NullaNet: Training Deep Neural Networks for Reduced-Memory-Access Inference

Mahdi Nazemi  
University of Southern California  
Los Angeles, California, USA  
mnazemi@usc.edu

Ghasem Pasandi  
University of Southern California  
Los Angeles, California, USA  
pasandi@usc.edu

Massoud Pedram  
University of Southern California  
Los Angeles, California, USA  
pedram@usc.edu

ABSTRACT

Deep neural networks have been successfully deployed in a wide variety of applications including computer vision and speech recognition. However, computational and storage complexity of these models has forced the majority of computations to be performed on high-end computing platforms or on the cloud. To cope with computational and storage complexity of these models, this paper presents a training method that enables a radically different approach for realization of deep neural networks through Boolean logic minimization. The aforementioned realization completely removes the energy-hungry step of accessing memory for obtaining model parameters, consumes about two orders of magnitude fewer computing resources compared to realizations that use floating-point operations, and has a substantially lower latency.

1 INTRODUCTION

Deep neural networks (DNNs) have proven successful in a wide variety of applications such as speech recognition and synthesis, computer vision, machine translation, and game playing, to name but a few. To sustain the ubiquitous deployment of deep learning models and cope with their computational and storage complexity, several solutions like weight pruning, knowledge distillation, tensor decomposition, and quantization have been proposed. The ultimate goal of these methods is to reduce the cost of computations and/or memory accesses without affecting classification accuracy significantly.

To demonstrate the cost associated with computations and memory accesses, Table 1 summarizes latency values for integer arithmetic operations and accesses to different levels of the memory hierarchy in Intel Haswell architecture [1, 2]. The latency values for Skylake architecture are close to the ones reported in Table 1 [3]. It can be observed that accessing different levels of memory hierarchy is 4× to 400× slower than integer operations. This slow operation of memories is a major bottleneck for deep neural networks not only because they have millions of parameters that need to be read from memory, but also because of frequent reading and writing of intermediate values of arithmetic operations.

Table 1. Latency values for 32-bit integer operations and memory accesses in Intel Haswell architecture

| Integer Operation | # of Operations | Latency (Clock Cycle) |
|-------------------|-----------------|-----------------------|
| Add               | 12              | 1                     |
| Multiply          | 4               | 1                     |

| Memory            | Size (KBytes)  | Latency (Clock Cycle) |
|-------------------|----------------|-----------------------|
| L1 Data Cache     | 32             | 4 – 5                 |
| L2 Cache          | 256            | 12                    |
| L3 Cache          | 8192           | 36 – 58               |
| DRAM              | –              | 230 – 422             |

Table 2 compares different types of arithmetic operations and memory accesses from an energy consumption point of view [4]. It can be observed that energy consumption of accessing level-1 cache is about 5× that of half-precision floating-point multiplication. Moreover, accessing DRAM consumes about 300× – 600× more energy compared to half-precision floating-point multiplication. This high energy consumption of memory elements necessitates designing memory-efficient methods for implementing deep neural networks, especially for energy-constrained platforms such as smartphones.

Table 2. Energy cost of various arithmetic operations and memory accesses in 45nm technology

| Arithmetic Operation | Bit Width | Energy (pJ) | Normalized Energy (per bit) |
|----------------------|-----------|-------------|------------------------------|
| Integer Add          | 32        | 0.1         | 1                            |
| Integer Multiply     | 32        | 3.1         | 31                           |
| Float Add            | 16        | 0.4         | 8                            |
| Float Add            | 32        | 0.9         | 9                            |
| Float Multiply       | 16        | 1.1         | 22                           |
| Float Multiply       | 32        | 3.7         | 37                           |

| Memory               | Bit Width | Energy (pJ) | Normalized Energy (per bit) |
|----------------------|-----------|-------------|------------------------------|
| L1 Data Cache        | 64        | 20          | 100                          |
| DRAM                 | 64        | 1,300 – 2,600 | 6,500 – 13,000               |

This paper presents a method for realizing deep neural networks that requires no memory accesses for reading model parameters. In this method, different layers of a deep neural network are trained to have binary input/output activations and floating-point weights. Layers that are trained this way can be treated as multi-input multi-output Boolean functions, which can be optimized and implemented efficiently using Boolean logic optimization algorithms. One of the major advantages of the present work compared to binarized neural networks is that it does not quantize weights and biases to any specific values, and therefore, does not affect the classification accuracy as much. In fact, similar to regular deep neural networks, weights and biases can assume any value. Such realization of neural networks reduces computational cost, power consumption, and latency by a large margin compared to floating-point or fixed-point implementations.

