

Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations



Itay Hubara*

*Department of Electrical Engineering
Technion - Israel Institute of Technology
Haifa, Israel*

ITAYH@CAMPUSE.TECHNION.AC.IL

Matthieu Courbariaux*

*Department of Computer Science and Department of Statistics
Université de Montréal
Montréal, Canada*

MATTHIEU.COURBARIAUX@GMAIL.COM

Daniel Soudry

*Department of Statistics
Columbia University
New York, USA*

DANIEL.SOUDRY@GMAIL.COM

Ran El-Yaniv

*Department of Computer Science
Technion - Israel Institute of Technology
Haifa, Israel*

RANI@CS.TECHNION.AC.IL

Yoshua Bengio

*Department of Computer Science and Department of Statistics
Université de Montréal
Montréal, Canada*

YOSHUA.UMONTREAL@GMAIL.COM

**Indicates first authors.*

Editor:

Abstract

We introduce a method to train Quantized Neural Networks (QNNs) — neural networks with extremely low precision (e.g., 1-bit) weights and activations, at run-time. At train-time the quantized weights and activations are used for computing the parameter gradients. During the forward pass, QNNs drastically reduce memory size and accesses, and replace most arithmetic operations with bit-wise operations. As a result, **power consumption** is expected to be drastically reduced. We trained QNNs over the MNIST, CIFAR-10, SVHN and ImageNet datasets. The resulting QNNs achieve prediction accuracy comparable to their 32-bit counterparts. For example, our quantized version of AlexNet with 1-bit weights and 2-bit activations achieves 51% top-1 accuracy. Moreover, we quantize the parameter gradients to 6-bits as well which enables gradients computation using only bit-wise operation. Quantized recurrent neural networks were tested over the Penn Treebank dataset, and achieved comparable accuracy as their 32-bit counterparts using only 4-bits. Last but not least, we programmed a binary matrix multiplication GPU kernel with which it is possible to run our MNIST QNN 7 times faster than with an **unoptimized** GPU kernel, without suffering any loss in classification accuracy. The QNN code is available online.

Keywords: Deep Learning, Neural Networks Compression, **Energy** Efficient Neural Networks, Computer vision, Language Models.

1. Introduction

Deep Neural Networks (DNNs) have substantially pushed Artificial Intelligence (AI) limits in a wide range of tasks, including but not limited to object recognition from images (Krizhevsky et al., 2012; Szegedy et al., 2014), speech recognition (Hinton et al., 2012; Sainath et al., 2013), statistical machine translation (Devlin et al., 2014; Sutskever et al., 2014; Bahdanau et al., 2015), Atari and Go games (Mnih et al., 2015; Silver et al., 2016), and even computer generation of abstract art (Mordvintsev et al., 2015).

Training or even just using neural network (NN) algorithms on conventional general-purpose digital hardware (Von Neumann architecture) has been found highly inefficient due to the massive amount of multiply-accumulate operations (MACs) required to compute the weighted sums of the neurons' inputs. Today, DNNs are almost exclusively trained on one or many very fast and power-hungry Graphic Processing Units (GPUs) (Coates et al., 2013). As a result, it is often a challenge to run DNNs on target low-power devices, and substantial research efforts are invested in speeding up DNNs at run-time on both general-purpose (Vanhoecke et al., 2011; Gong et al., 2014; Romero et al., 2014; Han et al., 2015b) and specialized computer hardware (Farabet et al., 2011a,b; Pham et al., 2012; Chen et al., 2014a,b; Esser et al., 2015).

The most common approach is to compress a trained (full precision) network. Hashed-Nets (Chen et al., 2015) reduce model sizes by using a hash function to randomly group connection weights and force them to share a single parameter value. Gong et al. (2014) compressed deep convnets using vector quantization, which resulted in only a 1% accuracy loss. However, both methods focused only on the fully connected layers. A recent work by Han and Dally (2015) successfully pruned several state-of-the-art large scale networks and showed that the number of parameters could be reduced by an order of magnitude.

Recent works have shown that more computationally efficient DNNs can be constructed by quantizing some of the parameters during the training phase. In most cases, DNNs are trained by minimizing some error function using Back-Propagation (BP) or related gradient descent methods. However, such an approach cannot be directly applied if the weights are restricted to binary values. Soudry et al. (2014) used a **variational** Bayesian approach with Mean-Field and Central Limit approximation to calculate the posterior distribution of the weights (the probability of each weight to be +1 or -1). During the inference stage (test phase), their method samples from this distribution one binary network and used it to predict the targets of the test set (More than one binary network can also be used). Courbariaux et al. (2015b) similarly used two sets of weights, real-valued and binary. They, however, updated the real valued version of the weights by using gradients computed by applying forward and backward propagation with the set of binary weights (which was obtained by quantizing the real-value weights to +1 and -1).

This study proposes a more advanced technique, referred to as Quantized Neural Network (QNN), for quantizing the neurons and weights during inference and training. In such networks, all MAC operations can be replaced with *XNOR* and *population count* (i.e., counting the number of ones in the binary number) operations. This is especially useful in

QNNs with the extremely low precision — for example, when only 1-bit is used per weight and activation, leading to a Binarized Neural Network (BNN). The proposed method is particularly beneficial for implementing large convolutional networks whose neuron-to-weight ratio is very large.

This paper makes the following contributions:

- We introduce a method to train Quantized-Neural-Networks (QNNs), neural networks with low precision weights and activations, at run-time, and when computing the parameter gradients at train-time. In the extreme case QNNs use only 1-bit per weight and activation (i.e., Binarized NN; see Section 2).
- We conduct two sets of experiments, each implemented on a different framework, namely Torch7 and Theano, which show that it is possible to train BNNs on MNIST, CIFAR-10 and SVHN and achieve near state-of-the-art results (see Section 4). Moreover, we report results on the challenging ImageNet dataset using binary weights/activations as well as quantized version of it (more than 1-bit).
- We present preliminary results on quantized gradients and show that it is possible to use only 6-bits with only small accuracy degradation.
- We present results for the Penn Treebank dataset using language models (vanilla RNNs and LSTMs) and show that with 4-bit weights and activations Recurrent QNNs achieve similar accuracies as their 32-bit floating point counterparts.
- We show that during the forward pass (both at run-time and train-time), QNNs drastically reduce memory consumption (size and number of accesses), and replace most arithmetic operations with bit-wise operations. A substantial increase in power efficiency is expected as a result (see Section 5). Moreover, a binarized CNN can lead to binary convolution kernel repetitions; we argue that dedicated hardware could reduce the time complexity by 60% .
- Last but not least, we programmed a binary matrix multiplication GPU kernel with which it is possible to run our MNIST BNN 7 times faster than with an unoptimized GPU kernel, without suffering any loss in classification accuracy (see Section 6).
- The code for training and applying our BNNs is available on-line (both the Theano ¹ and the Torch framework ²).

2. Binarized Neural Networks

In this section, we detail our binarization function, show how we use it to compute the parameter gradients, and how we backpropagate through it.

¹<https://github.com/MatthieuCourbariaux/BinaryNet>

²<https://github.com/itayhubara/BinaryNet>

2.1 Deterministic vs Stochastic Binarization

When training a BNN, we constrain both the weights and the activations to either $+1$ or -1 . Those two values are very advantageous from a hardware **perspective**, as we explain in Section 6. In order to transform the real-valued variables into those two values, we use two different binarization functions, as proposed by Courbariaux et al. (2015a). The first binarization function is deterministic:

$$x^b = \text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{otherwise,} \end{cases} \quad (1)$$

where x^b is the binarized variable (weight or activation) and x the real-valued variable. It is very straightforward to implement and works quite well in practice. The second binarization function is stochastic:

$$x^b = \begin{cases} +1 & \text{with probability } p = \sigma(x), \\ -1 & \text{with probability } 1 - p, \end{cases} \quad (2)$$

where σ is the “hard sigmoid” function:

$$\sigma(x) = \text{clip}\left(\frac{x+1}{2}, 0, 1\right) = \max(0, \min(1, \frac{x+1}{2})). \quad (3)$$

This stochastic binarization is more appealing theoretically (see Section 4) than the sign function, but somewhat harder to implement as it requires the hardware to generate random bits when quantizing (Torii et al., 2016). As a result, we mostly use the deterministic binarization function (i.e., the sign function), with the exception of activations at train-time in some of our experiments.

