Lightening the Load with Highly Accurate Storage- and Energy-Efficient LightNNs

RUIZHOU DING, ZEYE LIU, R. D. (SHAWN) BLANTON, and DIANA MARCULESCU,
Carnegie Mellon University, USA

Hardware implementations of deep neural networks (DNNs) have been adopted in many systems because of their higher classification speed. However, while they may be characterized by better accuracy, larger DNNs require significant energy and area, thereby limiting their wide adoption. The energy consumption of DNNs is driven by both memory accesses and computation. Binarized neural networks (BNNs), as a tradeoff between accuracy and energy consumption, can achieve great energy reduction and have good accuracy for large DNNs due to their regularization effect. However, BNNs show poor accuracy when a smaller DNN configuration is adopted. In this article, we propose a new DNN architecture, LightNN, which replaces the multiplications to one shift or a constrained number of shifts and adds. Our theoretical analysis for LightNNs shows that their accuracy is maintained while dramatically reducing storage and energy requirements. For a fixed DNN configuration, LightNNs have better accuracy at a slight energy increase than BNNs, yet are more energy efficient with only slightly less accuracy than conventional DNNs. Therefore, LightNNs provide more options for hardware designers to trade off accuracy and energy. Moreover, for large DNN configurations, LightNNs have a regularization effect, making them better in accuracy than conventional DNNs. These conclusions are verified by experiment using the MNIST and CIFAR-10 datasets for different DNN configurations. Our FPGA implementation for conventional DNNs and LightNNs confirms all theoretical and simulation results and shows that LightNNs reduce latency and use fewer FPGA resources compared to conventional DNN architectures.

CCS Concepts: • **Hardware** → **Hardware accelerators**;

Additional Key Words and Phrases: FPGA, deep neural networks, low-power design

ACM Reference format:
Ruizhou Ding, Zeye Liu, R. D. (Shawn) Blanton, and Diana Marculescu. 2018. Lightening the Load with Highly Accurate Storage- and Energy-Efficient LightNNs. *ACM Trans. Reconfigurable Technol. Syst.* 11, 3, Article 17 (December 2018), 24 pages.
https://doi.org/10.1145/3270689

1 INTRODUCTION

As a widely implemented machine-learning model, deep neural networks (DNNs) have shown significant efficiency in classification due to their nonlinear characteristics, flexible configurations, and self-adaptive features [45]. Increasingly in recent years, research has focused on DNNs implemented directly in hardware for a variety of reasons stemming from design requirements or

Authors' addresses: R. Ding, Z. Liu, R. D. (Shawn) Blanton, and D. Marculescu, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213; emails: {rding, zeyel, rblanton, dianam}@andrew.cmu.edu.

1936-7406/2018/12-ART17 $15.00
https://doi.org/10.1145/3270689

application characteristics. First, real-time classification applications (such as Siri and Google glass [11]) have great sensitivity to latency, thereby making pure hardware implementations better candidates than conventional architectures based on CPUs or GPUs. Second, neural networks implemented as custom Application-Specific Integrated Circuits (ASICs) or within FPGAs 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 [7, 34, 43, 46]. In heterogeneous systems, ASICs can handle specific tasks that are required frequently, such as classification tasks in a real-time image recognition application, while CPUs and GPUs can perform the online training. However, hardware implementations of neural networks face difficulties for large-scale deployment. With the increased use of DNNs, the number of layers and neurons increases significantly [5]. Google's AlphaGo adopts a 13-layer architecture, with hundreds of filters per layer [37]. Microsoft implements a 152-layer DNN for image classification [12]. An increasing number of neurons and connections in DNNs require significant energy and power, thereby limiting their wide adoption [5].

2 RELATED WORK

The two main factors limiting the energy efficiency of computing systems in general, and DNN architectures in particular, are memory accesses and computation. Much of the prior research has focused on the computation. For instance, Du et al. adopt inexact circuits for multipliers and adders with less dynamic energy consumption [7]. Li et al. reduce energy with memristor-based approximators [27], while Kim et al. propose an approximate adder using the carry prediction technique to reduce energy [21]. Sarwar et al. use alphabet set multipliers that implement multiplication by selection, shifts, and adds, reducing the energy of neural networks with a precomputed bank [36]. This prior work can reduce the energy consumption of DNNs, and some [7, 21, 27] are also compatible with the DNN architectures introduced in this article.

There has also been significant work on reducing energy of memory accesses. Gupta et al. reduce accesses to memory by limiting parameter precision [9]. Furthermore, Venkataramani et al. propose an energy-efficient neural network design by reducing the bit precision of resilient neurons [42], while Zhang et al. combine computation approximation and memory access reduction to form an approximate computing framework of NNs [47]. Han et al. compressed DNNs to reduce weight storage [10]. Hubara et al. proposed BNNs that constrain the weights and activations to binary values (+1 or −1); they achieve almost no accuracy loss for MNIST and CIFAR-10 datasets [15]. Moreover, the reported accuracy for MNIST is even better than the conventional DNN architecture, especially when the DNN configuration is very large, with many more layers and neurons than typical. On the other hand, BNNs have notably worse accuracy when the smaller Caffe example configurations [20] for MNIST and CIFAR-10 are implemented. Mellempudi et al. proposed ternary neural networks [32] that constrain weights to be α, 0, or $-\alpha$ and use fixed-point quantization for activations. In terms of accuracy and storage, they sit between BinaryConnect and LightNN, both with fixed-point activations. The accuracy of ternary networks for AlexNet is 49.04% [32], while LightNN can achieve 58.6% accuracy, as shown later in Table 4.

In this article, we focus on both computation and memory access energy reduction. We propose a new DNN architecture, LightNN [6], by replacing multipliers with one shift or a limited number of shifts and adds. Marchesi et al. proposed the idea of replacing multipliers of multilayer perceptrons (MLPs) with shifts [31]. Different from their proposed training algorithm where the weights are constrained to be power of two after every iteration, we maintain a set of continuous weights and only use the constrained weights in the forward pass. In addition, we test LightNNs on real datasets and compare their accuracy, energy, and area with conventional DNNs and BNNs. LightNNs are also different from Sarwar et al., where the weights are broken down into two parts with an equal number of shift operations [36]. Instead, LightNNs maintain weights as a whole and only constrain the total number of shifts. Furthermore, we explore the energy consumption of memory accesses. To the best of our knowledge, our work makes the following contributions:

- Similar to prior work [7, 9, 10, 15, 21, 27, 32, 36, 42, 47], we consider the scenario where DNNs are trained in software and used for classification in hardware. Different from prior work, we propose a new neural network architecture, LightNN, which features simpler, more energy-efficient logic. Note that LightNN is compatible with prior work [7, 9, 10] since it can be adopted either standalone or in conjunction with other approaches.
- Compared to BNNs where all the weights are constrained to $+1$ or -1, we relax this constraint and achieve better accuracy than BNNs while still maintaining lower power and area than conventional DNNs. The advantage of this relaxation is that hardware designers can have more options to select the DNN architecture according to their resource constraint. For a fixed DNN configuration, LightNNs have better accuracy at a slight energy increase than BNNs yet are more energy efficient with only slightly less accuracy than DNNs.
- We compare the accuracy, energy, and area of conventional DNNs, BNNs, and LightNNs via industrial-strength design simulation for the MNIST and CIFAR-10 datasets using both an industrial-strength design flow and an FPGA implementation on a VC709 board with a Xilinx FPGA chip Virtex7 VX690T [17]. We also provide a set of guidelines for DNN architecture selection *w.r.t.* accuracy and energy and implement a nonpipeline version of conventional DNNs and LightNNs for datasets from the UCI machine-learning repository [2]. Our results confirm that LightNNs are compatible with approaches that use limited bit precision for inputs and intermediate results [9].

The rest of this article is organized as follows. In Section 3, we first introduce conventional DNNs and BNNs and propose LightNN with its training scheme. Experimental results for the accuracy across different DNN architectures and configurations are shown in Section 4. In Section 5, gate-level hardware simulation results for energy and area of conventional DNNs, BNNs, and LightNNs are compared. In Section 6, we implement the convolutional layers of conventional DNNs and LightNNs on FPGA and compare their latency and resource usage. Finally, conclusions are drawn in Section 7.

3 DNN ARCHITECTURE

In this section, we first describe the operation of conventional DNNs. Then, we introduce the architecture and the training scheme of BNNs. Finally, we propose the new DNN architecture, LightNN, and describe its associated training scheme.

3.1 Conventional DNNs

An artificial neural network (ANN) is called a DNN when there are more than four layers, including the input and output layers. In the training phase with a back-propagation algorithm [25], the loss function, such as the l_2-norm [8], cross-entropy loss [8], or hinge loss [8], is computed using output values and data labels to update the weights used for the linear combination described above. In the deployment (testing) phase, the output neuron with the largest value indicates the prediction result. During deployment, the vast majority of computation resources are used for the multiplication within DNNs [7]. Therefore, prior research has focused on replacing multiplication with other types of logic operations that are more energy efficient. One successful example is BNN [3, 15].

3.2 BNNs

Two types of binarized neural networks (BNNs) have been proposed by Courbariaux et al. BinaryConnect [3], a type of BNN, only constrains the weights to +1 or −1 but leaves the inputs and intermediate results 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 keeps the input values as floating point.

To train a BinaryConnect or BinaryNet, the classic back-propagation algorithm is adopted. The only change is that during each forward pass, the weights are copied and binarized. Then, the binarized weights are used to compute the gradients. Note that the updated weights are always stored as floating-point values, and only in the forward pass of the training phase and during the testing phase, the weights are binarized [3].

During the testing, BinaryNet is different from BinaryConnect only in that it uses a binarized activation function, thereby having binarized intermediate results. In the testing phase, a sign function $f(x) = sign(x)$ is used as the activation function. In the training phase, the 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 can use an XNOR operation to replace multiplications in a hardware implementation.

3.3 LightNNs

Replacing multiplications with a shift or a limited number of shifts and adds can serve as a way to build more energy-efficient DNNs, thereby producing better accuracy. The detailed architecture and training algorithm are described next.

