A Multilayer Network-Based Approach to Represent, Explore and Handle Convolutional Neural Networks

COGNITIVE COMPUTATION
https://doi.org/10.1007/s12559-022-10084-6

A Multilayer Network Based Approach to Represent, Explore and Handle Convolutional Neural Networks

Alessia Amelio[1], Gianluca Bonifazi[2], Enrico Corradini[2], Domenico Ursino[2], and Luca Virgili[2*]
[1] InGeo, University "G. D'Annunzio" of Chieti-Pescara
[2] DII, Polytechnic University of Marche
* Contact Author
a.amelio@unich.it; g.bonifazi@univpm.it; e.corradini@pm.univpm.it; d.ursino@univpm.it; luca.virgili@univpm.it

Background: Deep learning techniques and tools have experienced enormous growth and widespread diffusion in recent years. Among the areas where deep learning has become more widespread there are computational biology and cognitive neuroscience. At the same time, the need for tools able to explore, understand, and possibly manipulate, a deep learning model has strongly emerged.

Methods: We propose an approach to map a deep learning model into a multilayer network. Our approach is tailored to Convolutional Neural Networks (CNN), but can be easily extended to other architectures. In order to show how our mapping approach enables the exploration and management of deep learning networks, we illustrate a technique for compressing a CNN. It detects whether there are convolutional layers that can be pruned without losing too much information and, in the affirmative case, returns a new CNN obtained from the original one by pruning such layers.

Results: We prove the effectiveness of the multilayer mapping approach and the corresponding compression algorithm on the VGG16 network and two benchmark datasets, namely MNIST, and CALTECH-101. In the former case, we obtain a 0.56% increase in accuracy, precision, and recall, and a 21.43% decrease in mean epoch time. In the latter case, we obtain an 11.09% increase in accuracy, 22.27% increase in precision, 38.66% increase in recall, and 47.22% decrease in mean epoch time. Finally, we compare our multilayer mapping

approach with a similar one based on single layers and show the effectiveness of the former.

Conclusions: We show that a multilayer network-based approach is able to capture and represent the complexity of a CNN. Furthermore, it allows several manipulations on it. An extensive experimental analysis described in the paper demonstrates the suitability of our approach and the goodness of its performance.

Keywords: Deep Learning; Convolutional Neural Networks; Multilayer Networks; Mapping CNNs into Multilayer Networks; Convolutional Layer Pruning

1 Introduction

In recent years, deep learning models have been introduced in different application areas for their ability to solve different kinds of optimization problem, such as document and text processing, object recognition in images and videos, image generation, speech and language recognition, and translation [1]. Due to the increasing complexity of these tasks, more demanding models have been required to achieve the best performances. Think, for instance, of residual networks, inception networks, and dense networks. These models require a lot of computational power with Graphical Processing Units for speeding up the time-consuming training process. In real-time decision making, a deep model with many parameters requires more time and resources to process its input, with further requirements in terms of energy and space. In mobile and edge devices, deep learning is a big opportunity but, at the same time, a big issue because, in this scenario, the computational power, storage capacity, and energy are limited [2, 3].

One of the most important features of deep learning systems is their ability to solve complex pattern recognition problems. Such patterns may involve images, signals, and sequences. These types of data are very common in many fields, such as biology and medicine. As a result, deep learning systems are having enormous popularity and success in contexts like computational biology and cognitive neuroscience [4, 5].

Several authors have begun to highlight the importance of reducing the size of deep networks and accelerating the models in terms of network structure and knowledge [6, 7]. Accordingly, most effort has been performed to introduce efficient and reduced-in-size ad-hoc deep network architectures, such as Mobile networks [8], SqueezeNet [9], ShuffleNet [10] and ESPNet [11]. Furthermore, different methods for reducing the model size whilst preserving the loss of performance have been proposed. The latter refers to four main categories of methods, i.e., *(i)* pruning, *(ii)* quantization, *(iii)* low-rank factorization, and *(iv)* knowledge distillation [3].

