13th Feb 2024

Deep Compression : Compressing deep neural networks with pruning, trained quantization and Huffman coding

Han, Song, Huizi Mao and William J. Dally. **International Conference on Learning Representation (ICLR), 2016.**

Best paper award ICLR 2016
9600+ Citations as of 2024

Presenters

Rahul Vigneswaran K
CS23MTECH02002

Nikhil Kumar Patel
CS23MTECH11013

Course Instructor

Rajesh Kedia

*Presented as part of the coursework for Hardware Architecture for Deep Learning (CS6490)

Table Of Contents

01 **Introduction**
- Need for compression
- Overview

02 **Pruning**
- Compressed Sparse Row (CSR)
- Relative index

03 **Quantization**
- Weight Sharing (Trained Quantization)
- Compression Rate
- Initialization of shared weights
- K-Means

04 **Huffman Coding**
- Huffman Tree

05 **Experimental Setup**

06 **Results**
- Compression
- Speed-up & Energy

07 **Future Works**
- Future works from paper
- Future works that we thought of

08 **Review**

09 **Conclusion**

Need for Compression

- Storage
  - AlexNet Caffemodel ~ 200MB
  - VGG-16 Caffemodel ~ 500MB
  - > 100 MB triggers Wi-Fi only
- Energy consumption

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

- 20fps would require (20Hz)(1G)(640pJ) = 12.8 W just for DRAM access
  - (20 Frames/s)x(10^9)x(640pJ/Frame)x(1 Watt/10^12 pJ) = 12.8 W

Han et al. "Learning both Weights and Connections for Efficient Neural Networks", NeurIPS 2015

Overview

original network
original size
Pruning
same accuracy
9x-13x reduction
Quantization
same accuracy
27x-31x reduction
Huffman Coding
same accuracy
35x-49x reduction

Pruning

Age	Number of Connections	Stage
at birth	50 Trillion	newly formed
1 year old	1000 Trillion	peak
10 year old	500 Trillion	pruned and stabilized

Table 1: The synapses pruning mechanism in human brain development

Christopher A Walsh. Peter huttenlocher (1931-2013). Nature, 2013.

Pruning

- Compression
  - AlexNet : 9x
  - VGG-16 : 13x
- What do you exactly mean by pruning?

- But this doesn't save any space at all. Why?

Figure 7: Weight distribution before and after parameter pruning. The right figure has $10\times$ smaller scale.

Han et al. "Learning both Weights and Connections for Efficient Neural Networks", NeurIPS 2015,

Pruning

- Compression
  - AlexNet : 9x
  - VGG-16 : 13x
- What do you exactly mean by pruning?
  - Make gradients zero at each step.
- But this doesn't save any space at all. Why?

Figure 7: Weight distribution before and after parameter pruning. The right figure has $10\times$ smaller scale.

Han et al. "Learning both Weights and Connections for Efficient Neural Networks", NeurIPS 2015,

Pruning

- Compression
  - AlexNet : 9x
  - VGG-16 : 13x
- What do you exactly mean by pruning?
  - Make gradients zero at each step.
- But this doesn't save any space at all. Why?
  - Exploiting sparsity

Figure 7: Weight distribution before and after parameter pruning. The right figure has $10\times$ smaller scale.

Han et al. "Learning both Weights and Connections for Efficient Neural Networks", NeurIPS 2015,

Compressed Sparse Row (CSR)

1	0	0	0	0	0	5	0	0	3	0	0

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

0	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	0	5	0	0	3	0	0

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

0	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	0	5	0	0	3	0	0

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

0	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	0	5	0	0	3	0	0

index	0	6	9
value	1	5	3

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

0	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	0	5	0	0	3	0	0

index	0	6	9
value	1	5	3

```
int[] reconstruct_from_CSR(int[] index, int[] value, int size) {
  int original[size];
  int index_ptr = 0;

  for (int i = 0; i < size; i++) {
    if (index[index_ptr] == i) {
      original[i] = value[index_ptr];
      index_ptr++;
    } else {
      original[i] = 0;
    }
  }

  return original;
}
```

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

Matrix:

1	7		
5		3	9
	2	8	
			6

Row:	0	0	1	1	1	2	2	3
Column:	0	1	0	2	3	1	2	3
Value:	1	7	5	3	9	2	8	6

RowPtrs:	0	2	5	7	8			
Column:	0	1	0	2	3	1	2	3
Value:	1	7	5	3	9	2	8	6

Izzat El Hajj. Lecture 16 - Sparse Matrix Computation (COO and CSR), Youtube 2023

