# BinaryConnect: Training Deep Neural Networks with binary weights during propagations

**Matthieu Courbariaux**
École Polytechnique de Montréal
matthieu.courbariaux@polymtl.ca

**Yoshua Bengio**
Université de Montréal, CIFAR Senior Fellow
yoshua.bengio@gmail.com

**Jean-Pierre David**
École Polytechnique de Montréal
jean-pierre.david@polymtl.ca

## Abstract

Deep Neural Networks (DNN) have achieved state-of-the-art results in a wide range of tasks, with the best results obtained with large training sets and large models. In the past, GPUs enabled these breakthroughs because of their greater computational speed. In the future, faster computation at both training and test time is likely to be crucial for further progress and for consumer applications on low-power devices. As a result, there is much interest in research and development of dedicated hardware for Deep Learning (DL). Binary weights, i.e., weights which are constrained to only two possible values (e.g. -1 or 1), would bring great benefits to specialized DL hardware by replacing many multiply-accumulate operations by simple accumulations, as multipliers are the most space and power-hungry components of the digital implementation of neural networks. We introduce BinaryConnect, a method which consists in training a DNN with binary weights during the forward and backward propagations, while retaining precision of the stored weights in which gradients are accumulated. Like other dropout schemes, we show that BinaryConnect acts as regularizer and we obtain near state-of-the-art results with BinaryConnect on the permutation-invariant MNIST, CIFAR-10 and SVHN.

## 1 Introduction

Deep Neural Networks (DNN) have substantially pushed the state-of-the-art in a wide range of tasks, especially in speech recognition [1, 2] and computer vision, notably object recognition from images [3, 4]. More recently, deep learning is making important strides in natural language processing, especially statistical machine translation [5, 6, 7]. Interestingly, one of the key factors that enabled this major progress has been the advent of Graphics Processing Units (GPUs), with speed-ups on the order of 10 to 30-fold, starting with [8], and similar improvements with distributed training [9, 10]. Indeed, the ability to train larger models on more data has enabled the kind of breakthroughs observed in the last few years. Today, researchers and developers designing new deep learning algorithms and applications often find themselves limited by computational capability. This along, with the drive to put deep learning systems on low-power devices (unlike GPUs) is greatly increasing the interest in research and development of specialized hardware for deep networks [11, 12, 13].

Most of the computation performed during training and application of deep networks regards the multiplication of a real-valued weight by a real-valued activation (in the recognition or forward propagation phase of the back-propagation algorithm) or gradient (in the backward propagation phase of the back-propagation algorithm). This paper proposes an approach called BinaryConnect

to eliminate the need for these multiplications by forcing the weights used in these forward and backward propagations to be binary, i.e. constrained to only two values (not necessarily 0 and 1). We show that state-of-the-art results can be achieved with BinaryConnect on the permutation-invariant MNIST, CIFAR-10 and SVHN.

What makes this workable are two ingredients:

1. Sufficient precision is necessary to accumulate and average a large number of stochastic gradients, but noisy weights (and we can view discretization into a small number of values as a form of noise, especially if we make this discretization stochastic) are quite compatible with Stochastic Gradient Descent (SGD), the main type of optimization algorithm for deep learning. SGD explores the space of parameters by making 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 sight suggests that high precision is absolutely required. [14] and [15] show that randomized or stochastic rounding can be used to provide unbiased discretization. [14] have shown that SGD requires weights with a precision of at least 6 to 8 bits and [16] successfully train DNNs with 12 bits dynamic fixed-point computation. Besides, the estimated precision of the brain synapses varies between 6 and 12 bits [17].
2. Noisy weights actually provide a form of regularization which can help to generalize better, as previously shown with variational weight noise [18], Dropout [19, 20] and DropConnect [21], which add noise to the activations or to the weights. For instance, DropConnect [21], which is closest to BinaryConnect, is a very efficient regularizer that randomly substitutes half of the weights with zeros during propagations. What these previous works show is that *only the expected value of the weight needs to have high precision*, and that noise can actually be beneficial.

The main contributions of this article are the following.