To maximize the effectiveness of these methods, it is extremely important to be able to explore the various layers and components of a deep learning architecture. Such exploration should allow the identification of the most

important components, the detection of interesting patterns and features, the tracking of information flow, the understanding of which parts of the network can be preserved, which can be replaced or removed, and so forth.

In this paper, we want to provide a contribution to this setting. We argue that complex networks, and specifically multilayer networks, can be a very useful tool to represent, analyze, explore and manipulate deep learning models. Based on this intuition, we first propose an approach to map deep learning networks into multilayer networks and then we use the latter to explore and handle the former. More specifically, we focus on one family of deep learning networks, namely Convolutional Neural Networks (hereafter, CNNs) [12], although the proposed approach could be extended to other ones. Multilayer networks [13] are sufficiently articulated to capture all aspects characterizing CNNs. Our mapping approach, which is the first main contribution of this paper, aims to map every aspect of a CNN (i.e., nodes, layers, filters, weights, etc.) in the four main components of a multilayer network, namely nodes, arcs, arc weights, and layers.

Once we have applied our mapping approach and obtained a multilayer network, representing a CNN, the latter can be used for several, both exploratory and manipulative, purposes. To give an idea of its potential, we will use it as a support structure for a convolutional layer pruning approach [14]. The objective of this approach is to identify whether there are layers of a CNN that can be pruned without losing too much information and, in the affirmative case, return a new CNN obtained from the original one by pruning these layers. Such a pruning approach represents the second main contribution of this paper.

In order to provide an overview of the steps of our approach, we report a corresponding workflow in Figure 1. As can be seen in this figure, the first step of the workflow consists of the creation of a multilayer network from a CNN trained on a suitable dataset. This task involves the computation of four elements, namely nodes, arcs, arc weights, and class networks. The multilayer network thus obtained is given as input to the next step, which performs its compression and returns a pruned version of it, with fewer layers and parameters. The pruned CNN represents the result of our approach.

Summarizing, the main contributions of this paper are as follows: *(i)* we propose a technique to map a CNN into a multilayer network; thanks to this technique, it is possible to explore and manage a CNN starting from the corresponding multilayer network-based representation; *(ii)* we present a technique to compress a CNN starting from its multilayer network-based representation; this technique allows for speeding up the training of the original CNN, as well as the extraction of information and knowledge from it.

2 Related Literature

In the last years, different approaches have been introduced in the literature for the pruning, quantization, and low-rank factorization of neural networks, as well as for extracting knowledge from them.

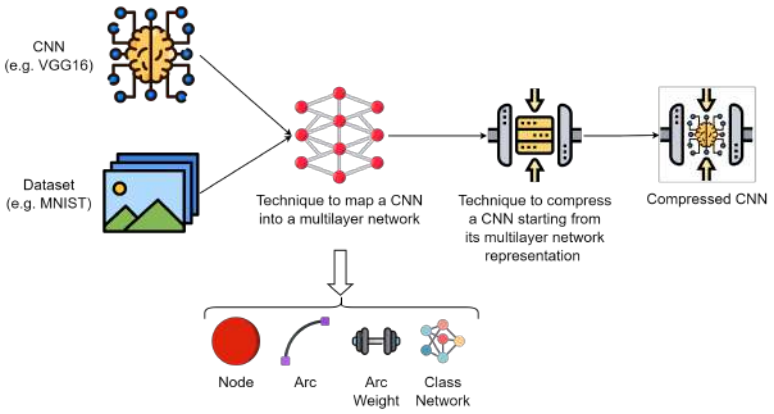


Fig. 1 Workflow of our approach to map a CNN into a multilayer network and then compress it

