Compression of Deep Neural Networks

Srinivas Rao
New York University
svr300@nyu.edu

Prashant Garmella
New York University
pg1910@nyu.edu

Abstract

Since the revolution of deep neural networks, architectures like AlexNet and VGG16 have become the gold standards of computer vision tasks. However, as architectures like these became smarter and deeper, the model storage size and the inference time also increases. In order to counter this, we need DNNs with a lower number of parameters. This can be done using either compressing a pre-trained network or designing an already compressed/compact network. In compressing a pre-trained network we apply 'Deep Compression' techniques on LeNet to reduce the number of parameters in the network. In designing an already compact network, we use architectural design strategies and study an architecture called SqueezeNet. We evaluate the number of parameters and performance of SqueezeNet to compare SqueezeNet with AlexNet and VGG16.

1 Introduction

LeCun et al., 1998 was the 1^{st} to give the idea of stacking convolutional layers and pooling layers for computer vision tasks. In 2012 and 2014, AlexNet and VGG16 architectures reached state-of-the-art in computer vision tasks. AlexNet has 5 convolution layers and 3 fully connected layers and has a model size of 264MBs, VGG-16 has 13 convolution layers and 3 fully connected layers to have a model size of 525MBs. This ever-increasing number of layers in order to make the network smarter makes it difficult to deploy DNNs on mobile systems. Industries like the smartphone industry, autonomous vehicle industry, require a network that can be fit into a mobile device easily but to have a high accuracy at the same time. In order to achieve this, there can be two directions of study, which is done in the following section. First, to compress existing state-of-the-art architectures by applying techniques on them to reduce their number of parameters while keeping the same base architecture. The second direction being, to design an architecture from scratch that is compact by itself. Both of these directions are studied by reviewing a technique called 'Deep Compression' to study the first direction and an architecture called 'SqueezeNet' to study the second direction.

2 Related Work

Deep Neural networks typically have a large number of parameters and there is significant redundancy in deep learning models. This drains both computation and memory. Quite a lot of research has been done to remove redundancy. Vanhoucke et al. explored a fixed-point implementation. Denton et al. 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 with 8-bit integer (vs 32-bit floating point) activations. Gong et al. compressed deep convolution nets using vector quantization. Reducing the number of by replacing the fully connected layers with global average pooling also became famous. The Network in Network architecture and GoogLenet achieved state-of-the-art results on several benchmarks by adopting this idea.

All of these techniques were the motivation and building blocks of 'Deep Compression' as well as 'SqueezeNet' which are studied in depth in the following section.

3 Deep Compression and SqueezeNet Explained

Going further we'll be talking in more detail about the three stages in deep compression and we will be discussing the SqueezeNet architecture and how does it accomplish groundbreaking results with way lesser parameters than the standard.

3.1 Deep Compression

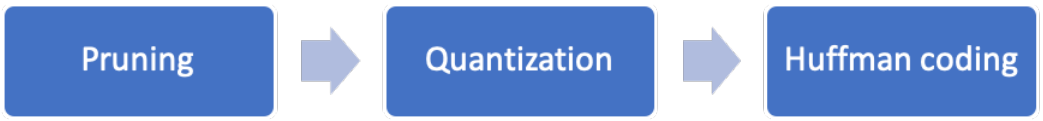


Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding.

3.1.1 Network Pruning

Pruning is a biological term in which the dead and diseased branches are cut off the main plant to encourage the growth of a healthier plant with structural integrity. In the world of deep learning, Pruning is a method employed in deep compression where we convert try to convert a dense neural network to a sparse neural network. Pruning involves some strategies which help to make a network sparse by removing the weights that contribute the least in providing accurate results. First, the neurons are ranked base on the L1/L2 norm of the neuron weights and then the weights with the lowest rankings are pruned. Just pruning a neural network causes a drop in accuracy. To solve that issue the pruned model is retrained with the dataset and we achieve very good accuracy after this step.