Compressed Sparse Row (CSR)

2	0	0	6
0	0	0	0
0	4	0	0

column	0	3	1
value	2	6	4

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

2	0	0	6
0	0	0	0
0	4	0	0

row	0	2	2	3
column	0	3	1	
value	2	6	4	

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Compressed Sparse Row (CSR)

2	0	0	6
0	0	0	0
0	4	0	0

row	0	2	2	3
column	0	3	1	
value	2	6	4	

```
int[][] reconstruct_from_CSR(int[] row, int[] col, int[] value, int
row_size, int col_size) {
  int original[row_size][col_size];

  for (int i = 0; i < row_size;) {
     int cur = row[i];
     int end = row[i+1];

     for (int j = 0; j < col_size; j++) {
        if (cur < end && col[cur] == j) {
          original[i][j] = value[cur];
          cur++;
        } else {
          original[i][j] = 0;
        }
     }
  }

  return original;
}
```

- COO: 3*non-zero elements
- CSR: 2*(non-zero elements) + (rows) + 1

Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021

Relative index

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

- Conv: 8 bits
- FC: 5 bits

Overview

Pruning: less number of weights
original network
original size
Train Connectivity
Prune Connections
Train Weights
same accuracy
9x-13x reduction
Quantization
same accuracy
27x-31x reduction

Overview

Pruning: less number of weights
Quantization: less bits per weight
original network
original size
Train Connectivity
Prune Connections
Train Weights
same accuracy
9x-13x reduction
Cluster the Weights
Generate Code Book
Quantize the Weights with Code Book
Retrain Code Book
same accuracy
27x-31x reduction

Weight Sharing (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

Weight Sharing (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

Weight Sharing (Trained Quantization)

Weight Sharing (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

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

Weight Sharing (Trained Quantization)

Weight Sharing (Trained Quantization)

Weight Sharing (Trained Quantization)

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

Compression Rate

$$r = \frac{nb}{nlog_2(k) + kb}$$

k : number of clusters .

$\log_2$(k) : required bits to encode the index.

n : number of connections.

b : bits representing each connections.

Compression Rate

Compression rate :

$$r = \frac{nb}{nlog_2(k) + kb}$$

16 ∗ 32/(16 * 2 + 4 * 32) = 3.2

Initialization of shared weights

- Centroid initialization impacts the quality of clustering and thus affects the network's prediction accuracy.
- Three initialization methods:
  - Forgy(random)
  - Density-based
  - linear initialization

Initialization of shared weights : Forgy (random)

- Randomly choose k observations.
- Bimodal distribution.
- Concentrate around those peaks.

Initialization of shared weights : Density Based

- Linearly space the CDF of the weights in the y-axis.
- Find the horizontal intersection with the CDF.
- Finds the vertical intersection on the x-axis, which becomes a centroid.
- Makes the centroids denser around the two peaks, but more scattered than the Forgy method.

Initialization of shared weights : Linear

- Linearly space the centroids between the [min, max] of the original weights.
- Invariant to the distribution of the weights and is the most scattered.

Initialization of shared weights after Fine Tuning

Loss Function for Weight Sharing

- k-means clustering.
- minimize the within-cluster sum of squares (WCSS):

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2$$

- $W = \{w_1, w_2, ..., w_n\}$, **n** original weights.
- $C = \{c_1, c_2, ..., c_k\}$, k clusters.
- $n >> k$.

Feed-Forward and Back-Propagation

- L : loss function
- W_{ij} : weight in the ith column and j^{th} row.
- I_{ij} : centroid index of element $W_{i,j}$.
- C_k : k^{th} centroid of the layer.
- 1(.) : Indicator function.
- Gradient of the centroids is calculated as:

$$\frac{\partial \mathcal{L}}{\partial C_k} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial W_{ij}} \frac{\partial W_{ij}}{\partial C_k} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial W_{ij}} \mathbb{1}(I_{ij} = k)$$

Pruning + Quantization

Steps to build Huffman Tree

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

character	Frequency
Internal Node	25
Internal Node	30
f	45

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

character	Frequency
f	45
Internal Node	55

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

character	Frequency
Internal Node	100

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Steps to print codes from Huffman Tree

Traverse the tree formed starting from the root. Assign 0 to the left child and 1 to the right child.

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Huffman Encoding

- In-frequent weights: Use more bits to represent.
- Frequent weights: Use less bits to represent.
- Experiments show that Huffman coding of non-uniformly distributed values of the last fully connected layer in AlexNet saves 20% – 30% of network storage.