3.3.1 Model Architecture. 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 equal to several shifts and additions:

$$\begin{aligned} w \cdot x &= sign(w) \cdot (2^{n_1} + 2^{n_2} + \cdots + 2^{n_K}) \cdot x \\ &= sign(w) \cdot (x << n_1 + x << n_2 + \cdots + x << n_K), \end{aligned} \tag{1}$$

where "$x << n_1$" indicates 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 1 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 approximate multiplier unit is only a shift.

Furthermore, we use a stochastic rounding scheme [9]. 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 either w_h or w_l based on the following:

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

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.

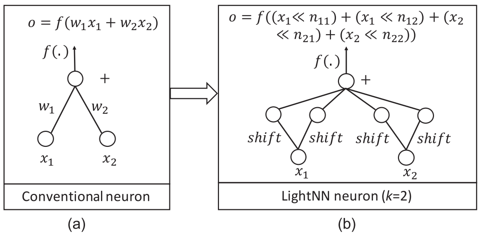


Fig. 1. (a) A conventional neuron and (b) LightNN neuron implemented using a 2-ones approximation.

3.3.2 LightNN Training. We adopt a training algorithm similar to that of BNNs. As shown in Algorithm 1, during each training epoch, the training data $\boldsymbol{x}$ and $\boldsymbol{y}$ are randomly split into mini-batches. For each mini-batch, the forward pass constrains the weights and provides intermediate results and the value of the loss function. Then, the backward pass computes the derivatives of the loss function over the parameters. The parameters are then updated based on the derivatives. It is worth noting that constraining the weights occurs only in the forward pass. When updating the weights, we use $\boldsymbol{w}_{t-1} - \eta \frac{\partial F}{\partial \boldsymbol{w}_c}$ instead of $\mathbf{w}_c - \eta \frac{\partial F}{\partial \boldsymbol{w}_c}$, where $\boldsymbol{w}_{t-1}$ is the real-value weights after $(t-1)$-th iterations, $\boldsymbol{w}_c$ is the constrained $\boldsymbol{w}_{t-1}$, η is the learning rate, and F is the loss function. Therefore, the weights are always accumulated in a floating-point form. As stated by Courbariaux et al. [3], the reason to maintain high resolution for the weights is that the noise needs to be averaged out by the stochastic gradient contributions accumulated for each weight. 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.

