Pruning Deep Convolutional Neural Networks Architectures with Evolution Strategy

Francisco E. Fernandes Jr.[a,1], Gary G. Yen[b,*]

[a]*Institute of Mathematical Sciences and Computing, University of São Paulo, São Carlos, SP, 13566-590, Brazil*
[b]*School of Electrical and Computer Engineering, Oklahoma State University, Stillwater, OK, 74078, USA*

Abstract

Currently, Deep Convolutional Neural Networks (DCNNs) are used to solve all kinds of problems in the field of machine learning and artificial intelligence due to their learning and adaptation capabilities. However, most successful DCNN models have a high computational complexity making them difficult to deploy on mobile or embedded platforms. This problem has prompted many researchers to develop algorithms and approaches to help reduce the computational complexity of such models. One of them is called filter pruning, where convolution filters are eliminated to reduce the number of parameters and, consequently, the computational complexity of the given model. In the present work, we propose a novel algorithm to perform filter pruning by using Multi-Objective Evolution Strategy (ES) algorithm, called DeepPruningES. Our approach avoids the need for using any knowledge during the pruning procedure and helps decision-makers by returning three pruned CNN models with different trade-offs between performance and computational complexity. We show that DeepPruningES can significantly reduce a model's computational complexity by testing it on three DCNN architectures: Convolutional Neural Networks (CNNs), Residual Neural Networks (ResNets), and Densely Connected Neural Networks (DenseNets).

Keywords: Deep Convolutional Neural Networks, Convolutional Filter Pruning, Multi-Objective Optimization, Evolution Strategies, Evolutionary Algorithm.

1. Introduction

1.1. Overview of the Problem

Deep Convolutional Neural Networks (DCNNs) are among the most popular machine learning models used for computer vision applications at the moment.

---

[*]Corresponding author
Email address: gyen@okstate.edu (Gary G. Yen)
[1]This work was completed when the author was with the Oklahoma State University.

Although the basic theory behind DCNNs was developed in the late 1980s [1], only recently, they were able to surpass humans in such tasks [2, 3]. Mainly due to recent advancements in big data analytics and general-purpose computing on graphics processing units (GPUs).

DCNNs are data and model-dependent. In order to achieve good classification results, DCNNs need to be designed with millions of nodes, also called neurons, in its structure, requiring a massive amount of computational power to use them. Training such large DCNNs with small amounts of data can lead to overfitting problems where the model learns the training data very well, but it is unable to generalize its knowledge to unseen data. For example, *ImageNet*, one of the most popular databases used for large-scale image classification, is composed of 14 million images distributed in $1,000$ classes [4, 5]. A typical DCNN capable of classifying such large databases may contain a few hundred million parameters. For instance, VGG-16, one of the top performer models in the ImageNet database in 2014, has 138 million parameters [6] and computational complexity of 15.5×10^9 floating-point operations (15.5 GFLOPS) [7].

The task of finding a good DCNN architecture for a given problem, called Neural Architecture Search (NAS), is typically achieved by trial and error, and it can take months to find a suitable model depending on the amount of data and computing power available. Hence, in recent years, researchers have been using Evolutionary Computation (EC) and Swarm Intelligence (SI) tools to devise algorithms capable of automatically finding DCNN architectures and with little to no prior knowledge of the problem at hand [8]. For example, Genetic Algorithms (GAs) were used with success to search for DCNN architectures by Yanan *et al.* [9, 10], Martín *et al.* [11], and many others [12, 13]. Particle Swarm Optimization (PSO), a particular SI algorithm, and Evolution Strategy (ES) were also successfully applied to the same task [14, 15, 16].

However, most NAS algorithms are designed to find DCNN architectures with the best classification accuracy in a given problem. Thus, in general, DCNNs used for image recognition and classification tasks still require powerful hardware such as GPU workstations and datacenters for training and inference. Consequently, these models are not suitable for consumer hardware, such as smartphones and portable appliances. Currently, a constant connection to the internet is required if a developer is interested in performing computer vision tasks in such devices, which may not always be available, or may not be reliable to perform such tasks seamlessly to users. Thus, the development of algorithms and techniques that would reduce the computational complexity of DCNN models is highly desirable.

1.2. Motivation

Machine Learning researchers are increasingly interested in finding novel ways of reducing the computational complexity of DCNNs while maintaining their performance. There are mainly three approaches to achieve this goal: (1) designing DCNNs architectures from the ground up taking into consideration the limitations of the target platform; (2) making use of specialized mobile hardware capable of running DCNNs efficiently; and (3) modifying existing

DCNNs architectures by removing redundant parameters. Each one of these techniques is detailed in the following.

In the first approach, novel DCNN operations and architectures can be created to take advantage of the particularities of mobile and embedded platforms. For instance, CondenseNets [17] and MobileNets [18] are two specialized DCNN architectures created to be used on mobile platforms, such as smartphones. Both architectures make use of specialized convolutional operations, i.e., group and depth-wise convolutions, which decreases their computational complexity without degrading their performance. Likewise, NAS can also be used to find DCNN architectures suitable for mobile and embedded devices. The Differentiable Architecture Search (DARTS) algorithm developed by Liu *et al.* is one example of a NAS algorithm capable of generating compact DCNNs by using a continuous search space [19]. The NetAdapt algorithm developed by Yang *et al.* is another classic example capable of generating suitable DCNNs architectures given a certain resource budget. NetAdapt can find specific DCNN architectures tailored to an individual device [20]. Another example is the FastDeepIoT algorithm developed by Yao *et al.*, where an initial highly accurate model is found, and then the algorithm compresses the model by taking into consideration the execution time in a specific mobile and embedded device [21].

The second approach to deal with the computational requirements of DCNN models is to develop specialized mobile and embedded hardware. One of the most iconic examples is the *Jetson* embedded platforms developed by Nvidia, which features a GPU capable of performing inference in real-time using complex DCNN models [22, 23]. Another area of intense interest is the use of Field-Programmable Gate Array (FPGA) for DCNN model inference. Some researchers have adapted deep learning frameworks and libraries to work with FPGA platforms resulting in inference time comparable with the ones from applying GPUs [24, 25]. However, the use of such platforms does not address the problem of how to reduce the computational complexity of DCNN models. In that sense, Ding *et al.* developed a method to reduce the computational complexity of Convolutional Neural Network (CNN) operations by using block-circulant weight matrices for implementation in FPGAs [26]. In their work, the weight matrices of each layer are converted to block-circulant matrices, and inference operations are performed using Fast Fourier Transform multiplications resulting in reduced computational complexity for inference and memory use.

Pruning is considered a third approach to reduce the computational complexity of DCNN models. In this case, the hypothesis is that many DCNN models handcrafted by humans have too many unnecessary parameters, and the identification and elimination of such parameters can reduce the model complexity without degrading its performance. Ding *et al.* developed a technique called Auto-balanced Filter Pruning, where entire convolutional filters[2] can be removed from multiple layers based on their importance by using $l2$ regulariza-

---

[2]In the present work, we refer to *convolutional filters* or *filters* as the trainable weights of a convolutional layer, while *feature maps* refer to the output tensor of a convolutional layer.

4

tion [27]. However, it requires that the initial DCNN model be trained with strong regularization, where weak filters are regularized with positive factors, while strong ones are regularized with negative factors. This initial training phase makes it easier to identify the filters to be removed during the pruning process, but it also requires the retraining of the original DCNN model used for pruning, which can take a long time. Luo *et al.* [28] also performed pruning of DCNN models by removing entire filters from multiple layers. Their method is called ThiNet, and it uses statistics information from the next layer instead of the information from the current layer being pruned. The method developed by Li *et al.* eliminates entire filters by computing the sum of the absolute values of the filters in a given layer, and removing the ones with the smallest sum [29]. Pruning has also been achieved with the use of *Evolutionary Algorithms* (EAs). The works developed by Wang *et al.* [30] and Zhou *et al.* [31] are two recent examples in which EAs are used to prune DCNN models with competitive results.

It is also worth mentioning other approaches used to deal with the high computational demands of DCNN models. For example, it is possible to perform weight quantization, reducing the number of bits needed to represents the network's weights [32]. An extreme form of weight quantization is used in binarized networks in which all weights are represented using a single bit [33, 34]. Another approach is to design DCNN models that compute the residual mapping of a function, also known as a shortcut or a skip connection, used by Deep Residual Networks (ResNets) [35]. The residual mapping avoids the problem of the vanishing gradient when creating models with hundreds or thousands of layers. Residual mapping does not directly reduce the computational complexity of a DCNN, but it helps in achieving high performance by using convolutional layers with smaller filter sizes. Densely Connected Convolutional Networks (DenseNets) [36] also use shortcut connections from previous layers, but, instead of using the output of only a single layer as in a ResNet, it uses the output from all previous layers as the input to the current one. In other words, each convolution layer in a DenseNet is connected to all subsequent ones in a fully-connected fashion. Similar to ResNets, the use of multiple shortcut connections between layers does not decrease the computational complexity of the network, but it allows the use of smaller networks with higher performance when compared with DCNNs with no shortcut connections.

We believe that all the works in building DCNN models with the best performance possible should not be discarded when trying to find models with reduced computational complexity. In this paper, we leverage such works by proposing a novel pruning strategy to eliminate unnecessary parameters from existing top-performing models. Specifically, we propose the elimination of redundant filters from convolutional layers to reduce the model's computational complexity. This approach is considered the most suitable one for most applications because it eliminates the need to have specialized hardware to run such models. It also gives us some insight into the structure of DCNNs in general and how most models could be created using fewer parameters. In this sense, the pruning of a DCNN model can be seen as a combinatorial optimization where we try to find

5

the optimal combination of convolutional filters in different layers from a given model. It can also be formulated as a multi-objective optimization problem where we have two conflicting objectives: (1) reduce the number of parameters in a given model as much as possible while (2) keeping the model's performance as high accuracy as possible. We are also interested in prune the models without relying on any prior knowledge about them, allowing us to prune many different DCNN architectures with a single algorithm. The use of *Evolutionary Computation* (EC) algorithms is ideal for this case. Such algorithms are well suitable for combinatorial optimization with conflicting objectives. Specifically, the *Evolution Strategy* (ES) [37] is one such algorithm where a population of candidate solutions is evolved by performing random perturbations in their parameters.

