

---

# Bitwise Neural Networks

---

**Minje Kim**

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA

MINJE@ILLINOIS.EDU

**Paris Smaragdis**

University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA  
Adobe Research, Adobe Systems Inc., San Francisco, CA 94103, USA

PARIS@ILLINOIS.EDU

## Abstract

Based on the assumption that there exists a neural network that efficiently represents a set of Boolean functions between all binary inputs and outputs, we propose a process for developing and deploying neural networks whose weight parameters, bias terms, input, and intermediate hidden layer output signals, are all binary-valued, and require only basic bit logic for the feedforward pass. The proposed Bitwise Neural Network (BNN) is especially suitable for resource-constrained environments, since it replaces either floating or fixed-point arithmetic with significantly more efficient bitwise operations. Hence, the BNN requires for less spatial complexity, less memory bandwidth, and less power consumption in hardware. In order to design such networks, we propose to add a few training schemes, such as weight compression and noisy backpropagation, which result in a bitwise network that performs almost as well as its corresponding real-valued network. We test the proposed network on the MNIST dataset, represented using binary features, and show that BNNs result in competitive performance while offering dramatic computational savings.

cated function, Deep Neural Networks (DNN) achieve the goal by learning a hierarchy of features in their multiple layers (Hinton et al., 2006; Bengio, 2009).

Although DNNs are extending the state of the art results for various tasks, such as image classification (Goodfellow et al., 2013), speech recognition (Hinton et al., 2012), speech enhancement (Xu et al., 2014), etc, it is also the case that the relatively bigger networks with more parameters than before call for more resources (processing power, memory, battery time, etc), which are sometimes critically constrained in applications running on embedded devices. Examples of those applications span from context-aware computing, collecting and analysing a variety of sensor signals on the device (Baldauf et al., 2007), to always-on computer vision applications (e.g. Google glasses), to speech-driven personal assistant services, such as “Hey, Siri.” A primary concern that hinders those applications from being more successful is that they assume an always-on pattern recognition engine on the device, which will drain the battery fast unless it is carefully implemented to minimize the use of resources. Additionally, even in an environment with the necessary resources being available, speeding up a DNN can greatly improve the user experience when it comes to tasks like searching big databases (Salakhutdinov & Hinton, 2009). In either case, a more compact yet still well-performing DNN is a welcome improvement.

Efficient computational structures for deploying artificial neural networks have long been studied in the literature. Most of the effort is focused on training networks whose weights can be transformed into some quantized representations with a minimal loss of performance (Fiesler et al., 1990; Hwang & Sung, 2014). They typically use the quantized weights in the feedforward step at every training iteration, so that the trained weights are robust to the known quantization noise caused by a limited precision. It was also shown that 10 bits and 12 bits are enough to represent gradients and storing weights for implementing the state-of-the-art maxout networks even for training the network (Courbariaux et al., 2014). However, in those quantized

## 1. Introduction

According to the universal approximation theorem, a single hidden layer with a finite number of units can approximate a continuous function with some mild assumptions (Cybenko, 1989; Hornik, 1991). While this theorem implies a shallow network with a potentially intractable number of hidden units when it comes to modeling a compli-

---

*Proceedings of the 31<sup>st</sup> International Conference on Machine Learning*, Lille, France, 2015. JMLR: W&CP volume 37. Copyright 2015 by the author(s).

networks one still needs to employ arithmetic operations, such as multiplication and addition, on fixed-point values. Even though faster than floating point, they still require relatively complex logic and can consume a lot of power.

With the proposed Bitwise Neural Networks (BNN), we take a more extreme view that every input node, output node, and weight, is represented by a single bit. For example, a weight matrix between two hidden layers of 1024 units is a  $1024 \times 1025$  matrix of binary values rather than quantized real values (including the bias). Although learning those bitwise weights as a Boolean concept is an NP-complete problem (Pitt & Valiant, 1988), the bitwise networks have been studied in the limited setting, such as  $\mu$ -perceptron networks where an input node is allowed to be connected to one and only one hidden node and its final layer is a union of those hidden nodes (Golea et al., 1992). A more practical network was proposed in (Soudry et al., 2014) recently, where the posterior probabilities of the binary weights were sought using the Expectation Back Propagation (EBP) scheme, which is similar to backpropagation in its form, but has some advantages, such as parameter-free learning and a straightforward discretization of the weights. Its promising results on binary text classification tasks however, rely on the real-valued bias terms and averaging of predictions from differently sampled parameters.

