From: 林苏昕 1340172084@qq.com
Subject:
Date: October 19, 2018 at 9:14 PM
To:

林

2 Binarized Neural Network Training Neural Networks with Weights and Activations Constrained to -1 or 1

Saturday, October 13, 2018
2:18 PM

<<1602.02830.pdf>>

arXiv:1602.02830v3 [cs.LG] 17 Mar 2016

Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to $+1$ or -1

Matthieu Courbariaux[*1] MATTHIEU.COURBARIAUX@GMAIL.COM
Itay Hubara[*2] ITAYHUBARA@GMAIL.COM
Daniel Soudry[3] DANIEL.SOUDRY@GMAIL.COM
Ran El-Yaniv[2] RANI@CS.TECHNION.AC.IL
Yoshua Bengio[1,4] YOSHUA.UMONTREAL@GMAIL.COM
[1]Université de Montréal
[2]Technion - Israel Institute of Technology
[3]Columbia University
[4]CIFAR Senior Fellow
*Indicates equal contribution. Ordering determined by coin flip.

Abstract

We introduce a method to train Binarized Neural Networks (BNNs) - neural networks with binary weights and activations at run-time. At training-time the binary weights and activations are used for computing the parameters gradients. During the forward pass, BNNs drastically reduce memory size and accesses, and replace most arithmetic operations with bit-wise operations, which is expected to substantially improve power-efficiency. To validate the effectiveness of BNNs we conduct two sets of experiments on the Torch7 and Theano frameworks. On both, BNNs achieved nearly state-of-the-art results over the MNIST, CIFAR-10 and SVHN datasets. Last but not least, we wrote 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. The code for training and running our BNNs is available on-line.

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 abstract art (Mordvintsev et al., 2015).

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 (Vanhoucke et al., 2011; Gong et al., 2014; Romero et al., 2014; Han et al., 2015) and specialized computer hardware (Farabet et al., 2011a;b; Pham et al., 2012; Chen et al., 2014a;b; Esser et al., 2015).

This paper makes the following contributions:

- We introduce a method to train Binarized-Neural-Networks (BNNs), neural networks with binary weights and activations, at run-time, and when computing the parameters gradients at train-time (see Section 1).
- We conduct two sets of experiments, each implemented on a different framework, namely Torch7 (Collobert et al., 2011) and Theano (Bergstra et al., 2010; Bastien et al., 2012), which show that it is possible to train BNNs on MNIST, CIFAR-10 and SVHN and achieve nearly state-of-the-art results (see Section 2).
- We show that during the forward pass (both at run-time and train-time), BNNs drastically reduce memory consumption (size and number of accesses), and

Introduces a method to train BNN at run time

Contribution: experiments

Memory less, time complexity down

Program GPU kernel

1 Binarized Neural Network

(1)Binarization :

both weight and acti
Deterministic:
X = sign(X)

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:

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

(2)Gradient Computation and Accumulation:

Compute parameter gradients by binary weights and activation — Diff from pa
The real_value gradient are accumulate in real_valued variable (need sufficient resolution)
SGD explores the space of parameter in small and noisy steps
Adding noise to the weight and activ form regularization to generalize,
Training method is variant of Dropout,but binarize a and w instead of random set half as 0 — Diff from D

replace most arithmetic operations with bit-wise operations, which potentially lead to a substantial increase in power-efficiency (see Section 3). 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 programed 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 4).
- The code for training and running our BNNs is available on-line (In both Theano framework [1] and Torch framework [2]).

1. Binarized Neural Networks

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

1.1. 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 4. In order to transform the real-valued variables into those two values, we use two different binarization functions, as in (Courbariaux et al., 2015). Our 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. Our second binarization function is stochastic: 进行概率随机赋值

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

where σ is the *"hard sigmoid"* function:

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

The stochastic binarization is more appealing than the sign function, but harder to implement as it requires the hardware to generate random bits when quantizing. 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.

1.2. Gradient Computation and Accumulation

Not binary

Although our BNN training method uses binary weights and activation to compute the parameters 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 Stochasic 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 keep 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 parameters gradients provide a form of regularization that can help to generalize better, as previously shown with variational weight noise (Graves, 2011), Dropout (Srivastava, 2013; 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 parameters gradients, we binarize both the activations and the weights.