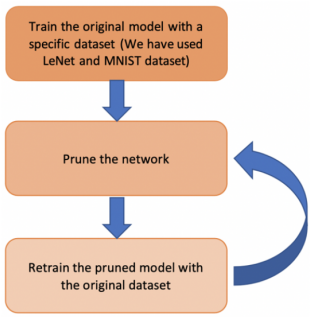


Figure 2: Steps in pruning. Pruning can be done in a single iteration of pruning and retraining or in multiple iterations.

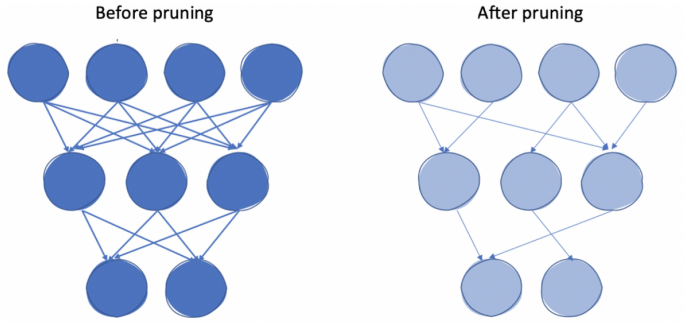


Figure 3: Connections before and after pruning. We can clearly see that the right diagram is sparse compared to the left diagram. This figure shows pruning of just the weights, but pruning can be done for the neurons as well.

We first train the LeNet model with the MNIST dataset for 100 epochs and then prune the connection by standard deviation and record the threshold values. And then we perform the retraining of the pruned model. We are able to prune the model by 95% and achieve the same accuracy as an unpruned network. As seen from Figure 2, Pruning and retraining can be done iteratively to achieve even more accuracy. But pruning can be only done to a certain extent. Pruning too much can result in a significant loss of accuracy. Once pruning has been implemented, we save the pruned model for further compression using the technique known as weight-sharing or quantization, which we will be discussing in detail in the section 3.1.2.

Pruning can be very valuable in future research works and can help tremendously in compressing and make it possible for deploying applications on smartphones, autonomous driving cars, and cloud. Pruning can cut the memory occupied by applications built on top of pruned networks to such extent that even the updates can be broadcasted over the internet, which would have been difficult in the case of unpruned networks. Pruning can be beneficial for performing transfer learning on a limited dataset. And not to forget the compressed models can be stored in SRAM rather than DRAM, which can decrease the energy consumption on mobile devices by manifolds.

3.1.2 Quantization

Weights in a neural network usually occupy 32 bits of memory. Quantization helps us bring down that number to as less as 2 to 4 bits depending on the type of layer (FC layer and convolution layer). Quantization is a fairly old and known concept in mathematics, where a large group of numbers is represented by a lesser group of numbers. For eg, let's take a group of numbers containing positive integers in the range n < 255. We can quantize this group of numbers using many functions like floor, K-means clustering to get a group of much lesser numbers. This way We can reduce both, the bits occupied by each weight, and finally there would be a significant decrease in total memory occupied by all the weights combined.

