USC University of Southern California

USC Viterbi School of Engineering

Energy-Efficient, Low-Latency Realization of Neural Networks Through Boolean Logic Minimization

Mahdi Nazemi, Ghasem Pasandi, and
Massoud Pedram

http://sportlab.usc.edu/

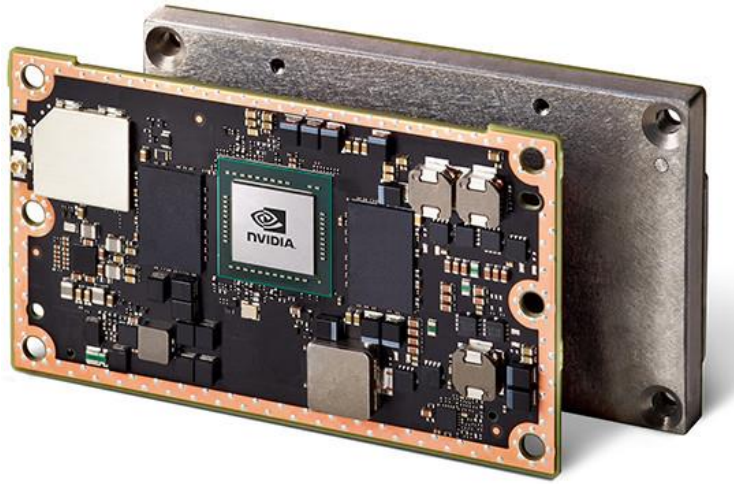
Asia and South Pacific Design Automation Conference

Jan. 22, 2019

Machine Learning Is Everywhere

- Search
- Ranking
- Recommendation systems
- Collaborative filtering
- Digital assistants
- Advertisements

But at What Cost



NVIDIA TITAN V (Volta)	
CUDA Cores	5120
Tensor Cores	640
Deep Learning Performance (TeraFLOPS)	110
Power Supply (W)	600
Thermal Design Power (W)	**250**

Jetson TX2 (Pascal)	
CUDA Cores	256
Tensor Cores	0
Deep Learning Performance (TeraFLOPS)	1
Power Supply (W)	-
Thermal Design Power (W)	**7.5**

// Tensor Cores perform A * B + C on 4x4 matrices

Cost Breakdown – Energy Consumption

Memory-to-Compute Energy Ratio for Multiply Operation				
	8KB Cache	**32KB Cache**	**1MB Cache**	**DRAM**
Int8	18.75	37.50	187.50	2437.50
Int32	4.84	9.68	48.39	629.03
Float16	6.82	13.64	68.19	886.36
Float32	4.05	8.11	40.54	527.03

- Energy consumption scaling factors
  - Multipliers: quadratic function of bit-width
  - Adders: linear function of bit-width
  - Caches: sublinear function of size
- Reducing the bit-width of operands through quantization widens the gap between energy consumption of memory accesses and computation

// Each multiplier/adder (MAC) generates 3 (4) accesses to the memory

M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," ISSCC, 2014.

Cost Breakdown - Latency

- Intel Haswell architecture
  - 32-bit integer operations

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

Memory	Size (KB)	Latency (Clock Cycle)
L1 Cache	32	4 – 5
L2 Cache	256	12
L3 Cache	8,192	36 – 58
DRAM	-	230 – 422

"Intel 64 and IA-32 architectures optimization reference manual," https://software.intel.com.
"Intel Haswell," https://www.7-cpu.com/cpu/Haswell.html.

Goals

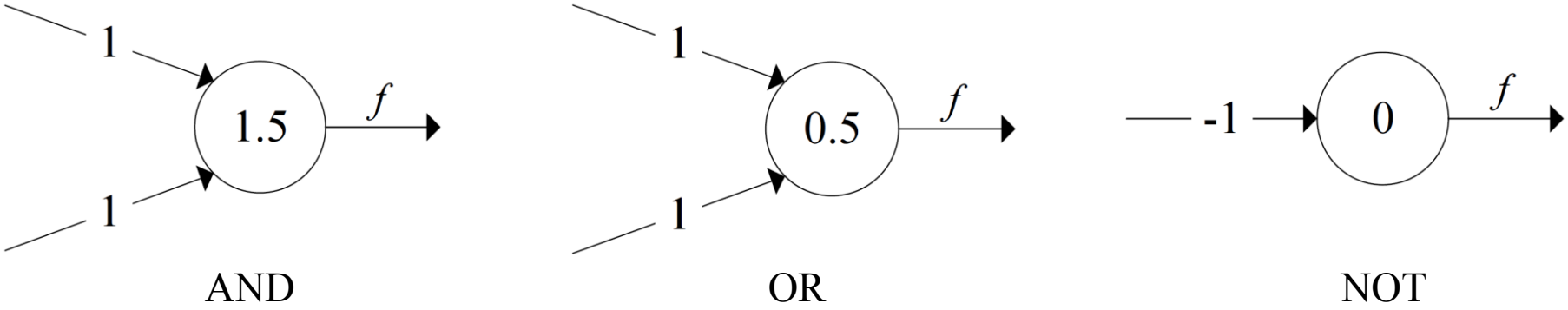
- Memory
  - Reduce accesses to the memory
  - Reduce storage requirements for model parameters
- Model performance
  - Cause little or no degradation in classification accuracy
- Computation
  - Reduce the number of required resources

