ClosNets: Batchless DNN Training with On-Chip A Priori Sparse Neural Topologies

Mihailo Isakov, Alan Ehret and Michel A. Kinsy
Adaptive and Secure Computing Systems Laboratory
Department of Electrical and Computer Engineering, Boston University
Email: {mihailo, ehretaj, mkinsy}@bu.edu

***Abstract*—The deployment of deep neural network (DNN) models is generally hindered by their training time. DNN training throughput is commonly limited by the fully-connected layers. This is due to their large size and low data reuse. Large batch sizes are often used to mitigate some of the effects. Increasing batch size can however hurt model accuracy, creating a tradeoff between accuracy and efficiency. We tackle the problem of training DNNs in on-chip memory, allowing us to train models without the use of batching. Pruning and quantizing dense layers can greatly reduce network size, allowing models to fit on the chip, but can only be applied after training. We propose a fully-connected but sparse layer that reduces the memory requirements of DNNs without sacrificing accuracy. We replace a dense matrix with a sparse matrix product with a predetermined topology. This allows us to: (1) train significantly smaller networks without a loss in accuracy, and (2) store weights without having to store connection indices. We therefore achieve significant training speedups due to the fast access to on-chip weights, smaller network size, and a reduced amount of computation per epoch.**

***Keywords*-neural network; hardware; acceleration; sparsity;**

I. Introduction

Training deep neural networks (DNN) is both time-consuming and power intensive. While the bulk of the computation in applications like image processing revolves around convolutional layers, this is not the case for many other problems, e.g., text-to-speech processing and machine translation. Fully-connected layers, i.e., dense layers, are more common in applications where there is less spatial correlation in the input data. Consequently, their computation has a much larger memory footprint, more data movement, and a longer processing time using significantly more power [1].

To speed up the training of dense neural networks, we investigate the bottlenecks of accelerating dense layers. To determine if a network is compute or memory bound, we measure the time required for training on one epoch with varying batch sizes. While the number of operations required to train on an epoch is independent of the batch size, the number of times we have to load all weight matrices is equal to the number of batches. Note that in Figure 1 the training time grows inversely with the batch size. We attribute this effect to the system being computationally bound for large batch sizes and memory bound for small batch sizes. To reduce the impact of this memory wall, one can either increase the memory bandwidth or decrease the amount of memory required for training. Here we focus on decreasing the DNN memory requirements.

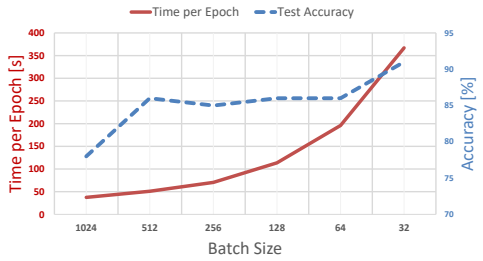

Figure 1: Time per epoch vs. batch size for a CIFAR-10 ResNet18 network trained on a NVidia Titan Xp GPU.

There are several approaches for decreasing the model size of a neural network. Some architectures have been shown to maintain accuracy with a fraction of the original size using smaller layers [2]. However, these works primarily target convolutional neural networks. Low-rank matrix decomposition has been explored as an architectural choice as well, as it allows the decoupling of the number of parameters and the number of neurons in the network. Weight quantization has been explored as a means of decreasing network size for inference [3], and training [4]. Pruning has been explored in several works, with [5] removing the least salient connections, and [1] removing the connections with the smallest magnitude and achieving a $50\times$ decrease in model size.

On the hardware side, training linear models with low-precision has been implemented on FPGAs in [6]. In FINN [7], authors present state-of-the-art results in classification speed using an FPGA binary neural net implementation. In [8], the authors train networks with binary activations in the forward pass and ternary errors in the backward pass, eliminating the need for multipliers.

A majority of the works listed here only speed up inference and not training. In this work, we tackle the challenge of speeding up deep neural network training by decreasing the size of these dense layers. To achieve this we break up the dense matrices into products of sparse matrices. By picking appropriate topologies, we make sure that these products retain full connectivity while requiring fewer parameters. Knowing the topology in advance allows us to store connections efficiently. This is due to (1) the connections being stored serially, which may not be the case for sparse matrices, and (2) not requiring storing indices, since we can calculate sparse matrix indices on-the-fly based on the fixed, known topology.

