A Deep Neural Network Compression Pipeline: Pruning, Quantization, Huffman Encoding

Song Han[1] **Huizi Mao**[2] **William J. Dally**[1]
[1]Stanford University [2]Tsinghua University
{songhan, huizi, 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 a three stage pipeline: pruning, quantization and Huffman encoding, that work together to reduce the storage requirement of neural networks by $35\times$ to $49\times$ 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 encoding. 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 $9\times$ to $13\times$; 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 $\mathbf{35}\times$, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG16 by $\mathbf{49}\times$ 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, which has $180\times$ less access energy.

1 Introduction

Neural networks have become ubiquitous in applications including computer vision [1], speech recognition [2], and natural language processing [3]. In this paper We focus on the convolutional neural network(CNN) which has beat traditional algorithms in computer vision tasks. In 1998 Lecun *et al.* designed a CNN model LeNet-5 of less than 1M parameters to classify handwritten digits. Krizhevsky textitet al. [1] made breakthrough in ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) with 60M parameters. In 2014, Simonyan *et al.* made further leap to improve the accuracy by 10% with VGG-16 model of 138M parameters. Our study is based on these standard models.

Though these large neural networks are very powerful, their size consumes considerable storage, memory bandwidth, and computational resources — which in turn consumes considerable energy. For embedded mobile applications, these power demands become prohibitive. Mobile devices are battery constrained, making power hungry applications such as deep neural networks hard to deploy.

Energy consumption is dominated by memory access. A 32 bit float add consumes 1pJ under 45nm CMOS technology [4], a 32bit SRAM cache access takes 3.5pJ, while a 32bit LPDDR2 DRAM access takes 640pJ. From this data we see the energy per connection is dominated by memory access, which ranges from 3.5pJ for 32b coefficients in on-chip SRAM to 640pJ for 32b coefficients in off-chip LPDDR2 DRAM. Large networks do not fit in on-chip storage and hence require the more costly DRAM accesses. Running a 1G 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.

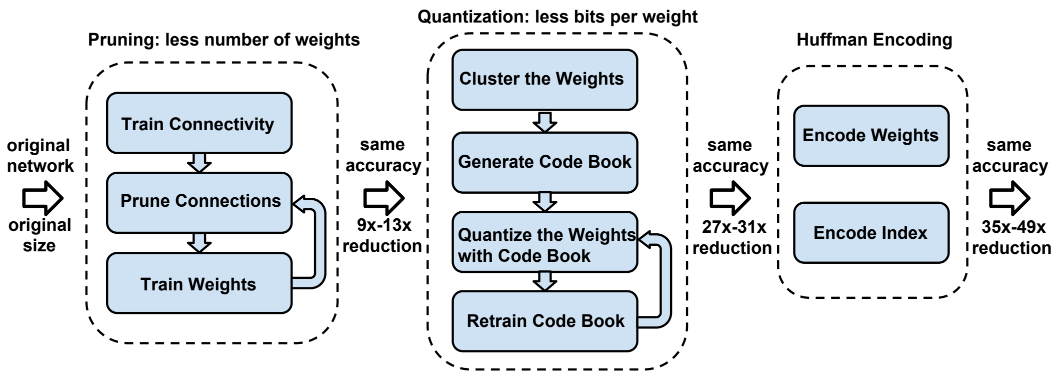


Figure 1: The three stage compression pipeline: pruning, quantization and Huffman Encoding. Pruning reduces the number of weights by $10\times$, while quantization further improve the compression rate to $27\times - 31\times$. Huffman encoding gives more reduction to $35\times$ to $50\times$. The compression rate already included the meta-date for sparse representation. The compression scheme doesn't incur any accuracy loss

Our goal in pruning and quantizing neural networks is to reduce the energy required to run such large networks so they can be run in real time on mobile devices.

To achieve this goal, we present a three-stage pipeline 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, making the connections sparse. 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 encoding to take advantage of the biased distribution of effective weights.

Our main insight is that, pruning and 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 even be cached on chip instead of the energy consuming data transfer between DRAM.

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 [5–7]. In [8], Han *et al.* pruned more recent large 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.

We store the sparse structure that results from pruning using compressed sparse row(CSR) or compressed sparse column (CSC) format [9], which requires $2nnz + n + 1$ numbers, where nnz 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 4 bits. When we need an index difference larger than 16, we have two solutions, as shown in figure 2. The first solution is zero padding: in case when the difference exceeds 16, the largest 4-bit unsigned number, we add a dummy zero. The second solution is adding overflow code, so that 1-15 represent normal jump, while 16 is the overflow code, meaning continue adding the next index difference to get a larger index difference.