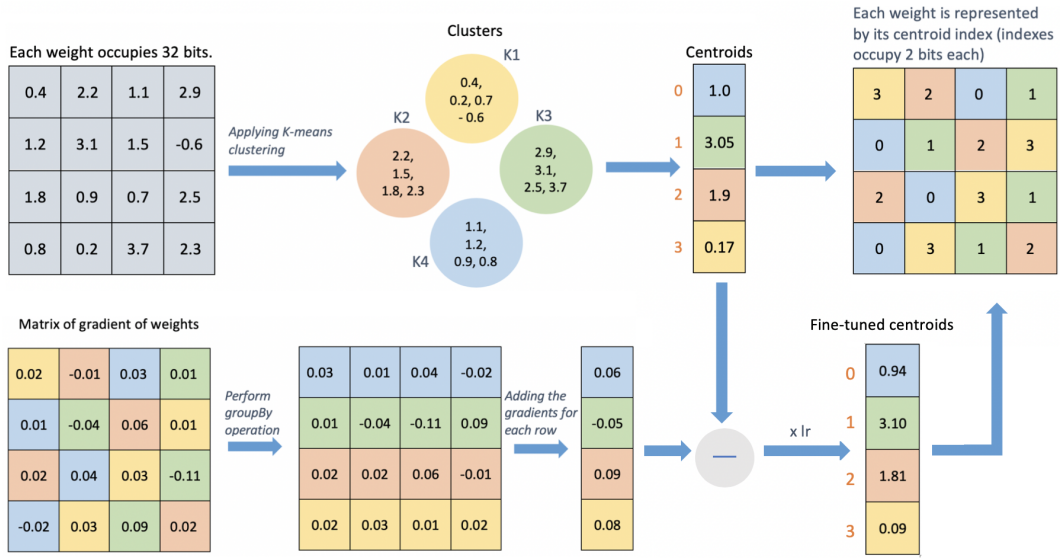


Figure 5: Quantization by K-means clustering (top) and centroids fine-tuning (bottom).

As we can see from Figure 5, the first 4 x 4 matrix represents the weight matrix and each cell occupies 32 bits. We perform k means clustering on the weights and get four centroids. The yellow-colored weights (K1 cluster) converge to 0.17, orange-colored weights (K2 cluster) to 1.9, green-colored weights (K3 cluster) converge to 3.05 and blue-colored weights (K4 cluster) converges to 1.0. We feed each of the centroid values to a dictionary where the indexes (Represented in orange as digits 0, 1, 2, and 3) are used to represent the centroids. As a result we use the index values instead of the centroid values to form the colored 4 x 4 matrix. Each index can be represented in exactly 2 bits in binary format. 0 can be represented as 00, 1 as 01, 2 as 10, and 3 as 11. This way each weight occupies only 2 bits. To further fine-tune our weights, we perform gradient descent. We group the gradient of the weights based on their cluster color and add them. The resulting 4 x 1 matrix is multiplied with a learning rate (in our case lr = 1) and then these values are subtracted from the original 4 x 1 centroid matrix. Then the final resulting matrix is represented by the indexes of these centroids and each cell now occupies 2 bits.

Let us represent the number of clusters using K, the number of connections or the number of weights using n, and the number of bits used to represent each weights by b. Let's say that we are trying to compress b bits to x bits. x = $log_2 k$ bits. The compression rate is represented by cr. We can find the compression rate by using the below formula:

$$cr = \frac{nb}{nx + Kb}$$

Using the above formula, we have $n = 4*4 = 16$. As mentioned in the Figure 5, $b = 32$ bits. Since we have $K = 4$ clusters, $x = log_2 K = log_2 4 = 2$ bits. Now that we have all the values, putting

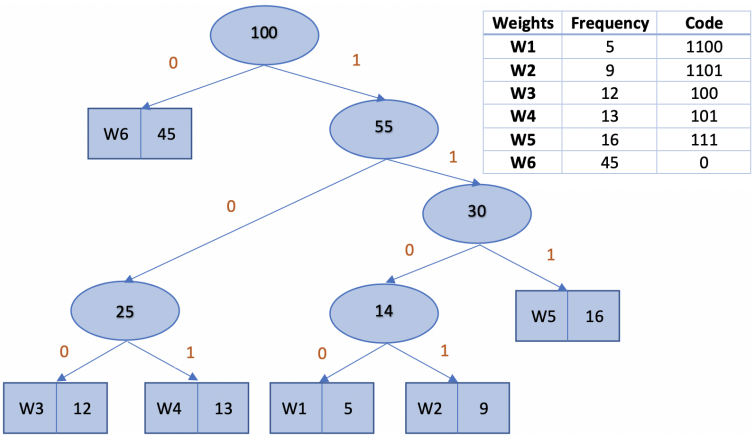
these in the above formula for compression rate, we get a compression rate of:

$$cr = \frac{16*32}{16*2 + 4*32} = 3.2$$

The weights in the FC layers can be compressed to 2 bits and the weights in the Convolution layer can be compressed to 4 bits. Not all neural networks can be compressed to the same values as mentioned before. Over compressing the network can result in a huge loss of accuracy. We compressed the FC layer weights to 5 bits and the Convolution layer weights to 8 bits to keep the accuracy of LeNet intact. Compared to pruning only, quantization does not improve or decrease the accuracy of the model but the main aim of quantization is to compress the network so that it occupies less space in the memory.