II. ClosNets

Pruning has been shown to reduce the size of networks by an order of magnitude without sacrificing accuracy [1].

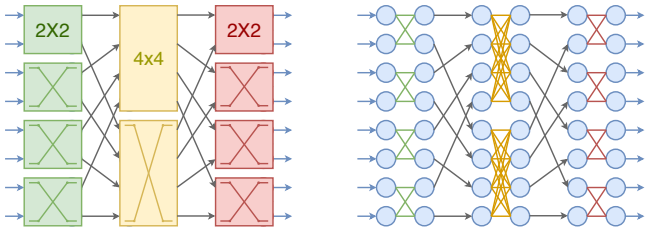

Figure 2: An 8-input, 8-output Clos network with 4 input, 2 middle, and 4 output routers in the networking (**left**) and DNN domain (**right**).

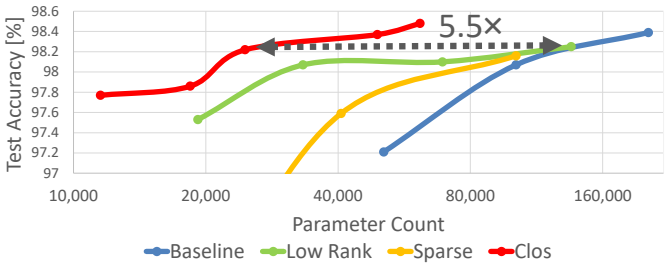

Figure 3: MNIST accuracy vs. parameter count for dense, low-rank, a priori randomly sparse, and Clos networks.

However, current pruning methods only work on trained networks, as there is no way to know which connections will be important before training. If an untrained network is randomly pruned, we risk the possibility of removing connections that may turn out to be important in a later stage. Figure 3 shows that randomly pruned networks suffer from a significantly lower accuracy compared to dense ones. To retain high accuracy, the network must maintain full connectivity between input and output neurons of a layer. To that end, we propose breaking up a large dense layer into a product of multiple sparse layers. This is equivalent to adding intermediate layers between layer's inputs and outputs, and only sparsely connecting the neurons.

We have analyzed a number of common topologies for our sparse layers, such as meshes, toruses, hypercubes, and butterflies. While these topologies do guarantee full connectivity given enough layers, they may cause our networks to become too deep. Indeed, splitting a 1024×1024 dense layer into ten sparse layers with the hypercube topology will result in full connectivity, but the network will be so deep that it will have trouble training. In our experiments, none of the above-listed topologies have been able to train due to the depth of the network and the vanishing gradient issue described in [9].

We therefore search for a topology that offers (1) full connectivity with (2) a small number of intermediate layers. We call the second property shallowness and require that the topology achieve full connectivity with only 1 or 2 middle layers. We also want the topology to have (3) pre-determined and calculable connectivity, i.e. not random, so that we do not have to store indices, but instead calculate them on the fly. Other desirable properties are (4) uniform path diversity, where all neuron input-output pairs have an equal number of paths between them, so that the ordering of inputs does not impact training; (5) high path diversity, where an input has many paths to every output neuron; and (6) an efficient hardware implementation, minimizing data movement and allowing high throughput.

One topology that grants all of these properties is the *Clos network*. A Clos network is a three-stage network in which each stage is composed of a number of crossbar switches [10]. While in the networking domain a Clos network is assumed to have the same number of input and output nodes, we define a more general Clos network as a 5-tuple (I, O, R_i, R_m, R_o). In this characterization, I is the number of inputs, O is the number of outputs, R_i is the number of input routers, R_m is the number of middle routers, and R_o is the number of output routers. In Figure 2 (left) we can see an example (8, 8, 4, 2, 4) Clos network. The intuition behind Clos networks is that since each middle router acts as a crossbar between the input and output routers, there are as many paths between an input-output pair (i, o) as there are middle routers. This gives the network designer a direct way to prevent network contention by increasing the path diversity.

To map Clos networks to the DNN domain, we replace all the input and output nodes with neurons, and each router becomes a smaller fully-connected network with its own hidden neurons, as in Figure 2 (right). Note that the connections between the routers do not have weights, but only permute activations. When converting a conventional dense layer to a network of three layers with the Clos topology, we call this group of layers a *Clos cascade*.

