Deep Learning Accelerators
Deep Learning is Changing Our Lives

Self-Driving Car

Machine Translation

AlphaGo

Smart Robots
Models are Getting Larger

**IMAGE RECOGNITION**

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Layers</th>
<th>GFLOP</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>AlexNet</td>
<td>8</td>
<td>1.4</td>
<td>~16%</td>
</tr>
<tr>
<td>2015</td>
<td>ResNet</td>
<td>152</td>
<td>22.6</td>
<td>~3.5%</td>
</tr>
</tbody>
</table>

**SPEECH RECOGNITION**

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>GFLOP</th>
<th>Data</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>Deep Speech 1</td>
<td>80</td>
<td>7,000</td>
<td>~8%</td>
</tr>
<tr>
<td>2015</td>
<td>Deep Speech 2</td>
<td>465</td>
<td>12,000</td>
<td>~5%</td>
</tr>
</tbody>
</table>

Dally, NIPS'2016 workshop on Efficient Methods for Deep Neural Networks
The first Challenge: Model Size

Hard to distribute large models through over-the-air update
## The Second Challenge: Speed

<table>
<thead>
<tr>
<th>Model</th>
<th>Error Rate</th>
<th>Training Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18</td>
<td>10.76%</td>
<td>2.5 days</td>
</tr>
<tr>
<td>ResNet50</td>
<td>7.02%</td>
<td>5 days</td>
</tr>
<tr>
<td>ResNet101</td>
<td>6.21%</td>
<td>1 week</td>
</tr>
<tr>
<td>ResNet152</td>
<td>6.16%</td>
<td>1.5 weeks</td>
</tr>
</tbody>
</table>

Such long training time limits ML researcher’s productivity.

Training time benchmarked with fb.resnet.torch using four M40 GPUs.
The Third Challenge: Energy Efficiency

AlphaGo: 1920 CPUs and 280 GPUs, $3000 electric bill per game

on mobile: drains battery
on data-center: increases TCO
Where is the Energy Consumed?

larger model => more memory reference => more energy
Where is the Energy Consumed?

larger model => more memory reference => more energy

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit int ADD</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>32 bit float ADD</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>32 bit Register File</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>32 bit int MULT</td>
<td>3.1</td>
<td></td>
</tr>
<tr>
<td>32 bit float MULT</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>32 bit SRAM Cache</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td><strong>32 bit DRAM Memory</strong></td>
<td><strong>640</strong></td>
<td></td>
</tr>
</tbody>
</table>

1 = 1000 × +

This image is in the public domain
Where is the Energy Consumed?

larger model => more memory reference => more energy

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy [pJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit int ADD</td>
<td>0.1</td>
</tr>
<tr>
<td>32 bit float ADD</td>
<td>0.9</td>
</tr>
<tr>
<td>32 bit Register File</td>
<td>1</td>
</tr>
<tr>
<td>32 bit int MULT</td>
<td>3.1</td>
</tr>
<tr>
<td>32 bit float MULT</td>
<td>3.7</td>
</tr>
<tr>
<td>32 bit SRAM Cache</td>
<td>5</td>
</tr>
<tr>
<td><strong>32 bit DRAM Memory</strong></td>
<td><strong>640</strong></td>
</tr>
</tbody>
</table>

how to make deep learning more efficient?
Improve the Efficiency of Deep Learning by Algorithm-Hardware Co-Design
Application as a Black Box

Algorithm

Hardware
Open the Box before Hardware Design

Breaks the boundary between algorithm and hardware
Agenda

Inference

Training
Agenda

- Algorithms for Efficient Inference
- Algorithms for Efficient Training
- Hardware for Efficient Inference
- Hardware for Efficient Training
Hardware 101: the Family

Hardware

General Purpose*

- CPU: latency oriented

Specialized HW

- GPU: throughput oriented
- FPGA: programmable logic
- ASIC: fixed logic

* including GPGPU
Hardware 101: Number Representation

\((-1)^S \times (1.M) \times 2^E\)

<table>
<thead>
<tr>
<th>Format</th>
<th>S</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>1</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>FP16</td>
<td>1</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Int32</td>
<td>1</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Int16</td>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Int8</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Fixed point</td>
<td>S</td>
<td>I</td>
<td>F</td>
</tr>
</tbody>
</table>

Range and Accuracy:

- **FP32**: $10^{-38}$ - $10^{38}$, Accuracy: .000006%
- **FP16**: $6 \times 10^{-5}$ - $6 \times 10^{4}$, Accuracy: .05%
- **Int32**: $0$ - $2 \times 10^9$, Half-precision
- **Int16**: $0$ - $6 \times 10^4$, Half-precision
- **Int8**: $0$ - 127, Half-precision
- **Fixed point**: -

Dally, High Performance Hardware for Machine Learning, NIPS'2015
Hardware 101: Number Representation

<table>
<thead>
<tr>
<th>Operation:</th>
<th>Energy (pJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b Add</td>
<td>0.03</td>
</tr>
<tr>
<td>16b Add</td>
<td>0.05</td>
</tr>
<tr>
<td>32b Add</td>
<td>0.1</td>
</tr>
<tr>
<td>16b FP Add</td>
<td>0.4</td>
</tr>
<tr>
<td>32b FP Add</td>
<td>0.9</td>
</tr>
<tr>
<td>8b Mult</td>
<td>0.2</td>
</tr>
<tr>
<td>32b Mult</td>
<td>3.1</td>
</tr>
<tr>
<td>16b FP Mult</td>
<td>1.1</td>
</tr>
<tr>
<td>32b FP Mult</td>
<td>3.7</td>
</tr>
<tr>
<td>32b SRAM Read (8KB)</td>
<td>5</td>
</tr>
<tr>
<td>32b DRAM Read</td>
<td>640</td>
</tr>
</tbody>
</table>

Energy numbers are from Mark Horowitz “Computing’s Energy Problem (and what we can do about it)”, ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.
Agenda

- **Algorithm**
  - Algorithms for Efficient Inference
  - Algorithms for Efficient Training

- **Inference**
  - Hardware for Efficient Inference

- **Training**
  - Hardware for Efficient Training
Part 1: Algorithms for Efficient Inference

• 1. Pruning
• 2. Weight Sharing
• 3. Quantization
• 4. Low Rank Approximation
• 5. Binary / Ternary Net
• 6. Winograd Transformation
Part 1: Algorithms for Efficient Inference

• 1. Pruning
• 2. Weight Sharing
• 3. Quantization
• 4. Low Rank Approximation
• 5. Binary / Ternary Net
• 6. Winograd Transformation
Pruning Neural Networks

[Lecun et al. NIPS’89]
[Han et al. NIPS’15]
Pruning Neural Networks

-0.01x^2 + x + 1

Train Connectivity

Prune Connections

Train Weights

60 Million
6M

10x less connections

[Han et al. NIPS’15]
Pruning Neural Networks

[Han et al. NIPS’15]

Train Connectivity

Accuracy Loss

Parameters Pruned Away

40% 50% 60% 70% 80% 90% 100%

4.5% 4.0% 3.5% 3.0% 2.5% 2.0% 1.5% 1.0% 0.5% 0.0% -0.5% -1.0% -1.5% -2.0% -2.5% -3.0% -3.5% -4.0% -4.5%
Pruning Neural Networks

[Han et al. NIPS’15]

Accuracy Loss

-4.5%
-4.0%
-3.5%
-3.0%
-2.5%
-2.0%
-1.5%
-1.0%
-0.5%
0.0%
0.5%

Parameters Pruned Away

40% 50% 60% 70% 80% 90% 100%

Pruning

Train Connectivity

Prune Connections
Retrain to Recover Accuracy

[Han et al. NIPS’15]

Accuracy Loss

Parameters Pruned Away

Train Connectivity

Prune Connections

Train Weights

Pruning

Pruning+Retraining
Iteratively Retrain to Recover Accuracy

- Parameters Pruned Away:
  - Pruning
  - Pruning + Retraining
  - Iterative Pruning and Retraining

Accuracy Loss

- Accuracy Loss Table:
  - Accuracy Loss: -4.5%, -4.0%, -3.5%, -3.0%, -2.5%, -2.0%, -1.5%, -1.0%, -0.5%, 0.0%, 0.5%

Parameters Pruned Away:

- 40%
- 50%
- 60%
- 70%
- 80%
- 90%
- 100%

[Han et al. NIPS’15]
Pruning RNN and LSTM


Figure copyright IEEE, 2015; reproduced for educational purposes.
Pruning RNN and LSTM

- **Original**: a basketball player in a white uniform is playing with a ball
- **Pruned 90%**: a basketball player in a white uniform is playing with a basketball

- **Original**: a brown dog is running through a grassy field
- **Pruned 90%**: a brown dog is running through a grassy area

- **Original**: a man is riding a surfboard on a wave
- **Pruned 90%**: a man in a wetsuit is riding a wave on a beach

- **Original**: a soccer player in red is running in the field
- **Pruned 95%**: a man in a red shirt and black and white black shirt is running through a field

[Han et al. NIPS’15]
Pruning Happens in Human Brain


50 Trillion Synapses (Newborn) → 1000 Trillion Synapses → 500 Trillion Synapses (Adolescent)

This image is in the public domain

This image is in the public domain

This image is in the public domain
Pruning Changes Weight Distribution

Conv5 layer of Alexnet. Representative for other network layers as well.
Exploring the Granularity of Sparsity that is Hardware-friendly

4 types of pruning granularity

irregular sparsity => regular sparsity => more regular sparsity => fully-dense model

[Mao et al, CVPR’17 W]
6. Advantages of Coarse-grained Sparsity

In conv layers, 2-D convolution is usually the primitive, like kernels or sub-kernel vectors, though it is still difficult for acceleration on general-purpose processors, there are several advantages over fine-grained sparsity. Those advantages enable simpler circuits and higher concurrency on custom hardware.