Overview

Pruning: less number of weights
Quantization: less bits per weight
original network
original size
Train Connectivity
Prune Connections
Train Weights
same accuracy
9x-13x reduction
Cluster the Weights
Generate Code Book
Quantize the Weights with Code Book
Retrain Code Book
same accuracy
27x-31x reduction
Huffman Coding
same accuracy
35x-49x reduction

Overview

Pruning: less number of weights
Quantization: less bits per weight
Huffman Encoding
original network
original size
Train Connectivity
Prune Connections
Train Weights
same accuracy
9x-13x reduction
Cluster the Weights
Generate Code Book
Quantize the Weights with Code Book
Retrain Code Book
same accuracy
27x-31x reduction
Encode Weights
Encode Index
same accuracy
35x-49x reduction

Experimental setup

- MNIST
  - Train: 60k
  - Test: 10k
  - 10 Classes
  - Arch
    - LeNet-300-100
    - LeNet-5
- ImageNet
  - Train: 1.2M
  - Validation: 50k
  - 1000 classes
  - Arch
    - AlexNet
    - VGG-16

Layer	#Weights
ip1	235K
ip2	30K
ip3	1K
Total	266K

LeNet-300-100

Layer	#Weights
conv1	0.5K
conv2	25K
ip1	400K
ip2	5K
Total	431K

LeNet-5

Layer	#Weights
conv1	35K
conv2	307K
conv3	885K
conv4	663K
conv5	442K
fc6	38M
fc7	17M
fc8	4M
Total	61M

AlexNet

Layer	#Weights
conv1_1	2K
conv1_2	37K
conv2_1	74K
conv2_2	148K
conv3_1	295K
conv3_2	590K
conv3_3	590K
conv4_1	1M
conv4_2	2M
conv4_3	2M
conv5_1	2M
conv5_2	2M
conv5_3	2M
fc6	103M
fc7	17M
fc8	4M
Total	138M

VGG-16

Results

Table 1: The compression pipeline can save 35× to 49× parameter storage with no loss of accuracy.

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	**40×**
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39×**
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35×**
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	**49×**

Table 2: Compression statistics for LeNet-300-100. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
ip1	235K	8%	6	4.4	5	3.7	3.1%	2.32%
ip2	30K	9%	6	4.4	5	4.3	3.8%	3.04%
ip3	1K	26%	6	4.3	5	3.2	15.7%	12.70%
Total	266K	8%(12×)	6	5.1	5	3.7	3.1% (**32**×)	2.49% (**40**×)

Table 3: Compression statistics for LeNet-5. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1	0.5K	66%	8	7.2	5	1.5	78.5%	67.45%
conv2	25K	12%	8	7.2	5	3.9	6.0%	5.28%
ip1	400K	8%	5	4.5	5	4.5	2.7%	2.45%
ip2	5K	19%	5	5.2	5	3.7	6.9%	6.13%
Total	431K	8%(12×)	5.3	4.1	5	4.4	3.05% (**33**×)	2.55% (**39**×)

Table 4: Compression statistics for AlexNet. P: pruning, Q: quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1	35K	84%	8	6.3	4	1.2	32.6%	20.53%
conv2	307K	38%	8	5.5	4	2.3	14.5%	9.43%
conv3	885K	35%	8	5.1	4	2.6	13.1%	8.44%
conv4	663K	37%	8	5.2	4	2.5	14.1%	9.11%
conv5	442K	37%	8	5.6	4	2.5	14.0%	9.43%
fc6	38M	9%	5	3.9	4	3.2	3.0%	2.39%
fc7	17M	9%	5	3.6	4	3.7	3.0%	2.46%
fc8	4M	25%	5	4	4	3.2	7.3%	5.85%
Total	61M	11%(9×)	5.4	4	4	3.2	3.7% (**27**×)	2.88% (**35**×)

