DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING

Song Han
Stanford University, Stanford, CA 94305, USA
songhan@stanford.edu

Huizi Mao
Tsinghua University, Beijing, 100084, China
mhz12@mails.tsinghua.edu.cn

William J. Dally
Stanford University, Stanford, CA 94305, USA
NVIDIA, Santa Clara, CA 95050, USA
dally@stanford.edu

ABSTRACT

Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources. To address this limitation, we introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35× to 49× without affecting their accuracy. Our method first prunes the network by learning only the important connections. Next, we quantize the weights to enforce weight sharing, finally, we apply Huffman coding. After the first two steps we retrain the network to fine tune the remaining connections and the quantized centroids. Pruning, reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. On the ImageNet dataset, our method reduced the storage required by AlexNet by **35**×, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG-16 by **49**× from 552MB to 11.3MB, again with no loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained. Benchmarked on CPU, GPU and mobile GPU, compressed network has 3× to 4× layerwise speedup and 3× to 7× better energy efficiency.

三个管道阶段

用到了梯度量化的方法

1 INTRODUCTION

Deep neural networks have evolved to the state-of-the-art technique for computer vision tasks (Krizhevsky et al., 2012)(Simonyan & Zisserman, 2014). Though these neural networks are very powerful, the large number of weights consumes considerable storage and memory bandwidth. For example, the AlexNet Caffemodel is over 200MB, and the VGG-16 Caffemodel is over 500MB (BVLC). This makes it difficult to deploy deep neural networks on mobile system.

First, for many mobile-first companies such as Baidu and Facebook, various apps are updated via different app stores, and they are very sensitive to the size of the binary files. For example, App Store has the restriction "apps above 100 MB will not download until you connect to Wi-Fi". As a result, a feature that increases the binary size by 100MB will receive much more scrutiny than one that increases it by 10MB. Although having deep neural networks running on mobile has many great

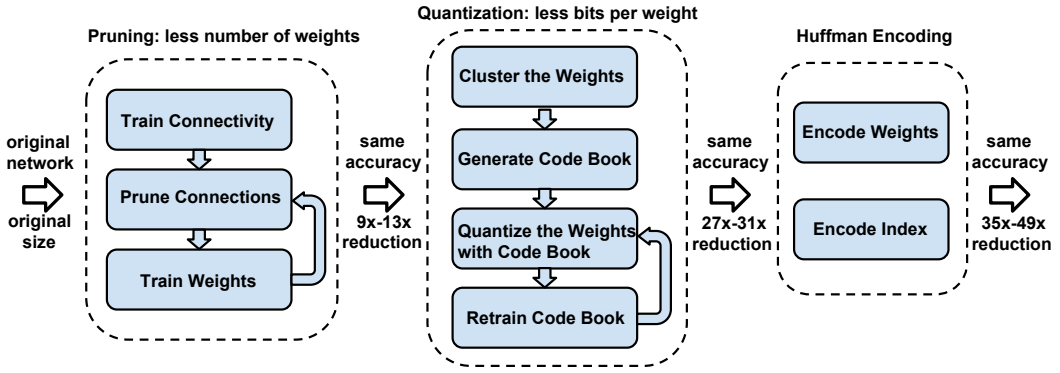

Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by $10\times$, while quantization further improves the compression rate: between $27\times$ and $31\times$. Huffman coding gives more compression: between $35\times$ and $49\times$. 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 $\mathbf{0.9pJ}$, a 32bit SRAM cache access takes $\mathbf{5pJ}$, while a 32bit DRAM memory access takes $\mathbf{640pJ}$, 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; Ström, 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.

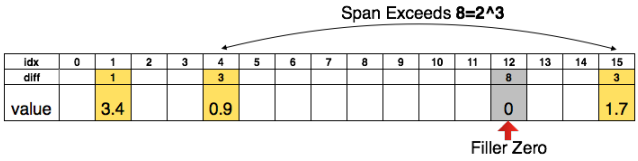

Figure 2: Representing the matrix sparsity with relative index. Padding filler zero to prevent overflow.