1.3. Contributions

The main objective of this paper is to propose the use of the *Evolution Strategy* for pruning DCNN models, which we call *DeepPruningES*. We chose to use ES because of its simplicity, but powerful ability to find reasonable solutions. We highlight that we are interested in finding reasonable solutions instead of optimal ones. Thus, when pruning DCNN models, we can only find solutions with some trade-off between computational complexity and performance. The best solutions are in the so-called Pareto frontier, where no solution is strictly better than the others. However, the task of finding the true Pareto frontier for a given model pruning task is not trivial. Hence, the following is a list of contributions of the proposed *DeepPruningES* algorithm:

- The task of pruning DCNN architectures is conceived as a two-objective optimization problem and solved using an Evolution Strategy (ES), where the training error and the computational complexity are chosen to be two competing objectives. Furthermore, the proposed algorithm performs a gradual pruning of a chosen DCNN model, where filters are removed from the chosen DCNN model over the generations.

- The proposed algorithm provides three solutions with different trade-offs at the end of the pruning procedure and avoids the need to find the true Pareto frontier. In this regard, a Decision-Maker (DM) can choose the best solutions that fit his/her needs. These solutions are the following:

  - The candidate solution with the smallest training error in the population called the *Boundary Heavy* solution;
  - The candidate solution with the smallest computational complexity called the *Boundary Light* solution;
  - The candidate solution with the best trade-off between training error and computational complexity called the *Knee* solution.

- A modified ES selection operator based on the *Minimum Manhattan Distance* (MMD) approach [38] is proposed to choose these three solutions from a population of candidate solutions. With this approach, the proposed algorithm does not require the use of any trade-off parameter.

 6

The proposed algorithm is also distinct from others' pruning algorithms with evolutionary approaches, as explained in the following:

- Wang *et al.* [30] developed an evolutionary algorithm that, at first glance, looks similar to the proposed one. However, the algorithm in [30] does not incorporate a Multi-Criteria Decision Making (MCDM) framework, and can only find a single solution with a very small computational complexity regardless of classification accuracy, while the proposed algorithm can find three solutions with different trade-offs between computational complexity and classification performance. In the proposed algorithm, the DM is not required to set up any trade-off parameter as in [30]. The proposed pruning algorithm is developed as a Multi-Objective Optimization (MOP) algorithm in which two conflicting objective functions are optimized at the same time, while the algorithm presented in [30] uses a single objective function. Moreover, the authors in [30] do not make it clear if their algorithm is capable of pruning all layers of a ResNet or just layers between the shortcut connections. On the contrary, the algorithm presented in this work can prune all layers of ResNet and DenseNets by using two binary strings to encode their filters. Thus, we can prune the entirety of a ResNet or DenseNet model, not just its intermediate layers.

- At first glance, another algorithm that looks similar to the proposed one is the Knee-Guided Evolutionary Algorithm (KGEA) for Compressing Deep Neural Networks developed by Zhou *et al.* [31]. The first novelty of the proposed algorithm compared with KGEA is that KGEA can only prune simple CNN models, such as VGG networks [6], while the proposed *DeepPruningES* can prune multiple DCNN topologies, such as VGGs, ResNets, and DenseNets. Another difference is that KGEA tries to maintain a diverse Pareto front by using a mechanism to ensure diversity, while in *DeepPruningES*, we are interested in finding good candidate solutions as fast as possible. Thus, *DeepPruningES* does not require a diverse Pareto front to be maintained during the pruning process. Furthermore, because we are interested in preserving only three candidate solutions, using an ES framework is better suited for the problem than a GA one as does in [31], which also results in a population with a smaller number of individuals than others' evolutionary approaches.

The proposed algorithm can prune Convolutional Neural Networks (CNNs), Residual Neural Networks (ResNets), and Densely Connected Neural Networks (DenseNets), which, to the best of our knowledge, no other algorithm proposed in the literature is capable of pruning. In general, it can achieve up to a 75% reduction in the number of Floating-Point Operations (FLOPs) with little degradation to the model's classification performance.

The remaining sections of the present work are organized as follows. Detailed background about Deep Neural Networks models, such as CNN, ResNet, and DenseNet, Convolutional Filter Pruning, Evolution Strategies, is given in Section II. The proposed algorithm is presented in Section III. The experimental

7

design used to validate the algorithm is discussed in Section IV. The experimental results are analyzed in Section V, followed by the conclusions and future work in Section VI.

2. Background

2.1. Deep Convolutional Neural Networks

A Deep Neural Network (DNN) is a computational model inspired by the way animal's brains work. Its primary computational node is known as an artificial neuron, which is composed of a set of weights and a transfer function. To compute the output of a neuron, first, a weight operation is performed with its inputs and weights. Then, the result is passed through a non-linear function, known as activation function, to produce an output which can be used as inputs to subsequent neurons in a given DNN. In this sense, multiples neurons stacked together constitute a layer, and multiple layers stacked together constitute the architecture of a DNN. A Deep Convolutional Neural Network (DCNN) is, then, defined here as a DNN containing one or more layers using convolutional operations as their weight operations. Mathematically, a DCNN is defined as in Eq. 1, where i represents the layer number, $\mathbf{X}$ is a tensor representing the inputs of the entire DCNN, $\mathbf{O}_i$ is a tensor representing the output of layer i, $f_i(.)$ represents the activation function used in layer i, and $\mathbf{Z}_i$ is the output of the weight operation, $g_i(.)$, between the output from the previous layer, $\mathbf{O}_{i-1}$, and the weights of the current layer, $\mathbf{W}_i$. The three most commonly used weight operations are defined in Eq. 2, and they are convolution, $\circledast$, max or average pooling, and matrix multiplication, used in fully-connected layers.

$$\begin{cases} \mathbf{O}_i = \mathbf{X} & i = 1 \\ \mathbf{O}_i = f_i(\mathbf{Z}_i) & i > 1 \\ \mathbf{Z}_i = g_i(\mathbf{O}_{i-1}, \mathbf{W}_i) \end{cases} \tag{1}$$

$$\begin{cases} \mathbf{Z}_i = \mathbf{W}_i \circledast \mathbf{O}_{i-1} & \text{if } i\text{-th layer is convolution} \\ \mathbf{Z}_i = Pool(\mathbf{O}_{i-1}) & \text{if } i\text{-th layer is pooling} \\ \mathbf{Z}_i = \mathbf{W}_i \cdot \mathbf{O}_{i-1} & \text{if } i\text{-th layer is fully-connected} \end{cases} \tag{2}$$

The convolutional layer was developed by LeCun *et al.* in 1989 [1], and allows DCNNs to go deeper due to its parameter sharing nature which reduces the total number of parameters when compared with an equivalent network constructed using only fully-connected layers. The pooling layer further decreases the number of parameters in a DCNN by subsampling its inputs with a max or average operation. Deep Convolutional Neural Networks (DCNNs) are constructed basically by stacking multiples convolution, pooling and fully-connected layers.

In the case of Residual Neural Networks (ResNets) [35], their architecture is composed of stacked blocks, called residual blocks, which is a combination of two convolution layers with one shortcut connection. If we define the output
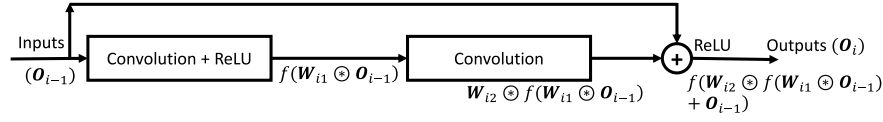
                                8
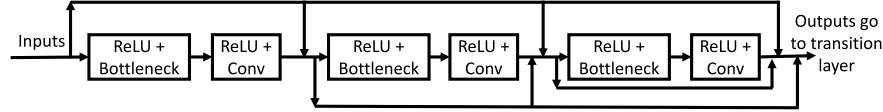
Figure 1: Example of residual block.

Figure 2: Example of a dense block with three bottleneck layers.

of the previous residual block as $\mathbf{O}_{i-1}$, we can define the output of the current block, $\mathbf{O}_i$, as in Eq. 3, where $f(.)$ is a rectified activation function, ReLU, i.e., $f(x) = max(0, x)$, $\mathbf{W}_{i1}$ are the weights of the first convolution layer, and $\mathbf{W}_{i2}$ are the weights of the second convolution layer. The visualization of such block can be seen in Fig. 1.

$$\mathbf{O}_i = f(\mathbf{W}_{i2} \circledast f(\mathbf{W}_{i1} \circledast \mathbf{O}_{i-1}) + \mathbf{O}_{i-1}) \tag{3}$$

Densely Connected Neural Networks (DenseNets) [36] are also composed of blocks, called *Dense Blocks*, which consist of multiples convolution layers. Their main difference compared with ResNets is that the outputs of each convolution layer in a given *Dense Block* is used as input for all subsequent convolution layers from the same *Dense Block*. The number of output feature maps from any given convolution layer is called *growth rate*, k. Because the outputs from previous layers are concatenate and used as the input of the current layer, the number of input feature maps in layer i is equal to $k_0 + k \times (i - 1)$, where k_0 is the number of output feature maps from the first layer. Furthermore, in order to improve computational efficiency, each layer in a *Dense Block* is composed of a bottleneck layer followed by a convolution layer. A bottleneck layer is simply a convolution layer with 1×1 convolution filters which receives all the inputs from previous layers. A transition layer is used between each *Dense Block* with the purpose of reducing the number of inputs to the next block. The transition layer is normally built by using a pooling layer followed by 1×1 convolution layer. Fig. 2 illustrates the connections between layers in a *Dense Block* with three bottleneck layers.