We want to investigate how accuracies differ at the same level of sparsity, so we use linear interpolation to obtain the estimated density and storage ratio.

Possible to prune a model that exactly matches the baseline accuracy constraint, we list the results of AlexNet, VGG-16 and GoogLeNet in Table 2. Here the storage ratio is defined as the model size of pruned 8-bit model (with 4-bit indexes) divided by that of baseline 8-bit model.

To better compare the compression ratio under the same accuracy, we use linear interpolation to obtain the estimated density and storage ratio.

As the model size of pruned 8-bit model (with 4-bit indexes) is almost the same as that of dense 8-bit model. Notice that it is almost impossible to prune a model that exactly matches the baseline accuracy constraint, we list the results of AlexNet, VGG-16 and GoogLeNet in Table 2. Here the storage ratio is defined as the model size of pruned 8-bit model (with 4-bit indexes) divided by that of baseline 8-bit model.

We do not use full-precision 32-bit weights but use 8-bit weights instead, as 8-bit weights have been proven to be sufficient in a lot of literature. We use 4-bit indexes to store the distances between adjacent non-zeros, following the method in Deep Compression [8]. Moreover, as in-
Part 1: Algorithms for Efficient Inference

- 1. Pruning
- 2. Weight Sharing
- 3. Quantization
- 4. Low Rank Approximation
- 5. Binary / Ternary Net
- 6. Winograd Transformation
Trained Quantization

2.09, 2.12, 1.92, 1.87

2.0
Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by \(9 \times 13\)\(^\times\), while quantization further improves the compression rate: between \(27 \times\) and \(31 \times\). Huffman coding gives more compression: between \(35 \times\) and \(49 \times\).

The compression rate already included the meta-data for sparse representation. The compression scheme doesn't incur any accuracy loss.

Features such as better privacy, less network bandwidth and real-time processing, the large storage overhead prevents deep neural networks from being incorporated into mobile apps.

The second issue is energy consumption. Running large neural networks require a lot of memory bandwidth to fetch the weights and a lot of computation to do dot products—which in turn consumes considerable energy. Mobile devices are battery-constrained, making power-hungry applications such as deep neural networks hard to deploy.

Energy consumption is dominated by memory access. Under 45nm CMOS technology, a 32 bit floating point add consumes 0.9pJ, a 32bit SRAM cache access takes 5pJ, while a 32bit DRAM memory access takes 640pJ, which is 3 orders of magnitude of an add operation. Large networks do not fit in on-chip storage and hence require the more costly DRAM accesses. Running a 1 billion connection neural network, for example, at 20fps would require \((20 \text{ Hz}) (1 \text{ G}) (640 \text{ pJ}) = 12.8 \text{ W}\) just for DRAM access—well beyond the power envelope of a typical mobile device.

Our goal is to reduce the storage and energy required to run inference on such large networks so they can be deployed on mobile devices. To achieve this goal, we present "deep compression": a three-stage pipeline (Figure 1) to reduce the storage required by neural network in a manner that preserves the original accuracy. First, we prune the networking by removing the redundant connections, keeping only the most informative connections. Next, the weights are quantized so that multiple connections share the same weight, thus only the codebook (effective weights) and the indices need to be stored. Finally, we apply Huffman coding to take advantage of the biased distribution of effective weights.

Our main insight is that, pruning and trained quantization are able to compress the network without interfering each other, thus lead to surprisingly high compression rate. It makes the required storage so small (a few megabytes) that all weights can be cached on chip instead of going to off-chip DRAM which is energy consuming. Based on "deep compression", the EIE hardware accelerator Han et al. (2016) was later proposed that works on the compressed model, achieving significant speedup and energy efficiency improvement.

Network pruning has been widely studied to compress CNN models. In early work, network pruning proved to be a valid way to reduce the network complexity and over-fitting (LeCun et al., 1989; Hanson & Pratt, 1989; Hassibi et al., 1993; Strom, 1997). Recently Han et al. (2015) pruned state-of-the-art CNN models with no loss of accuracy. We build on top of that approach. As shown on the left side of Figure 1, we start by learning the connectivity via normal network training. Next, we prune the small-weight connections: all connections with weights below a threshold are removed from the network. Finally, we retrain the network to learn the final weights for the remaining sparse connections. Pruning reduced the number of parameters by \(9 \times\) and \(13 \times\) for AlexNet and VGG-16 model.

Trained Quantization

- Cluster the Weights
- Generate Code Book
- Quantize the Weights with Code Book
- Retrain Code Book

**32 bit**

**4bit** 8x less memory footprint
Trained Quantization

weights
(32 bit float)

2.09 -0.98 1.48 0.09
0.05 -0.14 -1.08 2.12
-0.91 1.92 0 -1.03
1.87 0 1.53 1.49

Network quantization and weight sharing further compresses the pruned network by reducing the number of bits required to represent each weight. We limit the number of effective weights we need to store by having multiple connections share the same weight, and then fine-tune those shared weights. Weight sharing is illustrated in Figure 3. Suppose we have a layer that has 4 input neurons and 4 output neurons, the weight is a $4 \times 4$ matrix. On the top left is the $4 \times 4$ weight matrix, and on the bottom left is the $4 \times 4$ gradient matrix. The weights are quantized to 4 bins (denoted with 4 colors), all the weights in the same bin share the same value, thus for each weight, we then need to store only a small index into a table of shared weights. During update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from last iteration. For pruned AlexNet, we are able to quantize to 8-bits (256 shared weights) for each CONV layers, and 5-bits (32 shared weights) for each FC layer without any loss of accuracy.

To calculate the compression rate, given $k$ clusters, we only need $\log_2(k)$ bits to encode the index. In general, for a network with $n$ connections and each connection is represented with $b$ bits, constraining the connections to have only $k$ shared weights will result in a compression rate of:

$$r = nb \log_2(k) + kb$$
Trained Quantization

weights (32 bit float)

2.09 -0.98 1.48 0.09
0.05 -0.14 -1.08 2.12
-0.91 1.92 0 -1.03
1.87 0 1.53 1.49

centroids

3: 2.00
2: 1.50
1: 0.00
0: -1.00

Cluster index (2 bit uint)

Figure 2: Representing the matrix sparsity with relative index. Padding filler zero to prevent overflow.

Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

To compress further, we store the index difference instead of the absolute position, and encode this difference in 8 bits for conv layer and 5 bits for fc layer. When we need an index difference larger than the bound, we use the zero padding solution shown in Figure 2: in case when the difference exceeds 8, the largest 3-bit (as an example) unsigned number, we add a filler zero.

3 TRAINED QUANTIZATION AND WEIGHT SHARING

Network quantization and weight sharing further compresses the pruned network by reducing the number of bits required to represent each weight. We limit the number of effective weights we need to store by having multiple connections share the same weight, and then fine-tune those shared weights.

Weight sharing is illustrated in Figure 3. Suppose we have a layer that has 4 input neurons and 4 output neurons, the weight is a 4×4 matrix. On the top left is the 4×4 weight matrix, and on the bottom left is the 4×4 gradient matrix. The weights are quantized to 4 bins (denoted with 4 colors), all the weights in the same bin share the same value, thus for each weight, we then need to store only a small index into a table of shared weights. During update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from last iteration. For pruned AlexNet, we are able to quantize to 8-bits (256 shared weights) for each CONV layers, and 5-bits (32 shared weights) for each FC layer without any loss of accuracy.

To calculate the compression rate, given k clusters, we only need log2(k) bits to encode the index. In general, for a network with n connections and each connection is represented with b bits, constraining the connections to have only k shared weights will result in a compression rate of:

\[ r = \frac{nb}{n} \log_2(k) + k \cdot b \]
Trained Quantization

[Han et al. ICLR’16]

weights (32 bit float) | cluster index (2 bit uint) | centroids
---|---|---
2.09 | 3 0 2 1 | 2.00
0.05 | 1 1 0 3 | 1.50
-0.91 | 0 3 1 0 | 0.00
1.87 | 3 1 2 2 | -1.00
-0.98 | -1.03 | |
Trained Quantization

weights (32 bit float)

<table>
<thead>
<tr>
<th>2.09</th>
<th>-0.98</th>
<th>1.48</th>
<th>0.09</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>-0.14</td>
<td>-1.08</td>
<td>2.12</td>
</tr>
<tr>
<td>-0.91</td>
<td>1.92</td>
<td>0</td>
<td>-1.03</td>
</tr>
<tr>
<td>1.87</td>
<td>0</td>
<td>1.53</td>
<td>1.49</td>
</tr>
</tbody>
</table>

centroids

<table>
<thead>
<tr>
<th>cluster index (2 bit uint)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

gradient

<table>
<thead>
<tr>
<th>-0.03</th>
<th>-0.01</th>
<th>0.03</th>
<th>0.02</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.01</td>
<td>0.01</td>
<td>-0.02</td>
<td>0.12</td>
</tr>
<tr>
<td>-0.01</td>
<td>0.02</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>-0.07</td>
<td>-0.02</td>
<td>0.01</td>
<td>-0.02</td>
</tr>
</tbody>
</table>

