An Analysis of Compression Methods for Deep learning networks

CS 259: Learning Machines
Parth Shettiwar[†], Madhav Sankar Krishnakumar[†]
[†] University of California, Los Angeles
parthshettiwar@g.ucla.edu,
madhavsankar@g.ucla.edu

I. Abstract

The superior performance of DL models have provided inspiration to integrate them with IoT devices removing the human intervention altogether in various tasks. The models usually leave huge memory footprint leading to their inability of getting deployed on mobile or IoT devices. Various approaches have come up in recent past to compress the Deep learning models without much loss in accuracy. Through this work, we aim to quantify the performance of various network compression algorithms by their application on literature Deep learning models like ResNet on the basis of metrics like accuracy, inference time and model size. We have also presented the analysis and generation strategies for these best model by using the CIFAR 10 dataset. We then move on to analyze the combined performance of the various models and effect on ensembling. We finally summarize the results in the form of Takeaway messages and explore the reason behind the inference boost through the core kernel analysis on the Titan V GPU. Standalone compression methods were able to produce upto 40x reduction in model size with under 5% accuracy loss. Ensemble methods improved the accuracy by a further 2.5% for the same combined model size. Combining the compression methods appropriately generated a 215x decrease in model size with just an addition 1% loss in accuracy. With the huge decrease in model size, we observe upto 8x reduction in inference latency.

II. Introduction

The contradiction between the tremendous resource requirement of new deep learning technology and limited resource of hardware devices is hindering the application of deep learning technology. In order to enable deep learning applications deployed on these devices, various neural network compression strategies have been proposed. Much of the recent research on deep convolutional neural networks (CNNs) has focused on increasing accuracy on computer vision datasets. For a given accuracy level, there typically exist multiple CNN architectures that achieve that accuracy level. Given equivalent accuracy, a CNN architecture with fewer parameters has several advantages:

- **More efficient distributed training**: For distributed data-parallel training, communication overhead is directly proportional to the number of parameters in the model. In short, small models train faster due to requiring less communication.
- **Less overhead when exporting new models to clients**: Transfer of new models or updated models from server to client time to time is done. If models are big, then lots of overhead will be there in doing this job
- **Feasible FPGA and embedded deployment**: FPGAs often have less than 10MB of onchip memory and no off-chip memory or storage. For inference, a sufficiently small model could be stored directly on the FPGA instead of being bottlenecked by memory bandwidth
- Other advantages include lesser latency time, generalizibility of model giving less, variance and lesser compute and memory requirements reducing hardware resource requirements

Keeping this in mind, it becomes utmost importance to analyse the literature compression techniques, so that an user can leverage them based on his needs when time comes. Though in past, there have been survey papers like [1], [2], there has been no work yet to quantify the performance of various compression models uniformly, i.e. across same DL model, same dataset and training on same device. Overall, our contributions through this work are:

- We present a small survey like analysis, in form of takeaway messages, of existing in literature Compression methods for deep learning networks
- Based on evaluation metrics, we perform an uniform basis comparison of these methods (1 from each section, discussed before) quantitatively to understand their applicability, use cases and various pros and cons depending on situation.
- A design of a newer compression techniques, through ensembling and combination of the algorithms, to achieve better performance as compared to individual algorithms on CIFAR10 dataset.

III. Related Work

Current state-of-the-art Convolutional Neural Networks (CNNs) are not well suited for use on mobile devices due to their high memory and energy requirements. Since the start of the imagenet challenge [3], modern CNNs like Resnet [4] have primarily been focussing on accuracies. Thus the neural network architectures have evolved without any regard to model complexity and computational efficiency as shown in the figure 1. On the other hand, successful deployment of CNNs on mobile platforms such as smartphones, AR/VR devices (HoloLens), and drones require much smaller models to run within the limited on-device memory, and low latency to maintain user engagement. This has led to a new field of research that focuses on reducing the model size and inference time of CNNs with minimal accuracy losses.

Approaches roughly fall into two categories. The first category, exemplified by MobileNet [5] and SqueezeNet [6], designs new network architectures that exploit computation and memory efficient operations. The second category focusses on compressing existing model to bring down their memory and energy requirements and reduce latency. Various compression methods are applied to address this [2].

Quantization: Quantization in the DNN compression technique reduces the number of bits required for storing the weights and activations [1]. While the problems of numerical representation and quantization are as old as digital computing, Neural Nets offer unique opportunities for improvement. While we focus on quantization from inference standpoint, we should emphasize that it is equally an important success in CNN training [7]. We will be focussing on inference latencies. We can apply quantization post training and this static method is the easiest for deployment. A lot of study has taken place in this field to decide on the zero point and scales post training [8]. Training Aware Quantization is the more studies of the two types. Though most of the work in the field has been focussed on quantizing the weights [9], there has been recent advancements in quantizing the activations as well [10].

Pruning: Mainly done by removing unnecessary filters, neurons and layers to reduce the size of model. Two types: Structured (Removal of layers and filters in groups ensuring dense matrix operations)[[11], [12],[13], [14]] and Unstructured (Randomly removing layers and weights resulting in sparse matrix operations, but aggressive pruning can be done)[[15], [16],[17]], [18], [19]]. Training and inference with high levels of pruning/sparsity, while maintaining state-of-the-art performance, has remained an open problem

Knowledge Distillation: Model distillation [[20], [21], [22], [23], [24], [25], [26] involves large model and then using it as a teacher to train a more compact model. Instead of using "hard" class labels during the training of the student model, the key idea of model distillation is to leverage the "soft" probabilities produced by the teacher, as these probabilities can contain more information about the input.

Neural network architecture and hardware changes: This is more of recent line of work where the neural network architecture is developed for a particular target hardware platform [[27],[28],[29]]

These methods can also be combined to various degrees to generate more effective compression techniques. Pruning and quantization have been combined in various works [30]. But more recently there has been work to combine pruning and knowledge distillation [31]. The idea to combine Pruning,

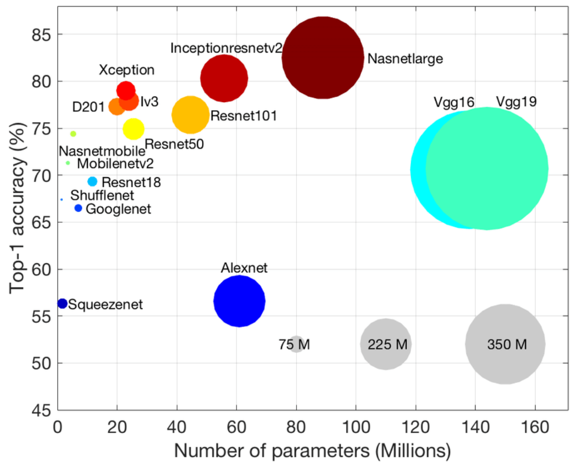


Fig. 1: Increasing model size with increase in accuracy.