The Clos neural networks or *ClosNets* have clear benefits over the other explored topologies. They are fully-connected and shallow. They have a parametrizable but uniform path diversity. Furthermore, they offer a simple hardware implementation. An added benefit of ClosNets is that they are not restricted to having either (1) an identical number of inputs and outputs, or (2) a number of inputs/outputs equal to a power of two. For a given Clos cascade (I, O, R_i, R_m, R_o), we can calculate the number of parameters P in the network as $P = R_m(I + O + R_i R_o)$, and path diversity D as: $D = R_m$.

We compare model accuracy with respect to the number of parameters for dense (our baseline), low-rank, a priori randomly sparse and ClosNets. Figure 3 shows the test accuracy of different networks trained on MNIST for 50 epochs with respect to their parameter count. As we reduce the parameter count, the networks degrade in accuracy. The performance degradation is more graceful for some networks compared to others. ClosNets have comparable accuracy with the baseline networks while having $5.5\times$ fewer parameters.

III. Architecture

Mapping Clos to a Torus: We propose a simple hardware implementation for our Clos cascade. In Figure 4, the cascade consists of three layers (green, yellow, red). Each layer contains two independent 2×2 dense layers ①. The layers are connected by the grey, weightless connections that permute the inputs to the next layer ②. While we can create an individual processing element (PE) for each neuron in the cascade, this would require more area and would limit the size of networks we can train. Since higher network layers are dependent on the activations of lower layers, we reuse the same PEs to calculate the activations of multiple neurons. In Figure 4, all neurons in each row ③ are mapped to a separate PE ④. Having mapped each

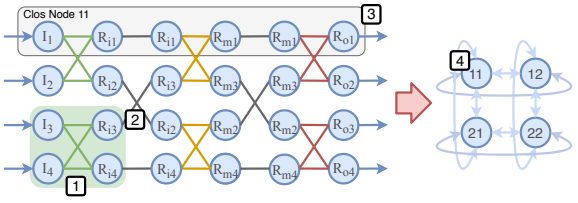

Figure 4: (**left**) A (4, 4, 2, 2, 2) Clos cascade with inputs I, intermediate results R_i and R_m, and outputs R_o. (**right**) A 2×2 torus we map the Clos cascade to. R_{m2} and R_{m3} are swapped so as to allow single cycle hops in Figure 5b.

neuron to a PE, we arrange the PEs in a torus topology. In Figure 4, assigning rows 1 to 4 to nodes 11, 12, 21, and 22 respectively allows all connections to be a single hop away, reducing latency and congestion in the network. With this mapping of neurons to torus nodes, we observe different layer dependencies. The first (green) layer's output neurons (R_{i1} - R_{i4}) only require values from the same torus row. The second (yellow) layer output neurons (R_{m1} - R_{m4}) require the values from the same column (before the values are sent through weightless (grey) connections). The third (red) layer again only requires connections from the same row. We implement each smaller fully-connected layer ① of size $m \times n$ using n PEs connected in a ring. Each PE calculates the value of one output neuron, with the input neuron values initially entering the ring from the outside (Figure 5a), and next circling the ring (Figures 5b and 5c). The outputs of the previous layers are used as inputs to the next layer (i.e. R_i values are used for calculating R_m values, etc.). During the forward pass, each PE k calculates and stores the values of R_{ik}, R_{mk}, R_{ok}. Since it also stores all the weights connected to that neuron, the outputs and weights are fixed to PEs, while inputs circulate the rings. During the backward pass, delta signals propagate backwards through the network layers (right to left in Figure 4). To calculate the delta value of a neuron in a lower layer, we need to sum deltas and weights stored on different neurons. We fix inputs to PEs (higher layer deltas) and keep weights in place, while

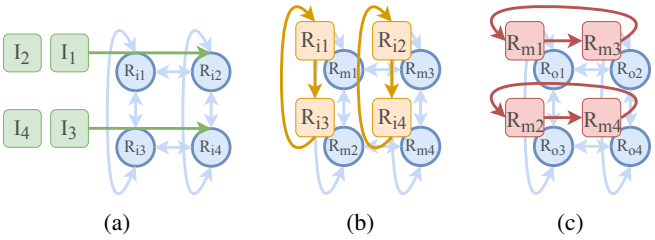

Figure 5: The movement of inputs and intermediate results in a (4, 4, 2, 2, 2) Clos cascade.

rotating outputs (lower layers deltas) and have each PE add their contribution to the output. Generalizing this approach, a dense $n^2 \times n^2$ network on a ring of n^2 PEs will require n^2 cycles to compute the inference outputs. A comparable Clos cascade on an $n \times n$ torus will only require $3n$ cycles, assuming $R_i = R_m = R_o = n$.