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 $(t-1)$-th iteration: $\boldsymbol{w}_{t-1}$
(weights) and $\boldsymbol{b}_{t-1}$ (biases); DNN forward computation function $g(x, w, b)$; $k$ value used for $k$-ones
approximation $\boldsymbol{approx}_k(\cdot)$; learning rate $\eta$.
Output: Updated weights $\boldsymbol{w}_t$ and biases $\boldsymbol{b}_t$.
for each mini-batch of $\boldsymbol{x}, \boldsymbol{y}$ do
    1. Constrain weights: $\boldsymbol{w}_c = \boldsymbol{approx}_k(\boldsymbol{w}_{t-1})$
    2. Forward: compute intermediate results and loss function $F$ with $g(\cdot)$, $\boldsymbol{w}_c$, $\boldsymbol{b}_{t-1}$, and mini-batch of $\boldsymbol{x}$
    3. Backward: compute derivatives $\frac{\partial F}{\partial \boldsymbol{w}_c}$ and $\frac{\partial F}{\partial \boldsymbol{b}_{t-1}}$
    4. Update parameters: $\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \eta \frac{\partial F}{\partial \boldsymbol{w}_c}$
end
```

3.3.3 LightNN with Binarized Activations. LightNN can also use a binarized activation function, as described in Section 3.2. The advantage of doing so is that 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.

3.4 LightNN Accuracy Analysis

To give a theoretical analysis to the convergence of LightNN, we use a similar methodology to Li et al. [28]. Let m be the number of mini-batches used for back-propagation. Let $F(\boldsymbol{w})$ be the loss function. Then, $F(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{w})$, where $f_i(\boldsymbol{w})$ is the loss function for the ith mini-batch. Assume that both $\nabla\tilde{f}(\cdot)$ and $F(\cdot)$ are L-Lipschitz smooth, i.e., $\forall \boldsymbol{x}, \boldsymbol{y} \in [-1, 1]^d$, $|\nabla\tilde{f}(\boldsymbol{x}) - \nabla\tilde{f}(\boldsymbol{y})| \leq L_1|\boldsymbol{x} - \boldsymbol{y}|$, and $|F(\boldsymbol{x}) - F(\boldsymbol{y})| \leq L_2|\boldsymbol{x} - \boldsymbol{y}|$. Let $\boldsymbol{w}^t$ be the weight vector after t iterations

($t \in \{1, 2, \ldots, T\}$, where T is the total number of iterations). Then, at each iteration the stochastic gradient descent algorithm updates the weights as follows:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \mu_t \nabla \tilde{f}(Q(\boldsymbol{w}^t)) = \boldsymbol{w}^t - \mu_t \nabla \tilde{f}(\boldsymbol{w}^t) + \boldsymbol{r}^t, \tag{2}$$

where $\boldsymbol{r}^t = \mu_t \nabla \tilde{f}(\boldsymbol{w}^t) - \mu_t \nabla \tilde{f}(Q(\boldsymbol{w}^t))$ is the error introduced by quantization, $Q(\cdot)$ is the quantization function using stochastic rounding, and μ_t is the learning rate at the tth iteration. The quantization function can be described by k and c, where k indicates that the LightNN uses k-ones approximation (also denoted as LightNN-k), and $\frac{1}{2^c}$ is the highest resolution for the quantization, i.e., the smallest positive quantized value. In addition, we assume that the gradient norm is bounded by a constant G^2: $\mathbb{E}||\nabla \tilde{f}(\boldsymbol{w}^t)||^2 \leq G^2$. Let D be the diameter of the weight domain. Then, the Euclidean distance between any two weights is bounded by D^2. Assume that $\forall i \in \{1, 2, \ldots, d\}$, where d is the dimension of $\boldsymbol{w}^t$; the probability density function (PDF) of the ith element of $\boldsymbol{w}^t$ monotonically decreases with its absolute value, i.e., $p_{\boldsymbol{w}_i^t}(a) \leq p_{\boldsymbol{w}_i^t}(b)$ if $|a| > |b|$.

Next, we first bound $\mathbb{E}[||\boldsymbol{r}^t||^2]$ and then bound the expected distance between the loss function of LightNN and conventional DNN.

3.4.1 Bounding $\mathbb{E}[||\boldsymbol{r}^t||^2]$. To bound $\mathbb{E}[||\boldsymbol{r}^t||^2]$, we first consider the case where c is very large, so we can show the rate of $\mathbb{E}[||\boldsymbol{r}^t||^2]$ in terms of k. Then, we add the constraint from c.

Let the quantization error for weights be denoted as $\boldsymbol{\gamma}^t = Q(\boldsymbol{w}^t) - \boldsymbol{w}^t$. Therefore, $\mathbb{E}[||\boldsymbol{r}^t||^2] \leq \mu_t^2 L_1^2 \mathbb{E}[||\boldsymbol{\gamma}^t||^2]$. Then, we can bound $\mathbb{E}[||\boldsymbol{r}^t||^2]$ by bounding $\mathbb{E}[||\boldsymbol{\gamma}^t||^2]$.

Lemma 3.1. *Assume that $c \to \infty$. Let $\mathbb{E}[||\boldsymbol{\gamma}^t||^2]$ for LightNN-k be denoted as $E(k)$. Then, $E(k+1) \leq \frac{1}{7}E(k)$, $\forall k \geq 1$.*

Proof. For LightNN-k, consider any two adjacent quantized levels a and b. Without losing generality, let us assume $0 < a < b$. Then, if $a < \boldsymbol{w}_i^t < b$, the quantization error for $\boldsymbol{w}_i^t$ follows: $\boldsymbol{\gamma}_i^t = Q(\boldsymbol{w}_i^t) - \boldsymbol{w}_i^t \leq b - a$. Therefore,

$$\mathbb{E}\left[\left(\boldsymbol{\gamma}_i^t\right)^2 | a < \boldsymbol{w}_i^t < b\right] \leq (b-a)^2. \tag{3}$$

For LightNN-($k+1$), a and b are still used as quantization levels. Furthermore, the legal quantized values between a and b also include $a + 2^{-1}(b-a), a + 2^{-2}(b-a), \ldots$. Let β_j denote the jth quantized value, i.e., $\beta_j = a + 2^{-j}(b-a)$, where $j = 1, 2, \ldots$. Then, $b > \beta_1 > \beta_2 > \cdots > a$. Therefore,

$$\begin{aligned}
\mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | a < \boldsymbol{w}_i^t < b] &= \sum_{j=1}^{\infty} \mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | \beta_{j+1} < \boldsymbol{w}_i^t < \beta_j] P(\beta_{j+1} < \boldsymbol{w}_i^t < \beta_j | a < \boldsymbol{w}_i^t < b) \\
&\leq \sum_{j=1}^{\infty} 2^{-2j}(b-a)^2 P(\beta_{j+1} < \boldsymbol{w}_i^t < \beta_j | a < \boldsymbol{w}_i^t < b) \\
&\leq \sum_{j=1}^{\infty} 2^{-2j}(b-a)^2 2^{-j} = \frac{(b-a)^2}{7}.
\end{aligned} \tag{4}$$

Note that $P(\beta_{j+1} < \boldsymbol{w}_i^t < \beta_j | a < \boldsymbol{w}_i^t < b) \leq 2^{-j}$ holds since the PDF of the weight is monotonically decreasing with its absolute value. This means that when we increase the number of shifts used for the k-approximation from k to $k+1$, $\mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | a < \boldsymbol{w}_i^t < b]$ will decrease by a factor less than $\frac{1}{7}$ for any interval $[a, b]$, where a and b are two adjacent quantization levels. Let all the quantized values for LightNN-k be denoted as $\{a_j\}_{j=1,2,\ldots}$, where $1 \geq a_j \geq a_{j+1} > 0$. Therefore,

$$\begin{aligned} E(k+1) &= \sum_{i=1}^{d}\sum_{j=1}^{\infty} \mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | a_{j+1} < \boldsymbol{w}_i^t < a_j; k+1] P(a_{j+1} < \boldsymbol{w}_i^t < a_j) \\ &\leq \sum_{i=1}^{d}\sum_{j=1}^{\infty} \frac{1}{7}\mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | a_{j+1} < \boldsymbol{w}_i^t < a_j; k] P(a_{j+1} < \boldsymbol{w}_i^t < a_j) = \frac{1}{7}E(k). \end{aligned} \tag{5}$$

□

Lemma 3.2. *Assume that $c \to \infty$. Let $\sigma_t^2 \triangleq \max_i \mathbb{E}[(\boldsymbol{w}_i^t)^2]$. LightNN-k quantization error satisfies $\mathbb{E}[||\boldsymbol{\gamma}^t||^2] \leq \frac{\sigma_t^2 d}{4 \cdot 7^{k-1}}$.*

Proof. We first bound $\mathbb{E}[||\boldsymbol{\gamma}^t||^2]$ for LightNN-1. Since the PDF for $\boldsymbol{w}_i^t$ is assumed to be symmetric *w.r.t.* $\boldsymbol{w}_i^t = 0$, $\mathbb{E}[||\boldsymbol{\gamma}^t||^2] = \mathbb{E}[||\boldsymbol{\gamma}^t||^2 | \boldsymbol{w}_i^t > 0]$. Therefore, we only need to consider $\boldsymbol{w}_i^t > 0$. Then, $\boldsymbol{w}_i^t$ is rounded to $2^{\lfloor \log_2 \boldsymbol{w}_i^t \rfloor}$ or $2^{\lceil \log_2 \boldsymbol{w}_i^t \rceil}$, where $\lfloor \cdot \rfloor$ is the floor operation and $\lceil \cdot \rceil$ is the ceiling operation. Therefore,

$$\begin{aligned} \mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | \boldsymbol{w}_i^t] &= (\boldsymbol{w}_i^t - 2^{\lfloor \log_2 \boldsymbol{w}_i^t \rfloor})^2 \cdot \frac{2^{\lceil \log_2 \boldsymbol{w}_i^t \rceil} - \boldsymbol{w}_i^t}{2^{\lfloor \log_2 \boldsymbol{w}_i^t \rfloor}} + (2^{\lceil \log_2 \boldsymbol{w}_i^t \rceil} - \boldsymbol{w}_i^t)^2 \cdot \frac{\boldsymbol{w}_i^t - 2^{\lfloor \log_2 \boldsymbol{w}_i^t \rfloor}}{2^{\lfloor \log_2 \boldsymbol{w}_i^t \rfloor}} \\ &= (\boldsymbol{w}_i^t - 2^{\lfloor \log_2 \boldsymbol{w}_i^t \rfloor})(2^{\lceil \log_2 \boldsymbol{w}_i^t \rceil} - \boldsymbol{w}_i^t) \leq \frac{(2^{\lfloor \log_2 \boldsymbol{w}_i^t \rfloor})^2}{4} \leq \frac{(\boldsymbol{w}_i^t)^2}{4}. \end{aligned} \tag{6}$$

Therefore,

$$\mathbb{E}[||\boldsymbol{\gamma}^t||^2] = \sum_{i=1}^{d} \mathbb{E}[(\boldsymbol{\gamma}_i^t)^2] = \sum_{i=1}^{d} \mathbb{E}[\mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | \boldsymbol{w}_i^t]] \leq \sum_{i=1}^{d} \mathbb{E}\left[\frac{(\boldsymbol{w}_i^t)^2}{4}\right] = \frac{\sigma_t^2 d}{4}. \tag{7}$$

Extending the bound to LightNN-k with Lemma 3.1, we have

$$\mathbb{E}[||\boldsymbol{\gamma}^t||^2] \leq \frac{\sigma_t^2 d}{4 \cdot 7^{k-1}}. \tag{8}$$

□

Now we add the constraint from c; that is, the smallest positive quantized value is $\frac{1}{2^c}$. For example, if $k = 1$ and $c = 3$, then the legal quantized levels include $\pm 1, \pm\frac{1}{2}, \pm\frac{1}{4}, \pm\frac{1}{8}$.

Theorem 3.3. *For LightNN-k, the error introduced by quantization is characterized by*

$$\mathbb{E}[||\boldsymbol{r}^t||^2] \leq \frac{\mu_t^2 L_1^2 \sigma_t^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}}. \tag{9}$$

Proof. Again, we consider the case $\boldsymbol{w}_i^t > 0$. First, let us check the way c influences the rounding error. Without constraining c (*i.e.*, $c \to \infty$), the quantized levels cut the range of $[0, 1]$ into infinite nonoverlapping intervals. Define $\mathcal{A} \subset [0, 1]$ as the set of values that fall into an interval whose length is smaller than $\frac{1}{2^c}$. Then, with the constraint of the smallest possible interval $\frac{1}{2^c}$, any $\boldsymbol{w}_i^t \in \mathcal{A}$ will have a larger expected rounding error, while any $\boldsymbol{w}_i^t \notin \mathcal{A}$ will not be influenced. Now suppose $\boldsymbol{w}_i^t \in [a, b]$, where $a, b \in [0, 1]$ are two adjacent quantized levels, and $b - a = \frac{1}{2^c}$. The expected rounding error for $\boldsymbol{w}_i^t$ is bounded by

