Quantized Deep Neural Networks for Energy Efficient Hardware-based Inference

Ruizhou Ding, Zeye Liu, R. D. (Shawn) Blanton, Diana Marculescu

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, U.S.A. 15213
E-mail: {rding, zeyel, rblanton, dianam}@andrew.cmu.edu

Abstract – Deep Neural Networks (DNNs) have been adopted in many systems because of their higher classification accuracy, with custom hardware implementations great candidates for high-speed, accurate inference. While progress in achieving large scale, highly accurate DNNs has been made, significant energy and area are required due to massive memory accesses and computations. Such demands pose a challenge to any DNN implementation, yet it is more natural to handle in a custom hardware platform. To alleviate the increased demand in storage and energy, *quantized DNNs* constrain their weights (and activations) from floating-point numbers to only a few discrete levels. Therefore, storage is reduced, thereby leading to less memory accesses. In this paper, we provide an overview of different types of quantized DNNs, as well as the training approaches for them. Among the various quantized DNNs, our LightNN (Light Neural Network) approach can reduce both memory accesses and computation energy, by filling the gap between classic, full-precision and binarized DNNs. We provide a detailed comparison between LightNNs, conventional DNNs and Binarized Neural Networks (BNNs), with MNIST and CIFAR-10 datasets. In contrast to other quantized DNNs that trade-off significant amounts of accuracy for lower memory requirements, LightNNs can significantly reduce storage, energy and area while still maintaining a test error similar to a large DNN configuration. Thus, LightNNs provide more options for hardware designers to trade-off accuracy and energy.

I Introduction

Deep neural networks (DNNs) have been widely adopted in various real-time classification applications. Speech recognition applications use recurrent DNNs to extract information from the speech data [29]; facial recognition via DNNs can be used for the security of personal properties [30] [31]; driver-assistance systems can detect passengers, vehicles and barriers using DNNs [32]. In these applications, DNNs have been shown to be very effective due to their non-linear characteristics, flexible configurations and self-adaptive features [1]. In addition, DNNs extract features layer by layer, and therefore can learn the pattern of input data from lower-level features to higher-level and more abstract features. For an image, DNNs can extract the dots and edges in the first few convolutional layers, and then generalize basic shapes and textures in the next layers. In the last few layers, the network combines these extracted features with fully-connected layers or a global-pooling layer, and produces the final classification result. However, this characteristic of DNNs indicates an essential requirement, that is, a large number of layers to guarantee sufficient accuracy. For example, Google's AlphaGo adopts a 13-layer architecture, with hundreds of filters per layer [6]. As another example, Microsoft implements a 152-layer DNN for image classification [7].

The increasing number of neurons and connections in DNNs brings pressure to the implementation of DNNs in real-time classification applications. Potential implementation platforms include CPU, GPU, FPGA and custom hardware (*e.g.*, ASIC). Recent research has focused on DNNs implemented directly in custom hardware for a variety of reasons stemming from design requirements or application characteristics. First, real-time classification applications (such as Siri and Google glass [2]) have great sensitivity to latency, thereby making custom hardware implementations better candidates than conventional architectures based on CPUs or GPUs. Second, neural networks implemented in custom ASICs require mostly logic with little complicated control, lending themselves to lower design effort. Third, heterogeneous architectures as a whole appear to be more suitable for DNN implementation due to the combined benefits of CPU, GPU, FPGA and ASIC-based hardware acceleration [3] [4]. In heterogeneous systems, ASICs can handle specific tasks that are required frequently, such as classification in real-time image recognition, while CPUs and GPUs can perform online training. In this paper, we also consider the scenario where training is accomplished in software, and inference is performed in hardware to ensure high-speed for real-time applications. However, due to the significant size of hardware DNNs which incurs significant energy and power, limits their wide adoption.

The energy consumption of DNN hardware implementations mainly comes from two parts – memory accesses and computation. A quantized DNN is poised to become a sound solution for reducing memory accesses and computation energy. Indeed, quantized DNNs constrain the weights (and activations) into a set of discretized values. Therefore, instead of using 32-bit floating point numbers which include $2^{32} \approx 4$ billion values, one can use much fewer discrete values. Therefore, significantly fewer bits are required for each weight, thereby reducing the overall memory accesses. For example, the LightNN-1 [8] introduced in section V-A only needs four bits for each weight. Furthermore, some prior art also quantizes the activation values, *i.e.*, the output of each convolutional and fully-connected layer. For example, BinaryNet [15] constrains both the weights and activations to be either $+1$ or -1, which can be represented using a single bit.

In addition to memory accesses, quantized DNNs can also reduce computation energy consumption. BinaryConnect [19] and BinaryNet [15] replace multipliers with an XNOR operator. Incremental network quantization [33] and LightNN-1 [8] use a shift operator to substitute the multiplier. LightNN-2 [8] converts the multiplication to two shifts and one addition. These new operators can reduce the energy consumption caused by the large number of multiplication operations in DNNs.

978-1-5090-0602-1/18/$31.00 ©2018 IEEE

Even for quantization approaches that still use multiplications, such as the fixed-point quantization introduced in section II-B, relying on fixed-point multiplication and addition can also reduce energy consumption compared to the floating point operations.

The rest of this paper is organized as follows. In Section II, we give a general view of various DNN quantization approaches. Section III introduces various approaches for training quantized DNNs. We introduce LightNN as well as its training scheme in Section IV. Experiment results are presented in Section V, including accuracy, storage, energy and area comparisons for conventional DNNs, Binarized Neural Networks (BNNs) and LightNNs. Section VI concludes the paper.

II. DNN Quantization

In this section, we first introduce full-precision DNNs and then go through different types of DNN quantization methods proposed in recent years.