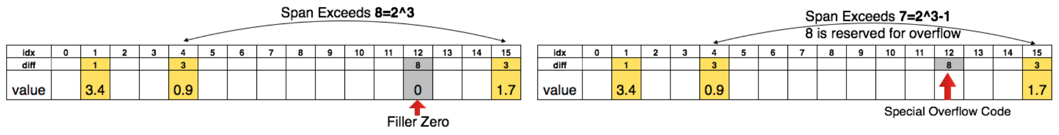


Figure 2: Two methods to prevent overflow. Left: add dummy zero. Right: have a special overflow code

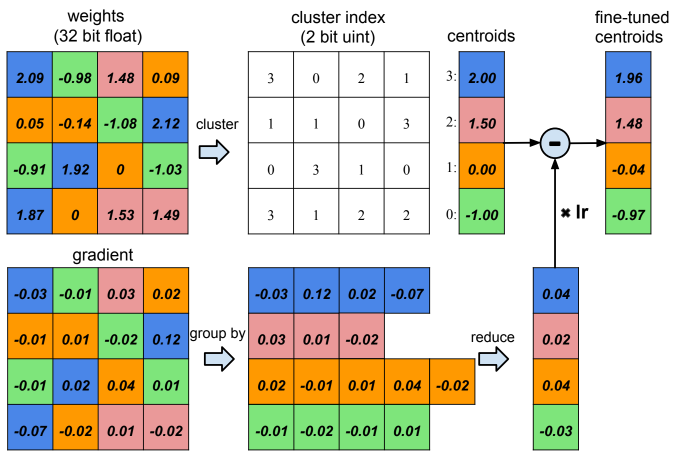


Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom)

3 Network Quantization and Weight Sharing

Network quantization, 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. For each connection, we then need to store only a small index into a table of shared weights. For pruned AlexNet, we are able to quantize to 8-bits (256 shared weights) for each CONV layers, and 4-bits (16 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:

$$r = \frac{Nb}{Nlog_2(K) + Kb} \tag{1}$$

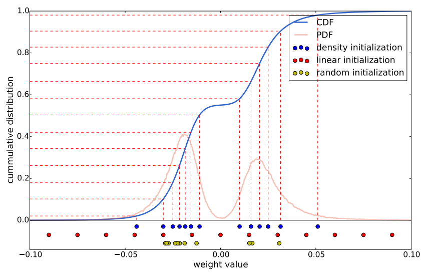
For example, figure 3 shows the weights of a single layer neural network with four input units and four output units. There are $4 \times 4 = 16$ weights originally but there are only 4 shared weights: similar weights are grouped together to share the same value. Originally we need to store 16 weights each 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 $C = \{c_1, c_2, ..., c_k\}$, $k \gg n$, so as to minimize the within-cluster sum of squares (WCSS):

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

Different from HashNet [10] 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.



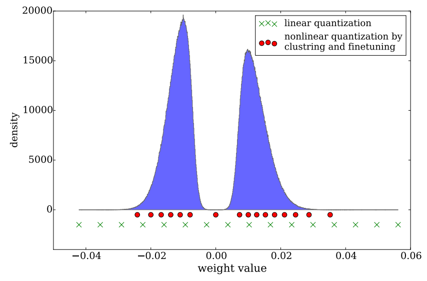


Figure 4: Left: Three different methods for centroids initialization. Right:Distribution of weights (blue) and distribution of codebook before(green) and after fine-tuning(red)

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. In this example, there are 13 clusters.

Forgy 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 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 dot. This method makes the centroids denser around the two peaks, but more scatted than the Forgy method.

Linear 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 [8]. However, larger weights have smaller quantity. Thus for both Forgy initialization and density based initialization, very few centroids have large absolute value which could harm the performance. Linear initialization does not suffer from this problem. The experiment section compares the accuracy of different initialization methods after clustering and fine-tuning.

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. 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.