System Architecture: Due to a complex scatter operation (gray connections in Figure 2), ClosNets are more efficient to implement on FPGAs than on GPUs. Furthermore, in order to enable batchless training, we aim to store weights on-chip. This approach allows efficient training as we are not bottlenecked by DRAM bandwidth (Figure 1), and is viable since we use both sparsity and low-precision to reduce the size of networks.

The top level view of our architecture is presented in Figure 6 (left). The architecture loads training inputs and targets from main memory, and stores them in separate caches. The inputs are then fed to a pipeline of Clos cascades. Each Clos cascade accepts layer inputs, and produces layer outputs. Final layer's activations are fed to an error calculator module which computes the top layer's delta values. These values are propagated back through the Clos cascades, and the appropriate weights are updated.

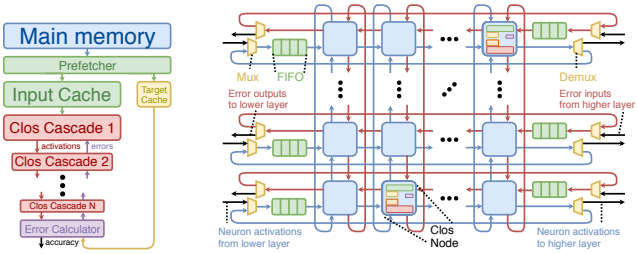

Figure 6: Top level (**left**) and cascade (**right**) architecture.

A Clos cascade consists of multiplexers and demultiplexers, FIFOs, and PEs arranged in a torus, as seen in Figure 6 (right). The cascade accepts neuron inputs from the left, stores them in the FIFOs, and sends them up and to the right (blue lines). FIFOs allow computing layers of sizes $m \times n$ on $n \times n$ toruses, where $m > n$. They store the $m - n$ activations until the torus can accept them.

Processing Element Architecture: In Figure 7 we see a single ClosNet processing element. This node performs three functions in parallel: it (1) calculates the next layer's activations in the forward pass unit, (2) propagates errors

back though the network in the backwards pass unit, and (3) uses current errors to make updates to the node's weights. Each PE stores all the weights it uses in a local block RAM. The only values that traverse the PEs are neuron inputs used in the forward pass, delta signals used in the backpropagation, and previous layer activations, used for updating the weights.

Initially, a node accepts inputs from a single training sample. It performs a forward pass and stores the neuron activations in the activation memory. Backpropagation starts when the node accepts delta signals, either from an error calculator or from a layer higher in the hierarchy, and computes the delta of its own neurons using equation 1:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{1}$$

To compute the delta signal δ^l, a node requires both the delta signal of the layer above δ^{l+1}, and the neuron activation z^l that this node calculated and stored during the forward pass. The delta signals δ^{l+1} are transmitted from the other nodes in the ring, while the backward pass reads the activation z^l from the activation memory. Finally, in parallel with propagating errors down the layers, the appropriate weights are updated using the equation:

$$\Delta w_{ij}^{l+1} = a_i^l \delta_j^{l+1} \tag{2}$$

The weight w_{ij}^{l+1} is stored at the PE j. To calculate the weight update Δw_{ij}^{l+1} we require both the delta δ_j^{l+1} from the layer above, which is stored at the PE j, and the activation a_i^l stored at PE i. In order to update all the weights at node j (i.e. not just w_{ij}^{l+1}), we have to circulate not only the delta signals, but also the activations. The activations and deltas follow the same movement pattern.

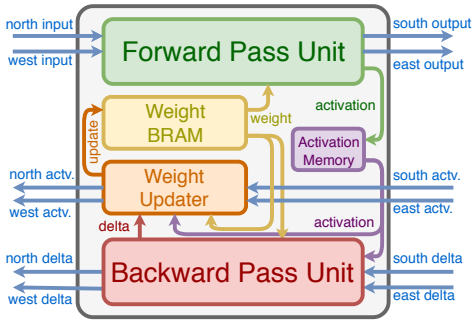

Figure 7: A ClosNet node calculating the next layer's activations, previous layer's errors, and weight updates.

