Nicolas Loerbroks Follow
Feb 9, 2018 · 14 min read


**State of the Art in Compressing Deep Convolutional Neural Networks**

Here at LeapMind we are working on making deep learning empowered intelligent IoT devices a widespread reality. One of the topics our R&D is most concerned about is the compression of deep learning models. In this blog post I would like to talk about why this is important and give an overview of the literature on this topic.

**Why think about compression?**

Deep Convolutional Neural Networks (CNN) have set the state of the art for a variety of applications, especially in the field of computer vision. One of the main obstacles for this technology to become more widespread is the huge data storage requirement. The original AlexNet architecture requires about 240MB of memory to store the weight parameters needed for classifying a single image, while its deeper successor VGG requires considerably larger memory (528MB) [1][2]. One of the reasons for this, especially in those early years of deep learning, may have been given by the fact that progress was mostly driven by achieving the best results on benchmarks such as ImgeNet. Besides, it was really more about making deep learning work at all.

In a cloud-based environment with abundant computational capabilities, enabled by multiple graphical processing units (GPUs), such massive memory requirements may not be considered a restriction. However, in case of mobile or edge-based embedded devices with limited computational capabilities, such resource intensive deep neural networks cannot be readily applied. Recently, the proliferation of deep learning applications on

mobile IoT devices, including smartphones, has unveiled this as a major hurdle for a wide spread use.

Thus, the design of deep neural networks that require less storage and computation power has established itself as a new research direction. Particularly, the modification of large cumbersome models that reduces the memory requirements while retaining as much of its performance as possible is referred to as compression of neural networks. Another direction is the design of more memory efficient network architectures from scratch. In the following Iwill discuss the different approaches in more detail.

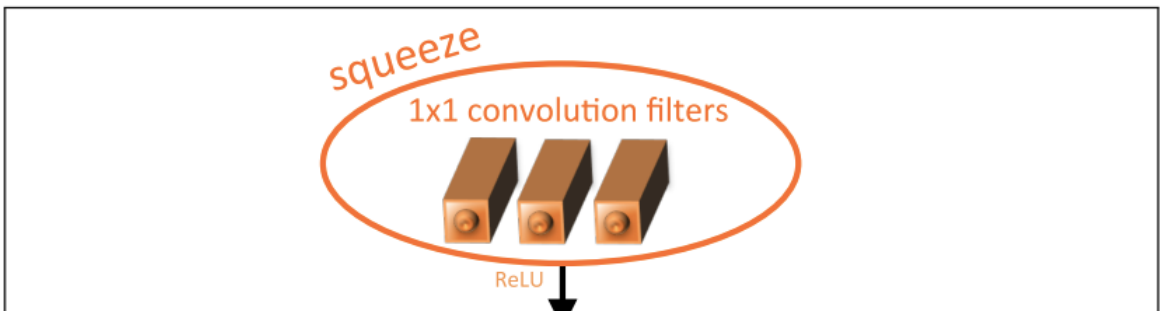
**Memory efficient architectures**

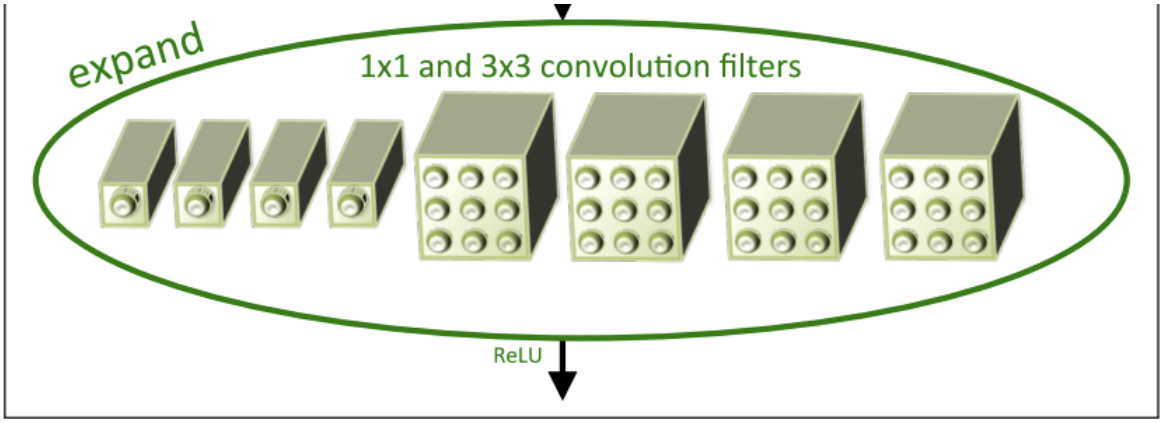
Rethinking the design of the CNN architecture is the most straightforward idea to arrive at more memory friendly models. First, let's recap the memory requirements for the standard building blocks of CNNs: For a fully connected layer with i input nodes and j output nodes the amount of necessary weights is given by i x j. For a convolutional layer with M input channels, N output channels and Kernel size K by K the amount of parameters for this layer is given by M x N x K x K.