Suppose that in a given layer there are n weights and m centroids. 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 could be calculated as below:

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

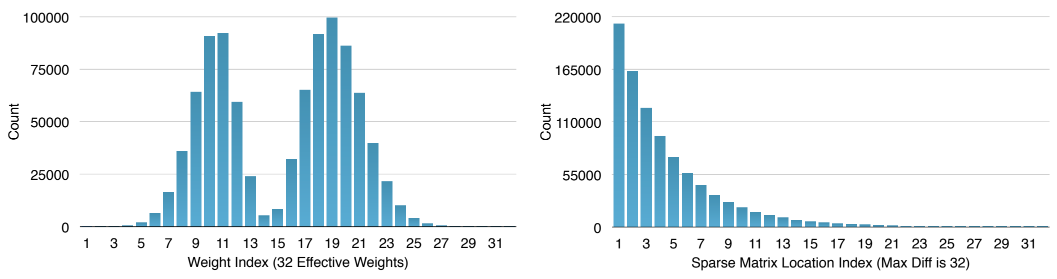


Figure 5: Distribution for weight (Left) and index (Right). The distribution is biased

4 Huffman Encoding

Huffman code is a type of optimal prefix code that is commonly used for loss-less data compression. It uses variable-length codewords to encode source symbol. 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, resulting a long tail. If those less frequently occurring weights and indexes could be represented with less number of bits, there could potentially have more storage saving.

Huffman encoding can be efficiently implemented, finding a code in linear time to the number of input weights if these weights are sorted [11]. Besides, Huffman code is a loss-less compression method, so the accuracy of the network won't be affected. Experiments shown Huffman encoding could save the last $20\% - 50\%$ storage of the network weights.

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 deteriorating the 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 Caffe framework [12]. Pruning is implemented by adding a mask to the blobs and mask out the update of those 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, then update the shared weight with all the gradients that fall into that bucket. Huffman encoding doesn't require training and is implemented offline after all the fine-tuning is finished.

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×**
VGG16 Ref	31.50%	11.32%	552 MB	
VGG16 Compressed	31.17%	10.91%	**11.3 MB**	**49×**

Table 1: The compression pipeline can save $35\times$ to $49\times$ parameter storage with no loss of accuracy

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

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (+H)	Index bits (P+Q)	Index bits (+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 2: Statistics of compression with LeNet-300-100. P: pruning, Q:quantization, H:Huffman Encoding

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (+H)	Index bits (P+Q)	Index bits (+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 3: Statistics of compression with LeNet-5. P: pruning, Q:quantization, H:Huffman Encoding

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (+H)	Index bits (P+Q)	Index bits (+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 4: Statistics of compression with AlexNet. P: pruning, Q: quantization, H:Huffman Encoding

5.1 LeNet on MNIST

We first experimented on MNIST dataset with LeNet-300-100 and LeNet-5 network [13]. LeNet-300-100 is a fully connected network with two hidden layers, with 300 and 100 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 compress rate already included the overhead of codebook and sparse indexes. Most of the saving comes from pruning and quantization (compressed to 3%), while Huffman coding gives marginal gain (down to 2.5%)

5.2 AlexNet on ImageNet

We further examine the performance of pruning 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 pruned to 2.88% of its original size without impacting accuracy. There are 256 shared weights in each conv layer, which are encoded with only 8 bits, and 32 shared weights in each fc layer, each encoded with 5 bits. The sparse index difference is encoded with 4 bits. Huffman encoding gains an additional 22%.

Layer	#Weights	Weights% (P)	Weigh bits (P+Q)	Weight bits (+H)	Index bits (P+Q)	Index bits (+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**×)

Table 5: Compression Statistics for VGG16. P: pruning, Q:quantization, H:Huffman Encoding

5.3 VGG16 on ImageNet

With promising results on AlexNet, we also looked at a larger, more recent network, VGG16 [14], on the same ILSVRC-2012 dataset. VGG16 has far more convolutional layers but still only three fully-connected layers. Following a similar methodology, we aggressively pruned both convolutional and fully-connected layers to realize a significant reduction in the number of weights, shown in Table5.

The VGG16 results are, like those for AlexNet, even more promising. The 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. In particular, note that the two largest fully-connected layers can each be pruned to less than 1.6% of their original size. This reduction 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 corresponds to 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 not 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 accuracy as a function of bits per connection for conv layers, fc layers and all layers. In each plot there are results for top-1 and top-5 accuracy. Dashed lines show the accuracy of quantization without pruning while solid lines show the accuracy of quantization after pruning. There is very little difference between the two. This shows that pruning works well with quantization.