This paper presents a completely bitwise network where all participating variables are bipolar binaries. Therefore, in its feedforward only XNOR and bit counting operations are used instead of multiplication, addition, and a nonlinear activation on floating or fixed-point variables. For training, we propose a two-stage approach, whose first part is typical network training with a weight compression technique that helps the real-valued model to easily be converted into a BNN. To train the actual BNN, we use those compressed weights to initialize the BNN parameters, and do noisy backpropagation based on the tentative bitwise parameters. To binarize the input signals, we can adapt any binarization techniques, e.g. fixed-point representations and hash codes. Regardless of the binarization scheme, each input node is given only a single bit at a time, as opposed to a bit packet representing a fixed-point number. This is significantly different from the networks with quantized inputs, where a real-valued signal is quantized into a set of bits, and then all those bits are fed to an input node in place of their corresponding single real value. Lastly, we apply the sign function as our activation function instead of a sigmoid to make sure the input to the next layer is bipolar binary as well. We compare the performance of the proposed BNN with its corresponding ordinary real-valued networks on hand-written digit recognition tasks, and show that the bitwise operations can do the job with a very small performance loss, while providing a large margin of improvement in terms of the necessary computational resources.

## 2. Feedforward in Bitwise Neural Networks

It has long been known that any Boolean function, which takes binary values as input and produces binary outputs as well, can be represented as a bitwise network with one hidden layer (McCulloch & Pitts, 1943), for example, by merely memorizing all the possible mappings between input and output patterns. We define the forward propagation procedure as follows based on the assumption that we have trained such a network with bipolar binary parameters:

$$a_i^l = b_i^l + \sum_j^{K^{l-1}} w_{i,j}^l \otimes z_j^{l-1}, \quad (1)$$

$$z_i^l = \text{sign}(a_i^l), \quad (2)$$

$$\mathbf{z}^l \in \mathbb{B}^{K^l}, \mathbf{W}^l \in \mathbb{B}^{K^l \times K^{l-1}}, \mathbf{b}^l \in \mathbb{B}^{K^l}, \quad (3)$$

where  $\mathbb{B}$  is the set of bipolar binaries, i.e.  $\pm 1$ <sup>1</sup>, and  $\otimes$  stands for the bitwise XNOR operation (see Figure 1 (a)).  $l, j$ , and  $i$  indicate a layer, input and output units of the layer, respectively. We use bold characters for a vector (or a matrix if capitalized).  $K^l$  is the number of input units at  $l$ -th layer. Therefore,  $\mathbf{z}^0$  equals to an input vector, where we omit the sample index for the notational convenience. We use the sign activation function to generate the bipolar outputs.

We can check the prediction error  $\mathcal{E}$  by measuring the bitwise agreement of target vector  $\mathbf{t}$  and the output units of  $L$ -th layer using XNOR as a multiplication operator,

$$\mathcal{E} = \sum_i^{K^{L+1}} (1 - t_i \otimes z_i^{L+1})/2, \quad (4)$$

but this error function can be tentatively replaced by involving a softmax layer during the training phase.

The XNOR operation is a faster substitute of binary multiplication. Therefore, (1) and (2) can be seen as a special version of the ordinary feedforward step that only works when the inputs, weights, and bias are all bipolar binaries. Note that these bipolar bits will in practice be implemented using 0/1 binary values, where (2) activation is equivalent to counting the number of 1's and then checking if the accumulation is bigger than the half of the number of input units plus 1. With no loss of generality, in this paper we will use the  $\pm 1$  bipolar representation since it is more flexible in defining hyperplanes and examining the network behavior.

Sometimes a BNN can solve the same problem as a real-valued network without any size modifications, but in general we should expect that a BNN could require larger network structures than a real-valued one. For example, the XOR problem in Figure 1 (b) can have an infinite number of solutions with real-valued parameters once a pair

<sup>1</sup>In the bipolar binary representation,  $+1$  stands for the "TRUE" status, while  $-1$  is for "FALSE."