Table 5: Compression statistics for VGG-16. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weigh bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1_1	2K	58%	8	6.8	5	1.7	40.0%	29.97%
conv1_2	37K	22%	8	6.5	5	2.6	9.8%	6.99%
conv2_1	74K	34%	8	5.6	5	2.4	14.3%	8.91%
conv2_2	148K	36%	8	5.9	5	2.3	14.7%	9.31%
conv3_1	295K	53%	8	4.8	5	1.8	21.7%	11.15%
conv3_2	590K	24%	8	4.6	5	2.9	9.7%	5.67%
conv3_3	590K	42%	8	4.6	5	2.2	17.0%	8.96%
conv4_1	1M	32%	8	4.6	5	2.6	13.1%	7.29%
conv4_2	2M	27%	8	4.2	5	2.9	10.9%	5.93%
conv4_3	2M	34%	8	4.4	5	2.5	14.0%	7.47%
conv5_1	2M	35%	8	4.7	5	2.5	14.3%	8.00%
conv5_2	2M	29%	8	4.6	5	2.7	11.7%	6.52%
conv5_3	2M	36%	8	4.6	5	2.3	14.8%	7.79%
fc6	103M	4%	5	3.6	5	3.5	1.6%	1.10%
fc7	17M	4%	5	4	5	4.3	1.5%	1.25%
fc8	4M	23%	5	4	5	3.4	7.1%	5.24%
Total	138M	7.5%(13×)	6.4	4.1	5	3.1	3.2% (**31**×)	2.05% (**49**×)

Figure 6: Accuracy v.s. compression rate under different compression methods. Pruning and quantization works best when combined.

*Most probably the above plot is shown for AlexNet on ImageNet.

Figure 7: Pruning doesn't hurt quantization. Dashed: quantization on unpruned network. Solid: quantization on pruned network; Accuracy begins to drop at the same number of quantization bits whether or not the network has been pruned. Although pruning made the number of parameters less, quantization still works well, or even better(3 bits case on the left figure) as in the unpruned network.

*Most probably the above plot is shown for AlexNet on ImageNet. Probably for Linear init.

Figure 8: Accuracy of different initialization methods. Left: top-1 accuracy. Right: top-5 accuracy. Linear initialization gives best result.

*Most probably the above plot is shown for AlexNet on ImageNet. Probably Conv only.

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.

#CONV bits / #FC bits	Top-1 Error	Top-5 Error	Top-1 Error Increase	Top-5 Error Increase
32bits / 32bits	42.78%	19.73%	-	-
8 bits / 5 bits	42.78%	19.70%	0.00%	-0.03%
8 bits / 4 bits	42.79%	19.73%	0.01%	0.00%
4 bits / 2 bits	44.77%	22.33%	1.99%	2.60%

Speed-up and Energy

- Hardware
  - CPU: Intel CoRE i7 5930K (Used as baseline)
  - GPU: Nvidia GeForce GTX Titan X (Desktop grade)
  - TK1: NVIDIA Tegra K1 (Mobile grade)

- Speed benchmark
  - CPU
    - Dense: MKL CBLAS GEMV
    - Pruned: SPBLAS CSRMV
  - GPU:
    - Dense: cuBLAS GEMV
    - Pruned: cuSPARSE CSRMV kernel

- Power benchmark (SOC+DRAM combined)
  - CPU
    - pcm-power utility by Intel
  - GPU
    - Titan: nvidia-smi by NVIDIA
    - Tegra: Directly using a power meter

Speed-up and Energy

- Hardware
  - CPU: Intel CoRE i7 5930K (Used as baseline)
  - GPU: Nvidia GeForce GTX Titan X (Desktop grade)
  - TK1: NVIDIA Tegra K1 (Mobile grade)

- Speed benchmark
  - CPU
    - Dense: MKL CBLAS GEMV
    - Pruned: SPBLAS CSRMV
  - GPU:
    - Dense: cuBLAS GEMV
    - Pruned: cuSPARSE CSRMV kernel

- Power benchmark (SOC+DRAM combined)
  - CPU
    - pcm-power utility by Intel
  - GPU
    - Titan: nvidia-smi by NVIDIA
    - Tegra: Directly using a power meter

Speed-up

Figure 9: Compared with the original network, pruned network layer achieved 3× speedup on CPU, 3.5× on GPU and 4.2× on mobile GPU on average. Batch size = 1 targeting real time processing. Performance number normalized to CPU.

Table 8: Average time on different layers. To avoid variance, we measured the time spent on each layer for 4096 input samples, and averaged the time regarding each input sample. For GPU, the time consumed by `cudaMalloc` and `cudaMemcpy` is not counted. For batch size = 1, `gemv` is used; For batch size = 64, `gemm` is used. For sparse case, `csrmv` and `csrmm` is used, respectively.

