Deep Model Compression and Architecture Optimization for Embedded Systems: A Survey

Anthony Berthelier, Thierry Chateau, Stefan Duffner, Christophe Garcia, Christophe Blanc

Noname manuscript No.
(will be inserted by the editor)

Deep Model Compression and Architecture Optimization for Embedded Systems: A Survey

Anthony Berthelier · Thierry Chateau · Stefan Duffner · Christophe Garcia · Christophe Blanc

Received: date / Accepted: date

Abstract Over the past, deep neural networks have proved to be an essential element for developing intelligent solutions. They have achieved remarkable performances at a cost of deeper layers and millions of parameters. Therefore utilising these networks on limited resource platforms for smart cameras is a challenging task. In this context, models need to be (i) accelerated and (ii) memory efficient without significantly compromising on performance. Numerous works have been done to obtain smaller, faster and accurate models. This paper presents a survey of methods suitable for porting deep neural networks on resource-limited devices, especially for smart cameras. These methods can be roughly divided in two main sections. In the first part, we present compression techniques. These techniques are categorized into: knowledge distillation, pruning, quantization, hashing, reduction of numerical precision and binarization. In the second part, we focus on architecture optimization. We introduce the methods to enhance networks structures as well as neural architecture search techniques. In each of their parts, we describe different methods, and analyse them. Finally, we conclude this paper with a discussion on these methods.

Keywords Deep learning · Compression · Neural networks · Architecture

A. Berthelier
Institut Pascal - 4 Avenue Blaise Pascal, 63178 Aubiere, France
Tel.: +33630899676
E-mail: anthony.berthelier@etu.uca.fr

T. Chateau
Institut Pascal - 4 Avenue Blaise Pascal, 63178 Aubiere, France

S. Duffner
LIRIS - 20, Avenue Albert Einstein, 69621 Villeurbanne Cedex, France

C. Garcia
LIRIS - 20, Avenue Albert Einstein, 69621 Villeurbanne Cedex, France

C. BLanc
Institut Pascal - 4 Avenue Blaise Pascal, 63178 Aubiere, France

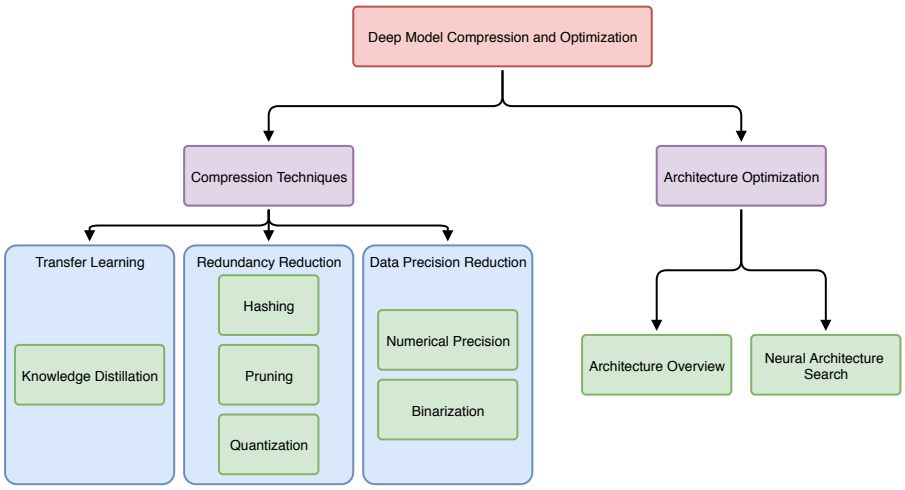

Fig. 1: Roadmap of our paper.

1 Introduction

Since the advent of deep neural network architectures and their massively parallelized implementations [1,2], deep learning based methods have achieved state-of-the-art performance in many applications such as face recognition, semantic segmentation, object detection, etc. In order to achieve these performances, a high computation capability is needed as these models have usually millions of parameters. Moreover, the implementation of these methods on resource-limited devices for smart cameras is difficult due to high memory consumption and strict size constraints. For example, AlexNet [1], is over 200MB and all the milestone models that followed such as VGG [3], GoogleNet [4] and ResNet [5] are not necessarily time or memory efficient. Thus finding solutions to implement deep models on resource-limited platforms such as mobile phones or smart cameras is essential. Each device has a different computational capacity. Therefore, to run these applications on embedded devices the deep models need to be less-parametrized in size and time efficient.

Few works has been done focusing on dedicated hardware or FPGA with a fixed specific architecture. Having a specific hardware is helpful to optimize a given application. However, it is difficult to generalise. The CPU architectures of the smartphones are different from each other. Thus, it is important to develop generic methods to help optimize neural networks. This paper aims to describe general compression methods for deep models that can be implemented on a large range of hardware architectures, especially on various generic-purpose CPU architectures. Moreover, we are specifically interested in multilayer perceptron (MLP) and Convolutional Neural Networks (CNNs) because these types of state-of-the-art models have a large number of parameters.

However, some methods could be applied to recurrent neural networks such as LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) [6, 7].

Few surveys exist on deep neural compression [8, 9]. However these works are mainly focused on compression and acceleration algorithms of existing models. In this paper, we present not only the methods to compress or accelerate deep model, but also the recent research concerning optimized architectures search and design. The article is organized as follows. The first part addresses the compression techniques (Section 2) which reduce the size and computation requirements of a model by using different algorithms. Knowledge distillation methods are explained to tackle the problem of transfer learning (Section 2.1). Followed are the hashing (Section 2.2), pruning (Section 2.3) and quantization (Section 2.4) methods which explore the redundancy of the networks. Numerical precision (Section 2.5) and binarization (Section 2.6) are presented by introducing the use of data with lower precision. The second part of this paper describes architecture optimization (Section 3). We begin with the description of implementations to optimize network architecture and the way different modules are designed to interact with each other (Section 3.1). Then we explain methods to automatically search optimized architecture (Section 3.2). In each of these parts, we present existing methods, their strengths, weaknesses and in which context they may be applied. The structure of the article is detailed in Figure 1.

2 Compression techniques

2.1 Knowledge distillation

To design a neural network, it is important to evaluate how deep the network needs to be. A neural network is composed of an input, an output and intermediate layers. A shallow neural network is a network with a lower number of intermediate layers as opposed to a deep neural network. A deeper network has more parameters and can potentially learn more complex functions e.g. hierarchical representations [10]. The theoretical work from [10] revealed the difficulty involved to train a shallow neural network with the same accuracy as a deep network. However, an attempt was made to train a shallow network on SIFT features in order to classify the Imagenet dataset [1]. The authors concluded that it was a challenging task to train highly accurate shallow models [10].

In spite of that, Ba et al. [11] reported that neural networks with a shallower architecture are able to learn the same function as deep networks, with a better accuracy and sometimes with a similar number of parameters (see Figure 2). Inspired by [12], their model compression consists in training a compact model to approximate, to mimic, the function learned by a complex model. This is what knowledge distillation is about : transfer the knwoledge learned by a model to another one. The preliminary step is to train a deep network (the teacher network) to generate automatically labelled data by sending un-

labelled data through this deep network. Next, this "synthetic" dataset is then used to train a smaller mimic model (the student network), which assimilates the function that was learned by the larger model. It is expected that the mimic model should produce same predictions and mistakes as the deep network. Thus, similar accuracy can be achieved between an ensemble of neural networks and its mimic model with 1000 times fewer parameters. In [11], the authors demonstrated this assertion on the CIFAR-10 dataset. An ensemble of deep CNN models was used to label some unlabeled data of the dataset. Next, the new data were used to train a shallow model with a single convolution and maxpooling layer followed by a fully connected layer with 30k non-linear units. In the end, the shallow model and the ensemble of CNN acquired the same level of accuracy. Further improvements have been made on student-teacher techniques, especially with the work of Hinton et al. [13]. Their framework utilizes the output from the teacher's network to penalize the student network. Additionally it is also capable of retrieving an ensemble of teacher networks to compress their knowledge into a student network of similar depth.