$$\begin{aligned} \mathbb{E}[(\boldsymbol{\gamma}_i^t)^2 | \boldsymbol{w}_i^t \in [a, b]] &= \frac{\boldsymbol{w}_i^t - a}{b - a} \cdot (b - \boldsymbol{w}_i^t)^2 + \frac{b - \boldsymbol{w}_i^t}{b - a} \cdot (\boldsymbol{w}_i^t - a)^2 \\ &= (\boldsymbol{w}_i^t - a)(b - \boldsymbol{w}_i^t) \leq \frac{(b-a)^2}{4} = \frac{1}{2^{2c+2}}. \end{aligned} \tag{10}$$

Note that this is larger than the bound in Equation (4) due to the constraint from c. Then, the quantization error bound for the weights in $\mathcal{A}$ will increase to $\frac{1}{2^{2c+2}}$. Therefore, the overall bound is changed to

$$\begin{aligned}\mathbb{E}[||\boldsymbol{\gamma}^t||^2] &= \sum_{i=1}^{d}\mathbb{E}[(\boldsymbol{\gamma}_i^t)^2] = \sum_{i=1}^{d}\mathbb{E}[(\boldsymbol{\gamma}_i^t)^2|\boldsymbol{w}_i^t \in \mathcal{A}]P(\mathcal{A}) + \mathbb{E}[(\boldsymbol{\gamma}_i^t)^2|\boldsymbol{w}_i^t \notin \mathcal{A}](1-P(\mathcal{A}))\\ &\leq \sum_{i=1}^{d}\mathbb{E}[(\boldsymbol{\gamma}_i^t)^2|\boldsymbol{w}_i^t \in \mathcal{A}] + \mathbb{E}[(\boldsymbol{\gamma}_i^t)^2|\boldsymbol{w}_i^t \notin \mathcal{A}] \leq \frac{d}{2^{2c+2}} + \frac{\sigma_t^2 d}{4\cdot 7^{k-1}}.\end{aligned} \tag{11}$$

Therefore,

$$\mathbb{E}[||\boldsymbol{r}^t||^2] \leq \mu_t^2 L_1^2 \mathbb{E}[||\boldsymbol{\gamma}^t||^2] \leq \frac{\mu_t^2 L_1^2 \sigma_t^2 d}{4\cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}}. \tag{12}$$

□

3.4.2 Convergence Analysis.

LEMMA 3.4. *$\forall i \in \{1,2,\ldots,d\}, \forall t \in \{1,2,\ldots,T\}$; if applying stochastic rounding on $\boldsymbol{w}_i^t$ leads to error $\boldsymbol{\gamma}_i^t$, then $\mathbb{E}[\boldsymbol{\gamma}_i^t \boldsymbol{w}_i^t] = 0$.*

PROOF. Since stochastic rounding has the property: $\mathbb{E}[\boldsymbol{\gamma}_i^t|\boldsymbol{w}_i^t] = 0$, we have

$$\mathbb{E}[\boldsymbol{\gamma}_i^t \boldsymbol{w}_i^t] = \mathbb{E}[\mathbb{E}[\boldsymbol{\gamma}_i^t \boldsymbol{w}_i^t|\boldsymbol{w}_i^t]] = \mathbb{E}[\boldsymbol{w}_i^t \mathbb{E}[\boldsymbol{\gamma}_i^t|\boldsymbol{w}_i^t]] = 0. \tag{13}$$

In other words, the expected bounding error is zero. □

We now give the bound for the expected distance between the loss function of LightNNs and corresponding conventional DNNs. The optimal solution in the continuous domain is defined as $\boldsymbol{w}^* \triangleq argmin_{\boldsymbol{w}\in[-1,1]^d} F(\boldsymbol{w})$. The quantized weights averaged by T training iterations are defined as $\tilde{\boldsymbol{w}}^T \triangleq \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{w}^t$.

THEOREM 3.5. *Assume that the loss function $F(\boldsymbol{w})$ is convex within domain $[-1,1]^d$ and the learning rate follows $\mu_t = \frac{\mu_1}{\sqrt{t}}$. Let $\sigma^2 = \max_t\{\sigma_t^2\} \leq 1$. Then, the expected distance between the loss function of the LightNN and the corresponding DNN vanishes for a large number of iterations T:*

$$\mathbb{E}[F(\tilde{\boldsymbol{w}}^T) - F(\boldsymbol{w}^*)] \leq \frac{D^2(\sqrt{T}+1)}{2T\mu_1} + \frac{\mu_1\sqrt{T+1}}{T}\left(\frac{1}{2}G^2 + \frac{L_1^2\sigma^2 d}{4\cdot 7^{k-1}} + \frac{L_1^2 d}{2^{2c+2}}\right) \sim O\left(\frac{1}{\sqrt{T}}\right). \tag{14}$$

PROOF. From Equation (2), we have

$$\begin{aligned}\mathbb{E}||\boldsymbol{w}^{t+1} - \boldsymbol{w}^*||^2 &= \mathbb{E}||\boldsymbol{w}^t - \mu_t\nabla\tilde{f}(\boldsymbol{w}^t) + \boldsymbol{r}^t - \boldsymbol{w}^*||^2\\ &= \mathbb{E}||\boldsymbol{w}^t - \boldsymbol{w}^*||^2 + \mathbb{E}||\mu_t\nabla\tilde{f}(\boldsymbol{w}^t) - \boldsymbol{r}^t||^2 - 2\mathbb{E} < \boldsymbol{w}^t - \boldsymbol{w}^*, \mu_t\nabla\tilde{f}(\boldsymbol{w}^t) - \boldsymbol{r}^t > .\end{aligned} \tag{15}$$

Due to Lemma 3.4 and $\mathbb{E}[\tilde{f}(\boldsymbol{w}^t)] = \mathbb{E}[F(\boldsymbol{w}^t)]$, we have

$$\mathbb{E}||\boldsymbol{w}^{t+1} - \boldsymbol{w}^*||^2 = \mathbb{E}||\boldsymbol{w}^t - \boldsymbol{w}^*||^2 + \mu_t^2\mathbb{E}||\nabla\tilde{f}(\boldsymbol{w}^t)||^2 + \mathbb{E}||\boldsymbol{r}^t||^2 - 2\mathbb{E}[< \boldsymbol{w}^t - \boldsymbol{w}^*, \mu_t\nabla F(\boldsymbol{w}^t) >]. \tag{16}$$

Analyzing the right-hand side, we have that $\mathbb{E}||\nabla\tilde{f}(\boldsymbol{w}^t)||^2 \leq G^2$, $\mathbb{E}||\boldsymbol{r}^t||^2$ is bounded by 3.3, and $\mathbb{E}[< \boldsymbol{w}^t - \boldsymbol{w}^*, \mu_t\nabla F(\boldsymbol{w}^t) >] \geq \mu_t\mathbb{E}[F(\boldsymbol{w}^t) - F(\boldsymbol{w}^*)]$, due to the convexity of function $F(.)$. Therefore, we have

$$\mathbb{E}||\boldsymbol{w}^{t+1} - \boldsymbol{w}^*||^2 \leq \mathbb{E}||\boldsymbol{w}^t - \boldsymbol{w}^*||^2 + \mu_t^2 G^2 + \frac{\mu_t^2 L_1^2 \sigma_t^2 d}{4\cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}} - 2\mu_t\mathbb{E}[F(\boldsymbol{w}^t) - F(\boldsymbol{w}^*)]. \tag{17}$$

This leads to

$$\mathbb{E}[F(\boldsymbol{w}^t) - F(\boldsymbol{w}^*)] \leq \frac{1}{2\mu_t}\left(\mathbb{E}||\boldsymbol{w}^t - \boldsymbol{w}^*||^2 - \mathbb{E}||\boldsymbol{w}^{t+1} - \boldsymbol{w}^*||^2 + \mu_t^2 G^2 + \frac{\mu_t^2 L_1^2 \sigma_t^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}}\right). \quad (18)$$

Summing up for t from 1 to T, we have

$$\sum_{t=1}^{T} \mathbb{E}[F(\boldsymbol{w}^t) - F(\boldsymbol{w}^*)] \leq \sum_{t=1}^{T} \frac{1}{2\mu_t}\left(\mathbb{E}||\boldsymbol{w}^t - \boldsymbol{w}^*||^2 - \mathbb{E}||\boldsymbol{w}^{t+1} - \boldsymbol{w}^*||^2 + \mu_t^2 G^2 + \frac{\mu_t^2 L_1^2 \sigma_t^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}}\right)$$
$$\leq \frac{1}{2\mu_1}\mathbb{E}||\boldsymbol{w}^1 - \boldsymbol{w}^*||^2 + \sum_{t=2}^{T}\left(\frac{1}{2\mu_t} - \frac{1}{2\mu_{t-1}}\right)||\boldsymbol{w}^t - \boldsymbol{w}^*||^2 + \sum_{t=1}^{T} \mu_t \left(\frac{1}{2}G^2 + \frac{L_1^2 \sigma_t^2 d}{4 \cdot 7^{k-1}} + \frac{L_1^2 d}{2^{2c+2}}\right). \quad (19)$$

Since $\mu_t = \frac{\mu_1}{\sqrt{t}}$, we have $\sum_{t=1}^{T} \mu_t \leq \mu_1 \sqrt{T+1}$, $\sqrt{t} - \sqrt{t-1} = \frac{1}{\sqrt{t}+\sqrt{t-1}} \leq 1$. Also, due to the convexity of function $F(.)$, $\mathbb{E}[F(\frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}^t) - F(\boldsymbol{w}^*)] \leq \frac{1}{T}\sum_{t=1}^{T} \mathbb{E}[F(\boldsymbol{w}^t) - F(\boldsymbol{w}^*)]$. In addition, $||\boldsymbol{w}^1 - \boldsymbol{w}^*||^2 \leq D^2$. Therefore,

$$\mathbb{E}[F(\tilde{\boldsymbol{w}}) - F(\boldsymbol{w}^*)] \leq \frac{D^2(\sqrt{T}+1)}{2T\mu_1} + \frac{\mu_1\sqrt{T+1}}{T}\left(\frac{1}{2}G^2 + \frac{L_1^2 \sigma^2 d}{4 \cdot 7^{k-1}} + \frac{L_1^2 d}{2^{2c+2}}\right). \quad (20)$$

□

When T goes to infinity, the expected distance between $F(\bar{\boldsymbol{w}})$ and $F(\boldsymbol{w}^*)$ converges to zero at the rate of $\frac{1}{\sqrt{T}}$. This indicates that at each iteration, although the gradients are computed in an approximate way $w.r.t.$ a close quantized point instead of the full-precision weights, the final trained solution will still converge to the optimal point.

Since we still need a one-step quantization for the weights at the end of the training process, there will be an extra error that remains positive as T goes to infinity. Using Equation (11), this can be obtained by

$$\mathbb{E}[F(Q(\bar{\boldsymbol{w}})) - F(\boldsymbol{w}^*)] = \mathbb{E}[F(Q(\bar{\boldsymbol{w}})) - F(\bar{\boldsymbol{w}}) + F(\bar{\boldsymbol{w}}) - F(\boldsymbol{w}^*)]$$
$$\leq L_2 \frac{d}{2^{2c+2}} + \frac{\sigma^2 d}{4 \cdot 7^{k-1}} + O\left(\frac{1}{\sqrt{T}}\right). \quad (21)$$

4 TRAINING AND TESTING EXPERIMENTS

We adopt the pattern of software training and hardware inference for DNNs. In this section, we describe our experiment setup and compare conventional DNNs, LightNNs, and BNNs in terms of accuracy and storage. Hardware implementation results are described in subsequent sections.

4.1 Accuracy

We first introduce the experiment setup, including the DNN architectures, datasets, DNN configurations, and training approaches. Then, the accuracy results of different architectures are reported and compared.

4.1.1 Setup. Different DNN architectures, configurations, and training settings are described as follows. Our goal is to involve (1) both Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) and (2) both large and small network configurations, to make the results more representative.

DNN architectures. Seven DNN architectures are considered: conventional DNN, LightNN-2, LightNN-1, BinaryConnect, LightNN-2-bin, LightNN-1-bin, and BinaryNet. Table 1 describes their main characteristics. The ReLU activation function is adopted in the first four architectures, while the last three architectures use the hard tanh function for training and the sign function for testing.

Table 1. Architecture Characteristics for the Experiments

Architecture	Weights	Activation Function	Intermediate Results	Inputs
Conventional DNN	32-bit floating point	ReLU	floating	floating
LightNN-2	$\pm(2^{-m_1} + 2^{-m_2}), m_1, m_2 = 0, 1, \ldots 7$	ReLU	floating	floating
LightNN-1	$\pm 2^{-m}, m = 0, 1, \ldots 7$	ReLU	floating	floating
BinaryConnect	± 1	ReLU	floating	floating
LightNN-2-bin	$\pm(2^{-m_1} + 2^{-m_2}), m_1, m_2 = 0, 1, \ldots 7$	Sign	± 1	floating
LightNN-1-bin	$\pm 2^{-m}, m = 0, 1, \ldots 7$	Sign	± 1	floating
BinaryNet	± 1	Sign	± 1	floating

Table 2. The Five Architecture Configurations for the MNIST and CIFAR-10 Experiments

Dataset	**Configuration**	Detail
MNIST	**1-hidden**	One hidden layer with 100 neurons
	2-conv	Two convolutional layers and two fully connected layers