The remainder of this paper is organized as follows. Section 2 reviews the related work and details the shortcomings of some of the prior work compared to our proposed method. Section 3 describes methodology. After that, Section 4 presents experimental results and finally, Section 5 concludes the paper.
2 RELATED WORK

There has been a considerable amount of work on ameliorating computational and/or memory cost of deep neural networks. The majority of work that tackle computational and/or memory cost of DNNs can be classified into two categories: the ones that introduce new methods for training DNNs such that they can be realized more efficiently, and those that accelerate networks trained using existing methods by introducing a paradigm shift in how computations are performed. Weight pruning, knowledge distillation, tensor decomposition, and quantization are among successful methods for improving computational and/or memory cost of realizing DNNs.

The idea of weight pruning is to remove weights of a DNN that are below a threshold, and optionally, retrain the reduced network to compensate for accuracy degradation [5]. This process can be repeated a few times to find a more compact DNN. The compact DNN has fewer parameters compared to the original network, and therefore, requires fewer computations and less storage. Han et al. [6] designed a specialized hardware that works directly on these compact DNNs and were able to achieve speedup and energy savings compared to floating-point-based implementations of regular deep neural networks.

Knowledge distillation methods take a different approach for finding networks that reach an acceptable classification accuracy while they require reasonably fewer computations and accesses to memory [7]. In these methods, a large, complex network (teacher) imparts its knowledge to a simpler network (student). This allows the simple network to achieve a classification accuracy that would be unachievable if it was trained directly on the same dataset.

Quantization methods use representations that are more efficient in terms of computation and/or storage compared to single-precision floating-point representation. This includes representing model parameters, activations, and/or gradients with a different data type such as fixed-point, or a representation that consumes fewer number of bits (e.g., fewer number of bits for representing exponent and mantissa of floating-point representation). Examples of using other data types and/or low-bit-width representations can be found in [8–19].

An extreme case of quantization is representing each weight and activation with a binary value (i.e., a single bit). Hubara et al. [20] introduce binarized neural networks where weights and activations take binary values (i.e. −1 and +1). This type of quantization allows replacing multiplication with a simple XNOR operation (\(\text{XNOR}(-1, -1) = \text{XNOR}(1, 1) = 1\); and \(\text{XNOR}(-1, 1) = \text{XNOR}(1, -1) = -1\)). On the surface, representing each value (i.e. weights and activations) with a single bit achieves 32× memory cost reduction compared to single-precision floating-point representation. However, as we will explain shortly, binarized neural networks require a higher storage for saving model parameters compared to networks that use floating-point representation.

Representing each value with a single bit incurs a high degradation in classification accuracy due to the quantization error. As a result, wider or larger binarized neural networks need to be trained to achieve the same classification accuracy as networks that utilize floating-point values and operations. For example, a binarized neural network with three hidden layers and 4,096 neurons per layer [20] achieves 99.04% classification accuracy on the MNIST dataset [21]. The number of trainable parameters in this network is about 36.80 millions and 4.39 MB storage is required for saving these parameters for inference. On the other hand, a neural network that uses floating-point representation needs two hidden layers with 512 neurons per layer [22] to achieve a 99.13% accuracy on the same dataset. The number of trainable parameters in the latter network is about 303.10 thousands, which is equivalent to 1.03 MB storage requirement for saving model parameters. It can be observed that storage requirement for the binarized network is about 3.8× that of the neural network that uses floating-point values, and that this ratio will be doubled if half-precision floating-point representation is used in the latter network. The number of computations in the binarized network is about 121× that of the latter network, however, each computation is an XNOR operation instead of a floating-point multiplication.

XNOR-Net [23] reduces quantization error of binarized neural networks by effectively implementing \(-w\) and \(+w\) weights instead of \(-1\) and \(+1\) weights. On the other hand, it incurs additional costs in computation, storage, and latency because of the newly introduced floating-point multiplications. One of the major advantages of the present work compared to binarized neural networks and XNOR-Net is that it does not quantize weights and biases to any specific values, and therefore, the classification accuracy is not affected as much. In fact, similar to regular deep neural networks, weights and biases can assume any value. There has been a few attempts at designing specialized hardware for realizing deep neural networks based on binarized neural networks or XNOR-Net. Examples of such implementations can be found in [24–26].