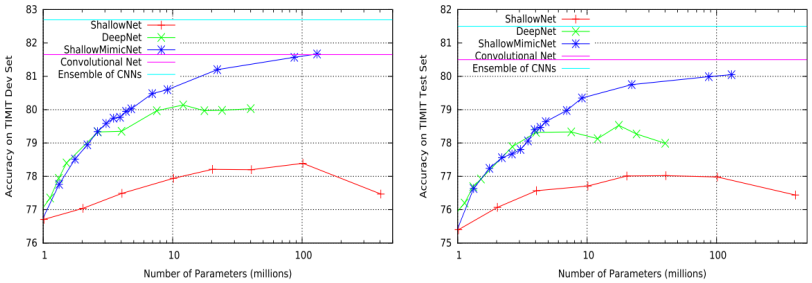

Fig. 2: Accuracy of different deep neural networks, shallow neural networks and shallow mimic neural networks against their number of parameters on TIMIT speech database Dev (left) and Test (right) sets. Results and figures are from [11].

In recent years, other compression methods that are described in this paper are preferred. However, some works are coupling transfer learning techniques with their own methods to achieve strong improvements. For example, the works of Chen et al. [14] and Huang et al. [15] follow this approach employing additional pruning techniques (see section 2.3). The former uses a deep metric learning model, whereas the latter handles the student-teacher problem as a distribution matching problem by trying to match neuron selectivity patterns between them to increase the performance. Aguilar et al. [16] propose to distill the internal representations of a teacher models into a simplified version of it to improve the learning and the performance of the student model. Lee et al. [17] use a self-supervised learning algorithm to improve transfer learning methods. These methods are efficient. However their performances can vary largely according to the application. Classification tasks are easy to learn for a shallow

model, but tasks like segmentation or tracking are difficult to apprehend even with a deep model. Furthermore, Muller et al. [18] recently showed with label smoothing experiments that teacher and student networks are sensitive to the format of the data. Thus, improving knowledge distillation methods is also a difficult mission.

2.2 Hashing

Hashing is employed to regroup data in a neural network to avoid redundancy and access the data faster. Through empirical studies, hashing methods have proven themselves to be an effective strategy for dimensionality reduction [19].

HashedNets [20] is a hashing methods utilized and developed by Nvidia. In this model, a hash function is used to uniformly and randomly group network connections into hash buckets. As a result, every connection that is in the i^{th} hash bucket has the same weight value w_i. This technique is especially efficient on fully connected feed forward neural networks. Moreover, It can also be used in conjunction with other neural network compression methods.

Several other hashing methods have been developed in the past few years. Spring et al. [21] proposed an approach where adaptive dropout [22] (i.e. choosing nodes with a probability proportional to some monotonic functions of their activations) and hash tables based on locality-sensitive hashing (LSH) [23–26] are utilized. These techniques once combined allowed the authors to construct a smart structure for maximum inner product search [27]. This technique exhibits better results, reducing computational costs for both training and testing. Furthermore, this kind of structure leads to sparse gradient updates and thus a massively asynchronous model. Thereby, models can be easily parallelized as the data dispersion could be wider. However, wider data dispersion can result in a slow down of the model. A trade-off between these criteria is necessary.

2.3 Pruning

The compression of neural networks by using pruning techniques has been widely studied. These techniques enable to remove parameters of a network that are not necessary for a good inference. The early work in this domain was aiming to reduce the complexity and the over-fitting in networks [28, 29]. In these papers, the authors used pruning techniques based on the Hessian of the loss function to reduce the number of connections inside the network. The method finds a set of parameters whose deletion would cause the least increase of the objective function by measuring the saliency of these parameters. The authors use numerous approximations to find these parameters. For instance, the objective function is approximated by a Taylor series. Finding parameters whose deletion does not increase this function is a difficult problem that involves, for example, the computation of huge matrices as well as second

derivatives. Also, these methods suggest that reducing the number of weights by using the Hessian of the loss function is more accurate than magnitude-based pruning like weight decay. Additionnally, it reduces the network over-fitting and complexity. However, the second-order derivatives introduce some computational overhead.

Signorini et al. [30] utilized an intuitive and efficient method to remove parameters. The first step is to learn the connectivity of the network via a conventional training of the network i.e. to learn which parameters (or connections) are more important than the other. The next step consists in pruning those connections with weights below a threshold i.e. converting a dense network into a sparse one. Further, the important step of this method is to retrain (fine-tune) the network to learn the weights of the remaining sparse connections. If the pruned network is not retrained, then the resulting accuracy is considerably lower. The general steps for pruning a network are presented on Figure 3.

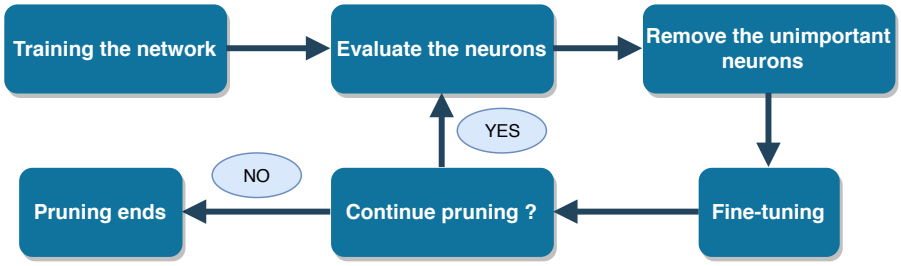

Fig. 3: Basic steps for pruning a deep network. Figure inspired by [31].

Anwar et al. [32] used a similar method. However, they state that pruning has the drawback of constructing a network that has "irregular" connections, which is inefficient for parallel computing. To avoid this problem, the authors introduced a structured sparsity at different scales for CNN. Thus, pruning is performed at : the feature map, the kernel and the intra-kernel levels. The idea is to force some weights to zero but also to use sparsity at well defined activation locations in the network. The technique consists in constraining each outgoing convolution connection for a source feature map to have similar stride and offset. This results in a significant reduction of both feature and kernel matrices. Usually, sparsity has been studied in numerous works in order to penalize non-essential parameters [33–36].

Similar pruning approach is seen in Molchanov et al. [31]. However different pruning criteria and technical considerations are defined to remove features maps and kernel weights, e.g. the minimum weight criteria [30]. They assume that if an activation value (an output feature map) is small, then the feature detector is not important in the application. Another criteria involves the mutual information which measures how much information is present in

a variable about another one. Further, the Taylor expansion is used similar to LeCun [28], to minimise the computational cost between the pruned and the non-pruned network. In this case, pruning is treated as an optimization problem.

A recent pruning method [37] consists in removing filters that are proven to have a small impact on the final accuracy of the network. This results in automatically removing the filter's corresponding feature map and related kernels in the next layer. The relative importance of a filter in each layer is measured by calculating the sum of its absolute weights, which gives an expectation of the magnitude of the output feature map. At each iteration, the filters with the smallest values are pruned. Recently, Jian-Hao et al. [38] developed a pruning network called ThiNet which, instead of using information of the current layer to prune unimportant filters of that layer, uses information and statistics of the subsequent layer to prune filters from a given layer. Not only weights and filters but also channels can be pruned [39] using complex thresholding methods.