[Han et al. ICLR’16]
Trained Quantization

Weights (32 bit float):

-0.98 1.48 0.09
-0.14 -1.08 2.12
1.92 0 -1.03
0 1.53 1.49

Cluster index (2 bit uint):

3 0 2 1
1 1 0 3
0 3 1 0
3 1 2 2

Centroids:

3: 2.00
2: 1.50
1: 0.00
0: -1.00

Gradient:

-0.03 -0.01 0.03 0.02
-0.01 0.01 -0.02 0.12
-0.01 0.02 0.04 0.01
-0.07 -0.02 0.01 -0.02

Group by:

-0.03 0.12 0.02 -0.07
0.03 0.01 -0.02
0.02 -0.01 0.01 0.04 -0.02
-0.01 -0.02 -0.01 0.01
Trained Quantization

weights (32 bit float)

2.09  -0.98  1.48  0.09
0.05  -0.14  -1.08  2.12
-0.91  1.92  0  -1.03
1.87  0  1.53  1.49

cluster index (2 bit uint)

3  0  2  1
1  1  0  3
0  3  1  0
3  1  2  2

centroids

3:  2.00
2:  1.50
1:  0.00
0:  -1.00

gradient

-0.03  -0.01  0.03  0.02
-0.01  0.01  -0.02  0.12
-0.01  0.02  0.04  0.01
-0.07  -0.02  0.01  -0.02

group by

-0.03  0.12  0.02  -0.07
0.03  0.01  -0.02
0.02  -0.01  0.01  0.04  -0.02
-0.01  -0.02  -0.01  0.01

reduce

0.04
0.02
0.04
-0.03

[Han et al. ICLR'16]
Trained Quantization

Network quantization and weight sharing further compresses the pruned network by reducing the number of bits required to represent each weight. We limit the number of effective weights we need to store by having multiple connections share the same weight, and then fine-tune those shared weights.

Weight sharing is illustrated in Figure 3. Suppose we have a layer that has 4 input neurons and 4 output neurons, the weight is a $4 \times 4$ matrix. On the top left is the $4 \times 4$ weight matrix, and on the bottom left is the $4 \times 4$ gradient matrix. The weights are quantized to 4 bins (denoted with 4 colors), all the weights in the same bin share the same value, thus for each weight, we then need to store only a small index into a table of shared weights. During update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from last iteration. For pruned AlexNet, we are able to quantize to 8-bits (256 shared weights) for each CONV layers, and 5-bits (32 shared weights) for each FC layer without any loss of accuracy.

To calculate the compression rate, given $k$ clusters, we only need $\log_2(k)$ bits to encode the index. In general, for a network with $n$ connections and each connection is represented with $b$ bits, constraining the connections to have only $k$ shared weights will result in a compression rate of:

$$r = nb \log_2(k) + k$$

For example, Figure 3 shows the weights of a single layer neural network with four input units and four output units. There are $4 \times 4 = 16$ weights originally but there are only $4$ shared weights: similar weights are grouped together to share the same value. Originally we need to store 16 weights each.
Before Trained Quantization: Continuous Weight

[Han et al. ICLR'16]
After Trained Quantization: Discrete Weight

[Han et al. ICLR’16]
After Trained Quantization: Discrete Weight after Training

[Han et al. ICLR’16]
How Many Bits do We Need?
How Many Bits do We Need?

Han et al. ICLR’16

Number of bits per effective weight in all Conv layers

Number of bits per effective weight in all FC layers
Pruning + Trained Quantization Work Together
Pruning + Trained Quantization Work Together

AlexNet on ImageNet

[Han et al. ICLR’16]
Huffman Coding

- In-frequent weights: use more bits to represent
- Frequent weights: use less bits to represent

Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by $10 \times$, while quantization further improves the compression rate: between $27 \times$ and $31 \times$. Huffman coding gives more compression: between $35 \times$ and $49 \times$. The compression rate already included the meta-data for sparse representation. The compression scheme doesn't incur any accuracy loss.

Features such as better privacy, less network bandwidth and real time processing, the large storage overhead prevents deep neural networks from being incorporated into mobile apps. The second issue is energy consumption. Running large neural networks require a lot of memory bandwidth to fetch the weights and a lot of computation to do dot products— which in turn consumes considerable energy. Mobile devices are battery constrained, making power hungry applications such as deep neural networks hard to deploy.

Energy consumption is dominated by memory access. Under 45nm CMOS technology, a 32 bit floating point add consumes $0.9pJ$, a 32bit SRAM cache access takes $5pJ$, while a 32bit DRAM memory access takes $640pJ$, which is 3 orders of magnitude of an add operation. Large networks do not fit in on-chip storage and hence require the more costly DRAM accesses. Running a 1 billion connection neural network, for example, at 20fps would require $(20Hz)(1G)(640pJ) = 12.8W$ just for DRAM access - well beyond the power envelope of a typical mobile device.

Our goal is to reduce the storage and energy required to run inference on such large networks so they can be deployed on mobile devices. To achieve this goal, we present "deep compression": a three-stage pipeline (Figure 1) to reduce the storage required by neural network in a manner that preserves the original accuracy. First, we prune the networking by removing the redundant connections, keeping only the most informative connections. Next, the weights are quantized so that multiple connections share the same weight, thus only the codebook (effective weights) and the indices need to be stored. Finally, we apply Huffman coding to take advantage of the biased distribution of effective weights. Our main insight is that, pruning and trained quantization are able to compress the network without interfering each other, thus lead to surprisingly high compression rate. It makes the required storage so small (a few megabytes) that all weights can be cached on chip instead of going to off-chip DRAM which is energy consuming. Based on "deep compression", the EIE hardware accelerator Han et al. (2016) was later proposed that works on the compressed model, achieving significant speedup and energy efficiency improvement.

2 NETWORK PRUNING

Network pruning has been widely studied to compress CNN models. In early work, network pruning proved to be a valid way to reduce the network complexity and over-fitting (LeCun et al., 1989; Hanson & Pratt, 1989; Hassibi et al., 1993; Strö̈m, 1997). Recently Han et al. (2015) pruned state-of-the-art CNN models with no loss of accuracy. We build on top of that approach. As shown on the left side of Figure 1, we start by learning the connectivity via normal network training. Next, we prune the small-weight connections: all connections with weights below a threshold are removed from the network. Finally, we retrain the network to learn the final weights for the remaining sparse connections. Pruning reduced the number of parameters by $9 \times$ and $13 \times$ for AlexNet and VGG-16 model.
Summary of Deep Compression

- **Pruning:** less number of weights
  - Train Connectivity
  - Prune Connections
  - Train Weights

- **Quantization:** less bits per weight
  - Cluster the Weights
  - Generate Code Book
  - Quantize the Weights with Code Book
  - Retrain Code Book

- **Huffman Encoding**
  - Encode Weights
  - Encode Index

**Figure 1:** The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by $10 \times - 13 \times$, while quantization further improves the compression rate: between $27 \times - 31 \times$. Huffman coding gives more compression: between $35 \times - 49 \times$. The compression rate already included the meta-data for sparse representation. The compression scheme doesn't incur any accuracy loss.

- **Energy consumption:** Running large neural networks require a lot of memory bandwidth to fetch the weights and a lot of computation to do dot products—which in turn consumes considerable energy. Mobile devices are battery constrained, making power hungry applications such as deep neural networks hard to deploy.

- **Network pruning** has been widely studied to compress CNN models. In early work, network pruning proved to be a valid way to reduce the network complexity and over-fitting (LeCun et al., 1989; Hanson & Pratt, 1989; Hassibi et al., 1993; Strom, 1997). Recently Han et al. (2015) pruned state-of-the-art CNN models with no loss of accuracy. We build on top of that approach. As shown on the left side of Figure 1, we start by learning the connectivity via normal network training. Next, we prune the small-weight connections: all connections with weights below a threshold are removed from the network. Finally, we retrain the network to learn the final weights for the remaining sparse connections. Pruning reduced the number of parameters by $9 \times - 13 \times$ for AlexNet and VGG-16 model.
## Results: Compression Ratio

<table>
<thead>
<tr>
<th>Network</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
<th>Original Accuracy</th>
<th>Compressed Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeNet-300</td>
<td>1070KB</td>
<td>27KB</td>
<td>40x</td>
<td>98.36%</td>
<td>98.42%</td>
</tr>
<tr>
<td>LeNet-5</td>
<td>1720KB</td>
<td>44KB</td>
<td>39x</td>
<td>99.20%</td>
<td>99.26%</td>
</tr>
<tr>
<td>AlexNet</td>
<td>240MB</td>
<td>6.9MB</td>
<td>35x</td>
<td>80.27%</td>
<td>80.30%</td>
</tr>
<tr>
<td>VGGNet</td>
<td>550MB</td>
<td>11.3MB</td>
<td>49x</td>
<td>88.68%</td>
<td>89.09%</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>28MB</td>
<td>2.8MB</td>
<td>10x</td>
<td>88.90%</td>
<td>88.92%</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>44.6MB</td>
<td>4.0MB</td>
<td>11x</td>
<td>89.24%</td>
<td>89.28%</td>
</tr>
</tbody>
</table>

Can we make compact models to begin with?

