Sapienza University of Rome

Neural Networks
Binarized Neural Networks

Ivan Bergonzani, Michele Cipriano

July 4, 2019

1 Introduction

The aim of the project is to implement a binarized neural network (BNNs), introduced in [1], and test it on the MNIST and the CIFAR-10 datasets. This new method introduces binary weights and activation functions in order to improve memory usage and replace arithmetic operations with bitwise operations.

The experiments achieved an accuracy on **0.961** on MNIST and an accuracy of **0.786** on CIFAR-10 using a BNN with a shift-based Batch Normalization, and a shift-based AdaMax optimizer. These results are in line with the results obtained in the original paper.

The project has been entirely developed in Python and TensorFlow with the exception of the GPU kernel for the matrix multiplication that has been developed in C++ and CUDA.

2 BNNs

The main characteristic of the binarized neural networks is that the weights and the activations are constrained to +1 and -1. This makes it possible to introduce an optimized usage of the hardware resources when dealing with memory management and operations on tensors. The activation function used in this project is the Sign function, defined as:

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

Of course, since the derivative of the Sign function is zero everywhere except at the discontinuity points, it will not be possible to update the weights during backpropagation. A method to solve this issue is to use the straight-through

estimator. Let's consider a cost function C that depends on a variable q defined as:

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

when computing the partial derivative of C with respect to r, using the chain rule:

$$\frac{\partial C}{\partial r} = \frac{\partial C}{\partial q}\frac{\partial q}{\partial r},$$

the straight-through estimator used in the paper simply states that, given an estimator g_q of $\frac{\partial C}{\partial q}$, the estimator g_r of $\frac{\partial C}{\partial r}$ is:

$$g_r = g_q 1_{|r|\leq 1},$$

in this way it is possible to both preserve the information of the gradient and to avoid gradient explosion when r becomes too large. Note that this estimation can be obtained when propagating the gradient through the Clip function:

$$\text{Clip}(r, -1, 1) = \max(-1, \min(1, r))$$
$$\frac{\partial \text{Clip}(r, -1, 1)}{\partial r} = \begin{cases} 1 & \text{if } |r| \leq 1 \\ 0 & \text{otherwise} \end{cases} = 1_{|r|\leq 1}$$

Hence, weights are, first, projected to values between -1 and 1 with the Clip function and are, then, binarized with the Sign function.

Batch Normalization[2] helps to accelerate training by reducing internal covariate shift. This can be obtained by whitening the distribution of the batches after each activation function of the network. The training time can be further improved by approximating the multiplication operations with shifting operations, using, hence, shift-based Batch Normalization (Algorithm 1). The algorithm gets as input a batch of size m, computes its mean (line 2), its variance (line 4), it normalizes (line 5), scales and shift (line 6) the distribution along each dimension and returns the new minibatch y.

Algorithm 1 Shift-based Batch Normalization[1].

1 Let x be minibatch of size m
2 $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
3 $C(x_i) \leftarrow x_i - \mu_B$
4 $\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(C(x_i) \ll\gg_P C(x_i))$
5 $\hat{x}_i \leftarrow C(x_i) \ll\gg_P (\sqrt{\sigma_B^2 + \epsilon})^{-1}$
6 $y_i \leftarrow \gamma \ll\gg_P \hat{x}_i$

Note that the operator $\ll\gg_P$ approximates the multiplication and it is defined as:

$$\alpha \ll\gg_P \beta = \begin{cases} \alpha \ll +\text{round}(\log2|\beta|) & \text{if } \log2|\beta| > 0 \\ \alpha \gg -\text{round}(\log2|\beta|) & \text{otherwise} \end{cases}$$

similary, the operator $\ll\gg_D$ approximates the division and it is defined as:

$$\alpha \ll\gg_D \beta = \begin{cases} \alpha \gg +\text{round}(\log2|\beta|) & \text{if } \log2|\beta| > 0 \\ \alpha \ll -\text{round}(\log2|\beta|) & \text{otherwise} \end{cases}$$

Of course, implementing these operators via hardware would make the computation more efficient, reducing the clock cycles needed to complete algorithm. Note that in our implementation, the operators $\ll\gg_P$ and $\ll\gg_D$ have been computed by using floating point multiplications, divisions and the approximate power of 2, defined as $AP2(x) = \text{sgn}(x) \times 2^{\text{round}(\log 2|x|)}$.

A shift-based AdaMax, based on the AdaMax implementation introduced in [3], has also been implemented in order to reduce the total amount of multiplications (Algorithm 2). The algorithm, which here shows the parameters update for each cycle, computes the gradient at line 3, updates the first moment estimate at line 4, updates the exponentially weighted infinity norm at line 5 and then updates the parameters at line 6, using the operators defined before.

Algorithm 2 Shift-based AdaMax learning rule[1].