2.2 Gradient Computation and Accumulation

Although our BNN training method utilizes binary weights and activations to compute the parameter gradients, the real-valued gradients of the weights are accumulated in real-valued variables, as per Algorithm 1. Real-valued weights are likely required for Stochastic Gradient Descent (SGD) to work at all. SGD explores the space of parameters in small and noisy steps, and that noise is *averaged out* by the stochastic gradient contributions **accumulated** in each weight. Therefore, it is important to maintain sufficient resolution for these accumulators, which at first glance suggests that high precision is absolutely required.

Moreover, adding noise to weights and activations when *computing* the parameter gradients provide a form of regularization that can help to generalize better, as previously shown with variational weight noise (Graves, 2011), Dropout (Srivastava et al., 2014) and DropConnect (Wan et al., 2013). Our method of training BNNs can be seen as a variant of Dropout, in which instead of randomly setting half of the activations to zero when computing the parameter gradients, we binarize both the activations and the weights.

2.3 Propagating Gradients Through Discretization

The derivative of the sign function is zero almost everywhere, making it apparently incompatible with back-propagation, since the exact gradients of the cost with respect to the



quantities before the discretization (pre-activations or weights) are zero. Note that this limitation remains even if stochastic quantization is used. Bengio (2013) studied the question of estimating or propagating gradients through stochastic discrete neurons. He found that the fastest training was obtained when using the “straight-through estimator,” previously introduced in Hinton’s lectures (Hinton, 2012). We follow a similar approach but use the version of the straight-through estimator that takes into account the saturation effect, and does use deterministic rather than stochastic sampling of the bit. Consider the sign function quantization

$$q = \text{Sign}(r),$$

and assume that an estimator g_q of the gradient $\frac{\partial C}{\partial q}$ has been obtained (with the straight-through estimator when needed). Then, our straight-through estimator of $\frac{\partial C}{\partial r}$ is simply

$$g_r = g_q 1_{|r| \leq 1}. \quad (4)$$

Note that this preserves the gradient information and cancels the gradient when r is too large. Not cancelling the gradient when r is too large significantly worsens performance. To better understand why the straight-through estimator works well, consider the stochastic binarization scheme in Eq. (2) and rewrite $\sigma(r) = (\text{HT}(r) + 1)/2$, where HT(r) is the well-known “hard tanh”,

$$\text{HT}(r) = \begin{cases} +1 & r > 1, \\ r & r \in [-1, 1], \\ -1 & r < -1. \end{cases} \quad (5)$$

In this case the input to the next layer has the following form,

$$\mathbf{W}_b h_b(\mathbf{r}) = \mathbf{W}_b \text{HT}(\mathbf{r}) + \mathbf{n}(\mathbf{r}),$$

where we use the fact that $\text{HT}(\mathbf{r})$ is the expectation over $h_b(\mathbf{x})$ (see Eqs. (2) and (5)), and define $\mathbf{n}(\mathbf{r})$ as binarization noise with mean equal to zero. When the layer is wide, we expect the deterministic mean term $\text{HT}(\mathbf{x})$ to dominate, because the noise term $\mathbf{n}(\mathbf{r})$ is a summation over many independent binarizations from all the neurons in the previous layer. Thus, we argue that the binarization noise $\mathbf{n}(\mathbf{x})$ can be ignored when performing differentiation in the backward propagation stage. Therefore, we replace $\frac{\partial h_b(r)}{\partial r}$ (which cannot be computed) with

$$\frac{\partial \text{HT}(\mathbf{r})}{\partial x} = \begin{cases} 0 & r > 1, \\ 1 & r \in [-1, 1], \\ 0 & r < -1, \end{cases} \quad (6)$$

which is exactly the straight-through estimator defined in Eq (4). The use of this straight-through estimator is illustrated in Algorithm 1.

A similar binarization process was applied for weights in which we combine two ingredients:

- Project each real-valued weight to $[-1, 1]$, i.e., clip the weights during training, as per Algorithm 1. The real-valued weights would **otherwise** grow very large without any impact on the binary weights.

- When using a weight w^r , quantize it using $w^b = \text{Sign}(w^r)$.

Projecting the weights to $[-1,1]$ is consistent with the gradient cancelling when $|w^r| > 1$, according to Eq. (4).



2.4 Shift-based Batch Normalization

Batch Normalization (BN) (Ioffe and Szegedy, 2015) accelerates the training and reduces the overall impact of the weight scale (Courbariaux et al., 2015a). The normalization procedure may also help to regularize the model. However, at train-time, BN requires many multiplications (calculating the standard deviation and dividing by it, namely, dividing by the running variance, which is the weighted mean of the training set activation variance). Although the number of scaling calculations is the same as the number of neurons, in the case of ConvNets this number is quite large. For example, in the CIFAR-10 dataset (using our architecture), the first convolution layer, consisting of only $128 \times 3 \times 3$ filter masks, converts an image of size $3 \times 32 \times 32$ to size $128 \times 28 \times 28$, which is almost two orders of magnitude larger than the number of weights (87.1 to be exact). To achieve the results that BN would obtain, we use a shift-based batch normalization (SBN) technique, presented in Algorithm 2. SBN approximates BN almost without multiplications. Define $\text{AP2}(z)$ as the approximate power-of-2 of z (i.e., the index of the most significant bit (MSB)), and \lll as both left and right binary shift. SBN replaces almost all multiplication with power-of-2 approximation and shift operations:

$$x \times y \rightarrow x \lll \text{AP2}(y). \quad (7)$$

The only operation which is not a binary shift or an add is the inverse square root (see normalization operation Algorithm 2). From the early work of Lomont (2003) we know that the inverse-square operation could be applied with approximately the same complexity as multiplication. There are also faster methods, which involve lookup table tricks that typically obtain lower accuracy (this may not be an issue, since our procedure already adds a lot of noise). However, the number of values on which we apply the inverse-square operation is rather small, since it is done after calculating the variance, i.e., after averaging (for a more precise calculation, see the BN analysis in Lin et al. (2015b). Furthermore, the size of the standard deviation vectors is relatively small. For example, these values **make up** only 0.3% of the network size (i.e., the number of learnable parameters) in the Cifar-10 network we used in our experiments.

In the experiment we observed no loss in accuracy when using the shift-based BN algorithm instead of the **vanilla** BN algorithm.

2.5 Shift Based AdaMax

The ADAM learning method (Kingma and Ba, 2014b) also reduces the impact of the weight scale. Since ADAM requires many multiplications, we suggest using instead the shift-based AdaMax we outlined in Algorithm 3. In the experiment we conducted we observed no loss in accuracy when using the shift-based AdaMax algorithm instead of the vanilla ADAM algorithm.

³Hardware implementation of AP2 is as simple as extracting the index of the most significant bit from the number's binary representation.

Algorithm 1 Training a BNN. C is the cost function for minibatch, λ , the learning rate decay factor, and L , the number of layers. (\circ) stands for element-wise multiplication. The function $\text{Binarize}(\cdot)$ specifies how to (stochastically or deterministically) binarize the activations and weights, and $\text{Clip}(\cdot)$, how to clip the weights. $\text{BatchNorm}(\cdot)$ specifies how to batch-normalize the activations, using either batch normalization (Ioffe and Szegedy, 2015) or its shift-based variant we describe in Algorithm 2. $\text{BackBatchNorm}(\cdot)$ specifies how to backpropagate through the normalization. $\text{Update}(\cdot)$ specifies how to update the parameters when their gradients are known, using either ADAM (Kingma and Ba, 2014b) or the shift-based AdaMax we describe in Algorithm 3.