[Han et al. ICLR’16]
SqueezeNet

Iandola et al, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”, arXiv 2016
### Compressing SqueezeNet

<table>
<thead>
<tr>
<th>Network</th>
<th>Approach</th>
<th>Size</th>
<th>Ratio</th>
<th>Top-1 Accuracy</th>
<th>Top-5 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>-</td>
<td>240MB</td>
<td>1x</td>
<td>57.2%</td>
<td>80.3%</td>
</tr>
<tr>
<td>AlexNet</td>
<td>SVD</td>
<td>48MB</td>
<td>5x</td>
<td>56.0%</td>
<td>79.4%</td>
</tr>
<tr>
<td>AlexNet</td>
<td>Deep Compression</td>
<td>6.9MB</td>
<td>35x</td>
<td>57.2%</td>
<td>80.3%</td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>-</td>
<td>4.8MB</td>
<td>50x</td>
<td>57.5%</td>
<td>80.3%</td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>Deep Compression</td>
<td>0.47MB</td>
<td>510x</td>
<td>57.5%</td>
<td>80.3%</td>
</tr>
</tbody>
</table>

Iandola et al, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”, arXiv 2016
Results: Speedup

- **Average Speedup**: The graph shows the speedup achieved for various models and configurations, averaging across different models and configurations.

- **Comparison Across Devices**: The graph compares performance across CPU, GPU, and mGPU, with distinct bars for each device type highlighting the speedup.

- **Model Specifics**: Specific models such as Alex-6, Alex-7, Alex-8, VGG-5, VGG-7, NT-Wa, NT-Wd, and NT-LSTM are analyzed, each with their respective speedup values.

- **Geo Mean**: The geometric mean of the speedup values is also shown, providing a single average value across all models and configurations.
Results: Energy Efficiency

- **Compression**
- **Acceleration**
- **Regularization**

- **CPU Dense (Baseline)**
- **CPU Compressed**
- **GPU Dense**
- **GPU Compressed**
- **mGPU Dense**
- **mGPU Compressed**

- **Energy Efficiency**

- **Average**

- **Geo Mean**
Part 1: Algorithms for Efficient Inference

- 1. Pruning
- 2. Weight Sharing
- 3. Quantization
- 4. Low Rank Approximation
- 5. Binary / Ternary Net
- 6. Winograd Transformation
Quantizing the Weight and Activation

- Train with float
- Quantizing the weight and activation:
  - Gather the statistics for weight and activation
  - Choose proper radix point position
- Fine-tune in float format
- Convert to fixed-point format

Qiu et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network, FPGA’16
Quantization Result

Qiu et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network, FPGA’16
Part 1: Algorithms for Efficient Inference

• 1. Pruning
• 2. Weight Sharing
• 3. Quantization
• 4. Low Rank Approximation
• 5. Binary / Ternary Net
• 6. Winograd Transformation
Low Rank Approximation for Conv

- Layer responses lie in a low-rank subspace
- Decompose a convolutional layer with $d$ filters with filter size $k \times k \times c$ to
  - A layer with $d'$ filters ($k \times k \times c$)
  - A layer with $d$ filter ($1 \times 1 \times d'$)

Zhang et al Efficient and Accurate Approximations of Nonlinear Convolutional Networks CVPR’15
## Low Rank Approximation for Conv

<table>
<thead>
<tr>
<th>speedup</th>
<th>rank sel.</th>
<th>Conv1</th>
<th>Conv2</th>
<th>Conv3</th>
<th>Conv4</th>
<th>Conv5</th>
<th>Conv6</th>
<th>Conv7</th>
<th>err. ↑ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2×</td>
<td>no</td>
<td>32</td>
<td>110</td>
<td>199</td>
<td>219</td>
<td>219</td>
<td>219</td>
<td>219</td>
<td>1.18</td>
</tr>
<tr>
<td>2×</td>
<td>yes</td>
<td>32</td>
<td>83</td>
<td>182</td>
<td>211</td>
<td>239</td>
<td>237</td>
<td>253</td>
<td><strong>0.93</strong></td>
</tr>
<tr>
<td>2.4×</td>
<td>no</td>
<td>32</td>
<td>96</td>
<td>174</td>
<td>191</td>
<td>191</td>
<td>191</td>
<td>191</td>
<td>1.77</td>
</tr>
<tr>
<td>2.4×</td>
<td>yes</td>
<td>32</td>
<td>74</td>
<td>162</td>
<td>187</td>
<td>207</td>
<td>205</td>
<td>219</td>
<td>1.35</td>
</tr>
<tr>
<td>3×</td>
<td>no</td>
<td>32</td>
<td>77</td>
<td>139</td>
<td>153</td>
<td>153</td>
<td>153</td>
<td>153</td>
<td>2.56</td>
</tr>
<tr>
<td>3×</td>
<td>yes</td>
<td>32</td>
<td>62</td>
<td>138</td>
<td>149</td>
<td>166</td>
<td>162</td>
<td>167</td>
<td>2.34</td>
</tr>
<tr>
<td>4×</td>
<td>no</td>
<td>32</td>
<td>57</td>
<td>104</td>
<td>115</td>
<td>115</td>
<td>115</td>
<td>115</td>
<td>4.32</td>
</tr>
<tr>
<td>4×</td>
<td>yes</td>
<td>32</td>
<td>50</td>
<td>112</td>
<td>114</td>
<td>122</td>
<td>117</td>
<td>119</td>
<td><strong>4.20</strong></td>
</tr>
<tr>
<td>5×</td>
<td>no</td>
<td>32</td>
<td>46</td>
<td>83</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>6.53</td>
</tr>
<tr>
<td>5×</td>
<td>yes</td>
<td>32</td>
<td>41</td>
<td>94</td>
<td>93</td>
<td>98</td>
<td>92</td>
<td>90</td>
<td><strong>6.47</strong></td>
</tr>
</tbody>
</table>
Low Rank Approximation for FC

Build a mapping from row / column indices of matrix $W = [W(x, y)]$ to vectors $i$ and $j$: $x \leftrightarrow i = (i_1, \ldots, i_d)$ and $y \leftrightarrow j = (j_1, \ldots, j_d)$.

TT-format for matrix $W$:

$$W(i_1, \ldots, i_d; j_1, \ldots, j_d) = W(x(i), y(j)) = \underbrace{G_1[i_1, j_1]}_{1 \times r} \underbrace{G_2[i_2, j_2]}_{r \times r} \cdots \underbrace{G_d[i_d, j_d]}_{r \times 1}$$

<table>
<thead>
<tr>
<th>Type</th>
<th>1 im. time (ms)</th>
<th>100 im. time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU fully-connected layer</td>
<td>16.1</td>
<td>97.2</td>
</tr>
<tr>
<td>CPU TT-layer</td>
<td>1.2</td>
<td>94.7</td>
</tr>
<tr>
<td>GPU fully-connected layer</td>
<td>2.7</td>
<td>33</td>
</tr>
<tr>
<td>GPU TT-layer</td>
<td>1.9</td>
<td>12.9</td>
</tr>
</tbody>
</table>

Novikov et al Tensorizing Neural Networks, NIPS’15
Part 1: Algorithms for Efficient Inference

• 1. Pruning
• 2. Weight Sharing
• 3. Quantization
• 4. Low Rank Approximation
• 5. Binary / Ternary Net
• 6. Winograd Transformation
We use the simple heuristic to quantify the importance of the weights using their absolute value. The weight distribution is very representative for VGGNet and ResNet as well. The original distribution with 1/10 the original learning rate (since the sparse network is already at a good local minima).

To visualize the DSD training flow, we plotted the progression of weight distribution in Figure 2. The figure is plotted using GoogLeNet inception_5b3x3 layer, and we found that this progression of weights (d), and after retraining the dense network (e).

Initial Dense Training:

The reason behind removing small weight is partially due to the Taylor expansion of the loss function, percentage of weights that are pruned to 0. For each layer applied the means smaller increase to the loss function.

Final Dense Training:

The final D step recovers the pruned connections, making the network dense again and are reinitialized to zero. Finally, in (e), the previously-pruned weights are retrained together themselves during the retraining phase, so in (c) the boundary becomes soft and forms a bimodal distribution. In (d), at the beginning of the re-dense training step, all the pruned weights come back again and are reinitialized to zero. Finally, in (e), the previously-pruned weights are retrained together after pruning the large center region is truncated away. The network parameters un-truncated adjust of weight is centered on zero with tails dropping off quickly. Pruning is based on absolute value so arrive at a better local minima compared with the sparse model from S step.

Hyper parameters like dropout ratios and weight decay remained unchanged. By restoring the pruned out parameters, picked the k-th largest one generated a binary mask to remove all the weights smaller than 0.