Quantization and Knowledge Distillation was arrived at about just a year ago and has been called the PQK technique [32]. Also, ensembling multiple models has found to produce increased accuracy for various models over the year [33]. More modern methods using Reinforcement learning to attain compression has also been introduced [34], [35].

IV. Problem Statement and Setup

We explore 3 prominent methods in the field of compression:

- Quantization: We implemented Static Quantization and Quantization Aware Training models using PyTorch libraries. Core idea of is that the technique reduces the number of bits required for storing the weights and activations from FP32 to INT8 [1].
- Pruning: We analyse [19], which is one of the first pruning techniques introduced. The major reason we chose this model is its versatility and intuitive reasoning behind its working, while at same an easy to use set of hyperparameters to work with.
- Knowledge Distillation: We analyse the performance of [24], which was the first work in this domain. Though there have been various works in this field, this original KD model introduces concept of student-teacher and is known to give excellent results with its devised loss function.

We use the same dataset for all methods to keep the uniformity: CIFAR10 and same base model: Resnet18. In the end, our goal would be to analyse these methods and note some takeaway messages. The performance metrics we consider are: **Model size**, **number of parameters**, **accuracy** and **Inference latency**. We also develop custom models to further better the performance using following techniques:

- Ensembling for Knowledge distillation
- Combining the 3 compression techniques in serial fashion

During the takeaway messages, we also put the hardware oriented implications of pruning and quantization to get better insights.

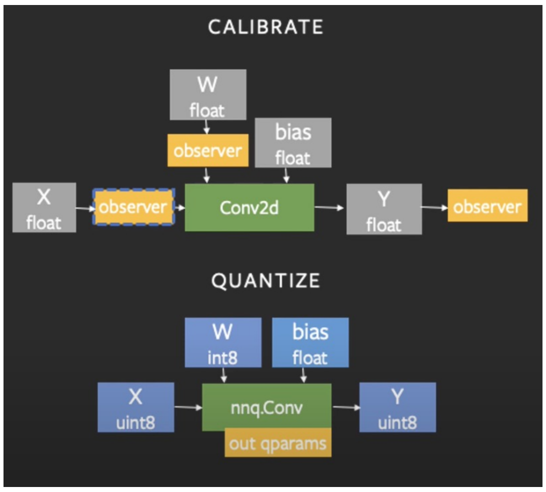


Fig. 2: Static Quantization.

V. Approach

A. Quantization

Quantization in the generic form, refers to techniques for performing computations and storing intermediate tensors at lower bitwidths than floating point precision. We study the more generally used method of converting the floating point operations into integer arithmetic. This helps to compress the model size and help to accelerate the inference time due to the high performance of vectorized intger operations on various hardware. The main takeaway of this solution is the improvement in inference time. We study two different types of Quantization techniques.

1) Static Quantization: Static quantization quantizes both the weights and activations of the model. It fuses the activations into preceding layers wherever possible. The scales and zero points for each of the values should be decided. They are determined prior to inference using a representative dataset, so that the inference time is faster. Therefore, static quantization is theoretically faster than a dynamic quantization where these parameters are determined in runtime.

The steps followed are:

- Train a base resnet18 model.
- Apply layer fusion to the conv, relu and batchnorm layers.
- Apply torch.quantization.QuantStub() to the inputs and outputs.
- Run post-training calibration.
- Convert the calibrated floating point model to quantized integer model.
- Save the quantized integer model.

2) Quantization Aware Training: Static Quantization loses some accuracy during the post training calibration. The scales and zero points may not be accurate. In such cases, we need to retrain the model in a way that it is aware of the quantization that will occur post training. Quantization aware training is capable of modeling the quantization effect during training.

The mechanism of quantization aware training is simple, it places fake quantization modules, i.e., quantization and dequantization modules, at the places where quantization happens. This simulates the quantization and monitors scales and zero points of the weights and activations. Once the quantization aware training is finished, the floating point model could be converted to quantized integer model immediately using the information stored in the fake quantization modules.

The steps followed are:

- Train a base resnet18 model.
- Apply layer fusion to the conv, relu and batchnorm layers.
- Apply torch.quantization.QuantStub() to the inputs and outputs.
- Prepare quantization model for quantization aware training.
- Move the model to CUDA and run quantization aware training using CUDA.
- Move the model to CPU and convert the quantization aware trained floating point model to quantized integer model.
- Save the quantized integer model.

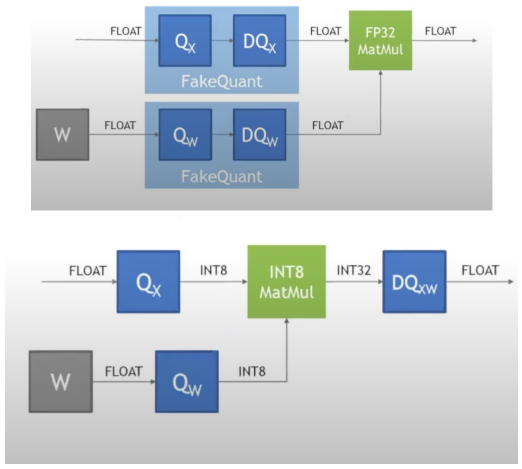


Fig. 3: Quantization Aware Training.

Model	Accuracy	Model Size (MB)	Latency (ms)
Baseline	86.02	84.6	18.07
Static	78.8	24.3	12.09
QAT	84.4	24.13	10.96

From the table we can observe that quantization reduces the Model Size to one-forth its initial size. This is in line with what is expected as converting floating point numbers to integers should reduce size by one-forth. More importantly we see a 2x improvement in the inference latency. In case of Quantization Aware Training, we observe that we obtain the 4x save in model size and 2x improvement in latency without much loss in accuracy. But this comes at the cost of extra training time and data. Static Quantization requires only a subset of data for calibration while QAT needs to retrain the entire model. In our case study, we target the problem of inference and hence QAT is a better solution.