	3-hidden	Three hidden layers each with 4,096 neurons
CIFAR-10	**3-conv**	Three convolutional layers and one fully connected layer
	6-conv	Six convolutional layers and three fully connected layers

Datasets. We test the seven architectures on two datasets: MNIST [26] and CIFAR-10 [23]. The MNIST dataset contains 70,000 gray-scale hand-written images, while CIFAR-10 contains 60,000 colored images for animals and vehicles.

DNN configurations. Both MLPs and CNNs are adopted for MNIST and CIFAR-10. We selected five configurations as shown in Table 2. The basic idea is to include both small and large DNN configurations to determine how different architectures perform under varying configurations: 3-hidden for MNIST and 6-conv for CIFAR-10 are two large configurations used by Courbariaux et al. [3, 15]; 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 simple configuration adopted by prior research [36].

Training approach. The training algorithm is described in Section 3.3.2. Batch normalization [19] and dropout [40] techniques are adopted to accelerate training and avoid overfitting, respectively. The dataset is divided into a 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 architectures and the test error of the epoch with the lowest validation error is reported. These architectures are trained on the Theano platform [41]. We use existing open-source models [14] to train the conventional DNN, BinaryConnect, and BinaryNet. Finally, the hinge loss function [8] and ADAM learning rule [22] are used to train all seven architectures [22].

4.1.2 *Results.*

Results for different DNN architectures and configurations are presented in Table 3.

Comparison. 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 architecture suffers from varying levels of accuracy loss.

Table 3. Test Error and Number of Parameters for All DNN Architectures

		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%	8.86%
	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%

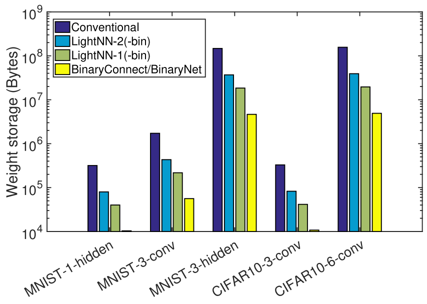


Fig. 2. Storage (for weights) required by different DNN architectures under varying datasets and configurations.

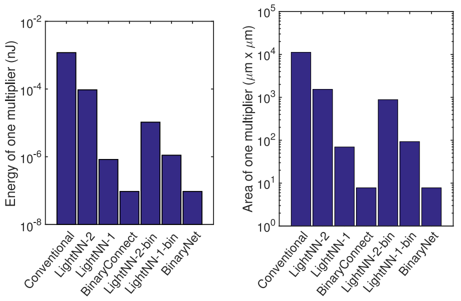


Fig. 3. Area and energy of an approximate multiply unit across all DNN architectures.

4.2 Storage Requirements

The weight storage requirements for different DNN architectures are compared in Figure 2. Since the constraint on activations does not affect weight storage, we only show four DNN architectures. While it has been shown that limited-bit precision can also lead to sufficient accuracy [9], we retain the 32-bit representation for the conventional DNN as baseline. Therefore, a weight in conventional DNNs has 4 bytes, while BinaryConnect and BinaryNet only require 1 bit. To store a weight $w = \pm 2^{-m}$, LightNN-1 and LightNN-1-bin need 4 bits: 1 bit for $sign(w)$ and another 3 bits for $|m|$. LightNN-2 and LightNN-2-bin need 7 bits for a weight $w = \pm(2^{-m_1} + 2^{-m_2})$: 1 bit for $sign(w)$, 3 bits for $|m_1|$, and 3 bits for $|m_2|$. For easier hardware implementation, 1 byte is used for LightNN-2 or LightNN-2-bin weights. Storage affects the number of memory accesses and is therefore essential to energy consumption, which is investigated in Section 5.

4.3 ImageNet Results

To see if LightNNs can perform well on large-scale datasets and large networks, we conduct experiments on the ImageNet dataset [4] with AlexNet [24] and VGG-16 [38]. The training settings for LightNN are the same as conventional DNNs, except for the customized learning rates. For AlexNet, we train LightNN from scratch, while for VGG-16, we train from pretrained full-precision weights due to time constraints. LightNN-1 is used for experiment, where each weight is constrained to be $\{\pm 2^0, \pm 2^{-1}, \ldots, \pm 2^{-7}\}$. The results are shown in Table 4. The test error for

Table 4. Test Error for Conventional DNNs, LightNN-1 and BinaryConnect on ImageNet

Model	**AlexNet**		**VGG-16**	
	Top-1 err.	Top-5 err.	Top-1 err.	Top-5 err.
Conventional	43.4%	19.8%	27.0%	8.8%
LightNN-1	41.4%	19.7%	28.2%	9.4%
BinaryConnect	64.6%	39.0%	N/A	N/A

The Accuracy for BinaryConnect is Borrowed from Rastegari et al. [35] Where Only AlexNet Results are Available.

Table 5. Comparison of LightNN and Fixed-Point Weights in Terms of Weight Storage and Test Error

Model	**Network size**							
	1,536		512		128		64	
	Storage	Error	Storage	Error	Storage	Error	Storage	Error
Conventional	165MB	6.45%	19MB	7.12%	1MB	10.59%	313KB	14.02%
Fixed-point$_{8W8A}$	41MB	6.46%	5MB	7.33%	0.3MB	10.68%	78KB	14.58%
Fixed-point$_{4W8A}$	20MB	6.57%	2MB	7.35%	0.1MB	10.91%	39KB	16.50%
LightNN-1$_{4W8A}$	20MB	6.49%	2MB	7.14%	0.1MB	10.69%	39KB	15.40%
BinaryConnect	5MB	6.89%	0.6MB	8.77%	0.04MB	15.28%	10KB	23.14%

Network size indicates the number of filters for the largest convolutional layer in the network. "xW" means x bits for weights, and "yA" means y bits for activations.

conventional DNNs is directly borrowed from prior work [24, 38]. LightNN-1 can achieve slightly better accuracy than conventional DNNs for AlexNet and has competitive accuracy for VGG-16.

4.4 Comparison with Fixed-Point Weights

Fixed-point quantization [9, 48] for weights and activations has also been used to reduce storage and energy for DNNs. LightNN, as a variant of quantized DNNs, is compatible with fixed-point activation quantization. We compare the fixed-point weights and LightNN, both with 8-bit fixed-point activations, in Table 5. We start from the network configuration by Hubara [13], which is a VGG-like network whose "widest" convolutional layer has 1,536 filters, and then reduce the number of filters per layer to see the model accuracy at different network sizes. We have also changed the activation function from ReLU to leaky ReLU [29] since the latter achieves higher accuracy for all the models. From Table 5, LightNN-1$_{4W8A}$ can achieve slightly higher accuracy than fixed-point$_{4W8A}$, while they have the same weight storage. Further comparison regarding their FPGA resources is shown in Table 12.

4.5 Varying Number of Shifts

Since the number of shift-and-add operators, k, determines the approximation accuracy per weight, it is worthwhile to analyze how accuracy changes when k changes. We conduct the experiment on a customized small convolutional network with only 19,738 parameters, so the gap between $k = 1$ and $k = 32$ is large. We constrain the maximum bits per shift to be 7, which is the same setting used for LightNNs in Table 1. The results are shown in Table 6. As expected, larger k has higher accuracy, yet in this case the accuracy for $k = 3$ is close to $k = 32$. The per-weight bit storage (PWBS) for $k = 32$ is 32 (not 128) because it can record which bits to shift as the 32-bit fixed-point weight usually does.

Table 6. Per-Weight Bit Storage (PWBS) and Test Error at Varied k (Number of Shift-and-Add Operators for an Equivalent Multiplier)

k	1	2	3	5	32
PWBS	4	8	12	20	32
Error	26.01%	22.01%	21.19%	20.97%	20.89%

Table 7. Per-Weight Bit Storage (PWBS) and Test Error of LightNN-1 at Varied c (Maximum Number of Bits to Shift)

c	1	2	3	7	15
PWBS	2	3	3	4	5
Error	38.51%	33.04%	26.32%	26.01%	25.88%

4.6 Varying Number of Shifting Bits

It is also interesting to see how the maximum number of shifting bits, c, affects the accuracy. We gradually increase c for LightNN-1 from 1 to 15, and show the results in Table 7. The small network used by Table 6 is also adopted here. For this case, the accuracy for $c = 7$ is close to $c = 15$ but can use 4 bits per weight instead of 5 bits.

5 HARDWARE EVALUATION

In this section, we show the gate-level circuit simulation results for energy and area across different DNN architectures and configurations. We also present a guideline for selecting DNN architectures for hardware implementations to satisfy the varying requirements of different applications.

5.1 Setup

For all seven DNN architectures under consideration, we designed pipelined implementations with one stage per neuron, where the computation unit is reused for each neuron. The weights and inputs are initially stored in the memory and fetched into the pipeline stage 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 constructed using Synopsys Designware commercial IP [39] (e.g., floating-point multiplication and addition). The Synopsys Design Compiler [30] 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 [16]. Cacti [33] is used to obtain the energy of memory accesses and registers. While prior work describes approaches (such as data reuse) to optimize the CNN hardware implementation [1], we keep all the DNN architectures implemented in an unoptimized fashion because our main objective is to compare how constraining weights impact 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.

5.2 Multipliers versus Approximate Multiply Units

Energy and area of each multiplier or approximate multiply unit in all DNN architectures 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 floating 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 energy consumption are reported in Figure 3.

5.3 Energy Consumption

Figure 4 shows the comparison of energy consumption for all seven DNN architectures considered. Under the same DNN configuration, conventional DNNs and BinaryNet are always the most and

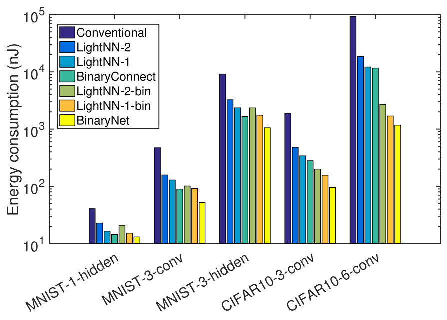


Fig. 4. Comparison of energy consumption of different DNN architectures under varying datasets/configurations. Energy consumption is measured for inferring a single image.

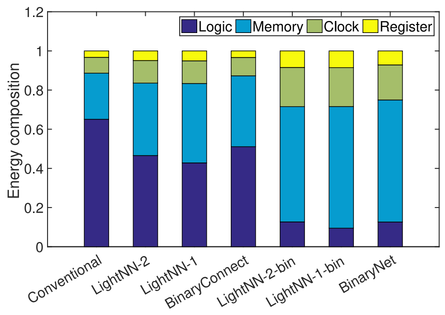


Fig. 5. Energy breakdown for all DNN architectures, averaged across different datasets and configurations.