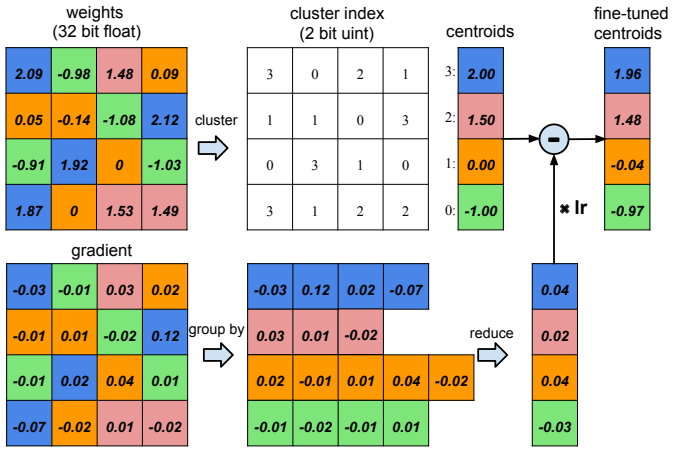

Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

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 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.

权重被量化到4个容器(用4种颜色表示)，同一容器中的所有权重共享相同的值，因此对于每个权重，我们只需要将一个小索引存储到共享权重表中

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:

为了计算压缩率，给定k个集群，我们只需要log2(k)位来编码索引。

$$r = \frac{nb}{nlog_2(k) + kb} \tag{1}$$

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

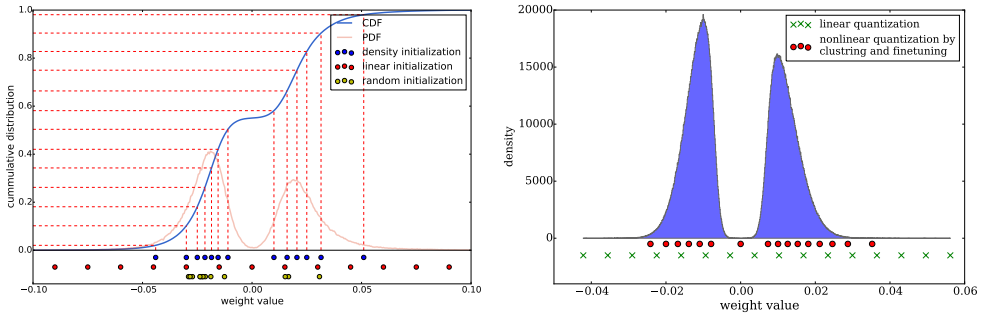

Figure 4: Left: Three different methods for centroids initialization. Right: Distribution of weights (blue) and distribution of codebook before (green cross) and after fine-tuning (red dot).

has 32 bits, now we need to store only 4 effective weights (blue, green, red and orange), each has 32 bits, together with 16 2-bit indices giving a compression rate of $16 * 32/(4 * 32 + 2 * 16) = 3.2$

3.1 Weight Sharing

We use k-means clustering to identify the shared weights for each layer of a trained network, so that all the weights that fall into the same cluster will share the same weight. Weights are not shared across layers. We partition n original weights $W = \{w_1, w_2, ..., w_n\}$ into k clusters $C = \{c_1, c_2, ..., c_k\}$, $n \gg k$, so as to minimize the within-cluster sum of squares (WCSS):

我们使用k-means聚类来识别一个训练过的网络的每一层的共享权重，这样所有属于同一个集群的权重都将共享相同的权重。

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

Different from HashNet (Chen et al., 2015) where weight sharing is determined by a hash function before the networks sees any training data, our method determines weight sharing after a network is fully trained, so that the shared weights approximate the original network.

3.2 Initialization of Shared Weights

Centroid initialization impacts the quality of clustering and thus affects the network's prediction accuracy. We examine three initialization methods: Forgy(random), density-based, and linear initialization. In Figure 4 we plotted the original weights' distribution of conv3 layer in AlexNet (CDF in blue, PDF in red). The weights forms a bimodal distribution after network pruning. On the bottom it plots the effective weights (centroids) with 3 different initialization methods (shown in blue, red and yellow). In this example, there are 13 clusters.

Forgy (random) initialization randomly chooses k observations from the data set and uses these as the initial centroids. The initialized centroids are shown in yellow. Since there are two peaks in the bimodal distribution, Forgy method tend to concentrate around those two peaks.

Density-based initialization linearly spaces the CDF of the weights in the y-axis, then finds the horizontal intersection with the CDF, and finally finds the vertical intersection on the x-axis, which becomes a centroid, as shown in blue dots. This method makes the centroids denser around the two peaks, but more scatted than the Forgy method.

Linear initialization linearly spaces the centroids between the [min, max] of the original weights. This initialization method is invariant to the distribution of the weights and is the most scattered compared with the former two methods.