Require: a minibatch of inputs and targets (a_0, a^*) , previous weights W , previous BatchNorm parameters θ , weight initialization coefficients from (Glorot and Bengio, 2010) γ , and previous learning rate η .

Ensure: updated weights W^{t+1} , updated BatchNorm parameters θ^{t+1} and updated learning rate η^{t+1} .

```

{1. Computing the parameter gradients:}
{1.1. Forward propagation:}
for  $k = 1$  to  $L$  do
     $W_k^b \leftarrow \text{Binarize}(W_k)$ 
     $s_k \leftarrow a_{k-1}^b W_k^b$ 
     $a_k \leftarrow \text{BatchNorm}(s_k, \theta_k)$ 
    if  $k < L$  then
         $a_k^b \leftarrow \text{Binarize}(a_k)$ 
    end if
end for
{1.2. Backward propagation:}
{Note that the gradients are not binary.}
Compute  $g_{a_L} = \frac{\partial C}{\partial a_L}$  knowing  $a_L$  and  $a^*$ 
for  $k = L$  to  $1$  do
    if  $k < L$  then
         $g_{a_k} \leftarrow g_{a_k^b} \circ 1_{|a_k| \leq 1}$ 
    end if
     $(g_{s_k}, g_{\theta_k}) \leftarrow \text{BackBatchNorm}(g_{a_k}, s_k, \theta_k)$ 
     $g_{a_{k-1}^b} \leftarrow g_{s_k} W_k^b$ 
     $g_{W_k^b} \leftarrow g_{s_k}^\top a_{k-1}^b$ 
end for
{2. Accumulating the parameter gradients:}
for  $k = 1$  to  $L$  do
     $\theta_k^{t+1} \leftarrow \text{Update}(\theta_k, \eta, g_{\theta_k})$ 
     $W_k^{t+1} \leftarrow \text{Clip}(\text{Update}(W_k, \gamma_k \eta, g_{W_k^b}), -1, 1)$ 
     $\eta^{t+1} \leftarrow \lambda \eta$ 
end for

```

Algorithm 2 Shift based Batch Normalizing Transform, applied to activation x over a mini-batch. $\text{AP2}(x) = \text{sign}(x) \times 2^{\text{round}(\log_2|x|)}$ is the approximate power-of-2³, and $\ll\gg$ stands for **both** left and right binary shift.

Require: Values of x over a mini-batch: $B = \{x_{1\dots m}\}$; Parameters to be learned: γ, β

Ensure: $\{y_i = \text{BN}(x_i, \gamma, \beta)\}$

$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$ {mini-batch mean}
 $C(x_i) \leftarrow (x_i - \mu_B)$ {centered input}
 $\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (C(x_i) \ll\gg \text{AP2}(C(x_i)))$ {apx variance}
 $\hat{x}_i \leftarrow C(x_i) \ll\gg \text{AP2}((\sqrt{\sigma_B^2 + \epsilon})^{-1})$ {normalize}
 $y_i \leftarrow \text{AP2}(\gamma) \ll\gg \hat{x}_i$ {scale and shift}

Algorithm 3 Shift based AdaMax learning rule (Kingma and Ba, 2014b). g_t^2 indicates the element-wise square $g_t \circ g_t$. Good default settings are $\alpha = 2^{-10}$, $1 - \beta_1 = 2^{-3}$, $1 - \beta_2 = 2^{-10}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: Previous parameters θ_{t-1} , their gradient g_t , and learning rate α .

Ensure: Updated parameters θ_t

{Biased 1st and 2nd raw moment estimates:}
 $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
 $v_t \leftarrow \max(\beta_2 \cdot v_{t-1}, |g_t|)$
 {Updated parameters:}
 $\theta_t \leftarrow \theta_{t-1} - (\alpha \ll\gg (1 - \beta_1)) \cdot \hat{m} \ll\gg v_t^{-1}$

2.6 First Layer

In a BNN, only the binarized values of the weights and activations are used in all calculations. As the output of one layer is the input of the next, the inputs of all the layers are binary, with the exception of the first layer. However, we do not believe this to be a major issue. First, in computer vision, the input representation typically has far fewer channels (e.g, red, green and blue) than internal representations (e.g., 512). **Consequently, the first layer of a ConvNet is often the smallest convolution layer, both in terms of parameters and computations** (Szegedy et al., 2014). Second, it is relatively easy to handle continuous-valued inputs as fixed point numbers, with m bits of precision. For example, in the common case of 8-bit fixed point inputs:



$$s = x \cdot w^b, \quad s = \sum_{n=1}^8 2^{n-1} (x^n \cdot w^b), \quad (8)$$

where x is a vector of 1024 8-bit inputs, x_1^8 is the most significant bit of the first input, w^b is a vector of 1024 1-bit weights, and s is the resulting weighted sum. This method is used in Algorithm 4.

Algorithm 4 Running a BNN with L layers.

Require: 8-bit input vector a_0 , binary weights W^b , and BatchNorm parameters θ .

Ensure: the MLP output a_L .

{1. First layer:}
 $a_1 \leftarrow 0$
for $n = 1$ to 8 **do**
 $a_1 \leftarrow a_1 + 2^{n-1} \times \text{XnorDotProduct}(a_0^n, W_1^b)$
end for
 $a_1^b \leftarrow \text{Sign}(\text{BatchNorm}(a_1, \theta_1))$
 {2. Remaining hidden layers:}
for $k = 2$ to $L - 1$ **do**
 $a_k \leftarrow \text{XnorDotProduct}(a_{k-1}^b, W_k^b)$
 $a_k^b \leftarrow \text{Sign}(\text{BatchNorm}(a_k, \theta_k))$
end for
 {3. Output layer:}
 $a_L \leftarrow \text{XnorDotProduct}(a_{L-1}^b, W_L^b)$
 $a_L \leftarrow \text{BatchNorm}(a_L, \theta_L)$



3. Qunatized Neural network - More than 1-bit

Observing Eq. (8), we can see that using 2-bit activations simply doubles the number of times we need to run our XnorPopCount Kernel (i.e., directly proportional to the activation bitwidth). This idea was recently proposed by Zhou et al. (2016) (DoReFa net) and Miyashita et al. (2016) (published on arXive shortly after our preliminary technical report was published there). However, in contrast to Zhou et al., we did not find it useful to initialize the network with weights obtained by training the network with full precision weights. Moreover, the Zhou et al. network did not quantize the weights of the first convolutional layer and the last fully-connected layer, whereas we binarized both. We followed the quantization schemes suggested by Miyashita et al. (2016), namely, linear quantization:

$$\text{LinearQuant}(x, \text{bitwidth}) = \text{Clip} \left(\text{round} \left(\frac{x}{\text{bitwidth}} \right) \times \text{bitwidth}, \text{minV}, \text{maxV} \right) \quad (9)$$



and logarithmic quantization:

$$\text{LogQuant}(x, \text{bitwidth})(\mathbf{x}) = \text{Clip}(\text{AP2}(x), \text{minV}, \text{maxV}), \quad (10)$$

where minV and maxV are the minimum and maximum scale range respectively. Where $\text{AP2}(x)$ is the approximate-power-of-2 of x as described in Section 2.4. In our experiments (detailed in Section 4) we applied the above quantization schemes on the weights, activations and gradients and tested them on the more challenging ImageNet dataset.

4. Benchmark Results

4.1 Results on MNIST,SVHN, and CIFAR-10

We performed two sets of experiments, each based on a different framework, namely Torch7 and Theano. Other than the framework, the two sets of experiments are very similar:

Table 1: Classification test error rates of DNNs trained on MNIST (fully connected architecture), CIFAR-10 and SVHN (convnet). No unsupervised pre-training or data augmentation was used.