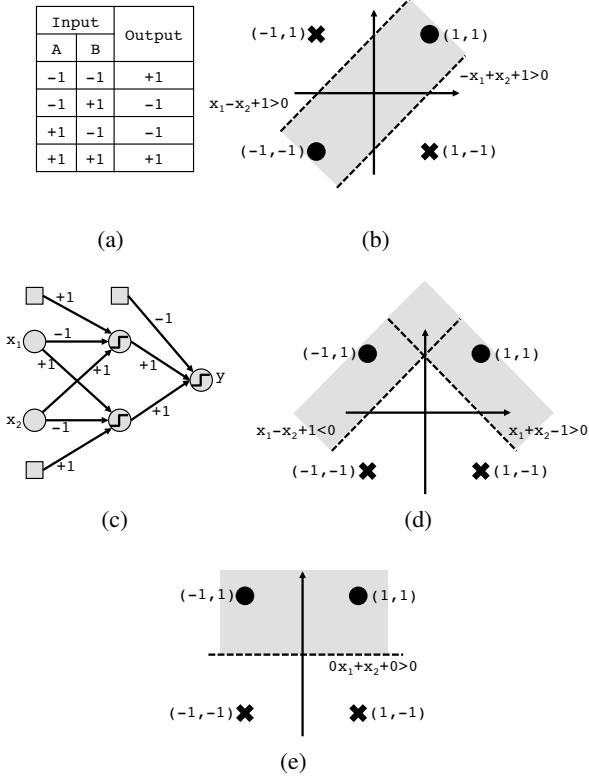


Figure 1. (a) An XNOR table. (b) The XOR problem that needs two hyperplanes. (c) A multi-layer perceptron that solves the XOR problem. (d) A linearly separable problem while bitwise networks need two hyperplanes to solve it ( $y = x_2$ ). (e) A bitwise network with zero weights that solves the  $y = x_2$  problem.

of hyperplanes can successfully discriminate  $(1, 1)$  and  $(-1, -1)$  from  $(1, -1)$  and  $(-1, 1)$ . Among all the possible solutions, we can see that binary weights and bias are enough to define the hyperplanes,  $x_1 - x_2 + 1 > 0$  and  $-x_1 + x_2 + 1 > 0$  (dashes). Likewise, the separation performance of the particular BNN defined in (c) has the same classification power once the inputs are binary as well.

Figure 1 (d) shows another example where BNN requires more hyperplanes than a real-valued network. This linearly separable problem is solvable with only one hyperplane, such as  $-0.1x_1 + x_2 + 0.5 > 0$ , but it is impossible to describe such a hyperplane with binary coefficients. We can instead come up with a solution by combining multiple binary hyperplanes that will eventually increase the perceived complexity of the model. However, even with a larger number of nodes, the BNN is not necessarily more complex than the smaller real-valued network. This is because a parameter or a node of BNN requires only one bit to represent while a real-valued node generally requires more than that, up to 64 bits. Moreover, the simple XNOR and bit counting operations of BNN bypass the computational complica-

tions of a real-valued system, such as the power consumption of multipliers and adders for the floating-point operations, various dynamic ranges of the fixed-point representations, erroneous flips of the most significant bits, etc. Note that if the bitwise parameters are sparse, we can further reduce the number of hyperplanes. For example, for an inactive element in the weight matrix  $\mathbf{W}$  due to the sparsity, we can simply ignore the computation for it similarly to the operations on the sparse representations. Conceptually, we can say that those inactive weights serve as zero weights, so that a BNN can solve the problem in Figure 1 (d) by using only one hyperplane as in (e). From now on, we will use this extended version of BNN with inactive weights, yet there are some cases where BNN needs more hyperplanes than a real-valued network even with the sparsity.

### 3. Training Bitwise Neural Networks

We first train some compressed network parameters, and then retrain them using noisy backpropagation for BNNs.