Pruning methods can be classified as: *(i) weight pruning*, where redundant connections, or connections having a weight below a threshold, are pruned; *(ii) neuron pruning*, where redundant neurons, together with incoming and outgoing connections, are pruned; *(iii) filter pruning*, where the least relevant filters, according to a given ranking, are pruned; *(iv) layer pruning*, where pruning of some layers is performed [3]. In this research area, the approach of [15] prunes network connections according to their impact on the training error. Specifically, it removes the connections units with the least impact on the error and, then, adopts the backpropagation algorithm for re-training the network. The approach of [16] also removes redundant neurons by pruning the weights providing the minimum change in the output activation of neurons. In [17], the authors propose an approach to handle sparsely connected networks. The approach described in this paper randomly deletes connections from a dense layer using a new sparse weight matrix. Furthermore, it proposes an efficient hardware architecture for reducing memory usage. In [18], the authors propose an approach for an efficient pruning of parameters based on the correlation between neuron activations in the inner layer and the increase in neuron correlation through additional output nodes. In the literature, several techniques for reducing the parameters of a fully connected layer have been also proposed. They replace this layer with an Adaptive Fastfood transform with non-linearity [19] and a global average pooling layer [20, 21]. In CNNs, pruning is performed by deleting redundant filters from convolutional layers and parameters of fully connected layers to reduce the storage and computational overhead of the network [3]. A new method for deleting redundant connections is proposed in [22]. It consists of two iterative steps, namely: *(i) pruning*, where redundant connections are deleted, and *(ii) splicing*, where deleted connections that are considered important are recovered. Also, the authors of [23] present an approach for pruning the convolutional layer filters having a low

ranking, computed according to their $L1$-norm, and weakly affecting model accuracy. They also show that the resulting model is fine-tuned/re-trained. In [24], the authors propose an approach that prunes and fine-tunes a network until a trade-off between accuracy and model size is obtained. Instead, the authors of [25] propose an approach that identifies the most relevant filters using the lasso regression method; then, it prunes irrelevant filters and reconstructs the output using unpruned filters and applying linear least squares. Inter-filter and intra-filter redundancy is investigated in [26], where the authors convert the operations of the convolutional layer into sparse matrix multiplication to process by means of a new efficient algorithm. In [27], the authors introduce a method based on gradual pruning of small magnitude weights in the training phase. Finally, the authors of [14] propose a method for pruning convolutional layers, which differs from the previous works on neuron, weight or filter pruning. Specifically, it identifies redundant connections based on the features learned in the convolutional layers and prunes the involved layers. Finally, other pruning methods propose to remove neurons using a statistical study of the derivatives of the model outputs with regard to each hidden neuron. In particular, the authors of [28] present a method to prune hidden neurons in a Multilayer Feedforward Network. They show that it is possible to preserve the performances on the validation and test set while decreasing the model complexity by deleting non-relevant neurons. Instead, the authors of [29] address this issue as a part of an approach aiming to improve recurrent network load forecasting.

As for quantization methods, they can be used during the training process, or after it, for an efficient inference [3]. In this context, the authors of [30] propose to use a hash function for randomly grouping weight connections into buckets; all the connections falling in the same bucket have the same weight; obtained weights are fine-tuned during the training process. In several studies, network weights are binary values (e.g., +1 and -1) during the training forward and backward phases [31] in such a way as to substitute multiply-accumulate operations with only accumulations. Instead, with quantized backpropagation [32], network weights fall in the ranges [-1, 0] and [0, +1]; in this way, multiplications in the forward and backward steps are avoided. An approach to also binarize activations is proposed in [33], whereas the authors of [34, 35] examine the effects of binarization and ternarization on the loss. In [36], the authors propose not only the quantization of weights and activations but also the stochastic quantization of gradients. The authors of [37] apply singular value decomposition on the filters of binarized CNNs to reduce both the parameters and the dimension of a network. The approach proposed in [38] first deletes redundant weight connections; then, it performs weight quantization using k-means; afterward, it applies a re-training step to improve accuracy; finally, it uses the Huffman coding on the quantized model to decrease its size. In [39], the authors introduce pruning at different granularity levels and scales. In order to detect candidates for pruning, they use particle filtering, where

the misclassification rate affects the weight of configurations. To further lower model size, they adopt a fixed-point optimization.

