure 3. This is possible because the typical error on output layer Ł almost normally distributed as shown in the histogram on the right of Figure 3. This is supported as 'Scheme 2' in searching.

In summary, distributing the errors, for a given $\sigma_{Y_Ł}$, differently amongst the layers may affect the final classification accuracy. However, as long as the user given constraint for relative accuracy loss is not too large (say less than 5%), any (realistic) error distribution will be able to satisfy it.

Because $\sigma_{Y_Ł}$ monotonically increases when accuracy decreases, we use a binary search on real numbers similar to [12] to search for the $\sigma_{Y_Ł}$ that will satisfy a certain accuracy level. We start by simply guessing an upper bound for $\sigma_{Y_Ł}$ (say, 1.0). If it violates the given constraint then we start the binary search procedure. Otherwise, we double the upper bound's value, and try until we find an upper bound that violates the accuracy constraints. The actual search is a standard binary search on real number, we stop when the distance between the two bounds is less than 0.01. The test for whether a value of $\sigma_{Y_Ł}$ is satisfied differs between the two aforementioned schemes 1 and 2. In Scheme 1, we use Eq. 7 to get all the values of Δ_{X_K} using $\sigma_{Y_Ł}$ and $\xi_K = 1/Ł$. We then inject uniform noise $[-\Delta_{X_K}, \Delta_{X_K}]$ to each layer K and measure the accuracy after doing a forward pass. For Scheme 2, we inject error only to $Y_Ł$ with Gaussian noise $\mathcal{N}(0, \sigma^2_{Y_Ł})$ and measure the accuracy. Each test runs on at least half of the test dataset used in the target CNN. After this step, we will get the amount of $\sigma_{Y_Ł}$ that ensures the classification accuracy will not worse than the user supplied threshold.

D. Multi-objective bitwidth optimization

Having found the value of $\sigma_{Y_Ł}$ for the entire CNN as well as λ_K and θ_K for each layer, we can now use Eq. 7 to quickly get the bitwidth for each layer. The robustness of Eq. 7 allows us to optimize bitwidth for different objectives, resulting in different ξ's. To demonstrate the multi-objective optimization problem, we use AlexNet as an example, using the same model and pretrained weights as in the baseline [1], [3] with a targeted 1% relative accuracy drop. The integer bitwidth are measured by doing a forward pass through all the layers, recording down the maximum absolute value of the input values. The signed integer bitwidth is derived using row $\max|X_K|$ of Table II, and they are $(9, 9, 9, 10, 10)$.

Let's first consider the objective is to minimize the total bandwidth used for reading the input data. From the baseline result in *Baseline* row of Table II, we can calculate the total

TABLE II
ALEXNET EXAMPLE FOR OPTIMIZING BITWIDTH ON TWO DIFFERENT OBJECTIVES WITH 1% ACCURACY LOSS

Layer	conv1	conv2	conv3	conv4	conv5	Total
#Input($\times 10^3$)	154.6	70	43.2	64.9	64.9	397.6
#MAC($\times 10^8$)	1.05	2.25	1.5	1.12	0.75	6.66
$\max\|X_K\|$	161	139	139	443	415	-
Baseline [1]	9	7	4	5	7	-
#Input_bits($\times 10^3$)	1391.3	489.9	173.1	324.5	454.3	2833
#MAC_bits($\times 10^8$)	9.49	15.68	5.98	5.61	5.23	41.98
Opt_for_#Input	6	6	5	6	7	-
#Input_bits($\times 10^3$)	927.5	419.9	216.3	389.3	454.3	2407
Opt_for_#MAC	7	5	5	6	7	-
#MAC_bits($\times 10^8$)	7.38	11.2	7.48	6.73	5.23	38.01

number of bits needed to read input data for each layer by multiplying the bitwidths with the respective number of input elements from the *#Input* row. The results is given in the *Input_bits* row. Let ρ_K be the coefficient that gives the relative importance of each layer K in the objective. These coefficients are exactly the *#Input* row in Table II. The more input elements there is in a layer, the more it contributes towards total bandwidth. We can now formulate an optimization problem where we minimize the total number of input bits based on the relationship between Δ_{X_K} and bitwidth from Section II-A. With $\Delta_{X_K} = \lambda_K \sigma_{Y_L} \sqrt{\xi_K} + \theta_K$ from Eq. 7, the objective is:

$$\min \ \mathbf{F} = \sum_{K}^{5} \rho_K(-\log_2(\Delta_{X_K})), \quad \text{s.t.} \ \sum_{k}^{5} \xi_K = 1 \qquad (8)$$

Using the search technique described in Section V-C, we found $\sigma_{Y_L} \approx 0.32$. λ_K and θ_K were then measured as described in Section V-A. We put the above optimization problem to Octave's `sqp` function, which yielded a solution $\xi = (0.32, 0.13, 0.14, 0.23, 0.18)$. We then computed Δ_{X_K} from Eq. 7. The results were then translated to the fraction bitwidth. These were then combined with the integer bitwidths found as discussed in Section II-A. Doing so yielded the total bitwidths that were enough to represent the input for each layer. These are given as in *Opt_for_#Input* row. We then recompute the objective values, which are the *Input_bits* values for each layer using the computed bitwidths. We can observe that the optimization will increase the bitwidth of layers *conv3* and *conv4* (which have fewer input elements) in order to decrease the bitwidth of layer *conv1* and *conv2* (which have more input elements). It results in the total 2.407×10^6 bits used for all input elements, a 15% saving over the baseline.