Diff. Dropout

1.3. Propagating Gradients Through Discretization

The derivative of the sign function is zero almost everywhere, making it apparently incompatible with backpropagation, since the exact gradient of the cost with respect to the quantities before the discretization (pre-activations or weights) would be zero. Note that this remains true even if stochastic quantization is used. Bengio (2013) studied the question of estimating or propagating gradients through stochastic discrete neurons. They found in their experiments that the fastest training was obtained when using the "straight-through estimator," previously introduced in Hinton (2012)'s lectures.

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

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

clip

Note that this preserves the gradient's information and cancels the gradient when r is too large. Not cancelling the gradient when r is too large significantly worsens the performance.

[1]https://github.com/MatthieuCourbariaux/

(3)propagating gradients through Discretization

Gradient of cost before the Discretization will be 0,
Obtain straight-through estimater+saturation effct
Clip(),preserve the gradient information from biarization
And conceal the gradient when it is too large
Sign(a)for hidden unis activations
Clip() the weight during training
Sign(W)

(4)Shift based Batch normalization

BN Accelerates training
Reduce the overall impact
Regulize the model

BN : Train time ,require many multiplication
(caculate the standard deviation and dividing by it),divided by the running variance.(the weighted mean of the training set activation)
Shift based BN :
Almost without multiplication

(5) Shift based Adamax

Old way Require mul

(6)first layer

All layer binary except the first layer
Ok :first conv parameter and computation less
It is easy to handle fix point number as

BinaryNet

[2]https://github.com/itayhubara/BinaryNet

formance. The use of this straight-through estimator is illustrated in Algorithm 1. The derivative $1_{|r|\le 1}$ can also be

algorithm 5

3 power in fofward

(1) Xnor count:

32 bit mul -> 1 bit Xnor count

(2) Filter Repetitons

Only 42 unique

Algorithm 1 Training a BNN. C is the cost function for minibatch, λ - the learning rate decay factor and L the number of layers. $\circ$ indicates element-wise multiplication. The function Binarize() specifies how to (stochastically or deterministically) binarize the activations and weights, and Clip(), how to clip the weights. BatchNorm() specifies how to batch-normalize the activations, using either batch normalization (Ioffe & Szegedy, 2015) or its shift-based variant we describe in Algorithm 3. BackBatchNorm() specifies how to backpropagate through the normalization. Update() specifies how to update the parameters when their gradients are known, using either ADAM (Kingma & Ba, 2014) or the shift-based AdaMax we describe in Algorithm 4.

Require: a minibatch of inputs and targets (a_0, a^*), previous weights W, previous BatchNorm parameters θ, weights initialization coefficients from (Glorot & 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 parameters 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:}
{Please 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**
    $g_{a_k} \leftarrow g_{a_k^b} \circ 1_{|a_k|\le 1}$
  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 parameters 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. $AP2(x) = \text{sign}(x) \times 2^{\text{round}(\log 2|x|)}$ is the approximate power-of-2 [3], and $\ll\gg$ stands for **both** left and right binary shift.

Require: Values of x over a mini-batch: $B = \{x_{1\ldots m}\}$; 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}
  $\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^m (C(x_i) \ll\gg AP2(C(x_i)))$ {apx variance}
  $\hat{x}_i \leftarrow C(x_i) \ll\gg AP2((\sqrt{\sigma_B^2 + \epsilon})^{-1})$ {normalize}
  $y_i \leftarrow AP2(\gamma) \ll\gg \hat{x}_i$ {scale and shift}

Algorithm 3 Shift based Batch Normalizing Transform, applied to activation (x) over a mini-batch. Where AP2 is the approximate power-of-2 and $\ll\gg$ stands for **both** left and right binary shift.

