Efficient Methods for Deep Learning

Song Han

Stanford University  
Sep 2016
Background: Deep Learning for Everything


Source: leon A. Gatys et al., “A Neural…”, arxiv:1508.06576
Hardware and Data enable Deep Learning

Dally, NIPS’2015 tutorial on High-Performance Hardware for Machine Learning
The Need for Speed

More data ➔ Bigger Models ➔ More Need for Compute

But Moore’s law is no longer providing more compute…
Goal: Improve the Efficiency of Deep Learning
For Mobile + Cloud
Embedded Applications: Self-Driving Cars

nVidia Drive PX2
24 Tps/sec @ 20W
Challenges for Efficient Deep Learning
Model Size!
Figure 1: Energy table for 45nm CMOS process. Memory access is 2 orders of magnitude more energy expensive than arithmetic operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy [pJ]</th>
<th>Relative Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit int ADD</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>32 bit float ADD</td>
<td>0.9</td>
<td>9</td>
</tr>
<tr>
<td>32 bit Register File</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>32 bit int MULT</td>
<td>3.1</td>
<td>31</td>
</tr>
<tr>
<td>32 bit float MULT</td>
<td>3.7</td>
<td>37</td>
</tr>
<tr>
<td>32 bit SRAM Cache</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>32 bit DRAM Memory</td>
<td>640</td>
<td>6400</td>
</tr>
</tbody>
</table>

Figure 1: Energy table for 45nm CMOS process. Memory access is 2 orders of magnitude more energy expensive than arithmetic operations.

To achieve this goal, we present a method to prune network connections in a manner that preserves the original accuracy. After an initial training phase, we remove all connections whose weight is lower than a threshold. This pruning converts a dense, fully-connected layer to a sparse layer. This first phase learns the topology of the networks — learning which connections are important and removing the unimportant connections. We then retrain the sparse network so the remaining connections can compensate for the connections that have been removed. The phases of pruning and retraining may be repeated iteratively to further reduce network complexity. In effect, this training process learns the network connectivity in addition to the weights — much as in the mammalian brain [8][9], where synapses are created in the first few months of a child’s development, followed by gradual pruning of little-used connections, falling to typical adult values.
Part 1: Deep Compression

Song Han
CVA group, Stanford University

Han et al. “Learning both Weights and Connections for Efficient Neural Networks”, NIPS’15
Han et al. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, ICLR 2016
Deep Compression

Problem 1: DNN Model Size too Large
Solution 1: Deep Compression
Problem 1: DNN Model Size too Large
Solution 1: Deep Compression
Deep Compression

Problem 1: DNN Model Size too Large
Solution 1: Deep Compression

Smaller Size
90% zeros in weights
4-bit weight
Deep Compression

Problem 1: DNN Model Size too Large
Solution 1: Deep Compression

Smaller Size
90% zeros in weights
4-bit weight

Accuracy
No loss of accuracy /
Improved accuracy
Problem 1: DNN Model Size too Large
Solution 1: Deep Compression

Smaller Size
90% zeros in weights
4-bit weight

Accuracy
No loss of accuracy /
Improved accuracy

On-chip
State-of-the-art DNN
fit on-chip SRAM
Deep Compression Overview

- AlexNet: 35x, 240MB => 6.9MB
- VGG16: 49x, 552MB => 11.3MB
- GoogLeNet: 10x, 28MB => 2.8MB
- SqueezeNet: 10x, 4.8MB => 0.47MB
- No loss of accuracy on ImageNet12
- Weights fits on-chip SRAM cache, taking 120x less energy than DRAM memory
Deep Compression Pipeline

- **Network Pruning:**
  10x fewer weights

- **Weight Sharing:**
  only 4-bits per remaining weight

- **Huffman Coding:**
  Entropy of the Total Remaining Weights
Deep Compression Pipeline

- **Network Pruning:**
  Less Number of Weights

- **Weight Sharing:**
  Reduce Storage for Each Remaining Weight

- **Huffman Coding:**
  Entropy of the Total Remaining Weights
1. Pruning

[1] LeCun et al. Optimal Brain Damage NIPS’90
[3] Han et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS’15
Pruning: Motivation

- Trillion of synapses are generated in the human brain during the first few months of birth.

- **1 year old**, peaked at **1000 trillion**

- Pruning begins to occur.

- **10 years old**, a child has nearly **500 trillion** synapses

- This ‘pruning’ mechanism removes redundant connections in the brain.