The same procedure can be applied for optimizing bitwidth of MAC operations (hence energy) by assigning ρ_K to be the *#MAC* row which contains the number of MAC operations of each layer. All other parameters remained the same. The resultant bitwidths are shown in the *Opt_for_#MAC* row. Doing the same calculation for the total number of bits spent on inputs in all the MAC units showed a 9.5% saving in total. Both sets of optimized bitwidths yielded $< 1\%$ error when tested against the 25,000 images in the ImageNet test set.

In Section VI we will do a more extensive experiment of these same two objectives using eight CNNs, some of which are the largest available now. Because the number of layers is large, we will normalize the total number of bits in the *Total* column by dividing them by the total number of input elements and MAC operations in row *#Input* and *#MAC*, respectively. In the above example, *effective_bitwidth* on each input element of the baseline is $2833/397.6 \approx 7.1$ and of the optimized bitwidth for input is $2407/397.6 \approx 6.05$. In short, if the bitwidth of each layer K is B_K, the *effective_bitwidth* is $\sum(\rho_K \cdot B_K)/\sum \rho_K$.

E. Searching for weight bitwidth

The extended version of Stripes [1], Loom [2] searches for weight bitwidth after the reduction in input bitwidth has been made. We integrated the same method at the end of the input optimization process and report the results in Section VI. Our bitwidth optimization method can also work well with other weights quantization techniques [11], [13] as a complementary and separate workflow.

VI. EXPERIMENTAL EVALUATION

We compared our results using the same representation format as in Stripes [1] and Loom [2]. Our weight bitwidths is similar to theirs. Thus, we compare the possible saving in terms of input bitwidths only, the weight bitwidths are shown in columns *W* for reference, and used in the computation of energy consumption. Because of the variation in accuracy when training a neural work, we used the four deepest networks in their experiments with pretrained weight available online, and four even deeper and more recent CNNs. The number of layers for each is shown in column *# layers* of Table III. Stripes ignored the fully connected layers, so we did the same for AlexNet, NiN, GoogleNet and VGG-19. The trained models are from the Caffe Model Zoo [14] and [15]. Due to space limitation, we can only report the *effective_bitwidth* optimized for two criteria as in Section V-D, which are *Optimized Input* and *Optimized MAC*. For each optimized bitwidth and the baseline, the *effective_bitwidth* is computed for both criteria (show in *Input* and *MAC* columns) to show the difference between two objectives. The result is reported in Table III for 1% and 5% relative top-1 accuracy loss. All optimized bitwidths were tested against 25,000 images of the ImageNet test dataset. No accuracy criterion was violated[1].

The performance gain for Stripes' MAC unit can be derived directly from the table because their performance scales almost linearly with the saving in *effective_bitwidth* (the *MAC* columns). The bandwidth gain is calculated directly from the *effective_bitwidth* for input elements (*Input* columns) and reported in column *BW Saving*. The objective *Optimized MAC* also implies minimizing energy consumption on all MAC units. To further estimate the possible saving in energy consumption, a standard MAC design available in the Synopsys DesignWare IP (DWIP) library is used for simulations with TSMC 40-nm Low-Power (LP) LVT technology at 0.9 VDD and 500 MHz frequency. The MAC has been synthesized with

[1]Stripes' bitwidth resulted in a 3.5% loss for NiN. Hence, we used this same accuracy target for a fairer comparison with their NiN experiment.

TABLE III
RESULT ON VARIOUS DEEP NEURAL NETWORKS OF OPTIMIZING FOR BANDWIDTH (BW) AND MAC ENERGY USING OUR PROPOSED METHOD

	# layers	1% relative accuracy drop										5% relative accuracy drop								
		W	Baseline		Optimized Input			Optimized MAC			W	Baseline	Optimized Input			Optimized Mac		
			Input	MAC	Input	MAC	BW save (%)	Input	MAC	Ener save (%)			Input	MAC	BW save (%)	Input	MAC	Ener save (%)
AlexNet	5	10	7.10	6.30	6.05	5.89	14.7	6.27	5.70	12.5	8	5	4.55	4.27	9	4.83	4.20	22
NiN	12	10	7.79	8.35	7.66	7.58	1.6	8.22	6.94	22.8	8	8	7.33	7.10	8.4	7.93	6.72	23.9
GoogleNet	57	10	8.66	8.58	6.96	7.17	19.7	7.22	6.79	29.3	8	6	5.97	6.23	0.5	6.26	5.83	5.2
VGG-19	16	11	9.93	10.90	7.20	7.64	27.5	7.29	7.27	45.8	9	7	6.37	6.93	9	6.62	6.84	0.9
ResNet-50	54	9	9	9	7.78	7.96	13.6	8.07	7.57	20.2	8	8	6.90	7.20	13.8	7.35	6.77	22.4
ResNet-152	156	11	12	12	9.70	9.37	19.2	9.90	9.27	32.1	8	11	8.90	8.86	19.1	9.37	8.76	26.8
SqueezeNet	26	8	9	9	9.24	8.77	-2.7	9.60	8.03	11.6	7	9	8.24	7.77	8.4	9.17	7.13	27.9
MobileNet	28	10	10	10	9.55	9.48	4.5	10.38	8.93	16.3	9	9	8.77	8.58	2.6	9.70	7.99	13.2
Average	-	-	-	-	-	-	12.3	-	-	23.8	-	-	-	-	8.8	-	-	17.8