#### 3.1. Real-valued Networks with Weight Compression

First, we train a real-valued network that takes either bitwise inputs or real-valued inputs ranged between  $-1$  and  $+1$ . A special part of this network is that we constrain the weights to have values between  $-1$  and  $+1$  as well by wrapping them with  $\tanh$ . Similarly, if we choose  $\tanh$  for the activation, we can say that the network is a relaxed version of the corresponding bipolar BNN. With this weight compression technique, the relaxed forward pass during training is defined as follows:

$$a_i^l = \tanh(\bar{b}_i^l) + \sum_j^{K^{l-1}} \tanh(\bar{w}_{i,j}^l) \bar{z}_j^{l-1}, \quad (5)$$

$$\bar{z}_i^l = \tanh(a_i^l), \quad (6)$$

where all the binary values in (1) and (2) are real for the time being:  $\bar{\mathbf{W}}^l \in \mathbb{R}^{K^l \times K^{l-1}}$ ,  $\bar{\mathbf{b}}^l \in \mathbb{R}^{K^l}$ , and  $\bar{\mathbf{z}}^l \in \mathbb{R}^{K^l}$ . The bars on top of the notations are for the distinction.

Weight compression needs some changes in the backpropagation procedure. In a hidden layer we calculate the error,

$$\delta_j^l(n) = \left( \sum_i^{K^{l+1}} \tanh(\bar{w}_{i,j}^{l+1}) \delta_i^{l+1}(n) \right) \cdot \left( 1 - \tanh^2(a_j^l) \right).$$

Note that the errors from the next layer are multiplied with the compressed versions of the weights. Hence, the gradients of the parameters in the case of batch learning are

$$\nabla \bar{w}_{i,j}^l = \left( \sum_n \delta_i^l(n) \bar{z}_j^{l-1} \right) \cdot \left( 1 - \tanh^2(\bar{w}_{i,j}^l) \right),$$

$$\nabla \bar{b}_i^l = \left( \sum_n \delta_i^l(n) \right) \cdot \left( 1 - \tanh^2(\bar{b}_i^l) \right),$$

with the additional term from the chain rule on the compressed weights.

### 3.2. Training BNN with Noisy Backpropagation

Since we have trained a real-valued network with a proper range of weights, what we do next is to train the actual bitwise network. The training procedure is similar to the ones with quantized weights (Fiesler et al., 1990; Hwang & Sung, 2014), except that the values we deal with are all bits, and the operations on them are bitwise. To this end, we first initialize all the real-valued parameters,  $\bar{\mathbf{W}}$  and  $\bar{\mathbf{b}}$ , with the ones learned from the previous section. Then, we setup a sparsity parameter  $\lambda$  which says the proportion of the zeros after the binarization. Then, we divide the parameters into three groups: +1, 0, or -1. Therefore,  $\lambda$  decides the boundaries  $\beta$ , e.g.  $w_{i,j}^l = -1$  if  $\bar{w}_{i,j}^l < -\beta$ . Note that the number of zero weights  $|\bar{w}_{i,j}^l| < \beta$  equals to  $\lambda K^l K^{l-1}$ .

The main idea of this second training phase is to feedforward using the binarized weights and the bit operations as in (1) and (2). Then, during noisy backpropagation the errors and gradients are calculated using those binarized weights and signals as well:

$$\begin{aligned} \delta_j^l(n) &= \sum_i^{K^{l+1}} w_{i,j}^{l+1} \delta_i^{l+1}(n), \\ \nabla \bar{w}_{i,j}^l &= \sum_n \delta_i^l(n) z_j^{l-1}, \quad \nabla \bar{b}_i^l = \sum_n \delta_i^l(n). \end{aligned} \quad (7)$$

In this way, the gradients and errors properly take the binarization of the weights and the signals into account. Since the gradients can get too small to update the binary parameters  $\mathbf{W}$  and  $\mathbf{b}$ , we instead update their corresponding real-valued parameters,

$$\bar{w}_{i,j}^l \leftarrow \bar{w}_{i,j}^l - \eta \nabla \bar{w}_{i,j}^l, \quad \bar{b}_i^l \leftarrow \bar{b}_i^l - \eta \nabla \bar{b}_i^l, \quad (8)$$

with  $\eta$  as a learning rate parameter. Finally, at the end of each update we binarize them again with  $\beta$ . We repeat this procedure at every epoch.