Usually a Neural Network (NN) with more than four layers is denoted as a DNN. Without loss of generality, a 4-layer fully-connected DNN model is presented in Figure 1. Suppose the input data has four features, each of which goes to one input neuron. Then, in each hidden layer and the output layer, a linear combination of the previous neurons is computed, and an activation function is applied to the result, respectively. The activation function (examples include sigmoid or ReLU [18]) applies a non-linear transformation to the output value. In Convolutional Neural Networks (CNNs), the hidden layers include convolutional layers, which compute a convolution between their input feature maps and the weights kernels. We consider online training for DNNs in software, with the hardware implementation used for deployment in inference. In the training phase with a back-propagation algorithm [18], the loss function, such as the l_2*-norm* or *cross-entropy loss*, is computed using output values and data labels. Then, the loss function and intermediates results are used to update the weights used for linear combination described above. In the *deployment (testing or inference) phase* of a classification task, the output neuron with the largest value indicates the prediction result.

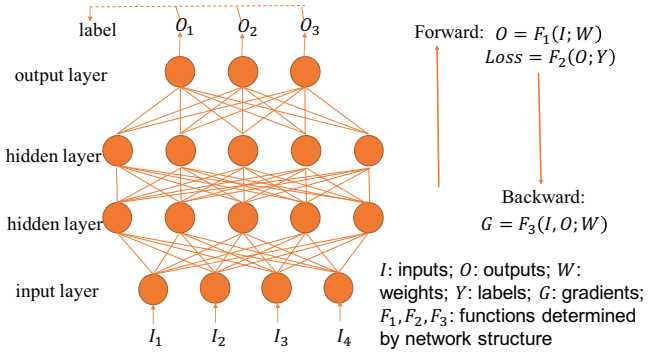


Figure 1. Model of a 4-layer DNN.

During deployment, the vast majority of computation resources are used for the multiplication within DNNs [3]. Therefore, by quantizing the weights (and activations), one can reduce the energy consumption of the multiplication operations, and thereby significantly reduce the overall computation energy. In the next sub-sections, we describe and compare several quantization approaches.

A. Binarized Neural Networks

Two types of *binarized neural networks* (BNNs) have been proposed by Courbariaux *et al*. BinaryConnect [19], a type of BNN, constrains the weights to $+1$ or -1, but the inputs and intermediate results remain as floating point values. On the other hand, a second BNN known as BinaryNet [15], constrains both weights and intermediate results (activations) to $+1$ or -1, and only the input values are represented as floating point.

During the inference phase, BinaryNet is different from BinaryConnect only in that it uses a binarized activation function, thereby having binarized intermediate results. In the inference phase, a sign function $f(x) = sign(x)$ is used as the activation function. In the training phase, *hard tanh* function, defined as $Htanh(x) = clip(x, -1, 1)$, is used as a substitute for the *sign* function [15]. The benefit of BinaryNet is that it further regularizes the DNN, and replaces each multiplication with a simple 1-bit XNOR operation.

However, among the experiments that demonstrate that BNNs produce an accuracy comparable to full-precision DNNs, a majority involve very larger network configurations, a result repeated in section V-B. This means that BNNs need more parameters to maintain accuracy for smaller networks, thereby diminishing their projected benefits in storage/computation reduction.

B. Fixed-point Quantization

Fixed-point quantization linearly discretizes the weights into multiple levels. Therefore, BNNs can be considered as fixed-point quantization using one bit. More generally, Gupta *et al*. replace the floating-point weights and activations with a fixed-point representation [QI, QF], where QI and QF are the integer and fractional part, respectively [11]. They constrained the number of bits for QI and QF into IL and FL, where IL and FL are both integers. Therefore, the weights and activations can have 2^{IL+FL} value levels, which are linearly aligned in the range $[-2^{IL-1}, -2^{IL-1} - 2^{-FL}]$. In addition, in the training phase, the gradients of the weights are constrained to fixed-point values. Their experiments using the MNIST and CIFAR-10 datasets show that the test error of fixed-point DNNs with $IL + FL = 16$ is close to floating-point implementations, as long as FL is sufficiently large and stochastic rounding is adopted.

To explore the methodology of fixed-point quantization, Lin *et al*. proposed a framework to convert a pre-trained full-precision DNN into a fixed-point DNN based on signal-to-quantization-noise-ratio (SQNR) [34]. This framework includes three steps. First, a large set of typical inputs is provided as input to the DNN, and the activations per layer is recorded. Then, the statistics of weights, biases, and activations per layer are collected. Observing weights and activations approximately follow a Gaussian distribution, they focus on the mean and standard deviation. Finally, they used these statistics to determine each layer's FL that can maximize SQNR.

Extensive experiments for fixed-point quantization are conducted by Zhou *et al*. [35]. They tried one-, two-, three-, four-, eight- and 32-bit fixed point quantization for the weights, activations, and gradients on Street View House Number

(SVHN) [9] and ImageNet datasets [5]. The resulting accuracy demonstrates a clear reduction with decrease in bit-width, especially the bit-width of activations and gradients.

C. Adaptive Quantization

Different from fixed-point quantization, *adaptive quantization* does not have to use pre-set fixed-point values. Instead, it selects the quantized values based on the weight/activation distribution. Furthermore, adaptive quantization allows the quantized levels to be aligned randomly, while fixed-point quantization relies on equidistant quantized levels.

Zhu *et al*. trained DNNs with ternary quantization [37], where the weights are quantized into three levels, $-W^n$, 0 and W^p. The quantization procedure repeats four steps. First, the full-precision weights are divided into three subsets based on their values. Next, the three subsets are assigned values -1, 0 and 1 from the smallest to the largest. Then, these three levels are multiplied by W^n, 0 and W^p, respectively. Subsequently, the DNN's loss function is computed and used to compute gradients for the parameters. Note that W^n and W^p are not pre-set values, but are two parameters that are searched/trained.