3 METHODOLOGY

This section explains details of a training method which allows the transformation of the DNN realization problem to a Boolean logic optimization problem. Furthermore, it presents how Boolean logic optimization can be applied to a given neural network in order to find an efficient realization in terms of computational resource consumption, memory cost, and latency.

3.1 Training

The specific goal of the training process in this work is to limit activations to binary values, so that each layer can be modeled as a multi-input multi-output Boolean function. To confine activations to binary values, one can apply the sign function as the activation function of a layer. However, the fact that the derivative of the sign function is zero (almost everywhere) prevents the back-propagation algorithm from properly updating model parameters. Bengio et al. [27] have studied estimation of gradients for neurons with hard non-linearities and have shown that a straight-through estimator (STE) [28] achieves the best validation and test error. This work uses the same STE that was presented in [20] and propagates gradients through a hard tanh function

\[ \text{Htanh}(x) = \max(-1, \min(1, x)). \]

Algorithm 1 summarizes different steps of forward propagation for deep neural networks with binary activations \(^{1}\) (steps that are

---

\(^{1}\)When discussing weights, biases, and/or activations in this paper, subscripts are used for indexing layers while superscripts are used for indexing neurons.
Algorithm 1 Forward Propagation

Input:
- \( L \): number of layers
- \( a_0 \): a mini-batch of inputs
- \( W \): weights
- \( \beta \): batch normalization parameters

Output:
- \( a_L \): network’s predictions

1: \( \text{for } i = 1 \text{ to } L \text{ do} \)
2: \( z_i = a_{i-1} W_i \)
3: \( a_i = \text{BatchNorm}(z_i, \beta) \)
4: \( \text{if } i < L \text{ then} \)
5: \( a_i = \text{Sign}(a_i) \)
6: \( \text{end if} \)
7: \( \text{end for} \)
8: return \( a_L \)

3.2 Boolean Logic Optimization

During early developments of neural networks, one of the initial problems was to implement AND, OR, and NOT gates using neurons. The rationale behind this was that because computers can be built using these three gates, implementing them using neurons allows building computers based on neural networks \cite{29}. Fig. 1 illustrates examples of logic gates implemented using McCulloch-Pitts neurons. McCulloch-Pitts neurons are defined according to Eq. 1

\[
f = \begin{cases} 
1, & \text{if } \sum a^j \times w^j \geq b \\
0, & \text{otherwise}
\end{cases} \tag{1}
\]

where \( a^j \) and \( w^j \) denote the \( j \)-th input and weight of the neuron, respectively, and \( b \) is the neuron’s bias (or threshold). This section explains a few methods that are capable of doing the reverse operation: taking a neuron as an input and implementing its functionality using primitive gates such as AND, OR, NOT, and XOR.

3.2.1 Realization Based on Input Enumeration. One of the methods that allows inference without storing model parameters is realization based on input enumeration. In this method, all different combinations of a neuron’s inputs are enumerated and the corresponding outputs are found according to the neuron’s weights and bias (Eq. 1). This is in fact equivalent to finding a truth table for each neuron. The truth table implements the same function as the neuron’s function when its output is calculated using Eq. 1.

Given the truth table, one can write the function of each neuron as a sum of product terms (SoP). The SoP for each neuron can be fed to a logic synthesis tool \cite{30, 31}, which implements the function using logic gates and optimizes the function for minimum area, delay, and/or power consumption. In other words, instead of realizing the output of a neuron through calculating the dot product of its inputs and weights, we implement the output using logic gates thorough synthesizing its Boolean expression. Fig. 2 illustrates an example of such realization. The advantage of such implementation is that the function represented using an SoP considers the neuron’s parameters implicitly and allows inference without reading model parameters from memory.

After optimizing all neurons, the neurons of each layer are put together (and optionally, further optimized as a larger block) to find an efficient realization for each layer. Fig. 3 illustrates an example of such optimization. Similarly, the optimized layers are put together to find an efficient realization for the whole network.

The disadvantage of this method is that it can only be applied to neurons with limited number of inputs because the size of truth table grows exponentially with the cardinality of inputs to a neuron. Depthwise separable convolutions are examples of layers that can be implemented using this method. In practice, this solution will be infeasible for neurons with tens or hundreds of inputs. The method explained in the next section addresses this issue.