2.2. Convolutional Filter Pruning

The weights of a convolution layer, also known as filters or kernels, can be represented as a four dimensional tensor $\mathbf{K}$ with dimensions equal to $(O \times D \times W \times H)$, where O is the total number of kernels with size $(D \times W \times H)$, D is the depth of each kernel, W is the width of each kernel, and H is the height of each kernel. Optionally, each one of the O kernels can have a single parameter to represent its bias. If one would like to reduce the computational complexity
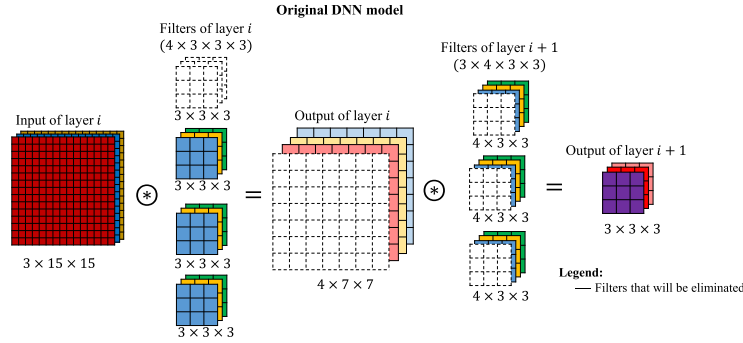
Figure 3: Example of convolutional filter pruning where the first filter of the current convolution layer is selected to be eliminated.
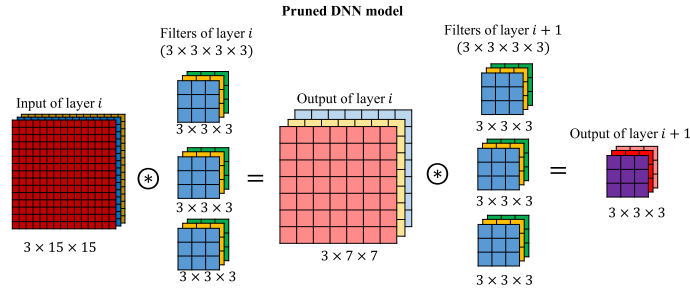

Figure 4: Pruned DCNN model after the first filter is removed.

of a single convolution layer, one approach would be to eliminate some of these O kernels in their entirety. This approach is indeed one of the most common techniques used to reduce the computational complexity of DCNN models, and it is known as filter pruning. This process is illustrated in Figs. 3 and 4 where a filter is first chosen to be eliminated, indicated by the dashed lines in Fig. 3, and, then, the entire DCNN model is adjusted resulting in a pruned model, shown in Fig. 4. It is important to note that when filters are eliminated in layer i, the corresponding depths of all filters in the subsequent layer $i + 1$ need to be removed to produce a valid architecture.

In the literature, there are multiple ways of choosing which filters to eliminate in each convolution layer, a procedure known as *filter selection*. In general, some statistics from the current layer is needed to determine which filters to eliminate. The following are some of the most used decision criteria found in the literature [39]:

- *Mean activation*: It was used by Molchanov *et al.* in [40] where the mean activation of each filter from a given layer is computed, and only the filters with the highest mean activation are kept.

- $l1$-norm: It was used by Li *et al.* in [29] where the importance of a filter

is determined by its $l1$-norm. Filters with smaller $l1$-norm are considered unimportant compared with filters with large $l1$-norm.

- *Average Percentage of Zeros (APoZ)*: It was proposed by Hu *et al.* in [41], and it eliminates the filters with a large number of zero activations during inference time.

- *Entropy*: In this case, the entropy of a single filter is computed by feeding input images to the network and grouping the corresponding outputs in different bins. The key idea is that unimportant filters will have very similar outputs with different inputs. It was proposed by Luo and Wu in [42].

- *Random*: It randomly select filters to be eliminated without using any prior knowledge of the DCNN architecture or filters. In [39], Mittal *et al.* showed that eliminating filters at random produced comparable results with other selection criteria.

2.3. Evolution Strategies

The work done by Mittal *et al.* in [39] shows that filter pruning with randomized heuristics produces results as good as when using heuristics based on prior knowledge. Moreover, the use of randomized heuristics was capable of producing excellent results even when used for automatic generation of DCNN architectures [43]. Thus, the use of an algorithm that relies on random changes of candidate solutions is ideal for performing filter pruning.

The evolutionary computation (EC) family of algorithms possess such characteristic. Specifically, Evolution Strategies (ESs) [37] relies more heavily on the use of random changes than any other EC algorithms. ESs are meta-heuristic algorithms used to solve optimization problems. Initially, ESs were created to solve design problems automatically, but its creators quickly realized that they could be used to solve all sorts of optimization problems, such as binary optimization and combinatorial optimization. Similar to other EC algorithms, ESs are population-based algorithms where an individual in the population is a possible solution.

In general, ESs work in a loop, known as generations, where these two basic steps are performed [37]:

- In every generation, modify the individuals' parameters by using recombination and/or mutation.

- Keep the best individuals for the next generation and repeat the process.

This family of algorithms uses the following standard notations: $(\mu/\rho, \lambda) - ES$ or $(\mu/\rho + \lambda) - ES$, where μ represents the number of individuals that will be kept and used as parents to the next generation, λ represents the number of individuals that will be generated at every new generation from the μ parents selected in the previous generation, and ρ is the number of individuals used for recombination. The *comma* (,) and *plus* (+) operators define how the selection

11

operator of the algorithm will work. In the *comma* version, the algorithm will select μ individuals only from λ offspring, and the old parents will be completely discarded even if they are better than the offspring. In the *plus* version, the algorithm will select λ individuals from all $\mu + \lambda$ individuals, i.e., parents and offspring are used together during the selection procedure. Thus, the *plus* version is an elitist algorithm where the best solutions are kept in the population indefinitely. Recombination is an optional step in ESs and it works by randomly selecting ρ individuals from λ parents. In ES, recombination always produces one new individual from two or more parents, and each one of its parameters are selected from one of the parents at random. Let $\mathbf{P}_j$ be a vector representing the N parameters of a parent individual j, then the parameters of a new offspring, $\mathbf{O}$, will be equal to

$$O_k := rand_j(P_{jk}), \text{ with } 1 \leq k \leq N \text{ and } j = 1, ..., \rho. \tag{4}$$

In the present work, DCNN models are pruned with the help of ES and without the use of any prior knowledge. In our case, the selection pressure produced by the evolutionary process will be responsible for finding a good combination of filters to be kept in a pruned DCNN model.

3. Proposed Algorithm

The general framework of the proposed *DeepPruningES* is shown in Algorithm 1. It is comprised mainly of the standard ES *plus* version adapted to perform DCNN pruning. It receives as input the size of the offspring population (λ_{size}), the maximum number of generations that the algorithm will run (gen), the original DCNN model ($model$) that will be pruned, the number of epochs to be used during an individual evaluation (e_{eval}), the learning rate used during an individual evaluation (α_{eval}), and the number of epochs (e_{fine}) and learning rate (α_{fine}) to be used during the fine-tuning of the best individuals found at the end. The final output of the proposed *DeepPruningES* will be three DCNN models with different trade-offs called here as *knee*, *boundary heavy*, and *boundary light* solutions.

There are four important pieces in Algorithm 1: *Population Initialization* (line 1), *Knee and Boundary Selection* (line 4), *Offspring Generation* (line 5), and *Fine-Tuning* of the best solutions found (line 7). Each one of them will be explained in detail in the following subsections, as well as the genetic representation of an individual.

3.1. Genetic Representation of an Individual

Representation is one of the most important aspects of any population-based algorithm. In the proposed *DeepPruningES*, we use two binary strings to represent the filters of a DCNN model. It is important to note that our proposed algorithm only prunes convolution layers because they have a much higher computational complexity compared to fully-connected ones [29]. In this representation, each bit represents one single filter. For example, if we want to represent

12

Algorithm 1: Proposed *DeepPruningES*

Input : Offspring size (λ_{size}), maximum number of generations (gen), mutation probability (p_m), original DCNN model ($model$), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}), number of epochs for fine-tuning (e_{fine}), learning rate for fine-tuning (α_{fine}).

Output: Three DCNN models: knee solution ($\mu.knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

---

1 $\mu, \lambda \leftarrow$ *Initialize Population*(λ_{size}, *model*);
2 **for** *g = 1 to gen* **do**
3 | $\mathbf{P} \leftarrow \boldsymbol{\mu} + \boldsymbol{\lambda}$
4 | $\boldsymbol{\mu} \leftarrow$ *Knee Boundary Selection*($\mathbf{P}$, *model*, e_{eval}, α_{eval});
5 | $\boldsymbol{\lambda} \leftarrow$ *Offspring Generation*($\boldsymbol{\mu}$, λ_{size}, p_m, *model*);
6 **end**
7 *Fine-tuning*($\boldsymbol{\mu}$, *model*, e_{fine}, α_{fine});
8 **return** $\mu.knee$, $\mu.heavy$, $\mu.light$;

---

two convolution layers, one with 32 filters and the other with 64 filters, we would need a binary string with 96 bits, and a bit assigned to zero means that the corresponding filter would be eliminated.