Larger weights play a more important role than smaller weights (Han et al., 2015), but there are fewer of these large weights. Thus for both Forgy initialization and density-based initialization, very few centroids have large absolute value which results in poor representation of these few large weights. Linear initialization does not suffer from this problem. The experiment section compares the accuracy

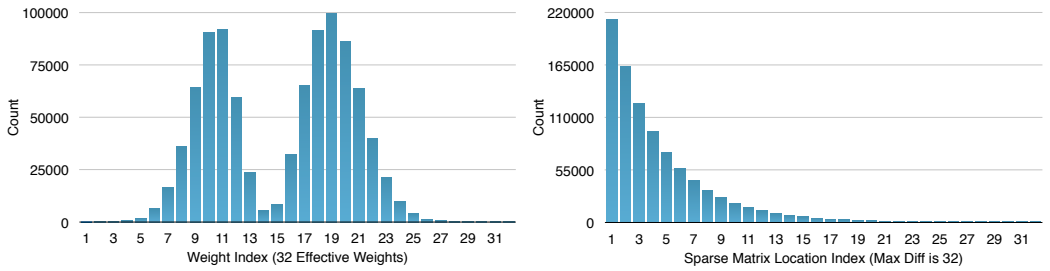

Figure 5: Distribution for weight (Left) and index (Right). The distribution is biased.

of different initialization methods after clustering and fine-tuning, showing that linear initialization works best.

3.3 Feed-forward and Back-propagation

The centroids of the one-dimensional k-means clustering are the shared weights. There is one level of indirection during feed forward phase and back-propagation phase looking up the weight table. An index into the shared weight table is stored for each connection. During back-propagation, the gradient for each shared weight is calculated and used to update the shared weight. This procedure is shown in Figure 3.

We denote the loss by $\mathcal{L}$, the weight in the ith column and jth row by W_{ij}, the centroid index of element $W_{i,j}$ by I_{ij}, the kth centroid of the layer by C_k. By using the indicator function $\mathbb{1}(.)$, the gradient of the centroids is calculated as:

$$\frac{\partial \mathcal{L}}{\partial C_k} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial W_{ij}} \frac{\partial W_{ij}}{\partial C_k} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial W_{ij}} \mathbb{1}(I_{ij} = k) \tag{3}$$

4 Huffman Coding

A Huffman code is an optimal prefix code commonly used for lossless data compression(Van Leeuwen, 1976). It uses variable-length codewords to encode source symbols. The table is derived from the occurrence probability for each symbol. More common symbols are represented with fewer bits.

Figure 5 shows the probability distribution of quantized weights and the sparse matrix index of the last fully connected layer in AlexNet. Both distributions are biased: most of the quantized weights are distributed around the two peaks; the sparse matrix index difference are rarely above 20. Experiments show that Huffman coding these non-uniformly distributed values saves $20\% - 30\%$ of network storage.

5 Experiments

We pruned, quantized, and Huffman encoded four networks: two on MNIST and two on ImageNet data-sets. The network parameters and accuracy-[1] before and after pruning are shown in Table 1. The compression pipeline saves network storage by $35\times$ to $49\times$ across different networks without loss of accuracy. The total size of AlexNet decreased from 240MB to 6.9MB, which is small enough to be put into on-chip SRAM, eliminating the need to store the model in energy-consuming DRAM memory.

Training is performed with the Caffe framework (Jia et al., 2014). Pruning is implemented by adding a mask to the blobs to mask out the update of the pruned connections. Quantization and weight sharing are implemented by maintaining a codebook structure that stores the shared weight, and group-by-index after calculating the gradient of each layer. Each shared weight is updated with all the gradients that fall into that bucket. Huffman coding doesn't require training and is implemented offline after all the fine-tuning is finished.

5.1 LeNet-300-100 and LeNet-5 on MNIST

We first experimented on MNIST dataset with LeNet-300-100 and LeNet-5 network (LeCun et al., 1998). LeNet-300-100 is a fully connected network with two hidden layers, with 300 and 100

[1]Reference model is from Caffe model zoo, accuracy is measured without data augmentation

Table 1: The compression pipeline can save $35\times$ to $49\times$ 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**×)

neurons each, which achieves 1.6% error rate on Mnist. LeNet-5 is a convolutional network that has two convolutional layers and two fully connected layers, which achieves 0.8% error rate on Mnist. Table 2 and table 3 show the statistics of the compression pipeline. The compression rate includes the overhead of the codebook and sparse indexes. Most of the saving comes from pruning and quantization (compressed $32\times$), while Huffman coding gives a marginal gain (compressed $40\times$)