3.2.2 Realization Based on Incompletely Specified Functions (ISFs). An incompletely specified function is a Boolean function where output values are defined only for a subset of input combinations. The input combinations that cause a logic one in the output constitute the ON-set and the input combinations that cause a logic zero in the output constitute the OFF-set. The input combinations for which the output value is not specified make up the DON’T CARE-set (or DC-set for short).

In this method, instead of enumerating all input combinations for each neuron, we only evaluate outputs of neurons for input combinations derived from samples in the training set and add the remaining input combinations to the DC-set of the neuron. As a
Deep Neural Network Optimization

Algorithm 2

Input:
- \( L \): number of layers
- \( u_i \), \( i = 0, 1, 2, \ldots, L \): number of neurons in layer \( i \)
- \( a_i \), \( i = 0, 1, 2, \ldots, L \): activations at layer \( i \)
  (for all training samples)

Output:
- optimized network

1. for \( i = 2 \) to \( L - 1 \) do
2.   for \( j = 0 \) to \( u_i - 1 \) do
3.     OptimizeNeuron(Inputs(i, j), Outputs(i, j))
4.   end for
5.  OptimizeLayer()
6.  Pythonize()
7. end for
8. network = OptimizeNetwork()
9. return network
implementing the shared logic once and providing its output to all corresponding neurons instead of implementing the logic separately for each neuron. ABC [31] uses algorithms such as rewriting [35], balancing, and refactoring [36] to optimize combinational designs. Combinational synthesis algorithms are capable of performing a variety of other advanced optimizations [37] that are not discussed here.

`Pythonize()` is a step that converts optimized Boolean expression of a layer into Python code that can be run on CPUs and GPUs. This not only allows studying the effect of using ISFs on classification accuracy, but also enables more efficient inference on the aforementioned platforms. Because the output of this step is a layer implemented using primitive gates such as AND, OR, NOT, and XOR, the layer can be implemented efficiently on different computing platforms.

`OptimizeNetwork()` is the last (optional) step of optimization that implements pipelining to increase the throughput of the design. Because each optimized layer is realized using a combinational logic, realization of the whole network will have a large combinatorial delay. Pipelining is a well-known method that breaks a large combinational design into smaller parts with lower delays. One may use both macro-pipelining and micro-pipelining to increase the throughput. A stage of the macro-pipeline includes a group of consecutive layers of the network while a stage in the micro-pipeline includes necessary logic for implementing those layers. Combinational synthesis algorithms are applied to layers that lie within a macro-pipeline stage in order to further optimize those layers across their boundaries (i.e. inputs and outputs).

After optimizing layers with binary input and output activations using Algorithm 2, the first and last layers need to be optimized to achieve an overall acceptable performance. One way of optimizing these two layers is to use fixed-point quantization to improve resource consumption and energy-efficiency. A second method of optimization, which can be used in conjunction with the first method, is to store the parameters associated with these layers in a low-latency, low-power memory such as L1 data cache. Because these layers are the only ones that need to access weights and biases, small, low-cost memories can store their parameters efficiently.

We should note that the amount of computations in the first layer is typically much smaller than the rest of layers in a deep neural network and therefore, the performance of this layer will have little impact on the network’s overall performance. It should also be noted that for the last layer, because all inputs are pseudo-Boolean variables, dot product of inputs and weights is replaced with additions and subtractions, which are more efficient operations than MACs used in dot product calculation.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

4.1.1 Dataset. In order to demonstrate the applicability of the proposed method in practice, we optimize two different neural networks for the MNIST dataset of handwritten digits [21]. This dataset includes 60,000 samples for training and 10,000 samples for testing, where each sample is a 28×28 grayscale image. The last 10,000 samples of the training set are used as validation set for model selection. The objective is to classify each image into one of ten classes 0–9.

4.1.2 Training Procedure. All networks presented in this section are trained for 100 epochs, with a mini-batch size of 64, and with dropout. For all networks, negative log likelihood loss is minimized using Adamax [38] optimization method. The learning rate is initially set to 0.003 and is gradually decreased during training.