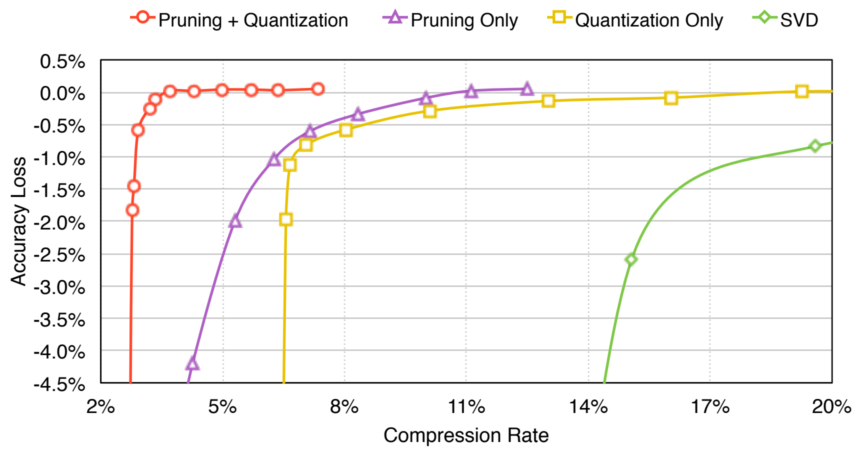


Figure 6: Accuracy v.s. compression rate under different compression methods. Pruning and Quantization works best when combined.

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. This shows the fc layer has more redundancy than the conv layer.

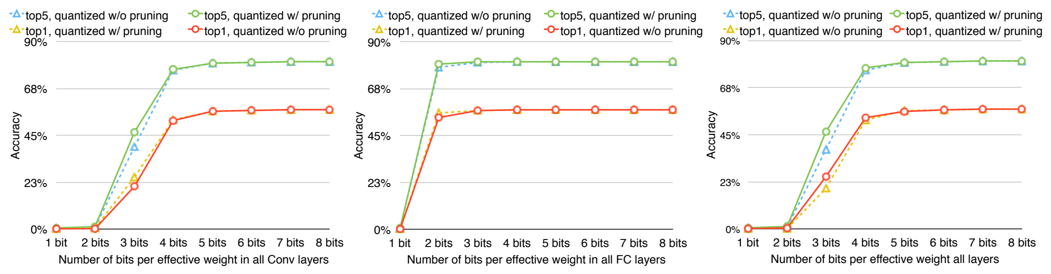


Figure 7: Pruning doesn't hurt quantization. Dashed: quantization on unpruned network. Solid: quantization on pruned network; Accuracy begins to drop at the same number of quantization bits no matter if the network has been pruned. This shows that although pruning made the number of parameters less, quantization still works well, or even better(3 bits case on the left figure)

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. In both figures, 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 [8]. Neither random nor density-based initialization tends to keep large centroids. And large weights will compromise to be clustered to the small centroids at convergence, since the proportion of large weights are small. On the contrary, linear initialization gave large weights a good chance to form a large centroid.

6.3 Ratio of Weights, Index and Codebook

Pruning made the weight matrix sparse, so extra space is needed to store the sparse index. Quantization adds storage for a codebook. All the above results have considered these two factors. Figure 9 showed the breakdown of three different components when quantizing four networks. Since on average both the weights and the sparse index are encoded with 5 bits, their storage is roughly half and half. The overhead of codebook is very small and often negligible.

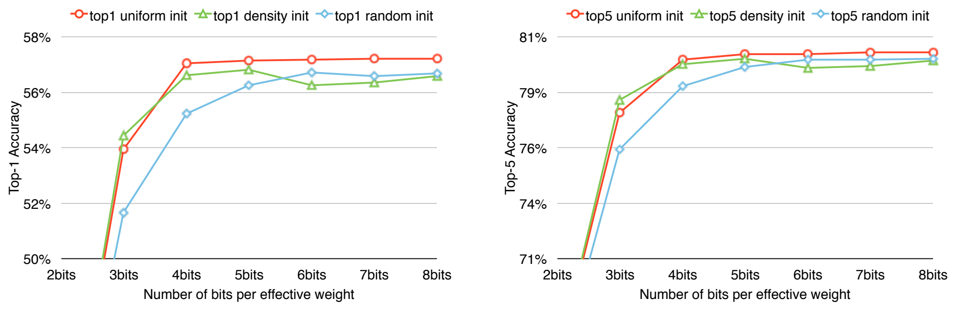


Figure 8: Accuracy of different initialization methods. Left: top-1 accuracy. Right: top-5 accuracy. Linear initialization gives best result.