AlexNet & VGGNet

Han et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS 2015
Retrain to Recover Accuracy

Train Connectivity

Prune Connections

Train Weights

Han et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS 2015
## Pruning: Result

<table>
<thead>
<tr>
<th>Network</th>
<th>Top-1 Error</th>
<th>Top-5 Error</th>
<th>Parameters</th>
<th>Compression Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeNet-300-100 Ref</td>
<td>1.64%</td>
<td>-</td>
<td>267K</td>
<td>12×</td>
</tr>
<tr>
<td>LeNet-300-100 Pruned</td>
<td>1.59%</td>
<td>-</td>
<td>22K</td>
<td></td>
</tr>
<tr>
<td>LeNet-5 Ref</td>
<td>0.80%</td>
<td>-</td>
<td>431K</td>
<td>12×</td>
</tr>
<tr>
<td>LeNet-5 Pruned</td>
<td>0.77%</td>
<td>-</td>
<td>36K</td>
<td></td>
</tr>
<tr>
<td>AlexNet Ref</td>
<td>42.78%</td>
<td>19.73%</td>
<td>61M</td>
<td></td>
</tr>
<tr>
<td>AlexNet Pruned</td>
<td>42.77%</td>
<td>19.67%</td>
<td>6.7M</td>
<td>9×</td>
</tr>
<tr>
<td>VGG16 Ref</td>
<td>31.50%</td>
<td>11.32%</td>
<td>138M</td>
<td></td>
</tr>
<tr>
<td>VGG16 Pruned</td>
<td>31.34%</td>
<td>10.88%</td>
<td>10.3M</td>
<td>13×</td>
</tr>
</tbody>
</table>

Table 1: Network pruning can save 9× to 13× parameters with no drop in predictive performance.
Pruning RNN and LSTM

- Pruning away 90% parameters in NeuralTalk doesn’t hurt BLUE score with proper retraining.

Pruning NeuralTalk 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
Deep Compression Pipeline

• **Network Pruning:**
  Less Number of Weights

• **Weight Sharing:**
  Reduce Storage for Each Remaining Weight

• **Huffman Coding:**
  Entropy of the Total Remaining Weights
Weight Sharing: Overview

<table>
<thead>
<tr>
<th>weights</th>
<th>(32 bit float)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.09</td>
<td>-0.98</td>
</tr>
<tr>
<td>0.05</td>
<td>-0.14</td>
</tr>
<tr>
<td>-0.91</td>
<td>1.92</td>
</tr>
<tr>
<td>1.87</td>
<td>0</td>
</tr>
</tbody>
</table>

We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires \(2a + n + 1\) numbers, where \(a\) is the number of non-zero elements and \(n\) is the number of rows or columns. 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.

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.
Weight Sharing: Overview

We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires \(2a + n + 1\) numbers, where \(a\) is the number of non-zero elements and \(n\) is the number of rows or columns.

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 the largest 3-bit (as an example) unsigned number, we add a filler zero.

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).

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:

\[
 r = n b + k b \log_2(k) + k b (1)
\]

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
Weight Sharing: Overview

We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires $a + n + 1$ numbers, where $a$ is the number of non-zero elements and $n$ is the number of rows or columns.

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.

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:

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

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
Weight Sharing: Overview

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

Cluster weights (32 bit float)

<table>
<thead>
<tr>
<th>weights</th>
<th>cluster index (2 bit uint)</th>
<th>centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.09</td>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>-0.98</td>
<td>0</td>
<td>1.50</td>
</tr>
<tr>
<td>1.48</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>0.09</td>
<td>1</td>
<td>-1.00</td>
</tr>
<tr>
<td>0.05</td>
<td>0</td>
<td>-0.02</td>
</tr>
<tr>
<td>-0.14</td>
<td>3</td>
<td>0.12</td>
</tr>
<tr>
<td>-1.08</td>
<td>1</td>
<td>0.04</td>
</tr>
<tr>
<td>2.12</td>
<td>2</td>
<td>0.01</td>
</tr>
<tr>
<td>-0.91</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>1.92</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0.04</td>
</tr>
<tr>
<td>-1.03</td>
<td>2</td>
<td>0.02</td>
</tr>
<tr>
<td>1.87</td>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1.50</td>
</tr>
<tr>
<td>1.53</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>1.49</td>
<td>2</td>
<td>-1.00</td>
</tr>
</tbody>
</table>

Gradient

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 the 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:

\[
r = nb \log_2(k) + kb \]

For example, Figure 3 shows the weights of a single layer neural network with four input units and four output units. There are 4 × 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
Weight Sharing: Overview

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).

We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires \(2a + n + 1\) numbers, where \(a\) is the number of non-zero elements and \(n\) is the number of rows or columns.

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 the largest 3-bit (as an example) unsigned number, we add a filler zero.