However, the representation depends on the type of DCNN model being pruned. When pruning simple CNN models, only a single binary string is used, and each bit will represent a filter of the CNN model, as illustrated in Figs. 5 and 6. In the case of ResNet models, two binary strings are used to encode the filters: one represents the first convolution layer, while the other represents the second convolution layer of a residual block. Because a tensor entering a residual block needs to be added with a tensor exiting the residual block, see Eq. 3, we must prune the second layer of all residual blocks at the same time in order to maintain the size of all tensors in the model consistent. This process is illustrated in Fig. 7 where, in the left side, we have one binary string encoding the first layers of each residual block, and, in the right side, we have one binary string encoding the second layers of multiple residual blocks. Note that the second layers from blocks having the same number of filters, for example 16 filters, are encoded together. For example, in Fig. 7, the layers in green (represented by small bars in the right side of the box) only need 16 bits to be encoded, and, thus, if we eliminate four filters from the first green layer, we will also need to delete four filters from all other green layers. For last, the binary representation used for DenseNet models also uses two binary strings: one to represent the bottleneck layers, and the other to represent the convolution layers, as shown in Fig. 8. Because DenseNets use concatenation instead of addition, it is possible to prune each layer of the model independently, and, therefore, they are easier to represent than ResNets.
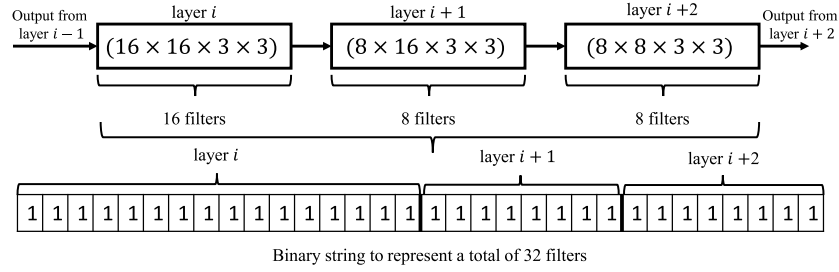
Figure 5: Binary representation of a CNN model. All layers are convolution layers.
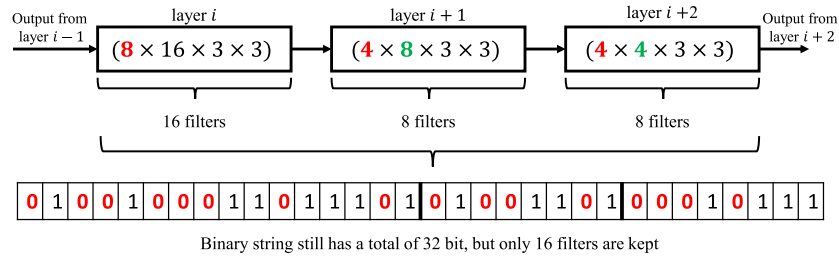

Figure 6: Binary representation of a CNN model after removing half of the filters from each layer.

3.2. Population Initialization

The first step of the proposed *DeepPruningES* is to initialize a population of $3 + \lambda_{size}$ individuals. First, all individuals are initialized as an identical copy of the original DCNN model being pruned. Then, their genes are mutated with probability p_m to generate a pruned version of the original model. Finally, the main *for-loop*, shown in Algorithm 1, line 2, can be executed.

The main idea of initializing the population in this way instead of a purely randomized initialization is that the population will resemble the original DCNN model allowing us to achieve good results even when using smaller populations and fewer generations.

3.3. Individual Evaluation

Before we can talk about how the knee and boundaries solutions are selected, it is important to explain how an individual is evaluated in the proposed algorithm. This process is achieved by, first, eliminating the filters indicated in the individual's gene, and, then, fine tuning the resulting pruned model using Stochastic Gradient Descent (SGD) for e_{eval} epochs and α_{eval} learning rate in a small sample of the dataset that was used to train the original model. This allows the individual to restore some of its performance before the selection operation is performed. More details about how the dataset is sampled and for how long the pruned model is fine-tuned are presented in Subsection 4.1.

The individuals are evaluated in two objectives: the training error in the chosen dataset, and the number of floating-point operations (FLOPs) for a single

Figure 7: Binary representation of a ResNet with 20 layers.

input. These are conflicting objectives because it is not possible to maintain a small training error with a small number of floating-point operations. We choose to use the number of FLOPs instead of the total number of parameters or the total number of filters because the time to compute the outputs of any given layer is also dependent on the size of its inputs not only on its number of parameters. Thus, the total number of FLOPs of a model is considered a better measurement of its computational complexity.

Figure 8: Binary representation of a single dense block used in DenseNet models.

3.4. Knee and Boundary Selection

In many real-world problems, decision makers (DM) have to optimize multiple parameters at the same time to accomplish some desired behavior. These problems are called multi-criteria decision making (MCDM) problems, where there is n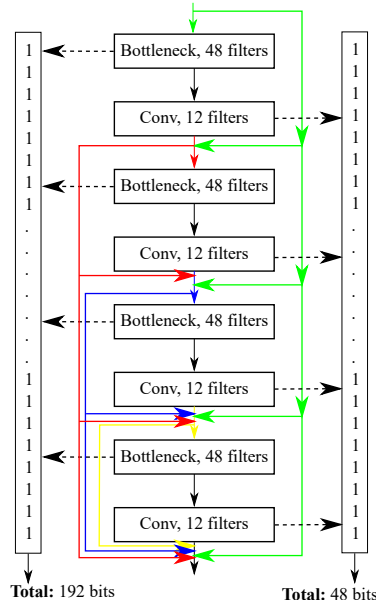o single desired or optimal output to a given problem. In many cases, these problems are multi-objective optimization problems where the modification of a single parameter from a candidate solution will change the output of different objective functions. However, not all possible solutions are of interest in MCDM problems. Only the solutions located in the so-called *Pareto optimal front* in the objective space is of interest. This region is where solutions that are no worse than others in all objective functions are located. The *knee* solution is always found in the *Pareto optimal front*. The ability to locate the *knee* solution is of extreme importance in MCDM problems because this is the solution that can improve the most in any of the objectives with small changes in its parameters [38].

In the case of DCNNs, the parameters are the number of filters still present in the architecture, and the objective functions that we would like to optimize are the DCNN's computational complexity, and its accuracy in a given dataset. Furthermore, we would like to provide three DCNN models to DMs. So, depending on what type of device the DCNN is being deployed one model could be more suitable than the others.

Commonly, all the solutions located in *Pareto optimal front* need to be found before the *knee* can be identified. Chiu *et al.* in [38] showed that *knee* is

16

Algorithm 2: Knee and Boundary Selection

Input : Individuals in the Population (**P**), original DCNN model ($model$), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}).

Output: Three individuals: knee solution ($\mu.knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

---

1 *Evaluate Population*(**P**, $model$, e_{eval}, α_{eval});

2 Find $min(f_1), min(f_2), max(f_1), max(f_2)$;

3 $\mu.heavy \leftarrow P_i$, where $f_1(P_i) = min(f_1)$;

4 $\mu.light \leftarrow P_j$, where $f_2(P_j) = min(f_2)$;

5 **for** $k = 1$ to $len(\mathbf{P})$ **do**

6 $\quad | \quad dist(k) = \frac{f_1(P_k) - min(f_1)}{max(f_1) - min(f_1)} + \frac{f_2(P_k) - min(f_2)}{max(f_2) - min(f_2)}$;

7 **end**

8 $\mu.knee \leftarrow P_k$, where P_k has the minimum $dist(k)$;

9 **return** $\mu.knee$, $\mu.heavy$, $\mu.light$;

---

the solution in *Pareto optimal front* with the smallest Manhattan distance to the origin point of the objective space. However, finding the whole *Pareto optimal front* is a non-trivial task. Thus, we propose a modification of the Minimum Manhattan Distance algorithm developed by Chiu *et al.* to locate the *knee* solution. Because we are not interested in finding the whole *Pareto optimal front*, we can use the geometric position of all candidate solutions in the objective space to facilitate the search of the *knee* solution. This is done by simply computing the Manhattan distances of all candidate solutions, and finding the one with the smallest distance.

Thus, the proposed algorithm to perform the *Knee and Boundary Selection* is shown in Algorihtm 2, where three individuals, called here as *knee*, *boundary heavy*, and *boundary light* solutions, are always selected at every generation. First, we need to evaluate the individuals in the population, **P**, and to obtain their training error (f_1) and number of FLOPs (f_2). Second, we need to find the minimum and maximum values of both objectives in the population (Algorithm 2, line 2). Third, the *Boundary Heavy* solution will be the individual with the smallest training error in the population (i.e., the individual with its f_1 equal to $min(f_1)$). While the *Boundary Light* solution will be the individual with the smallest number of FLOPs in the population, the individual with its f_2 equal to $min(f_2)$. Finally, we compute the Manhattan Distance of all individuals in the population by following the equation in Algorithm 2, line 6, and select the individual with the Minimum Manhattan Distance (MMD) to be our *knee* solution. The selection procedure is illustrated in Fig. 9, and only these three individuals are returned at the end of selection.
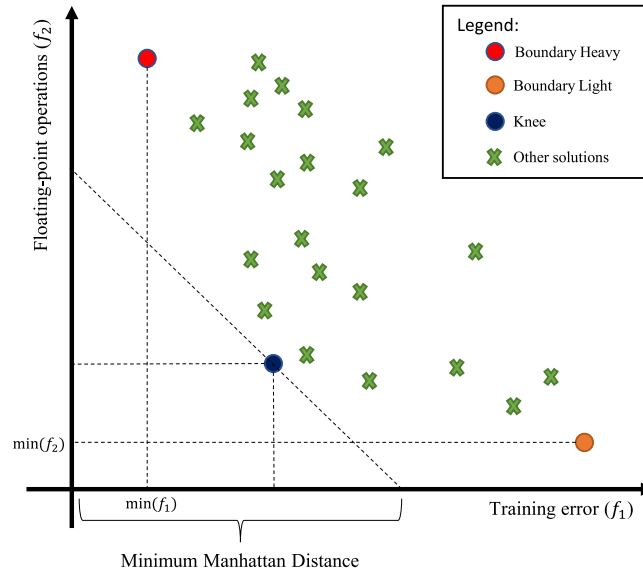
Figure 9: Example of the *knee*, *boundary heavy*, and *boundary light* solutions.