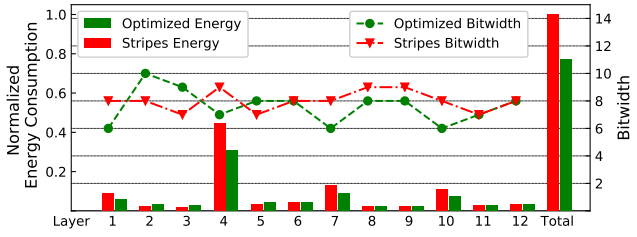


Fig. 4. NiN example with 12 layers, by sacrificing bitwidth of less power-hungry layers (2,3,5), the bitwidth of more power-hungry layers (1,4,7,10) can be reduced. It results in a saving of 22.8% in the total energy comsumed by all MAC units compared to the baseline.

commercially available cells in a standard digital flow and used conservative wire models for synthesis and power estimations. The total energy consumed by all MAC operations in each CNN to process one image is shown in the *Ener Save* column. The baseline bitwidths are taken directly from Stripes where available. Otherwise, we used the smallest possible uniform bitwidth for all layers as the baseline.

A. Discussion

Our method transformed the time-consuming searching method in previous works into two simpler tasks: (1) profiling λ_K and θ_K which takes a few minutes and (2) binary search for σ_{Y_L}. After profiling, the user can put their constraints for bitwidth optimization to our tool. It costs only 5 minutes for optimization and less than 1 hour for binary search on the deepest Resnet-152 using 1 Nvidia P100 GPU. Changing the user constraints only requires re-running the last optimization step. Optimizing for bandwidth and MAC energy will yield different solutions for bitwidths. For example, in Figure 4, optimizing for energy will yield a bandwidth that is 5.6% worse than the baseline. However, there will be a 22.8% saving in MAC energy. It is conceivable that designers can formulate different optimization criteria using our framework.

VII. CONCLUSION

The key insight of this paper is that the propagation of rounding errors between layers of a learned neural networks is governed by constants, λ_K and θ_K, that can be measured. This allows us to compute the layer-level precision allocation for very deep networks of over 150 layers. While we do not expect edge devices to implement such huge networks, our experiments confirmed that our method is robust. More importantly, we have showed how our technique can be used to optimize bitwidths in deep neural network according to different resource constraints if the final classification accuracy is slightly relaxed. Our technique outperform the state-of-the-art in both the quality of the results and compute time. The tool has been integrated into Caffe [16]. We believe it will aid in the deployment of efficient deep neural network accelerators.

ACKNOWLEDGEMENT

This work was supported in part by Singapore's Ministry of Education research grant MOE2016-T2-1-091.

REFERENCES

[1] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *49th MICRO*. IEEE, 2016, pp. 1–12.
[2] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *55th DAC*. ACM, 2018, p. 20.
[3] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-precision strategies for bounded memory in deep neural nets," *arXiv preprint arXiv:1511.05236*, 2015.
[4] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *ICML*, 2016, pp. 2849–2858.
[5] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *ICML*, 2017, pp. 3007–3016.
[6] Z. Song, Z. Liu, C. Wang, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," *arXiv preprint arXiv:1709.07776*, 2017.
[7] A. A. Gaffar, O. Mencer, W. Luk, P. Y. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation." in *FPT*, vol. 2, 2002, pp. 158–165.
[8] B. Widrow, I. Kollar, and M.-C. Liu, "Statistical theory of quantization," *IEEE Trans. Instrum. Meas.*, vol. 45, no. 2, pp. 353–361, 1996.
[9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.
[10] J. Behboodian, "On a mixture of normal distributions," *Jstor*, 1970.
[11] Y. Zhou, S. M. Moosavi Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive quantization for deep neural network," in *AAAI*, 2018.
[12] L. F. Williams Jr, "A modification to the half-interval search (binary search) method," in *Proc. 14th annual Southeast regional conference*. ACM, 1976, pp. 95–101.
[13] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *IEEE CVPR*, 2017.
[14] Y. Jia and E. Shelhamer, "Caffe Model Zoo," 2018, https://github.com/BVLC/caffe/wiki/Model-Zoo.
[15] "Mobilenet pretrained models," 2018, https://github.com/shicai/MobileNet-Caffe.
[16] "Github link - Mupod," https://github.com/minhhn2910/mupod.