Binarizing the Convolutional Neural Networks in an End to End Fashion

Abhinav Kumar
u1209853
University of Utah
abhinav3663@gmail.com

Manish Roy
u1145372
University of Utah
manish.roy@utah.edu

Abstract

Fast predictions using CNNs serves as a gateway to real-time deployment on resource constrained devices (particularly on devices which do not have GPUs). This could unleash the potential of the state of the art CNN algorithms on resource-constrained platforms.

Previous works use an approximation of the signum function $= x|x|$ *in the region* $|x| <= 1$ *to get an end to end differentiable network. We would try other differentiable functions which mimic the signum function much more closely instead of the one proposed in the literature to see if some gain can be achieved in the performance.*

Keywords: Binary Nets, Image Classification, Faster Inference

1. Introduction

Convolutional Neural Networks (CNNs) have become the state of the art in several object recognition and object classification tasks [5, 8]. However, using CNNs on resource constrained devices in real-time is limited by the inference time due to huge computational overload such as multiplications in 2-D conv layers. An obvious way to reduce the complexity is to use XNOR networks [7]. In these networks, both weights and inputs to the convolutional layers are binarized. This replaces the traditional multiplications with binary operations and then using a scalar to approximate a floating point multiplication resulting in significant speedup.

1.1. Why is it interesting

The project will also make us dive a bit deeper into CNNs which mostly will be covered much later in the course.

The project also applies ML algorithms to the field of Computer Vision and therefore is heavily aligned with the research that we both are pursuing.

1.2. Why is it relevant to the class?

Faster inference from CNN has tremendous impact on the usability of CNNs. The model is built on the stacking of perceptrons taught in the class. This project will let us see the limitations which the current MLP models face and also demonstrate one of the interesting solutions to the same.

2. Literature Survey

While Deep Neural Networks (DNNs) have the ability to learn a hierarchy of features in the multiple layers, which is transforming applications such as image classification, speech recognition/enhancement etc. albeit posing new challenges to accommodate for relatively bigger networks having large number of parameters. Thus the gravity of the research that focused on training networks whose weights can be transformed into some quantized while avoiding loss of performance as much as possible. The resource crunch is at the motivation that drives the idea of binarization and allied techniques.

Since the factor of limited hardware resource is crucial while training deep neural networks, the apt use of memory and arithmetic operators becomes essential. Among the various arithmetic operators, multipliers consume a lot of space and energy and thus are the obvious areas where we can aim for optimization. The work in [2] shows the use of low precision multipliers for training Deep Neural Networks (DNNs), and hence can be used to for optimized memory usage and power efficient hardware that can support deep learning. The Bit wise neural networks [4] presents a bitwise representation of NNs using a bitwise representation of inputs, weights, biases, hidden units and outputs and the operations use simple bitwise logic.

The training and application of deep networks essentially involves the multiplication of real valued weights with a real valued activation. The idea of binarizing by forcing the weights to hold binary values, for the forward or backward propagation has been affirmed using the BinaryConnect [3]. The idea of making the discretization stochastic is quite compatible with Stochastic Gradient Descent (SGD),

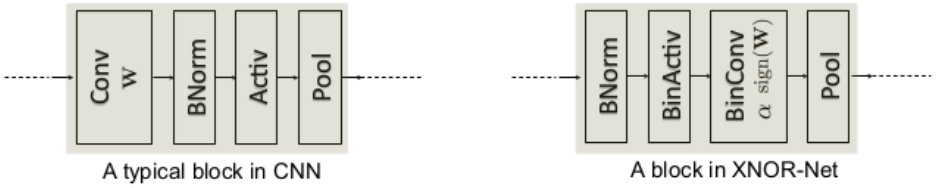


Figure 1: Block structure in the XNOR-Network (right) vs Block structure in a typical CNN (left) [7]

the optimization algorithm used for training deep neural networks.