1 Let t be the current timestep, θ_{t-1} the parameters of the previous timestep, $f(\cdot)$ the objective function, $\beta_1, \beta_2 \in [0, 1)$ exponential decay rates
2 $t \leftarrow t + 1$
3 $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
4 $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
5 $u_t \leftarrow \max(\beta_2 \cdot u_{t+1}, |g_t|)$
6 $\theta_t \leftarrow \theta_{t-1} - (\alpha \ll\gg_D (1 - \beta_1^t)) \cdot m_t \ll\gg_D u_t$

In a binarized neural network, the binarized values of the weights and the activations are used in the forward pass. Here, each activation is whitened through Batch Normalization between each layer. Note that, when training on CIFAR-10, the application of the Batch Normalization must keep the convolutional property, hence, different elements of the same feature map are normalized in the same way.

3 Experiments

The model has been tested on the datasets MNIST and CIFAR-10 comparing three different DNNs:

- A standard deep neural network with ReLU nonlinearity, the original implementation of Batch Normalization (`tf.layers.batch_normalization`) and a vanilla implementation of ADAM (`tf.optimizers.AdamOptimizer`) as optimizer
- A binarized neural network with the Sign function for nonlinearity, the original implementation of Batch Normalization and a vanilla implementation of ADAM (`tf.optimizers.AdamOptimizer`) as optimizer
- A binarized neural network with the Sign function for nonlinearity, a shifted-based implementation of Batch Normalization and a shifted-based implementation of AdaMax

On the dataset MNIST, the networks have been trained for 20 epochs with a batch size of 100, while, on the dataset CIFAR-10, the networks have been trained for 50 epochs with a batch size of 50. For both datasets the learning rate has been initially set to 0.001 and divided by 2 after each 10 epochs (which

Network	Batch Norm.	Dataset	Train Acc.	Test. Acc.
Standard	Standard	MNIST	1.000	0.984
Binarized	Standard	MNIST	0.999	0.984
Binarized	Shift-based	MNIST	0.965	0.961
Standard	Standard	CIFAR-10	1.000	0.884
Binarized	Standard	CIFAR-10	0.999	0.784
Binarized	Shift-based	CIFAR-10	0.897	0.786

Table 3.1: The table shows the results of the experiments. A network can either be "Standard" or "Binarized", in the latter case the weights and the activation functions are binarized. Similarly, Batch Normalization can be "Standard" if implemented as in the original paper[2] or "Shift-based" if it makes use of shift operators in order to exploit hardware implementations. Binarized networks with shift-based batch normalization makes use of a shift-based implementation of AdaMax as optimizer, while all the other experiments use a vanilla implementation of ADAM.

could be obtained by shifting the bits to the right by 1). All the experiments used the cross entropy as loss function.

As it is possible to see from table 3.1, on the dataset MNIST, the networks achieved an accuracy of **0.984** on the test set when using the ReLU activation function, the Batch Normalization implementation of TensorFlow and the ADAM optimizer, **0.984** when using the Sign activation function, the Batch Normalization implementation of TensorFlow and the ADAM optimizer, and **0.961** when using the Sign activation function, and a binarized implementation of both Batch Normalization and AdaMax optimizer. On the dataset CIFAR-10, the networks achieved an accuracy of **0.884** on the test set when using the ReLU activation function, the Batch Normalization implementation of TensorFlow and the ADAM optimizer, **0.784** when using the Sign activation function, the Batch Normalization implementation of TensorFlow and the ADAM optimizer, and **0.786** when using the Sign activation function, and a binarized implementation of both Batch Normalization and AdaMax optimizer.

The experiments have been performed on Google Compute Engine using an NVIDIA Tesla K80 to speed up the training.

The forward pass of a binarized neural network could be further optimized by using a kernel that exploits operators that are already implemented on hardware. In our implementation a GPU kernel for the matrix multiplication has been developed in order to test how the use of XNOR operator could speed up the computation with respect to the original TensorFlow implementation of `tf.matmul`. Since, in a BNN, the matrices which represent the weights and activations are binary, it is possible to store the single values into bits by concatenating the rows of the left operand and the columns of the right operand into integers. In this way it is possible to group 32/64 elements into the 32/64 bit of a single integer. Once the two matrices have been obtained, the rows-columns multiplication problem is reduced to a compositions of the bitwise operations