Zhou *et al*. proposed Incremental Network Quantization (INQ) [33], which quantized the weights to be powers of two. Therefore, the hardware implementation for the multiplier is equivalent to a shift operator. This quantization method is the same as the independently developed LightNN-1 approach [8]. INQ uses an incremental training approach which involves multiple rounds of training, while LightNN-1 trains the network only once.

To compress the DNNs, Han *et al*. adopted network pruning, quantization and Huffman coding approaches [14]. In the quantization step, they assigned quantized levels within the range of the full-precision weights. Random, density-based and linear quantization methods are compared. Interestingly, the results demonstrate that linear quantization exhibited the best accuracy.

Instead of considering each weight individually, Wu *et al*. split the weight matrices into smaller pieces of vectors, reduced the dimensionality of these vectors, and recorded the mapping relationship in a codebook [36]. Furthermore, they proposed an algorithm to correct the quantization error, for convolutional layers and fully-connected layers of a DNN.

Compared to fixed-point quantization, the advantage of adaptive quantization is that it can use fewer bits to encode the weights, thereby reducing the storage. However, it requires a codebook to record the quantized levels and a decoder to map the encoded bits to continuous values [14] [33] [36] [37], thereby bringing overhead to inference computation.

III. Training Quantized DNNs

Quantizing a trained full-precision network without retraining can lead to significant accuracy loss [3]. Therefore, the quantized DNNs introduced in Section II incorporate the quantization in the training, and thus, can compensate the quantization error. Generally, the training algorithms for these quantized DNNs can be divided into two types, training from scratch [11] [15] [19] [35] or training from a trained full-precision network [14] [33] [34] [36] [37].

A. Training from scratch

Training a quantized DNN from scratch usually follows the training scheme in Figure 2 [11] [15] [19] [35]. First, all the weights are initialized randomly from a uniform or normal distribution. Then, the forward, backward and update steps are repeated for many iterations. In the forward step, the weights are quantized with the pre-set quantization scheme. Then, the intermediate results of the DNN are computed from the lowest layer to the highest one. In the backward step, the weight gradients are computed using the loss function value and all the quantized weights and (quantized) activations. After that, the weights before quantization are updated with the computed gradients. This training scheme is equivalent to the conventional backpropagation algorithm except the quantization operation in the forward pass.

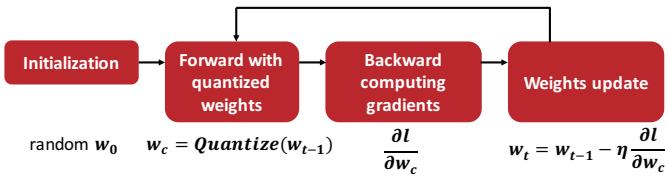


Figure 2. Training quantized DNNs from scratch.

B. Training from a trained full-precision DNN

With a pre-trained full precision DNN, one can also retrain this DNN and involve quantization in the retraining. Zhu *et al*. [37] started from a full-precision network. Then, they quantized and retrained the DNN following a similar scheme as Figure 2. Zhou *et al*. [33] also started with a full-precision DNN, but incrementally quantized a subset of the weights. The quantized weights are frozen for the next iterations, while the remaining floating-point weights continue with this procedure.

Compared with the training-from-scratch scheme, starting from a trained model probably requires fewer iterations since quantizing a trained network even without retraining already has a lower loss function value than the randomly initialized network [15]. However, if no trained model is available, the overall training time will include both training the full-precision DNN and its retraining. In terms of accuracy, retraining from a trained model may be more prone to arrive at a local optimum. However, recent studies show that local optimum does not lead to low accuracy [16].

C. DNN robustness

Quantization maps the weights (and activations) from a huge space into a much smaller one, but does not harm the accuracy with a good training approach and a suitable quantization type. This phenomenon stems from the robustness of DNNs. First, small weight distortion (*e.g*., random Gaussian noise) does not lead to poor accuracy [12]. Second, training or retraining with quantization adaptively changes the weights to compensate for the quantization error. Third, most of the activation functions have a saturation area [18]. If an activation locates at this area, then it is insensitive to small changes of its associated weight.

From a theoretical view, we can explain this robustness with the analysis provided by Choromanska *et al*. [13]. They claim that there exists a nice band in the DNN's loss surface. Within this band, there are many local minima of high quality in terms

of test error. Therefore, as long as the quantization does not lead the weight vector out of this band, the loss function will probably achieve a low value with retraining.

IV. LightNN: A Cost-Effective, Highly Accurate Quantization Approach

LightNN is another quantization approach that aims to reduce both memory accesses and computation energy consumption in DNN's hardware implementation. Its quantization uses a K-ones approximation introduced below.

A. K-ones approximation

In binary representations, any parameter w can be written as a sum of powers of two $w = sign(w) \cdot (2^{n_1} + 2^{n_2} + \cdots + 2^{n_K})$, where K is the number of 1s in w's binary representation. A multiplication of two values w and x is equivalent to several shifts and additions:

$$w \cdot x = sign(w) \cdot (2^{n_1} + 2^{n_2} + \cdots + 2^{n_K}) \cdot x$$
$$= sign(w) \cdot (x \ll n_1 + x \ll n_2 + \cdots + x \ll n_K) \quad (1)$$

where "$x \ll n_1$" means left-shifting x by n_1 bits. For negative values of n_1, right-shifts are used instead. Assuming $n_1 > n_2 > \cdots > n_K$, smaller n values correspond to a less significant part of the result $w \cdot x$. Furthermore, logical shift units are more energy efficient than multipliers. Therefore, the computation energy consumption can be reduced by converting multiplications to approximate versions using a limited number of shifts (and adds). LightNNs change the computation logic of each neuron. A k-ones approximation drops the least significant powers of two in equation (1) such that the resulting value has at most k ones in its binary representation. Figure 3 illustrates a basic example that utilizes a neuron with two inputs. Two weights w_1 and w_2 are both converted to a 2-ones approximation: $w_1 \approx 2^{n_{11}} + 2^{n_{12}}$, $w_2 \approx 2^{n_{21}} + 2^{n_{22}}$. Therefore, a multiplication $w \cdot x$ is changed to two shifts and one addition. Moreover, when $k = 1$, the equivalent multiplier unit is only a shift.