Data set	MNIST	SVHN	CIFAR-10
Binarized activations+weights, during training and test			
BNN (Torch7)	1.40%	2.53%	10.15%
BNN (Theano)	0.96%	2.80%	11.40%
Committee Machines' Array Baldassi et al. (2015)	1.35%	-	-
Binarized weights, during training and test			
BinaryConnect Courbariaux et al. (2015a)	$1.29 \pm 0.08\%$	2.30%	9.90%
Binarized activations+weights, during test			
EBP Cheng et al. (2015)	$2.2 \pm 0.1\%$	-	-
Bitwise DNNs Kim and Smaragdis (2016)	1.33%	-	-
Ternary weights, binary activations, during test			
Hwang and Sung (2014)	1.45%	-	-
No binarization (standard results)			
No reg	$1.3 \pm 0.2\%$	2.44%	10.94%
Maxout Networks Goodfellow et al. (2013b)	0.94%	2.47%	11.68%
Gated pooling Lee et al. (2015)	-	1.69%	7.62%

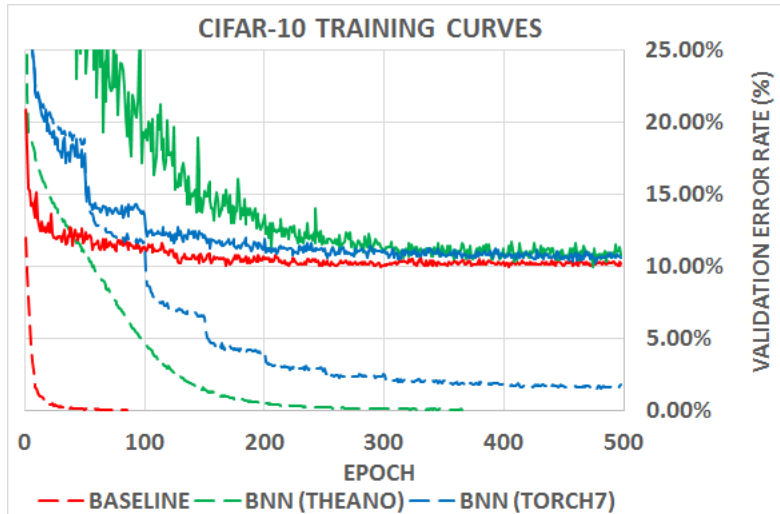
- In both sets of experiments, we obtain near state-of-the-art results with BNNs on MNIST, CIFAR-10 and the SVHN benchmark datasets.
- In our Torch7 experiments, the activations are *stochastically* binarized at train-time, whereas in our Theano experiments they are *deterministically* binarized.
- In our Torch7 experiments, we use the *shift-based BN and AdaMax* variants, which are detailed in Algorithms 2 and 3, whereas in our Theano experiments, we use *vanilla BN and ADAM*.

Results are reported in Table 1. Implementation details are reported in Appendix A.

MNIST MNIST is an image classification benchmark dataset (LeCun et al., 1998). It consists of a training set of 60K and a test set of 10K 28×28 gray-scale images representing digits ranging from 0 to 9. The Multi-Layer-Perceptron (MLP) we train on MNIST consists of 3 hidden layers. In our Theano implementation we used hidden layers of size 4096 whereas in our Torch implementation we used much smaller size 2048. This difference explains the accuracy gap between the two implementations.

CIFAR-10 CIFAR-10 is an image classification benchmark dataset. It consists of a training set of size 50K and a test set of size 10K, where instances are 32×32 color images representing airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships and trucks. Both implementations share the same structure as reported in Appendix A. Since the Torch implementation uses stochastic binarization, it achieved slightly better results.

Figure 1: Training curves for different methods on the CIFAR-10 dataset. The dotted lines represent the training costs (square hinge losses) and the continuous lines the corresponding validation error rates. Although BNNs are slower to train, they are nearly as accurate as 32-bit float DNNs.



SVHN Street View House Numbers (SVHN) is also an image classification benchmark dataset. It consists of a training set of size 604K examples and a test set of size 26K, where instances are 32×32 color images representing digits ranging from 0 to 9. Here again we obtained a small improvement in the performance by using stochastic binarization scheme.

4.2 Results on ImageNet

To test the strength of our method, we applied it to the challenging ImageNet classification task, which is probably the most important classification benchmark dataset. It consists of a training set of size 1.2M samples and test set of size 50K. Each instance is labeled with one of 1000 categories including objects, animals, scenes, and even some abstract shapes. On ImageNet, it is customary to report two error rates: top-1 and top-5, where the top- x error rate is the fraction of test images for which the correct label is not among the x labels considered most probable by the model. Considerable research has been concerned with compressing ImageNet architectures while preserving high accuracy. Previous approaches include pruning near zero weights (Gong et al., 2014; Han et al., 2015a) using matrix factorization techniques (Zhang et al., 2015), quantizing the weights (Gupta et al., 2015), using shared weights (Chen et al., 2015) and applying Huffman codes (Han et al., 2015a) among others.

To the best of our knowledge, before the first revision of this paper was published on arXive, no one had reported on successfully quantizing the network’s activations. On the contrary, a recent work (Han et al., 2015a) showed that accuracy significantly deteriorates when trying to quantize convolutional layers’ weights below 4-bit (FC layers are more robust to quantization and can operate quite well with only 2 bits). In the present work we

attempted to tackle the difficult task of binarizing both weights and activations. Employing the well-known AlexNet and GoogleNet architectures, we applied our techniques and achieved 41.8% top-1 and 67.1% top-5 accuracy using AlexNet and 47.1% top-1 and 69.1% top-5 accuracy using GoogleNet. While these performance results leave room for improvement (relative to full precision nets), they are by far better than all previous attempts to compress ImageNet architectures using less than 4-bit precision for the weights. Moreover, this advantage is achieved while also **binarizing** neuron activations.

4.3 Relaxing “hard tanh” boundaries

We discovered that after training the network it is useful to widen the “hard tanh” boundaries and retrain the network. As explained in Section 2.3, the straight-through estimator (which can be written as “hard tanh”) cancels gradients coming from neurons with absolute values higher than 1. Hence, towards the last training iterations most of the gradient values are zero and the weight values **cease** to update. By relaxing the “hard tanh” boundaries we allow more gradients to flow in the back-propagation phase and improve top-1 accuracies by 1.5% on AlexNet topology using vanilla BNN.

4.4 2-bit activations

While training BNNs on the ImageNet dataset we noticed that we could not force the training set error rate to converge to zero. In fact the training error rate stayed fairly close to the validation error rate. This observation led us to investigate a more relaxed activation quantization (more than 1-bit). As can be seen in Table 2, the results are quite impressive and illustrate an approximate 5.6% drop in performance (top-1 accuracy) relative to floating point representation, using only 1-bit weights and 2-bit activation. Following Miyashita et al. (2016), we also tried quantizing the gradients and discovered that only logarithmic quantization works. With 6-bit gradients we achieved 46.8% degradation. Those results are presently state-of-the-art, **surpassing** those obtained by the DoReFa net (Zhou et al., 2016). As opposed to DoReFa, we utilized a deterministic quantization process rather than a stochastic one. Moreover, it is important to note that while quantizing the gradients, DoReFa assigns for each instance in a mini-batch its own scaling factor, which increases the number of MAC operations.

While AlexNet can be compressed rather easily, compressing GoogleNet is much harder due to its small number of parameters. When using vanilla BNNs, we observed a large degradation in the top-1 results. However, by using QNNs with 4-bit weights and activation, we were able to achieve 66.5% top-1 accuracy (only a 5.5% drop in performance compared to the 32-bit floating point architecture), which is the current state-of-the-art-compression result over GoogleNet. Moreover, by using QNNs with 6-bit weights, activations and gradients we achieved 66.4% top-1 accuracy. Full implementation details of our experiments are reported in Appendix A.6.

4.5 Language Models

Recurrent neural networks (RNNs) are very demanding in memory and computational power in comparison to feed forward networks. There are a large variety of recurrent models with

Table 2: Classification test error rates of the AlexNet model trained on the ImageNet 1000 classification task. No unsupervised pre-training or data augmentation was used.