3.1.3 Huffman Coding

Huffman coding is a type of prefix code used for lossless data compression. In Huffman coding, the weights with a higher frequency are represented by a lesser number of bits, and the weights with lower frequency is represented by more number of bits. Let's look at the algorithm for Huffman coding. As seen from Figure 6, first the weights are arranged in decreasing order of their frequencies. Then we start building a min-heap from the bottom to the top, starting with the lowest two frequent weights (W1 and W2) and connect them to their parent, which is the sum of their frequencies ($f(W1) = 5, f(W2) = 9$ and $f(W1)+f(W2) = 14$). In min-heap, the higher frequency weights are represented as the right child and lower frequency weights are represented as the left child. Then we take the next higher frequency weights (W3 and W4). Similarly, we keep building the min-heap and reach the root of the tree. In this process, we represent the left branch of a node by 0 and the right branch of the node by 1. To get the prefix code of a weight, we traverse the path from the root node to the respective weight node, noting the 1's and 0's that we cross in the path. For eg, to find the code for W2, we start from the root node (100). We come across 1, then 1, then 0, and then 1. This translates to the code 1101. As we can see from the table in Figure 6, the highest frequency weight (W6) is represented by the least number of bits (1-bit), and the least frequent weights (W1 and W2) are represented using the most number of bits (4-bits). This is how compression is achieved using Huffman coding.



Weights	Frequency	Code
W1	5	1100
W2	9	1101
W3	12	100
W4	13	101
W5	16	111
W6	45	0

Figure 6: Minheap tree model used in Huffman coding

3.2 SqueezeNet

Above, we saw techniques of compressing CNNs after they are trained. But it would be of great interest if the network has a small model size itself such that it could have the benefits of a compressed network(low power consumption and low storage capacity) even without any compression. On similar thoughts, Han et. al proposed the SqueezeNet which used architectural design strategies to have a low model size.

3.2.1 Architectural Design Strategies

From various studies in CNNs, there have been architectural strategies proposed to have fewer parameters while maintaining good accuracy. SqueezeNet used some of these strategies applied together to form a compact architecture. These strategies are as below:

Using 1x1 filters instead of traditional 3X3 filters: Ever since the VGG architecture gained popularity, 3X3 filters started to be put into use in CNN architectures. Replacing these 3X3 filters with 1X1 filters makes the parameters 1/9 times than that of a 3X3 filter.

Reducing the number of input channels of 3X3 filters As we know that if a convolution layer has only 3x3 filters, the total number of parameters in this layer would be directly proportional to the number of input channels as *total parameters in a layer = no. of input channels * no. of filters * (3*3)*. So, along with decreasing the number of 3X3 filters, by decreasing the number of input channels to 3X3 filters, a large number of computations and parameters are saved upon.

Having large activation maps and Delayed Downsampling Applying pooling layers after convolution layers became a common practice in CNN architectures(e.g. (Szegedy et al., 2014; Simonyan Zisserman, 2014; Krizhevsky et al., 2012)). Every convolution layer in a CNN produces an output activation map on which generally pooling is applied. However, He Sun, 2015, applied delayed downsampling and found that it lead to better classification accuracy. The size of activation maps is dependent upon the size of the input data, the stride of pooling operations and convolution operations; and the layers on which the pooling operation is applied. Early downsampling would lead to smaller activation maps which can incur losses hence delayed downsampling i.e downsampling after 3-4 convolutional layers instead of every convolutional layer is suggested when a number of layers are stacked up.

3.2.2 The Fire Module

Han et al.,2016 defined the Fire module as depicted in Figure 7. In Figure 7, consider the 1X1 convolution operation, this is done so as to reduce the number of input channels in for a 3X3 convolution to follow. As it reduces the number of input channels, this part of the module is known as *squeeze layer*. Followed by this we have the activation map(after ReLU applied) to be passed into a chosen number of 1X1 and 3X3 filters. These are then concatenated and therefore the number of input channels is increased again, therefore this part is known as the expand layer. These chosen number of 1X1/3X3 filters in the fire module are the hyperparameters,s_{1x1} being the number of 1x1 filters in the squeeze layer,e_{1x1} and e_{3x3} being the number of 1x1 and 3X3 filters in the squeeze and expand layers respectively. Instead of simply applying 3X3 filters the traditional way, this squeeze-expand combination decreases the number of input channels to 3X3 filters. For this, s_{1x1} is set to be less than (e_{1x1} + e_{3x3}).