- We introduce BinaryConnect, a method which consists in training a DNN with binary weights during the forward and backward propagations (Section 2).
- We show that BinaryConnect is a regularizer and we obtain near state-of-the-art results on the permutation-invariant MNIST, CIFAR-10 and SVHN (Section 3).
- We make the code for BinaryConnect available [1].

## 2 BinaryConnect

In this section we give a more detailed view of BinaryConnect, considering which two values to choose, how to discretize, how to train and how to perform inference.

### 2.1 $+1$ or $-1$

Applying a DNN mainly consists in convolutions and matrix multiplications. The key arithmetic operation of DL is thus the multiply-accumulate operation. Artificial neurons are basically multiply-accumulators computing weighted sums of their inputs.

BinaryConnect constraints the weights to either $+1$ or $-1$ during propagations. As a result, many multiply-accumulate operations are replaced by simple additions (and subtractions). This is a huge gain, as fixed-point adders are much less expensive both in terms of area and energy than fixed-point multiply-accumulators [22].

### 2.2 Deterministic vs stochastic binarization

The binarization operation transforms the real-valued weights into the two possible values. A very straightforward binarization operation would be based on the sign function:

$$w_b = \begin{cases} +1 & \text{if } w \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{1}$$

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

Where $w_b$ is the binarized weight and $w$ the real-valued weight. Although this is a deterministic operation, averaging this discretization over the many input weights of a hidden unit could compensate for the loss of information. An alternative that allows a finer and more correct averaging process to take place is to binarize stochastically:

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w), \\ -1 & \text{with probability } 1 - p. \end{cases} \tag{2}$$

where $\sigma$ is the *"hard sigmoid"* function:

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

We use such a hard sigmoid rather than the soft version because it is far less computationally expensive (both in software and specialized hardware implementations) and yielded excellent results in our experiments. It is similar to the "hard tanh" non-linearity introduced by [23]. It is also piece-wise linear and corresponds to a bounded form of the rectifier [24].

### 2.3 Propagations vs updates

Let us consider the different steps of back-propagation with SGD udpates and whether it makes sense, or not, to discretize the weights, at each of these steps.

1. Given the DNN input, compute the unit activations layer by layer, leading to the top layer which is the output of the DNN, given its input. This step is referred as the forward propagation.
2. Given the DNN target, compute the training objective's gradient w.r.t. each layer's activations, starting from the top layer and going down layer by layer until the first hidden layer. This step is referred to as the backward propagation or backward phase of back-propagation.
3. Compute the gradient w.r.t. each layer's parameters and then update the parameters using their computed gradients and their previous values. This step is referred to as the parameter update.

**Algorithm 1** SGD training with BinaryConnect. $C$ is the cost function for minibatch and the functions binarize($w$) and clip($w$) specify how to binarize and clip weights. $L$ is the number of layers.

**Require:** a minibatch of (inputs, targets), previous parameters $w_{t-1}$ (weights) and $b_{t-1}$ (biases), and learning rate $\eta$.

**Ensure:** updated parameters $w_t$ and $b_t$.

**1. Forward propagation:**

$w_b \leftarrow \text{binarize}(w_{t-1})$

For $k = 1$ to $L$, compute $a_k$ knowing $a_{k-1}$, $w_b$ and $b_{t-1}$

**2. Backward propagation:**

Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$

For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and $w_b$

**3. Parameter update:**

Compute $\frac{\partial C}{\partial w_b}$ and $\frac{\partial C}{\partial b_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and $a_{k-1}$

$w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_b})$

$b_t \leftarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$

A key point to understand with BinaryConnect is that we only binarize the weights during the forward and backward propagations (steps 1 and 2) but not during the parameter update (step 3), as illustrated in Algorithm 1. Keeping good precision weights during the updates is necessary for SGD to work at all. These parameter changes are tiny by virtue of being obtained by gradient descent, i.e., SGD performs a large number of almost infinitesimal changes in the direction that most improves the training objective (plus noise). One way to picture all this is to hypothesize that what matters

most at the end of training is the sign of the weights, $w^*$, but that in order to figure it out, we perform a lot of small changes to a continuous-valued quantity $w$, and only at the end consider its sign:

$$w^* = \text{sign}(\sum_t g_t) \tag{4}$$

where $g_t$ is a noisy estimator of $\frac{\partial C(f(x_t,w_{t-1},b_{t-1}),y_t)}{\partial w_{t-1}}$, where $C(f(x_t, w_{t-1}, b_{t-1}), y_t)$ is the value of the objective function on (input,target) example $(x_t, y_t)$, when $w_{t-1}$ are the previous weights and $w^*$ is its final discretized value of the weights.