Time (us)		AlexNet FC6	AlexNet FC7	AlexNet FC8	VGG16 FC6	VGG16 FC7	VGG16 FC8
Titan X	dense (batch=1)	541.5	243.0	80.5	1467.8	243.0	80.5
	sparse (batch=1)	134.8	65.8	54.6	167.0	39.8	48.0
	dense (batch=64)	19.8	8.9	5.9	53.6	8.9	5.9
	sparse (batch=64)	94.6	51.5	23.2	121.5	24.4	22.0
Core i7-5930k	dense (batch=1)	7516.2	6187.1	1134.9	35022.8	5372.8	774.2
	sparse (batch=1)	3066.5	1282.1	890.5	3774.3	545.1	777.3
	dense (batch=64)	318.4	188.9	45.8	1056.0	188.3	45.7
	sparse (batch=64)	1417.6	682.1	407.7	1780.3	274.9	363.1
Tegra K1	dense (batch=1)	12437.2	5765.0	2252.1	35427.0	5544.3	2243.1
	sparse (batch=1)	2879.3	1256.5	837.0	4377.2	626.3	745.1
	dense (batch=64)	1663.6	2056.8	298.0	2001.4	2050.7	483.9
	sparse (batch=64)	4003.9	1372.8	576.7	8024.8	660.2	544.1

Energy

Figure 10: Compared with the original network, pruned network layer takes 7× less energy on CPU, 3.3× less on GPU and 4.2× less on mobile GPU on average. Batch size = 1 targeting real time processing. Energy number normalized to CPU.

Table 9: Power consumption of different layers. We measured the Titan X GPU power with `nvidia-smi`, Core i7-5930k CPU power with `pcm-power` and Tegra K1 mobile GPU power with an external power meter (scaled to AP+DRAM, see paper discussion). During power measurement, we repeated each computation multiple times in order to get stable numbers. On CPU, dense matrix multiplications consume $2x$ energy than sparse ones because it is accelerated with multi-threading.

Power (Watts)		AlexNet FC6	AlexNet FC7	AlexNet FC8	VGG16 FC6	VGG16 FC7	VGG16 FC8
TitanX	dense (batch=1)	157	159	159	166	163	159
	sparse (batch=1)	181	183	162	189	166	162
	dense (batch=64)	168	173	166	173	173	167
	sparse (batch=64)	156	158	163	160	158	161
Core i7-5930k	dense (batch=1)	83.5	72.8	77.6	70.6	74.6	77.0
	sparse (batch=1)	42.3	37.4	36.5	38.0	37.4	36.0
	dense (batch=64)	85.4	84.7	101.6	83.1	97.1	87.5
	sparse (batch=64)	37.2	37.1	38	39.5	36.6	38.2
Tegra K1	dense (batch=1)	5.1	5.1	5.4	5.3	5.3	5.4
	sparse (batch=1)	5.9	6.1	5.8	5.6	6.3	5.8
	dense (batch=64)	5.6	5.6	6.3	5.4	5.6	6.3
	sparse (batch=64)	5.0	4.6	5.1	4.8	4.7	5.0

Figure 11: Storage ratio of weight, index and codebook.

Future work from paper

- Software solution
  - Current off-the-shelf cuSPARSE or MKL SPBLAS libraries lack support for the following
    - Quantized networks with weight sharing
    - Relative index in CSR
- Hardware solution
  - Build custom ASIC architecture specialized to traverse the sparse and quantized network structure, which also supports customized quantization bit width.
    - Already done in “Han, Song et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network.” ISCA, 2016.”

Future work that we thought of

- FC layers are more robust to using less bits and are more compressible compared to Convs. But FC is not as good as Conv layers. But MLP mixer models are Conv and attention layers made of FC connections which has accuracy on-par with Conv networks. Compressing MLP-Mixer would leverage best of both worlds.

Figure 1: MLP-Mixer consists of per-patch linear embeddings, Mixer layers, and a classifier head. Mixer layers contain one token-mixing MLP and one channel-mixing MLP, each consisting of two fully-connected layers and a GELU nonlinearity. Other components include: skip-connections, dropout, and layer norm on the channels.

Tolstikhin, Ilya O. et al. "MLP-Mixer: An all-MLP Architecture for Vision." NeurIPS 2021.