In Figure 8 we see the Forward Pass Unit (FPU). The module accepts north or west neuron inputs ①, multiplies them by the weight ②, and adds the result to the sum ③. The inputs are also stored in the FIFO ④ and forwarded in the same direction they came from - north to south or west to east. Once the FPU has processed all connections, the neuron output sum is passed through the activation function ⑤. This neuron activation is then fed back to the FPU ⑥ to be used as an input in the next layer. Next, the activation is stored in the activation FIFO ⑦, where it is later read by the weight update unit and the backward pass unit.

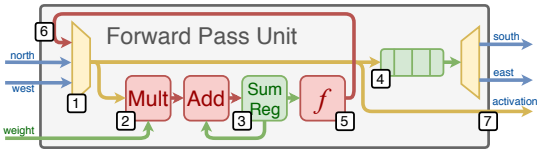

Figure 8: Forward Pass Unit (FPU)

In Figure 9 we see the Backward Pass Unit (BPU) and the Weight Update Unit (WUU). The BPU accepts current layer delta signals δ^{l+1}, and calculates the previous layer's delta signals δ^l. It begins by accepting an input from south or east, passing it through the disabled adder and multiplier, and storing it in the delta register ①. Simultaneously, the multiplexer ② feeds a zero into the FIFO ③. This zero is the value of the previous layer's delta value we are calculating, and it circulates all PEs in the ring. Next, following equation 1, the register value is multiplied by different weights ④, and the product is added to the incoming delta sum ⑤. Notice that for the FPU the sum was fixed to the FPU, while in the BPU this sum has to circulate the ring. This is because the weights and activations needed for the calculation are distributed amongst the PEs. Finally, when the all connections are processed, the sum from ⑤ is fed into the multiplier ⑥ which multiplies it by the activation derivative $\sigma'(z^l)$, and the next layer's delta is calculated.

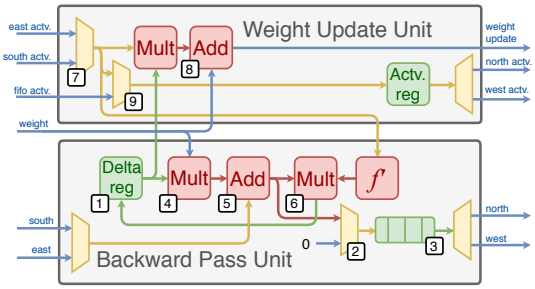

Figure 9: Backward Pass and Weight Update Unit

In parallel, WUU calculates the updates from equation 2. It multiplies the current delta value ① with the incoming activation ⑦, and adds it to the current weight ⑧. Since neuron activations are distributed and each Clos PE contains a small FIFO for activations, the multiplexer ⑨ chooses whether to accept the local activation or to transmit the neighbor ones. Initially, all WUUs choose their local activations, and for the rest of the layer they circulate their neighbor updates.

IV. Evaluation

ClosNets apply both machine learning and architecture solutions to the problem of training networks more efficiently. We evaluate accuracy separately from performance and area.

Accuracy: We evaluate the accuracy of dense neural networks and ClosNets on the MNIST dataset. We compare three baseline configurations of 784 inputs, 10 outputs, and 256, 128, and 64 hidden layer neurons. For ClosNets, we

use the 784-256-10 neuron configuration, and replace the first layer with a Clos cascade since it contains 99% of the network parameters. For simplicity, we pad the 784 inputs with zeros so that the input size is 1024. We test four router configurations: (1024, 256, 32, 16, 32), (1024, 256, 32, 8, 32), (1024, 256, 16, 8, 16), and (1024, 256, 16, 5, 16). The results of the MNIST test set accuracy over epochs are presented in Figure 10 (right). Given enough path diversity, ClosNets perform as well as or better than dense networks. In Figure 10 (left) we show the accuracy per FLOP during training, which better illustrates the convergence speed. Both Clos and dense networks use 32bit floating point operations.