Train on Dense (D)

Recover Zero Weights

Quantize

Pruning the Network

Inference Time

Normalized
Error Rate on ImageNet

Validation

- Top1 38%
- Top5 12%
- Top1 42.8%
- Top5 19.8%

DoReFa-Net, TWN, Ours, Full precision (with Dropout)
Part 1: Algorithms for Efficient Inference

- 1. Pruning
- 2. Weight Sharing
- 3. Quantization
- 4. Low Rank Approximation
- 5. Binary / Ternary Net
- 6. Winograd Transformation
3x3 DIRECT Convolutions

Compute Bound

Direct convolution: we need $9 \times C \times 4 = 36 \times C$ FMAs for 4 outputs

Julien Demouth, Convolution OPTIMIZATION: Winograd, NVIDIA
3x3 WINOGRAD Convolutions
Transform Data to Reduce Math Intensity

Direct convolution: we need $9 \times C \times 4 = 36 \times C$ FMAs for 4 outputs
Winograd convolution: we need $16 \times C$ FMAs for 4 outputs: $2.25 \times$ fewer FMAs

See A. Lavin & S. Gray, "Fast Algorithms for Convolutional Neural Networks

Julien Demouth, Convolution OPTIMIZATION: Winograd, NVIDIA
Speedup of Winograd Convolution

VGG16, Batch Size 1 - Relative Performance

Measured on Maxwell TITAN X

Julien Demouth, Convolution OPTIMIZATION: Winograd, NVIDIA
Agenda

- Algorithms for Efficient Inference
- Algorithms for Efficient Training
- Hardware for Efficient Inference
- Hardware for Efficient Training
Hardware for Efficient Inference

a common goal: minimize memory access

Eyeriss
MIT
RS Dataflow

DaDiannao
CAS
eDRAM

TPU
Google
8-bit Integer

EIE
Stanford
Compression/Sparsity

“This unit is designed for dense matrices. Sparse architectural support was omitted for time-to-deploy reasons. Sparsity will have high priority in future designs”
Google TPU

TPU Card to replace a disk
Up to 4 cards / server
Google TPU

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
- 700 MHz clock rate
- Peak: 92T operations/second
  - 65,536 * 2 * 700M
- >25X as many MACs vs GPU
- >100X as many MACs vs CPU
- 4 MiB of on-chip Accumulator memory
- 24 MiB of on-chip Unified Buffer (activation memory)
- 3.5X as much on-chip memory vs GPU
- Two 2133MHz DDR3 DRAM channels
- 8 GiB of off-chip weight DRAM memory

TPU: High-level Chip Architecture
# Google TPU

<table>
<thead>
<tr>
<th>Processor</th>
<th>$mm^2$</th>
<th>Clock MHz</th>
<th>TDP Watts</th>
<th>Idle Watts</th>
<th>Memory GB/sec</th>
<th>Peak TOPS/chip 8b int.</th>
<th>32b FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Haswell (18 core)</td>
<td>662</td>
<td>2300</td>
<td>145</td>
<td>41</td>
<td>51</td>
<td>2.6</td>
<td>1.3</td>
</tr>
<tr>
<td>GPU: Nvidia K80 (2 / card)</td>
<td>561</td>
<td>560</td>
<td>150</td>
<td>25</td>
<td>160</td>
<td>--</td>
<td>2.8</td>
</tr>
<tr>
<td>TPU</td>
<td>&lt;331*</td>
<td>700</td>
<td>75</td>
<td>28</td>
<td>34</td>
<td>91.8</td>
<td>--</td>
</tr>
</tbody>
</table>

*TPU is less than half die size of the Intel Haswell processor

K80 and TPU in 28 nm process; Haswell fabbed in Intel 22 nm process

These chips and platforms chosen for comparison because widely deployed in Google data centers
## Inference Datacenter Workload

### Little data reuse, lots of memory footprint, lots of communication overhead

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Layers</th>
<th>Nonlinear function</th>
<th>Weights</th>
<th>TPU Ops / Weight Byte</th>
<th>TPU Batch Size</th>
<th>% Deployed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLP0</td>
<td>0.1k</td>
<td>FC: 5</td>
<td>Conv: 5</td>
<td>ReLU</td>
<td>20M</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>MLP1</td>
<td>1k</td>
<td>Conv: 4</td>
<td>Vector: 4</td>
<td>ReLU</td>
<td>5M</td>
<td>168</td>
<td>168</td>
</tr>
<tr>
<td>LSTM0</td>
<td>1k</td>
<td>FC: 24</td>
<td>Pool: 34</td>
<td>tanh,</td>
<td>52M</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>LSTM1</td>
<td>1.5k</td>
<td>Conv: 37</td>
<td>Vector: 19</td>
<td>tanh,</td>
<td>34M</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>CNN0</td>
<td>1k</td>
<td>Conv: 16</td>
<td>Pool: 16</td>
<td>ReLU</td>
<td>8M</td>
<td>2888</td>
<td>8</td>
</tr>
<tr>
<td>CNN1</td>
<td>1k</td>
<td>Conv: 4</td>
<td>Pool: 72</td>
<td>ReLU</td>
<td>100M</td>
<td>1750</td>
<td>32</td>
</tr>
</tbody>
</table>

### Lots of data reuse, less memory footprint, less communication overhead

Google TPU Team, In-Data Center Performance Analysis of a Tensor Processing Unit
Roofline Model: Identify Performance Bottleneck

\[ \text{GFLOP/s} = \min(\text{Peak GFLOP/s}, \text{Peak GB/s} \times \text{AI}) \]

![Diagram showing the Roofline Model](image_url)

In-Data Center Performance Analysis of a Tensor Processing Unit
TPU Roofline

Operational Intensity: Ops/weight byte (log scale)
Log Rooflines for CPU, GPU, TPU

In-Data Center Performance Analysis of a Tensor Processing Unit

★ Star = TPU
△ Triangle = GPU
○ Circle = CPU
Linear Rooflines for CPU, GPU, TPU

Operational Intensity: Ops/weight byte (linear scale)

Star = TPU
△ Triangle = GPU
〇 Circle = CPU

In-Data Center Performance Analysis of a Tensor Processing Unit
Why so far below Rooflines?
Low latency requirement => Can’t batch more => low ops/byte

How to Solve this?
less memory footprint => need compress the model

Challenge:
Hardware that can infer on compressed model
EIE: the First DNN Accelerator for Sparse, Compressed Model
EIE: the First DNN Accelerator for Sparse, Compressed Model

- **Sparse Weight**
  - 90% static sparsity
  - 10x less computation
  - 5x less memory footprint

- **Sparse Activation**
  - 70% dynamic sparsity
  - 3x less computation

- **Weight Sharing**
  - 4-bit weights
  - 8x less memory footprint

\[ 0 \times A = 0 \]
\[ W \times 0 = 0 \]
\[ 2.09, 1.92 \Rightarrow 2 \]

[Han et al. ISCA'16]
EIE: Reduce Memory Access by Compression

\[
\begin{align*}
\tilde{a} & = \begin{pmatrix} 0 & a_1 & 0 & a_3 \end{pmatrix} \\
\begin{pmatrix}
PE0 & w_{0,0} & w_{0,1} & 0 & w_{0,3} \\
PE1 & 0 & 0 & w_{1,2} & 0 \\
PE2 & 0 & w_{2,1} & 0 & w_{2,3} \\
PE3 & 0 & 0 & 0 & 0 \\
\end{pmatrix} & \times & \begin{pmatrix}
 b_0 \\
 b_1 \\
 -b_2 \\
 b_3 \\
 -b_4 \\
 b_5 \\
 b_6 \\
 -b_7 \\
\end{pmatrix} & = & \begin{pmatrix}
\text{ReLU} \\
\Rightarrow \\
 b_0 \\
 b_1 \\
 0 \\
 b_3 \\
 0 \\
 b_5 \\
 b_6 \\
 0 \\
\end{pmatrix}
\end{align*}
\]

 logically

physically

**Virtual Weight**

<table>
<thead>
<tr>
<th></th>
<th>(W_{0,0})</th>
<th>(W_{0,1})</th>
<th>(W_{4,2})</th>
<th>(W_{0,3})</th>
<th>(W_{4,3})</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Relative Index</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Column Pointer</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Dataflow

\[
\bar{a} \left( \begin{array}{cccc}
0 & a_1 & 0 & a_3 \\
\end{array} \right) \times \\
\begin{array}{c}
PE0 \\
PE1 \\
PE2 \\
PE3 \\
\end{array}
\left( \begin{array}{cccc}
w_{0,0} & w_{0,1} & 0 & w_{0,3} \\
0 & 0 & w_{1,2} & 0 \\
0 & w_{2,1} & 0 & w_{2,3} \\
0 & 0 & 0 & 0 \\
0 & 0 & w_{4,2} & w_{4,3} \\
w_{5,0} & 0 & 0 & 0 \\
0 & 0 & 0 & w_{6,3} \\
0 & w_{7,1} & 0 & 0 \\
\end{array} \right) = \\
\begin{array}{c}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7 \\
\end{array}
\Rightarrow ReLU
\begin{array}{c}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7 \\
\end{array}
\]

rule of thumb:
\[0 \times A = 0 \quad W \times 0 = 0\]
Dataflow

\[
\tilde{a} \begin{pmatrix}
0 & a_1 & 0 & a_3 \\
\end{pmatrix}
\times
\begin{pmatrix}
w_{0,0} & w_{0,1} & 0 & w_{0,3} \\
0 & 0 & w_{1,2} & 0 \\
0 & w_{2,1} & 0 & w_{2,3} \\
0 & 0 & 0 & 0 \\
w_{5,0} & 0 & 0 & 0 \\
0 & 0 & 0 & w_{6,3} \\
0 & w_{7,1} & 0 & 0 \\
\end{pmatrix}
= \begin{pmatrix}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7 \\
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7 \\
\end{pmatrix}
\]

rule of thumb:
\[0 \ast A = 0 \quad W \ast 0 = 0\]
Dataflow

\[
\begin{align*}
\tilde{a} & = \begin{pmatrix} 0 & a_1 & 0 & a_3 \end{pmatrix} \\
\begin{pmatrix} w_{0,0} & w_{0,1} & 0 & w_{0,3} \\
0 & 0 & w_{1,2} & 0 \\
0 & w_{2,1} & 0 & w_{2,3} \\
0 & 0 & 0 & 0 \\
w_{5,0} & 0 & 0 & 0 \\
0 & 0 & 0 & w_{6,3} \\
0 & w_{7,1} & 0 & 0 \\
\end{pmatrix} & \times \\
\begin{pmatrix} b_0 \\
b_1 \\
-b_2 \\
b_3 \\
-b_4 \\
b_5 \\
-b_6 \\
0 \\
\end{pmatrix} & = \\
\begin{pmatrix} b_0 \\
b_1 \\
0 \\
b_3 \\
0 \\
b_5 \\
b_6 \\
0 \\
\end{pmatrix} & \text{ReLU}
\end{align*}
\]

rule of thumb:
\[
0 \times A = 0 \quad W \times 0 = 0
\]
Dataflow

\[ \tilde{a} \begin{pmatrix} 0 & a_1 & 0 & a_3 \end{pmatrix} \times \begin{pmatrix} \begin{bmatrix} w_{0,0} & w_{0,1} & 0 & w_{0,3} \\ 0 & 0 & w_{1,2} & 0 \\ 0 & w_{2,1} & 0 & w_{2,3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & w_{4,2} & w_{4,3} \\ w_{5,0} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{6,3} \\ 0 & w_{7,1} & 0 & 0 \end{bmatrix} \end{bmatrix} \right) \begin{pmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ b_6 \\ -b_7 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0 \end{pmatrix} \]