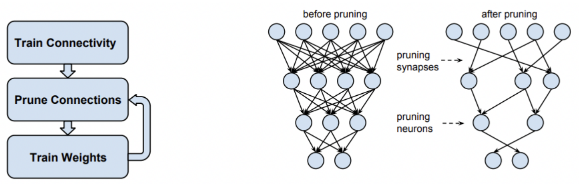


Fig. 4: Magnitude based Pruning

B. Pruning

As part of this section, we explore [19], based on magnitude based pruning. The concept of pruning arises from the idea where we prune the weights of a neural network. After each training, the link with small weights are removed. Thus the salience of a link is just the absolute size of its weight. The following steps are followed:

- Train a base resnet18 model
- Prune the low-weight connections: All connections for a particular layer are first sorted by absolute value of weights and then top K connections are chosen for next iteration, where K is percentile based on the prune ratio, converting a dense network into a sparse network.
- Now we retrain the network to learn the final weights for the remaining sparse connections. Point to note is that, if the pruned network is used without retraining, accuracy is significantly impacted.
- Go to step 2, to further prune the network iteratively until desired compression ratio is obtained

4 shows this algorithm. Another key point to note is that, if the prune ratio is huge, then directly removing those many weights is not desirable and can affect the training of pruned network and this has to be done in steps (mentioned in step 4 of above algorithm). Hence we first start with a small prune ratio and successively increase in steps, each time retraining. This idea is called iterative pruning. In addition to decreasing model size, pruning also helps to decrease model variance, and helps avoiding overfitting of complex models on datasets. Furthermore, with decreased parameters, and model size, inference latency also decreases, making this strategy a great candidate for deploying on IoT devices.

In terms of implementation, the model creates a mask for every neural network parameter in the model, which is later trained on, since the existing PyTorch library doesn't support sparse dictionary savings and loading. This leads to increased space overhead, in saving the masks.

C. Knowledge distillation

As part of this section, we have explored the original knowledge distillation idea proposed by [24]. The goal of knowledge distillation, as name suggests, is to distill the knowledge from a complicated network to a much smaller network in size. The intuition is that once the complicated network has been trained, also called teacher model, the student model (smaller, simpler model) will can learn effective embeddings from the teacher model with much higher learning rate. Effectively Knowledge is transferred from the teacher model to the student by minimizing a loss function in which the target is the distribution of class probabilities predicted by the teacher model. However, in many cases, this probability distribution has the correct class at a very high probability, with all other class probabilities very close to 0. As such, it doesn't provide much information beyond the ground truth labels already provided in the dataset. To tackle this issue, softmax temperature (T) is introduced. The probability p_i of class i is calculated from the logits z as:

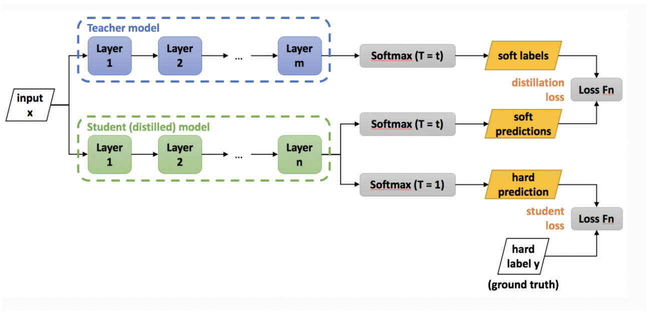


Fig. 5: Knowledge distillation general framework

$$p_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)}$$

The intuition here is that when the soft targets have high entropy, they provide much more information per training case than hard targets and much less variance in the gradient between training cases, so the small model can often be trained on much less data than the original cumbersome model while using a much higher learning rate. Effectively, the loss function, L_{KD} for training of student is formulated as follows:

$$L_{KD}\left(W_{\text{student}}\right) = \alpha A + (1-\alpha) B$$

where

$$A = T^2 * \text{CrossEntropy}\left(Q_S^\tau, Q_T^\tau\right)$$

$$B = \text{CrossEntropy}\left(Q_S, y_{\text{true}}\right)$$

Here: Q_S^τ are the softmax temperature labels from student, Q_S are the hard labels prediction from student, y_{true} are the true groundtruth labels Q_T^τ are the softmax temperature labels from teacher, α is just a hyperparameter to weight the two cross entropy losses and T is the softmax temperature. With this, the loss is framed in a way that model will get information from both teacher (through A) and from the groundtruth (through B). Figure 5 describes the general knowledge distillation framework. The steps followed are:

- Train a base resnet18 model.
- Design a simpler, smaller student model
- Train the student model using the teacher pretrained resent18 model on L_{KD} loss

- While inference on student model, keep temperature $T = 1$, that is predict hard labels

As a result, knowledge distillation leads to considerable decrease in model size as the student model has much simpler architecture with very less parameters, without compromising on accuracy much. Furthermore, inference latency also decreases as a result, leading to easier integration and deployment on IoT devices, with much lesser memory footprint. Like pruning, this method also leads to a simpler model, leading to decreasing model variance and avoid the problem of over-fitting prevalent nowadays in deep learning networks.

VI. Methodology (*Intuition behind our goal*)

As part of the evaluation framework, we chose the Resnet18 model, and has in past created benchmark for image classification datasets. We further chose CIFAR10 dataset as representative of Imagenet dataset, since Imagenet is huge in size and not feasible to conduct any experiments with the current hardware resources and time constraints. The compression techniques of knowledge distillation, pruning and quantization were chosen since a good amount of work has been done in these fields and great potential lies ahead of them for further research, where on other hand architectural changes like using reinforcement learning for same is an active area of research, hence were not explored. The specific models which were analysed were chosen based on the ease of conducting experiments and analysing their performance and set a great benchmark for any improvements to be done on top of them. The model parameters of model size, inference latency, accuracy and number of parameters were chosen as evaluation metrics since the goal of compression methods is to reduce model size and parameters while not compromising accuracy. These parameters play a crucial role while working in industry, where strict SLA requirements are enforced for evaluation purposes. All of these metrics are correlated and accuracy has a tradeoff with other parameters.

VII. Evaluation

1) Takeaway Message 1: Model Design for Knowledge Distillation: In case of Knowledge Distillation, we need to come up with the student model, which is basically a double edged sword. We can customize models and also learn very different models. But this also means, it is not the easiest of the compression techniques. So we take into consideration a few simple strategies that can help produce the most compressed model. We first take a simple baseline student model with 4 convolution layers followed by 2 FC layers. We can then apply these strategies to arrive at simpler models.