Model	Top-1	Top-5
Binarized activations+weights, during training and test		
BNN	41.8%	67.1%
Xnor-Nets ⁴ (Rastegari et al., 2016)	44.2%	69.2%
Binary weights and Quantize activations during training and test		
<u>QNN 2-bit activation</u>	51.03%	73.67%
DoReFaNet 2-bit activation ⁴ (Zhou et al., 2016)	50.7%	72.57%
Quantize weights, during test		
Deep Compression 4/2-bit (conv/FC layer) (Han et al., 2015a)	55.34%	77.67%
(Gysel et al., 2016) - 2-bit	0.01%	-%
No Quantization (standard results)		
AlexNet - our implementation	56.6%	80.2%

Table 3: Classification test error rates of the GoogleNet model trained on the ImageNet 1000 classification task. No unsupervised pre-training or data augmentation was used.

Model	Top-1	Top-5
Binarized activations+weights, during training and test		
BNN	47.1%	69.1%
<u>Quantize weights and activations during training and test</u>		
<u>QNN 4-bit</u>	<u>66.5%</u>	<u>83.4%</u>
Quantize activation,weights and gradients during training and test		
QNN 6-bit	66.4%	83.1%
No Quantization (standard results)		
GoogleNet - our implementation	71.6%	91.2%

the Long Short Term Memory networks (LSTMs) introduced by Hochreiter and Schmidhuber (1997) are being the most popular model. LSTMs are a special kind of RNN, capable of learning long-term dependencies using unique gating mechanisms. Recently, Ott et al. (2016) tried to quantize the RNNs weight matrices using similar techniques as described in Section 2. They observed that the weight binarization methods do not work with RNNs. However, by using 2-bits (i.e., $-1, 0, 1$), they have been able to achieve similar and even higher accuracy on several datasets. Here we report on the first attempt to quantize both weights and activations by trying to evaluate the accuracy of quantized recurrent models trained on the Penn Treebank dataset. The Penn Treebank Corpus (Marcus et al., 1993) contains 10K unique words. We followed the same setting as in (Mikolov and Zweig, 2012) which resulted in 18.55K words for training set, 14.5K and 16K words in the validation

and test sets respectively. We experimented with both vanilla RNNs and LSTMs. For our vanilla RNN model we used one hidden layers of size 2048 and ReLU as the activation function. For our LSTM model we use 1 hidden layer of size 300. Our RNN implementation was constructed to predict the next character hence performance was measured using the bits-per-character (BPC) metric. In the LSTM model we tried to predict the next word so performance was measured using the perplexity per word (PPW) metric. Similar to (Ott et al., 2016), our preliminary results indicate that binarization of weight matrices lead to large accuracy degradation. However, as can be seen in Table 4, with 4-bits activations and weights we can achieve similar accuracies as their 32-bit floating point counterparts.

Table 4: Language Models results on Penn Treebank dataset.

Language Models results on Penn Treebank dataset. FP stands for 32-bit floating point

Model	Layers	Hidden Units	bits(weights)	bits(activation)	Accuracy
RNN	1	2048	3	3	1.81 BPC
RNN	1	2048	2	4	1.67 BPC
RNN	1	2048	3	4	1.11 BPC
RNN	1	2048	3	4	1.05 BPC
RNN	1	2048	FP	FP	1.05 BPC
LSTM	1	300	2	3	220 PPW
LSTM	1	300	3	4	110 PPW
LSTM	1	300	4	4	100 PPW
LSTM	1	900	4	4	97 PPW
LSTM	1	300	FP	FP	97 PPW

5. High Power Efficiency during the Forward Pass

Table 5: Energy consumption of multiply- accumulations; see Horowitz (2014)

Operation	MUL	ADD
8-bit Integer	0.2pJ	0.03pJ
32-bit Integer	3.1pJ	0.1pJ
16-bit Floating Point	1.1pJ	0.4pJ
32-bit Floating Point	3.7pJ	0.9pJ

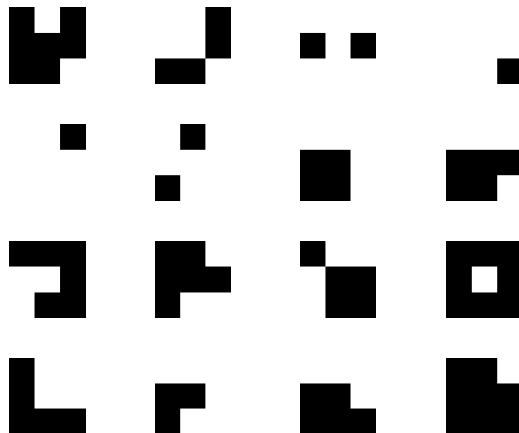
Table 6: Energy consumption of memory accesses; see Horowitz (2014)

Memory size	64-bit Cache
8K	10pJ
32K	20pJ
1M	100pJ
DRAM	1.3-2.6nJ

⁴ First and last layers were not binarized (i.e., using 32-bit precision weights and activation)

Computer hardware, be it general-purpose or specialized, is composed of memories, arithmetic operators and control logic. During the forward pass (both at run-time and train-time), BNNs drastically reduce memory size and accesses, and replace most arithmetic operations with bit-wise operations, which might lead to vastly improved power-efficiency. Moreover, a binarized CNN can lead to binary convolution kernel repetitions, and we argue that dedicated hardware could reduce the time complexity by 60% .


Figure 2: Binary weight filters, sampled from of the first convolution layer. Since we have only 2^{k^2} unique 2D filters (where k is the filter size), filter replication is very common. For instance, on our CIFAR-10 ConvNet, only 42% of the filters are **unique**.



Memory Size and Accesses Improving computing performance has always been and remains a challenge. Over the last decade, power has been the main constraint on performance (Horowitz, 2014). This is why considerable research efforts have been devoted to reducing the energy consumption of neural networks. Horowitz (2014) provides rough numbers for the energy consumed by the computation (the given numbers are for 45nm technology), as summarized in Tables 5 and 6. Importantly, we can see that memory accesses typically consume more energy than arithmetic operations, and *memory access cost increases with memory size*. In comparison with 32-bit DNNs, BNNs require 32 times smaller memory size *and* 32 times fewer memory accesses. This is expected to reduce energy consumption drastically (i.e., by a factor larger than 32).

XNOR-Count Applying a DNN mainly involves convolutions and matrix multiplications. The key arithmetic operation of deep learning is thus the multiply-accumulate operation. Artificial neurons are basically multiply-accumulators computing weighted sums of their inputs. In BNNs, both the activations and the weights are constrained to either -1 or $+1$. As a result, most of the 32-bit floating point multiply-accumulations are replaced

by 1-bit XNOR-count operations. This could have a big impact on dedicated deep learning hardware. For instance, a 32-bit floating point multiplier costs about 200 Xilinx FPGA slices (Govindu et al., 2004; Beauchamp et al., 2006), whereas a 1-bit XNOR gate only costs a single slice.



When using a ConvNet architecture with binary weights, the number of unique filters is bounded by the filter size. For example, in our implementation we use filters of size 3×3 , so the maximum number of unique 2D filters is $2^9 = 512$. However, this should not prevent expanding the number of feature maps beyond this number, since the actual filter is a 3D matrix. Assuming we have M_ℓ filters in the ℓ convolutional layer, we have to store a 4D weight matrix of size $M_\ell \times M_{\ell-1} \times k \times k$. Consequently, the number of unique filters is $2^{k^2 M_{\ell-1}}$. When necessary, we apply each filter on the map and perform the required multiply-accumulate (MAC) operations (in our case, using XNOR and popcount operations). Since we now have binary filters, many 2D filters of size $k \times k$ repeat themselves. By using dedicated hardware/software, we can apply only the unique 2D filters on each feature map and sum the results to receive each 3D filter’s convolutional result. Note that an inverse filter (i.e., $[-1, 1, -1]$ is the inverse of $[1, -1, 1]$) can also be treated as a repetition; it is merely a multiplication of the original filter by -1 . For example, in our ConvNet architecture trained on the CIFAR-10 benchmark, there are only 42% unique filters per layer on average. Hence we can reduce the number of the XNOR-popcount operations by 3.