least energy-consuming architecture, 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, a conventional DNN with 2-conv configuration can surpass BinaryNet with 3-hidden configuration in terms of both accuracy and energy consumption for the MNIST dataset.

To explore the energy composition for each DNN architecture, we report the results in Figure 5. 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 DNN architectures and configurations.

5.4 Area Comparison

We also compare the area of the seven DNN architectures under varying configurations in Figure 6. 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 a larger area but fewer memory fetches, the absolute values for the area encompass the energy consumption reported in Figure 4. However, the comparison of different DNN architectures within a configuration is still meaningful since they use the same (largest) neuron count. The DNN architectures follow a consistent order (from larger area to smaller): Conventional DNNs, LightNN-2, LightNN-1, BinaryConnect, LightNN-2-bin, LightNN-1-bin, and BinaryNet.

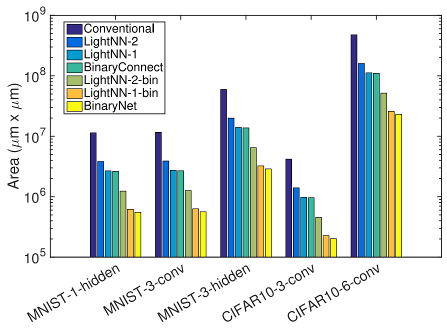
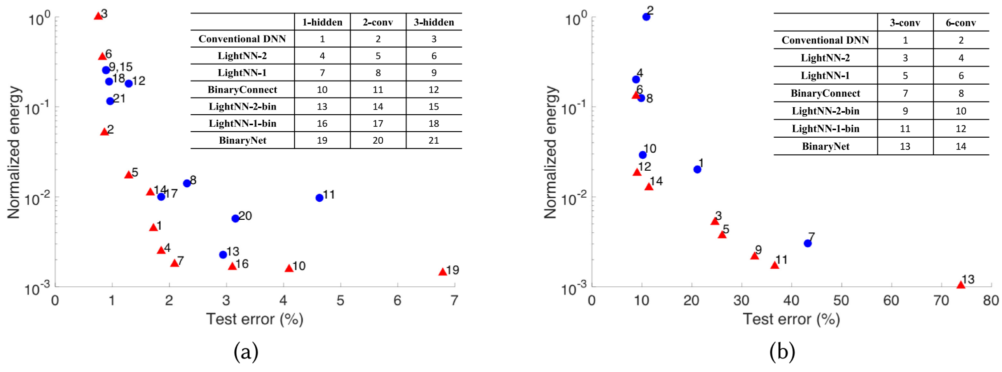


Fig. 6. Comparison of area of different DNN architectures under varying datasets and configurations.



	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

	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

Fig. 7. Normalized energy and test error of varying DNN architectures and configurations for (a) MNIST and (b) Cifar-10. Red triangles are Pareto optimal.

5.5 Guidelines for Architecture Selection

An interesting question is whether there is a DNN architecture that always surpasses another in terms of both accuracy and energy, under varying configurations and datasets. The answer is no. However, with constraints on accuracy or energy, some DNN architectures 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 architecture with the 2-conv configuration is the best, since it has the highest accuracy.

Does higher accuracy always require higher energy? Again, the answer is no. A comparison of different architectures and configurations is presented in Figure 7. Only the red triangles are the Pareto-optimal ones in terms of accuracy and energy.

5.6 Non-Pipeline Implementation

To further explore how LightNNs perform in a nonpipeline implementation, we implement the ANNs for five benchmarks from the UCI machine-learning repository [2]: abalone, banknote-authentication, transfusion, sinknonsink, and balance-scale. They are chosen because of their small ANN configurations, making direct implementations of whole ANNs possible. The size of the datasets varies from 600 (balance) to 200,000 (sinknonsink), so a greater coverage is ensured. Instead of computing each neuron at one stage, the nonpipeline implementation builds the whole ANN for each dataset. Moreover, to confirm that LightNNs are compatible with the use of limited

Table 8. Test Error of Conventional DNNs and LightNNs with 32-Bit and 12-Bit Implementation

Model	Benchmark					Average
	sinknonsink	transfusion	banknote	balance	abalone	
32-Bit Conventional	1.44%	14.87%	1.00%	2.40%	17.20%	7.18%
32-Bit LightNN	1.44%	15.54%	1.00%	2.40%	19.80%	7.84%
12-Bit Conventional	1.44%	16.22%	1.00%	2.40%	18.00%	7.61%
12-Bit LightNN	1.44%	16.22%	1.00%	2.40%	20.00%	8.01%

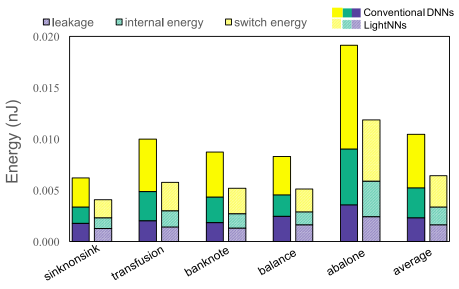


Fig. 8. Energy of 12-bit conventional DNNs and LightNNs.

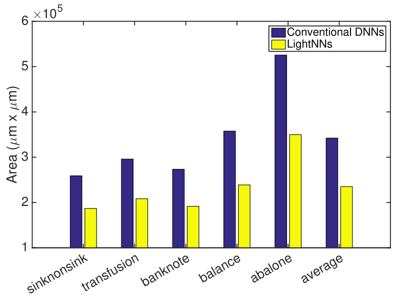


Fig. 9. Area of 12-bit conventional DNNs and LightNNs.

bit precision for inputs and intermediate results [9], we use both 32-bit and 12-implementations. Similar to Section 5, the ANNs are implemented using Synopsys Designware commercial IP [39].

5.6.1 Accuracy. Table 8 shows the accuracy for the testing phase for the five benchmarks. All of them are 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 DNNs and LightNNs. 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, on average across the five benchmarks. Furthermore, the accuracy results for 32- and 12-bit data confirm that the additional error incurred by limiting numerical precision is quite small [9]. 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 [9, 42, 47].

5.6.2 Energy and Area. Twelve-bit conventional DNNs and LightNNs for the five benchmarks are implemented and all three types of energy consumption are measured: leakage, internal, and switch energy, all shown in Figure 8. The area is also compared in Figure 9. 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 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.

6 FPGA IMPLEMENTATION

In this section, the FPGA implementations of conventional DNNs and LightNN are compared. Our goal is to demonstrate that the quantization method provided by LightNN leads to a resource-efficient and low-latency FPGA implementation. There are no detailed analytic models of FPGA architectures and circuitry, so we must evaluate architectures with actual mapped hardware.

However, for any DNN implementation, there are many potential solutions that result in a large design space. Work by Zhang et al. [44] finds that there could be as much as a 90% performance difference between two different solutions with the same FPGA logic utilization. It is nontrivial to determine the optimal solution, especially when limitations on computation resources and memory bandwidth of the FPGA platform are considered. Therefore, in this experiment, to evaluate the efficacy, we will use the same design optimization solution for both conventional DNNs and LightNN.

6.1 HLS Accelerator Implementation

The Vivado high-level synthesis (HLS) [18] is used to describe the functionality for each layer within a conventional DNN and a LightNN. A primary capability of Vivado HLS lies in the rapid assessment of resources, throughput, and performance tradeoffs. The FPGA implementation is produced according to pragma and directives inserted into the C code (or into a separate directive file) that instruct the HLS compiler how to synthesize the algorithm. Therefore, the same pragma and directives are used for both the conventional DNNs and the LightNN.

In addition, a previous study [44] proved that convolution operations typically take over 90% of the computation time. Therefore, in this work, we will focus on the comparison of convolutional layer implementation in conventional DNN and LightNN. A more specific design optimization solution for LightNN will be studied in future work.

The pseudo code shown in Algorithm 2 illustrates the detailed design optimization solution for a convolutional layer. The convolutional layer transforms a 3D input volume to a 3D output volume of neuron activations. For example, the depth of the 3D input volume shown in Algorithm 2 is N. In other words, there are N input channels. Similarly, the depth of the 3D output volume shown in Algorithm 2 is M, which means that there are M output channels. The parameters of the convolutional layer consist of a set of trainable weight kernels. Every weight kernel is small spatially (along width and height) but extends through the full depth of the input volume. For example, a weight kernel of the convolutional layer shown in Algorithm 2 has dimensions $N \times K \times K$ (i.e., K pixels width and height and N input channels because input volume has depth N). Although the convolutional layer significantly reduces data volumes by weight sharing, loop tiling is still mandatory to fit a small portion of data on-chip for FPGA implementation. Therefore, we begin by applying loop tiling for the 3D input volume. To be specific, the depth of the input volume is tiled by T_n, the height of the input volume is tiled by T_r, and the width of the input volume is tiled by T_c (shown in lines 3 to 4 in Algorithm 2). Therefore, the input volume is tiled into small blocks with dimensions $T_n \times T_r \times T_c$. The depth of the 3D output volume is tiled by T_m as well (shown in line 1 in Algorithm 2). Then, the tiled inputs, weights, and biases (which we did not show in Algorithm 2 for simplicity) are loaded on-chip. However, an improper tiling may degrade the efficiency of data reuse and parallelism of data processing. Thus, different tiling settings (such as different values of T_m, T_n, T_r, and T_c) are exercised to identify the optimal solution in terms of the timing latency and resource usage.

Second, the computation engine is optimized as well for the tiled data. In Algorithm 2, the objective of computation optimization is to enable efficient loop unrolling (shown in lines 13 and 15) and pipelining (shown in line 11) while fully utilizing all computation resources provided by the FPGA hardware platform. Loop unrolling can be used to increase the utilization of massive computation resources in FPGA devices. Unrolling along different loop dimensions will generate different implementation variants. Whether and to what extent two unrolling execution instances share data will affect the complexity of generated hardware and eventually affect the number of unrolled copies and the hardware operation frequency.

Table 9. Data-Sharing Relations of Convolutional Operations

Loop Iterator	Tiled Inputs	Tiled Weights	Tiled Outputs
row_k	dependent	independent	irrelevant
col_k	dependent	independent	irrelevant
row	dependent	irrelevant	independent
col	dependent	irrelevant	independent
tt_o	irrelevant	independent	independent
tt_i	independent	independent	irrelevant

ALGORITHM 2: Pseudo code of a convolutional layer implementation using Vivado HLS

Input: weights W, inputs for layer I, tiling size for output channels T_m, tiling size for input channels T_n, tiling size for the width of input volume T_c, tiling size for the height of input volume T_r.
Output: outputs for layer O.