- *Strategy 1: Reduce the Filter Size:*
  We can reduce the filter size to 3 or 1 and this reduces the parameter count by a huge value. Also, if we stick to a single convolution filter layer, there are existing accelerators customized for filter sizes like 3 and we can make use of them. From our experiments, we observed that the filter size of 3 works best. Filter size of 1 compresses the model but loses out on accuracy.
- *Strategy 2: Reduce the number of channels:*
  We can bring down the number of channels to reduce the parameters and hence the model size.
- *Strategy 3: Replace Fully Connected Layers with more layers of Convolutions:*
  As we know the FC layer accounts for a large proportion of the parameters. Therefore, replacing an FC layer with layers of convolution can bring down the model size. But this doesn't guarantee an improvement in the inference time.

Model	Accuracy	Size(MB)	Params (K)	Latency(ms)
Resnet18	90.3	44.7	11170	0.856
Baseline	85.495	7.5	654	0.173
Model 1	85.2	4.6	399	0.134
Model 2	80.9	3.8	376	0.161
Model 3	83.5	3.2	279	0.198
Model 4	85.1	2.9	253	0.157
Model 5	84	2.2	187	0.164
Model 6	84.5	2.1	191	0.177

- Baseline Model: 4 convolution layers followed by 2 FC layers.
- Model 1: One FC layer removed from Baseline, resulting in 4 Convolutions followed by a single FC layer.
- Model 2: Baseline Model + All filter sizes set to 1 with padding 0 to ensure the input and output dimensions are same.
- Model 3: 6 Convolution Layer followed by a single FC layer.
- Model 4: Baseline Model + Reduced the number of channels in each convolution layer.
- Model 5: 5 Convolution Layer followed by a single FC layer.
- Model 6: Combine Models 2 and 3.

As we can see strategies 2 and 3 produce great results. But in case of strategy 1, 3*3 filters seem to be a better fit than the 1*1 filters.

2) Takeaway Message 2:Model inspection of pruning at different compression rates: As part of this section, we tried to analyse the pruning performance with varying compression rates. Pruning, as mentioned before, is done in successive trainings. Here we have started with Base resnet18 model and in every iteration pruned by 15% of the current model. We have shown results for few of the intermediate compression ratios, in following table. As mentioned before, the model size is calculated by saving all the non-zero weights of the resulting model (This is done by first multiplying the mask for each layer with the weight).

Model	Accuracy (%)	Parameters (M)	Model size (MB)
Base Resnet18	90.3	11.17	44.7
Prune 15%	91.66	9.52	38.10
Prune 48%	92.03	5.92	23.69
Prune 77%	92.2	2.7	10.92
Prune 83%	92.03	2.02	8.1
Prune 90%	90.75	1.3	5.26
Prune 93%	88.87	0.87	3.51
Prune 96%	84.84	0.6	2.44
Prune 97%	79.19	0.5	1.97
Prune 98%	68.44	0.4	1.63

TABLE I: Performance of pruning with various compression rates

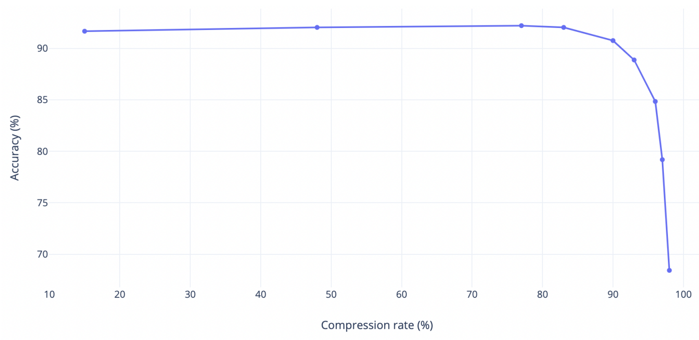


Fig. 6: Accuracy vs Compression rate plot for magnitude based pruning

We first observe that accuracy increases a little after pruning. The main reasons behind this is to the retraining of model being done after pruning some weights in every iteration. Hence pruning some few weights is always good for model generalizabilty and avoid overfitting. This accuracy is or more or less constant for upto 83% of pruning! The main reason for this can be to the fact that CIFAR10 is a small dataset and Resnet18 is a huge model which can easily overfit over this small dataset. As a result, there are lot of redundant weights in the model and hence can be removed. After this constancy, there is an exponential decrease in model accuracy. We can call this a ”kink”. We speculate an exponential decrease is to the fact that after some point, the most important weights start getting pruned, which contribute most to the model output and hence accuracy gets severely affected. On an overall basis, we got the model size down to 5.26 MB from 44.76MB base model without decreasing in accuracy at all!

3) Takeaway Message 3: Model accuracy not compromised much in all 3 models: We can observe that though the models are compressed to as much as 40x in some cases, the accuracy is still fairly high. This goes on to explain how all the three methods of compression are effective and the reason why they have been used widely.

Model	Compression	Accuracy
Quantization (QAT)	3.5x	84.4
Pruning (96%)	34.6x	84.84
KD (Model 6)	40.2x	84.5

4) Takeaway Message 4: Knowledge distillation decreases the model size by highest margin: Based on accuracy to compression ratio we can see that Knowledge Distillation has an edge as it still manages to produce better results than Quantization (QAT) even though it compresses the model more. This goes on to show that the model is too large for the problem at hand and a smaller model (40x smaller) is able to learn the features and avoid overfitting. This could be because of the CIFAR 10 data being used. As there are only 10 classes, even a resnet18 model is too large. This also makes sense as resnet was originally intended to tackle the ImageNet problem with over 20,000 categories. Pruning also does well but KD comes with the added advantage that we can learn any different student model from a teacher model. Therefore, for our problem statement, Knowledge Distillation seems to provide the smallest model sizes for a reasonably good accuracy.

5) Takeaway Message 5: Number of parameters variation across models: We know that Quantization doesn’t affect the parameters. But the number of parameters can be seen as an important metric to the size of the model and also as a measure of computations needed. As we can see pruning removes about 95% of the parameters but it is still much larger than the number of parameters in Knowledge Distillation. The much smaller models of KD seems to produce the least parameters.

Model	Accuracy	Number of params
Quantization (QAT)	84.4	11M
Pruning (96%)	84.84	0.6M
KD (Model 6)	84.5	0.19M

6) Takeaway Message 6: Inference latency: In the above analysis, we consider the number of parameters and model size as measure of the the computational cost. But, it is more important to verify that these parameter reduction actually translates into speedup on hardware. We see that the inference latency speeds up 5x for pruning and knowledge distillation. This is because of the much smaller models that are generated by these compression techniques.