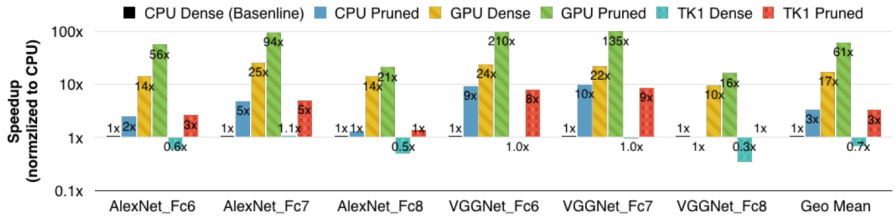

Fig. 4: Comparison of the speed of AlexNet and VGG before and after pruning on CPU, GPU and TK1. Figure from [40].

These past few years, numerous networks compression algorithms using pruning methods and achieving state-of-the-art results have emerged. Yu *et al.* [41] proposed a neurons importance score propagation (NISP) method based on the response of the final layers to evaluate the pruning impact of the prior layers. Zhuang *et al.* [42] developed discrimination-aware losses in order to determine the most useful channels in intermediate layers. Some methods such as Filter Pruning Via Geometric Median (FPGM) [43] are not focused on pruning filters with less importance but only by evaluating their redundancy. Similarly, Lin *et al.* [44] tackled the problem of redundant structures by proposing a generative adversarial learning method (GAL) (not only to remove filters, but also branches and blocks).

Factorization methods are also use such as matrix or tensor decomposition [45,46]. However decomposition operations are computationally expensive and factorization methods are also time-consuming as the model needs to be retrained numerous times. As a result, we will not go into detail on these methods in this paper. However, an overview of these techniques can be found in [47].

Numerous pruning methods exist and each of them has strength and weaknesses. The main disadvantage of these methods is that it takes a long time to prune networks due to the constant retraining that they demand. Recent techniques like [48] try to bypass some steps by pruning neural networks during their training by using recurrent neural networks. However, all of them result in considerable reduction of parameters. Pruning methods allow to eliminate 10 to 30 percent of the network's weights. Regardless of the method, the size of a network can be decreased with pruning without change or significant drop in accuracy. The inference with the resulting models will also be faster (see Figure 4) but the actual speed depends on which method has been utilized and the sparsity of the network after pruning.

2.4 Quantization

Network quantization is similar to pruning as this is a common technique in the deep learning community. It aims to reduce the number of bits required to represent every weight. In other words, it decreases the number of parameters by exploiting redundancy. Quantization reduces the storage size with minimal loss in performance. In a neural network, it means that parameters will be stacked into clusters and share the same value with the parameters within the same cluster.

Gong et al. [49] performed a study on a series of vector quantization methods and found that performing scalar quantization on parameter values using a simple k-means is sufficient to compress them 8 to 16 times without a huge loss in accuracy. Few years later, Han et al. [40] utilized a trivial quantization method using k-means clustering. They performed a pruning step before and a Huffman coding step after the quantization in order to perform a larger compression of the network. In their experiments, the authors were able to reduce network storage by 35 to 49 times across different networks. Pruning and quantization are methods that are often used together to achieve a solid compression rate. For example, for a LeNet5-like network [50], pruning and quantization compressed the model 32 times and with huffman coding even 40 times.

It is possible to apply several quantization methods on neural networks. Choi Y. et al. [51] defined a Hessian-weighted distortion measure as an objective function in order to decrease the quantization loss locally. Further, a Hessian-weighted k-means clustering is used for quantization purposes to minimize the performance loss. Recent neural network optimizers can provide alternatives to the Hessian and thus reduce the overall computation cost, like Adam [52], AdaGrad [53], Adadelta [54] or RMSProp [55]. However one of the advantages of using the Hessian-weighted method is that the parameters of all layers in a neural network can be quantized together at once compared to the layer-by-layer quantization used previously [40, 49].

Quantization techniques are efficient as they achieve an impressive compression rate and can be coupled with other methods to compress the models

further. Their efficiency is integrated in some frameworks and tools to directly quantify a network and port it on mobile devices [56,57].

2.5 Reducing numerical precision

Although the number of weights can be considerably reduced using pruning or quantization methods, the overall number of parameters and costly matrix multiplications might still be enormous. A solution is to reduce the computational complexity by limiting the numerical precision of the data. Deep neural networks are usually trained using 32-bit floating-point precision for parameters and activations. The aim is to decrease the number of bits used (16, 8 or even less) and to change from floating-point to a fixed-point representation. Selecting the precision of data has always been a fundamental choice when it comes to embedded systems. When committed to a specific system, the models and algorithms can be optimized for the specific computing and memory architecture of the device [58–60].

However, applying quantization for deep neural networks is a challenging task. Quantization errors might be propagated and amplified throughout the model and thus have a large impact on the overall performance. Since the beginning of the 90's, experiments have been made in order to limit the precision of the data in a neural network, especially during backpropagation. Iwata et al. [61] created a backpropagation algorithm with 24-bit floating-point processing units. Hammerstrom [62] presented an architecture for on-chip learning using 8-16 bits fixed-point arithmetic. Furthermore, Holt and Hwang [63] showed empirically that only 8-16 bits are enough for backpropagation learning. Nonetheless, even if all these works are helping to understand the impact of limited numerical precision on neural networks, they are done on rather small models such as multilayers perceptron with only a single hidden layer and very few units. More sophisticated algorithms are required for more complex deep models.

In 2015, Gupta et al. [64] trained deep CNN using 16-bit fixed-point instead of 32-bit floating-point precision. It constrained neural networks parameters such as bias, weights and other variables used during the backpropagation such as activations, backpropagated error, weight updates and bias updates. Different experimentations have been made with this 16-bit fixed-point word length, e.g. varying the number of bits that encode the fractional (integer) part between 8 (8), 10 (6) and 14 (2), respectively. In other terms, the number of integer bits IL added to the number of fractional bit FL is always equal to 16. Tested on the MNIST and CIFAR-10 datasets with a fully connected and a convolutional network, the results were nearly the same as the floating-point baseline when decreasing the fractional part to 12-bit precision.

The crucial part in this method is the conversion of a floating point number (or higher precision format) into a lower precision representation. To achieve this, [64] describe two rounding schemes. The first one is the round-to-nearest method. It consists of defining $\lfloor x \rfloor$ as the largest integer multiple of $\epsilon = 2^{-FL}$

less than or equal to x. So given a number x and the target representation (IL,FL), the rounding is done as follows:

$$\begin{cases} \lfloor x \rfloor & if \lfloor x \rfloor \leq x \leq \lfloor x \rfloor + \frac{\epsilon}{2} \\ \lfloor x \rfloor + \epsilon & if \lfloor x \rfloor + \frac{\epsilon}{2} \leq x \leq \lfloor x \rfloor + \epsilon \ . \end{cases} \tag{1}$$

The second rounding scheme is stochastic rounding. It is a statistic and unbiased rounding where the probability of x to be rounded to $\lfloor x \rfloor$ is proportional to its proximity to $\lfloor x \rfloor$:

$$\begin{cases} \lfloor x \rfloor & w.p. \quad 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & w.p. \quad \frac{x - \lfloor x \rfloor}{\epsilon} \ . \end{cases} \tag{2}$$

Courbariaux et al. [65] investigated the impact of numerical precision, especially to reduce the computational cost of multiplications. Their experiments were performed with three formats: floating point, fixed point [64] and dynamic fixed point [66] (which is a compromise of the first two). Instead of having a single scaling factor with a fixed number for the integer part and another fixed number for the fractional part, several scaling factors are shared between grouped variables and are updated from time to time. The authors achieved similar conclusions as [64]: a low precision is sufficient to run and train a deep neural network. However, limited precision can be efficient when it is paired and optimized with a specific hardware. Gupta et al. [64] achieved good results when they paired the fixed point format with FPGA-based hardware but the hardware optimization of dynamic fixed point representations is not as simple. Neural networks with limited-precision parameters and their optimized integration on hardware have already been studied in the past. For example, Mamalet et al. [67] and Roux et al. [68] developed optimized CNNs to detect faces and facial features in videos on embedded platforms in real-time. They used a fixed-point parameter representation but also optimized the inference algorithms for specific platforms. This allowed them to exploit parallel computing and memory locality.