As for knowledge distillation, the authors of [40] prove that the knowledge acquired by a large model, for which the extraction of features is easy, can be moved to a smaller model to facilitate the deployment operation. Specifically, they adopt a temperature for creating the soft output from the teacher model. Then, they use this temperature for training the student model from teacher one with the objective of minimizing the error between the outputs of the two models. The authors of [41] propose a new method for training deep neural networks. Here, the training of the student model is driven by the middle layer of the teacher model, which is used not only for outputs but also to increase the accuracy of the student model. In [42], the authors paraphrase the knowledge of the teacher model into a simpler form using convolution operations involving the paraphraser and the translator. This knowledge is then moved onto a student model and allows the latter to learn more easily knowledge from the teacher model. The authors of [43] propose to match the gradients, instead of the soft outputs, for moving the knowledge from the teacher model to the student one. In [44], the authors propose an approach that generates a student model with low precision (quantization) from the knowledge of the teacher. Then, it uses stochastic gradient descent for optimizing the quantized elements to improve fitting with the teacher model. The authors of [45] introduce *On-the-fly Native Ensemble* (ONE). This method is first trained by creating a multi-branch variant of the target neural network through the addition of auxiliary branches. Then, it generates the teacher model as an ensemble of all the branches; each branch is trained using two loss terms, namely softmax cross-entropy loss and distillation loss. In [46], the information flow through a neural network is captured by a new representation called relational graph. The authors prove that the clustering coefficient and the average path length of this graph affect the neural network's predictive performance. Thus, there is a *sweet spot* of relational graphs leading to neural networks with an improved performance. Finally, in [47], the authors represent a social network graph as an artificial neural network and, then, explore the internal dynamics of the latter.

As far as the low-rank factorization is concerned, the authors of [48] propose to factorize the weight matrix of the final weight layer, of size $m \times n$ and rank r, in two matrices of size $m \times r$ and $n \times r$, to decrease the number of parameters of a deep neural network by a factor p, such that $p > \frac{r \cdot (m+n)}{m \cdot n}$. In order to decrease the number of computations in both the convolutional and the fully connected layers, the authors of [49] adopt a low-rank approximation; they obtain a noticeable reduction in terms of memory usage for the weights in both convolutional and dense layers. In order to decrease the number of parameters of a CNN, the authors of [50] use singular value decomposition, which decomposes a tensor for speeding up the deep neural network. The authors of [51] introduce a new approach for compressing a network model; it performs rank selection, low-rank tensor decomposition, and fine-tuning and

minimizes the tensor's reconstruction error. Low-rank decomposition of filters learned from scratch during training, instead of pre-training, is proposed in [52, 53]. Also, in [54], the authors introduce a new method for compressing and accelerating very deep CNNs; it performs network approximation in terms of non-linear units, instead of linear responses or filters. Generalized singular value decomposition, instead of stochastic gradient descend, has been used for solving non-linear problems. The authors of [55] propose a constrained-based optimization approach for detecting an optimal low-rank approximation of a trained CNN. The adopted constraints are the number of multiply-accumulate operations and the memory usage of the model.

In the past literature, different approaches have been introduced for interpreting deep learning models [56]. Some of them provide an explanation of the deep networks through the visualization and the localized inspection of high-level representations of graphs. In particular, the authors of [57] present ActiVis, which can visualize how neurons are activated by user-specified instances, or instance subsets, for explaining how a model generates its predictions. They provide a graph-based representation of the model allowing the local inspections of the activations at each layer codified as a node. The authors of [58] propose SUMMIT, an interactive tool aiming to detect relevant neurons and their relationships in the network. SUMMIT is based on an attribution graph that represents and summarizes important neuron connections and network substructures determining the model's outcome. In [59], the authors perform the interpretation of a CNN through the generation of an explanatory graph representing the hidden knowledge embedded in it. Each node of the graph corresponds to a part pattern codified in a filter, while each edge maps co-activation and spatial relationships between patterns. In [60], the authors propose to use a decision tree to interpret the role of filters in a convolutional layer for prediction; it also determines at which extent filters contribute to prediction and which object parts mainly affect the latter. The authors of [61] propose a method to build a model for hierarchical object recognition based on the semantics hidden in the convolutional layers of a pre-trained CNN. This method extracts an interpretable And-Or graph with four layers in order to explain the semantic hierarchy hidden in a CNN.

3 Mapping a Convolutional Neural Network into a multilayer network

In this section, we illustrate our technique for mapping a CNN into a multilayer network, which is the first part of the approach proposed in this paper. Specifically, in Subsection 3.1, we introduce the concept of class network, while, in Subsection 3.2, we describe how the class network is used to map the corresponding CNN into a multilayer network.