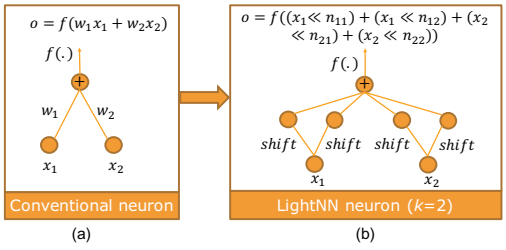


Figure 3. (a) A conventional neuron and, (b) LightNN neuron implemented using a 2-ones approximation.

B. Stochastic rounding

To ensure higher accuracy, LightNNs rely on a stochastic rounding scheme [11]. As opposed to a rounding-to-nearest scheme, the stochastic rounding scheme finds both the nearest higher value w_h and the nearest lower value w_l, and stochastically rounds w to one of them based on a probability distribution:

$$w = \begin{cases} w_h, & w.p.\ p \\ w_l, & w.p.\ 1-p \end{cases},$$

where $p = \frac{w - w_l}{w_h - w_l}$. Intuitively, stochastic rounding ensures that the expected error introduced by the rounding scheme is zero.

C. Activation quantization

LightNNs also use a binarized activation function as described in Section II-A. The advantages of doing so are twofold. First, the multiplication of a weight and an input of a neuron will be limited to ± 1 multiplied by a power of two if the weights are constrained to k-ones approximations where $k = 1$. This leads to an increased likelihood of achieving energy reduction within a hardware implementation. Second, binarized activations also inherently perform a regularization, beneficial to eliminating overfitting for large DNN configurations.

D. LightNN training

We train LightNN following the scheme shown in Figure 2. As shown in Algorithm 1, during each training epoch, the forward pass constrains the weights, and provides intermediate results and loss function. Then, the backward pass computes the derivatives of the loss function over the parameters. After that, the parameters are updated based on the derivatives. It is worth noting that weight constraint occurs only in the forward pass. When updating the weights, we use $w_{t-1} - \eta \frac{\partial l}{\partial w_c}$ instead of $w_c - \eta \frac{\partial l}{\partial w_c}$, where w_{t-1} is the real-value weights after $t-1$ iterations, w_c is the constrained w_{t-1}, η is the learning rate, and l is the loss function. Therefore, the weights are always accumulated in a floating point form. As stated by Courbariaux *et al.* [19], the reason to maintain high resolution for the weights is that the noise needs to be averaged out by the stochastic gradient contributions accumulated in each weight.

Algorithm 1. LightNN Training Epoch

Input: Training dataset $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is input and $\boldsymbol{y}$ is label; parameters after the $(\boldsymbol{t-1})$-th iteration: $\boldsymbol{w_{t-1}}$ (weights) and $\boldsymbol{b_{t-1}}$ (biases); DNN forward computation function $\boldsymbol{f(x, w, b)}$; $\boldsymbol{k}$ value used for $\boldsymbol{k}$-ones approximation $\boldsymbol{approx_k(\cdot)}$; learning rate $\boldsymbol{\eta}$.

Output: Updated weights $\boldsymbol{w_t}$ and biases $\boldsymbol{b_t}$

For each mini-batch of $(\boldsymbol{x}, \boldsymbol{y})$, **do**

1. **Constrain weights**: $\boldsymbol{w_c = approx_k(w_{t-1})}$
2. **Forward**: compute intermediate results and loss function $\boldsymbol{l}$ with $\boldsymbol{f(\cdot)}$, $\boldsymbol{w_c}$, $\boldsymbol{b_{t-1}}$, and mini-batch of $\boldsymbol{x}$
3. **Backward**: compute derivatives $\frac{\partial l}{\partial w_c}$ and $\frac{\partial l}{\partial b_{t-1}}$
4. **Update parameters**: $\boldsymbol{w_t = w_{t-1} - \eta \frac{\partial l}{\partial w_c}}$, and $\boldsymbol{b_t = b_{t-1} - \eta \frac{\partial l}{\partial b_{t-1}}}$

End For

After a LightNN is trained, the last step is to perform the k-ones approximation for all weights. Then, the constrained weights are used for testing.

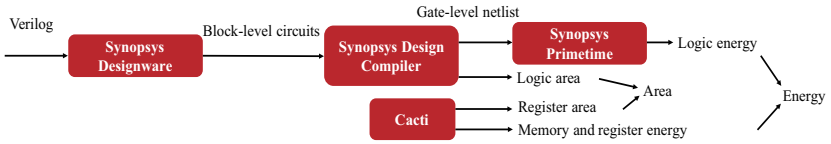


Figure 4. Workflow of energy and area measurement.

V. Experiments

In this section, we compare conventional DNNs, LightNNs, and BNNs in terms of accuracy, storage, energy consumption and area. We also present a guideline for selecting models for hardware implementations to meet the needs of different applications.

A. Set-up

Experiment set-up including model and dataset selection, DNN configurations, training detail and hardware implementation is introduced in this subsection.

Software training. We compare LightNNs with conventional DNNs and BNNs, since BNNs are the most hardware-friendly one among various quantization approaches introduced in section II-D. Seven models are compared – conventional DNN, LightNN-2, LightNN-1, BinaryConnect, LightNN-2-bin, LightNN-1-bin, and BinaryNet. Table 1 describes their main characteristics. ReLU activation function is adopted in the first four models, while the last three models use the hard tanh function for training and the sign function for testing.

Table 1. Constraints on seven models.