Require: Values of x over a mini-batch: $B = \{x_{1\ldots m}\}$; 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}
  $\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^m (C(x_i) \ll\gg AP2(C(x_i)))$ {apx variance}
  $\hat{x}_i \leftarrow C(x_i) \ll\gg AP2((\sqrt{\sigma_B^2 + \epsilon})^{-1})$ {normalize}
  $y_i \leftarrow AP2(\gamma) \ll\gg \hat{x}_i$ {scale and shift}

seen as propagating the gradient through *hard tanh*, which is the following piece-wise linear activation function:

$$\text{Htanh}(x) = \text{Clip}(x, -1, 1) = \max(-1, \min(1, x)). \quad (5)$$

For hidden units, we use the sign function non-linearity to obtain binary activations, and for weights we combine two ingredients:

- Constrain each real-valued weight between -1 and 1, by projecting w^r to -1 or 1 when the weight update brings w^r outside of $[-1, 1]$, i.e., clipping 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 $w^b = \text{Sign}(w^r)$.

This is consistent with the gradient canceling when $|w^r| > 1$, according to Eq. 4.

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

- Question list:
  (3) Dropout
  (4) Discretization
  (5) estimater of gradient
  (6) saturation effect
  (6)weight scale
  (7) run time
  (8)shift based batch normalization transform
  (9) EBP in paper[1] only binary in inference

Answer sheet：

(1)

(2) clip(x) = min(1(max(x,-1))

(3) dropout

Algorithm 4 Shift based AdaMax learning rule (Kingma & Ba, 2014). 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} and 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:}
  $\theta_t \leftarrow \theta_{t-1} - (\alpha \ll\gg (1 - \beta_1)) \cdot \hat{m} \ll\gg v_t^{-1})$

Algorithm 5 Running a BNN. L is the number of layers.

Require: a vector of 8-bit inputs a_0, the binary weights W^b, and the BatchNorm parameters θ.
Ensure: the MLP output a_L.
  {1. First layer:}
  $a_1 \leftarrow 0$
  for $n = 1$ to 8 **do**
    $a_1 \leftarrow a_1 + 2^{n-1} \times \text{XnorDotProduct}(\mathrm{a}_0^n, \mathrm{W}_1^b)$
  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:}
  $a_L \leftarrow \text{XnorDotProduct}(a_{L-1}^b, W_L^b)$
  $a_L \leftarrow \text{BatchNorm}(a_L, \theta_L)$

1.4. Shift based Batch Normalization

Batch Normalization (BN) (Ioffe & Szegedy, 2015), accelerates the training and also seems to reduces the overall impact of the weights' scale. The normalization noise 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 (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 $3 \times 128 \times 28 \times 28$, which is two orders of magnitude larger than the number of weights. To achieve the results that BN would obtain, we use a shift-based batch normalization (SBN) technique, detailed in Algorithm 3. SBN approximates BN almost without multiplications. In the experiment we conducted we did not observe accuracy loss when using the shift based BN algorithm instead of the vanilla BN algorithm.

1.5. Shift based AdaMax

The ADAM learning rule (Kingma & Ba, 2014) also seems to reduce the impact of the weight scale. Since ADAM requires many multiplications, we suggest using instead the shift-based AdaMax we detail in Algorithm 4. In the experiment we conducted we did not observe accuracy loss when using the shift-based AdaMax algorithm instead of the vanilla ADAM algorithm.

1.6. 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, all the layers inputs 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 much fewer channels (e.g, Red, Green and Blue) than internal representations (e.g, 512). As a result, 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:

$$s = x \cdot w^b \quad (6)$$

$$s = \sum_{n=1}^{8} 2^{n-1}(x^n \cdot w^b), \quad (7)$$

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 trick is used in Algorithm 5.

2. Benchmark Results

We conduct two sets of experiments, each based on a different framework, namely Torch7 (Collobert et al., 2011) and Theano (Bergstra et al., 2010; Bastien et al., 2012). Other than the framework, the two sets of experiments are very similar:

- 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*

Table 1. Classification test error rates of DNNs trained on MNIST (MLP architecture without unsupervised pretraining), CIFAR-10

(without data augmentation) and SVHN.