Standard Neuron Model

- McCulloch-Pitts neuron

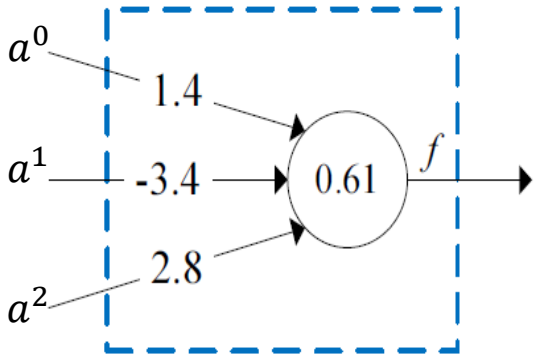
$$f = \begin{cases} 1, & \sum_j a^j \times w^j \geq b \\ 0, & otherwise. \end{cases}$$

a^j: binary inputs to the neuron
w^j: weights
b: bias or threshold



// Rationale: Digital circuits and processors could be built using these neuron-based logic gates. Let's do the inverse, i.e., utilizing digital logic gates to perform inference without having to read parameters off memory and performing expensive computations.

Optimizing Individual Neurons

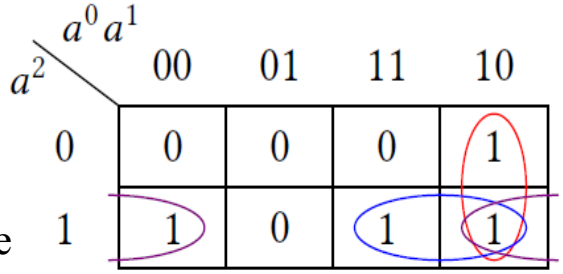


(a)

a^0	a^1	a^2	$\sum_{j=0}^{2} a^j \times w^j$	f
0	0	0	0.0	0
0	0	1	2.8	1
0	1	0	-3.4	0
0	1	1	-0.6	0
1	0	0	1.4	1
1	0	1	4.2	1
1	1	0	-2.0	0
1	1	1	0.8	1

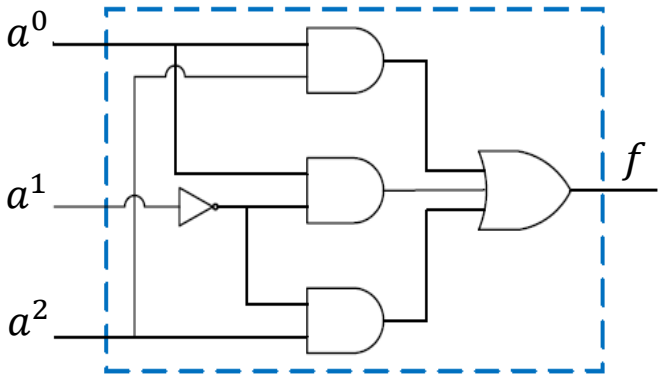
(b)

// Only quantize activations to binary values and leave weights/biases unconstrained (otherwise to maintain output accuracy size of the DNN will have to be greatly increased)



$$f = a^0\overline{a^1} + a^0a^2 + \overline{a^1}a^2$$

(c)