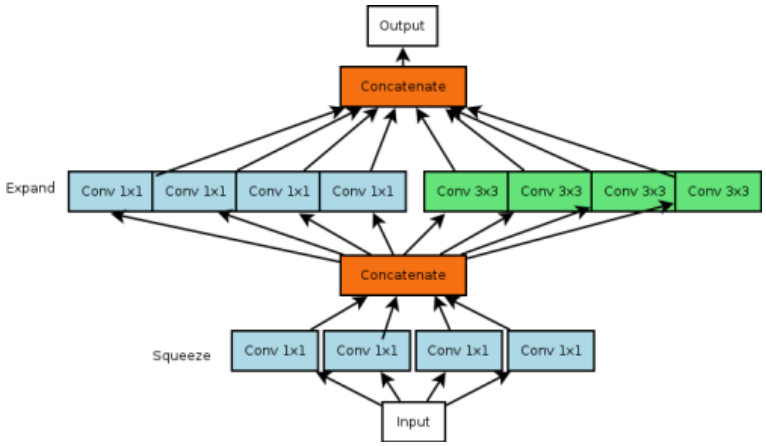


Figure 7: The fire module

3.2.3 The SqueezeNet Architecture

After seeing the design strategies and fire module, we study the full CNN architecture of the SqueezeNet. As we know, LeNet, AlexNet and VGG use the concept of stacking convolution and pooling layers. Now instead of stacking convolution and pooling layers, we stack up fire modules one after the other with varying hyperparameters. A SqueezeNet can be considered as a 10 layer architecture with a convolution layer applied to the input followed by 8 fire modules and then a convolution layer is used in the classifier. Here we use delayed downsampling strategy as explained in 3.2.1 and only use maxpool layer after conv1, fire4 and fire8 so as to keep larger output activation maps throughout. Also, CNN architectures use a fully connected layer(s) to apply softmax activation in the classifier. However, in the case of SqueezeNet, a convolution layer is used in the classifier instead of a fully connected layer.