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., 2015)	1.29± 0.08%	2.30%	9.90%
Binarized activations+weights, during test			
EBP (Cheng et al., 2015)	2.2± 0.1%	-	-
Bitwise DNNs (Kim & Smaragdis, 2016)	1.33%	-	-
Ternary weights, binary activations, during test			
(Hwang & Sung, 2014)	1.45%	-	-
No binarization (standard results)			
Maxout Networks (Goodfellow et al.)	0.94%	2.47%	11.68%
Network in Network (Lin et al.)	-	2.35%	10.41%
Gated pooling (Lee et al., 2015)	-	1.69%	7.62%

Figure 1. Training curves of a ConvNet on CIFAR-10 depending on the method. 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.

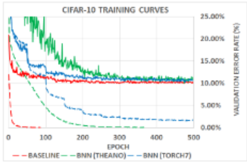


Figure 2. Binary weight filters, sampled from of the first convolution layer. Since we have only 2^{k^2} unique 2D filters (where k is the filter size), filter replication is very common. For instance, on our CIFAR-10 ConvNet, only 42% of the filters are unique.

and AdaMax variants, which are detailed in Algorithms 3 and 4, whereas in our Theano experiments, we use vanilla BN and ADAM.

2.1. 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 MLP we train on MNIST consists of 3 hidden layers of 4096 binary units (see Section 1) 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, 2013; Srivastava et al., 2014). The square hinge loss is minimized with the ADAM adaptive learning rate method (Kingma & Ba, 2014). 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 & Bengio, 2010), as suggested by Courbariaux et al. (2015). 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). The results are reported in Table 1.

2.2. 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. The results are presented in Table 1.

2.3. 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 instance are 32 × 32 color images representing airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships and trucks. We do not use any preprocessing or data-augmentation (which can really be a game changer for this dataset (Graham, 2014)). The architecture of our ConvNet is the same architecture as **?**'s except for the binarization of the activations. Courbariaux et al. (2015)'s architecture is itself mainly inspired by VGG (Simonyan & 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 & 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). The results are presented in Table 1 and Figure 1.

2.4. 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. The results are presented in Table 1 and Figure 1.

2.5. 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). The results are given in Table 1.

3. Very Power Efficient in Forward Pass

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 a great increase in 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% .

Table 2. Energy consumption of multiply-accumulations (Horowitz, 2014)

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

Table 3. Energy consumption of memory accesses (Horowitz, 2014)

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

3.1. 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 much research effort has been devoted to reducing the energy consumption of neural networks. Horowitz (2014) provides rough numbers for the computations' energy consumption (the given numbers are for 45nm technology) as summarized in Tables 2 and 3. Importantly, we can see that memory accesses typically consume more energy than arithmetic operations, and *memory access' cost augments 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., more than 32 times).

3.2. XNOR-Count

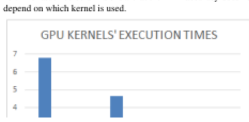
Applying a DNN mainly consists of convolutions and matrix multiplications. The key arithmetic operation of deep learning is thus the multiply-accumulate operation. Artificial neurons are basically multiply-accumulators computing weighted sums of their inputs. In BNNs, both the activations and the weights 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 deep learning dedicated 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.

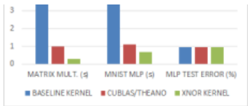
3.3. Exploiting Filter Repetitions

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.

Figure 3. The first three columns represent 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. We can see that our XNOR kernel is 23 times faster than our baseline kernel and 3.4 times faster than cuBLAS. The next three columns represent the time it takes to run the MLP from Section 2 on the full MNIST test set. 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.



By using dedicated hardware/software, we can apply only the unique 2D filters on each feature map and sum the result wisely 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.



4. Seven Times Faster on GPU at Run-Time

It is possible to speed up GPU implementations of BNNs, 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:

$$a_1 += \mathrm{popcount}(\mathrm{xnor}(a_0^{32b}, w_1^{32b})), \qquad (8)$$

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 by 6 times.

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

- The first kernel (baseline) is a quite unoptimized matrix multiplication kernel.
- The second kernel (XNOR) is nearly identical to the baseline kernel, except that it uses the SWAR method, as in Equation (8).

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 2 runs 7 times faster with the XNOR kernel than with the baseline kernel, without suffering any loss in classification accuracy (see Figure 3).