QNNs complexity scale up linearly with the number of bits per weight/activation, since it requires the application of the XNOR kernel several times (see Section 3). As of now, QNNs still supply the best compression to accuracy ratio. Moreover, quantizing the gradients allows us to use the XNOR kernel for the backward pass, leading to fully fixed point layers with low bitwidth. By accelerating the training phase, QNNs can play an important role in future power demanding tasks.

6. Seven Times Faster on GPU at Run-Time

It is possible to speed up GPU implementations of QNNs, by using a method sometimes called SIMD (single instruction, multiple data) within a register (SWAR). The basic idea of SWAR is to *concatenate* groups of 32 binary variables into 32-bit registers, and thus obtain a 32-times speed-up on bitwise operations (e.g., XNOR). Using SWAR, it is possible to evaluate 32 connections with only 3 instructions:

$$a_1 + = \text{popcount}(\text{xnor}(a_0^{32b}, w_1^{32b})), \quad (11)$$

where a_1 is the resulting weighted sum, and a_0^{32b} and w_1^{32b} are the concatenated inputs and weights. Those 3 instructions (accumulation, popcount, xnor) take $1 + 4 + 1 = 6$ clock cycles on recent Nvidia GPUs (and if they were to become a fused instruction, it would only take a single clock cycle). Consequently, we obtain a theoretical Nvidia GPU speed-up of factor of $32/6 \approx 5.3$. In practice, this speed-up is quite easy to obtain as the memory bandwidth to computation ratio is also increased 6 times.

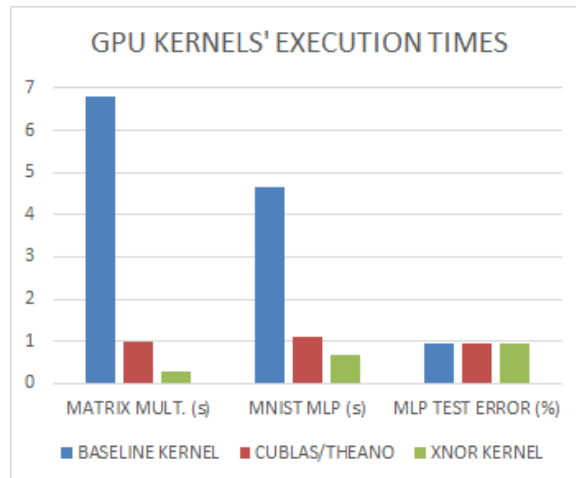
In order to validate those theoretical results, we programmed two GPU kernels:

- An unoptimized matrix multiplication kernel that serves as our baseline.

- The XNOR kernel, which is nearly identical to the baseline, except that it uses the SWAR method, as in Equation (11).

The two GPU kernels return identical outputs when their inputs are constrained to -1 or $+1$ (but not otherwise). The XNOR kernel is about *23 times faster than the baseline kernel* and *3.4 times faster than cuBLAS*, as shown in Figure 3. Last but not least, the MLP from Section 4 runs 7 times faster with the XNOR kernel than with the baseline kernel, without suffering any loss in classification accuracy (see Figure 3). As MNIST’s images are not binary, the first layer’s computations are always performed by the baseline kernel. The last three columns show that the MLP accuracy does not depend on which kernel is used.

Figure 3: The first 3 columns show the time it takes to perform a $8192 \times 8192 \times 8192$ (binary) matrix multiplication on a GTX750 Nvidia GPU, depending on which kernel is used. The next three columns show the time it takes to run the MLP from Section 3 on the full MNIST test set. The last three columns show that the MLP accuracy does not depend on the kernel



7. Discussion and Related Work

Until recently, the use of extremely low-precision networks (binary in the extreme case) was believed to substantially degrade the network performance (Courbariaux et al., 2014). Soudry et al. (2014) and Cheng et al. (2015) proved the contrary by showing that good performance could be achieved even if all neurons and weights are binarized to ± 1 . This was done using Expectation BackPropagation (EBP), a variational Bayesian approach, which infers networks with binary weights and neurons by updating the posterior distributions over the weights. These distributions are updated by differentiating their parameters (e.g., mean values) via the back propagation (BP) algorithm. Esser et al. (2015) implemented a fully binary network at run time using a very similar approach to EBP, showing significant

improvement in energy efficiency. The drawback of EBP is that the binarized parameters are only used during inference.

The probabilistic idea behind EBP was extended in the BinaryConnect algorithm of Courbariaux et al. (2015a). In BinaryConnect, the real-valued version of the weights is saved and used as a key reference for the binarization process. The binarization noise is independent between different weights, either by construction (by using stochastic quantization) or by assumption (a common simplification; see Spang and Schultheiss, 1962). The noise would have little effect on the next neuron’s input because the input is a summation over many weighted neurons. Thus, the real-valued version could be updated using the back propagated error by simply ignoring the binarization noise in the update. With this method, Courbariaux et al. (2015a) were the first to binarize weights in CNNs and achieved near state-of-the-art performance on several datasets. They also argued that noisy weights provide a form of regularization, which could help to improve generalization, as previously shown by Wan et al. (2013). This method binarized weights while still maintaining full precision neurons.

Lin et al. (2015a) carried over the work of Courbariaux et al. (2015a) to the back-propagation process by quantizing the representations at each layer of the network, to convert some of the remaining multiplications into binary shifts by restricting the neurons’ values to be power-of-two integers. Lin et al. (2015a)’s work and ours seem to share similar characteristics. However, their approach continues to use full precision weights during the test phase. Moreover, Lin et al. (2015a) quantize the neurons only during the back propagation process, and not during forward propagation.

Other research (Baldassi et al., 2015) showed that full binary training and testing is possible in an array of committee machines with randomized input, where only one weight layer is being adjusted. Gong et al. (2014) aimed to compress a fully trained high precision network by using quantization or matrix factorization methods. These methods required training the network with full precision weights and neurons, thus requiring numerous MAC operations (which the proposed QNN algorithm avoids). Hwang and Sung (2014) focused on a fixed-point neural network design and achieved performance almost identical to that of the floating-point architecture. Kim and Smaragdis (2016) *retrained* neural networks with binary weights and activations.

As far as we know, before the first revision of this paper was published on arXiv, no work succeeded in binarizing weights *and* neurons, at the inference phase *and* the entire training phase of a deep network. This was achieved in the present work. We relied on the idea that binarization can be done stochastically, or be approximated as random noise. This was previously done for the weights by Courbariaux et al. (2015a), but our BNNs extend this to the activations. Note that the binary activations are especially important for ConvNets, where there are typically many more neurons than free weights. This allows highly efficient operation of the binarized DNN at run time, and at the forward-propagation phase during training. Moreover, our training method has almost no multiplications, and therefore might be implemented efficiently in dedicated hardware. However, we have to save the value of the full precision weights. This is a remaining computational bottleneck during training, since it is an energy-consuming operation.