4.1.3 Hardware Setup. The target platform is an Intel Arria 10 GT 1150 FPGA, which includes 427,200 adaptive logic modules (ALMs), 55,562,240 bits of block RAM, and 1,518 DSP blocks. Table 3 summarizes resource utilization, clock frequency, latency, and power consumption for implementing single-precision (i.e. 32-bit) and half-precision (i.e. 16-bit) floating-point adders, multipliers, and unfused MACs on the target FPGA. All reported values are found after placement and routing. The Verilog designs for these operations are based on [39]. Both adders and multipliers are pipelined and have four and two pipeline stages, respectively. MACs are implemented using these adders and multipliers and have six pipeline stages. The adders, multipliers, and MACs are realized using ALMs instead of DSPs to allow comparison of our work with designs that are based on floating-point operations.

| Operation     | ALMs  | Registers (bits) | Clock Frequency (MHz) | Latency (ns) | Power (mW) |
|---------------|-------|------------------|-----------------------|--------------|------------|
| Add (16)      | 115   | 120              | 393.08                | 10.18        | 64.46      |
| Multiply (16)| 86    | 56               | 263.85                | 7.58         | 57.79      |
| MAC (16)      | 195   | 191              | 281.37                | 21.32        | 68.25      |
| Add (32)      | 253   | 247              | 295.77                | 13.52        | 81.05      |
| Multiply (32)| 302   | 101              | 181.00                | 11.05        | 80.77      |
| MAC (32)      | 541   | 377              | 173.01                | 34.68        | 107.87     |

When calculating the cost associated with layers that use MACs, we take account of four accesses to the memory: three for reading activation, weight, and previous partial result, and one for writing the updated partial result. Note that when an activation is a binary value, only a single bit has to be read from the memory.

4.2 Results & Discussion

4.2.1 Multi-Layer Perceptron. The first neural network studied in this section is an MLP with three hidden layers and 100 neurons per layer (Net 1.1). The sign non-linearity is applied to all hidden layers as activation function according to Algorithm 1 (Net 1.1.a). The second and third hidden layers have binary input and output activations and have been optimized using Algorithm 2 (Net 1.1.b). Each of these layers is considered as a macro-pipeline stage and is not further micro-pipelined for increasing the throughput. The first and last layers of this network are implemented using floating-point MACs (fixed-point quantization is not applied). As a result, there will be no savings in the first layer. However, as explained earlier, there will be about 25% memory savings in the last layer because the activations are binary values.

The classification accuracy and hardware cost of implementing Net 1.1 is compared to a network with the same architecture, but...
with ReLU activation function. The operations in the latter network are implemented once using single-precision floating-point MACs (Net 1.2) and another time using half-precision floating-point MACs (Net 1.3).

Table 4 compares classification accuracy of the aforementioned networks. It is observed that quantization of activations (Net 1.1.a) has caused a 1.38% accuracy degradation compared to networks trained with ReLU non-linearity. Moreover, it is observed that optimizing the second and third hidden layers (Net 1.1.b) has increased the classification accuracy by 0.12% compared to Net 1.1.a.

Table 4. Classification accuracy of different MLPs

| Network   | Net 1.1.a | Net 1.1.b | Net 1.2 | Net 1.3 |
|-----------|-----------|-----------|---------|---------|
| Accuracy (%) | 96.89    | 97.01    | 98.27   | 98.27   |

Table 5 presents the hardware cost of the second and third hidden layers of Net 1.1.b on the target FPGA. Implementing these layers in Net 1.2 and Net 1.3 requires 20,000 MAC operations. Obviously, this high number of operations have to be performed on the target platform during several cycles. However, the optimized implementation presented in Table 5 calculates all outputs in parallel with a latency that is equal to 0.88× latency of a single 32-bit MAC and 1.44× latency of a single 16-bit MAC. The optimized realization consumes 112,173 ALMs, which is about 207× that of a 32-bit MAC and 575× that of a 16-bit MAC. Assuming that all 20,000 MACs in Net 1.2 and Net 1.3 could be realized in parallel, the number of ALMs used in the optimized realization is about 97× and 35× lower than the number of ALMs used for Net 1.2 and Net 1.3, respectively. Last but not least, the optimized representation needs to read/write 400 bits from/to memory while 32-bit and 16-bit MAC-based representations need to read/write 312.5 KB and 156.25 KB from/to memory, respectively. This is equivalent to 6,400× and 3,200× savings in accesses to the memory. These substantial savings in memory accesses lead to enormous latency reduction and energy savings according to Table 1 and Table 2.