5. Discussion and Related Work

Until recently, the use of extremely low-precision networks (binary in the extreme case) was believed to be highly destructive to the network performance (Courbariaux et al., 2014). Soudry et al. (2014); **?** showed 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 bi-

nary 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 were only used during inference.

The probabilistic idea behind EBP was extended in the BinaryConnect algorithm of Courbariaux et al. (2015). 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 (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 by the back propagated error by simply ignoring the binarization noise in the update. Using this method, Courbariaux et al. (2015) 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 in (Wan et al., 2013). This method binarized weights while still maintaining full precision neurons.

Lin et al. (2015) carried over the work of Courbariaux et al. (2015) 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 of power-of-two integers. Lin et al. (2015)'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. (2015) quantize the neurons only during the back propagation process, and not during forward propagation.

Other research (Baldassi et al., 2015) showed that fully binary training and testing is possible in an array of committee machines with randomized input, where only one weight layer is being adjusted. Judd et al. and Gong et al. aimed to compress a fully trained high precision network by using a quantization or matrix factorization methods. These methods required training the network with full precision weights and neurons, thus requiring numerous MAC operations avoided by the proposed BNN algorithm. Hwang & Sung (2014) focused on a fixed-point neural network design and achieved performance almost identical to that of the floating-point architecture. Kim et al. (2014) provided evidence that DNNs with ternary weights, used on a dedicated circuit, consume very low power and can be operated with only on-chip memory, at run time. Sung et al. also indicated satisfactory empirical performance of neural networks with 8-bit precision. Kim & Paris (2015) *retrained* neural networks with binary weights and activations.

So far, to the best of our knowledge, no work has 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. (2015), 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 requires relatively high energy resources. Novel memory devices might be used to alleviate this issue in the future; see e.g. (Soudry et al.).

Conclusion

We have introduced BNNs, DNNs with binary weights and activations at run-time and when computing the parameters gradients at train-time (see Section 1). We have conducted two sets of experiments on two different frameworks, Torch7 and Theano, which show that it is possible to train BNNs on MNIST, CIFAR-10 and SVHN, and achieve nearly state-of-the-art results (see Section 2). Moreover, 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 a great increase in power-efficiency (see Section 3). Last but not least, we programed a binary matrix multiplication GPU kernel with which it is possible to run our MNIST MLP 7 times faster than with an unoptimized GPU kernel, without suffering any loss in classification accuracy (see Section 4). Future works should explore how to extend the speed-up to train-time (e.g., by binarizing some gradients), and also extend benchmark results to other models (e.g, RNN) and datasets (e.g, ImageNet).

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 which allowed us to easily develop a fast and optimized code for GPU. We also thank the developers of Pylearn2 (Goodfellow et al., 2013) 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 CIFAR, NSERC, IBM, Samsung, and the Israel Science Foundation (ISF).

References

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. In *ICLR'2015, arXiv:1409.0473*, 2015.

Baldassi, Carlo, Ingrosso, Alessandro, Lucibello, Carlo, Saglietti, Luca, and Zecchina, Riccardo. 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.

Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Bergstra, James, Goodfellow, Ian J., Bergeron, Arnaud, Bouchard, Nicolas, and Bengio, Yoshua. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

Beauchamp, Michael J, Hauck, Scott, Underwood, Keith D, and Hemmert, K Scott. Embedded floating-point units in FPGAs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pp. 12–20. ACM, 2006.

Bengio, Yoshua. Estimating or propagating gradients through stochastic neurons. Technical Report arXiv:1305.2982, Universite de Montreal, 2013.

Bergstra, James, Breuleux, Olivier, Bastien, Frédéric, Lam-

Coates, Adam, Huval, Brody, Wang, Tao, Wu, David, Catanzaro, Bryan, and Andrew, Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th international conference on machine learning*, pp. 1337–1345, 2013.

Collobert, Ronan, Kavukcuoglu, Koray, and Farabet, Clément. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Courbariaux, Matthieu, Bengio, Yoshua, and David, Jean-Pierre. Training deep neural networks with low precision multiplications. *ArXiv e-prints*, abs/1412.7024, December 2014.