Related Works

- Fixed point implementation
  - With 8-bit integer (vs 32-bit floating point) activations.
  - Fixed-point network with ternary weights and 3-bit activations.
- Quantizing the NN's using L2 error minimization
  - Achieved better accuracy on MNIST and CIFAR-10 datasets.
- Exploiting the linear structure of the NN's
  - Find an appropriate low-rank approximation of the parameters and keep the accuracy within 1% of the original model.
- Using a hash function
  - Randomly group connection weights, so that all connections within the same hash bucket share a single parameter value.
- Using vector quantization, which resulted in 1% accuracy loss.
- Replacing the fully connected layer with global average pooling.
  - The Network in Network architecture and GoogLenet achieves state-of-the-art results on several benchmarks.
  - Reusing features learned on the ImageNet dataset and applying them to new tasks by only fine-tuning the fully connected layers, is more difficult with this approach.
    - Add a linear layer on the top of their networks to enable transfer learning.
- Network pruning has been used both to reduce network complexity and to reduce overfitting.
- Attempts to reduce the number of activations for both compression and acceleration

Related Works

Table 7: Comparison with other compression methods on AlexNet. (Collins & Kohli, 2014) reduced the parameters by 4× and with inferior accuracy. Deep Fried Convnets(Yang et al., 2014) worked on fully connected layers and reduced the parameters by less than 4×. SVD save parameters but suffers from large accuracy loss as much as 2%. Network pruning (Han et al., 2015) reduced the parameters by 9×, not including index overhead. On other networks similar to AlexNet, (Denton et al., 2014) exploited linear structure of convnets and compressed the network by 2.4× to 13.4× layer wise, with 0.9% accuracy loss on compressing a single layer. (Gong et al., 2014) experimented with vector quantization and compressed the network by 16× to 24×, incurring 1% accuracy loss.

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
Baseline Caffemodel (BVLC)	42.78%	19.73%	240MB	1×
Fastfood-32-AD (Yang et al., 2014)	41.93%	-	131MB	2×
Fastfood-16-AD (Yang et al., 2014)	42.90%	-	64MB	3.7×
Collins & Kohli (Collins & Kohli, 2014)	44.40%	-	61MB	4×
SVD (Denton et al., 2014)	44.02%	20.56%	47.6MB	5×
Pruning (Han et al., 2015)	42.77%	19.67%	27MB	9×
Pruning+Quantization	42.78%	19.70%	8.9MB	27×
Pruning+Quantization+Huffman	**42.78%**	**19.70%**	**6.9MB**	**35×**

Review of the paper

Positives

- Introduces groundbreaking concepts ahead of its time and is identified as a pivotal contribution to the compression of Neural Networks.

Negatives

- Potential for more detailed descriptions of specific topics within the paper.
- Some of the plots and tables are not self explanatory.
- They don’t say what setup the plots belongs to.
- The details of how and where the huffman coding is used is not provided properly.

Conclusion

- Deep Compression: pruning + quantizing using weight sharing + Huffman encoding.
- Experiments on AlexNet which reduced the weight storage by 35× without loss of accuracy.
- VGG-16 and LeNet networks compressed by 49× and 39× without loss of accuracy.
- Smaller storage requirement.
- Networks fit into on-chip SRAM cache (5pJ/access) rather than requiring off-chip DRAM memory (640pJ/access).
- Energy efficient to run on mobile.

Citations

- Han et al. "Learning both Weights and Connections for Efficient Neural Networks", NeurIPS 2015
- Christopher A Walsh. Peter huttenlocher (1931-2013). Nature, 2013.
- Kerdcharoen. Compressed Sparse Row (CSR) — Introduction and Explanation, Medium Blog 2021
- Izzat El Hajj. Lecture 16 - Sparse Matrix Computation (COO and CSR), Youtube 2023
- Han, Song et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network.” ISCA, 2016.
- Huffman Tree: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Rahul Vigneswaran K

Nikhil Kumar Patel

Reading Materials

- CMOS: https://www.elprocus.com/cmos-working-principle-and-applications/
- EIE Accelerator: https://dl.acm.org/doi/10.1109/ISCA.2016.30
- Nice material on memory hierarchy: https://computationstructures.org/lectures/caches/caches.html
- CSR Matrix Multiplication & Trade-offs: https://www.youtube.com/watch?v=SRgFScHYTy4
- SVD Compression: http://timbaumann.info/svd-image-compression-demo/#:~:text=Using%20SVD%20for%20image%20compression,representing%20one%20of%20the%20channels.
- Codebase:
  - Official: Caffe - https://github.com/songhan/Deep-Compression-AlexNet
  - Unofficial:
    - PyTorch - https://github.com/mightydeveloper/Deep-Compression-PyTorch
    - PyTorch Lightning - https://github.com/ciodar/deep-compression
- Energy Vs Power:
  - Technical: https://www.youtube.com/watch?v=4ls7qX0sAG8
  - Non-Technica: https://www.youtube.com/watch?v=UjkOERi8i-s