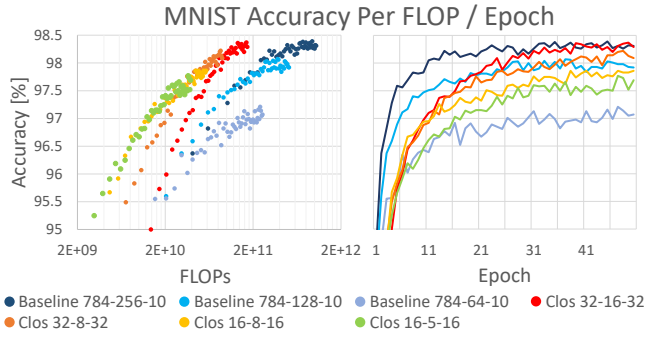

Figure 10: MNIST accuracy of dense and Clos networks.

Performance: The throughput of an l-cascade ClosNet is limited by the slowest cascade in the pipeline. For an (I, O, Ri, Rm, Ro) Clos cascade, the first phase will require I/Ri cycles to complete, the second one Ri cycles, and the third one Rm cycles. On top of this, the network may require some cycles to move data in place before beginning. This gives us the calculation for the number of cycles per sample $C = I/Ri + Ri + Rm$. Notice that C does not depend on the size of the output O. This is due to us mapping each output to an individual processing element, avoiding time multiplexing.

We compare the performance of our design with that of a similar ring-based dense layer implementation. Here, both designs have 1024 PEs, and all weights are stored in PEs using them. In the case of the dense layer, PEs are arranged in a ring, and the inputs circle all the PEs. We measure the number of cycles it takes to perform one forward pass or one backward pass. We assume that the weight matrices can fit into on-chip BRAM, and have a single cycle read latency. In Table I we show ideal and results measured on a simulation for the number of cycles per sample. Due to a design issue, the Clos backward pass is underperforming. The issue will be removed in the next iteration of the design.

Area: We synthesize several torus sizes with instructions to implement two Clos cascades on an Altera Cyclone IV FPGA with 150k Logic Elements (LE). For all our experiments, we use on chip memory for storing weights, and embedded 9 bit, fixed-point multipliers for multiplication operations. 9 bit multiplications incur a negligible accuracy loss compared to a 32 bit floating-point implementation [11], but significantly reduce power and area requirements. Table II shows the resource usage for 2x2, 4x4 and 8x8 toruses.

	Cycles / sample		Speedup	
	Ideal	Measured	Ideal	Measured
Dense Forward	1024	1024	1x	1x
Clos Forward	96	128	10.6x	**8.0x**
Dense Backward	1024	1024	1x	1x
Clos Backward	96	224	10.6x	**4.57x**

Table I: The ideal and measured throughput for a 1024-input, 1024-output fully-connected and Clos networks.

Network Size	Logic Element Utilization	Total Registers	Block RAM (Bits)	9 bit Multipliers
2x2	3657	1784	0	12
4x4	10458	4872	4608	48
8x8	44278	19168	27648	192

Table II: Synthesis results for Clos networks on an Altera Cyclone IV FPGA with 150k Logic Elements (LE).

V. Conclusion

In this work we introduce a novel approach for reducing the size of dense neural nets. We present ClosNets - fully-connected cascades of sparse layers with the Clos topology. We show that ClosNets have comparable accuracy and a 5-10× smaller size over fully-connected layers, and propose a simple torus-based implementation for the network.

References

[1] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[2] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "Enet: A deep neural network architecture for real-time semantic segmentation," *CoRR*, vol. abs/1606.02147, 2016.

[3] 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.

[4] H. Zhang, K. Kara, J. Li, D. Alistarh, J. Liu, and C. Zhang, "Zipml: An end-to-end bitwise framework for dense generalized linear models," *CoRR*, vol. abs/1611.05402, 2016.

[5] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 598–605.

[6] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "FPGA-accelerated dense linear machine learning: A precision-convergence trade-off," *Proceedings - IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017*, pp. 160–167, apr 2017.

[7] Y. Umuroglu, N. J. Fraser, G. Gambardella, and M. Blott, "FINN : A Framework for Fast , Scalable Binarized Neural Network Inference," *Fpga*, no. February, 2017.

[8] H. Mostafa, B. Pedroni, S. Sheik, and G. Cauwenberghs, "Hardware-efficient on-line learning through pipelined truncated-error backpropagation in binary-state networks."

[9] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar 1994.

[10] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[11] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *Iclr*, no. Section 5, p. 10, 2015.