Model	Inference latency
Base Resnet18	1x
QAT	0.6x
Knowledge distillation (Model 6)	0.2x
Pruning (70% compression)	0.21x

7) Takeaway Message 7: Model size decrement happens in different fashion across algorithms: For quantization, we can clearly observe a 4x decrement in model size always. At same time knowlegde distillation allows user to vary the model size by defining the model itself. Pruning also allows user to vary the model size by defining the compression rate. An important point to note here is that, model size decrement has a discrete space in knowledge distillation (defined by student model design, or number of layers) while the decrement can be done in a continuous space in pruning (defined by float compression rate). This is huge difference between the 2 models, which has practical industrial implications where an user in real world, if wants to optimize the accuracy for a fixed model size, can easily go for pruning, as he just has to calculate compression rate in that case, but would have a hard time in designing the student model to get close to the maximum model size allowed. An important point to note here is that, for knowledge distillation, a cleverly designed model might be able to get 20x-40x decrement in model size, but at same time, as we saw in case of pruning, a "kink" in model accuracy happens after some threshold pruning compression rate, limiting the maximum model size decrement it can get.

8) Takeaway Message 8a: Retraining requirement: Most of the compression techniques we came across requires retraining. Pruning, Knowledge Distillation and QAT requires the models to be retrained. This is where Static Quantization excels. If we do not have enough dataset or the hardware resources to train, we could just load a pretrained model and use static quantization. We just need a very small amount of data to compute the scales and zero points. These are low compute intensive processes unlike training and can be done with a small subset of data. Furthermore, pruning requires retraining many times in steps until we get the desired pruning ratio (we cant simply prune 90% of the weights and retrain, this would lead to catastrophic loss of accuracy). Hence knowledge distillation excels here the most. If user has limited resources for training, then he can go for Knowledge distillation approaches which get an easy 20x decrease in model size after training for only once.

9) Takeaway Message 8b: User effort in performing compression is least in quantization and pruning : As we observe, for knowledge distillation, an user has to design a complete student model, which is a big hurdle in the accessibility of this model to a general user in industry who has little or no idea of how knowledge distillation works. On other hand, an user can easily compute an optimal compression rate he needs to meet his specifications. At same time, an user has to do nothing, if he just performs quantization. The following table II summarises the discussion made so far in last 2 takeaway messages.

	Quantization	Knowledge distillation	Pruning
Need retraining?	Yes, once	Yes, once	Yes, multiple times
Model size decrement	Discrete and fixed (4x)	Discrete	Continuous
User involvement	Nothing	Design student model	Calculate compression rate

TABLE II: Comparison of compression techniques

10) Takeaway Message 9: Hardware Implication of Quantization: We can understand the Quantization compression method from a matrix multiplication standpoint to better understand the practical results we saw earlier. For this purpose, we wrote a CUDA matrix multiplication code and ran it to compute floating point multiplications and integer multiplications.

We observe that the execution time for a simple matrix multiplication (4096 * 25088 and 25088 * 16), which takes about 170ms using floating point numbers, taken only 125ms with integer. This drop is very intuitive to understand as integer takes 8 bits to store a number as opposed to 32 bits of floating point numbers. By convention, we might expect each kernel operation to be 4x faster. But, in practice, the speed of each calculation very much depends on the actual hardware. For instance, a modern CPU in a desktop machine does float arithmetic as fast as integer arithmetic. On the other hand, GPUs are more optimized towards single precision float calculations. So this explains why a baseline INT kernel is not 4x faster but is still faster. We might have to optimize the kernel further to observe close to 4x advantage we saw in the model inference time.

11) Takeaway Message 10: Hardware Implication of Pruning - Sparse Matrices: We know that Pruning removes the less important weights and sets them to 0, giving rise to a sparse matrix. We saw the reduction in model size due to this. Now let us try to estimate the inference time gains obtained through pruning. The core of this gain is due to the difference between matrix multiplication and sparse matrix multiplication. So we implemented a sparse matrix kernel in CUDA and tried to understand the time it takes through nvprof. We represented the sparse matrix in the CSR-Scalar sparse matrix format (as shown in the figure).

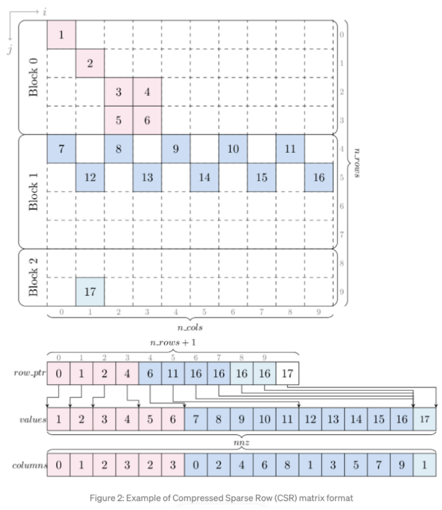


Fig. 7: CSR Sparse Matrix

We performed the matrix vector multiplication as this is an estimate of the inference time. We observed the following results. We observe that the CUDA memcpy HtoD dropped by around 50% due to size difference. We have used a 25088

* 25088 matrix. So it has around 629M values. The sparse matrix representation has around 63M values. So it takes approximates 63M * 2 to store the values and corresponding column indices. The row pointers are at max of length 25088. So the total input size is just 126M and this explains the huge difference in memcpy.

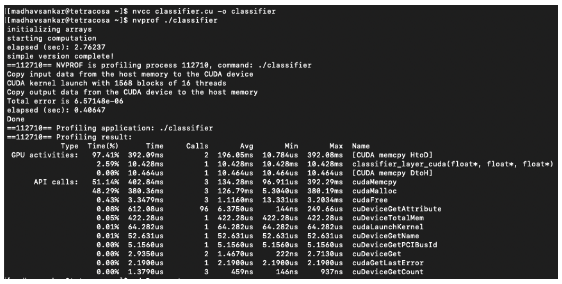
Similarly the core kernel improves 3x from 10ms to 3ms and this can be explained due to the reduction in the number of FLOPs. Sparse matrix multiplication incur a lot less floating point operations than dense one.

Kernel	Input Size	CUDA memcpy HtoD (ms)	Multiplication (ms)
Dense	629M	392.09	10.428
Sparse	63M	220.96	3.5484

We ran similar experiments on matrix matrix multiplication (Square matrices of 1000 * 1000 size). We used the optimized dense matrix multiplication code from MiniProject and CuSparse code for sparse matrix multiplication. We observed a 3x speedup with 15% sparsity.