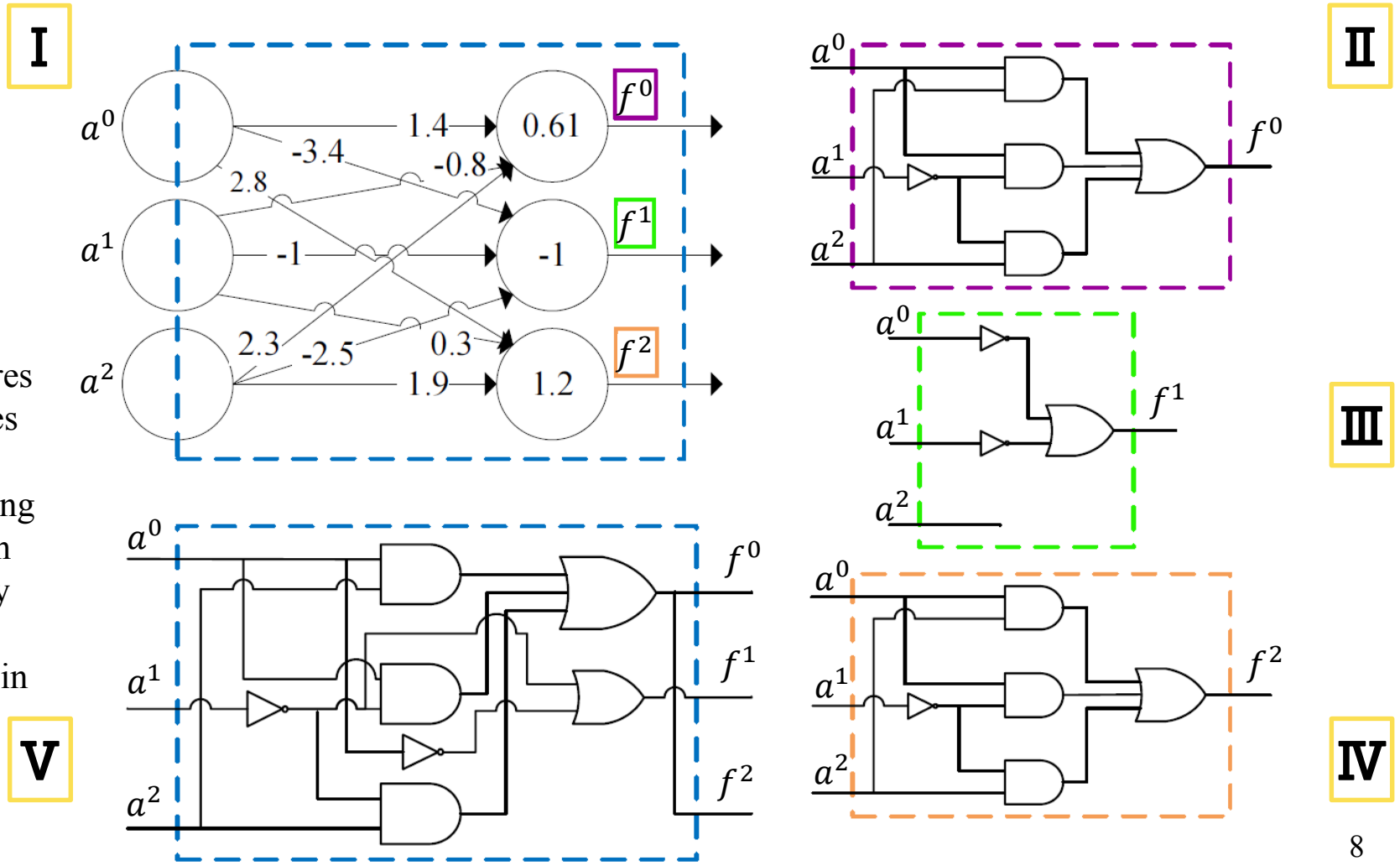


(d)

Optimizing Layers

- Apply common logic expression extraction to share resources among outputs of a layer (steps I thru V)

// Optimized logic requires 7 logic gates while implementing each neuron individually requires 13 logic gates in total.



Optimizing Neurons with a Large Number of Inputs

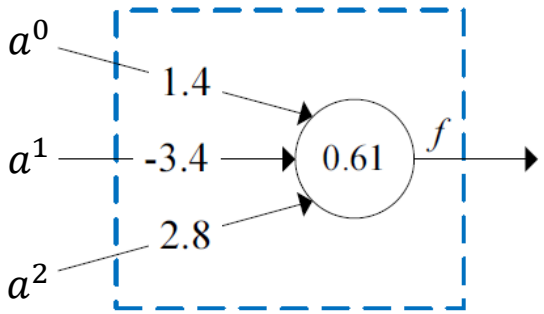
- The size of truth table grows exponentially with the number of inputs to a neuron
  - Leads to scalability issues