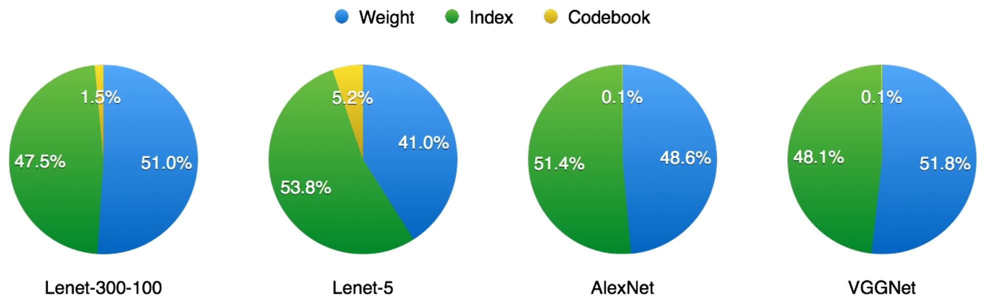


Figure 9: Storage ratio of weight, index and codebook.

7 Related Work

Neural networks are typically over-parametrized, and there is significant redundancy for deep learning models [15]. This results in a waste of both computation and memory usage. There have been various proposals to remove the redundancy: Vanhoucke *et al.* [16] explored a fixed-point implementation with 8-bit integer (vs 32-bit floating point) activations. Denton *et al.* [17] 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.

Much work has been focused on binning the network parameters into buckets, and only the values in the buckets need to be stored. HashedNets [10] is a recent technique to 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.* [18] 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 [22] and GoogLenet [23] 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 [23] 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 [6]. Optimal Brain Damage [5] and Optimal Brain Surgeon [7] 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. However, calculating the second derivative is costly for today's large scale neural networks. A recent work [8] 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.

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
Baseline Caffemodel [19]	42.78%	19.73%	240MB	$1\times$
Fastfood-32-AD [20]	41.93%	-	131MB	$2\times$
Fastfood-16-AD [20]	42.90%	-	64MB	$3.7\times$
Collins & Kohli [21]	44.40%	-	61MB	$4\times$
SVD [17]	44.02%	20.56%	47.6MB	$5\times$
Pruning [8]	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**	**$35\times$**

Table 6: Comparison with other model reduction methods on AlexNet. [21] reduced the parameters by $4\times$ and with inferior accuracy. Deep Fried Convnets [20] worked on fully connected layers only and reduced the parameters by less than $4\times$. SVD save parameters but suffers from large accuracy loss as much as 2%. Network pruning [8] reduced the parameters by $9\times$ without accuracy loss but the compression rate is only one third of this work. On other networks similar to AlexNet, [17] exploited linear structure of convnets and compressed the network by $2.4\times$ to $13.4\times$ layer wise, but had significant accuracy loss: as much as 0.9% even compressing a single layer. [18] experimented with vector quantization and compressed the network by $16\times -24\times$, but again incurred as much as 1% accuracy loss.

8 Conclusion

We have presented a method to improve the efficiency of neural networks without affecting accuracy by finding the right connections and quantizing the weights. Our method operates by pruning the unimportant connections, quantizing the network using weight sharing, and then applying Huffman encoding. We highlight our experiments on AlexNet on ImageNet, showing that both fully connected and convolutional layers can be pruned, quantized, and Huffman encoded, which reduced the weight storage by $35\times$ without loss of accuracy. We show similar results for VGG16 and LeNet networks compressed by $39\times$ and $49\times$ without accuracy loss. This leads to smaller memory capacity and bandwidth requirements for real-time image processing.

Our compression method reduces the size of these networks so that they fit into on-chip SRAM cache (3.5pJ/access) rather than requiring off-chip DRAM memory (640pJ/access). This gives an additional $180\times$ reduction in model access energy, giving a total reduction of access energy of $6,300\times$ to $8,800\times$.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[3] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.

[4] Mark Horowitz. Energy table for 45nm process.

[5] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990.

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

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

[8] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[9] Wikipedia. Sparse matrix.

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

[11] Jan van Leeuwen. On the construction of huffman trees. In *ICALP*, pages 382–410, 1976.

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

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

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

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

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

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

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

[19] Yangqing Jia. Bvlc caffe model zoo.

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

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

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

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