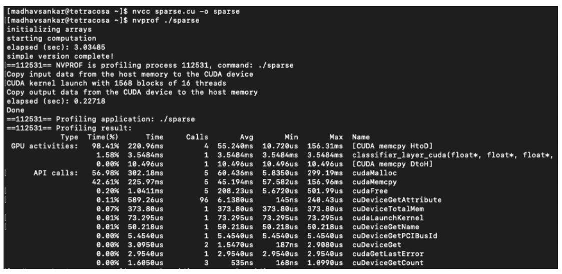
Kernel	Compression Ratio	Execution Time (ms)
Dense	-	162.75
Sparse	15%	55.375
Sparse	30%	35.548
Sparse	90%	4.364

We can see a drastic reduction in a single Sparse matrix operation. This when applied over all the weights across the model can help bring down the inference latency by a huge margin. This helps us understand how pruning could help bring the latency time.



```
[madhavsankar@tetracosa ~]$ nvcc classifier.cu -o classifier
[madhavsankar@tetracosa ~]$ nvprof ./classifier
initializing arrays
starting computation
elapsed (sec): 2.76237
simple version complete!
==112710== NVPROF is profiling process 112710, command: ./classifier
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 1568 blocks of 16 threads
Copy output data from the CUDA device to the host memory
Total error is 6.57148e-06
elapsed (sec): 0.40647
Done
==112710== Profiling application: ./classifier
==112710== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   97.41%  392.09ms         2  196.05ms  10.784us  392.08ms  [CUDA memcpy HtoD]
                    2.59%  10.428ms         1  10.428ms  10.428ms  10.428ms  classifier_layer_cuda(float*, float*, float*)
                    0.00%  10.464us         1  10.464us  10.464us  10.464us  [CUDA memcpy DtoH]
      API calls:   51.14%  402.84ms         3  134.28ms  96.911us  392.29ms  cudaMemcpy
                   48.29%  380.36ms         3  126.79ms  5.3040us  380.19ms  cudaMalloc
                    0.43%  3.3479ms         3  1.1160ms  13.331us  3.2034ms  cudaFree
                    0.08%  612.08us        96  6.3750us     144ns  249.66us  cuDeviceGetAttribute
                    0.05%  422.28us         1  422.28us  422.28us  422.28us  cuDeviceTotalMem
                    0.01%  64.282us         1  64.282us  64.282us  64.282us  cudaLaunchKernel
                    0.01%  52.631us         1  52.631us  52.631us  52.631us  cuDeviceGetName
                    0.00%  5.1560us         1  5.1560us  5.1560us  5.1560us  cuDeviceGetPCIBusId
                    0.00%  2.9350us         2  1.4670us     222ns  2.7130us  cuDeviceGet
                    0.00%  2.1900us         1  2.1900us  2.1900us  2.1900us  cudaGetLastError
                    0.00%  1.3790us         3     459ns     146ns     937ns  cuDeviceGetCount
```

Fig. 8: Dense Matrix Vector Multiplication



```
[madhavsankar@tetracosa ~]$ nvcc sparse.cu -o sparse
[madhavsankar@tetracosa ~]$ nvprof ./sparse
initializing arrays
starting computation
elapsed (sec): 3.03485
simple version complete!
==112531== NVPROF is profiling process 112531, command: ./sparse
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 1568 blocks of 16 threads
Copy output data from the CUDA device to the host memory
elapsed (sec): 0.22718
Done
==112531== Profiling application: ./sparse
==112531== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.41%  220.96ms         4  55.240ms  10.720us  156.31ms  [CUDA memcpy HtoD]
                    1.58%  3.5484ms         1  3.5484ms  3.5484ms  3.5484ms  classifier_layer_cuda(float*, float*, float*,
                    0.00%  10.496us         1  10.496us  10.496us  10.496us  [CUDA memcpy DtoH]
      API calls:   56.98%  302.18ms         5  60.436ms  5.8350us  299.19ms  cudaMalloc
                   42.61%  225.97ms         5  45.194ms  57.582us  156.96ms  cudaMemcpy
                    0.20%  1.0411ms         5  208.23us  5.6720us  501.99us  cudaFree
                    0.11%  589.26us        96  6.1380us     145ns  240.43us  cuDeviceGetAttribute
                    0.07%  373.80us         1  373.80us  373.80us  373.80us  cuDeviceTotalMem
                    0.01%  73.295us         1  73.295us  73.295us  73.295us  cudaLaunchKernel
                    0.01%  50.218us         1  50.218us  50.218us  50.218us  cuDeviceGetName
                    0.00%  5.4540us         1  5.4540us  5.4540us  5.4540us  cuDeviceGetPCIBusId
                    0.00%  3.0950us         2  1.5470us     187ns  2.9080us  cuDeviceGet
                    0.00%  2.9540us         1  2.9540us  2.9540us  2.9540us  cudaGetLastError
                    0.00%  1.6050us         3     535ns     168ns  1.0990us  cuDeviceGetCount
```

Fig. 9: Sparse Matrix Vector Multiplication

12) Takeaway Message 11: Ensemble of Knowledge Distillation Models: Instead of one model with 654 parameters, we took 3 separate models that sum up to 631 parameters and ran the classification on them. We then took average of their probabilities. As most of the misclassification occurs between closely related classes, averaging the probabilities can help avoid these small errors. This is why we have taken this method over majority voting across classes. We observe staggering results - a clear 2.5% increase in the accuracy with reduced model size.

Model	Accuracy	Size(MB)	Params (K)
Baseline	85.495	7.5	654
Ensemble (1+2+3)	**88.02**	7.2	631
SubModel 1	85.1	2.9	253
SubModel 2	84	2.2	187
SubModel 3	84.5	2.1	191

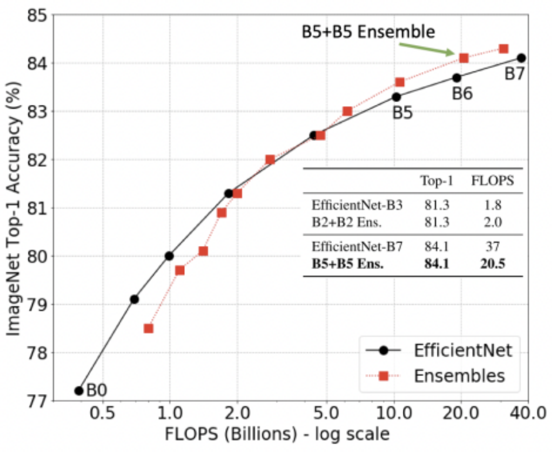
Ensemble of models raises some interesting questions. Here we try to summarize some key aspects.