For instance, in VGG16 the amount of weights in one layer of the last convolutional block is given by 512 x 512 x 3 x 3 = 2.4M while feeding the final 7-by-7-feature maps into the first 4096-node-fc-layer accounts for 512 x 7 x 7 x 4096 = 102.8M parameters alone. Thus, abandoning the final fc-layers has become common practice in more recent architectures such as ResNet or Inception, since it leads to a significant decrease in memory usage while these models are by an order of magnitude deeper [3][4].

In recent time there have been a variety of papers that address the individual factors that contribute to the complexity of convolutional layers.

SqueezeNet in early 2016 was the first paper that was concerned with building a memory efficient architecture [5]. Here 1x1-convolutional kernels are applied to "squeeze" the input, i.e. reduce the amount of channels before applying the more expensive 3x3-kernels. More precisely, the architecture consists of consecutive "Fire-Modules" that consist of 2 convolutional layers. The first layer consists entirely of 1x1-convolutions through which the amount of channels is being reduced (i.e. $N < M$), before applying a combination of 1x1- and 3x3-kernels in the next layer, arriving at 4.8MB of parameters at AlexNet level accuracy.



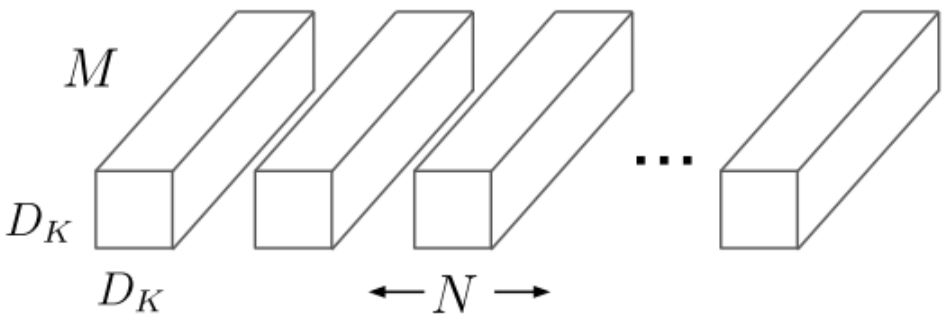



A "Fire-Module" of the SqueezeNet architecture. The figure is taken from the paper[5]

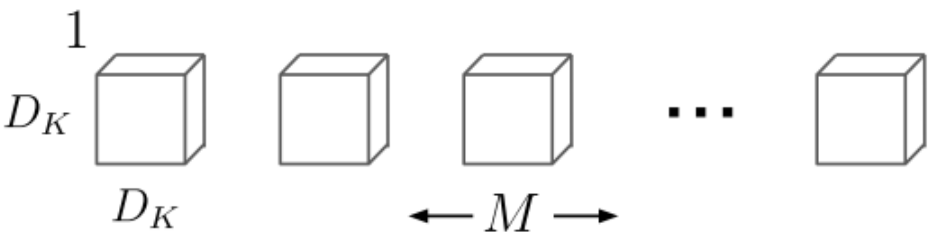
Google's MobileNets goes one step further by modifying the convolutional operation as such [6].