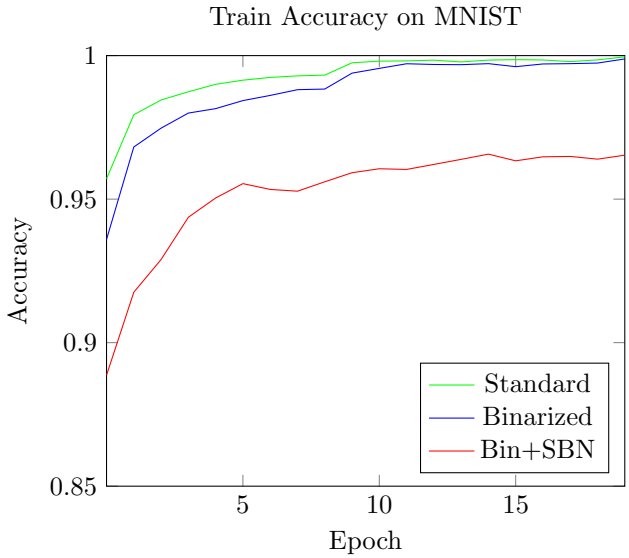

Figure 3.1: The plot shows the how the train accuracy varies for the dataset MNIST using standard activations, binary activations and binary activations with shift-based Batch Normalization.

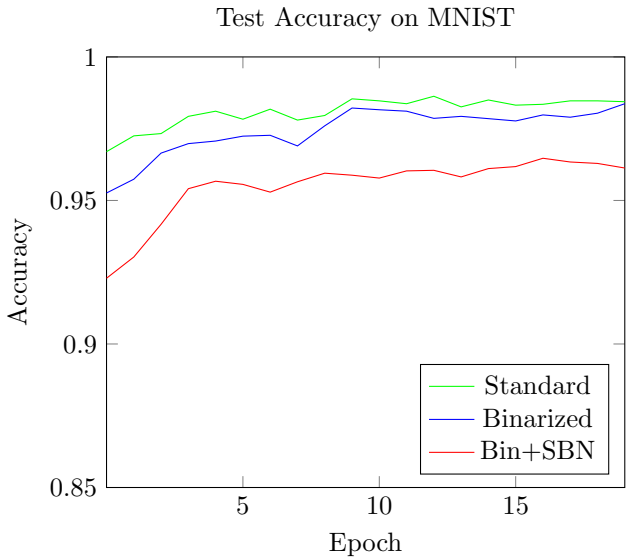

Figure 3.2: The plot shows the how the test accuracy varies for the dataset MNIST using standard activations, binary activations and binary activations with shift-based Batch Normalization.

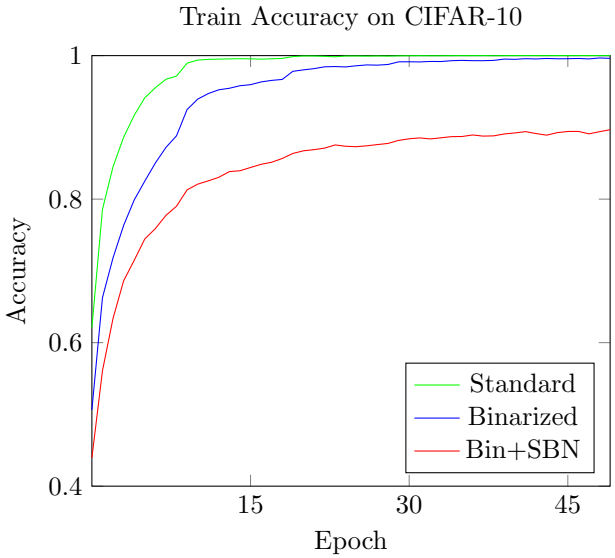

Figure 3.3: The plot shows the how the train accuracy varies for the dataset CIFAR-10 using standard activations, binary activations and binary activations with shift-based Batch Normalization.

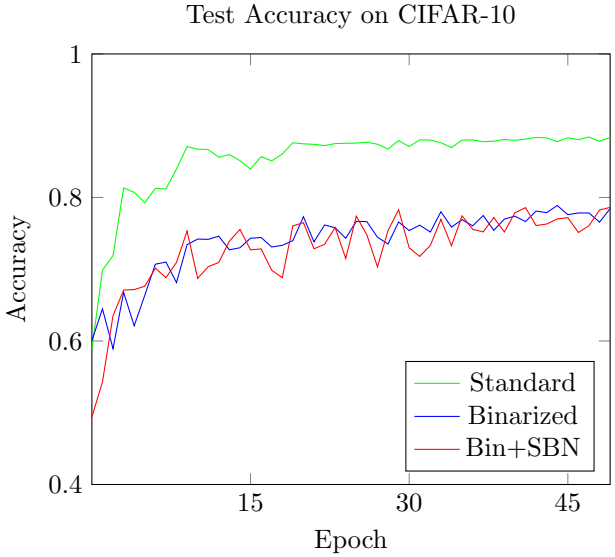

Figure 3.4: The plot shows the how the test accuracy varies for the dataset CIFAR-10 using standard activations, binary activations and binary activations with shift-based Batch Normalization.