Shortly after the first version of this paper was posted on arXiv, several papers tried to improve and extend it. Rastegari et al. (2016) made a small modification to our algo-

rithm (namely multiplying the binary weights and input by their L_1 norm) and published promising results on the ImageNet dataset. Note that their method, named Xnor-Net, requires additional multiplication by a different scaling factor for each patch in each sample (Rastegari et al., 2016) Section 3.2 Eq. 10 and figure 2). This in itself, requires many multiplications and prevents efficient implementation of XnorNet on known hardware designs. Moreover, (Rastegari et al., 2016) didn't quantize first and last layers, therefore XNOR-Net are only partially binarized NNs. Miyashita et al. (2016) suggested a more relaxed quantization (more than 1-bit) for both the weights and activation. Their idea was to quantize both and use shift operations as in our Eq. (4). They proposed to quantize the parameters in their non-uniform, base-2 logarithmic representation. This idea was inspired by the fact that the weights and activations in a trained network naturally have non-uniform distributions. They moreover showed that they can quantize the gradients as well to 6-bit without significant losses in performance (on the Cifar-10 dataset). Zhou et al. (2016) applied similar ideas to the ImageNet dataset and showed that by using 1-bit weights, 2-bit activations and 6-bit gradients they can achieve 46.1% top-1 accuracies using the AlexNet architecture. They named this method DoReFa net. Here we outperform DoReFa net and achieve 46.8% using a 1-2-6 bit quantization scheme (weight-activation-gradients) and 51% using a 1-2-32 quantization scheme. These results confirm that we can achieve comparable results even on a large dataset by applying the Xnor kernel several times. Merolla et al. (2016) showed that DNN can be robust to more than just weight binarization. They applied several different distortions to the weights, including additive and multiplicative noise, and a class of non-linear projections. This was shown to improve robustness to other distortions and even boost results. Zheng and Tang tried to apply our binarization scheme to recurrent neural network for language modeling and achieved comparable results as well. Andri et al. (2016) even created a hardware implementation to speed up BNNs.

Conclusion

We have introduced BNNs, which binarize deep neural networks and can lead to dramatic improvements in both power consumption and computation speed. During the forward pass (both at run-time and train-time), BNNs drastically reduce memory size and accesses, and replace most arithmetic operations with bit-wise operations. Our estimates indicate that power efficiency can be improved by more than one order of magnitude (see Section 5). In terms of speed, we programmed a binary matrix multiplication GPU kernel that enabled running MLP over the MNIST dataset 7 times faster (than with an unoptimized GPU kernel) without any loss of accuracy (see Section 6).

We have shown that BNNs can handle MNIST, CIFAR-10 and SVHN while achieving nearly state-of-the-art accuracy. While our results for the challenging ImageNet are not on par with the best results achievable with full precision networks, they significantly improve all previous attempts to compress ImageNet-capable architectures. Moreover, by quantizing the weights and activations to more than 1-bit (i.e., QNNs), we have been able to achieve comparable results to the 32-bit floating point architectures (see Section 4.4 and supplementary material - Appendix B). A major open research avenue would be to further improve our results on ImageNet. Substantial progress in this direction might go a long way towards facilitating DNN usability in low power instruments such as mobile phones.

Acknowledgments

We would like to express our appreciation to Elad Hoffer, for his technical assistance and constructive comments. We thank our fellow MILA lab members who took the time to read the article and give us some feedback. We thank the developers of Torch, (Collobert et al., 2011) a Lua based environment, and Theano (Bergstra et al., 2010; Bastien et al., 2012), a Python library that allowed us to easily develop fast and optimized code for GPU. We also thank the developers of Pylearn2 (Goodfellow et al., 2013a) and Lasagne (Dieleman et al., 2015), two deep learning libraries built on the top of Theano. We thank Yuxin Wu for helping us compare our GPU kernels with cuBLAS. We are also grateful for funding from NSERC, the Canada Research Chairs, Compute Canada, and CIFAR. We are also grateful for funding from CIFAR, NSERC, IBM, Samsung. This research was supported by The Israel Science Foundation (grant No. 1890/14)

Appendix A. Implementation Details

In this section we give full implementation details over our MNIST,SVHN, CIFAR-10 and ImageNet datasets.

A.1 MLP on MNIST (Theano)

MNIST is an image classification benchmark dataset (LeCun et al., 1998). It consists of a training set of 60K and a test set of 10K 28×28 gray-scale images representing digits ranging from 0 to 9. In order for this benchmark to remain a challenge, we did not use any convolution, data-augmentation, preprocessing or unsupervised learning. The Multi-Layer-Perceptron (MLP) we train on MNIST consists of 3 hidden layers of 4096 binary units and a L2-SVM output layer; L2-SVM has been shown to perform better than Softmax on several classification benchmarks (Tang, 2013; Lee et al., 2014). We regularize the model with Dropout (Srivastava et al., 2014). The square hinge loss is minimized with the ADAM adaptive learning rate method (Kingma and Ba, 2014b). We use an exponentially decaying global learning rate, as per Algorithm 1, and also scale the learning rates of the weights with their initialization coefficients from (Glorot and Bengio, 2010), as suggested by Courbariaux et al. (2015a). We use Batch Normalization with a minibatch of size 100 to speed up the training. As is typical, we use the last 10K samples of the training set as a validation set for early stopping and model selection. We report the test error rate associated with the best validation error rate after 1000 epochs (we do not retrain on the validation set).

A.2 MLP on MNIST (Torch7)

We use a similar architecture as in our Theano experiments, without dropout, and with 2048 binary units per layer instead of 4096. Additionally, we use the shift base AdaMax and BN (with a minibatch of size 100) instead of the vanilla implementations, to reduce the number of multiplications. Likewise, we decay the learning rate by using a 1-bit right shift every 10 epochs.

A.3 ConvNet on CIFAR-10 (Theano)

CIFAR-10 is an image classification benchmark dataset. It consists of a training set of size 50K and a test set of size 10K, where instances are 32×32 color images representing airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships and trucks. We do not use data-augmentation (which can really be a game changer for this dataset; see Graham 2014). The architecture of our ConvNet is identical to that used by Courbariaux et al. (2015b) except for the binarization of the activations. The Courbariaux et al. (2015a) architecture is itself mainly inspired by VGG (Simonyan and Zisserman, 2015). The square hinge loss is minimized with ADAM. We use an exponentially decaying learning rate, as we did for MNIST. We scale the learning rates of the weights with their initialization coefficients from (Glorot and Bengio, 2010). We use Batch Normalization with a minibatch of size 50 to speed up the training. We use the last 5000 samples of the training set as a validation set. We report the test error rate associated with the best validation error rate after 500 training epochs (we do not retrain on the validation set).

Table 7: Architecture of our CIFAR-10 ConvNet. We only use "same" convolutions as in VGG (Simonyan and Zisserman, 2015).

CIFAR-10 ConvNet architecture
Input: 32×32 - RGB image
3×3 - 128 convolution layer
BatchNorm and Binarization layers
3×3 - 128 convolution and 2×2 max-pooling layers
BatchNorm and Binarization layers
3×3 - 256 convolution layer
BatchNorm and Binarization layers
3×3 - 256 convolution and 2×2 max-pooling layers
BatchNorm and Binarization layers
3×3 - 512 convolution layer
BatchNorm and Binarization layers
3×3 - 512 convolution and 2×2 max-pooling layers
BatchNorm and Binarization layers
1024 fully connected layer
BatchNorm and Binarization layers
1024 fully connected layer
BatchNorm and Binarization layers
10 fully connected layer
BatchNorm layer (no binarization)
Cost: Mean square hinge loss

A.4 ConvNet on CIFAR-10 (Torch7)

We use the same architecture as in our Theano experiments. We apply shift-based AdaMax and BN (with a minibatch of size 200) instead of the vanilla implementations to reduce the number of multiplications. Likewise, we decay the learning rate by using a 1-bit right shift every 50 epochs.

A.5 ConvNet on SVHN

SVHN is also an image classification benchmark dataset. It consists of a training set of size 604K examples and a test set of size 26K, where instances are 32×32 color images representing digits ranging from 0 to 9. In both sets of experiments, we follow the same procedure used for the CIFAR-10 experiments, with a few notable exceptions: we use half the number of units in the convolution layers, and we train for 200 epochs instead of 500 (because SVHN is a much larger dataset than CIFAR-10).

A.6 ConvNet on ImageNet

ImageNet classification task consists of a training set of size 1.2M samples and test set of size 50K. Each instance is labeled with one of 1000 categories including objects, animals, scenes, and even some abstract shapes.

AlexNet: Our AlexNet implementation consists of 5 convolution layers followed by 3 fully connected layers (see Section 8). Additionally, we use Adam as our optimization method and batch-normalization layers (with a minibatch of size 512). Likewise, we decay the learning rate by 0.1 every 20 epochs.