ReLU

rule of thumb:
\[ 0 \times A = 0 \quad W \times 0 = 0 \]
**Dataflow**

\[
\tilde{a} \times \begin{pmatrix}
\begin{array}{cccc}
0 & a_1 & 0 & a_3 \\
\end{array}
\end{pmatrix}
\times
\begin{pmatrix}
\begin{array}{cccc}
\begin{pmatrix}
0,0 & w_{0,1} & 0 & w_{0,3} \\
0 & 0 & w_{1,2} & 0 \\
0 & w_{2,1} & 0 & w_{2,3} \\
0 & 0 & 0 & 0 \\
0 & 0 & w_{4,2} & w_{4,3} \\
0 & 0 & 0 & 0 \\
0 & w_{7,1} & 0 & 0 \\
\end{pmatrix}
\end{array}
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
\begin{array}{c}
b_0 \\
b_1 \\
-b_2 \\
0 \\
b_3 \\
b_4 \\
-b_5 \\
b_6 \\
-b_7 \\
\end{array}
\end{pmatrix}
\]

rule of thumb:
\[
0 \ast A = 0 \quad W \ast 0 = 0
\]
### Dataflow

<table>
<thead>
<tr>
<th>PE0</th>
<th>PE1</th>
<th>PE2</th>
<th>PE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_{0,0}$</td>
<td>$w_{0,1}$</td>
<td>$w_{1,2}$</td>
<td>$w_{0,3}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$w_{2,1}$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>$w_{2,1}$</td>
<td>0</td>
<td>$w_{3,2}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$w_{5,0}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>$w_{7,1}$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\tilde{a} \times \begin{pmatrix} \mathbf{w}_{0,0} & \mathbf{w}_{0,1} & \mathbf{w}_{1,2} & \mathbf{w}_{0,3} \\
\mathbf{0} & \mathbf{0} & \mathbf{w}_{2,1} & \mathbf{0} \\
\mathbf{0} & \mathbf{w}_{2,1} & \mathbf{0} & \mathbf{w}_{3,2} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{w}_{4,2} & \mathbf{w}_{4,3} \\
\mathbf{w}_{5,0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{w}_{6,3} & \mathbf{0} \\
\mathbf{0} & \mathbf{w}_{7,1} & \mathbf{0} & \mathbf{0} \\
\end{pmatrix} = \begin{pmatrix} \mathbf{b}_0 \\
\mathbf{b}_1 \\
-\mathbf{b}_2 \\
\mathbf{b}_3 \\
-\mathbf{b}_4 \\
\mathbf{b}_5 \\
\mathbf{b}_6 \\
-\mathbf{b}_7 \\
\end{pmatrix} \xrightarrow{ReLU} \begin{pmatrix} \mathbf{b}_0 \\
\mathbf{b}_1 \\
0 \\
\mathbf{b}_3 \\
0 \\
\mathbf{b}_5 \\
\mathbf{b}_6 \\
0 \\
\end{pmatrix}

### Rule of Thumb:

\[
0 \times \mathbf{A} = \mathbf{0} \quad \mathbf{W} \times \mathbf{0} = \mathbf{0}
\]
Dataflow

\[
\begin{bmatrix}
0 & a_1 & 0 & a_3 \\
\end{bmatrix}
\times
\begin{bmatrix}
w_{0,0} & w_{0,1} & 0 & w_{0,3} \\
0 & 0 & w_{1,2} & 0 \\
0 & w_{2,1} & 0 & w_{2,3} \\
0 & 0 & 0 & 0 \\
0 & 0 & w_{4,2} & w_{4,3} \\
w_{5,0} & 0 & 0 & 0 \\
0 & 0 & 0 & w_{6,3} \\
0 & w_{7,1} & 0 & 0 \\
\end{bmatrix}
= \begin{bmatrix}
b_0 \\
b_1 \\
-b_2 \\
b_3 \\
-b_4 \\
b_5 \\
b_6 \\
-b_7 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
b_0 \\
b_1 \\
b_3 \\
0 \\
0 \\
b_5 \\
b_6 \\
0 \\
\end{bmatrix}
\]

rule of thumb:
\[
0 \times A = 0 \quad W \times 0 = 0
\]
Dataflow

rule of thumb:
\[0 \times A = 0 \quad W \times 0 = 0\]
Dataflow

\[
\tilde{a} \times \begin{pmatrix}
w_{0,0} & w_{0,1} & 0 & w_{0,3} \\
0 & 0 & w_{1,2} & 0 \\
0 & w_{2,1} & 0 & w_{2,3} \\
0 & 0 & 0 & 0 \\
0 & 0 & w_{4,2} & w_{4,3} \\
w_{5,0} & 0 & 0 & 0 \\
0 & 0 & 0 & w_{6,3} \\
0 & w_{7,1} & 0 & 0
\end{pmatrix} = \begin{pmatrix}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7
\end{pmatrix}
\Rightarrow \text{ReLU}
\]

rule of thumb:
\[0 \ast A = 0 \quad W \ast 0 = 0\]
Dataflow

\[
\begin{aligned}
\tilde{a} &= \begin{pmatrix} 0 & a_1 & 0 & a_3 \end{pmatrix} \\
\times \\
\begin{pmatrix}
\begin{bmatrix} w_{0,0} & w_{0,1} & 0 & w_{0,3} \\
0 & 0 & w_{1,2} & 0 \\
0 & w_{2,1} & 0 & w_{2,3} \\
0 & 0 & 0 & 0 \\
0 & 0 & w_{4,2} & w_{4,3} \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & w_{6,3} \\
0 & w_{7,1} & 0 & 0
\end{bmatrix}
\end{bmatrix}
\end{aligned}
\]

\begin{align*}
\Rightarrow \quad \tilde{b} &= \begin{pmatrix} b_0 \\
\begin{bmatrix} b_1 \\
-b_2 \\
-b_3 \\
0 \\
-b_4 \\
b_5 \\
-b_6 \\
0
\end{bmatrix}
\end{pmatrix}
\end{align*}

\[
ReLU
\]

rule of thumb:

\[
0 * A = 0 \quad W * 0 = 0
\]
Large DNN models are very powerful but consume large amounts of energy because the model must be stored in systems [8].

Previously proposed ‘Deep Compression’ makes it possible to fit modern networks such as AlexNet (60M parameters, 240MB), and VGG-16 (130M parameters, 520MB) in on-chip SRAM. Processing these parameter reuse in FC layers. Data batching has become computationally and memory intensive, making them difficult to deploy on large networks [14]. Unlike CONV layers, there is no data reuse. The energy cost per fetch ranges from 5pJ for 32b coefficients in off-chip LPDDR2 DRAM to 640pJ for 32b coefficients in on-chip SRAM to 640pJ for 32b coefficients in on-chip SRAM.

While custom hardware helps the computation, the power budget is dominated by the required memory access if there is no data reuse. The energy cost per fetch ranges from 5pJ for 32b coefficients in off-chip LPDDR2 DRAM to 640pJ for 32b coefficients in on-chip SRAM to 640pJ for 32b coefficients in on-chip SRAM.

Accelerating dense, uncompressed models - limiting their utility in speech sample. For embedded mobile applications, neural network model for machine learning applications. Figure 1. Efficient inference engine that works on the compressed deep neural network.
Micro Architecture for each PE

[Han et al. ISCA'16]
Speedup on EIE

![Speedup graph](image)

**Comparison Baseline.** We compare EIE with three different baselines: 1) CPU. 2) Mobile GPU. 3) mGPU. We use NVIDIA Tegra K1 that has 192 CUDA cores as our mobile GPU baseline. We used a CPU socket and DRAM power meter, then assumed that it doesn't have software interface to report power consumption, so we measured the total power consumption with a power-meter, then assumed that the CPU socket and DRAM behaves the same way as a mGPU. We used PARSE CSRMV for the compressed sparse model. Tegra K1 has 288 CUDA cores as our mobile GPU baseline. We used cuBLAS GEMV for the original dense model and cuSPARSE CSRMV kernel, which is optimized for sparse matrices. We stored the sparse matrix in in CSR format, and used cuSPARSE CSRMV kernel to perform the matrix-vector multiplication on GPUs.