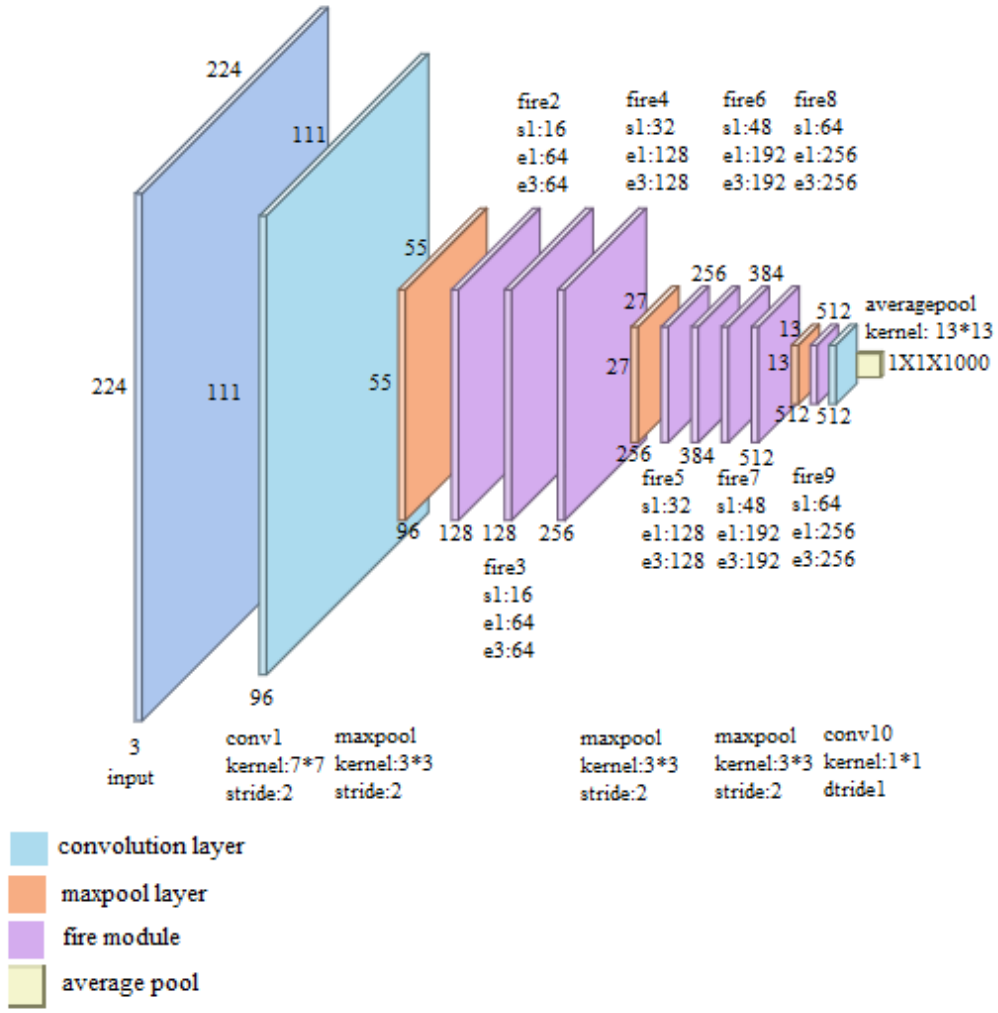


Figure 8: The SqueezeNet architecture.

4 Experiments

4.1 Evaluation of Deep Compression

We perform deep compression for a basic LeNet with only the fully connected layers on the MNIST dataset. We compare the three stages of Pruning, Quantization and Huffman coding, and evaluate the model parameters for the entire pipeline. After training the original LeNet model with the MNIST dataset, we record an accuracy of 95.24%. Then we prune the network for each of the fully connected layers and the final pruned network is 95.43% pruned. We record a compress rate of 21.88x and the accuarcy after pruning drop down to 90.95%. To get back the original accuracy, we retrain the pruned model with the original MNIST dataset and get an accuracy of 96.74%. We save the pruned model. After this we perform quantization on the previuosly saved pruned model. We set the weight bits and index bits to 5 bits (Earlier the weight bits were 32 bits). There is no change in compress rate and the number of weights at this step. We record the accuracy after quantization and find it to be 96.78%. In the end, we encode the weights using Huffman coding and achieve a compression of 25.37% over the model generated after Pruning and Quantization.

LAYER	#WEIGHTS BEFORE PRUNING	#WEIGHTS PRUNED	#WEIGHTS ALIVE AFTER PRUNING	WEIGHTS ALIVE AFTER PRUNING (%)	THRESHOLD	COMPRESS RATE
FC1	235200	224753	10447	4.44%	0.2320	22.51 x
FC2	30000	28749	1251	4.17%	0.1832	23.98 x
FC3	1000	922	78	7.8%	0.1880	12.82 x
TOTAL	266610	254424	12186	4.57%		21.88 x

Table 1: Compression results for LeNet after Pruning.

LAYER	#WEIGHTS BEFORE PRUNING	#WEIGHTS ALIVE AFTER PRUNING	WEIGHT BITS BEFORE QUANTIZATION	WEIGHT BITS AFTER QUANTIZATION	INDEX BITS AFTER QUANTIZATION	COMPRESS RATE
FC1	235200	10447	32	5	5	22.51 x
FC2	30000	1251	32	5	5	23.98 x
FC3	1000	78	32	5	5	12.82 x
TOTAL	266610	12186				21.88 x

Table 2: Compression results for LeNet after Quantization

LAYER	# OF BYTES BEFORE HUFFMAN CODING (B)	# OF BYTES AFTER HUFFMAN CODING (A)	IMPROVEMENT = B/A	% COMPRESSED = A/B * 100
FC1	84780	19667	4.31x	23.20%
FC2	10412	3022	3.45x	29.02%
FC3	668	404	1.65x	60.48%
TOTAL	95860	12186	3.94x	25.37%

Table 3: Compression results for LeNet after Huffman coding

4.2 A comparative evaluation of SqueezeNet vs AlexNet vs VGG16

We now study the performance of SqueezeNet on ImageNet and compare it with AlexNet and VGG architectures. We calculate the total number of parameters in each layer(Fire Module in case of SqueezeNet) and have the following findings which help us to study the advantages of SqueezeNet.