To conclude, a limited numerical precision is sufficient to train deep models. It is helpful to save memory storage and computation time, even more if a dedicated hardware is used. However, not every step can be done with low precision in a neural network. For instance, a higher precision must be used to update the parameters during training.

2.6 Binarization

In recent works, limited numerical precision was extended to binary operations. In a binary network, the weights and the activations at least are constrained to either $+1$ or -1. Following the same idea as previously with limited numerical precision [65], the same authors decided to apply two rounding schemes

to binarize a variable: deterministic and stochastic rounding [69]. The most common rounding method is to maintain the sign of the variable. So for a variable x, its binary value x^b will be the sign of x ($+1$ if $x \geq 0$, -1 otherwise). The second binarization scheme is a stochastic rounding. Thus $x^b = +1$ with probability $p = \sigma(x)$ and $x^b = -1$ with probability $1 - p$ where σ is the hard sigmoid function [69]. The stochastic method is difficult to implement as it requires randomly generating bits from the hardware. As a result, the deterministic method is commonly used. However, recent works like [70] are focusing on alternative methods to approximate the weight values in order to obtain a more accurate network. For example, weights can be approximated using a linear combination of multiple binary weight bases.

Nevertheless, just like limited numerical precision, a higher precision is required at some point and real-valued weights are required during the backpropagation phase. Adding noise to weights and activations (such as dropout [71, 72]) is beneficial to generalization when the gradient of the parameters is computed. Binarization can also be seen as a regularization method [69].

With all these observations, Courbariaux et al. [73] developed a method called BinaryConnect to train deep neural networks using binary weights during the forward and backward propagation, while storing the true precision of the weights in order to compute the gradients. Firstly the forward propagation: layer-by-layer, the weights are binarized and the computation of the neuron's activation is faster because multiplications are becoming additions. Secondly the backward propagation: the training objective's gradient is computed in function of each layer's activation (from the top layer and going down layer-by-layer until the first hidden layer). Lastly the parameter update: the parameters are updated using their previous values and their computed gradients. During this final step more precision is needed.

As a consequence the real values are used (the weights are binarized only during the first two steps). Tests on datasets like MNIST, CIFAR-10 and SVNH can achieve state-of-the-art results with two-thirds less multiplications, training time accelerated by a factor of 3 and a memory requirement decreased by at least 16.

In a binary weight network, only weight values are approximated with binary values. This also works on CNNs where the models are significantly smaller (up to 32 times). Then, the operation of convolution can be simplified as follows:

$$I * W \approx (I \oplus B)\alpha \,, \tag{3}$$

where, I is the input, W the real-value weight filter, B the binary filter ($sign(W)$), α a scaling factor such that $W \approx \alpha B$ and $\oplus$ indicates a convolution without multiplications. Further improvements have been done with the XNOR-Net proposed by Rastegari et al. [74] where both the weights and the input to the convolutional and fully connected layers are binarized. In this case, all the operands of the convolutions are binary, and thus the convolution can be performed by only XNOR and bitcounting operations:

$$I * W \approx (sign(I) \oplus sign(W)) \odot K\alpha \,, \tag{4}$$

where, I is the input, W is the real-value weight filter and K is composed of the scaling factors for all sub-tensors in the input I.

The resulting network in [74] is as accurate as a single-precision network. It also runs faster (58 times on GPU) and is smaller (AlexNet is reduced to 7MB). Many existing models (like the hourglass model [75]) have been enhanced with the XNOR-Net method to achieve state-of-the-art results [76]. Recently, the XNOR-Net method has been studied to be transformed from a binarization task to a ternarization task [77]. Values are constrained in a ternary space -1, 0, +1. It allows to remove the need for full-precision values during the training by using a discretization method.

3 Architecture optimization

Compression methods are widely studied. In present times, some of them are part of popular deep learning frameworks. Tensorflow Lite [56] has tools to quantify models, allowing to transfer models to mobile devices easier. Core ML [57], the Apple framework for deep learning, is also able to apply some of these methods on the devices of the brand. Thus, on the one hand, a few compression techniques are already integrated in useful tools for developers but on the other hand, we are still quite far from understanding the intricacies of deep neural models.

However, these methods are usually applied on already constructed models as they aim to reduce their complexity. Thereby, recent research focuses directly on the architectures of these deep models, i.e. creating optimized architectures from the ground-up instead of finding methods to optimize them afterwards. This section of the survey is adressing these approaches. Firstly, a review of optimized architectures and modules to obtain efficient models is performed. Secondly, we present neural architecture search (NAS) methods to construct models "from scratch".

3.1 Architecture overview

To begin with, convolution operations are responsible for an important fraction of the computation time in a network. In early works of LeCun et al. [78], 5x5 and 3x3 filters are used. Although, some common deep models use larger kernels (e.g. Alexnet [1]), recent works recommend the use of 3x3 filters (e.g. VGG [3]). An architecture like GoogleNet [4] even use 1x1 filters. GoogleNet introduced the idea of modules, followed by ResNet [79] and DenseNet [80]. Modules are blocks composed of multiple convolution layers with different sizes and with a specific organization.

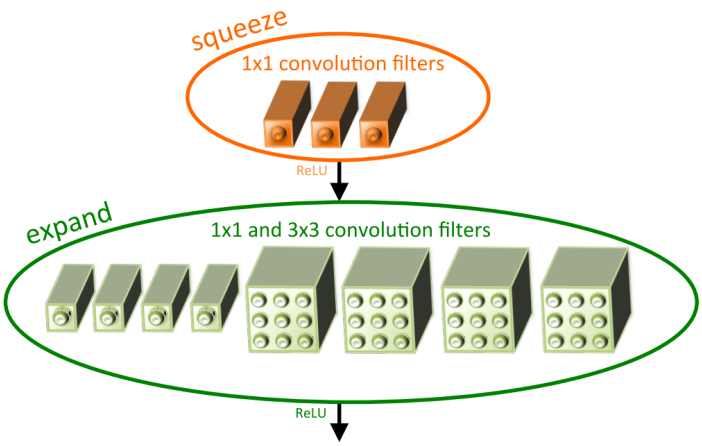

Fig. 5: Architecture of the SqueezeNet Fire module. Figure from [81].

Using the concept of modules, Iandola et al. [81] designed an architecture called SqueezeNet, which relies on a particular organisation of its layers. The *fire* module (see Figure 5) allowed to decrease the number of parameters of the network which helped to reduce the model size. The design strategy of this module is based on three main choices:

- the use of 1x1 filters to replace most of the 3x3 filters that are usually present in CNNs,
- decreasing the number of input channels with 3x3 filters,
- downsampling later in the network in order to have convolution layers with larger activation maps.