- For each neuron, define an **incompletely specified Boolean function** by considering input combinations, which are encountered when we apply the training data to the neural network
  - Permutations that lead to an output of 1 constitute the ON-set
  - Permutations that lead to an output of 0 constitute the OFF-set
  - Permutations that are not encountered constitute the DON'T CARE-set (DC-set)

- Advantages
  - Small number of entries specified in the truth table
  - Increased potential for logic optimization

Example

- Assume the depicted neuron never encounters the following input permutations:
  - $a^0a^1a^2 = 000$
  - $a^0a^1a^2 = 110$

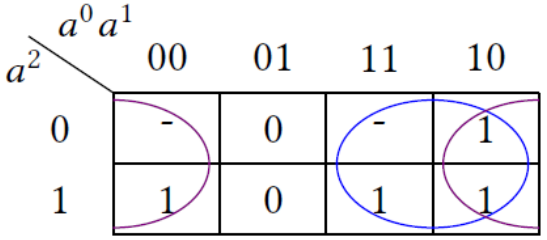


(a)

a^0	a^1	a^2	$\sum_{j=0}^{2} a^j \times w^j$	f
0	0	0	0.0	-
0	0	1	2.8	1
0	1	0	-3.4	0
0	1	1	-0.6	0
1	0	0	1.4	1
1	0	1	4.2	1
1	1	0	-2.0	-
1	1	1	0.8	1

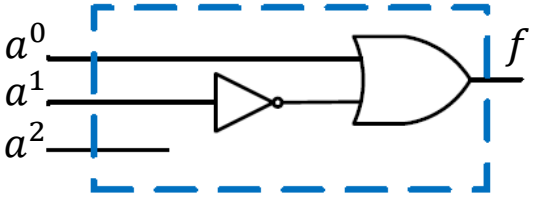
(b)

- This allows defining a DC-set which in turn enables further optimization of the logic function of the output signal



$f = a^0 + \overline{a^1}$

(c)



(d)

Generating a Prime & Irredundant Cover

1. Sort cubes of ON-set in <u>increasing order of probability of being covered by the expansion of other cubes</u>
   1. Define a weight matrix where each row corresponds to a cube in the ON-set, each column corresponds to a variable in the cube, and $0/1$ values are replaced with $-1/+1$.
   2. Find the dot product of each row with the column sum of weight matrix
   3. Sort the outputs of dot product in increasing order of values (Pick randomly when there is a tie)

- For the previous example:

ON-set	Weight Matrix	Column Sums	Dot Product
001	-1 +1 +1	+2 +2 +2	+2
110	+1 +1 -1		+2
101	+1 -1 +1		+2
111	+1 +1 +1		+6

Prime & Irredundant Cover (Cont'd)

2. Raise variables in an uncovered cube of sorted ON-set while maintaining orthogonality with OFF-set
   - Make prime

3. Mark all cubes in the sorted ON-set that are covered by the prime cube as covered
   - Do single cube containment

4. Repeat steps 2 and 3 until we get a prime and irredundant cover of the function

// The *expand* and *make_irredundant* functions consist of a number of highly parallelizable for loops => Can speed up optimization by writing a multi-threaded CPU program or a CUDA program to run the function on GPUs.

Summary of the Realization Method

Input:

L: number of layers

u_i, $i = 0, 1, 2, ..., L$: number of neurons in layer i

$\mathbf{a}_i$, $i = 0, 1, 2, ..., L$: activations at layer i (for all training samples)

Output:

optimized neural network

1: **for** $i = 2$ to $L - 1$ **do**

2: **for** $j = 0$ to $u_i - 1$ **do**

3: $OptimizeNeuron(Inputs(i, j, \mathbf{a}_{i-1}), Outputs(i, j, \mathbf{a}_i))$

4: **end for**

5: $OptimizeLayer()$

6: $Pythonize()$

7: **end for**

// The first and last layers are optimized separately using full quantization to achieve an overall acceptable performance

// Use the Glow compiler to map to hardware accelerators designed for processing neural networks

// We can apply logic optimizations across two or more consecutive layers at the same time. We may also use min-cut partitioning to reduce memory accesses for intermediate values.