Another way to conceive of this discretization is as a form of corruption, and hence as a regularizer, and our empirical results confirm this hypothesis. In addition, *we can make the discretization errors on different weights approximately cancel each other while keeping a lot of precision by randomizing the discretization appropriately.* We propose a form of randomized discretization that *preserves the expected value of the discretized weight.*

Hence, at training time, BinaryConnect randomly picks one of two values for each weight, for each minibatch, for both the forward and backward propagation phases of backprop. However, the SGD update is accumulated in a real-valued variable storing the parameter.

An interesting analogy to understand BinaryConnect is the DropConnect algorithm [21]. Just like BinaryConnect, DropConnect only injects noise to the weights during the propagations. Whereas DropConnect's noise is added Gaussian noise, BinaryConnect's noise is a binary sampling process. In both cases the corrupted value has as expected value the clean original value.

### 2.4 Clipping

Since the binarization operation is not influenced by variations of the real-valued weights $w$ when its magnitude is beyond the binary values $\pm 1$, and since it is a common practice to bound weights (usually the weight vector) in order to regularize them, we have chosen to clip the real-valued weights within the $[-1, 1]$ interval right after the weight updates, as per Algorithm 1. The real-valued weights would otherwise grow very large without any impact on the binary weights.

### 2.5 A few more tricks

| Optimization | No learning rate scaling | Learning rate scaling |
|---|---|---|
| SGD | | 11.45% |
| Nesterov momentum | 15.65% | 11.30% |
| ADAM | 12.81% | **10.47%** |

Table 1: Test error rates of a (small) CNN trained on CIFAR-10 depending on optimization method and on whether the learning rate is scaled with the weights initialization coefficients from [25].

We use Batch Normalization (BN) [26] in all of our experiments, not only because it accelerates the training by reducing internal covariate shift, but also because it reduces the overall impact of the weights scale. Moreover, we use the ADAM learning rule [27] in all of our CNN experiments. Last but not least, we scale the weights learning rates respectively with the weights initialization coefficients from [25] when optimizing with ADAM, and with the squares of those coefficients when optimizing with SGD or Nesterov momentum [28]. Table 1 illustrates the effectiveness of those tricks.

### 2.6 Test-Time Inference

Up to now we have introduced different ways of *training* a DNN with on-the-fly weight binarization. What are reasonable ways of using such a trained network, i.e., performing test-time inference on new examples? We have considered three reasonable alternatives:

1. Use the resulting binary weights $w_b$ (this makes most sense with the deterministic form of BinaryConnect).

2. Use the real-valued weights $w$, i.e., the binarization only helps to achieve faster training but not faster test-time performance.
3. In the stochastic case, many different networks can be sampled by sampling a $w_b$ for each weight according to Eq. 2. The ensemble output of these networks can then be obtained by averaging the outputs from individual networks.

We use the first method with the deterministic form of BinaryConnect. As for the stochastic form of BinaryConnect, we focused on the training advantage and used the second method in the experiments, i.e., test-time inference using the real-valued weights. This follows the practice of Dropout methods, where at test-time the "noise" is removed.