```
for ($t_o$= 0; $t_o$< $M$; $t_o$+=$T_m$) {
  for ($t_i$= 0; $t_i$< $N$; $t_i$+=$T_n$) {
    for ($row_t$= 0; $row_t$< $R$; $row_t$+=$T_r$) {
      for ($col_t$= 0; $col_t$< $C$; $col_t$+=$T_c$){
        $I_t = I[t_i : t_i + T_n][row_t : row_t + T_r][col_t : col_t + T_c]$        // load tiled inputs
        $W_t = W[t_o : t_o + T_m][t_i : t_i + T_n][0 : K][0 : K]$        // load tiled weights
        for ($row_k$= 0; $row$<$K$; $row_k$+=1) {
          for ($col_k$= 0; $col$<$K$; $col_k$+=1) {
            for ($row$= $row_t$; $row$<$row_t$+$T_r$; $row$+=1) {
              for ($col$= $col_t$; $col$<$col_t$+$T_c$; $col$+=1) {
              #pragma HLS pipeline
                for ($tt_o$= $t_o$; $tt_o$<$t_o$+$T_m$; $tt_o$+=1) {
                #pragma HLS UNROLL
                  for ($tt_i$= $t_i$; $tt_i$<$t_i$+$T_n$; $tt_i$+=1) {
                  #pragma HLS UNROLL
                  $O_t[tt_o][row][col]$ += $W_t[tt_o][tt_i][row_k][col_k] \times I_t[tt_i][S \times row + row_k][S \times col + col_k]$;
        }}}}}}
        // write back tiled outputs $O_t$ to $O$
}}}}
```

Table 9 shows the data-sharing relation of convolutional computation. The data-sharing relations between different loop iterations of a loop dimension on a given array can be classified into three categories: irrelevant, independent, and dependent [44]:

- *Irrelevant.* If loop iterator i does not appear in any access function of an array A, the corresponding loop dimension i is irrelevant to array A. For example, loop iterator row does not appear in any access of the tiled weight W_t shown in Algorithm 2. Therefore, the corresponding loop of loop iterator row is irrelevant to the tiled weight W_t.
- *Independent.* If the data space accessed on an array A is totally separable along a certain loop dimension i, the loop dimension i is independent of array A. For example, the first two dimensions accessed on the tiled weight W_t are totally separable along the certain loops whose iterators are tt_o and tt_i, respectively. Therefore, the corresponding loop of iterators tt_o and tt_i is independent of the tiled weight W_t.

Table 10. DNN Configuration for CIFAR-10

Layer	Input Channel	Output Channel	Output Dim
Conv1	3	128	32
Conv2	128	128	32
Conv3	128	256	16
Conv4	256	256	16
Conv5	256	512	8
Conv6	512	512	8
FC1	8192	1024	1
FC2	1024	10	1
FC3	1024	10	1

- *Dependent.* If the data space accessed on an array is not separable along a certain loop dimension i, the loop dimension i is dependent on A. For example, the last two dimensions accessed on the tiled input I_t cannot be separable along the certain loops whose loop iterators are row, col, row_k, and col_k. Therefore, the corresponding loop of the iterator row, col, row_k, and col_k is dependent on the tiled input I_t.

An independent data-sharing relation generates direct connections between buffers and computation engines. An irrelevant data-sharing relation generates broadcast connections. A dependent data-sharing relation generates interconnections with complex topology. The data-sharing relations of the pseudo code are shown in Table 9. Loop dimensions tt_o and tt_i are selected to be unrolled to avoid complex connection topologies for all arrays. tt_o and tt_i are permuted to the innermost loop levels to simplify HLS code generation.

On the other hand, loop pipelining is another key optimization technique in HLS to improve system throughput by overlapping the execution of operations from different loop iterations. The throughput achieved is limited both by resource constraints and data dependencies in the application. Loop-carried dependence will prevent loops from being fully pipelined. According to the analysis by Zhang et al. [44], the loop *col* can be fully pipelined for achieving better performance.

6.2 Experiment Setup

As described previously, conventional DNNs and LightNN are implemented with Vivado HLS [18]. The C code of DNN designs is parallelized by adding HLS-defined pragma and the parallel version is validated with the Vivado HLS timing analysis tool. This tool enables fast presynthesis simulation for the implemented design. Presynthesis resource reports are used for design space exploration and performance estimation. Our implementation is built on the VC709 board, which has a Xilinx FPGA chip Virtex7 485t. Its working frequency is 100MHz. Presynthesis is executed on an Intel i7-4790 CPU (3.6GHz) with 16GB RAM.

Table 10 summarizes the detailed DNN configuration used in this experiment. This configuration is the 6-conv for CIFAR-10 dataset from Section 4.1.1, which contains sixty thousand 32x32 three-channel images. The configuration consists of six convolutional layers followed by three fully connected layers. All convolutional layers use 3x3 filters and edge padding.

6.3 Experimental Results

In this subsection, we first compare design space exploration results between conventional DNNs and LightNN. Then, the resource usage comparison among different design solutions is reported.

Table 11. Timing Performance Comparison between Conventional DNN, Fixed-Point DNN, and LightNN

Layer	**Tiling Size**				**Execution Cycles**			**Speedup**[†]	
	T_m	T_n	T_r	T_c	Conventional	FP_{4W8A}	LN-1_{4W8A}	FP_{4W8A}	LN-1_{4W8A}
Conv1	128	3	32	32	727,852	596,135	596,132	22%	22%
Conv2	128	128	32	32	1,016,853	862,944	861,977	18%	18%
Conv3	256	128	16	16	697,364	582,907	582,905	20%	20%
	128	128	16	16	1,034,279	879,364	879,359		
Conv4	256	256	16	16	1,034,262	879,355	879,353	18%	18%
	128	128	16	16	3,182,669	2,921,362	2,921,361		
Conv5	512	256	8	8	N/A[⋆]	2,532,367	2,532,429	35%	35%
	256	256	16	16	2,592,811	2,206,110	2,206,088		
	128	128	16	16	9,871,513	9,033,054	9,032,871		
Conv6	512	512	8	8	N/A[⋆]	4,516,331	4,516,286	119%	119%
	256	256	8	8	9,871,437	9,033,141	9,033,050		
	128	128	8	8	38,584,609	36,692,238	36,691,500		

FP_{4W8A} is short for fixed-point$_{4W8A}$, i.e., fixed-point network with 4-bit weights and 8-bit activations. LN-1_{4W8A} refers to LightNN-1_{4W8A}, i.e., LightNN-1 with 4-bit weights and 8-bit activations. Both LN-1_{4W8A} and FP_{4W8A} can achieve higher speed without running out of FPGA resources.
[⋆]N/A indicates that FPGA runs out of its resource for achieving the corresponding tiled setting.
[†]Speedup describes the speed increase for the FP_{4W8A} and the LN-1_{4W8A} compared to conventional DNN under their best tiling size for each layer, respectively.

Table 11 shows different tile size tuples $<T_m, T_n, T_r, T_c>$ and their corresponding execution cycles for the DNN configuration shown in Table 10. The execution cycles are reported by Vivado presynthesis simulation for three different quantization methods: (1) Conventional, which encodes the weight and output of a neuron as 32-bit floating-point numbers; (2) FP_{4W8A}, which uses the 4-bit fixed-point representation for weights and the 8-bit fixed-point representation for intermediate results; and (3) LN-1_{4W8A}, which uses the 4-bit predefined format to represent weights and the 8-bit fixed-point representation for the intermediate results. The N/A entries in Table 11 show that FPGA resources are exhausted for the corresponding tiled setting. The first observation from Table 11 shows that with a large tiling size, DNN can achieve a significant advantage in timing latency. Another observation is that for the same tiling size, the FP_{4W8A} and the LN-1_{4W8A} have an advantage in execution cycles, which leads to smaller latency. However, the latency improvement achieved for the same tiling size between the conventional DNN and the LN-1_{4W8A} is not as large as expected. A possible explanation may be that latency improves for the optimized design of the conventional DNN but not LN-1_{4W8A}. Therefore, a more specific design optimization solution for the LN-1_{4W8A} should be sought for latency improvement.

Table 11 also shows the speedup for the FP_{4W8A} and the LN-1_{4W8A} compared to the conventional DNN, where the best tiling size for each DNN is used. Since the conventional DNN requires many more resources than the FP_{4W8A} and the LN-1_{4W8A} with the same tiling size (more detail shown in Table 12), the conventional DNN runs out of FPGA resources when increasing the tiling size to $512 \times 512 \times 8 \times 8$ for the largest two layers (*Conv*5 and *Conv*6). Therefore, the FP_{4W8A} and the LN-1_{4W8A}, which reduce the resource requirement, allow larger tiling sizes, and therefore reduce the latency. In addition, we notice that FP_{4W8A} and the LN-1_{4W8A} have similar speedups, which might also be explained by the fact that optimized design for the conventional DNN doesn't benefit from the significant reduction in FPGA resources that LN-1_{4W8A} has.

Table 12. FPGA Resource Usage Comparison between Conventional DNN, Fixed-Point DNN, and LightNN

Layer	**Tiling Size**				**DNN Architecture**	**FPGA Resource**			
	T_m	T_n	T_r	T_c		BRAM	DSP	FF	LUT
Conv1	128	3	32	32	Conventional	40	32	3,455	3,522
					FP_{4W8A}	13	8	461	1,112
					$\mathrm{LN\text{-}1}_{4W8A}$	13	1	401	1,120
Conv2	128	128	32	32	Conventional	1,040	1,281	120,788	96,350
					FP_{4W8A}	448	257	2,627	7,798
					$\mathrm{LN\text{-}1}_{4W8A}$	448	1	701	9,711
Conv3	256	128	16	16	Conventional	1,280	1,281	120,759	96,340
					FP_{4W8A}	416	257	2,611	7,774
					$\mathrm{LN\text{-}1}_{4W8A}$	416	1	687	9,686
	128	128	16	16	Conventional	768	1,281	120,748	96,035
					FP_{4W8A}	416	257	2,605	7,483
					$\mathrm{LN\text{-}1}_{4W8A}$	416	1	681	9,396
Conv4	256	256	16	16	Conventional	2,432	2,561	240,838	191,380
					FP_{4W8A}	800	513	4,746	14,524
					$\mathrm{LN\text{-}1}_{4W8A}$	800	1	902	18,388
	128	128	16	16	Conventional	768	1,281	120,747	96,125
					FP_{4W8A}	416	257	2,613	7,493
					$\mathrm{LN\text{-}1}_{4W8A}$	416	1	689	9,406
Conv5	512	256	8	8	Conventional	N/A⋆	N/A⋆	N/A⋆	N/A⋆
					FP_{4W8A}	1,040	513	8,817	17,832
					$\mathrm{LN\text{-}1}_{4W8A}$	1,040	1	4,976	21,695