Model	Weights	Activation function	Intermediate results	Inputs
Conventional DNN	floating	ReLU	floating	floating
LightNN-2	$\pm(2^{-m_1}+2^{-m_2})$, $m_1, m_2 = 0,1,\dots,7$	ReLU	floating	floating
LightNN-1	$\pm 2^{-m}, m = 0,1,\dots,7$	ReLU	floating	floating
BinaryConnect	+1 or -1	ReLU	floating	floating
LightNN-2-bin	$\pm(2^{-m_1}+2^{-m_2})$, $m_1, m_2 = 0,1,\dots,7$	Sign	+1 or -1	floating
LightNN-1-bin	$\pm 2^{-m}, m = 0,1,\dots,7$	Sign	+1 or -1	floating
BinaryNet	+1 or -1	Sign	+1 or -1	floating

Table 2. Five configurations for two datasets.

Dataset	Configuration	Detail
MNIST	**1-hidden**	One hidden layer with 100 neurons
	2-conv	Two convolution layers and two fully-connected layers
	3-hidden	Three hidden layers each with 4096 neurons
CIFAR-10	**3-conv**	Three convolution layers and one fully-connected layer
	6-conv	Six convolution layers and three fully-connected layers

We experiment on both small and large DNN configurations on MNIST and CIFAR-10 datasets. Both multi-layer perceptrons (MLPs) and convolutional neural networks (CNNs) are adopted. We selected five configurations as shown in Table 2. 3-hidden for MNIST and 6-conv for CIFAR-10 are two large configurations used by Courbariaux *et al.* [15] [19]. 2-conv for MNIST and 3-conv for CIFAR-10 are two smaller configurations borrowed from Caffe examples [20]. 1-hidden for MNSIT is a small network adopted by prior research [10].

In the training phase, we follow the algorithm described in Section IV. Batch normalization and dropout techniques are adopted to accelerate training and avoid overfitting, respectively. MNIST and CIFAR-10 are used for experiment, since Hubara *et al*. provide BNN results on these two datasets [15]. The MNIST dataset contains 70,000 gray-scale hand-written images, while CIFAR-10 contains 60,000 colored images for animals or vehicles. The dataset is divided into training set, validation set, and test set. The validation set is used for selecting the best epoch. The same number of total training epochs is applied to all models and the test error of the epoch with the lowest validation error is reported. These models are trained on Theano platform [21]. We use existing open source models [22] to train conventional DNN, BinaryConnect and BinaryNet. Finally, hinge loss function and ADAM learning rule are used to train all seven models [23].

Hardware inference. We design pipelined implementations with one stage per neuron, for all seven models under consideration. The weights and inputs are initially stored in the memory, and fetched to the pipeline stage logic to compute the output for all neurons in that layer; intermediate results are written back. A 65nm commercial standard library is adopted. The logic computations circuit of one neuron is composed using Synopsys DesignWare commercial IP [24] (*e.g.*, floating point multiplication and addition). The Synopsys Design Compiler [25] is used to generate the gate-level netlist and measure the circuit area. The power consumption of one neuron circuit is calculated using Synopsys Primetime [26]. Cacti [27] is used to obtain the power of memory accesses and registers. The overall workflow is shown in Figure 4. While prior work describes approaches to optimize the CNN hardware implementation (such as data reuse) [28], we keep all the models implemented an unoptimized fashion because our main objective is to compare how constraining weights impacts both computation and memory access energy. The size of the register files is chosen to accommodate the data size required for the computation of the largest neuron.

Table 3. Test error and the number of parameters for all models.

		MNIST			CIFAR-10	
		1-hidden	2-conv	3-hidden	3-conv	6-conv
Number of parameters		79,510	431,080	36,818,954	82,208	39,191,690
Test error	Conventional	1.72%	0.86%	0.75%	21.16%	10.94%
	LightNN-2	1.86%	1.29%	0.83%	24.62%	8.84%
	LightNN-1	2.09%	2.31%	0.89%	26.11%	8.79%
	BinaryConnect	4.10%	4.63%	1.29%	43.22%	9.90%
	LightNN-2-bin	2.94%	1.67%	0.89%	32.58%	10.12%
	LightNN-1-bin	3.10%	1.86%	0.94%	36.56%	9.05%
	BinaryNet	6.79%	3.16%	0.96%	73.82%	11.40%

B. Accuracy

Table 3 summarizes the test error for all configurations and datasets. For most configurations (except a few), the accuracy

decreases from: conventional, LightNN-2, LightNN-1, LightNN-2-bin, LightNN-1-bin, BinaryConnect, BinaryNet. This is because when we constrain the weights and activations, the model suffers from varying levels of accuracy loss. Interestingly, we note that for CIFAR-10 6-conv configuration, the conventional DNN performs no better than other models. This shows the regularization effect of weight constraints. When the DNN is relatively large, it tends to overfit and behave poorly on the testing set. In this case, weight constraints serve as a regularization method to avoid overfitting [19].

C. Storage

Figure 5 compares the weight storage requirements for different models. Since the constraint on activations does not affect weight storage, we only show four models. While it has been shown that limited bit precision can also lead to good accuracy [11], we retain the 32-bit representation for the conventional DNN as baseline. Therefore, a weight in conventional DNNs has four bytes, while in BinaryConnect and BinaryNet, it has only one bit. To store a weight $w = \pm 2^{-m}$, LightNN-1 and LightNN-1-bin need four bits: one bit for $sign(w)$ and another three bits for $|m|$. LightNN-2 and LightNN-2-bin need seven bits for a weight $w = \pm(2^{-m_1} + 2^{-m_2})$: one bit for $sign(w)$, three bits for $|m_1|$ and three bits for $|m_2|$. For easier hardware implementation, one byte is used for a weight of LightNN-2 or LightNN-2-bin. Storage affects the number of memory accesses, and is thereby essential to energy consumption, which is shown in Section V-D.