The first two choices are aiming to reduce the global number of parameters. The third point improves the classification accuracy due to the large activation maps induced by the 1x1 filters and the delay of the downsampling step [5]. Thereby, the *fire* module is composed of a *squeeze* convolution layer with only 1x1 filters followed by an *expand* layer incorporating a mix of 1x1 and 3x3 filters. The final SqueezeNet model is 50 times smaller than AlexNet while maintaining the same accuracy.

This architecture has been taken one step further by Nanafack et al. [82] to create the Squeeze-SegNet architecture, a deep fully convolutional neural network for pixel-wise semantic segmentation. This model is an encoder-decoder style network. The encoder part is similar to the SqueezeNet architecture while the decoder part is composed of inverted *fire* and convolutional layers proposed by the authors and inspired by the SqueezeNet architecture. Thus, the inverted *fire* module is called a *DFire* module, which is a series of alternating *expand* and *squeeze* modules. Both of these modules are retrieved from SqueezeNet. The downsampling stages are replaced by upsampling steps as the model needs to produce dense activation maps. Inspired by SegNet [83], the Squeeze-SegNet

model is able to get the same level of accuracy as SegNet [83] on a dataset like CamVid [84] with 10 times fewer parameters.

In 2012, Mamalet et al. [85] introduced the simplification of convolutional filters by using separable convolution layers. This work was improved by Howard et al. [86] in the MobileNet model. Inspired by the work of Chollet [87], the core layers of their architecture is based on depthwise separable features [88]. Stated differently, the convolution step is factorized into two separate steps to decrease the computation time taken by multiplication operations. Firstly, a depth-wise convolution applies a single filter to each input channel. Secondly, a point-wise convolution applies a 1x1 convolution in order to combine the outputs of the depth-wise convolution. This factorisation introduced in [88] drastically reduces the computational cost of the convolutions.

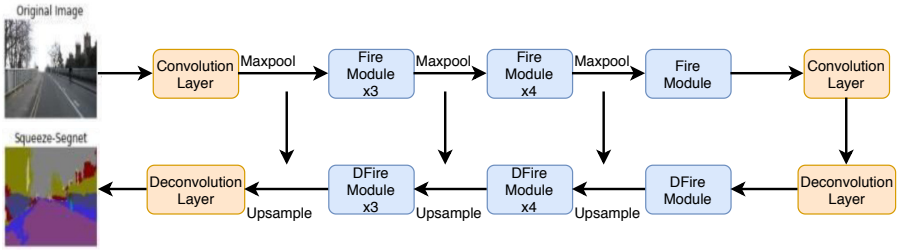

Fig. 6: Simplified architecture of the Squeeze-SegNet network. Figure inspired by [82].

Separable convolution layers have become an effective solution to accelerate convolution operations. Zhang et al. [89] also investigated this path with a neural network called ShuffleNet by adding to the depth-wise separable features a ShuffleNet unit. This unit allows the model to shuffle channels for group convolutions. Usually, each output channel is only related to a group of input channels. Here, we suppose a convolution layer with $g * n$ channels and g groups. The output channel dimension is first reshaped into (g, n) and then transposed and flattened as the input of the next layer. Compared to other models, the complexity is widely reduced. Compared to MobileNet [86], the efficiency and accuracy are slightly improved.

In section 2.6, we presented methods to binarize deep models. Inspired by this approach, Bulat et al. [76] developed a binary version of an architecture called the stacked hourglass network [75], a state of the art model in human pose estimation. The main contribution of the binarized model is to improve a bottleneck layer by limiting 1x1 filters and augmenting skip layers to limit the loss of binary information. On 3D face alignment, this model outperforms the current best performing methods up to 35%. However on human pose estimation tasks, the binary model is far behind the real-valued version. Thus there is still room for improvements on binary networks. It is important to note that an architecture can be changed and improved in order to use parameters

with limited numerical precision. Compression and architecture design are deeply intertwined.

3.2 Neural architecture search

These past few years, the understanding of deep networks has grown due to the development of new modules and architectures. However, knowing which model to use for a specific idea is still a difficult task. Tasks have become more challenging and to overcome them, the key is to find an architecture to fit them perfectly. But the more challenging is the task, the more difficult it is to design a network "by hand". Since constructing a proper architecture can be time-consuming, work has been done to study the possibility of letting networks automatically grow, adapt or even construct their own architectures. It is interesting to note that the first works in this field were oriented around physics and biology. Rosenblatt [90] Kohonen [91] or Willshaw and al. [92] were associating the organisation of the brain structure to the neural networks in order to find theoretical self-organising processes. Since then, numerous works on the subject have been done and they could be regrouped under the name of neural architecture search (NAS).

We give an overview of different methods in this field. We begin by introducing NAS with the early works in the domain regarding *neural gas*, followed by the *neuroevolution* methods, inpired by genetical algorithms, and the *network morphism* methods which aim to transform trained architectures. In these methods, the designed architectures are mostly optimized to obtain the best performance for a resulting task. However the size or memory consumption of these structures may not be optimized. Thus, in a last section we describes *supergraph* methods capable of finding structure optimized on these criteria.

3.2.1 Neural gas

Followed by these initial works, the *neural gas* methods, introduced by Martinetz and Schulten [93] were among the first approaches to push forward the idea of self-organized structures. they aimed to find an optimal data representation based on features vectors. In the beginning of the 90's, the works of Fritzke B. [94–97] studied the basis of self-organizing and incremental neural networks by enhancing the *neural gas* methods into growing structures. The authors mainly explored two ideas:

The first one, described in [97], was to develop an unsupervized learning approach for data visualisation, clustering and vector quantization to find a suitable architecture automatically. In this work, a neural network could be seen as a graph where a controlled growth process is applied. Furthermore, a supervized learning approach was also developed adding a radial basis function. This addition permitted, for the first time, to add new units and to supervise the training of the parameters at the same time. Moreover, the new

units were not added randomly anymore, leading to small networks that were able to generalise better. Their method was tested on vowel recognition problems and had better results than the nearest neighbour algorithm (the former state-of-the-art technique).

The second idea described by Fritzke B. in [94] is an extension of the first method based on a Hebb-like learning rule. As opposed to [93], the model has no parameters which change overtime but is still able of continuous learning until a performance criterion is met. From a theoretical point of view, these research works helped to better understand how the information is passed and transmitted inside a network.

3.2.2 Neuroevolution

Genetic algorithms are a well-known technique to find solutions of complex optimization problems. Adapted to deep networks, these methods were used to design evolutionary architectures and named *neuroevolution*. The basic statement of the evolutionary methods is as follows: an evolving topology along with weights should provide an advantage over evolving weights on a fixed topology. For decades, *neuroevolution* methods [98,99] were successfully applied on sequential decision tasks. Part of this success comes from the fact that sequential decision tasks are optimizing the weights of the neural networks instead of the gradient descent. Stanley et al. [100] went further ahead with a method called *NeuroEvolution of Augmenting Topologies* (NEAT). This technique minimises the dimensionality of the search space of connection weights, resulting in an efficient and rapid search of new topologies.