	256	256	8	8	Conventional	2,368	2,561	240,782	190,637
					FP_{4W8A}	528	513	8,811	17,283
					$\mathrm{LN\text{-}1}_{4W8A}$	528	1	4,970	21,147
	128	128	8	8	Conventional	704	1,281	120,698	95,622
					FP_{4W8A}	272	257	4,638	8,856
					$\mathrm{LN\text{-}1}_{4W8A}$	272	1	2,715	10,769
Conv6	512	512	8	8	Conventional	N/A⋆	N/A⋆	N/A⋆	N/A⋆
					FP_{4W8A}	1,040	1,025	17,161	34,565
					$\mathrm{LN\text{-}1}_{4W8A}$	1,040	1	9,405	42,364
	256	256	8	8	Conventional	2,368	2,561	240,800	190,642
					FP_{4W8A}	528	513	8,820	17,292
					$\mathrm{LN\text{-}1}_{4W8A}$	528	1	4,979	21,156
	128	128	8	8	Conventional	704	1,281	120,722	95,636
					FP_{4W8A}	272	257	4,646	8,876
					$\mathrm{LN\text{-}1}_{4W8A}$	272	1	2,723	10,789
Available resource						2,940	3,600	866,400	433,200

FP_{4W8A} is short for fixed-point$_{4W8A}$, i.e., fixed-point network with 4-bit weights and 8-bit activations. $\mathrm{LN\text{-}1}_{4W8A}$ refers to LightNN-1_{4W8A}, i.e., LightNN-1 with 4-bit weights and 8-bit activations. With the same tiling size, LightNN-1 requires fewer resources than conventional DNN and fixed-point DNN.

⋆N/A indicates that FPGA runs out of its resource for achieving the corresponding tiled setting.

Table 13. FPGA Resource Utilization, Latency, and Test Error Comparison

DNN Architecture	Maximum Resource Utilization				Normalized Latency*	Test Error
	BRAM	DSP	FF	LUT		
Conventional	80.5%	71.1%	27.8%	44.0%	100%	8.86%
FP_{4W8A}	35.3%	28.4%	1.98%	7.98%	18.0%	8.98%
LN-1_{4W8A}	35.3%	0.03%	1.09%	9.78%	18.0%	8.79%

*Latency is computed by summing up the lowest latency for six convolutional layers under FPGA resource constraint, and normalized by the latency of conventional DNN.

To be specific, Table 12 illustrates the FPGA resource usage comparison between the conventional DNN, the FP_{4W8A}, and the LN-1_{4W8A}. In more detail, the BRAM, DSP, flip-flop, and LUT usage are reported by the Vivado tool for different tiling settings. The last row of Table 12 shows the available resources for the Virtex7-VC709 FPGA board used in this experiment. Similar to Table 11, an N/A entry means that the FPGA resources are exhausted for the corresponding tiling size. The first observation from Table 12 shows that with the same tiling size, the FP_{4W8A} and the LN-1_{4W8A} require fewer FPGA resources than the conventional DNN. These reductions can be explained by the fact that the quantization methods provided by the FP_{4W8A} and the LN-1_{4W8A} simplify the computation of DNN. Another observation is that with the same tiling size, the LN-1_{4W8A} requires fewer DSPs and FFs but more LUTs than the FP_{4W8A}. This is due to the fact that the LN-1_{4W8A} utilizes the LUT to implement the described multiplier replacement (shift operation) instead of the standard DSP unit. Overall, Table 12 indicates that LightNN-1 can reduce latency by using a larger tiling size while requiring fewer FPGA resources.

Table 13 compares three DNNs $w.r.t.$ the maximum resource utilization, latency, and test error. The maximum resource utilization and normalized latency are computed with the tiling size that (1) does not exhaust FPGA resources and (2) minimizes the latency. Therefore, the LightNN-1 can significantly reduce both the FPGA resource usage and latency while maintaining accuracy.

7 CONCLUSION

The increasing demand of deep neural networks for real-time classification and the trend of heterogeneous systems leveraging the high speed of ASICs and FPGAs accelerate the pace of hardware implementations for DNNs. Due to the increasing size of DNNs, energy and area requirements have become a very challenging problem. 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 also reduce weight storage, thereby decreasing the memory access energy. Experimental results of gate-level hardware simulation show that LightNNs fill the gap between conventional DNNs and BNNs in terms of accuracy, storage, energy, and area and provide more options for hardware designers to select DNN architectures based on their accuracy and resource constraints. We also show that the FPGA implementation for LightNN requires fewer resources and reduces the latency compared to the conventional DNNs.

ACKNOWLEDGMENTS

This research was supported in part by NSF CCF Grant No. 1331804, NSF CCF Grant No. 1564022, and NSF CCF Grant No. 1314876.

REFERENCES

[1] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. 2016. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI'16)*. IEEE, 236–241.

[2] D. Dua and E. Karra Taniskidou. 2017. UCI Machine Learning Repository. University of California, School of Information and Computer Science. http://archive.ics.uci.edu/ml.

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition, 2009 (CVPR'09)*. IEEE, 248–255.

[5] Li Deng, Geoffrey Hinton, and Brian Kingsbury. 2013. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*. IEEE, 8599–8603.

[6] Ruizhou Ding, Zeye Liu, Rongye Shi, Diana Marculescu, and R. D. Blanton. 2017. LightNN: Filling the gap between conventional deep neural networks and binarized networks. In *Proceedings of the Great Lakes Symposium on VLSI 2017*. ACM, 35–40.

[7] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna Palem, Olivier Temam, and Chengyong Wu. 2014. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC'14)*. IEEE, 201–206.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. Retrieved from http://www.deeplearningbook.org.

[9] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*. 1737–1746.

[10] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*. 1135–1143.

[11] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2015. Djinn and tonic: DNN as a service and its implications for future warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 27–40.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[13] Itay Hubara. 2017. BinaryNet PyTorch code. Retrieved from https://github.com/itayhubara/BinaryNet.pytorch/blob/master/models/vgg_cifar10.py.

[14] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. BinaryNet Theano code. Retrieved from https://github.com/MatthieuCourbariaux/BinaryNet.

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems*. 4107–4115.

[16] Synopsys Inc. 2016. Synopsys Design Compiler. Retrieved from https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html.

[17] Xilinx Inc. 2017a. Xilinx FPGA. Retrieved from https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html.

[18] Xilinx Inc. 2017b. Xilinx Vivado. Retrieved from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[19] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*. 448–456.

[20] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe model examples. Retrieved from https://github.com/BVLC/caffe/tree/master/examples.

[21] Yongtae Kim, Yong Zhang, and Peng Li. 2013. An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 130–137.

[22] Diederik P. Kingma and Jimmy Lei Ba. 2015. A method for stochastic optimization. In *International Conference on Learning Representations (ICLR'15)*.

[23] Alex Krizhevsky and Geoffrey Hinton. 2012. Learning multiple layers of features from tiny images. University of Toronto.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324.

[27] Boxun Li, Yi Shan, Miao Hu, Yu Wang, Yiran Chen, and Huazhong Yang. 2013. Memristor-based approximated computation. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*. IEEE Press, 242–247.

[28] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. 2017. Training quantized nets: A deeper understanding. *arXiv Preprint arXiv:1706.02379* (2017).

[29] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of International Conference of Machine Learning (ICML'13)*, Vol. 30. 3.

[30] Synopsys. 2010. Synopsys MEDICI User's Manual. Synopsys, *Mountain View, CA*.

[31] Michele Marchesi, Gianni Orlandi, Francesco Piazza, and Aurelio Uncini. 1993. Fast neural networks without multipliers. *IEEE Transactions on Neural Networks* 4, 1 (1993), 53–62.

[32] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. 2017. Ternary neural networks with fine-grained quantization. *arXiv Preprint arXiv:1705.01462* (2017).

[33] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories*, 22–31.

[34] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, and Yu Wang. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 26–35.

[35] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[36] Syed Shakib Sarwar, Swagath Venkataramani, Anand Raghunathan, and Kaushik Roy. 2016. Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*. IEEE, 145–150.

[37] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[38] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv Preprint arXiv:1409.1556* (2014).

[39] Douglas R. Smith. 1999. Design ware: Software Development by Refinement. *Electronic Notes in Theoretical Computer Science* 29 (1999), 275–287.

[40] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[41] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). Retrieved from http://arxiv.org/abs/1605.02688.

[42] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2014. AxNN: Energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*. ACM, 27–32.

[43] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural acceleration for gpu throughput processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. IEEE, 482–493.

[44] Chen Zhang, Peng Li, Guanyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[45] Guoqiang Peter Zhang. 2000. Neural networks for classification: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 30, 4 (2000), 451–462.

[46] Jialiang Zhang and Jing Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 25–34.

[47] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. 2015. ApproxANN: An approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 701–706.

[48] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv Preprint arXiv:1606.06160* (2016).

Received December 2017; revised June 2018; accepted August 2018