5.2 ALEXNET ON IMAGENET

We further examine the performance of Deep Compression on the ImageNet ILSVRC-2012 dataset, which has 1.2M training examples and 50k validation examples. We use the AlexNet Caffe model as the reference model, which has 61 million parameters and achieved a top-1 accuracy of 57.2% and a top-5 accuracy of 80.3%. Table 4 shows that AlexNet can be compressed to 2.88% of its original size without impacting accuracy. There are 256 shared weights in each CONV layer, which are encoded with 8 bits, and 32 shared weights in each FC layer, which are encoded with only 5 bits. The relative sparse index is encoded with 4 bits. Huffman coding compressed additional 22%, resulting in $35\times$ compression in total.

5.3 VGG-16 ON IMAGENET

With promising results on AlexNet, we also looked at a larger, more recent network, VGG-16 (Simonyan & Zisserman, 2014), on the same ILSVRC-2012 dataset. VGG-16 has far more convolutional layers but still only three fully-connected layers. Following a similar methodology, we aggressively compressed both convolutional and fully-connected layers to realize a significant reduction in the number of effective weights, shown in Table5.

The VGG16 network as a whole has been compressed by $49\times$. Weights in the CONV layers are represented with 8 bits, and FC layers use 5 bits, which does not impact the accuracy. The two largest fully-connected layers can each be pruned to less than 1.6% of their original size. This reduction

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**×)

is critical for real time image processing, where there is little reuse of these layers across images (unlike batch processing). This is also critical for fast object detection algorithms where one CONV pass is used by many FC passes. The reduced layers will fit in an on-chip SRAM and have modest bandwidth requirements. Without the reduction, the bandwidth requirements are prohibitive.

6 Discussions

6.1 Pruning and Quantization Working Together

Figure 6 shows the accuracy at different compression rates for pruning and quantization together or individually. When working individually, as shown in the purple and yellow lines, accuracy of pruned network begins to drop significantly when compressed below 8% of its original size; accuracy of quantized network also begins to drop significantly when compressed below 8% of its original size. But when combined, as shown in the red line, the network can be compressed to 3% of original size with no loss of accuracy. On the far right side compared the result of SVD, which is inexpensive but has a poor compression rate.

The three plots in Figure 7 show how accuracy drops with fewer bits per connection for CONV layers (left), FC layers (middle) and all layers (right). Each plot reports both top-1 and top-5 accuracy. Dashed lines only applied quantization but without pruning; solid lines did both quantization and pruning. There is very little difference between the two. This shows that pruning works well with quantization.

Quantization works well on pruned network because unpruned AlexNet has 60 million weights to quantize, while pruned AlexNet has only 6.7 million weights to quantize. Given the same amount of centroids, the latter has less error.

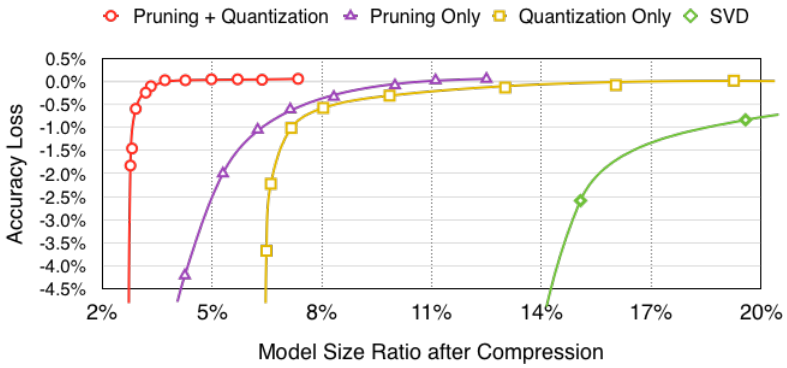

Figure 6: Accuracy v.s. compression rate under different compression methods. Pruning and quantization works best when combined.