Different models reduce variance of predictions: Model training through stochastic training algorithms leads to high sensitivity to specifics of the training data and the possibility of overfitting to them. This means that training the model in different subsets of the entire model can lead to different models that have a high variance in their prediction. To avoid this, we can train multiple models at the same time and generate a combined output from them. This way each model can learn different aspects of the data and together they can reduce the variability of the model and add consistency to the results. Moreover, the accuracy of the final model can also theoretically be larger than any single model.

Model size: Instead of a single four convolution + FC model with 654K parameters, we replaced it with 3 models with slightly lesser accuracies using the strategies mentioned earlier. These 3 models sum up to 631K parameters. Therefore, we can get the benefits of ensembling like the increased accuracy without having to lose on the memory requirements.

Due to experimental limitations, we were not able to test throughputs and other parameters for such ensemble models, so we present results from a recent research paper exploring the same idea [36].

Efficiency and training speed: It's not surprising that ensembles can increase accuracy, but the disadvantage is that using multiple models in an ensemble may introduce extra computational cost at runtime. So, a more accurate measurement is to compare an ensemble accuracy with that of a single model that has the same computational cost. We compared this in the table above. We observed a 2.5% increase in accuracy for the same model size. We also present a related industrial results published by Google using their CNN models (They use ImageNet unlike our CIFAR). For example, an ensemble of two EfficientNet-B5 models matches the accuracy of a single EfficientNet-B7 model, but does so using 50% fewer FLOPS. This demonstrates that instead of using a large model, in this situation, one should use an ensemble of multiple considerably smaller models, which will reduce computation requirements while maintaining accuracy. Results on the training days of the models have also shown great advantage with ensembling. For the above example, the two B5 models combined took 96 TPU days total while one B7 model took 160 TPU days. In practice, we know that model ensemble training can be parallelized using multiple accelerators leading to even further reductions.



	Top-1	FLOPS
EfficientNet-B3	81.3	1.8
B2+B2 Ens.	81.3	2.0
EfficientNet-B7	84.1	37
B5+B5 Ens.	**84.1**	**20.5**

Fig. 10: Ensembling Accuracy

Throughput: Combining multiple models together can help improve throughput. Even with the resnet scenario, there are operators like the fully connected layers that are compute intensive and the skip-connections are memory intensive. Co-running multiple smaller models with efficient scheduling strategies can give a high boost in throughput. The following are the throughput comparison between the above said B7 model and the ensemble of two B5 models adjusted to the same accuracy.

Model	Accuracy	Throughput	SpeedUp
Baseline	82	192	-
Ensemble	82.3	409	2.1x
Baseline	85	54	-
Ensemble	85.2	125	2.3x

Inference Latency: In the above analysis, we consider the number of parameters and FLOPS as measure of the the computational cost. But, it is more important to verify that these parameter reduction obtained by ensembling actually translates into speedup on hardware. The following results present the reduction in the average online latency on TPUv3 of up to 1.7x for ensembles of models from the EfficientNet family compared to single models with comparable accuracy. The B4 model has 16.6ms latency while the ensemble completes in 9.6ms, giving 1.7x speedup.

13) Takeaway Message 12: Combination of Compression Techniques: As part of this analysis, we experimented with the combination of the compression techniques. Specifically, we considered all possible serializations of KD, pruning and quantization. The pipeline was always in the order: Knowledge distillation, always preceding pruning and quantization if it exists and pruning preceding quantization if it exists. The major reason for following this order is: knowledge distillation first helps to distill a very complex network to a smaller student network, hence it makes sense to apply it at start only, rather than on top of pruned or quantized teacher model (would lead to inefficient learning). Similarly, pruning is applied before quantization, since if we do reverse, then after applying quantization, pruning would lead to greater loss in weights and information learnt by model (would be in discrete steps as opposed to efficient learning by pruning on float values)

For the experiment, we chose the following configurations:

- Knowledge distillation: Used a 3 layer convolution network followed by 2 fully connected layers
- Pruning: Done at 77% compression rate
- Quantization: The QAT training is done which has been shown to be effective in performance over static quantization

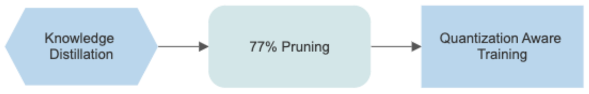


Fig. 11: Flowchart of Combined Compression Model

The following table summarizes the result: Before writing

	Model Accuracy	Model size (MB)	Parameters (K)
Quantization (QAT)	84.4	24.13	11170
Knowledge Distillation (KD)	84.47	1.43	357.5
Pruning 77%	92.2	10.92[1]	2700
KD $\rightarrow$ Pruning 77%	80.8	0.34[1]	83.8
KD$\rightarrow$Quantization	84.1	0.38	357.5
Pruning 77%$\rightarrow$Quantization	92.8	11.32	2700
KD$\rightarrow$Pruning 77%$\rightarrow$Quantization	**82.9**	**0.38**	**83.8**

TABLE III: Performance of combination of compression approaches.

any comments on the table, we would like to highlight the model size reported for pruning has been based on only non-zero weights in the model and saving them to a different file (marked as [1] in the table)
An interesting observation is that quantization on top of any previous models has lead to both improved accuracy and decreased model size as compared to previous chain of models. For example, KD$\rightarrow$Pruning 77%$\rightarrow$Quantization has better accuracy and model size as compared to KD$\rightarrow$Pruning 77%. A major reason for this is the training process applied in QAT which helps to further better the performance. We also observe that, the combination of all three models in a sequential fashion, still keeps an accuracy of 82.9% and gets a tremendous decrease in model size to 0.38 MB as compared

[1].

to 90-93% accuracy base resnet18 model with huge size of about 80MB. This is about 200x decrease in model size and still keeping the model performance in line.

VIII. Conclusion and future work

In this work, we analysed the various compression techniques on performance metrics and understood the pros and cons of each approach. We took a few takeaway messages based on model design, how performance metrics vary across different techniques, understood the hardware implication of pruning and quantization and lastly, improved the performance of models themselves by using an ensembling and combination of algorithms, while keeping in mind the hardware parameters of model size and number of parameters, which determine the compute.

An interesting work to follow after this analysis can be to generate or suggest an optimal model automatically given a persons hardware constraints like memory and compute power. Furthermore, currently we explored how serialization of algorithms perform, which has a demerit of lot of training time required (addition of all three algorithms individual training times). An integration of all three algorithms in one training itself, would be an interesting followup, where KD, pruning and quantization, update the model weights together. This will reduce the training time by great margin.