The key feature of this architecture are the so called depthwise separable convolutions. In standard convolutions, if we interpret a kernel as a 3-dimensional object, there are N x (K x K x M) kernels where each kernel takes the same M feature maps and modifies them according to its weights to arrive at N feature maps for the next layer. In depthwise separable convolutions the standard convolutions are replaced by 2 steps. First, a single Kernel of size K x K x M is applied to the input (Depthwise convolution). Then, after batch normalization and ReLU activation, N 1x1-Filters are applied (Pointwise convolution) to arrive at N separate feature maps in the output layer. By effectively reducing the amount of expensive K x K-Kernels by a factor of N, a significant amount of memory and computation can be saved.

By applying this technique MobileNets offers 70.6% top-1 accuracy on ImageNet (compared to ~57% for AlexNet) with a memory requirement of 16MB.




(a) Standard Convolution Filters



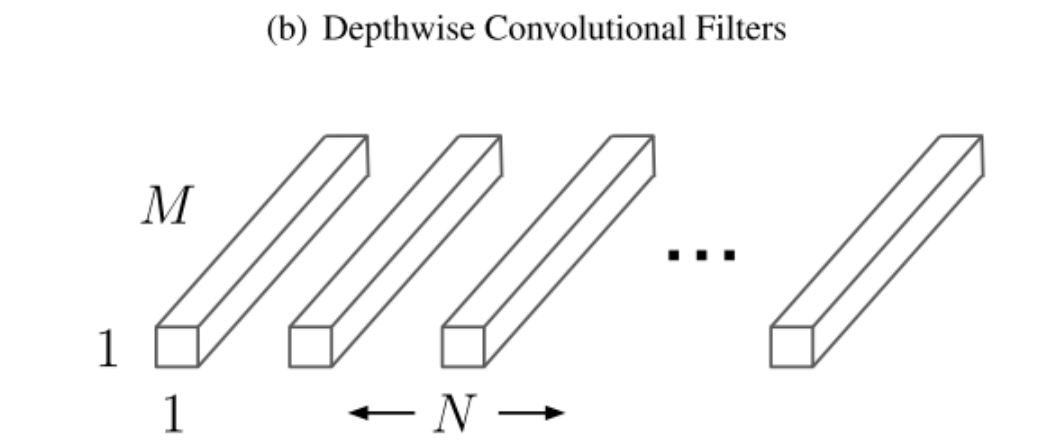



Figure 2. The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter.

Illustration of MobileNets' depthwise separable convolution. The figure is taken from the paper[6]

The idea of depthwise convolutions in MobileNets can be generalized to group-wise convolutions as in SEP-Nets [7] or ShuffleNets [8]. In MobileNets' Depthwise convolutions a single K x K x M-Kernel is applied to the input. Group-wise convolutions generalized this idea by partitioning the feature maps of the input into k groups and applying the same Kernel to all feature maps of the group, reducing the amount of parameters by a factor of k/M compared to regular convolutions. (i.e. Group-wise convolution = Depthwise convolution iff. k=1 and Groupwise convolution = regular convolution iff. k=M).

The amount of desired groups is now implemented as an argument for a convolutional layer in Pytorch.

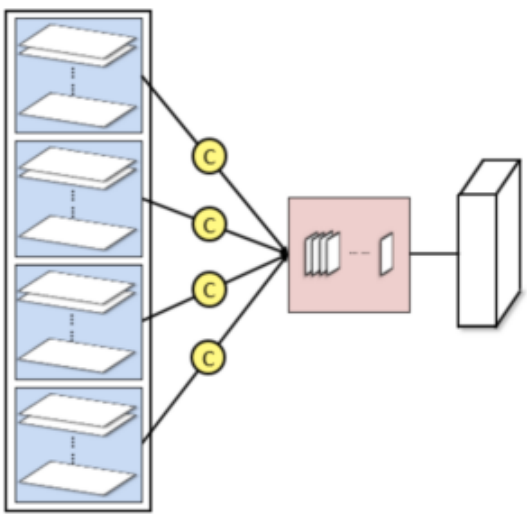


Illustration of Group-wise convolutions. The figure is taken from the SEP-Nets paper[7]