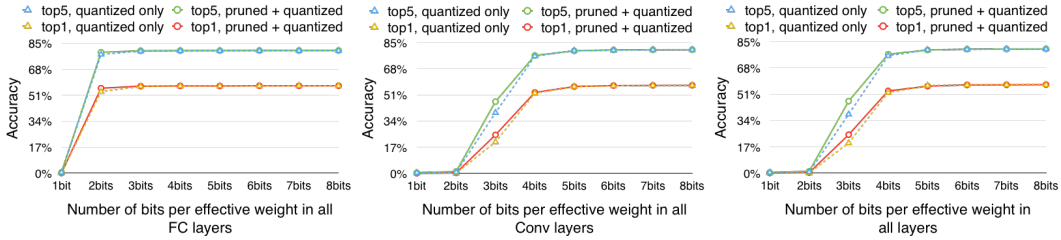

Figure 7: Pruning doesn't hurt quantization. Dashed: quantization on unpruned network. Solid: quantization on pruned network; Accuracy begins to drop at the same number of quantization bits whether or not the network has been pruned. Although pruning made the number of parameters less, quantization still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

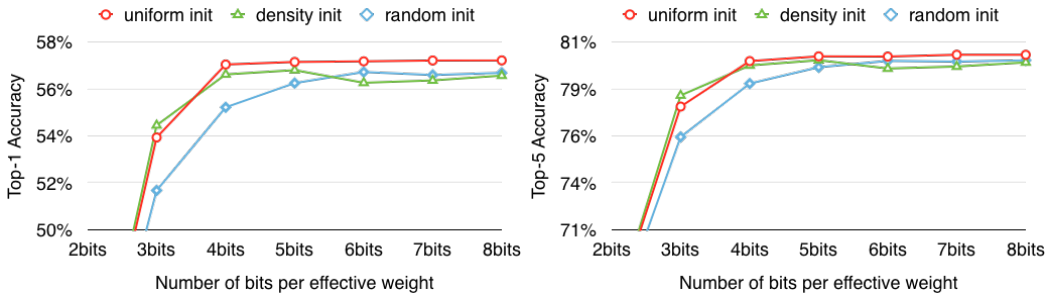

Figure 8: Accuracy of different initialization methods. Left: top-1 accuracy. Right: top-5 accuracy. Linear initialization gives best result.

The first two plots in Figure 7 show that CONV layers require more bits of precision than FC layers. For CONV layers, accuracy drops significantly below 4 bits, while FC layer is more robust: not until 2 bits did the accuracy drop significantly.

6.2 Centroid Initialization

Figure 8 compares the accuracy of the three different initialization methods with respect to top-1 accuracy (Left) and top-5 accuracy (Right). The network is quantized to $2 \sim 8$ bits as shown on x-axis. Linear initialization outperforms the density initialization and random initialization in all cases except at 3 bits.

The initial centroids of linear initialization spread equally across the x-axis, from the min value to the max value. That helps to maintain the large weights as the large weights play a more important role than smaller ones, which is also shown in network pruning Han et al. (2015). Neither random nor density-based initialization retains large centroids. With these initialization methods, large weights are clustered to the small centroids because there are few large weights. In contrast, linear initialization allows large weights a better chance to form a large centroid.

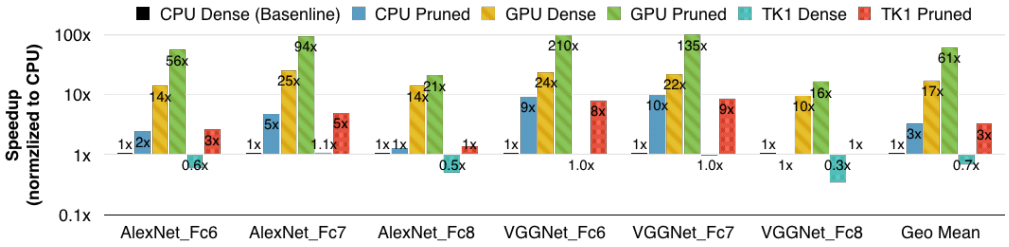

Figure 9: Compared with the original network, pruned network layer achieved $3\times$ speedup on CPU, $3.5\times$ on GPU and $4.2\times$ on mobile GPU on average. Batch size = 1 targeting real time processing. Performance number normalized to CPU.

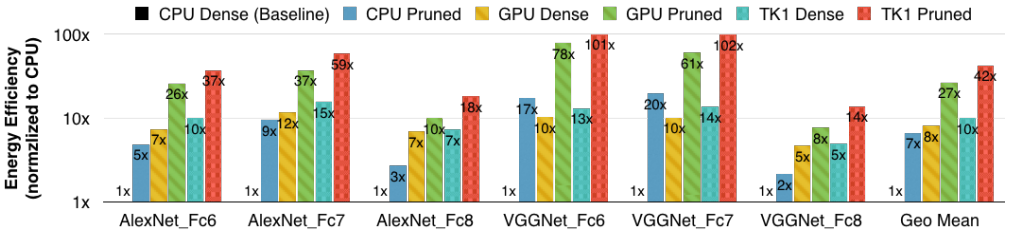

Figure 10: Compared with the original network, pruned network layer takes $7\times$ less energy on CPU, $3.3\times$ less on GPU and $4.2\times$ less on mobile GPU on average. Batch size = 1 targeting real time processing. Energy number normalized to CPU.