GoogleNet: Our GoogleNet implementation consist of 2 convolution layers followed by 10 inception layers, spatial-average-pooling and a fully connected classifier. We also used the 2 auxilary classifiers. Additionally, we use Adam (Kingma and Ba, 2014a) as our optimization method and batch-normalization layers (with a minibatch of size 64). Likewise, we decay the learning rate by 0.1 every 10 epochs.

Table 8: Our AlexNet Architecture.



AlexNet ConvNet architecture
Input: <u>32 × 32 - RGB image</u>
<u>11 × 11 - 64 convolution layer and 3 × 3 max-pooling layers</u>
BatchNorm and Binarization layers
5 × 5 - 192 convolution layer and 3 × 3 max-pooling layers
BatchNorm and Binarization layers
3 × 3 - 384 convolution layer
BatchNorm and Binarization layers
3 × 3 - 256 convolution layer
BatchNorm and Binarization layers
3 × 3 - 256 convolution layer
BatchNorm and Binarization layers
4096 fully connected layer
BatchNorm and Binarization layers
4096 fully connected layer
BatchNorm and Binarization layers
1000 fully connected layer
BatchNorm layer (no binarization)
SoftMax layer (no binarization)
Cost: <u>Negative log likelihood</u>

References

- Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. *arXiv preprint arXiv:1606.05487*, 2016.
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR'2015*, *arXiv:1409.0473*, 2015.
- Carlo Baldassi, Alessandro Ingrosso, Carlo Lucibello, Luca Saglietti, and Riccardo Zecchina. Subdominant Dense Clusters Allow for Simple Learning and High Computational Performance in Neural Networks with Discrete Synapses. *Physical Review Letters*, 115(12): 1–5, 2015. ISSN 10797114. doi: 10.1103/PhysRevLett.115.128101.
- Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and

- speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.
- Michael J Beauchamp, Scott Hauck, Keith D Underwood, and K Scott Hemmert. Embedded floating-point units in FPGAs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 12–20. ACM, 2006.
- Yoshua Bengio. Estimating or propagating gradients through stochastic neurons. Technical Report arXiv:1305.2982, Universite de Montreal, 2013.
- James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.
- Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM, 2014a.
- Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. *arXiv preprint arXiv:1504.04788*, 2015.
- Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 609–622. IEEE, 2014b.
- Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhenzhong Lan. Training binary multilayer neural networks for image classification using expectation backpropagation. *arXiv preprint arXiv:1503.03562*, 2015.
- Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with COTS HPC systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 2013.
- Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.
- Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *ArXiv e-prints*, abs/1412.7024, December 2014.
- Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *ArXiv e-prints*, abs/1511.00363, November 2015a.
- Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *Nips*, pages 1–9, 2015b. URL <http://arxiv.org/abs/1511.00363>.

- Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *Proc. ACL'2014*, 2014.
- Sander Dieleman, Jan Schlter, Colin Raffel, Eben Olson, Sren Kaae Snderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, diogo149, Brian McFee, Hendrik Weideman, takacsg84, peterderivaz, Jon, instagibbs, Dr. Kashif Rasul, CongLiu, Britefury, and Jonas Degraive. Lasagne: First release., August 2015. URL <http://dx.doi.org/10.5281/zenodo.27878>.
- Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems*, pages 1117–1125, 2015.
- Clément Farabet, Yann LeCun, Koray Kavukcuoglu, Eugenio Culurciello, Berin Martini, Polina Akselrod, and Selcuk Talay. Large-scale FPGA-based convolutional networks. *Machine Learning on Very Large Data Sets*, 1, 2011a.
- Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neufow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE, 2011b.
- Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS'2010*, 2010.
- Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.
- Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013a.
- Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout Networks. *arXiv preprint*, pages 1319–1327, 2013b. URL <http://arxiv.org/abs/1302.4389>.
- Gokul Govindu, Ling Zhuo, Seonil Choi, and Viktor Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 149. IEEE, 2004.
- Benjamin Graham. Spatially-sparse convolutional neural networks. *arXiv preprint arXiv:1409.6070*, 2014.
- Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356, 2011.
- Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 392, 2015.

- Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*, 2016.
- Huizi Mao Han, Song and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint*, pages 1–11, 2015. URL <http://arxiv.org/abs/1510.00149>.
- Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.
- Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015b.
- Geoffrey Hinton. Neural networks for machine learning. Coursera, video lectures, 2012.
- Geoffrey Hinton, Li Deng, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov. 2012.
- Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- Mark Horowitz. Computing’s Energy Problem (and what we can do about it). *IEEE Interational Solid State Circuits Conference*, pages 10–14, 2014. ISSN 0018-9200. doi: 10.1109/JSSC.2014.2361354.
- Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE, 2014.
- Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.
- M. Kim and P. Smaragdis. Bitwise Neural Networks. *ArXiv e-prints*, January 2016.
- Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, pages 1–13, 2014a. URL <http://arxiv.org/abs/1412.6980>~~delimiter"026E30F\$nh~~<http://www.arxiv.org/pdf/1412.6980.pdf>.
- Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014b.
- A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS’2012*. 2012.
- Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

- Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*, 2014.
- Chen-Yu Lee, Patrick W Gallagher, and Zhuowen Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. *arXiv preprint arXiv:1509.08985*, 2015.
- Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *ArXiv e-prints*, abs/1510.03009, October 2015a.
- Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural Networks with Few Multiplications. *Iclr*, pages 1–8, 2015b. URL <http://arxiv.org/abs/1510.03009>.
- Chris Lomont. Fast inverse square root. *Tech-315 nical Report*, page 32, 2003.
- Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.
- Paul Merolla, Rathinakumar Appuswamy, John Arthur, Steve K Esser, and Dharmendra Modha. Deep neural networks are robust to weight binarization and other non-linear distortions. *arXiv preprint arXiv:1606.01981*, 2016.
- Tomas Mikolov and Geoffrey Zweig. Context dependent recurrent neural network language model. In *SLT*, pages 234–239, 2012.
- Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fiedel, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharsan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.
- Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks, 2015. URL <http://googleresearch.blogspot.co.uk/2015/06/inceptionism-going-deeper-into-neural.html>. Accessed: 2015-06-30.
- Joachim Ott, Zhouhan Lin, Ying Zhang, Shih-Chii Liu, and Yoshua Bengio. Recurrent neural networks with limited numerical precision. *arXiv preprint arXiv:1608.06902*, 2016.
- Phi-Hung Pham, Darko Jelaca, Clement Farabet, Berin Martini, Yann LeCun, and Eugenio Culurciello. Neuflo: Dataflow vision processing system-on-a-chip. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 1044–1047. IEEE, 2012.
- Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016.

- Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.
- Tara Sainath, Abdel rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep convolutional neural networks for LVCSR. In *ICASSP 2013*, 2013.
- 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. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. ISSN 0028-0836. URL <http://dx.doi.org/10.1038/nature16961>. Article.
- Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.
- Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NIPS'2014*, 2014.
- H Spang and P Schultheiss. Reduction of quantizing noise by use of feedback. *IRE Transactions on Communications Systems*, 10(4):373–380, 1962.
- Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL <http://jmlr.org/papers/v15/srivastava14a.html>.
- Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS'2014*, 2014.
- Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. Technical report, arXiv:1409.4842, 2014.
- Yichuan Tang. Deep learning using linear support vector machines. Workshop on Challenges in Representation Learning, ICML, 2013.
- Naoya Torii, Hirotaka Kokubo, Dai Yamamoto, Kouichi Itoh, Masahiko Takenaka, and Tsutomu Matsumoto. Asic implementation of random number generators using sr latches and its evaluation. *EURASIP Journal on Information Security*, 2016(1):1–12, 2016.
- Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.
- Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect. In *ICML'2013*, 2013.

Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and accurate approximations of nonlinear convolutional networks. pages 1984–1992, 2015.

Weiyi Zheng and Yina Tang. Binarized neural networks for language modeling.

Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.