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:

\[
r = \frac{nb - n\log_2(k) + kb}{b}
\]

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

Han et al. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, ICLR 2016
We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires $a + n + 1$ numbers, where $a$ is the number of non-zero elements and $n$ is the number of rows or columns.

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 the largest 3-bit (as an example) unsigned number, we add a filler zero.

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.
Weight Sharing: Overview

We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires $\frac{a}{2} + n + 1$ numbers, where $a$ is the number of non-zero elements and $n$ is the number of rows or columns.

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.

3T 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 $4 \times 4$ 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:

$$r = \frac{nb}{n \log_2(k)} + k$$

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
Bits Per Weight

Han et al. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, ICLR 2016
# Pruning + Trained Quantization

### Table 6: Accuracy of AlexNet with different aggressiveness of weight sharing and quantization. 8/5 bit quantization has no loss of accuracy; 8/4 bit quantization, which is more hardware friendly, has negligible loss of accuracy of 0.01%; To be really aggressive, 4/2 bit quantization resulted in 1.99% and 2.60% loss of accuracy.

<table>
<thead>
<tr>
<th>#CONV bits / #FC bits</th>
<th>Top-1 Error</th>
<th>Top-5 Error</th>
<th>Top-1 Error Increase</th>
<th>Top-5 Error Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bits / 32bits</td>
<td>42.78%</td>
<td>19.73%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8 bits / 5 bits</td>
<td>42.78%</td>
<td>19.70%</td>
<td>0.00%</td>
<td>-0.03%</td>
</tr>
<tr>
<td>8 bits / 4 bits</td>
<td>42.79%</td>
<td>19.73%</td>
<td>0.01%</td>
<td>0.00%</td>
</tr>
<tr>
<td>4 bits / 2 bits</td>
<td>44.77%</td>
<td>22.33%</td>
<td>1.99%</td>
<td>2.60%</td>
</tr>
</tbody>
</table>
Pruning + Trained Quantization

Han et al. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, ICLR 2016
Deep Compression Pipeline

- **Network Pruning:**
  Less Number of Weights

- **Weight Sharing:**
  Reduce Storage for Each Remaining Weight

- **Huffman Coding:**
  Entropy of the Total Remaining Weights

Han et al. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, ICLR 2016
**Huffman Coding**

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

Han et al. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, ICLR 2016
# Deep Compression Results

<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>SqueezeNet</td>
<td>4.8MB</td>
<td>0.47MB</td>
<td>10x</td>
<td>80.32%</td>
<td>80.35%</td>
</tr>
</tbody>
</table>

- No loss of accuracy after compression.
- Fits in SRAM cache (120x less energy than DRAM).
660KB model, AlexNet-accuracy

https://github.com/songhan/SqueezeNet_compressed

Iandola, Han, et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size” arXiv 2016
Conclusion

• **Complex DNNs can be put in mobile applications (<10MB total)**
  – 500MB with-FC network (125M weights) becomes 10MB
  – 10MB all-CONV network (2.5M weights) becomes 1MB

• **Memory bandwidth reduced by 10-50x**
  – Particularly for FC layers in real-time applications with no reuse

• **Faster Prediction**
  – Works well for sparsity level 10%-20%. Ads, Speech…
What happens once DNN size is so small that it fits in SRAM Cache?
Speedup/Energy Efficiency on CPU/GPU

CPU: Core i-7 5930k; GPU: GTX TitanX; mobile GPU: Tegra K1; All scenarios: batchsize = 1
Facebook is using this to speedup ads click prediction.

CPU: Core i-7 5930k; GPU: GTX TitanX; mobile GPU: Tegra K1; All scenarios: batchsize = 1
Part 2: EIE

Efficient Inference Engine on Compressed Deep Neural Network

Song Han
CVA group, Stanford University

Problem 2: Faster, Energy Efficient

Problem 2: Faster, Energy Efficient
Solution 2: EIE accelerator

EIE: First Accelerator for Compressed Sparse Neural Network

Problem 2: Faster, Energy Efficient
Solution 2: EIE accelerator

Problem 2: Faster, Energy Efficient
Solution 2: EIE accelerator

Sparse Matrix

90% static sparsity in the weights,
10x less computation,
5x less memory footprint
EIE: First Accelerator for Compressed Sparse Neural Network

Problem 2: Faster, Energy Efficient
Solution 2: EIE accelerator

Sparse Matrix
- 90% static sparsity in the weights,
- 10x less computation,
- 5x less memory footprint

Sparse Vector
- 70% dynamic sparsity in the activation
- 3x less computation

EIE: First Accelerator for Compressed Sparse Neural Network

Problem 2: Faster, Energy Efficient

Solution 2: EIE accelerator

Sparse Matrix
90% static sparsity in the weights,
10x less computation, 5x less memory footprint