Courbariaux, Matthieu, Bengio, Yoshua, and David, Jean-Pierre. Binaryconnect: Training deep neural networks with binary weights during propagations. *ArXiv e-prints*, abs/1511.00363, November 2015.

Devlin, Jacob, Zbib, Rabih, Huang, Zhongqiang, Lamar, Thomas, Schwartz, Richard, and Makhoul, John. Fast and robust neural network joint models for statistical machine translation. In *Proc. ACL'2014*, 2014.

Dieleman, Sander, Schlter, Jan, Raffel, Colin, Olson, Eben, Snderby, Sren Kaae, Nouri, Daniel, Maturana, Daniel, Thoma, Martin, Battenberg, Eric, Kelly, Jack, Fauw, Jeffrey De, Heilman, Michael, diogo149, McFee, Brian, Weideman, Hendrik, takacsg84, peterderivaz, Jon, instagibbs, Rasul, Dr. Kashif, CongLiu, Britefury, and Degrave, Jonas. Lasagne: First release., August 2015.

Esser, Steve K, Appuswamy, Rathinakumar, Merolla, Paul, Arthur, John V, and Modha, Dharmendra S. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems*, pp. 1117–1125, 2015.

Farabet, Clément, LeCun, Yann, Kavukcuoglu, Koray, Culurciello, Eugenio, Martini, Berin, Akselrod, Polina, and Talay, Selcuk. Large-scale FPGA-based convolutional networks. *Machine Learning on Very Large Data Sets*, 1, 2011a.

Farabet, Clément, Martini, Berin, Corda, Benoit, Akselrod, Polina, Culurciello, Eugenio, and LeCun, Yann. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Com-*

Bergstra, James, Breuleux, Olivier, Bastien, Frederic, Lamblin, Pascal, Pascanu, Razvan, Desjardins, Guillaume, Turian, Joseph, Warde-Farley, David, and Bengio, Yoshua. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

Chen, Tianshi, Du, Zidong, Sun, Ninghui, Wang, Jia, Wu, Chengyong, Chen, Yunji, and Temam, Olivier. 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*, pp. 269–284. ACM, 2014a.

Chen, Yunji, Luo, Tao, Liu, Shaoli, Zhang, Shijin, He, Liqiang, Wang, Jia, Li, Ling, Chen, Tianshi, Xu, Zhiwei, Sun, Ninghui, et al. Dadiannao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622. IEEE, 2014b.

Cheng, Zhiyong, Soudry, Daniel, Mao, Zexi, and Lan, Zhenzhong. Training binary multilayer neural networks for image classification using expectation backpropgation. *arXiv preprint arXiv:1503.03562*, 2015.

runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 109–116. IEEE, 2011b.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS'2010*, 2010.

Gong, Yunchao, Liu, Liu, Yang, Ming, and Bourdev, Lubomir. Compressing Deep Convolutional Networks using Vector Quantization. pp. 1–10.

Gong, Yunchao, Liu, Liu, Yang, Ming, and Bourdev, Lubomir. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Goodfellow, Ian J., Warde-Farley, David, Mirza, Mehdi, Courville, Aaron, and Bengio, Yoshua. Maxout Networks. *arXiv preprint*, pp. 1319–1327.

Goodfellow, Ian J., Warde-Farley, David, Lamblin, Pascal, Dumoulin, Vincent, Mirza, Mehdi, Pascanu, Razvan, Bergstra, James, Bastien, Frédéric, and Bengio, Yoshua. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

Govindu, Gokul, Zhuo, Ling, Choi, Seonil, and Prasanna, Viktor. Analysis of high-performance floating-point arithmetic on FPGAs. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pp. 149. IEEE, 2004.

Graham, Benjamin. Spatially-sparse convolutional neural networks. *arXiv preprint arXiv:1409.6070*, 2014.

Graves, Alex. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pp. 2348–2356, 2011.

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

Hinton, Geoffrey. Neural networks for machine learning. Coursera, video lectures, 2012.