**From Float to low bitwidth — Compression through Quantization**

Another straightforward way to exploit the redundancy of a Deep Neural Network is to reduce the bitwidth of its parameters. It appears that reducing the bitwidth of a large convolutional neural network from 32bit floating point down to 8bit floating point numbers as a post processing step does not significantly impact performance. This holds even for network architectures that were already designed to save memory, such as MobileNets or SqueezeNet. In fact, this is already supported by Tensorflow [9]. However, if going below 8 bits, it might be unavoidable to train the network with respect to the low bitwidth weights. Since all the optimization algorithms, such as stochastic gradient descent or Adam, employed for neural networks rely on higher precision weights and gradients, the question of how to train such networks has let to an emerging research direction.

Another strongly related idea is the quantization of not only weights but also activations (i.e. inputs into the convolution operation). While the quantization of weights is sufficient to reduce the memory requirements for the models, additionally quantizing the activations allows for significant speed up on dedicated hardware (such as FPGAs) where in the case of 1 or 2 bits for both weights and activations the matrix multiplications could be entirely replaced by logical operations. Partly for this reason, hardware acceleration of deep learning has established itself as a separate research field that could easily fill another blog post. I will stick to the software aspect and summarize relevant papers regarding the quantization of neural networks.

The general idea is to first train a quantized model on GPU before deploying it on an edge device for inference.

The most extreme case of quantization is binarization, i.e. restricting the weights to be either -1 or +1 during interference which has been extensively studied by Courbariaux et al. [10][11][12]. In their original BinaryConnect paper, they propose to train those networks by retaining the full precision weights for the weight update but backpropagate with respect to the binary weights. In other words, the loss is calculated with respect to the binary weights, but the parameter update is performed in full precision. The binarization operation that is performed after every weight update, is simply given by the sign function (or a stochastic version of it that sets a weight w to +1 with probability $\max(0, \min(1, (w+1)/2))$ to regularize the training).

They extend this idea to binary activations in their BNN paper [11]. In this case not only the weight update but also backpropagation has to be performed with respect to the full precision weight since the gradient would be zero almost everywhere with respect to the discrete values. In their paper called XNOR [13] Rastegari et al. extend this approach with their main contribution being the introduction of a scaling factor for the binary convolution which is given by the average value of the full precision weights for every Kernel.

In their QNN paper [12] Courbariaux et al. further extend the strictly binary convolutions to higher bitcounts by generalizing the use of the signum function for binarization to linear quantization.

Another approach is given by Ternary weights where the model parameters can take the values -1, 0 and +1. Li et al. propose an appropriate scaling factor and threshold delta to extend the approach of BinaryConnect to the ternary case such that a full precision weight w is set to -1 if w<-delta, 0 if w<|delta| and 1 if w>delta[14].

A nonlinear quantization approach is proposed by Zhou et al. where weights are quantized to either powers of two or zero [15]. This idea allows efficient bit shift operations to replace floating-point multiplications even for higher bitwidths. Furthermore, the authors introduce a quantization procedure different from aforementioned publications: Instead of quantizing all weights at the same time, quantization is performed incrementally. In each iteration, only some weights are quantized and the remaining floating-point weights are retrained to compensate for the loss in accuracy. The fraction of quantized weights is stepwise increased until reaching 100%. Experiments with different architectures on ImageNet show improved accuracy compared to the full-precision model when quantizing to 5bits. Using a bitwidth of 2, hence ternary weights, ResNet-18 could be quantized with a loss in accuracy of 2.3%.

**Optimal Brain Damage — Removing Redundancy through Pruning**

Despite the utilization of powerful regularization techniques like dropout or weight decay some weights of a network will always contribute more to the prediction than others. The process of removing the less contributing weights to compress (and or further regularize) the network is called pruning. After the some weights have been pruned, the network typically has to be fine tuned again to have it adapt to the change.

This idea was first proposed by Yann Le Cunn et.al back in 1990 in their famous paper called Optimal Brain Damage (OBD) and was later applied to modern deep networks [16].