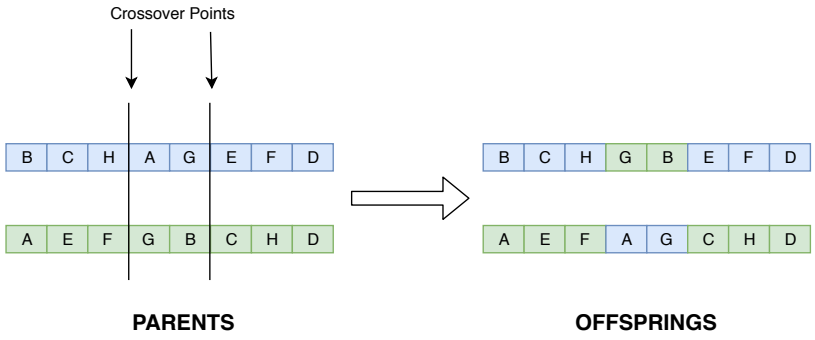

Fig. 7: Example of a crossover step. The two parent structures (left) are randomly decomposed at certain points and reconstructed to build offspring structures (right).

However, an important issue in *neuroevolution* is the permutation problem [101]. In evolutionary methods, there is more than one way to express a solution. In neural networks, it means that there is more than one architecture

to express the same weight optimization problem. The basis of evolutionary algorithms is to automatically design topologies that will compete with each others. The best topologies (or solutions) are mixed together to obtain an even better topology. This step is called a crossover (see Figure 7). These processes are repeated until the solution can not be improved anymore. During, these processes, the permutation problem is occurring when structures representing the same solution do not have the same encoding. Indeed, if two topologies with the same architecture but different encoding are mixed with each other, the resulting structure may lead to damaged structures and missing information (see Figure 8). Thereby, these algorithms are following conventions for fixed or constrained topologies such as non-redundant genetic encoding [102]. Stated differently, it becomes impossible to obtain two similar structures. Nonetheless, on neural networks where both weights and topologies are constantly evolving, these conventions may not be respected. Thus, using neural networks, the permutation problem is difficult to avoid. The work of Stanley et al. [100] has found one solution by keeping tracks of the history of the networks in order to define which parts must be shuffled without losing information. However, this method is memory-consuming due to the constant tracking of the networks. As a consequence NEAT model [100] is only successful with small networks.

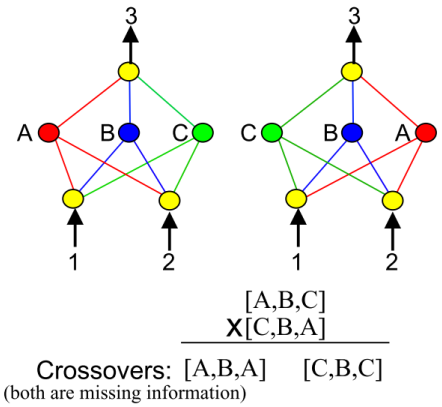

Fig. 8: The two networks compute the same solution, even if their units are appearing in a different order. This is making crossover impossible or one of the main unit will disappear. Figure from [100].

The NEAT model [100] is enhanced in a method called CoDeepNEAT [103]. This improvement consists of a coevolution of components, topologies and hyperparameters. Moreover, the evolution and optimization are based on the gradient, as opposed to previous methods where the optimization is based on the weights. On the CIFAR-10 image recognition dataset, CoDeepNEAT is able to automatically discover structures that have performances comparable to the state-of-the-art. Furthermore, on image captionning problem the

approach is able to find better structures than human design with enough computation time. These methods are able to find automatically adapted and efficient structure for a specific task. However the computational cost is expensive.

3.2.3 Network morphism

NAS is not only limited to *neuroevolution* methods. Network morphism [104–106] is also an important part of the domain. This approach aims to modify (to morph) a trained neural network into a new architecture. Thus, morphism operations are applied on the network *e.g* inserting a layer or adding a skip-connection. As a consequence, the main difficulty is to determine which operation should be applied. In [106], the authors use Bayesian optimization and select the most promising operations each time in the search space. The upside of this technique is that it does not need an important additional number of epochs to be operational. However, as the morphism operations are limited to a layer level, the topologies of the networks are constrained to be chain-structured. As most of the state-of-the-art networks are in multi-path structures, this is an noticeable limitation. Nonetheless, Cai et al. [107] were able to expand the search space to multi-path structures by allowing weight reusing and tree-structured architectures. As a result, expensive computational resources to achieve these searches are needed.

3.2.4 Supergraphs

Nowadays, an important number of NAS techniques and numerous other methods can be found in the literature. One of the principal issues that these methods are confronting with is the enormous size of the search space of possible solutions. Theoretically, this search space is infinite. Thereby, it is necessary to limit this space. In the methods described in the previous section, this limitation is mainly done by limiting and controlling the number of operations that could be done during the evolution of the networks. However an alternative approach would be not to limit the operations but the search space where they are operating. This is the idea behind *supergraphs*.
A *supergraph* is a large computational graph, where each subgraph is a neural network [108–110] (see Figure 9 for an example). Thus, the *supergraph* is becoming the search space and the solution to the task at hand will be one of its subgraphs. The benefit of this method is the reduced computational capacity needed as the solution is searched in a smaller space than in other methods.
In recent times the work of Pham et al.[109] have decreased the computation resources required to produce networks with strong performance. Their idea is as follows: all the subgraph (child models) produced by the supergraph are sharing parameters to avoid a constant retraining from scratch. We could argue that sharing parameters between different models could lead to numerous mistakes. However, transfer learning methods have shown that parameters learned for a specific task can be used by other models on other tasks [13,

111]. Thus, in principle the fact that child models are using their parameters for different purposes is not a problem.

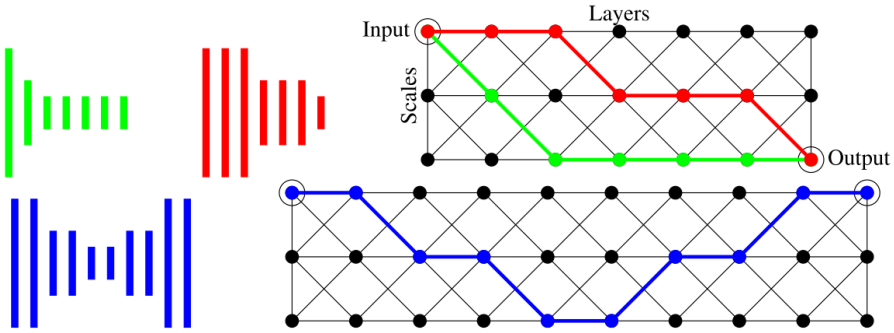

Fig. 9: Example of subgraph found via the convolutional neural fabrics method [108]. All edges are convolutionnal layers. All edges are oriented to the right. Feature map size of each layers are given by height. Thus, red and green are seven-layer convolutional layers and blue is a ten-layer convolutional-deconvolutional network. Figure from [108].

A different approach was developed by Veniat et al. [110] in their method called Budgeted Super Networks (BSN). Here, the *supergraph* is defining a set of possible architectures but a maximum authorized cost must also be defined. This cost is directly related to the performances, computation capability and memory consumption, allowing the authors to chose which criteria the algorithm must privilege during the search of the solution. To achieve this, a stochastic model is proposed and is optimized using policy-gradient-inspired methods. Moreover, the authors have also demonstrated that the solution of their stochastic model corresponds to the optimal constrained architecture of the *supergraph*. Tested on the CIFAR image classification and segmentation datasets, their solution was able to converge to design architectures similar to the ResNet model [79]. Furthermore, the designed models have the same computational cost or memory consumption than the original ResNet but with a better accuracy.

Several conclusion can be drawn from these experiments. Firstly, constraining the search of optimized architectures leads to a reduction of the computational capability needed for this search. Secondly, it also permits to obtain networks that have a limited computation and memory cost with negligible compromise on the accuracy.