6.3 Speedup and Energy Efficiency

Deep Compression is targeting extremely latency-focused applications running on mobile, which requires real-time inference, such as pedestrian detection on an embedded processor inside an autonomous vehicle. Waiting for a batch to assemble significantly adds latency. So when benchmarking the performance and energy efficiency, we consider the case when batch size = 1. The cases of batching are given in Appendix A.

Fully connected layer dominates the model size (more than 90%) and got compressed the most by Deep Compression (96% weights pruned in VGG-16). In state-of-the-art object detection algorithms such as fast R-CNN (Girshick, 2015), upto 38% computation time is consumed on FC layers on uncompressed model. So it's interesting to benchmark on FC layers, to see the effect of Deep Compression on performance and energy. Thus we setup our benchmark on FC6, FC7, FC8 layers of AlexNet and VGG-16. In the non-batched case, the activation matrix is a vector with just one column, so the computation boils down to dense / sparse matrix-vector multiplication for original / pruned model, respectively. Since current BLAS library on CPU and GPU doesn't support indirect look-up and relative indexing, we didn't benchmark the quantized model.

We compare three different off-the-shelf hardware: the NVIDIA GeForce GTX Titan X and the Intel Core i7 5930K as desktop processors (same package as NVIDIA Digits Dev Box) and NVIDIA Tegra K1 as mobile processor. To run the benchmark on GPU, we used cuBLAS GEMV for the original dense layer. For the pruned sparse layer, we stored the sparse matrix in in CSR format, and used cuSPARSE CSRMV kernel, which is optimized for sparse matrix-vector multiplication on GPU. To run the benchmark on CPU, we used MKL CBLAS GEMV for the original dense model and MKL SPBLAS CSRMV for the pruned sparse model.

To compare power consumption between different systems, it is important to measure power at a consistent manner (NVIDIA, b). For our analysis, we are comparing pre-regulation power of the entire application processor (AP) / SOC and DRAM combined. On CPU, the benchmark is running on single socket with a single Haswell-E class Core i7-5930K processor. CPU socket and DRAM power are as reported by the `pcm-power` utility provided by Intel. For GPU, we used `nvidia-smi` utility to report the power of Titan X. For mobile GPU, we use a Jetson TK1 development board and measured the total power consumption with a power-meter. We assume 15% AC to DC conversion loss, 85% regulator efficiency and 15% power consumed by peripheral components (NVIDIA, a) to report the AP+DRAM power for Tegra K1.

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%

The ratio of memory access over computation characteristic with and without batching is different. When the input activations are batched to a matrix the computation becomes matrix-matrix multiplication, where locality can be improved by blocking. Matrix could be blocked to fit in caches and reused efficiently. In this case, the amount of memory access is $O(n^2)$, and that of computation is $O(n^3)$, the ratio between memory access and computation is in the order of $1/n$.

In real time processing when batching is not allowed, the input activation is a single vector and the computation is matrix-vector multiplication. In this case, the amount of memory access is $O(n^2)$, and the computation is $O(n^2)$, memory access and computation are of the same magnitude (as opposed to $1/n$). That indicates MV is more memory-bounded than MM. So reducing the memory footprint is critical for the non-batching case.

Figure 9 illustrates the speedup of pruning on different hardware. There are 6 columns for each benchmark, showing the computation time of CPU / GPU / TK1 on dense / pruned network. Time is normalized to CPU. When batch size = 1, pruned network layer obtained $3\times$ to $4\times$ speedup over the dense network on average because it has smaller memory footprint and alleviates the data transferring overhead, especially for large matrices that are unable to fit into the caches. For example VGG16's FC6 layer, the largest layer in our experiment, contains $25088 \times 4096 \times 4\ Bytes \approx 400MB$ data, which is far from the capacity of L3 cache.

In those latency-tolerating applications , batching improves memory locality, where weights could be blocked and reused in matrix-matrix multiplication. In this scenario, pruned network no longer shows its advantage. We give detailed timing results in Appendix A.