3.5. Offspring Generation

After the three individuals from the knee and boundaries are selected, the algorithm will use them to generate new offspring. First, from these three individuals, one is randomly selected to be the parent of a single offspring. Then, each one of its gene elements is randomly flipped with probability equal to p_m. This process is repeated until we have an offspring population equal to λ_{size}. It is also important to point out that we are following the standard ES algorithm where only the knee and boundaries solutions are used to generate a new offspring population. Thus, a newly created offspring is never used to generate another offspring. This method ensures that new offspring are always generated closer to the knee or boundary solutions allowing the algorithm to exploit these regions of interest efficiently.

3.6. Fine Tuning

The last step of the proposed algorithm is to fine-tune the knee and boundaries solutions found in the last generation. They are fine-tuned using Stochastic Gradient Descent (SGD) during e_{fine} epochs and learning rate equal to α_{fine}. Different from the Individual Evaluation, the fine-tuning process uses the entire training set in order to improve these three solutions as much as possible. The number of epochs during fine-tuning is also higher than the one used during individual evaluation. At the end of this process, the three solutions are saved on disk for late use.

18

4. Experimental Design

In this section, we discuss the experimental design used to evaluate the proposed *DeepPruningES*. We begin by presenting the DCNN architectures tested with the proposed work. Then, we analyze the algorithm parameters that was used to prune those DCNN architectures. Furthermore, all results presented in this work were obtained using a Laptop PC with a Core i7-8750H CPU, 16 GB of RAM, and a NVIDIA GTX 1060 6GB GPU running PyTorch on Windows 10 Pro 64-bits.

4.1. Chosen DCNN Architectures for Pruning

The proposed *DeepPruningES* is evaluated on state-of-the-art DCNN architectures. As stated in the introduction of this work, the proposed algorithm can prune three types of DCNN architectures: Convolutional Neural Networks (CNN), Residual Neural Networks (ResNet), and Densely Connected Neural Networks (DenseNets). We choose to test the algorithm with DCNNs from these architectures, given a challenging dataset.

The CIFAR10 and CIFAR100 datasets created by Krizhevsky [44] are the chosen ones to be used in our experiments because, besides being a challenging dataset, there are many state-of-the-art DCNNs results widely reported to allow us to stay focus on quantifying the quality of the pruning task. The CIFAR10 dataset contains 50,000 training images and 10,000 test images distributed in a total of 10 classes. All images are in colors and have size equal to 32×32 pixels. Some examples of images found in the CIFAR10 dataset can be seen in Fig. 10. Likewise, the CIFAR100 dataset contains the same amount of images with the same resolution as the CIFAR10, but these images are distributed in a total of 100 classes instead of only 10. To speed up and reduce the cost of the pruning process, only 1,000 training images, with an equal number of images per class, from the CIFAR10 and CIFAR100 datasets are used to perform an *Individual Evaluation*. At the end of the proposed algorithm, when fine-tuning the three selected individuals, the whole 50,000 training images from the these datasets are used to allow the candidate solutions to recover from any loss of performance due to the pruning process.

We choose to evaluate the proposed algorithm with the following networks: *VGG16*, *VGG19*, *ResNet56*, *ResNet110*, *DenseNet50*, and *DenseNet100*. The *VGG16* and *VGG19* [6] are two examples of CNNs created in 2014 which achieved good results in challenging image classifications datasets, such as CIFAR10 and ImageNet. As the name suggests, *VGG16* has a total of 16 layers while *VGG19* has a total of 19 layers. The *VGG16* and *VGG19* have computational complexities of 3.15×10^8 and 4.01×10^8 FLOPs, respectively. Their original and fully trained architectures can achieve a test error of 6.06% and 6.18% on the CIFAR10 dataset, respectively, and 32.33% and 32.57% test error on the CIFAR100 dataset, respectively. The *ResNet56* and *ResNet110* [35] are two example of ResNets, and they are the ones first proposed by the authors in [35] to be used with the CIFAR10 dataset. Their computational complexities are of 1.27×10^8 and 2.57×10^8 FLOPs, respectively. When fully trained, *ResNet56*
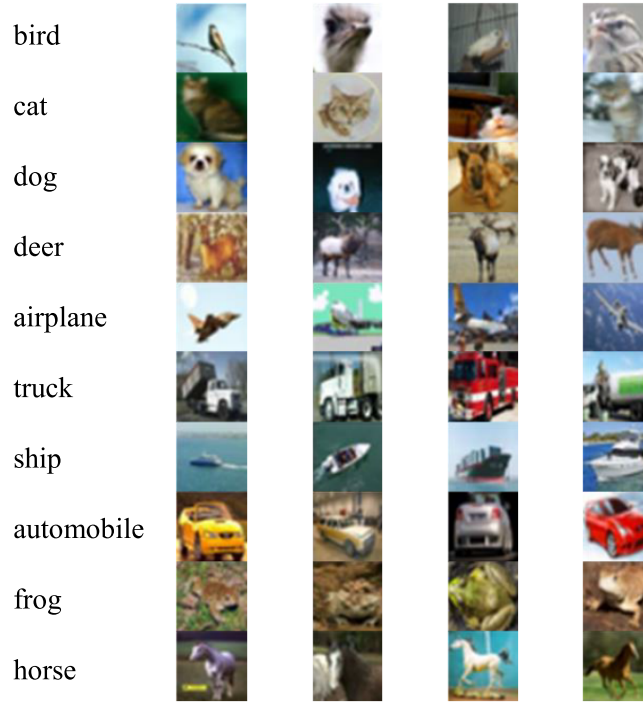
Figure 10: Example of images from each class in the CIFAR10 dataset.

Table 1: Overview of the DCNN architectures used to evaluate the proposed *DeepPruningES*.

DCNN	# layers	# FLOPs	Test error on CIFAR10	Test error on CIFAR100
VGG16	16	3.15×10^8	6.06%	32.33%
VGG19	19	4.01×10^8	6.18%	32.57%
ResNet56	56	1.27×10^8	6.63%	41.94%
ResNet110	110	2.57×10^8	6.2%	50.47%
DenseNet50	50	0.93×10^8	6.92%	41.25%
DenseNet100	100	3.05×10^8	5.66%	33.06%

and *ResNet110* can achieve a test error of 6.63% and 6.2% on CIFAR10, respectively, and 41.94% and 50.47% on the CIFAR100 dataset, respectively. The *ResNet56* contains a total of 56 layers, while the *ResNet110* contains a total of 110 layers. For last, the *DenseNet50* and *DenseNet100* [36] are also some of the proposed architectures based on the DenseNet's authors to classify the CIFAR10 dataset. They have 50 and 100 layers, respectively, computational complexity of 0.93×10^8 and 3.05×10^8 FLOPs, respectively, and test error of 6.92% and 5.66% on the CIFAR10 dataset, respectively, and 41.25% and 33.06% on the CIFAR100 dataset, respectively. Table 1 sums up the details about the DCNN architectures used to evaluate the proposed algorithm.

4.2. Algorithm Parameters

The parameters' values used to evaluate the proposed *DeepPruningES* algorithm are shown in Table 2. Throughout this study, all results shown related to the proposed algorithm are obtained by using the parameters from Table 2. If some results are obtained by using a different set of parameters, the difference is highlighted in the discussions. The next paragraphs will discuss the influence of each parameter in the final output of the *DeepPruningES*.

The size of the offspring (λ_{size}) defines how many offspring is generated at each generation. More offspring helps the algorithm in covering more of the objective space and increases the probability of finding good solutions. Likewise, a large number of generations (gen) also increases the probability of finding good solutions. However, when the number of offspring and/or generation increases, the computational cost to run the algorithm also increases. In our tests, 20 offspring and 10 generations are considered enough to achieve good results given affordable computational complexity.

The mutation probability (p_m) is the probability of a single bit of an individual's gene being inverted. A very small mutation probability would take too long to produce significant changes in the population, which would require more individuals and more generations to reach a satisfactory result; while a very large mutation probability will produce too many changes in the population resulting in no improvement at all.. In our experiments, a probability of 0.1 is regarded as the best value heuristically that makes the individuals change in a way that improves the overall quality of the population.

The number of epochs and learning rate used during an individual evaluation (e_{eval}, α_{eval}) will control how long an individual will be fine-tuned before its training error and number of FLOPs is evaluated. The values for these two parameters depend on the computational power available to the user. Ideally, if an individual can be fine-tuned much longer, this will give us a better sense of its overall fitness. We chose to use 5 epochs, which is compatible with other EC based algorithms used for DCNNs architecture generation and pruning [28, 42, 15], and a learning rate of 0.1. In our case, we are using a large learning rate to improve the accuracy of the solutions faster.

For last, the number of epochs and learning rate to fine-tune the final solutions (e_{fine}, α_{fine}) will improve the training error of the three final solutions. We chose to fine-tune them for 50 epochs with a learning rate of 0.01, which is also compatible with others' work [29, 27]. In this case, we use a smaller learning rate than that during an individual evaluation to avoid training instabilities due to excessive removed filters. It is possible to improve the solutions found by the algorithm even further if they are fine-tuned for longer than 50 epochs. However, our limited computational power did not allow us to experiment with very long fine-tuning sessions.