Full Suite of Logic Optimization Techniques

- Logic synthesis allows optimizing individual neurons, layers, and collections of layers that constitute the neural network
- Two-level logic minimization algorithms are utilized to find an efficient implementation for neurons defined using an incompletely specified function
- Multi-level logic minimization algorithms are utilized to optimize groups of neurons within a single layer or multiple layers across a neural network
  - Common logic expression extraction
  - SAT-based resource sharing
  - Retiming to minimize register count
- Can improve performance by pipelining and/or register retiming

Mapping to Hardware

- Map the optimized netlist directly to hardware (e.g. FPGA)– similar to Microsoft Brainwave
  - Weights are stored on the FPGA's BRAM
  - Best achievable latency
  - High resource consumption (without any resource reuse)

- Compile the high-level graph defined in Python (PyTorch) onto a CPU, a GPU, or a specialized hardware accelerator
  - For example, Intel CPUs are capable of performing 1,024 AND/NAND/OR/XOR operations in a single cycle
  - The compiler needs to find 1,024 independent bitwise operations defined in the high-level graph and pack them together to optimize for Intel CPUs

Forward Propagation in MLPs

Input:

L: number of layers

$\mathbf{a}_0$: a mini-batch of inputs

$\mathbf{W}$: weights

$\boldsymbol{\beta}$: batch normalization parameters

Output:

$\mathbf{a}_L$: network's predictions

1: **for** $i = 1$ to L **do**
2: $\quad \mathbf{z}_i = \mathbf{a}_{i-1}\mathbf{W}_i$
3: $\quad \mathbf{a}_i = BatchNorm(\mathbf{z}_i, \boldsymbol{\beta})$
4: $\quad$ **if** $i < L$ **then**
5: $\quad\quad \mathbf{a}_i = Sign(\mathbf{a}_i)$
6: $\quad$ **end if**
7: **end for**
8: return $\mathbf{a}_L$

// All operations are similar to those for MLPs (multi-layer perceptrons), except the activation function, which is replaced with the *sign* function.

Experimental Setup

- MNIST dataset of handwritten digits
  - 28x28 grayscale images
  - 60,000 training samples
  - 10,000 test samples
- Intel Arria 10 GT 1150 FPGA
  - 427,000 adaptive logic modules (ALMs)
  - 5,562,240 bits of block RAM
  - 1,518 DSP blocks
- 32-bit floating-point multiply-accumulate operation
  - 541 ALMs (this count constitutes our baseline)
  - Latency: 34.68ns (again latency of a single ALM is our baseline)

Experimental Results – Accuracy

- MLP architecture
  - 784 – 100 – 100 – 100 – 10 (FC1 – FC2 – FC3 – FC4)
- CNN architecture
  - Conv1: $3 \times 3 \times 1$, 10 output channels, 2×2 max pooling
  - Conv2: $3 \times 3 \times 10$, 20 output channels, 2×2 max pooling
  - FC1: 500 – 10 (fully connected)

Neural Network	Classification Accuracy (%)		
	Baseline	**SW-Q1**	**HW-Q1**
MLP	98.27	96.89	97.01 **(+0.12)**
CNN	99.00	98.21	97.92 **(-0.29)**

- Baseline uses float-32 representation
- SW-Q1 represents the same neural network as the baseline with activations quantized to binary values
- HW-Q1 is the neural network solution obtained by our proposed solution

Experimental Results – Hardware Cost

- Improvements in optimized layers (layers 2 through $L - 1$)
  - 100% saving in storage required for model parameters
  - 100% saving in memory accesses for reading model parameters

- The following table reports the overall improvements:

// The setup is somewhat in favor of baseline because it assumes that all weights and activations can be read once and reused as many times as needed and that there are an unlimited number of resources.