| Method | MNIST | CIFAR-10 | SVHN |
|---|---|---|---|
| No regularizer | $1.30 \pm 0.04\%$ | 10.64% | 2.44% |
| BinaryConnect (det.) | $1.29 \pm 0.08\%$ | 9.90% | 2.30% |
| BinaryConnect (stoch.) | $1.18 \pm 0.04\%$ | **8.27%** | 2.15% |
| 50% Dropout | $1.01 \pm 0.04\%$ | | |
| Maxout Networks [29] | 0.94% | 11.68% | 2.47% |
| Deep L2-SVM [30] | **0.87%** | | |
| Network in Network [31] | | 10.41% | 2.35% |
| DropConnect [21] | | | 1.94% |
| Deeply-Supervised Nets [32] | | 9.78% | **1.92%** |

Table 2: Test error rates of DNNs trained on the MNIST (no convolution and no unsupervised pretraining), CIFAR-10 (no data augmentation) and SVHN, depending on the method. We see that in spite of using only a single bit per weight during propagation, performance is not worse than ordinary (no regularizer) DNNs, it is actually better, especially with the stochastic version, suggesting that BinaryConnect acts as a regularizer.

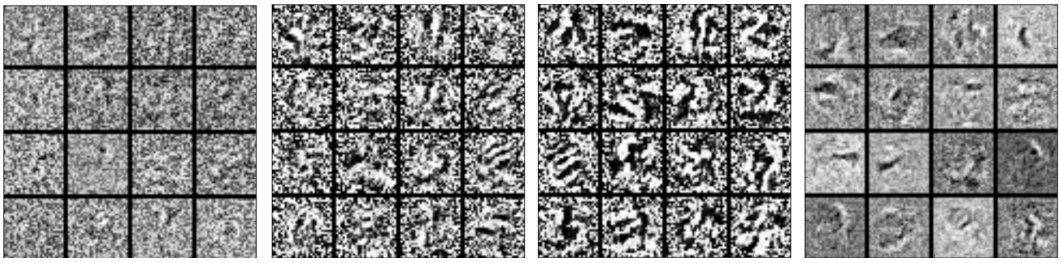

Figure 1: Features of the first layer of an MLP trained on MNIST depending on the regularizer. From left to right: no regularizer, deterministic BinaryConnect, stochastic BinaryConnect and Dropout.

## 3 Benchmark results

In this section, we show that BinaryConnect acts as regularizer and we obtain near state-of-the-art results with BinaryConnect on the permutation-invariant MNIST, CIFAR-10 and SVHN.

### 3.1 Permutation-invariant MNIST

MNIST is a benchmark image classification dataset [33]. It consists in a training set of 60000 and a test set of 10000 $28 \times 28$ gray-scale images representing digits ranging from 0 to 9. Permutation-invariance means that the model must be unaware of the image (2-D) structure of the data (in other words, CNNs are forbidden). Besides, we do not use any data-augmentation, preprocessing or unsupervised pretraining. The MLP we train on MNIST consists in 3 hidden layers of 1024 Rectifier Linear Units (ReLU) [34, 24, 3] and a L2-SVM output layer (L2-SVM has been shown to perform better than Softmax on several classification benchmarks [30, 32]). The square hinge loss is minimized with SGD without momentum. We use an exponentially decaying learning rate. We use Batch

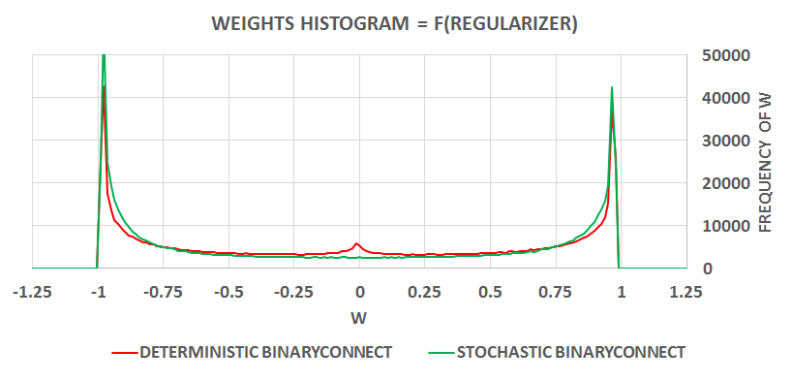


Figure 2: Histogram of the weights of the first layer of an MLP trained on MNIST depending on the regularizer. In both cases, it seems that the weights are trying to become deterministic to reduce the training error. It also seems that some of the weights of deterministic BinaryConnect are stuck around 0, hesitating between $-1$ and 1.