**Speedup on EIE**

- **CPU Dense (Baseline)**
- **CPU Compressed**
- **GPU Dense**
- **GPU Compressed**
- **mGPU Dense**
- **mGPU Compressed**
- **EIE**

**Figure 5** shows the layout (after place-and-route) of an EIE processing element. The power/area breakdown is as follows: SpMat 4096, 25% 37.5% 10%, ActRW 1.122 (12.25%) 18,934 (2.97%), ArithmUnit 1.162 (12.68%) 3,110 (0.49%), SpmatRead 4.955 (54.11%) 469,412 (73.57%), ActRW 1.122 (12.25%) 18,934 (2.97%).

**Figure 6** shows the speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

**Geo Mean**

- **CPU** Geo Mean 1x
- **GPU** Geo Mean 10000x
- **mGPU** Geo Mean 102x
- **EIE** Geo Mean 189x

**Activation Read/Write.** Each activation register file holds 64 16-bit activations. The activation is broadcast back to all the PEs via a separate wire placed in an H-tree. Each LNZD node finds the next non-zero activation across sparsity, this requires a dense DNN accelerator 3TOP/s to model the RTL behavior of synchronous circuits. Each activation register file holds 64 16-bit activations. A single 3-bits select one of the eight entries in that row. A single b Catalyst selects an SRAM row, and the low p Catalyst combines with the high p Catalyst to form the data path that is stored in a single register.

**Table III**

<table>
<thead>
<tr>
<th>Model</th>
<th>Alex-6</th>
<th>Alex-7</th>
<th>Alex-8</th>
<th>VGG-6</th>
<th>VGG-7</th>
<th>VGG-8</th>
<th>NT-We</th>
<th>NT-Wd</th>
<th>NT-LSTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2.5x</td>
<td>3x</td>
<td>1.5x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>GPU</td>
<td>14x</td>
<td>9x</td>
<td>3.5x</td>
<td>3.0x</td>
<td>3.0x</td>
<td>3.0x</td>
<td>3.0x</td>
<td>4.5x</td>
<td>15x</td>
</tr>
<tr>
<td>mGPU</td>
<td>60x</td>
<td>63x</td>
<td>59x</td>
<td>25088</td>
<td>18.3%</td>
<td>1%</td>
<td>20x</td>
<td>5x</td>
<td>102x</td>
</tr>
<tr>
<td>EIE</td>
<td>0.1x</td>
<td>0.1x</td>
<td>0.1x</td>
<td>0.1x</td>
<td>0.1x</td>
<td>0.1x</td>
<td>0.1x</td>
<td>0.1x</td>
<td>0.1x</td>
</tr>
</tbody>
</table>

**Comparison with NeuralTalk.** We compare EIE with three different baselines: 1) CPU. 2) Mobile GPU. 3) mGPU. We use NVIDIA Tegra K1 that has 192 CUDA cores as our mobile GPU baseline. We use cuBLAS GEMV for the original dense model and cuSPARSE CSRMV kernel, which is optimized for sparse matrices. We stored the sparse matrix in in CSR format, and used cuSPARSE CSRMV kernel to perform the matrix-vector multiplication on GPUs.
Energy Efficiency on EIE

- **CPU Dense (Baseline)**
- **CPU Compressed**
- **GPU Dense**
- **GPU Compressed**
- **mGPU Dense**
- **mGPU Compressed**
- **EIE**

**BENCHMARK FROM STATE-OF-THE-ART DNN MODELS**

- **Alex-6**
- **Alex-7**
- **Alex-8**
- **VGG-6**
- **VGG-7**
- **VGG-8**
- **NT-We**
- **NT-Wd**
- **NT-LSTM**

Geo Mean: **24,207x**

- **CPU**
- **GPU**
- **mGPU**
- **EIE**
Comparison: Throughput

Throughput (Layers/s in log scale)

- Core-i7 5930k CPU (22nm)
- TitanX GPU (28nm)
- Tegra K1 mGPU (28nm)
- A-Eye FPGA (28nm)
- DaDianNao ASIC (28nm)
- TrueNorth ASIC (28nm)
- EIE ASIC (45nm)
- EIE ASIC (28nm, 64PEs)
- EIE ASIC (28nm, 256PEs)

[Han et al. ISCA'16]
Comparison: Energy Efficiency

Energy Efficiency (Layers/J in log scale)

- CPU: Core-i7 5930k, 22nm CPU
- GPU: TitanX, 28nm GPU
- mGPU: Tegra K1, 28nm mGPU
- FPGA: A-Eye, 28nm FPGA
- ASIC: DaDianNao, 28nm ASIC
- ASIC: TrueNorth, 28nm ASIC
- ASIC: EIE, 45nm ASIC 64PEs
- ASIC: EIE, 28nm ASIC 256PEs

[Han et al. ISCA’16]
Agenda

- Algorithms for Efficient Inference
- Algorithms for Efficient Training
- Hardware for Efficient Inference
- Hardware for Efficient Training
Part 3: Efficient Training — Algorithms

• 1. Parallelization

• 2. Mixed Precision with FP16 and FP32

• 3. Model Distillation

• 4. DSD: Dense-Sparse-Dense Training
Part 3: Efficient Training — Algorithms

• 1. Parallelization

• 2. Mixed Precision with FP16 and FP32

• 3. Model Distillation

• 4. DSD: Dense-Sparse-Dense Training
Moore’s law made CPUs 300x faster than in 1990
But its over...

C Moore, Data Processing in ExaScale-Class Computer Systems, Salishan, April 2011
Data Parallel – Run multiple inputs in parallel

Dally, High Performance Hardware for Machine Learning, NIPS'2015
Data Parallel – Run multiple inputs in parallel

- Doesn’t affect latency for one input
- Requires P-fold larger batch size
- For training requires coordinated weight update

Dally, High Performance Hardware for Machine Learning, NIPS’2015
Parameter Update

Parameter Server: $p' = p + \Delta p$

Model!
Workers

Data!
Shards

Large Scale Distributed Deep Networks, Jeff Dean et al., 2013
Model Parallel
Split up the Model – i.e. the network

Dally, High Performance Hardware for Machine Learning, NIPS'2015
Model-Parallel Convolution – by output region \((x,y)\)

6D Loop

For all output map \(j\)
- For each input map \(k\)
  - For each pixel \(x,y\)
    - For each kernel element \(u,v\)
      \[
      B_{xyj} += A_{(x-u)(y-v)k} \times K_{uvkj}
      \]

Dally, High Performance Hardware for Machine Learning, NIPS'2015
Model-Parallel Convolution – By output map $j$

(filter)

6D Loop
For all output map $j$
For each input map $k$
For each pixel $x,y$
For each kernel element $u,v$

$$B_{xyj} += A_{(x-u)(y-v)k} \times K_{uvkj}$$

Dally, High Performance Hardware for Machine Learning, NIPS'2015
Model Parallel Fully-Connected Layer (M x V)

\[ b_i = W_{ij} \times a_j \]

Dally, High Performance Hardware for Machine Learning, NIPS'2015
Model Parallel Fully-Connected Layer (M x V)

Dally, High Performance Hardware for Machine Learning, NIPS'2015
Hyper-Parameter Parallel
Try many alternative networks in parallel

Dally, High Performance Hardware for Machine Learning, NIPS’2015
Summary of Parallelism

• Lots of parallelism in DNNs
  • 16M independent multiplies in one FC layer
  • Limited by overhead to exploit a fraction of this

• Data parallel
  • Run multiple training examples in parallel
  • Limited by batch size

• Model parallel
  • Split model over multiple processors
  • By layer
  • Conv layers by map region
  • Fully connected layers by output activation

• Easy to get 16-64 GPUs training one model in parallel

Dally, High Performance Hardware for Machine Learning, NIPS’2015
Part 3: Efficient Training — Algorithms

1. Parallelization
2. Mixed Precision with FP16 and FP32
3. Model Distillation
4. DSD: Dense-Sparse-Dense Training
Mixed Precision

Mixed Precision Training

Boris Ginsburg, Sergei Nikolaev, Paulius Micikevicius, “Training with mixed precision”, NVIDIA GTC 2017
Inception V1

Boris Ginsburg, Sergei Nikolaev, Paulius Micikevicius, “Training with mixed precision”, NVIDIA GTC 2017
ResNet

Boris Ginsburg, Sergei Nikolaev, Paulius Micikevicius, “Training with mixed precision”, NVIDIA GTC 2017
### AlexNet

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1 accuracy, %</th>
<th>Top5 accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>58.62</td>
<td>81.25</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>58.12</td>
<td>80.71</td>
</tr>
</tbody>
</table>

### Inception V3

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1 accuracy, %</th>
<th>Top5 accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>71.75</td>
<td>90.52</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>71.17</td>
<td>90.10</td>
</tr>
</tbody>
</table>

### ResNet-50

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1 accuracy, %</th>
<th>Top5 accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>73.85</td>
<td>91.44</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>73.6</td>
<td>91.11</td>
</tr>
</tbody>
</table>

Boris Ginsburg, Sergei Nikolaev, Paulius Micikevicius, “Training with mixed precision”, NVIDIA GTC 2017
Part 3: Efficient Training Algorithm

• 1. Parallelization
• 2. Mixed Precision with FP16 and FP32
• 3. Model Distillation
• 4. DSD: Dense-Sparse-Dense Training
Model Distillation

Teacher model 1 (Googlenet)
Teacher model 2 (Vggnet)
Teacher model 3 (Resnet)

student model

Knowledge

student model has much smaller model size
Softened outputs reveal the dark knowledge