Sparse Vector
70% dynamic sparsity in the activation
3x less computation

Weight Sharing
4bits weights
8x less memory footprint

EIE: First Accelerator for Compressed Sparse Neural Network

Problem 2: Faster, Energy Efficient
Solution 2: EIE accelerator

Sparse Matrix
- 90% static sparsity in the weights,
- 10x less computation,
- 5x less memory footprint

Sparse Vector
- 70% dynamic sparsity in the activation
- 3x less computation

Weight Sharing
- 4bits weights
- 8x less memory footprint

Fully fits in SRAM
- 120x less energy than DRAM

EIE: First Accelerator for Compressed Sparse Neural Network

Problem 2: Faster, Energy Efficient
Solution 2: EIE accelerator

Sparse Matrix
90% static sparsity in the weights,
10x less computation,
5x less memory footprint

Sparse Vector
70% dynamic sparsity in the activation
3x less computation

Weight Sharing
4bits weights
8x less memory footprint

Fully fits in SRAM
120x less energy than DRAM

Savings are multiplicative: 5x3x8x120=14,400 theoretical energy improvement.

Benchmark

- CPU: Intel Core-i7 5930k
- GPU: NVIDIA TitanX
- Mobile GPU: NVIDIA Jetson TK1

<table>
<thead>
<tr>
<th>Layer</th>
<th>Size</th>
<th>Weight Density</th>
<th>Activation Density</th>
<th>FLOP %</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet-6</td>
<td>4096 × 9216</td>
<td>9%</td>
<td>35.1%</td>
<td>3%</td>
<td>AlexNet for image classification</td>
</tr>
<tr>
<td>AlexNet-7</td>
<td>4096 × 4096</td>
<td>9%</td>
<td>35.3%</td>
<td>3%</td>
<td></td>
</tr>
<tr>
<td>AlexNet-8</td>
<td>1000 × 4096</td>
<td>25%</td>
<td>37.5%</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>VGG-6</td>
<td>4096 × 25088</td>
<td>4%</td>
<td>18.3%</td>
<td>1%</td>
<td>VGG-16 for image classification</td>
</tr>
<tr>
<td>VGG-7</td>
<td>4096 × 4096</td>
<td>4%</td>
<td>37.5%</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>VGG-8</td>
<td>1000 × 4096</td>
<td>23%</td>
<td>41.1%</td>
<td>9%</td>
<td></td>
</tr>
<tr>
<td>NeuralTalk-We</td>
<td>600 × 4096</td>
<td>10%</td>
<td>100%</td>
<td>10%</td>
<td>RNN and LSTM for image caption</td>
</tr>
<tr>
<td>NeuralTalk-Wd</td>
<td>8791 × 600</td>
<td>11%</td>
<td>100%</td>
<td>11%</td>
<td></td>
</tr>
<tr>
<td>NeuralTalk-LSTM</td>
<td>2400 × 1201</td>
<td>10%</td>
<td>100%</td>
<td>11%</td>
<td></td>
</tr>
</tbody>
</table>

Result of EIE

1. Post layout result
2. Throughput measured on AlexNet FC-7
Speedup on EIE

Compared to CPU and GPU:
189x and 13x faster

Baseline:
- Intel Core i7 5930K: MKL CBLAS GEMV, MKL SPBLAS CSRMV
- NVIDIA GeForce GTX Titan X: cuBLAS GEMV, cuSPARSE CSRMV
- NVIDIA Tegra K1: cuBLAS GEMV, cuSPARSE CSRMV
### Energy Efficiency on EIE

Compared to CPU and GPU:

24,000x and 3,400x more energy efficient

Baseline:

- **Intel Core i7 5930K**: reported by pcm-power utility
- **NVIDIA GeForce GTX Titan X**: reported by nvidia-smi utility
- **NVIDIA Tegra K1**: measured with power-meter, 60% AP+DRAM power
Where are the savings from?

- Four factors for energy saving:
  - 10× *static* weight sparsity; less work to do; less bricks to carry.
  - 3× *dynamic* activation sparsity; carry only good bricks; ignore broken bricks.
  - Weight sharing with only 4-bits per weight; lighter bricks to carry.
  - DRAM => SRAM, no need to go off-chip; carry bricks from NY to Stanford => SF to Stanford.
Conclusion

- EIE: first accelerator for compressed, sparse neural network.
- Compression => Acceleration, no loss accuracy.
- Distributed storage/computation to parallelize/load balance across PEs.
- 13x faster and 3,400x more energy efficient than GPU. 2.9x faster and 19x more energy efficient than past ASICs.