As stated before, the proposed *DeepPruningES* uses the ES *plus* version for performing DCNN filter pruning, which is an elitist version of ES where the best individuals are never lost. In Section V, an experiment using the ES *comma* version is presented, and it does not yield good results because the algorithm eliminates too many filters even from the boundary heavy and knee solutions.

Table 2: Parameters used to evaluate the proposed *DeepPruningES*.

Parameter	Value
Offspring size (λ_{size})	20
Maximum number of generations (gen)	10
Mutation probability (p_m)	0.1
Number of epochs for individual evaluation (e_{eval})	5
Learning rate for individual evaluation (α_{eval})	0.1
Number of epochs for fine-tuning (e_{fine})	50
Learning rate for fine-tuning (α_{fine})	0.01

Table 3: Pruning results obtained with the proposed *DeepPruningES* on the CIFAR10 dataset.

DCNN Model	DeepPruningES						
	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
VGG16	Knee	9.04%	9.58%	1.09×10^8	1.22×10^8	65.49%	61.58%
	Boundary Heavy	8.21%	8.6%	2.15×10^8	2.49×10^8	32.01%	20.88%
	Boundary Light	10.51%	11.41%	0.88×10^8	0.9×10^8	72.17%	71.36%
VGG19	Knee	9.04%	9.87%	1.53×10^8	1.72×10^8	61.86%	57.15%
	Boundary Heavy	8.21%	8.77%	2.7×10^8	3.12×10^8	32.56%	22.28%
	Boundary Light	10.53%	12.03%	1.13×10^8	1.18×10^8	71.74%	70.69%
ResNet56	Knee	9.28%	9.98%	0.432×10^8	0.523×10^8	66.23%	59.15%
	Boundary Heavy	8.11%	8.77%	1.01×10^8	1.08×10^8	21.31%	15.23%
	Boundary Light	11.42%	13.36%	0.244×10^8	0.286×10^8	80.89%	77.67%
ResNet110	Knee	8.66%	9.42%	0.905×10^8	1.03×10^8	64.84%	59.89%
	Boundary Heavy	7.43%	7.93%	2.14×10^8	2.21×10^8	16.72%	14.14%
	Boundary Light	10.27%	12.9%	0.43×10^8	0.56×10^8	83.29%	77.86%
DenseNet50	Knee	9.8%	10.43%	0.41×10^8	0.466×10^8	56.05%	50.15%
	Boundary Heavy	8.91%	9.26%	0.756×10^8	0.779×10^8	19.16%	16.59%
	Boundary Light	13.04%	14.8%	0.229×10^8	0.244×10^8	75.53%	73.91%
DenseNet100	Knee	9.04%	9.37%	1.11×10^8	1.21×10^8	63.64%	60.31%
	Boundary Heavy	8.34%	8.39%	2.46×10^8	2.49×10^8	19.33%	18.24%
	Boundary Light	10.47%	11.90%	0.82×10^8	0.879×10^8	73.09%	71.16%

5. Experimental Results and Discussion

In this section, our pruning results are first presented and compared with peer competitors. Then, a detailed discussion about the obtained results follows.

5.1. Experimental Results

To evaluate the proposed *DeepPruningES*, the DCNN architectures shown in Table 1 were pruned 10 times. The results using the CIFAR10 dataset are shown in Table 3, where, for each DCNN architecture, we show the best and mean test error and number of FLOPs for the knee, boundary heavy, and boundary light solutions after fine-tuning for 50 epochs. The results using the CIFAR100 dataset are shown in Table 4, where the three selected solution are fine-tuned for 100 epochs. In addition, in Table 5, we show the results reported by the peer competitors who used the same CIFAR10 dataset. Other works found in the literature are not shown here because they used the ImageNet dataset, which is too big for us to run in our equipment and does not produce comparable results with other datasets.

On the CIFAR10 dataset, for the *VGG16* and *VGG19*, the proposed pruning algorithm obtain similar results. The boundary heavy solutions have an average of 20% reduction in the number of FLOPs, the boundary light solutions an average of 70% reduction, and the knee solutions an average of 57% reduction. The

Table 4: Pruning results obtained with the proposed *DeepPruningES* on the CIFAR100 dataset.

DCNN Model	DeepPruningES						
	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
VGG16	Knee	34.27%	34.41%	1.289×10^8	1.411×10^8	59.19%	55.37%
	Boundary Heavy	32.94%	33.08%	2.531×10^8	2.577×10^8	19.93%	18.47%
	Boundary Light	35.68%	36.36%	6.786×10^7	7.015×10^7	78.53%	77.81%
	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
VGG19	Knee	34.6%	35.09%	1.652×10^8	1.882×10^8	58.82%	53.08%
	Boundary Heavy	33.11%	33.41%	3.246×10^8	3.277×10^8	19.09%	18.31%
	Boundary Light	37.05%	37.41%	8.640×10^7	8.970×10^7	78.46%	77.64%
	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
ResNet56	Knee	53.98%	55.57%	1.484×10^7	1.681×10^7	88.40%	86.86%
	Boundary Heavy	42.19%	42.49%	1.072×10^8	1.096×10^8	16.19%	14.26%
	Boundary Light	53.89%	56.70%	1.421×10^7	1.587×10^7	88.89%	87.61%
	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
ResNet110	Knee	59.06%	64.09%	3.068×10^7	3.633×10^7	88.08%	85.89%
	Boundary Heavy	50.97%	51.49%	2.117×10^8	2.175×10^8	17.73%	15.63%
	Boundary Light	62.07%	64.93%	3.069×10^7	3.365×10^7	88.07%	86.93%
	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
DenseNet50	Knee	42.09%	62.01%	1.606×10^7	2.643×10^7	82.82%	71.73%
	Boundary Heavy	41.59%	42.02%	7.569×10^7	7.815×10^7	19.05%	16.42%
	Boundary Light	63.44%	65.06%	1.606×10^7	1.705×10^7	82.82%	81.76%
	Solution	Test error (best)	Test error (mean)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best)	% Pruned (mean)
DenseNet100	Knee	50.46%	49.13%	6.351×10^7	6.503×10^7	79.18%	78.69%
	Boundary Heavy	35.344%	34.41%	2.469×10^8	2.50×10^8	19.05%	18.04%
	Boundary Light	50.24%	49.24%	6.185×10^7	6.316×10^7	79.73%	79.29%

average test errors of the boundary heavy, boundary light, and knee solutions are around $8.6\% \sim 8.77\%$, $11.51\% \sim 12.01\%$, and $9.58\% \sim 9.87\%$, respectively. These are the expected results where the *boundary heavy* solutions were the least pruned, and also the ones with the highest accuracy. On the other hand, the *boundary light* solutions were the most pruned, and the ones with the lowest accuracy. For last, the *knee* solutions obtained pruned results and accuracy in between.

On the CIFAR10 dataset, for the *ResNet56* and *ResNet110*, the boundary heavy solutions see an average of 15% reduction in the number of FLOPs, the boundary light solutions an average of 77% reduction in the number of FLOPs, and the knee solutions an average of 59% reduction in the number of FLOPs. After fine-tuning, the boundary heavy, boundary light, and knee solutions obtain test errors on average of $7.93\% \sim 8.77\%$, $12.9\% \sim 13.46\%$, and $9.42\% \sim 9.98\%$, respectively. Even though ResNets are smaller than conventional CNNs, such as VGG16 and VCGG19, by nature, these results show that they can still be pruned without losing too much accuracy.

On the CIFAR10 dataset, for the DenseNet50 and DenseNet100, the average reduction in the number of FLOPs for the boundary heavy, boundary light and knee solutions are around 16%, 73%, and 50%, respectively. On the other hand, the average test errors for the boundary heavy, boundary light and knee solutions are $8.39\% \sim 9.26\%$, $11.9\% \sim 14.8\%$, and $9.37\% \sim 10.43\%$, respectively. These results are similar to the ones obtained with the pruning of ResNets which are impressive considering that DenseNets are even smaller than ResNets.

5.2. Results Discussion

As shown by the results in Tables 3 and 4, the proposed pruning algorithm can reduce the number of FLOPs in DCNN architectures significantly while preserving good test errors. As shown in Table 5, when comparing our results with the ones obtained by Li *et al.* in [29], the proposed *DeepPruningES* obtains higher pruning ratios with slight worse test errors on CIFAR10. Ding

Table 5: Pruning results from peer competitors using the CIFAR10 dataset.

Approach	DCNN Model	% FLOPs Pruned	Test error
Li *et al.* [29]	VGG16	34.2%	6.60%
	ResNet56	27.6%	6.94%
	ResNet110	38.6%	6.70%
Ding *et al.* [27]	VGG16	81.39%	7.56%
	ResNet56	66.88%	9.43%

Table 6: Efficient mobile deep convolutional neural networks trained from scratch results.

	Test error on CIFAR100
MobileNet v2	30.91%
SqueezeNet	44.53%
EfficientNet-B0	48.02%

et al. in [27] were able to obtained better pruning ratios and test errors than the ones from the proposed algorithm. However, in their algorithm, the DC-NNs needs to be trained with strong and specialized regularization. The proposed *DeepPruningES* uses only the information about the DCNN classification performance and computational complexity to prune it. Thus, without using any regularization or extra knowledge about the DCNN being pruned, *DeepPruningES* is still capable of achieving quality results comparable with the ones from Ding et al. in [27]. Moreover, Table 6 shows the test errors on the CI-FAR100 dataset of multiple efficient mobile DCNNs trained for 150 epochs, and the results show that the pruned version of full-fledged DCNN architecture can achieve competitive classification performance compared with their mobile counterparts. Beyond the qualitative results, the proposed *DeepPruningES* algorithm is capable of finding pruned DCNN models in a couple of hours in an Nvidia GTX 1060 6GB GPU. However, its bottleneck comes once the pruned models are fine-tuned (Algorithm 1, line 7) using the whole dataset, which, depending on the DCNN model being pruned, could take a couple of days.