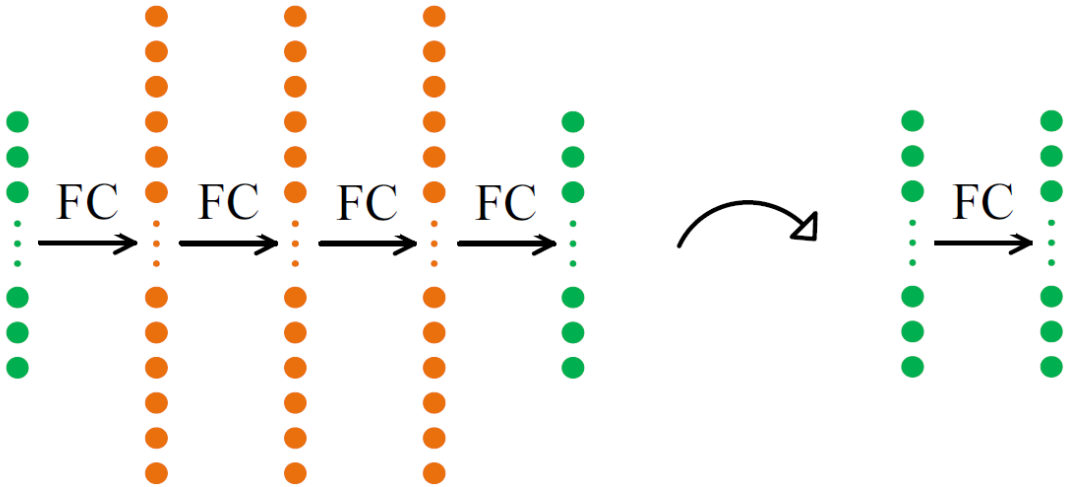
Neural Network		Computation		Memory
		MACs	Latency	(Bytes)
MLP	Baseline	20,000	14	80,800
	HW-Q1	207	0.88	25
Improvement		**97x**	**16x**	**3,232x**
CNN	Baseline	217,800	7	23,640
	HW-Q1	3,630	0.41	514
Improvement		**60x**	**17x**	**46x**

Goals

- ☑ Memory
  - ☑ Reduce accesses to the memory
  - ☑ Reduce storage requirements for model parameters
- ☐ Model performance
  - ☐ Cause little or no degradation on classification accuracy
- ☑ Computation
  - ☑ Reduce the number of required resources

Ongoing Work: Vestigial Networks

- Replace a few consecutive layers as a single equivalent layer
  - The truth table inputs are provided by the first green layer while the outputs are evaluated at the second green layer
  - In contrast to having a separate truth table for each pair of consecutive layers
- The intermediate layers may be arbitrarily deep and/or wide



Multivalued Quantization

- Binary quantization may lead to a substantial drop in accuracy on relatively harder datasets
  - LeNet-5 on CIFAR-10: 75.91%
  - LeNet-5 on CIFAR-10 (binary activations): 57.05% **(-18.86%)**

- Multivalued quantization allows compensating accuracy loss due to binary quantization

Neural Network	Classification Accuracy (%)				
	Baseline	**SW-Q1**	**SW-Q2**	**SW-Q3**	**SW-Q4**
LeNet-5	75.91	57.05%	68.75%	75.01%	76.17%

// Need to apply multivalued logic minimization in Espresso to find an optimized hardware realization of the neural network.

Multivalued Quantization (Cont'd)

- Use a clipped ReLU as the activation function
  - Trainable clipping value

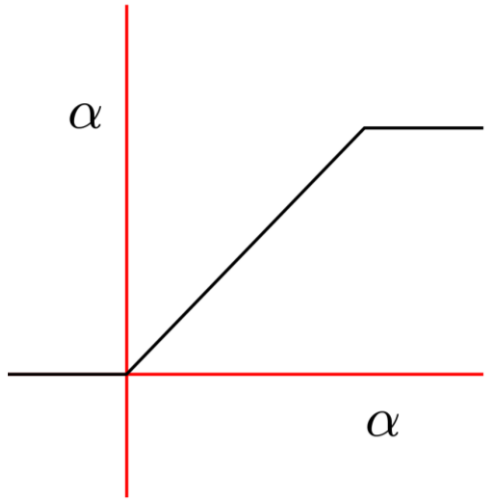
$$y = PACT(x) = 0.5(|x| - |x-\alpha| + \alpha) = \begin{cases} 0, & x \in (-\inf, 0) \\ x, & x \in [0, \alpha) \\ \alpha, & x \in [\alpha, +\inf) \end{cases}$$

- Quantize activations using k bits

$$y_q = round(y \frac{2^k - 1}{\alpha}) \frac{\alpha}{2^k - 1}$$

// Lower α => lower quantization error
Higher α => closer resemblance of the stanard ReLU function

Set α high in the beginning to allow gradients to propagate.



J. Choi *et* al., "PACT: Parameterized Clipping Activation for Quantized Neural Networks," 2018.

Conclusion

- Memory is the bottleneck in processing deep neural networks
  - Energy consumption
  - Latency

- This work presented a realization method that allows inference without reading model parameters from memory

- There are a few solutions for compensating the accuracy loss due to binary quantization (work in progress)
  - The resulting solution will have the same level of accuracy as the baseline while enjoying the benefits of proposed solution