Figure 10 illustrates the energy efficiency of pruning on different hardware. We multiply power consumption with computation time to get energy consumption, then normalized to CPU to get energy efficiency. When batch size = 1, pruned network layer consumes $3\times$ to $7\times$ less energy over the dense network on average. Reported by `nvidia-smi`, GPU utilization is 99% for both dense and sparse cases.

6.4 Ratio of Weights, Index and Codebook

Pruning makes the weight matrix sparse, so extra space is needed to store the indexes of non-zero elements. Quantization adds storage for a codebook. The experiment section has already included these two factors. Figure 11 shows the breakdown of three different components when quantizing four networks. Since on average both the weights and the sparse indexes are encoded with 5 bits, their storage is roughly half and half. The overhead of codebook is very small and often negligible.

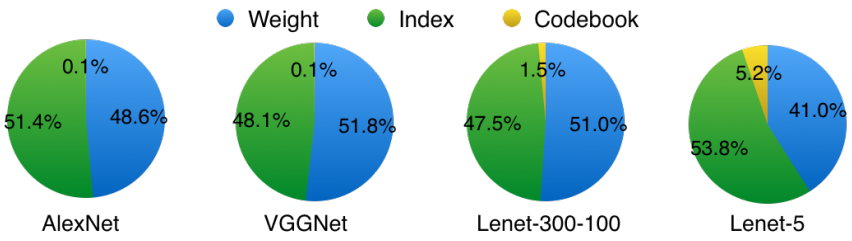

Figure 11: Storage ratio of weight, index and codebook.

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

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

7 Related Work

Neural networks are typically over-parametrized, and there is significant redundancy for deep learning models(Denil et al., 2013). This results in a waste of both computation and memory usage. There have been various proposals to remove the redundancy: Vanhoucke et al. (2011) explored a fixed-point implementation with 8-bit integer (vs 32-bit floating point) activations. Hwang & Sung (2014) proposed an optimization method for the fixed-point network with ternary weights and 3-bit activations. Anwar et al. (2015) quantized the neural network using L2 error minimization and achieved better accuracy on MNIST and CIFAR-10 datasets.Denton et al. (2014) exploited the linear structure of the neural network by finding an appropriate low-rank approximation of the parameters and keeping the accuracy within 1% of the original model.

The empirical success in this paper is consistent with the theoretical study of random-like sparse networks with +1/0/-1 weights (Arora et al., 2014), which have been proved to enjoy nice properties (e.g. reversibility), and to allow a provably polynomial time algorithm for training.

Much work has been focused on binning the network parameters into buckets, and only the values in the buckets need to be stored. HashedNets(Chen et al., 2015) reduce model sizes by using a hash function to randomly group connection weights, so that all connections within the same hash bucket share a single parameter value. In their method, the weight binning is pre-determined by the hash function, instead of being learned through training, which doesn't capture the nature of images. Gong et al. (2014) compressed deep convnets using vector quantization, which resulted in 1% accuracy loss. Both methods studied only the fully connected layer, ignoring the convolutional layers.

There have been other attempts to reduce the number of parameters of neural networks by replacing the fully connected layer with global average pooling. The Network in Network architecture(Lin et al., 2013) and GoogLenet(Szegedy et al., 2014) achieves state-of-the-art results on several benchmarks by adopting this idea. However, transfer learning, i.e. 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. This problem is noted by Szegedy et al. (2014) and motivates them to 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 over-fitting. An early approach to pruning was biased weight decay (Hanson & Pratt, 1989). Optimal Brain Damage (LeCun et al., 1989) and Optimal Brain Surgeon (Hassibi et al., 1993) prune networks to reduce the number of connections based on the Hessian of the loss function and suggest that such pruning is more accurate than magnitude-based pruning such as weight decay. A recent work (Han et al., 2015) successfully pruned several state of the art large scale networks and showed that the number of parameters could be reduce by an order of magnitude. There are also attempts to reduce the number of activations for both compression and acceleration Van Nguyen et al. (2015).

8 FUTURE WORK

While the *pruned* network has been benchmarked on various hardware, the *quantized* network with weight sharing has not, because off-the-shelf cuSPARSE or MKL SPBLAS library does not support indirect matrix entry lookup, nor is the relative index in CSC or CSR format supported. So the full advantage of Deep Compression that fit the model in cache is not fully unveiled. A software solution is to write customized GPU kernels that support this. A hardware solution is to build custom ASIC architecture specialized to traverse the sparse and quantized network structure, which also supports customized quantization bit width. We expect this architecture to have energy dominated by on-chip SRAM access instead of off-chip DRAM access.