1) Fully Connected Layers Take up most of the memory in the network: Almost 90-95% of the parameters are those of the fully connected layers. More parameters mean more storage memory. SqueezeNet having no FC in the classifier does not face this issue of high memory being occupied.
2) Fire Modules are compact: As the number of input channels to 3*3 filters were kept low we see in Table 2 that fire modules have significantly less number of parameters traditional convolutional layers and are therefore compact.
3) Impressive accuracy of SqueezeNet : The SqueezeNet model gives an accuracy of 58.10% which is more than a little more than AlexNet with 49x lesser parameters. It has 111x lesser parameters than the VGG model but VGG has a better accuracy than SqueezeNet. Given the model size, SqueezeNet shows a very good performance on image classification tasks.

Type of Layer	AlexNet		%age Parameters	VGG16		%age Parameters	SqueezeNet		%age Parameters
Convolution /Fire	Conv1	23296	4.04%	Conv1	1792	10.64%	Conv1	14208	100%
	Conv2	307392		Conv2	36928		Fire2	11920	
	Conv3	663936		Conv3	73856		Fire3	12432	
	Conv4	884992		Conv4	147584		Fire4	45344	
	Conv5	590080		Conv5	295168		Fire5	49440	
				Conv6	590080		Fire6	104880	
				Conv7	590080		Fire7	111024	
				Conv8	1180160		Fire8	188992	
				Conv9	2359808		Fire9	197184	
				Conv10	2359808		Conv10	513000	
				Conv11	2359808				
				Conv12	2359808				
				Conv13	2359808				
Fully Connected	FC1	37752832	95.96%	FC1	102764544	89.36%	None		0%
	FC2	16781312		FC2	16781312				
	FC3	4097000		FC3	4097000				
Total No. of Parameters	61100840		100%	138357544		100%	1248424		100%
ImageNet Top 1 Accuracy	56.55%			71.59%			58.10%		
ImageNet Top 5 Accuracy	79.09%			90.38%			80.42%		

Table 4: Number of parameters and accuracies of SqueezeNet as compared to AlexNet and VGG16.

5 Conclusion and Future Works

We conclude by saying that both of these directions of study work well in reducing high amounts of inference time as well as millions and millions of parameters in a network. As they do not hurt the accuracy much, both these studies are applied in real-life phenomena and being deployed on mobile devices. For example, Han. et al. applied Deep compression on SqueezeNet to achieve AlexNet level accuracy in less than 0.5MB.
Dense-Sparse-Dense and similar research is motivated by the first direction of study. Similarly architectural design strategies are used in InceptionNet, GoogleNet to keep the model size as low as possible. Compression is an active research topic and these two directions explained above are the building blocks of many current/further research.
Also, a possible direction to explore is that to networks that are heavily compressed by the above techniques, recent learning-to-learn strategies, which require high amounts of training computation can be applied to them to make them smarter.

References

A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," in NIPS, 2012.

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

S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in Proceedings of the 28th International Conference on Neural Information Processing Systems, ser. NIPS'15, 2015.

S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," International Conference on Learning Representations (ICLR), 2016.

F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5mb model size," arXiv:1602.07360, 2016.

Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, "Compressing deep convolutional networks using vector quantization," CoRR, vol. abs/1412.6115, 2014.

Y. W. Q. H. Jiaxiang Wu, Cong Leng and J. Cheng, "Quantized convolutional neural networks for mobile devices," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011, 2011.

S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ser. ICML'15, 2015, pp. 1737–1746.

Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus.

Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In NIPS, pages 1269–1277, 2014.

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

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

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.

M. Andrychowicz, M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," in Neural Information Processing Systems (NIPS), 2016.

D. Ha, A. Dai, and Q. Le, "Hypernetworks," in International Conference on Learning Representations 2016, 2016.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.

Mark Horowitz. Energy table for 45nm process, Stanford VLSI wiki.