Hinton, Geoffrey, Deng, Li, Dahl, George E., Mohamed, Abdelrahman, Jaitly, Navdeep, Senior, Andrew, Vanhoucke, Vincent, Nguyen, Patrick, Sainath, Tara, and Kingsbury, Brian. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov. 2012.

Horowitz, Mark. Computing's Energy Problem (and what we can do about it). *IEEE Interational Solid State Circuits Conference*, pp. 10–14, 2014.

Hwang, Kyuyeon and Sung, Wonyong. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6. IEEE, 2014.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

Judd, Patrick, Albericio, Jorge, Hetherington, Tayler, Aamodt, Tor, Jerger, Natalie Enright, Urtasun, Raquel, and Moshovos, Andreas. Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets. pp. 12.

Kim, Jonghong, Hwang, Kyuyeon, and Sung, Wonyong. X1000 real-time phoneme recognition vlsi using feed-forward deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 7510–7514. IEEE, 2014.

Kim, M. and Smaragdis, P. Bitwise Neural Networks. *ArXiv e-prints*, January 2016.

Kim, Minje and Paris, Smaragdis. Bitwise Neural Networks. *ICML Workshop on Resource-Efficient Machine Learning*, 37, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, A., Sutskever, I., and Hinton, G. ImageNet classification with deep convolutional neural networks. In *NIPS'2012*. 2012.

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

Lee, Chen-Yu, Xie, Saining, Gallagher, Patrick, Zhang, Zhengyou, and Tu, Zhuowen. Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*, 2014.

Lee, Chen-Yu, Gallagher, Patrick W, and Tu, Zhuowen. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. *arXiv preprint arXiv:1509.08985*, 2015.

Lin, Min, Chen, Qiang, and Yan, Shuicheng. Network In Network. *arXiv preprint*, pp. 10.

Lin, Zhouhan, Courbariaux, Matthieu, Memisevic, Roland, and Bengio, Yoshua. Neural networks with few multiplications. *ArXiv e-prints*, abs/1510.03009, October 2015.

Mnih, Volodymyr, Kavukcuoglo, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidgeland, Andreas K., Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharsan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

Mordvintsev, Alexander, Olah, Christopher, and Tyka, Mike. Inceptionism: Going deeper into neural networks, 2015. Accessed: 2015-06-30.

Pham, Phi-Hung, Jelaca, Darko, Farabet, Clement, Martini, Berin, LeCun, Yann, and Culurciello, Eugenio. Neuflow: dataflow vision processing system-on-a-chip. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pp. 1044–1047. IEEE, 2012.

Romero, Adriana, Ballas, Nicolas, Kahou, Samira Ebrahimi, Chassang, Antoine, Gatta, Carlo, and Bengio, Yoshua. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

Sainath, Tara, rahman Mohamed, Abdel, Kingsbury, Brian, and Ramabhadran, Bhuvana. Deep convolutional neural networks for LVCSR. In *ICASSP 2013*, 2013.

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

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Soudry, Daniel, Di Castro, Dotan, Gal, Asaf, Kolodny, Avinoam, and Kvatinsky, Shahar. Memristor-Based Multilayer Neural Networks With Online Gradient Descent Training. *IEEE Transactions on Neural Networks and Learning Systems*, (10): 2408–2421.

Soudry, Daniel, Hubara, Itay, and Meir, Ron. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NIPS'2014*, 2014.

Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to $+1$ or -1

Srivastava, Nitish. Improving neural networks with dropout. Master's thesis, U. Toronto, 2013.

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

Sung, Wonyong, Shin, Sungho, and Hwang, Kyuyeon. Resiliency of Deep Neural Networks under Quantization. (2014):1–9.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *NIPS'2014*, 2014.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. Going deeper with convolutions. Technical report, arXiv:1409.4842, 2014.

Tang, Yichuan. Deep learning using linear support vector machines. Workshop on Challenges in Representation Learning, ICML, 2013.

Vanhoucke, Vincent, Senior, Andrew, and Mao, Mark Z. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

Wan, Li, Zeiler, Matthew, Zhang, Sixin, LeCun, Yann, and Fergus, Rob. Regularization of neural networks using dropconnect. In *ICML'2013*, 2013.