It is also important to point out that the primary goal of the proposed algorithm is to find a set of three pruned DCNN models. Thus, a Decision-Maker (DM) has, in fact, four possible options to choose from: the original DCNN model, the *boundary heavy* solution, the *boundary light* solution, or the *knee* solution. Therefore, although the *boundary heavy* solution can look unappealing at first glance, it is still an essential candidate solution that a DM can use depending on his or her available hardware.

The test errors showed in Tables 3 and 4 can still be further improved given more computational resources are available. To demonstrate this, we choose one pruned solution from VGG16 to be retrained from scratch on the CIFAR10 dataset, which means that all weights are reinitialized to random values and the network is trained from there. More specifically, a knee solution with 58.64% pruning ratio and test error of 9.94% is chosen to be retrained from scratch for
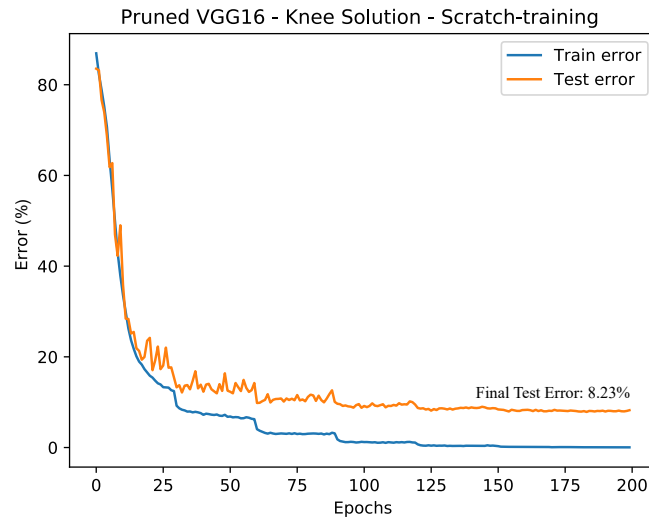
24

Figure 11: Knee solution from the pruned VGG16 network trained from scratch for 200 more epochs.
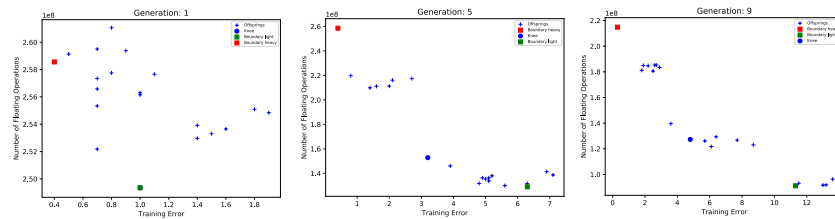

Figure 12: Evolution of the population when pruning VGG16 using the *plus* version of the proposed DeepPruningES.

200 more epochs. The training curves for this knee solution is shown in Fig. 11. Its final test error is equal to 8.23% which is considerable better than the test error obtained after 50 fine-tuning epochs.

Our results are more comprehensive than the ones from peer competitors because six different DCNN models with three different architectures and number of layers are evaluated. For example, to the best of our knowledge, no work in the literature has attempted to prune DenseNet architectures. Although DenseNets already have smaller computational complexity with better classification performance than standard CNNs, our results showed that they can still be pruned and still maintain competitive classification performance.

We also compared the proposed *DeepPruningES*, which is based on the elitist ES *plus* version, with a non-elitist *comma* version. The evolution of the population for 10 generations for the *plus* version is shown in Fig. 12 and for the *comma* version is shown in Fig. 13. In the proposed *plus* version, it is easier to keep the individual with the best training error throughout the generations
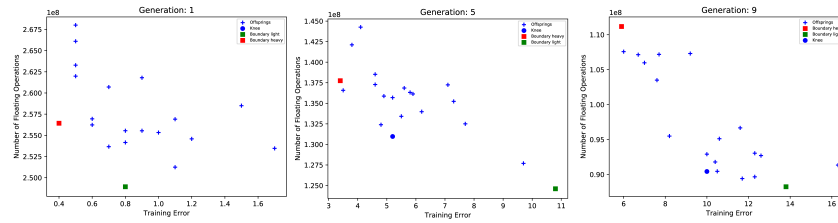
Figure 13: Evolution of the population when pruning VGG16 using the *comma* version of the proposed DeepPruningES.

than in the *comma* version. Moreover, the final solutions found when using the *comma* version are very close to each other in terms of error and number of FLOPs which defeats our objective of getting multiple solutions with different trade-offs. Thus, if one is going to use the proposed *DeepPruningES* for pruning DCNN architectures, the plus version of the algorithm is the suggested one to use.

For last, we would like to discuss why an Evolution Strategy (ES) framework was chosen instead of a standard Genetic Algorithm (GA) one. We wanted to give a set of candidate solutions to Decision-Makers (DMs) to choose from instead of a single one. Thus, by maintaining only the three best individuals in the population, the proposed selection operator fits in an ES framework without the developing a fully fledged Multi-Objective GA. Furthermore, although ES was originally designed to deal with real-valued vectors, it is perfectly suitable for use with binary vectors [37].

6. Conclusions and Future Work

In this work, we propose an algorithm for pruning Deep Convolutional Neural Networks (DCNN) architectures using Evolution Strategies, called here as *DeepPruningES*. The pruning is performed by eliminating convolution filters from multiples layers throughout the network architecture. While other works use the current layer's statistics to decide which filter to eliminate, the proposed algorithm eliminates filters by exploiting the objective space of each solution in the population. Moreover, to the best of our knowledge, the current work is one of the firsts to frame the pruning problem as a multi-objective optimization problem. In this sense, the proposed algorithm can find three different solutions with different trade-offs between training/test error and computational complexity without relying on any prior knowledge from the architecture being pruned. This set of solutions eliminates the need of manually setting a trade-off parameter as seen others' pruning algorithms.

The proposed *DeepPruningES* can achieve competitive results with a population of only 20 individuals and 10 generations when compared with state-of-the-art peer competitors work. The algorithm is also one of the firsts to pruning

multiples DCNN architectures, such as CNNs, ResNets, and DenseNets architectures. Our results are limited only by our available computational power. With more computational resources available, the proposed algorithm could achieve even better results.

For future works, the proposed algorithm will be extended to prune more DCNN architectures, such as the Inception architecture, and we will develop an algorithm for searching neural network architectures automatically and integrate it with the proposed pruning algorithm. This will create an algorithm capable of finding new DCNN architectures with good performance while maintaining a small computational complexity.

7. Acknowledgments

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, Backpropagation Applied to Handwritten Zip Code Recognition, Neural Computation 1 (4) (1989) 541–551. `doi:10.1162/neco.1989.1.4.541`.
URL `http://www.mitpressjournals.org/doi/10.1162/neco.1989.1.4.541`

[2] K. He, X. Zhang, S. Ren, J. Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, in: 2015 IEEE International Conference on Computer Vision (ICCV), IEEE, Santiago, Chile, 2015, pp. 1026–1034. `doi:10.1109/ICCV.2015.123`.
URL `http://ieeexplore.ieee.org/document/7410480/`

[3] S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, in: Proceedings of the 32nd International Conference on Machine Learning, Vol. 37 of Proceedings of Machine Learning Research, PMLR, Lille, France, 2015, pp. 448–456.
URL `http://proceedings.mlr.press/v37/ioffe15.html`

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, Li Fei-Fei, ImageNet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, Miami, FL, 2009, pp. 248–255. `doi:10.1109/CVPR.2009.5206848`.
URL `http://ieeexplore.ieee.org/document/5206848/`

[5] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, L. Fei-Fei, ImageNet Large Scale Visual Recognition Challenge, International Journal of Computer Vision 115 (3) (2015) 211–252. `doi:10.1007/s11263-015-0816-y`.
URL `http://link.springer.com/10.1007/s11263-015-0816-y`

[6] K. Simonyan, A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, arXiv:1409.1556ArXiv: 1409.1556 (Sep. 2014).
URL `http://arxiv.org/abs/1409.1556`

[7] W. Li, X. Zhu, S. Gong, Person Re-Identification by Deep Joint Learning of Multi-Loss Classification, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, Melbourne, Australia, 2017, pp. 2194–2200. `doi:10.24963/ijcai.2017/305`.
URL `https://www.ijcai.org/proceedings/2017/305`

[8] A. Darwish, A. E. Hassanien, S. Das, A survey of swarm and evolutionary computing approaches for deep learning, Artificial Intelligence Review 53 (3) (2020) 1767–1812. `doi:10.1007/s10462-019-09719-2`.
URL `http://link.springer.com/10.1007/s10462-019-09719-2`

[9] Y. Sun, B. Xue, M. Zhang, G. G. Yen, Evolving Deep Convolutional Neural Networks for Image Classification, IEEE Transactions on Evolutionary Computation (2019) 1–1`doi:10.1109/TEVC.2019.2916183`.
URL `https://ieeexplore.ieee.org/document/8712430/`

[10] Y. Sun, G. G. Yen, Z. Yi, Evolving Unsupervised Deep Neural Networks for Learning Meaningful Representations, IEEE Transactions on Evolutionary Computation 23 (1) (2019) 89–103. `doi:10.1109/TEVC.2018.2808689`.
URL `https://ieeexplore.ieee.org/document/8300639/`

[11] A. Martín, R. Lara-Cabrera, F. Fuentes-Hurtado, V. Naranjo, D. Camacho, EvoDeep: A new evolutionary approach for automatic Deep Neural Networks parametrisation, Journal of Parallel and Distributed Computing 117 (2018) 180–191. `doi:10.1016/j.jpdc.2017.09.006`.
URL `https://linkinghub.elsevier.com/retrieve/pii/S0743731517302605`