## 4. Experiments

In this section we go over the details and results of the hand-written digit recognition task on the MNIST data set (LeCun et al., 1998) using the proposed BNN system. Throughout the training, we adopt the softmax output layer for these multiclass classification cases. All the networks have three hidden layers with 1024 units per layer.

From the first round of training, we get a regular dropout network with the same setting suggested in (Srivastava et al., 2014), except the fact that we used the hyperbolic tangent for both weight compression and activation to make

Table 1. Classification errors for real-valued and bitwise networks on different types of bitwise features

NETWORKS	BIPOLAR	0 OR 1	FIXED-POINT (2BITS)
FLOATING-POINT NETWORKS (64BITS)	1.17%	1.32%	1.36%
BNN	1.33%	1.36%	1.47%

the network suitable for initializing the following bipolar bitwise network. The number of iterations from 500 to 1,000 was enough to build a baseline. The first row of Table 1 shows the performance of the baseline real-valued network with 64bits floating-point. As for the input to the real-valued networks, we rescale the pixel intensities into the bipolar range, i.e. from -1 to +1, for the bipolar case (the first column). In the second column, we use the original input between 0 and 1 as it is. For the third column, we encode the four equally spaced regions between 0 to 1 into two bits, and feed each bit into each input node. Hence, the baseline network for the third input type has 1,568 binary input nodes rather than 784 as in the other cases.

Once we learn the real-valued parameters, now we train the BNN, but with binarized inputs. For instance, instead of real values between -1 and +1 in the bipolar case, we take their sign as the bipolar binary features. As for the 0/1 binaries, we simply round the pixel intensity. Fixed-point inputs are already binarized. Now we train the new BNN with the noisy backpropagation technique as described in 3.2. The second row of Table 1 shows the BNN results. We see that the bitwise networks perform well with very small additional errors. Note that the performance of the original real-valued dropout network with similar network topology (logistic units without max-norm constraint) is 1.35%.

## 5. Conclusion

In this work we propose a bitwise version of artificial neural networks, where all the inputs, weights, biases, hidden units, and outputs can be represented with single bits and operated on using simple bitwise logic. Such a network is very computationally efficient and can be valuable for resource-constrained situations, particularly in cases where floating-point / fixed-point variables and operations are prohibitively expensive. In the future we plan to investigate a bitwise version of convolutive neural networks, where efficient computing is more desirable.

## References

- Baldauf, M., Dustdar, S., and Rosenberg, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, January 2007.
- Bengio, Y. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.
- Courbariaux, M., Bengio, Y., and David, J.-P. Low precision arithmetic for deep learning. *arXiv preprint arXiv:1412.7024*, 2014.
- Cybenko, G. Approximations by superpositions of sigmoidal functions. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.
- Fiesler, E., Choudry, A., and Caulfield, H. J. Weight discretization paradigm for optical neural networks. In *The Hague'90, 12-16 April*, pp. 164–173. International Society for Optics and Photonics, 1990.
- Golea, M., Marchand, M., and Hancock, T. R. On learning  $\mu$ -perceptron networks with binary weights. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 591–598, 1992.
- Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. Maxout networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2013.
- Hinton, G., Deng, L., Yu, D., Dahl, G. E., M., Abdelrahman, Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
- Hinton, G. E., Osindero, S., and Teh, Y. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.
- Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.
- Hwang, K. and Sung, W. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, Oct 2014.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.
- McCulloch, W. S. and Pitts, W. H. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.
- Pitt, L. and Valiant, L. G. Computational limitations on learning from examples. *Journal of the Association for Computing Machinery*, 35:965–984, 1988.
- Salakhutdinov, R. and Hinton, G. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.
- Soudry, D., Hubara, I., and Meir, R. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems (NIPS)*, 2014.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, January 2014.
- Xu, Y., Du, J., Dai, L.-R., and Lee, C.-H. An experimental study on speech enhancement based on deep neural networks. *IEEE Signal Processing Letters*, 21(1):65–68, 2014.