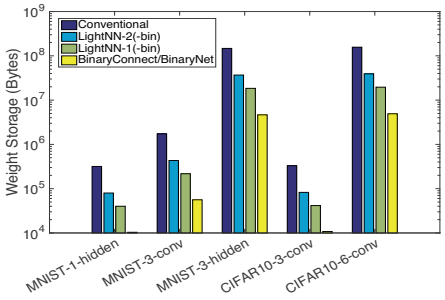


Figure 5. Storage (for weights) required by different models under varying datasets and configurations.

D. Energy and Area

Before we compare the energy and area consumption for a layer or network, the parameters for each multiplier or equivalent multiply unit in all models under consideration are explored first. For BinaryConnect and BinaryNet, a multiply unit is simply an XNOR gate [15]. For LightNN-1 and LightNN-1-bin, it is a shift unit. Since operands (*e.g.*, unbinarized weights, activations and inputs) are represented as single-precision float-point., the shift operation is equivalent to an integer addition for the exponent. LightNN-2 and LightNN-2-bin both rely on two shifts and an add operation. The adder required by LightNN-2 is floating point, while LightNN-2-bin only needs an integer adder to perform fixed point addition. The area and power are reported in Figure 6.

Figure 7 shows the comparison of energy consumption for all seven models considered. Under the same DNN configuration, conventional DNNs and BinaryNet are always the most and least energy-consuming model, respectively. Furthermore, LightNN-2 is more energy-consuming than LightNN-1 and LightNN-2-bin, both of which consume more energy than LightNN-1-bin. When comparing LightNN-1 and LightNN-2-bin, the former has fewer bits for each weight, and therefore consumes less energy for each memory access, while the latter has more energy-efficient logic. The results in Figure 4 show that LightNN-1 has higher energy consumption than LightNN-2-bin in all configurations except MNIST 1-hidden. The same comparison holds for the BinaryConnect and LightNN-1-bin, where BinaryConnect has more energy-consuming logic circuitry (*e.g.*, floating point adder) while LightNN-1-bin has larger weight storage. Although BinaryNet always has the lowest energy consumption under the same configuration, its high accuracy only occurs when the configuration is very large. For example, conventional DNN with 2-conv can surpass BinaryNet with 3-hidden in terms of both accuracy and energy consumption.

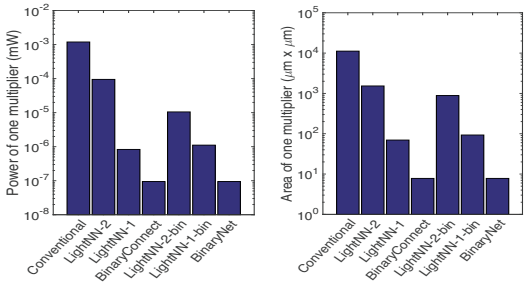


Figure 6. Area and power of an equivalent multiply unit across all models.

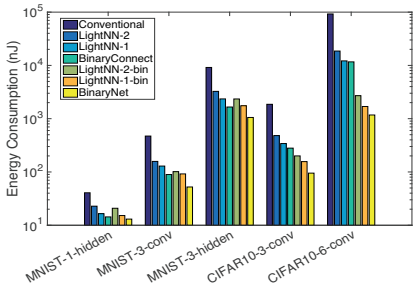


Figure 7. Comparison of energy consumption of different models under varying datasets/configurations. Energy consumption is measured for inferring a single image.

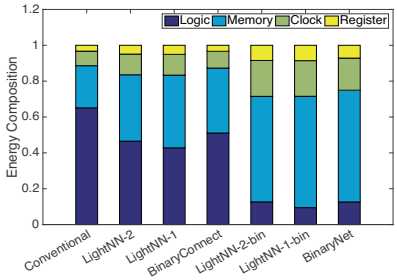


Figure 8. Energy breakdown for all models, averaged across different datasets and configurations.

Figure 8 reports the energy composition for each model. Specifically, the various components of energy are averaged across different configurations and datasets. For the conventional DNNs with floating point circuitry, the most energy-consuming part is the computational portions, while the majority of energy in LightNN-2-bin, LightNN-1-bin and BinaryNet is consumed by memory accesses, though the

absolute values are still smaller than that of conventional DNNs. We also break down the logic energy into leakage, switch and internal energy, where the switch energy is caused by switched load capacitance, and the internal energy is due to internal device switching. The leakage, switch and internal energy take 31.6%, 35.2% and 33.2% respectively, averaged on all models and configurations.

We also compare the area of the seven models under varying configurations in Figure 9. Note that the reported area includes both logic circuits and register files. Also note that the DNN inference is implemented in a pipeline fashion, and the logic is set to handle the largest neuron count in each configuration. Since more computation modules indicate larger area, but fewer memory fetches, the absolute values for the area encompass the energy consumption reported in Figure 7. However, the comparison of different models within a configuration is still meaningful since they use the same (largest) neuron count. The models follow a consistent order (from larger area to smaller): Conventional DNNs, LightNN-2, LightNN-1, BinaryConnect, LightNN-2-bin, LightNN-1-bin, and BinaryNet.