<table>
<thead>
<tr>
<th></th>
<th>cow</th>
<th>dog</th>
<th>cat</th>
<th>car</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

original hard targets

<table>
<thead>
<tr>
<th></th>
<th>cow</th>
<th>dog</th>
<th>cat</th>
<th>car</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^{-6}</td>
<td>.9</td>
<td>.1</td>
<td>10^{-9}</td>
<td></td>
</tr>
</tbody>
</table>

output of geometric ensemble

<table>
<thead>
<tr>
<th></th>
<th>cow</th>
<th>dog</th>
<th>cat</th>
<th>car</th>
</tr>
</thead>
<tbody>
<tr>
<td>.05</td>
<td>.3</td>
<td>.2</td>
<td>.005</td>
<td></td>
</tr>
</tbody>
</table>

softened output of ensemble

Hinton et al. Dark knowledge / Distilling the Knowledge in a Neural Network
Softened outputs reveal the dark knowledge

\[ p_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)} \]

- Method: Divide score by a “temperature” to get a much softer distribution

- Result: Start with a trained model that classifies 58.9% of the test frames correctly. The new model converges to 57.0% correct even when it is only trained on 3% of the data

Hinton et al. Dark knowledge / Distilling the Knowledge in a Neural Network
Part 3: Efficient Training Algorithm

• 1. Parallelization
• 2. Mixed Precision with FP16 and FP32
• 3. Model Distillation
• 4. DSD: Dense-Sparse-Dense Training
DSD produces same model architecture but can find better optimization solution, arrives at better local minima, and achieves higher prediction accuracy across a wide range of deep neural networks on CNNs / RNNs / LSTMs.

Han et al. “DSD: Dense-Sparse-Dense Training for Deep Neural Networks”, ICLR 2017
DSD: Intuition

learn the trunk first
then learn the leaves

Han et al. “DSD: Dense-Sparse-Dense Training for Deep Neural Networks”, ICLR 2017
### DSD is General Purpose: Vision, Speech, Natural Language

<table>
<thead>
<tr>
<th>Network</th>
<th>Domain</th>
<th>Dataset</th>
<th>Type</th>
<th>Baseline</th>
<th>DSD</th>
<th>Abs. Imp.</th>
<th>Rel. Imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoogleNet</td>
<td>Vision</td>
<td>ImageNet</td>
<td>CNN</td>
<td>31.1%</td>
<td>30.0%</td>
<td>1.1%</td>
<td>3.6%</td>
</tr>
<tr>
<td>VGG-16</td>
<td>Vision</td>
<td>ImageNet</td>
<td>CNN</td>
<td>31.5%</td>
<td>27.2%</td>
<td>4.3%</td>
<td>13.7%</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>Vision</td>
<td>ImageNet</td>
<td>CNN</td>
<td>30.4%</td>
<td>29.3%</td>
<td>1.1%</td>
<td>3.7%</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>Vision</td>
<td>ImageNet</td>
<td>CNN</td>
<td>24.0%</td>
<td>23.2%</td>
<td>0.9%</td>
<td>3.5%</td>
</tr>
<tr>
<td>NeuralTalk</td>
<td>Caption</td>
<td>Flickr-8K</td>
<td>LSTM</td>
<td>16.8</td>
<td>18.5</td>
<td>1.7</td>
<td>10.1%</td>
</tr>
<tr>
<td>DeepSpeech</td>
<td>Speech</td>
<td>WSJ’93</td>
<td>RNN</td>
<td>33.6%</td>
<td>31.6%</td>
<td>2.0%</td>
<td>5.8%</td>
</tr>
<tr>
<td>DeepSpeech-2</td>
<td>Speech</td>
<td>WSJ’93</td>
<td>RNN</td>
<td>14.5%</td>
<td>13.4%</td>
<td>1.1%</td>
<td>7.4%</td>
</tr>
</tbody>
</table>

Open Sourced DSD Model Zoo: [https://songhan.github.io/DSD](https://songhan.github.io/DSD)

The baseline results of AlexNet, VGG16, GoogleNet, SqueezeNet are from [Caffe Model Zoo](https://github.com/BVLC/caffe). ResNet18, ResNet50 are from [fb.resnet.torch](https://github.com/facebook/fb.resnet.torch).
Agenda

Algorithm

Inference

- Algorithms for Efficient Inference

Hardware

- Hardware for Efficient Inference

Training

- Algorithms for Efficient Training

- Hardware for Efficient Training
CPUs for Training

Intel Knights Landing (2016)

- 7 TFLOPS FP32
- 16GB MCDRAM– 400 GB/s
- 245W TDP
- 29 GFLOPS/W (FP32)
- 14nm process

Knights Mill: next gen Xeon Phi “optimized for deep learning”

Intel announced the addition of new vector instructions for deep learning (AVX512-4VNNIW and AVX512-4FMAPS), October 2016

Slide Source: Sze et al Survey of DNN Hardware, MICRO’16 Tutorial.
Image Source: Intel, Data Source: Next Platform
GPUs for Training

Nvidia PASCAL GP100 (2016)

- 10/20 TFLOPS FP32/FP16
- 16GB HBM – 750 GB/s
- 300W TDP
- 67 GFLOPS/W (FP16)
- 16nm process
- 160GB/s NV Link

Slide Source: Sze et al Survey of DNN Hardware, MICRO’16 Tutorial.
Data Source: NVIDIA
GPUs for Training

Nvidia Volta GV100 (2017)

- 15 FP32 TFLOPS
- 120 Tensor TFLOPS
- 16GB HBM2 @ 900GB/s
- 300W TDP
- 12nm process
- 21B Transistors
- die size: 815 mm$^2$
- 300GB/s NVLink

Data Source: NVIDIA
**What's new in Volta: Tensor Core**

A new instruction that performs 4x4x4 FMA mixed-precision operations per clock.

12X increase in throughput for the Volta V100 compared to the Pascal P100.

Tesla V100 Tensor Cores and CUDA 9 deliver up to 9x higher performance for GEMM operations.

Pascal v.s. Volta

ResNet-50 Training

Left: Tesla V100 trains the ResNet-50 deep neural network 2.4x faster than Tesla P100.

ResNet-50 Inference
TensorRT - 7ms Latency

Right: Given a target latency per image of 7ms, Tesla V100 is able to perform inference using the ResNet-50 deep neural network 3.7x faster than Tesla P100.

The GV100 SM is partitioned into four processing blocks, each with:

- 8 FP64 Cores
- 16 FP32 Cores
- 16 INT32 Cores
- two of the new mixed-precision Tensor Cores for deep learning
- a new L0 instruction cache
- one warp scheduler
- one dispatch unit
- a 64 KB Register File.

<table>
<thead>
<tr>
<th>Tesla Product</th>
<th>Tesla K40</th>
<th>Tesla M40</th>
<th>Tesla P100</th>
<th>Tesla V100</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>GK110 (Kepler)</td>
<td>GM200 (Maxwell)</td>
<td>GP100 (Pascal)</td>
<td>GV100 (Volta)</td>
</tr>
<tr>
<td><strong>GPU Boost Clock</strong></td>
<td>810/875 MHz</td>
<td>1114 MHz</td>
<td>1480 MHz</td>
<td>1455 MHz</td>
</tr>
<tr>
<td><strong>Peak FP32 TFLOP/s</strong></td>
<td>5.04</td>
<td>6.8</td>
<td>10.6</td>
<td>15</td>
</tr>
<tr>
<td><strong>Peak Tensor Core TFLOP/s</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>120</td>
</tr>
<tr>
<td><strong>Memory Interface</strong></td>
<td>384-bit GDDR5</td>
<td>384-bit GDDR5</td>
<td>4096-bit HBM2</td>
<td>4096-bit HBM2</td>
</tr>
<tr>
<td><strong>Memory Size</strong></td>
<td>Up to 12 GB</td>
<td>Up to 24 GB</td>
<td>16 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td><strong>TDP</strong></td>
<td>235 Watts</td>
<td>250 Watts</td>
<td>300 Watts</td>
<td>300 Watts</td>
</tr>
<tr>
<td><strong>Transistors</strong></td>
<td>7.1 billion</td>
<td>8 billion</td>
<td>15.3 billion</td>
<td>21.1 billion</td>
</tr>
<tr>
<td><strong>GPU Die Size</strong></td>
<td>551 mm²</td>
<td>601 mm²</td>
<td>610 mm²</td>
<td>815 mm²</td>
</tr>
<tr>
<td><strong>Manufacturing Process</strong></td>
<td>28 nm</td>
<td>28 nm</td>
<td>16 nm FinFET+</td>
<td>12 nm FFN</td>
</tr>
</tbody>
</table>

Sparsity is Natively supported by NVIDIA A100 GPU

2X peak performance, 1.5X measured BERT speedup
Google Cloud TPU

Cloud TPU delivers up to 180 teraflops to train and run machine learning models.

source: Google Blog
A “TPU pod” built with 64 second-generation TPUs delivers up to 11.5 petaflops of machine learning acceleration.

“One of our new large-scale translation models used to take a full day to train on 32 of the best commercially-available GPUs—now it trains to the same accuracy in an afternoon using just one eighth of a TPU pod.” — Google Blog
Wrap-Up

Inference

Training

Algorithm

Algorithms for Efficient Inference

Algorithms for Efficient Training

Hardware

Hardware for Efficient Inference

Hardware for Efficient Training
Outlook: the Focus for Computation

PC Era                      Mobile-First Era               AI-First Era

Computing   Mobile Computing   Brain-Inspired Cognitive Computing
Thank you!