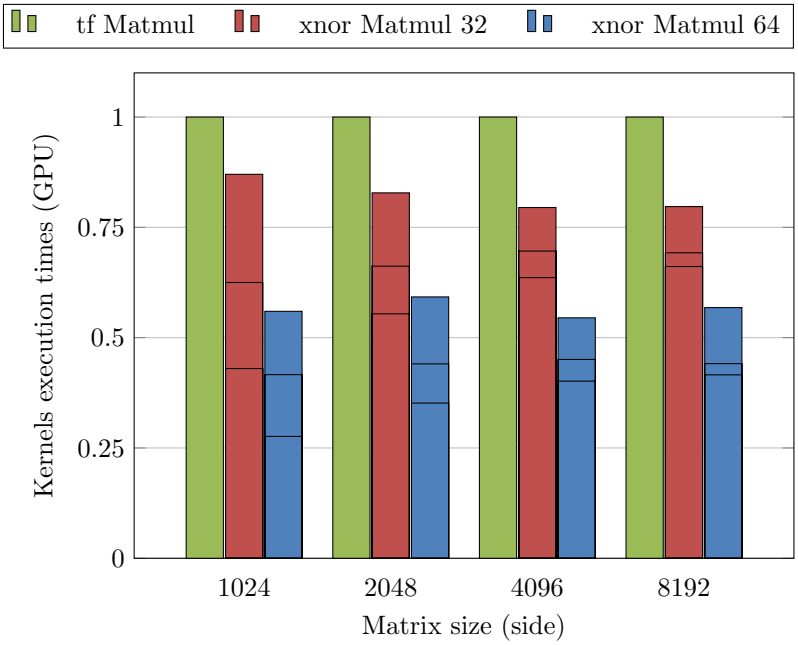

Figure 3.5: The figure shows the relationship between the execution time of `tf.matmul` and our matrix multiplication kernels which use the XNOR operator on 32 and 64 bits. The columns representing these last two kernel are divided in three parts, the concatenate columns operation on the top, the concatenate rows in the middle and the actual matrix multiplication on the bottom.

popcount[1], xnor and an accumulator of their result:

$$b_{cnt} = \sum_{1}^{n} \text{popcount}(\text{xnor}(a_{rj}, w_{jc}))$$

$$y_{rc} = b_{cnt} - (m - b_{cnt})$$

where m is the length of row r and column c. Value n represents instead the number of integer needed to store their binary values.

The matrices concatenations and multiplication kernels have been developed in C++ with the use of CUDA libraries and it have been tested on Google Compute Engine with an NVIDIA Tesla K80.

Figure 3.5 shows the time needed to perform a matrix multiplication with different kernels changing the size of the matrices themselves. Here, it is possible to see that our implementation is faster than the original TensorFlow implementation.

The implemented xnor matmul has been tested also on the forward propagation of a multilayer perceptron with 1024 input nodes, 3 hidden dense layers of 4096 nodes and an output layer of 10 nodes. The hidden dense layers were implemented using the xnor multiplication followed by a sign activation. This network was compared against a similar one which used only the vanilla tensorflow matmul on the evaluation of a dataset with around 60000 entries.

[1]popcount returns the number of bit in a number

	batch size				
kernel	8192	4096	2048	1024	512
`tf.matmul`	2.623	2.504	2.718	2.832	3.252
`xnor32`	2.142	2.137	2.417	2.783	3.785
`xnor64`	1.707	1.687	1.958	2.303	3.288

Table 3.2: The table shows the execution time (in seconds) of different multiplication kernels on the forward propagation over a dataset with around 60000 entries. `tf` is the TensorFlow implementation, `xnor32` is the implementation using our kernel with 32 bits while `xnor64` is the implementation using our kernel with 64 bits.

These results are highlighted on table 3.2 which shows the relationship between the execution times of the different kernels, grouped with respect to the size of the batches.

A further advantage is given by the fact that during inference the weight matrices are fixed and their columns concatenation can be done just once at initialization time.

4 Conclusions

Binarized Neural Networks promise to speed up the training phase by exploiting hardware implementation, which becomes possible because of the use of binary weights and activation functions. Moreover, since multiplications and division are much harder to implement than shifting, BNNs would use less power. Finally, working on bits allows to reduce the amount of memory needed in the networks, resulting in a smaller number of memory accesses which would further speed up the computation and reduce the power consumption.

The implementation of a GPU kernel for the matrix multiplication using the XNOR operator showed how it could be possible to reduce the training time when using binarized values. Notice that this could be improved even more by fusing the hardware instructions altogether.

Our implementation reflects the results presented in the paper, achieving good results in both MNIST and CIFAR-10 datasets. Of course, due to the unavailability of the GPU resources it has not been possible to identically reproduce the experiments of the authors.

References

[1] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016.

[2] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 448–456, JMLR.org, 2015.

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization.," *CoRR*, vol. abs/1412.6980, 2014.