9 CONCLUSION

We have presented "Deep Compression" that compressed neural networks without affecting accuracy. Our method operates by pruning the unimportant connections, quantizing the network using weight sharing, and then applying Huffman coding. We highlight our experiments on AlexNet which reduced the weight storage by $35\times$ without loss of accuracy. We show similar results for VGG-16 and LeNet networks compressed by $49\times$ and $39\times$ without loss of accuracy. This leads to smaller storage requirement of putting convnets into mobile app. After Deep Compression the size of these networks fit into on-chip SRAM cache (5pJ/access) rather than requiring off-chip DRAM memory (640pJ/access). This potentially makes deep neural networks more energy efficient to run on mobile. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained.

REFERENCES

Anwar, Sajid, Hwang, Kyuyeon, and Sung, Wonyong. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pp. 1131–1135. IEEE, 2015.

Arora, Sanjeev, Bhaskara, Aditya, Ge, Rong, and Ma, Tengyu. Provable bounds for learning some deep representations. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014*, pp. 584–592, 2014.

BVLC. Caffe model zoo. URL `http://caffe.berkeleyvision.org/model_zoo`.

Chen, Wenlin, Wilson, James T., Tyree, Stephen, Weinberger, Kilian Q., and Chen, Yixin. Compressing neural networks with the hashing trick. *arXiv preprint arXiv:1504.04788*, 2015.

Collins, Maxwell D and Kohli, Pushmeet. Memory bounded deep convolutional networks. *arXiv preprint arXiv:1412.1442*, 2014.

Denil, Misha, Shakibi, Babak, Dinh, Laurent, de Freitas, Nando, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pp. 2148–2156, 2013.

Denton, Emily L, Zaremba, Wojciech, Bruna, Joan, LeCun, Yann, and Fergus, Rob. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pp. 1269–1277, 2014.

Girshick, Ross. Fast r-cnn. *arXiv preprint arXiv:1504.08083*, 2015.

Gong, Yunchao, Liu, Liu, Yang, Ming, and Bourdev, Lubomir. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Han, Song, Pool, Jeff, Tran, John, and Dally, William J. Learning both weights and connections for efficient neural networks. In *Advances in Neural Information Processing Systems*, 2015.

Han, Song, Liu, Xingyu, Mao, Huizi, Pu, Jing, Pedram, Ardavan, Horowitz, Mark A, and Dally, William J. EIE: Efficient inference engine on compressed deep neural network. *arXiv preprint arXiv:1602.01528*, 2016.

Hanson, Stephen José and Pratt, Lorien Y. Comparing biases for minimal network construction with back-propagation. In *Advances in neural information processing systems*, pp. 177–185, 1989.

Hassibi, Babak, Stork, David G, et al. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, pp. 164–164, 1993.

Hwang, Kyuyeon and Sung, Wonyong. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6. IEEE, 2014.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *NIPS*, pp. 1097–1105, 2012.

LeCun, Yann, Denker, John S, Solla, Sara A, Howard, Richard E, and Jackel, Lawrence D. Optimal brain damage. In *NIPs*, volume 89, 1989.

LeCun, Yann, Bottou, Leon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lin, Min, Chen, Qiang, and Yan, Shuicheng. Network in network. *arXiv:1312.4400*, 2013.

NVIDIA. Technical brief: NVIDIA jetson TK1 development kit bringing GPU-accelerated computing to embedded systems, a. URL `http://www.nvidia.com`.

NVIDIA. Whitepaper: GPU-based deep learning inference: A performance and power analysis, b. URL `http://www.nvidia.com/object/white-papers.html`.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Ström, Nikko. Phoneme probability estimation with dynamic sparsely connected artificial neural networks. *The Free Speech Journal*, 1(5):1–41, 1997.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

Van Leeuwen, Jan. On the construction of huffman trees. In *ICALP*, pp. 382–410, 1976.

Van Nguyen, Hien, Zhou, Kevin, and Vemulapalli, Raviteja. Cross-domain synthesis of medical images using efficient location-sensitive deep network. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015*, pp. 677–684. Springer, 2015.

Vanhoucke, Vincent, Senior, Andrew, and Mao, Mark Z. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

Yang, Zichao, Moczulski, Marcin, Denil, Misha, de Freitas, Nando, Smola, Alex, Song, Le, and Wang, Ziyu. Deep fried convnets. *arXiv preprint arXiv:1412.7149*, 2014.

A APPENDIX: DETAILED TIMING / POWER REPORTS OF DENSE & SPARSE NETWORK LAYERS

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

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