2.1. Optimal Binary Weights for minimum loss

In order to constrain a convolutional neural network $\langle \mathcal{I}, \mathcal{W}, * \rangle$ to have binary weights, we estimate the real-value weight filter $\mathbf{W} \in \mathcal{W}$ using a binary filter $\mathbf{B} \in \{+1, -1\}^{c \times w \times h}$ and a scaling factor $\alpha \in \mathbb{R}^+$ such that $\mathbf{W} \approx \alpha \mathbf{B}$.

A convolutional operation can be approximated by:

$$\mathbf{I} * \mathbf{W} \approx (\mathbf{I} \oplus \mathbf{B})\, \alpha \tag{1}$$

where, $\oplus$ indicates a convolution without any multiplication. Since the weight values are binary, we can implement the convolution with additions and subtractions.

We represent a CNN with binary weights by $\langle \mathcal{I}, \mathcal{B}, \mathcal{A}, \oplus \rangle$, where $\mathcal{B}$ is a set of binary tensors and $\mathcal{A}$ is a set of positive real scalars, such that $\mathbf{B} = \mathcal{B}_{lk}$ is a binary filter and $\alpha = \mathcal{A}_{lk}$ is an scaling factor and $\mathcal{W}_{lk} \approx \mathcal{A}_{lk}\mathcal{B}_{lk}$

Without loss of generality we assume $\mathbf{W}, \mathbf{B}$ are vectors in $\mathbb{R}^n$, where $n = c \times w \times h$. To find an optimal estimation for $\mathbf{W} \approx \alpha \mathbf{B}$, we solve the following optimization:

$$\begin{aligned} J(\mathbf{B}, \alpha) &= \|\mathbf{W} - \alpha \mathbf{B}\|^2 \\ \alpha^*, \mathbf{B}^* &= \arg\min_{\alpha, \mathbf{B}} J(\mathbf{B}, \alpha) \end{aligned} \tag{2}$$

by expanding equation 2, we have

$$J(\mathbf{B}, \alpha) = \alpha^2 \mathbf{B}^T \mathbf{B} - 2\alpha \mathbf{W}^T \mathbf{B} + \mathbf{W}^T \mathbf{W} \tag{3}$$

since $\mathbf{B} \in \{+1, -1\}^n$, $\mathbf{B}^T \mathbf{B} = n$ is a constant . $\mathbf{W}^T \mathbf{W}$ is also a constant because $\mathbf{W}$ is a known variable. Lets define $\mathbf{c} = \mathbf{W}^T \mathbf{W}$. Now, we can rewrite the equation 3 as follow: $J(\mathbf{B}, \alpha) = \alpha^2 n - 2\alpha \mathbf{W}^T \mathbf{B} + \mathbf{c}$. The optimal solution for $\mathbf{B}$ can be achieved by maximizing the following constrained optimization: (note that α is a positive value in equation 2, therefore it can be ignored in the maximization)

$$\mathbf{B}^* = \arg\max_{\mathbf{B}} \mathbf{W}^T \mathbf{B} \quad s.t. \;\; \mathbf{B} \in \{+1, -1\}^n \tag{4}$$

This optimization can be solved by assigning $\mathbf{B}_i = +1$ if $\mathbf{W}_i \geq 0$ and $\mathbf{B}_i = -1$ if $\mathbf{W}_i < 0$, therefore the optimal solution is $\mathbf{B}^* = sign(\mathbf{B})$. In order to find the optimal value for the scaling factor α^*, we take the derivative of J with respect to α and set it to zero:

$$\alpha^* = \frac{\mathbf{W}^T \mathbf{B}^*}{n} \tag{5}$$

By replacing $\mathbf{B}^*$ with $sign(\mathbf{W})$

$$\alpha^* = \frac{\mathbf{W}^T sign(\mathbf{W})}{n} = \frac{\sum |\mathbf{W}_i|}{n} = \frac{1}{n} \|\mathbf{W}\|_{\ell 1} \tag{6}$$

therefore, the optimal estimation of a binary weight filter can be simply achieved by taking the sign of weight values. The optimal scaling factor is the average of absolute weight values.