Tan et al. [112] have pushed their research in this direction. The authors proposed an automated NAS specifically for mobile devices called MnasNet. In their work the accuracy and the inference latency of the model on a mobile device are both taken into account for the optimization of the model. Thus, the model is trying to find the best trade-off between these two criterias. In

order to reach this balance, two different hierarchical search spaces are used: one to factorise a deep network into a sequence of blocks and another one to determine the layer architecture for each block. Thereby, different layers are able to use different operations but all the layers in one block are sharing the same structure. The adapted block is chosen at different depth of the network to reduce the overall latency. These improvements allow to reduce the search space while finding architecture that are performing better and faster than MobileNets [86, 113].

Technique	Method	Pros	Cons
Knowledge distillation	Using a deep CNN to train a smaller CNN.	Small models with comparable performances.	Models can only be trained from scratch; Difficult for the tasks other than classification.
Hashing	Indexing neurons into a hash table.	Better parallelization; Better data dispersion; Less computation time.	Considerably slower if the model is too sparse.
Pruning	Deleting neurons that have minor influence on the performance.	Significant speed up and size reduction; Compression rate is 10x to 15x (up to 30x).	Pruning process is time-consuming; Less interesting for too sparse model.
Quantization	Reducing the number of distinct neurons by gathering them into clusters.	High compression rate : 10x to 15x; Can be coupled with pruning.	Considerably slower if the model is too sparse.
Numerical Precision	Decreasing the numerical precision of the neurons.	High compression rate and speed up.	Higher precision is needed during the parameters update; Could require specific hardwares.
Binarization	Decreasing the numerical precision of the data to 2 bits.	Very high compression rate (30x) and speed up (50x to 60x).	Higher precision is needed during the parameters update.

Table 1: Summary of different compression methods.

4 Discussion and Conclusion

In Table 1, we summarized and compared various deep-learning model compression methods from the literature discussed in this paper. These methods aim to reduce the size, computation time or the memory employed by deep models. However, a sparse model may not always be computationally efficient. Pruning and quantization can be utilized to achieve impressive performances on trained models. However, they can easily lead to sparse model (same problem for hashing methods). In this case, binarization or reducing the numerical

precision method can be one of the solutions. The speed gained for limiting the numerical precision is important, especially if the structures is well designed. Nevertheless, higher precision is needed in some steps and accuracy could vary significantly. In the end, compressing a deep model will always lead to a trade-off between accuracy and computational efficiency.

Faster models provide a great benefit for resource-limited devices and further work needs to be done in this direction if we want to leverage all of their power on mobile devices. However, finding new methods to compress deep models is not the only solution. We can focus on how the models are constructed beforehand. For example a simple architecture like SqueezeNet [81] is able to reach the same accuracy as a deep model like AlexNet [1], but is 50 times smaller.

Compared to the size of the model, computational efficiency is crucial for running such algorithms on mobile platforms. Despite the effort on hardware optimization, algorithmic optimizations like [85] and recent works such as Mobile-Net[86] and Shuffle-Net[89] have shown that it is promising to not only compress models but also to construct them intelligently. Thus a well-designed architecture is the first key to optimized networks.

The works on NAS design an optimized architecture (performance-wise and computational efficiency-wise) for a specific tasks. Though this is a challenging exercise, some research works have already shown promising results through new algorithms and theories like the *lottery ticket hypothesis* [114]. All these works are pushing forward the understanding of the mechanics behind deep models and towards building optimized models capable of solving challenging applications at a lower cost.

Acknowledgements: This work has been sponsored by the Auvergne Regional Council and the European funds of regional development (FEDER).

References

1. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *NIPS*, pp. 1097–1105, 2012.
2. Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, pp. 436–444, 2015.
3. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale Image recognition," *CoRR*, pp. 1–14, 2015.
4. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CVPR*, pp. 1–9, 2015.
5. K. He and J. Sun, "Convolutional neural networks at constrained time cost," pp. 5353–5360, 2015.
6. H. Chuangxia, K. Hanfeng, C. Xiaohong, and W. Fenghua, "An lmi approach for dynamics of switched cellular neural networks with mixed delays," *Abstract and Applied Analysis*, 2013.
7. H. Chuangxia, C. Jie, and W. Peng, "Attractor and boundedness of switched stochastic cohen-grossberg neural networks," *Discrete Dynamics in Nature and Society*, 2016.
8. Y. Cheng, D. Wang, P. Zhou, T. Zhang, and S. Member, "A Survey of Model Compression and Acceleration for Deep Neural Networks," *IEEE Signal Processing Magazine*, 2018.

9. J. Cheng, P. Wang, G. Li, Q. Hu, and H. Lu, "Recent Advances in Efficient Computation of Deep Convolutional Neural Networks," *Frontiers of Information Technology & Electronic Engineering*, 2018.
10. Y. N. Dauphin and Y. Bengio, "Big neural networks waste capacity," 2013.
11. J. Ba and R. Caruana, "Do deep nets really need to be deep?," *NIPS*, pp. 2654–2662, 2014.
12. C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," pp. 535–541, 2006.
13. G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *NIPS 2014 Deep Learning Workshop*, pp. 1–9, 2014.
14. Y. Chen, N. Wang, and Z. Zhang, "Darkrank: Accelerating deep metric learning via cross sample similarities transfer," 2017.
15. Z. Huang and N. Wang, "Like what you like: Knowledge distill via neuron selectivity transfer," 2017.
16. G. Aguilar, Y. Ling, Y. Zhang, B. Yao, X. Fan, and C. Guo, "Knowledge distillation from internal representations," 2020.
17. H. Lee, S. J. Hwang, and J. Shin, "Self-supervised label augmentation via input transformations," *ICML*, 2020.
18. R. Müller, S. Kornblith, and G. Hinton, "When does label smoothing help?," *NIPS*, 2019.
19. K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," pp. 1113–1120, 2009.
20. W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," pp. 2285–2294, 2015.
21. R. Spring and A. Shrivastava, "Scalable and sustainable deep learning via randomized hashing," pp. 445–454, 2017.
22. J. Ba and B. Frey, "Adaptive dropout for training deep neural networks," pp. 3084–3092, 2013.
23. A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 518–529, 1999.
24. R. Shinde, A. Goel, P. Gupta, and D. Dutta, "Similarity search and locality sensitive hashing using ternary content addressable memories," pp. 375–386, 2010.
25. N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey, "Streaming similarity search over one billion tweets using parallel locality-sensitive hashing," *Proceedings of the VLDB Endowment*, pp. 1930–1941, 2013.
26. Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, pp. 1–12, 2015.
27. A. Shrivastava and P. Li, "Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips)," pp. 2321–2329, 2014.
28. Y. L. Cun, J. S. Denker, and S. a. Solla, "Optimal Brain Damage," *Advances in Neural Information Processing Systems*, pp. 598–605, 1990.
29. B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," pp. 164–171, 1993.
30. S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *NIPS*, pp. 1135–1143, 2015.
31. P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *ICLR*, 2017.
32. S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems*, 2017.
33. W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *NIPS*, p. 2082–2090, 2016.
34. H. Zhou, J. M. Alvarez, and F. Porikli, *Less Is More: Towards Compact CNNs*, pp. 662–677. Cham: Springer International Publishing, 2016.
35. J. M. Alvarez and M. Salzmann, "Learning the number of neurons in deep networks," pp. 2270–2278, 2016.
36. V. Lebedev and V. Lempitsky, "Fast convnets using group-wise brain damage," pp. 2554–2564, 2016.