Research on pruning is mostly concerned with the question of how the contribution of the weights should be measured. In OBD the contribution is measured by the effect on the training error in the case of setting this particular parameter to zero. Obviously, this method becomes computationally infeasible for deep networks. In Deep Compression, Han et. al simply prune the weights with lowest absolute value, reducing the amount of weights to 10% of its original size for fully connected, and around 60% for convolutional layers at no loss of prediction accuracy [17].

Tu et al. proposed a method to accurately measure the Fisher information associated with a weight and use it as a measure for its contribution [18]. More recently, more advanced layer-wise methods have also been proposed [19][20].

**From a large model to a small one — Knowledge Transfer through Distillation**

A more generic way to compress a given model, is to force a smaller model to mimic its behaviour. In the context of Deep Learning, this idea is known as the teacher-student approach or knowledge distillation [22]. Let's say we have trained a large state-of-the-art model like ResNet or Inception (i.e. the teacher) and want to make use of its predictive power to train a smaller model (i.e. the student). To achieve this, when training the student, we perform forward passes on both, the teacher and the student, and calculate the cross entropy between the prediction of the teacher model and the student model which is added to the loss of the student. Trained in this way, the student does not only learn based on the ground truth labels but additionally learns from the teacher what is referred to as the "dark knowledge" of the model, i.e. which categories can be considered close to one another.

For example, let's say our task is to classify images into four categories: Cats, dogs, horses and zebras. A strong model will learn that cats are more similar to dogs than to horses and zebras and can be expected to give a softmax output like (0.7, 0.25, 0.035, 0.015). In practice the softmax outputs of the teacher model are smoothed by dividing the pre-softmax outputs by a factor — which becomes a hyperparameter referred to as temperature — to drive them further away from the hard targets (i.e. the ground truth labels). Thus, if the student model is additionally trained with this information rather than the ground truth labels alone it can be expected to perform better than if trained from scratch. In a more abstract sense we can say that we have compressed the teacher model by transferring its knowledge to a smaller model that gives (ideally) the same output.

Illustration of the Distillation pipeline

This idea was first proposed by Ba and Caruana [21] and later by Hinton [22] and since then some modifications to the algorithm have been proposed that can boost the performance in specific cases [23]. Distillation has also been successfully applied to Object detection, showcasing that it scales to large benchmark datasets, even for more complex tasks [24][25].

**Putting it all Together**

Until today the only research papers in which several of the above techniques have been successfully combined are the work by Song Han et al. [5][17]. In their work called Deep Compression, first, a trained network is pruned by setting connections to zero if the absolute value of the weight is below a certain threshold. Second, quantization and weight sharing is applied. The weights are clustered into 256 groups for convolutional, and 32 groups for fc-layers, respectively, using k-means. Hence a weight can be represented using 8 bit (in convolutional layers) and 5 bit (in fc-layers) indices representing the centroids of the corresponding cluster. Weights are not shared across layers. The centroids are then fine tuned in an additional retraining phase, where the loss with respect to the centroid of a cluster is simply given by the additive loss of the weights that belong to it. Finally Huffman-coding is applied to the indices and centroids to further compress the representation of the parameters. Applying this method to SqueezeNet resulted in 10x compression using 64 clusters (i.e. 6 bit representations for weights) without loss of accuracy on a network architecture which is already optimized for compression.

Deep Compression was first published in 2015 and is already quite old by the standards of literature on deep learning. In fact, the vast majority of research papers that we introduced above are much more recent, however, they are usually only concerned with exploring a single aspect of compression in isolation, e.g. pruning full precision weights or quantizing large cumbersome networks like ResNet.

Unfortunately, it is yet not very well understood how those methods behave in combination, and there is an intuitive concern that all those approaches just exploit the same redundancy contained in deep learning models in a different way. If you don’t believe me, take a MobileNets, prune it, and then binarize the weights.

Coming up with new ways for compression and combining existing approaches in an innovative fashion will be a key factor to make deep learning empowered mobile devices a widespread reality.