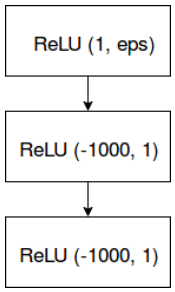
2.2. Changes in Architecture

A typical block in CNN contains several different layers which is shown in Figure 1 (left). This block has four layers in the following order: 1-Convolutional, 2-Batch Normalization, 3-Activation and 4-Pooling. Batch Normalization layer normalizes the input batch by its mean and variance. The activation is an element-wise non-linear function (e.g. ReLU). The pooling layer applies any type of pooling (e.g.,max) on the input batch.

[7] also suggest changes in the order of the blocks to obtain better quantized weights. Applying pooling on binary input results in significant loss of information. Therefore, they put pooling layer put after the convolution. To further decrease the information loss due to binarization, the inputs are normalized before binarization. This ensures the data to hold zero mean, therefore, thresholding at zero leads to less quantization error. The order of layers in a block of binary CNN or XORNet is shown in figure 1 (right).

3. Proposed end to end binary network

The basic problem of [7] is that it uses different updates for forward and backward pass because of the explicit binary operation being carried out. We eliminate this by using



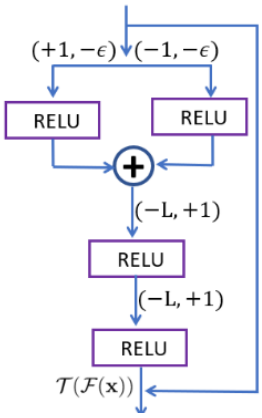


Figure 2: (a) Our thresholder block made of 3 ReLU. The value of eps is taken to be a small value of $1e-6$ in our implementation. (b) Sparsity Controller of [9] which outputs 1 if the absolute value of input is $> \epsilon$ and 0 if the absolute value of input is $< \epsilon$.

a block of 3 ReLUs to threshold the weights and make the network end to end differentiable. The block of 3 ReLUs is shown in figure 2 which is similar to the one proposed in [9].

We seed the original network with random weights and use the 3 ReLU thresholder block to binarize the weights in training itself. This makes the network end to end differentiable without worrying about the binarization explicitly. The process of achieving the binarization is shown in figure 3.

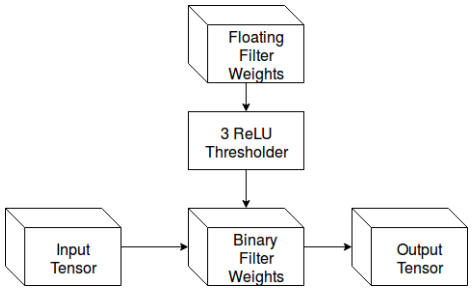


Figure 3: Proposed binarization scheme of filter weights because of the differentiable thresholder blocks. At inference time, we only need to keep the binary weights and we can throw the floating point weights.

4. Experiments

We use the publicly available MNIST dataset for all our experiments. MNIST is a standard digit classification dataset with $50,000$ train images and $10,000$ test images. The size of each image is 28×28. The prime reason for using smaller dataset and smaller architectures first is to see if our initial implementation is correct. Also, this also allows us to train the model on smaller GPUs.

The architecture used for running these experiments is LeNet which is shown in figure 4.

We carried out all our experiments in Pytorch [6] on an 12 GB Nvidia Titan-XP GPU.

5. Results and Discussions

We compare our results with the standard LeNet, the one obtained by binarizing the weights and the one which we proposed. These are shown in table 1

Model	**Accuracy** (%)
LeNet (Full precision)	99.34
LeNet (Binarized Weights) [1]	99.23
LeNet (Binarized Weights + inputs) (End to end binarized)	**98.64**

Table 1: Accuracy Comparison of different networks on MNIST.

The validation accuracy and validation loss of training end to end binarized network is shown in figure 5.