37. H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," *ICLR*, pp. 1–10, 2017.
38. J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *ICCV*, 2017.
39. Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning Efficient Convolutional Networks through Network Slimming," *ICCV*, pp. 2736–2744, 2017.
40. S. Han, H. Mao, and W. J. Dally, "Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *ICLR*, pp. 1–13, 2016.
41. 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," *CVPR*, 2018.
42. Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *Advances in Neural Information Processing Systems 31*, pp. 875–886, 2018.
43. Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," *CVPR*, 2019.
44. S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. S. Doermann, "Towards optimal structured CNN pruning via generative adversarial learning," *CVPR*, pp. 2790–2799, 2019.
45. T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets," *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6655–6659, 2013.
46. T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, pp. 455–500, 2009.
47. Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *arXiv preprint arXiv:1710.09282*, 2017.
48. J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," pp. 2178–2188, 2017.
49. Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," 2014.
50. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, pp. 2278–2324, 1998.
51. Y. Choi, M. El-Khamy, and J. Lee, "Towards the limit of network quantization," *ICLR*, 2017.
52. D. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.
53. J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, pp. 2121–2159, 2011.
54. M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," p. 6, 2012.
55. G. E. Hinton, N. Srivastava, and K. Swersky, "Lecture 6a- overview of mini-batch gradient descent," *COURSERA: Neural Networks for Machine Learning*, p. 31, 2012.
56. M. Abadi, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.
57. J. Ahmad, J. Beers, M. Ciurus, R. Critz, M. Katz, A. Pereira, M. Pringle, and J. Rames, *iOS 11 by Tutorials: Learning the New iOS APIs with Swift 4*. Razeware LLC, 1st ed., 2017.
58. C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. Lecun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2011.
59. V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 696–701, 2014.
60. 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.
61. A. Iwata, Y. Yoshida, S. Matsuda, Y. Sato, and Y. Suzumura, "An artificial neural network accelerator using general purpose 24 bit floating point digital signal processors," *Proc. IJCNN*, pp. 171–175, 1989.

62. D. Hammerstrom, "A VLSI architecture for high-performance, low-cost, on-chip learning," *IJCNN International Joint Conference on Neural Networks*, pp. 537–544, 1990.
63. J. L. Holt and J. N. Hwang, "Finite precision error analysis of neural network hardware implementations," *IEEE Transactions on Computers*, pp. 281–290, 1993.
64. S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," *ICML*, pp. 1737–1746, 2015.
65. M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *ICLR*, 2014.
66. D. Williamson, "Dynamically scaled fixed point arithmetic," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*, pp. 315–318, 1991.
67. F. Mamalet, S. Roux, and C. Garcia, "Real-time video convolutional face finder on embedded platforms," *Eurasip Journal on Embedded Systems*, 2007.
68. S. Roux, F. Mamalet, C. Garcia, and S. Duffner, "An embedded robust facial feature detector," *Proceedings of the 2007 IEEE Signal Processing Society Workshop, MLSP*, pp. 170–175, 2007.
69. M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," 2016.
70. X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," pp. 344–352, 2017.
71. N. Srivastava, "Improving Neural Networks with Dropout," *Master's thesis*, 2013.
72. N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of machine learning research*, pp. 1929–1958, 2014.
73. M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," pp. 3123–3131, 2015.
74. M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," pp. 525–542, 2016.
75. A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," pp. 483–499, 2016.
76. A. Bulat and G. Tzimiropoulos, "Binarized convolutional landmark localizers for human pose estimation and face alignment with limited resources," Oct 2017.
77. L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "Gated xnor networks: Deep neural networks with ternary weights and activations under a unified discretization framework," 2017.
78. Y. LeCun, "Generalization and network design strategies," *Connections in Perspective. North-Holland, Amsterdam*, pp. 143–155, 1989.
79. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CVPR*, pp. 770–778, 2016.
80. G. Huang, Z. Liu, K. Q. Weinberger, and L. V. D. Maaten, "Densely connected convolutional networks," *CVPR*, 2017.
81. 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< 0.5 mb model size," 2016.
82. G. Nanfack, A. Elhassouny, and R. O. H. Thami, "Squeeze-segnet: A new fast deep convolutional neural network for semantic segmentation," *CoRR*, 2017.
83. V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.
84. G. Brostow, J. Shotton, J. Fauqueur, and R. Cipolla, "Segmentation and recognition using structure from motion point clouds," 2008.
85. F. Mamalet and C. Garcia, "Simplifying convnets for fast learning," *ICANN 2012*, pp. 58–65, 2012.
86. 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," 2017.
87. F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *arXiv preprint arXiv:1610.02357*, 2016.

88. L. Sifre and M. Stephane, "Rigid-Motion Scattering For Image Classification," *CoRR*, 2014.
89. X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," 2017.
90. F. Rosenblatt, "Perceptrons and the Theory of Brain Mechanics," p. 621, 1962.
91. T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, no. 1, pp. 59–69, 1982.
92. "How patterned neural connections can be set up by self-organization," *Proceedings of the Royal Society of London. Series B, Biological Sciences*, vol. 194, no. 1117, pp. 431–445, 1976.
93. T. M. Martinetz, S. G. Berkovich, and K. J. Schulten, ""Neural-Gas" Network for Vector Quantization and its Application to Time-Series Prediction," 1993.
94. B. Fritzke, "A Growing Neural Gas Learns Topologies," *Advances in Neural Information Processing Systems*, vol. 7, pp. 625–632, 1995.
95. B. Fritzke, "Supervised Learning with Growing Cell Structures," *Advances in Neural Information Processing Systems 6*, no. 1989, pp. 255–262, 1994.
96. B. Fritzke and R.-u. Bochum, "Fast learning with incremental RBF Networks 1 Introduction 2 Model description," *Processing*, vol. 1, no. 1, pp. 2–5, 1994.
97. B. Fritzke, "Growing cell structures-A self-organizing network for unsupervised and supervised learning," *Neural Networks*, vol. 7, no. 9, pp. 1441–1460, 1994.
98. D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 762–767, 1989.
99. D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, 2008.
100. K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, 2002.
101. N. J. Radcliffe, "Genetic set recombination and its application to neural network topology optimisation," *Neural Computing & Applications*, vol. 1, no. 1, pp. 67–90, 1993.
102. D. Thierens, "Non-redundant genetic coding of neural networks," pp. 571–575, 1996.
103. R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving Deep Neural Networks," 2017.
104. T. Elsken, J. H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," 2018.
105. H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient Architecture Search by Network Transformation," pp. 2787–2794, 2018.
106. H. Jin, Q. Song, and X. Hu, "Efficient Neural Architecture Search with Network Morphism," 2018.
107. H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," in *Proceedings of the 35th International Conference on Machine Learning*, pp. 678–687, 2018.
108. S. Saxena and J. Verbeek, "Convolutional Neural Fabrics," *NIPS 2016*, 2016.
109. H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proceedings of the 35th International Conference on Machine Learning*, vol. 80, 2018.
110. T. Veniat and L. Denoyer, "Learning time/memory-efficient deep architectures with budgeted super networks," in *Conference on Computer Vision and Pattern Recognition*, pp. 3492–3500, 2018.
111. B. Zoph, D. Yuret, J. May, and K. Knight, "Transfer Learning for Low-Resource Neural Machine Translation," 2016.
112. M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," *CVPR*, 2019.
113. M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation," 2018.
114. J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *ICLR*, 2019.