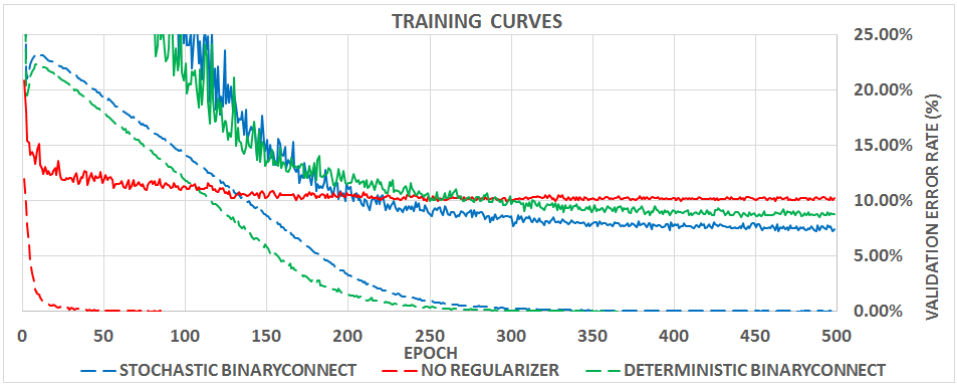


Figure 3: Training curves of a CNN on CIFAR-10 depending on the regularizer. The dotted lines represent the training costs (square hinge losses) and the continuous lines the corresponding validation error rates. Both versions of BinaryConnect significantly augment the training cost, slow down the training and lower the validation error rate, which is what we would expect from a Dropout scheme.

Normalization with a minibatch of size 200 to speed up the training. As typically done, we use the last 10000 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). We repeat each experiment 6 times with different initializations. The results are in Table 2. They suggest that the stochastic version of BinaryConnect can be considered a regularizer, although a slightly less powerful one than Dropout, in this context.

### 3.2 CIFAR-10

CIFAR-10 is a benchmark image classification dataset. It consists in a training set of 50000 and a test set of 10000 32 $\times$ 32 color images representing airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships and trucks. We preprocess the data using global contrast normalization and ZCA whitening. We do not use any data-augmentation (which can really be a game changer for this dataset [35]). The architecture of our CNN is:

$$(2\times128C3)-MP2-(2\times256C3)-MP2-(2\times512C3)-MP2-(2\times1024FC)-10SVM \quad (5)$$

Where $C3$ is a 3 $\times$ 3 ReLU convolution layer, $MP2$ is a 2 $\times$ 2 max-pooling layer, $FC$ a fully connected layer, and SVM a L2-SVM output layer. This architecture is greatly inspired from VGG [36]. The square hinge loss is minimized with ADAM. We use an exponentially decaying learning

rate. 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 in Table 2 and Figure 3.

### 3.3 SVHN

SVHN is a benchmark image classification dataset. It consists in a training set of 604K and a test set of 26K $32 \times 32$ color images representing digits ranging from 0 to 9. We follow the same procedure that we used for CIFAR-10, with a few notable exceptions: we use half the number of hidden units and we train for 200 epochs instead of 500 (because SVHN is quite a big dataset). The results are in Table 2.

## 4 Related works

Training DNNs with binary weights has been the subject of very recent works [37, 38, 39, 40]. Even though we share the same objective, our approaches are quite different. [37, 38] do not train their DNN with Backpropagation (BP) but with a variant called Expectation Backpropagation (EBP). EBP is based on Expectation Propagation (EP) [41], which is a variational Bayes method used to do inference in probabilistic graphical models. Let us compare their method to ours:

- It optimizes the weights posterior distribution (which is not binary). In this regard, our method is quite similar as we keep a real-valued version of the weights.
- It binarizes both the neurons outputs and weights, which is more hardware friendly than just binarizing the weights.
- It yields a good classification accuracy for fully connected networks (on MNIST) but not (yet) for ConvNets.