Table 5. Hardware realization results for second and third hidden layers of Net 1.1.b

| ALMs | Registers | Clock Frequency | Latency | Power |
|------|-----------|-----------------|---------|-------|
| 112,173 | 302 | 65.3 | 30.63 | 396.46 |

It is needless to say that the overall savings in the network are smaller than the aforementioned values because of the cost associated with the first and last layers. However, as the networks get deeper, the first and last layers will play a less important role in the overall network performance. Table 6 compares the cost of implementing different layers of Net 1.1.b and Net 1.2. To allow comparison of our work with MAC-based implementations, we present the cost associated with the second and third hidden layers in terms of MACs (through dividing the number of ALMs used in these layers by the number of ALMs used in a single MAC). It is observed that Net 1.1.b requires 79.61 k MAC operations and has to read/write 1.21 MB data from/to memory. On the other hand, Net 1.2 requires 99.4 k MAC operations and has to read/write 1.52 MB data from/to memory. This translates into 20% savings in computations and 20% savings in memory accesses for implementing the whole network.

For the other network discussed in this section, we do not explain the layer-by-layer costs in detail and report the savings briefly.

Table 6. Cost of realizing different layers of Net 1.1.b and Net 1.2

| Layer | MACs | Memory (Bytes) |
|-------|------|----------------|
| FC1   | 78,400 | 1,254,400 |
| FC2 + FC3 | 207 | 50 |
| FC4   | 1,000 | 12,125 |
| Total | 79,607 | 1,266,575 |

4.2.2 Convolutional Neural Network (CNN). The second neural network studied in this section is a CNN, where the first convolutional layer implements 3×3 convolutions and has 10 output channels while the second convolutional layer implements 3×3 convolutions and has 20 output channels (Net 2.1). Both convolutional layers are followed by 2×2 max pooling and the sign non-linearity is applied to all hidden layers as activation function according to Algorithm 1 (Net 2.1.a). The second convolutional layer has binary input and output activations and has been optimized using Algorithm 2 (Net 2.1.b). Implementation of first and last layers and pipelining method are similar to Net 1.1.b.

Similar to the previous section, the cost of implementing Net 2.1 is compared with a network with the same architecture, but with ReLU activation function. Net 2.2 and Net 2.3 represent implementations of the latter network using single-precision and half-precision floating-point operations, respectively.

Table 7 compares classification accuracy of aforementioned networks. It is observed that quantization of activations (Net 2.1.a) has caused a 0.79% accuracy degradation compared to networks trained with ReLU non-linearity. In addition, it is observed that using ISFs in Net 2.1.b has decreased classification accuracy by 0.29% compared to the case where outputs are calculated using dot products (Net 2.1.a).

Table 7. Classification accuracy of different CNNs

| Network | Net 2.1.a | Net 2.1.b | Net 2.2 | Net 2.3 |
|---------|-----------|-----------|---------|---------|
| Accuracy (%) | 98.21 | 97.92 | 99.00 | 99.00 |

Table 8 presents the hardware cost of realizing the kernels of the second convolutional layer of Net 2.1.b on the target FPGA. Implementing the convolution on each patch of input in Net 2.2 and Net 2.3 requires 1,800 MAC operations. The optimized implementation presented in Table 8 calculates all outputs in parallel with a latency that is equal to 0.41× latency of a single 32-bit MAC.
and 0.67× latency of a single 16-bit MAC. The optimized realization consumes 15,990 ALMs, which is about 30× that of a 32-bit MAC and 82× that of a 16-bit MAC. Assuming that all 1,800 MACs in Net 2.2 and Net 2.3 could be realized in parallel, the number of ALMs used for realizing the aforementioned kernels is about 60× and 22× lower than the number of ALMs used for Net 2.2 and Net 2.3. Last but not least, the optimized representation needs to read/write 110 bits from/to memory for each patch of the input while 32-bit and 16-bit MAC-based representations need to read/write 28.13 KB and 14.06 KB from/to memory, respectively. This is equivalent to 2095× and 1047× savings in accesses to the memory.

| ALMs | Registers (bits) | Clock Frequency (MHz) | Latency (ns) | Power (mW) |
|------|------------------|-----------------------|--------------|------------|
| 15,990 | 110 | 70.12 | 14.26 | 41.77 |

In this example, Net 2.1.b requires 69.47 k MAC operations and has to read/write 1011.45 KB data from/to memory. On the other hand, Net 2.2 requires 283.64 k MAC operations and has to read/write 4.33 MB data from/to memory. This translates into 76% savings in computations and 77% savings in memory accesses for implementing the whole network.