IX. Statement of Work

A. Madhav Sankar Krishnakumar

- Baseline implementation of Quantization.
- Model Design for Knowledge Distillation using various strategies to balance compression and accuracy.
- Hardware implications of Quantization - CUDA kernels.
- Hardware implications of Pruning - Sparse Matrix multiplication implementation using CSR format and CuSparse.
- Ensemble models implementation and evaluation.
- Hardware implications of ensembling.

B. Parth Shettiwar

- Overall project formulation and ideation
- Baseline implementation of Pruning and Knowledge Distillation.
- Model Inspection for pruning at different compression rates.
- Combination of compression techniques.
- Evaluation of various permutations of compression technique combinations.

References

[1] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *ArXiv*, vol. abs/2103.13630, 2022.

[2] R. Mishra, H. P. Gupta, and T. Dutta, "A survey on deep neural network compression: Challenges, overview, and solutions," *ArXiv*, vol. abs/2010.03954, 2020.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[6] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[7] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," *CoRR*, vol. abs/1805.11046, 2018. [Online]. Available: http://arxiv.org/abs/1805.11046

[8] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry, "ACIQ: analytical clipping for integer quantization of neural networks," *CoRR*, vol. abs/1810.05723, 2018. [Online]. Available: http://arxiv.org/abs/1810.05723

[9] Y. Choukroun, E. Kravchik, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," *CoRR*, vol. abs/1902.06822, 2019. [Online]. Available: http://arxiv.org/abs/1902.06822

[10] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: parameterized clipping activation for quantized neural networks," *CoRR*, vol. abs/1805.06085, 2018. [Online]. Available: http://arxiv.org/abs/1805.06085

[11] S. Lin, R. Ji, Y. Li, Y. Wu, F. Huang, and B. Zhang, "Accelerating convolutional networks via global & dynamic filter pruning," in *IJCAI*, 2018.

[12] J. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *CoRR*, vol. abs/1707.06342, 2017. [Online]. Available: http://arxiv.org/abs/1707.06342

[13] R. Yu, A. Li, C. Chen, J. Lai, V. I. Morariu, X. Han, M. Gao, C. Lin, and L. S. Davis, "NISP: pruning networks using neuron importance score propagation," *CoRR*, vol. abs/1711.05908, 2017. [Online]. Available: http://arxiv.org/abs/1711.05908

[14] S. Yu, Z. Yao, A. Gholami, Z. Dong, M. W. Mahoney, and K. Keutzer, "Hessian-aware pruning and optimal neural implant," *CoRR*, vol. abs/2101.08940, 2021. [Online]. Available: https://arxiv.org/abs/2101.08940

[15] N. Lee, T. Ajanthan, and P. H. S. Torr, "SNIP: single-shot network pruning based on connection sensitivity," *CoRR*, vol. abs/1810.02340, 2018. [Online]. Available: http://arxiv.org/abs/1810.02340

[16] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *NIPS*, 1992.

[17] X. Dong, S. Chen, and S. J. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon," *CoRR*, vol. abs/1705.07565, 2017. [Online]. Available: http://arxiv.org/abs/1705.07565

[18] S. Park*, J. Lee*, S. Mo, and J. Shin, "Lookahead: A far-sighted alternative of magnitude-based pruning," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=ryl3ygHYDB

[19] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015. [Online]. Available: http://arxiv.org/abs/1506.02626

[20] S. Ahn, S. X. Hu, A. C. Damianou, N. D. Lawrence, and Z. Dai, "Variational information distillation for knowledge transfer," *CoRR*, vol. abs/1904.05835, 2019. [Online]. Available: http://arxiv.org/abs/1904.05835

[21] Y. Li, J. Yang, Y. Song, L. Cao, J. Luo, and J. Li, "Learning from noisy labels with distillation," *CoRR*, vol. abs/1703.02391, 2017. [Online]. Available: http://arxiv.org/abs/1703.02391

[22] A. K. Mishra and D. Marr, "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy," *CoRR*, vol. abs/1711.05852, 2017. [Online]. Available: http://arxiv.org/abs/1711.05852

[23] G. E. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *ArXiv*, vol. abs/1503.02531, 2015.

[24] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015. [Online]. Available: https://arxiv.org/abs/1503.02531

[25] S. Yun, J. Park, K. Lee, and J. Shin, “Regularizing class-wise predictions via self-knowledge distillation,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 13 873–13 882.

[26] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” *CoRR*, vol. abs/1802.05668, 2018. [Online]. Available: http://arxiv.org/abs/1802.05668

[27] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” *CoRR*, vol. abs/1812.00332, 2018. [Online]. Available: http://arxiv.org/abs/1812.00332

[28] H. Cai, C. Gan, and S. Han, “Once for all: Train one network and specialize it for efficient deployment,” *CoRR*, vol. abs/1908.09791, 2019. [Online]. Available: http://arxiv.org/abs/1908.09791

[29] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. H. Jin, S. Zhao, and K. Keutzer, “Squeezenext: Hardware-aware neural network design,” *CoRR*, vol. abs/1803.10615, 2018. [Online]. Available: http://arxiv.org/abs/1803.10615

[30] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[31] N. Aghli and E. Ribeiro, “Combining weight pruning and knowledge distillation for cnn compression,” in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2021, pp. 3185–3192.

[32] J. Kim, S. Chang, and N. Kwak, “PQK: model compression via pruning, quantization, and knowledge distillation,” *CoRR*, vol. abs/2106.14681, 2021. [Online]. Available: https://arxiv.org/abs/2106.14681

[33] M. A. Ganaie, M. Hu, M. Tanveer, and P. N. Suganthan, “Ensemble deep learning: A review,” *CoRR*, vol. abs/2104.02395, 2021. [Online]. Available: https://arxiv.org/abs/2104.02395

[34] Y. He and S. Han, “ADC: automated deep compression and acceleration with reinforcement learning,” *CoRR*, vol. abs/1802.03494, 2018. [Online]. Available: http://arxiv.org/abs/1802.03494

[35] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, “Amc: Automl for model compression and acceleration on mobile devices,” in *Computer Vision – ECCV 2018*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Springer International Publishing, 2018, pp. 815–832.

[36] X. Wang, D. Kondratyuk, K. M. Kitani, Y. Movshovitz-Attias, and E. Eban, “Multiple networks are more efficient than one: Fast and accurate models via ensembles and cascades,” *CoRR*, vol. abs/2012.01988, 2020. [Online]. Available: https://arxiv.org/abs/2012.01988