[39, 40] *retrain* neural networks with *ternary* weights during forward and backward propagations, i.e.:

- They train a neural network with high-precision,
- After training, they ternarize the weights to three possible values $-H$, 0 and $+H$ and adjust $H$ to minimize the output error,
- And eventually, they retrain with ternary weights during propagations and high-precision weights during updates.

By comparison, we *train all the way* with *binary* weights during propagations, i.e., our training procedure could be implemented with efficient specialized hardware avoiding the forward and backward propagations multiplications, which amounts to about $2/3$ of the multiplications (cf. Algorithm 1).

## 5 Conclusion and future works

We have introduced a novel binarization scheme for weights during forward and backward propagations called BinaryConnect. We have shown that it is possible to train DNNs with BinaryConnect on the permutation invariant MNIST, CIFAR-10 and SVHN datasets and achieve nearly state-of-the-art results. The impact of such a method on specialized hardware implementations of deep networks could be major, by removing the need for about 2/3 of the multiplications, and thus potentially allowing to speed-up by a factor of 3 at training time. With the deterministic version of BinaryConnect the impact at test time could be even more important, getting rid of the multiplications altogether and reducing by a factor of at least 16 (from 16 bits single-float precision to single bit precision) the memory requirement of deep networks, which has an impact on the memory to computation bandwidth and on the size of the models that can be run. Future works should extend those results to other models and datasets, and explore getting rid of the multiplications altogether during training, by removing their need from the weight update computation.

## 6 Acknowledgments

We thank the reviewers for their many constructive comments. We also thank Roland Memisevic for helpful discussions. We thank the developers of Theano [42, 43], a Python library which allowed us to easily develop a fast and optimized code for GPU. We also thank the developers of Pylearn2 [44] and Lasagne, two Deep Learning libraries built on the top of Theano. We are also grateful for funding from NSERC, the Canada Research Chairs, Compute Canada, and CIFAR.

## References

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

[8] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *ICML'2009*, 2009.

[9] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[10] J. Dean, G.S Corrado, R. Monga, K. Chen, M. Devin, Q.V. Le, M.Z. Mao, M.A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS'2012*, 2012.

[11] Sang Kyun Kim, Lawrence C McAfee, Peter Leonard McMahon, and Kunle Olukotun. A highly scalable restricted Boltzmann machine FPGA implementation. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 367–372. IEEE, 2009.

[12] 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, 2014.

[13] 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, 2014.

[14] Lorenz K Muller and Giacomo Indiveri. Rounding methods for neural networks with low resolution synaptic weights. *arXiv preprint arXiv:1504.05767*, 2015.

[15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML'2015*, 2015.

[16] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. In *Arxiv:1412.7024, ICLR'2015 Workshop*, 2015.

[17] Thomas M Bartol, Cailey Bromer, Justin P Kinney, Michael A Chirillo, Jennifer N Bourne, Kristen M Harris, and Terrence J Sejnowski. Hippocampal spine head sizes are highly precise. *bioRxiv*, 2015.

[18] Alex Graves. Practical variational inference for neural networks. In J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc., 2011.

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

[20] 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.

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

[22] J.P. David, K. Kalach, and N. Tittley. Hardware complexity of modular multiplication and exponentiation. *Computers, IEEE Transactions on*, 56(10):1308–1319, Oct 2007.

[23] R. Collobert. *Large Scale Machine Learning*. PhD thesis, Université de Paris VI, LIP6, 2004.

[24] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *AISTATS'2011*, 2011.

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

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

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

[28] Yu Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. *Doklady AN SSSR (translated as Soviet. Math. Docl.)*, 269:543–547, 1983.

[29] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. Technical Report Arxiv report 1302.4389, Université de Montréal, February 2013.

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

[31] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

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

[33] 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.

[34] V. Nair and G.E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *ICML'2010*, 2010.

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

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

[37] 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.

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

[39] 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.

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

[41] Thomas P Minka. Expectation propagation for approximate bayesian inference. In *UAI'2001*, 2001.

[42] 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.

[43] 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.

[44] 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*, 2013.