5 CONCLUSION

In this paper, we presented a method for efficient realization of neural networks through reformulation of the realization problem into a Boolean logic optimization problem. In this method, layers of a neural network are trained to have binary input and output activations. This allows treating each layer as a multi-input multi-output Boolean function, which can be optimized using Boolean logic optimization algorithms. Our experimental results show substantial savings in memory accesses and input consumption.

ACKNOWLEDGEMENTS

Mahdi Nazemi proposed the idea of NULLA Net as well as realizations based on technology mapping and input enumeration. He also performed machine learning experiments and implemented different steps of realization. Ghasem Pasandi proposed the idea of performing offline simulation for extracting the truth table of a neuron and subsequently optimizing its Boolean function.

REFERENCES

[1] Intel 64 and IA-32 architectures optimization reference manual. https://software.intel.com.
[2] Intel Skylake. https://www.7-cpu.com/cpu/Haswell.html.
[3] Intel Skylake. https://www.7-cpu.com/cpu/Skylake.html.
[4] M. Horowitz. 1.1 computing’s energy problem (and what we can do about it). In *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (ISSCC), 2014.
[5] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.
[6] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: efficient inference engine on compressed deep neural network. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.
[7] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.
[8] F. Li, B. Zhang, and B. Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.
[9] G. Venkatesh, E. Nuvotidla, and D. Marr. Accelerating deep convolutional networks using low-precision and sparsity. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.
[10] D. Miyashita, H. Y. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.
[11] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DDeReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.
[12] U. Köster, T. Webb, X. Wang, M. Nasser, A. K. Bansal, W. Constable, O. Elboud, S. Gray, S. Hall, L. Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.
[13] M. Nazemi and M. Pedram. Deploying customized data representation and approximate computing in machine learning applications. *International Symposium on Low Power Electronics and Design (ISLPED)*, 2018.
[14] M. Ghasemzadeh, M. Samragh, and F. Koushanfar. ReNet: Residual binarized neural network. *arXiv preprint arXiv:1711.01243*, 2018.
[15] E. Chung, J. Powers, K. Ortmorov, M. Pampamichael, A. Caustfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, et al. Serving CNNs in real time at datacenter scale with project Brainwave. *IEEE Micro*, 2018.
[16] A. Polino, R. Pascani, and D. Aloisio. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.
[17] J. Achterhold, M. Koheler, A. Schmeni, and T. Genewein. Variational network quantization. 2018.
[18] T. Zhou and T. Kwo. Loss-aware weight quantization of deep networks. *arXiv preprint arXiv:1802.08635*, 2018.
[19] S. Khoram and J. Li. Adaptive quantization of neural networks. 2018.
[20] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and T. Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2016.
[21] Y. LeCun, C. Cortes, and C. J. Burges. MNIST handwritten digit database. AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist, 2010.
[22] Y. Tang. Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239*, 2013.
[23] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, 2016.
[24] R. Andri, L. Caviglil, D. Ross, and L. Benini. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016.
[25] A. A. Bahou, G. Karunaratne, R. Andri, L. Caviglil, and L. Benini. XNORBIN: A 95 TFlops/w hardware accelerator for binary convolutional neural networks. *arXiv preprint arXiv:1803.05849*, 2018.
[26] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Lesng, M. Jahre, and K. V. Siers. Finis: A framework for fast, scalable binarized neural network inference. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2017.
[27] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradi- ents through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.
[28] G. Hinton. Neural networks for machine learning. *Coursera*, 2012.
[29] M. A. Arbib. *Brains, machines, and mathematics*. Springer Science & Business Media, 2002.
[30] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgia, A. Saldanha, H. Savoj, P. R. Stephán, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. 1992.
[31] A. Mishchenko et al. ABC: A system for sequential synthesis and verification. 2007.
[32] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. Logic minimization algorithms for VLSI synthesis. *Springer Science & Business Media*, 1984.
[33] O. Coudert and T. Sasao. Two-level logic minimization. In *Logic Synthesis and Verification*. Springer, 2002.
[34] P. Fiser and H. Kubatová. Two-level Boolean minimizer BOOM-II. In *International Workshop on Boolean Problems (IWSBP)*, 2004.
[35] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006.
[36] L. Hong, S. Rio, and A. Sangiovanni-Vincentelli. Symbolic sequential circuit synthesis. 1990.
[37] G. D. Hachtel and F. Sommen. *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2006.
[38] B. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[39] H. Mao. chisel-float. https://github.com/hmao/chisel-float.