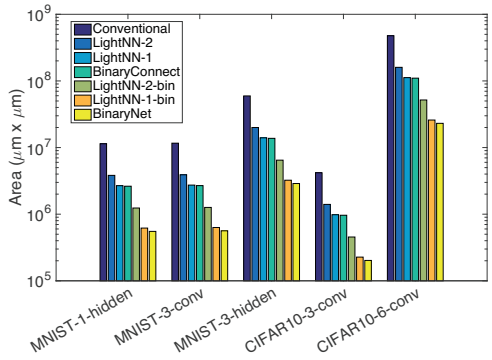
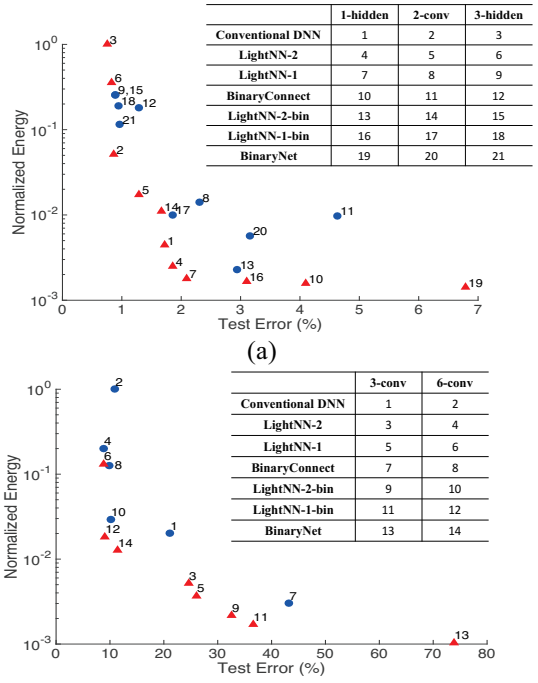


Figure 9. Comparison of area of different models under varying datasets and configurations.



	1-hidden	2-conv	3-hidden
Conventional DNN	1	2	3
LightNN-2	4	5	6
LightNN-1	7	8	9
BinaryConnect	10	11	12
LightNN-2-bin	13	14	15
LightNN-1-bin	16	17	18
BinaryNet	19	20	21

(a)

	3-conv	6-conv
Conventional DNN	1	2
LightNN-2	3	4
LightNN-1	5	6
BinaryConnect	7	8
LightNN-2-bin	9	10
LightNN-1-bin	11	12
BinaryNet	13	14

(b)

Figure 10. Normalized energy and test error of varying models and configurations for (a) MNIST and (b) Cifar-10. Red triangles are Pareto-optimal.

E. Pareto Optimality

A comparison of different models and configurations are presented in Figure 10. Only the red triangles are the Pareto-optimal ones in terms of accuracy and energy. From the figure, we can observe that there is not a model that surpasses all other models in terms of both accuracy and energy. However, with constraints on accuracy or energy, some models are more preferable than others. Suppose one will implement DNN for MNIST on hardware with the constraint that energy consumption per inference is below 200nJ. In this case, the LightNN-2 model with 2-conv configuration is selected, since it has the highest accuracy.

F. Non-pipeline implementation

To further explore how LightNNs perform in a non-pipeline implementation, we implement the DNNs for five benchmarks from the UCI machine learning repository [17]: abalone, banknote-authentication, transfusion, sinknonsink, and balance-scale. They are chosen because of their small DNN configurations, making direct implementations of each DNN practical. (Note that these networks have only three layers, but we still refer to them as DNNs to avoid confusion.) The size of the datasets varies from 600 (balance) to 200,000 (sinknonsink), thus ensuring a large range. Instead of computing each neuron at one stage, the non-pipeline implementation builds the whole DNNs for each dataset. Moreover, to confirm that LightNNs are compatible with the use of limited bit precision for inputs and intermediate results [11], we use both 32-bit and 12-bit implementations. Similar to section V-D, the NNs are implemented using Synopsys Designware commercial IP [24].

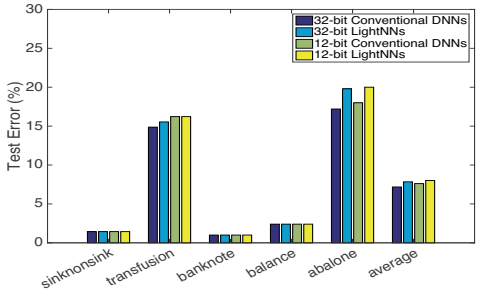


Figure 11. Test accuracy of conventional DNNs and LightNNs with 32-bit and 12-bit implementation.

Figure 11 shows the accuracy for the testing phase for the five benchmarks. Each DNN is trained with 2-ones stochastic approximation. For each benchmark, the accuracy of the 12-bit and 32-bit configurations are shown for both the conventional DNN and LightNN. From an accuracy standpoint, LightNNs lose only 0.66% and 0.40% accuracy when compared to conventional DNNs for 32- and 12-bit implementations, respectively. Furthermore, the accuracy results for 32- and 12-bit data confirm that the additional error incurred by limiting numerical precision is quite small [11]. Finally, the small accuracy differences between 32-bit and 12-bit LightNNs prove that they have high tolerance for limited precision, thereby showing their compatibility with prior work [11].

Twelve-bit conventional DNNs and LightNNs for the five benchmarks are implemented, and three types of energy consumption are measured: leakage, internal energy (caused when the transistor is turned on), and switch energy (caused by toggle), which are all plotted in Figure 12. Estimates of circuit area are also compared in Figure 13. Note that the energy and

area are reported only for the logic. The use of LightNNs reduces total energy by 38.8% on average. Both leakage and dynamic (including internal and switch) energy are reduced, benefiting from the more energy efficient logic implementation. More precisely, the area of a 12-bit multiplier is reduced by 61.6% by the use of the LightNN multiplier. As a result, fewer transistors are required by LightNNs, leading to less leakage and dynamic energy consumption.

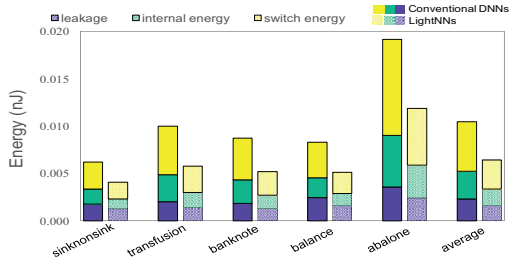


Figure 12. Energy of 12-bit conventional DNNs.