[12] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, A. Kurakin, Large-Scale Evolution of Image Classifiers, arXiv:1703.01041 [cs]ArXiv: 1703.01041 (Jun. 2017).
URL `http://arxiv.org/abs/1703.01041`

[13] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, B. Hodjat, Evolving Deep Neural Networks, arXiv:1703.00548 [cs]ArXiv: 1703.00548 (Mar. 2017).
URL `http://arxiv.org/abs/1703.00548`

[14] J. Jiang, F. Han, Q. Ling, J. Wang, T. Li, H. Han, Efficient network architecture search via multiobjective particle swarm optimization based on decomposition, Neural Networks (2019) S0893608019303971`doi:10.1016/j.neunet.2019.12.005`.
URL `https://linkinghub.elsevier.com/retrieve/pii/S0893608019303971`

[15] F. E. Fernandes Jr., G. G. Yen, Particle swarm optimization of deep neural networks architectures for image classification, Swarm and Evolutionary Computation 49 (2019) 62–74. `doi:10.1016/j.swevo.2019.05.010`.
URL `https://linkinghub.elsevier.com/retrieve/pii/S2210650218309246`

[16] T. Shinozaki, S. Watanabe, Structure discovery of deep neural network based on evolutionary algorithms, in: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, South Brisbane, Queensland, Australia, 2015, pp. 4979–4983. `doi:10.1109/ICASSP.2015.7178918`.
URL `http://ieeexplore.ieee.org/document/7178918/`

[17] G. Huang, S. Liu, L. v. d. Maaten, K. Q. Weinberger, CondenseNet: An Efficient DenseNet Using Learned Group Convolutions, in: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, IEEE, Salt Lake City, UT, 2018, pp. 2752–2761. `doi:10.1109/CVPR.2018.00291`.
URL `https://ieeexplore.ieee.org/document/8578389/`

[18] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, arXiv:1704.04861ArXiv: 1704.04861 (Apr. 2017).
URL `http://arxiv.org/abs/1704.04861`

[19] H. Liu, K. Simonyan, Y. Yang, DARTS: Differentiable Architecture Search, arXiv:1806.09055 [cs, stat]ArXiv: 1806.09055 (Apr. 2019).
URL `http://arxiv.org/abs/1806.09055`

[20] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, H. Adam, NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications, in: V. Ferrari, M. Hebert, C. Sminchisescu, Y. Weiss (Eds.), Computer Vision – ECCV 2018, Vol. 11214, Springer International Publishing, Cham, 2018, pp. 289–304. `doi:10.1007/978-3-030-01249-6_18`.
URL `http://link.springer.com/10.1007/978-3-030-01249-6_18`

[21] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, T. Abdelzaher, FastDeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices, in: Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems - SenSys '18,

ACM Press, Shenzhen, China, 2018, pp. 278–291. `doi:10.1145/3274783.3274840`.
URL `http://dl.acm.org/citation.cfm?doid=3274783.3274840`

[22] H. Halawa, H. A. Abdelhafez, A. Boktor, M. Ripeanu, NVIDIA Jetson Platform Characterization, in: F. F. Rivera, T. F. Pena, J. C. Cabaleiro (Eds.), Euro-Par 2017: Parallel Processing, Vol. 10417, Springer International Publishing, Cham, 2017, pp. 92–105. `doi:10.1007/978-3-319-64203-1_7`.
URL `http://link.springer.com/10.1007/978-3-319-64203-1_7`

[23] M. Su, J. Tan, C.-Y. Lin, J. Ye, C.-H. Wang, C.-L. Hung, Constructing a Mobility and Acceleration Computing Platform with NVIDIA Jetson TK1, in: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, IEEE, New York, NY, 2015, pp. 1854–1858. `doi:10.1109/HPCC-CSS-ICESS.2015.212`.
URL `https://ieeexplore.ieee.org/document/7336442/`

[24] D. Danopoulos, C. Kachris, D. Soudris, Acceleration of image classification with Caffe framework using FPGA, in: 2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST), IEEE, Thessaloniki, 2018, pp. 1–4. `doi:10.1109/MOCAST.2018.8376580`.
URL `https://ieeexplore.ieee.org/document/8376580/`

[25] J. Bottleson, S. Kim, J. Andrews, P. Bindu, D. N. Murthy, J. Jin, clCaffe: OpenCL Accelerated Caffe for Convolutional Neural Networks, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, Chicago, IL, USA, 2016, pp. 50–57. `doi:10.1109/IPDPSW.2016.182`.
URL `http://ieeexplore.ieee.org/document/7529850/`

[26] C. Ding, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, B. Yuan, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, CIRCNN: accelerating and compressing deep neural networks using block-circulant weight matrices, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17, ACM Press, Cambridge, Massachusetts, 2017, pp. 395–408. `doi:10.1145/3123939.3124552`.
URL `http://dl.acm.org/citation.cfm?doid=3123939.3124552`

[27] X. Ding, G. Ding, J. Han, S. Tang, Auto-Balanced Filter Pruning for Efficient Convolutional Neural Networks, in: AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, 2018.
URL `https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16450`

[28] J.-H. Luo, J. Wu, W. Lin, ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression, in: 2017 IEEE International Conference on Computer Vision (ICCV), IEEE, Venice, 2017, pp. 5068–5076. `doi:10.1109/ICCV.2017.541`.
URL `http://ieeexplore.ieee.org/document/8237803/`

[29] H. Li, A. Kadav, I. Durdanovic, H. Samet, H. P. Graf, Pruning Filters for Efficient ConvNets, in: International Conference on Learning Representations, Toulon, France, 2017.

[30] Y. Wang, C. Xu, J. Qiu, C. Xu, D. Tao, Towards Evolutionary Compression, in: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, London United Kingdom, 2018, pp. 2476–2485. `doi:10.1145/3219819.3219970`.
URL `https://dl.acm.org/doi/10.1145/3219819.3219970`

[31] Y. Zhou, G. G. Yen, Z. Yi, A Knee-Guided Evolutionary Algorithm for Compressing Deep Neural Networks, IEEE Transactions on Cybernetics (2019) 1–13`doi:10.1109/TCYB.2019.2928174`.
URL `https://ieeexplore.ieee.org/document/8781874/`

[32] S. Han, H. Mao, W. J. Dally, Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, in: International Conference on Learning Representations, San Juan, Puerto Rico, USA, 2016, arXiv: 1510.00149.
URL `http://arxiv.org/abs/1510.00149`

[33] M. Courbariaux, Y. Bengio, J.-P. David, BinaryConnect: Training Deep Neural Networks with binary weights during propagations, in: Advances in Neural Information Processing Systems 28, Curran Associates, Inc., 2015, pp. 3123–3131.

[34] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized Neural Networks, in: 30th Conference on Neural Information Processing System, Barcelona, Spain, 2016.
URL `https://papers.nips.cc/paper/6573-binarized-neural-networks.pdf`

[35] K. He, X. Zhang, S. Ren, J. Sun, Deep Residual Learning for Image Recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, Las Vegas, NV, USA, 2016, pp. 770–778. `doi:10.1109/CVPR.2016.90`.
URL `http://ieeexplore.ieee.org/document/7780459/`

[36] G. Huang, Z. Liu, L. V. D. Maaten, K. Q. Weinberger, Densely Connected Convolutional Networks, in: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, Honolulu, HI, 2017, pp. 2261–2269. `doi:10.1109/CVPR.2017.243`.
URL `http://ieeexplore.ieee.org/document/8099726/`

[37] H.-G. Beyer, H.-P. Schwefel, Evolution strategies – A comprehensive introduction, Natural Computing 1 (1) (2002) 3–52. `doi:10.1023/A:1015059928466`.
URL `http://link.springer.com/10.1023/A:1015059928466`

[38] W.-Y. Chiu, G. G. Yen, T.-K. Juan, Minimum Manhattan Distance Approach to Multiple Criteria Decision Making in Multiobjective Optimization Problems, IEEE Transactions on Evolutionary Computation 20 (6) (2016) 972–985. `doi:10.1109/TEVC.2016.2564158`.
URL `http://ieeexplore.ieee.org/document/7465803/`

[39] D. Mittal, S. Bhardwaj, M. M. Khapra, B. Ravindran, Studying the plasticity in deep convolutional neural networks using random pruning, Machine Vision and Applications 30 (2) (2019) 203–216. `doi:10.1007/s00138-018-01001-9`.
URL `http://link.springer.com/10.1007/s00138-018-01001-9`

[40] P. Molchanov, S. Tyree, T. Karras, T. Aila, J. Kautz, Pruning Convolutional Neural Networks for Resource Efficient Inference, arXiv:1611.06440ArXiv: 1611.06440 (Nov. 2016).
URL `http://arxiv.org/abs/1611.06440`

[41] H. Hu, R. Peng, Y.-W. Tai, C.-K. Tang, Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures, arXiv:1607.03250ArXiv: 1607.03250 (Jul. 2016).
URL `http://arxiv.org/abs/1607.03250`

[42] J.-H. Luo, J. Wu, An Entropy-based Pruning Method for CNN Compression, arXiv:1706.05791ArXiv: 1706.05791 (Jun. 2017).
URL `http://arxiv.org/abs/1706.05791`

[43] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, K. Kavukcuoglu, Hierarchical Representations for Efficient Architecture Search, arXiv:1711.00436ArXiv: 1711.00436 (Nov. 2017).
URL `http://arxiv.org/abs/1711.00436`

[44] A. Krizhevsky, Learning Multiple Layers of Features from Tiny Images, Technical Report, University of Toronto (Apr. 2009).
URL `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`