**References**

[1] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (n.d.). ImageNet Classification with Deep Convolutional Neural Networks. Retrieved from

https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] Simonyan, K., & Zisserman, A. (2015). VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. Retrieved from https://arxiv.org/pdf/1409.1556.pdf

[3] He, K., Zhang, X., Ren, S., & Sun, J. (n.d.). Deep Residual Learning for Image Recognition. Retrieved from https://arxiv.org/pdf/1512.03385.pdf

[4] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. (n.d.). Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. Retrieved from https://arxiv.org/pdf/1602.07261.pdf

[5] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (n.d.). SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND <0.5MB MODEL SIZE

[6] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., … Adam, H. (n.d.). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. Retrieved from https://arxiv.org/pdf/1704.04861.pdf

[7] Li, Z., Wang, X., Lv, X., & Yang, T. (n.d.). SEP-Nets: Small and Effective Pattern Networks. Retrieved from https://arxiv.org/pdf/1706.03912.pdf

[8] Zhang, X., Zhou, X., Lin, M., & Sun, J. (n.d.). ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. Retrieved from https://arxiv.org/pdf/1707.01083v1.pdf

[9] https://www.tensorflow.org/performance/quantization

[10] Courbariaux, M., & David, J. (n.d.). BinaryConnect : Training Deep Neural Networks with binary weights during propagations https://arxiv.org/pdf/1511.00363.pdf

[11] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., Il, R. T. A., Bengio, Y., & Com, Y. U. (n.d.). Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1. Retrieved from https://arxiv.org/pdf/1602.02830.pdf

[12] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Quantized Neural Networks Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. Retrieved from https://arxiv.org/pdf/1609.07061.pdf

[13] Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (n.d.). XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. Retrieved from https://arxiv.org/pdf/1603.05279.pdf

[14] Li, F., Zhang, B., & Liu, B. (n.d.). Ternary weight networks. Retrieved from https://arxiv.org/pdf/1605.04711.pdf

[15] Zhou, A., Yao, A., Guo, Y., Xu, L., & Chen, Y. (n.d.). INCREMENTAL NETWORK QUANTIZATION: TOWARDS LOSSLESS CNNS WITH LOW-PRECISION WEIGHTS.

[16] Cun, L., Denker, S., Le Cun, Y., Denker, J. S., & Sol1a, S. A. (n.d.). Optimal Brain Damage. Retrieved from https://papers.nips.cc/paper/250-optimal-brain-damage.pdf

[17] Han, S., Mao, H., & Dally, W. J. (n.d.). DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING. Retrieved from https://arxiv.org/pdf/1510.00149.pdf

[18] Tu, M., Berisha, V., Woolf, M., Seo, J.-S., & Cao, Y. (n.d.). RANKING THE PARAMETERS OF DEEP NEURAL NETWORKS USING THE FISHER INFORMATION. Retrieved from http://www.mirlab.org/conference_papers/International_Conference/ICASSP 2016/pdfs/0002647.pdf

[19] Dong, X., Chen, S., & Pan, S. J. (n.d.). Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon. Retrieved from https://arxiv.org/pdf/1705.07565.pdf

[20] Aghasi, A., Abdi, A., Nguyen, N., & Romberg, J. (n.d.). Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee. Retrieved from https://arxiv.org/pdf/1611.05162.pdf

[21] Lei Jimmy Ba and Rich Caruana: Do Deep Nets Really Need to be Deep? https://arxiv.org/pdf/1312.6184.pdf

[22] Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the Knowledge in a Neural Network. Retrieved from https://arxiv.org/pdf/1503.02531.pdf

[23] Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., & Bengio, Y. (2015). FITNETS: HINTS FOR THIN DEEP NETS. Retrieved from https://arxiv.org/pdf/1412.6550.pdf

[24] Chen, G., Choi, W., Yu, X., Han, T., & Chandraker, M. (n.d.). Learning Efficient Object Detection Models with Knowledge Distillation. Retrieved from https://papers.nips.cc/paper/6676-learning-efficient-object-detection-models-with-knowledge-distillation.pdf

[25] Wang, C., Lan, X., & Zhang, Y. (n.d.). Model Distillation with Knowledge Transfer from Face Classification to Alignment and Verification. Retrieved from https://arxiv.org/pdf/1709.02929.pdf

Machine Learning | Deep Learning | Convolutional Network | Pruning | Neural Net Compression