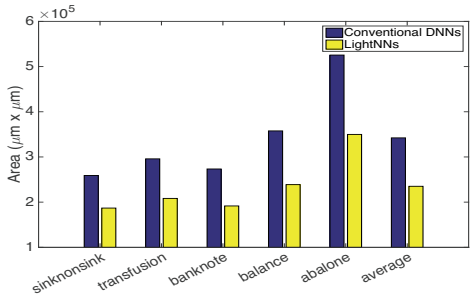


Figure 13. Area of 12-bit conventional DNNs and LightNNs.

VI. Summary and Conclusions

Deep neural networks on hardware have been increasingly demanded, due to the speed requirement for real-time classification applications and the trend of heterogeneous systems. However, the increasing size of DNNs leads to large energy consumption and area requirement. One solution for this challenge is to quantize DNNs' weights (and activations) to reduce the memory accesses and computation energy. As one of the quantization approach, LightNNs modify the computation logic of conventional DNNs by making reasonable approximations, and replace the multipliers with more energy-efficient operators involving only one shift or limited shift-and-add operations. In addition, LightNNs(-bin) also reduce the memory accesses for weights (and activations) storage, thereby further decreasing the energy consumption. Experiment results on conventional DNNs, BNNs and LightNNs show that LightNNs can reduce the energy and area while still maintaining a good accuracy even with fairly small network configurations.

References

[1] G. P. Zhang, "Neural networks for classification: a survey." Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 30(4), pp. 451-462, 2000.

[2] J. Hauswald *et al*., "DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers." ACM ISCA, pp. 27-40, 2015.

[3] Z. Du *et al*., "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators." ASP-DAC, pp. 201-206, 2014.

[4] A. Yazdanbakhsh *et al*., "Neural acceleration for GPU throughput processors." ACM MICRO, pp. 482-493, 2015.

[5] J. Deng, et al., "Imagenet: A large-scale hierarchical image database." IEEE CVPR, pp. 248-255, 2009.

[6] D. Silver *et al*., "Mastering the game of Go with deep neural networks and tree search." Nature, 529(7587), pp. 484-489, 2016.

[7] K. He *et al*., "Deep Residual Learning for Image Recognition." CVPR. arXiv preprint arXiv:1512.03385, 2016.

[8] R. Ding, et al., "LightNN: Filling the Gap between Conventional Deep Neural Networks and Binarized Networks." ACM GLSVLSI, pp. 35-40, 2017.

[9] Y. Netzer, et al., "Reading digits in natural images with unsupervised feature learning." NIPS, No. 2, p. 5, 2011.

[10] S. S. Sarwar, *et al*., "Multiplier-less Artificial Neurons exploiting error resiliency for energy-efficient neural computing." IEEE DATE, pp. 145-150, 2016.

[11] S. Gupta *et al*., "Deep learning with limited numerical precision." ICML, JMLR: W&CP volume 37. arXiv preprint arXiv:1502.02551, 2015.

[12] P. Merolla, P., *et al*., "Deep neural networks are robust to weight binarization and other non-linear distortions." arXiv preprint arXiv:1606.01981, 2016.

[13] A. Choromanska, *et al*. "The loss surfaces of multilayer networks." AISTATS, pp. 192-204, 2016.

[14] H. Song, *et al*., "Learning both weights and connections for efficient neural network." NIPS, pp. 1135-1143, 2015.

[15] I. Hubara, *et al*., "Binarized Neural Networks." NIPS. arXiv preprint arXiv:1602.02505, 2016.

[16] K. Kawaguchi, "Deep learning without poor local minima." NIPS, pp. 586-594, 2016.

[17] A. Asuncion *et al*., "UCI machine learning repository.", 2007. http://archive.ics.uci.edu/ml/

[18] Y. LeCun *et al*., "Deep learning." Nature, 521(7553), pp.436-444, 2015.

[19] M. Courbariaux, *et al*., "Binaryconnect: Training deep neural networks with binary weights during propagations." NIPS, pp. 3123-3131, 2015.

[20] https://github.com/BVLC/caffe/tree/master/examples

[21] http://deeplearning.net/software/theano

[22] https://github.com/MatthieuCourbariaux/BinaryNet

[23] D. Kingma, *et al*., "A method for stochastic optimization." arXiv preprint arXiv:1412.6980, 2014.

[24] https://www.synopsys.com/designware-ip/soc-infrastructure-ip/designware-library.html

[25] Synopsys, Inc., Synopsys Design Compiler. Product Version 14.9. 2014.

[26] https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html

[27] N. Muralimanohar, *et al*., "Cacti 6.0: A tool to model large caches." HP Laboratories, pp. 22–31, 2009.

[28] R. Andri, *et al*., "YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights." IEEE ISVLSI, pp. 236-241, 2016.

[29] A. Graves, *et al*., "Speech recognition with deep recurrent neural networks." IEEE ICASSP, pp. 6645-6649, 2013.

[30] J. S. Coffin, *et al*, U.S. Patent No. 5,991,429. Washington, DC: U.S. Patent and Trademark Office, 1999.

[31] http://anyvision.co/

[32] C. Szegedy, *et al*., "Deep neural networks for object detection." NIPS, pp. 2553-2561, 2013.

[33] A. Zhou, *et al*., "Incremental network quantization: towards lossless CNNs with low-precision weights. ICLR, 2017.

[34] D. Lin, *et al.*, "Fixed point quantization of deep convolutional networks." ICML, pp. 2849-2858, 2016.

[35] S. Zhou, *et al*., "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients." arXiv preprint arXiv:1606.06160, 2016.

[36] J. Wu, *et al*., "Quantized convolutional neural networks for mobile devices." CVPR, pp. 4820-4828, 2016.

[37] C. Zhu, *et al*., "Trained ternary quantization." arXiv preprint arXiv:1612.01064, 2016.