The accuracy shown in table 1 shows that the full-precision network achieves the highest accuracy. Upon binarizing the weights, the output drops slightly. Our implementation binarizes both the weights and the inputs and therefore achieves slightly lower accuracies as expected.

In addition, we also visualise the binary filters in the layer2 learnt by our end to end trainable model in figure 6. We only show the outputs corresponding to one of the output channel.

6. Conclusions and Future Work

The end to end trainable network allows us to obtain the binary weights without storing binary weights or even ex-

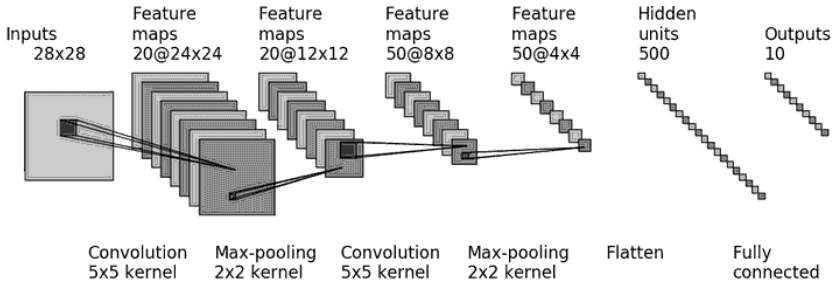


Figure 4: LeNet architecture. Image Courtesy - `https://www.researchgate.net/figure/Structure-of-LeNet-5_fig1_312170477`

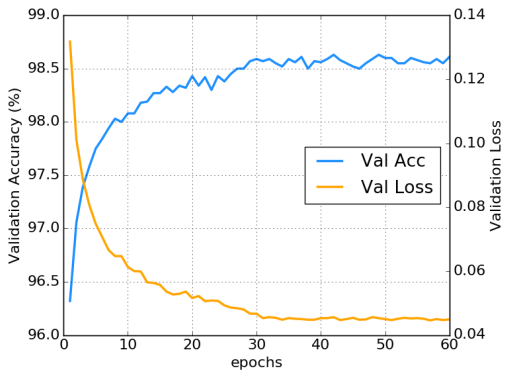


Figure 5: Plot of validation accuracy and validation loss with epochs using end to end binarized network on MNIST.

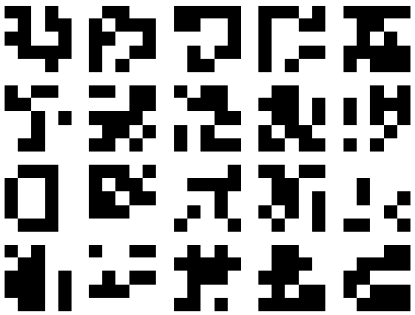


Figure 6: Visualisation of 20 5×5 binary filters learnt by our end to end trainable model.

plicit thresholding the weights using an approximation of the thresholding operation. We achieve almost the same accuracy even after binarizing the inputs as well as the convolution weights.

The drivers of the machine got broke and therefore we could not run our experiments on CIFAR-10. Future Works would include running the end to end binary networks on bigger networks such as AlexNet [5] and VGGNet [8] and on bigger datasets such as ImageNet. Also the faster inference could not be tested in Pytorch since this library doesnot support bitwise operations and does not have bit tensors.

Acknowledgements

The authors would like to thank Prof Vivek Srikumar from the School of Computing, University of Utah for his guidance and support.

References

[1] Xnor-net-pytorch: Pytorch implementation of xnor-net. `https://github.com/jiecaoyu/XNOR-Net-PyTorch`. Accessed: 2018-12-13.

[2] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications, 2014.

[3] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations, 2015.

[4] M. Kim and P. Smaragdis. Bitwise neural networks, 2016.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[6] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[7] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[8] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[9] X. Yu, Z. Yu, and S. Ramalingam. Learning strict identity mappings in deep residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4432–4440, 2018.