3.1 Class network definition

In this subsection, we provide a description of a CNN in terms of a single-layer network, called *class network*. Formally, a *class network* is a weighted directed graph $G = (V, E, W)$, where V is the set of nodes, E is the set of arcs, and W is the set of arc weights. The definition of each element of the class network is tailored to the specificities of the CNN architecture. Therefore, when we present the various elements of the former and their correspondences with the latter, we will provide a formalization of the CNN. It will help to better understand the way in which the class network elements have been defined.

3.1.1 Node definition

A CNN consists of M convolutional layers, each characterized by a number x of filters (also called "kernels"). In a convolutional layer, each filter slides over the input with a given stride to create a feature map. The input $\mathcal{I}$ is the original image for the first convolutional layer or a feature map for the next convolutional layers. The application of a filter at the position (i, j) of $\mathcal{I}$ (hereafter, $\mathcal{I}(i, j)$) provides a new element $\mathcal{O}(i, j)$ of the output feature map $\mathcal{O}$. Figure 2(a) shows the application of a filter of size 3×3 at position $\mathcal{I}(8, 8)$, whereas Figure 2(b) shows the new produced element $\mathcal{O}(8, 8)$ (red colored). The area of the filter is green colored.

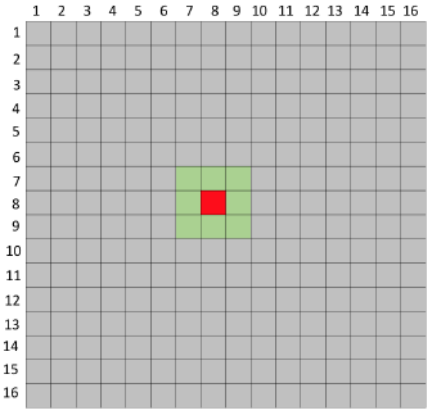
From all aforementioned, the set V of the nodes of G consists of a set $\{V_1, V_2, ..., V_M\}$ of node subsets. Here, the subset V_k denotes the contribution of the k^{th} convolutional layer c_k, obtained by applying the x_k filters of this layer to the input of c_k. Therefore, a node $p \in V_k$ represents the output obtained by applying the x_k filters of c_k at some position (i, j) of the input.

3.1.2 Arc definition

Since the application of a filter at $\mathcal{I}(i, j)$ generates a new element $\mathcal{O}(i, j)$ (see Figure 2), a direct connection between $\mathcal{I}(i, j)$ and $\mathcal{O}(i, j)$ is straightforward. Actually, in order to keep the context information, a direct connection is provided not only between $\mathcal{I}(i, j)$ and $\mathcal{O}(i, j)$, but also between $\mathcal{I}(i, j)$ and each element adjacent to $\mathcal{O}(i, j)$ within the filter area. Figure 3 shows the direct connections between $\mathcal{I}(8, 8)$, on which a filter of size 3×3 is applied, and the elements $\mathcal{O}(8 + a, 8 + b)$, $-1 \leq a \leq 1$ and $-1 \leq b \leq 1$.

Considering the set of x_k filters of a convolutional layer c_k, there are x_k sets of direct connections (like the ones of Figure 3) between $\mathcal{I}(i, j)$ and $\mathcal{O}_k(i, j)$, one for each filter x_k. In particular, Figure 4 shows the direct connections between $\mathcal{I}(8, 8)$, where three different filters of size 3×3 are applied, the new produced elements $\mathcal{O}_1(8, 8)$, $\mathcal{O}_2(8, 8)$ and $\mathcal{O}_3(8, 8)$ of the three feature maps, and their eight neighbors at positions $\mathcal{O}_h(8+a, 8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$ and $1 \leq h \leq 3$ of each feature map.

Since the set of x_k filters is applied to the input with a given stride, a set of similar connections towards the feature maps is generated for different positions of the input.



(a)

(b)

Fig. 2 (a) Application of a filter of size 3×3 (green colored) to $\mathcal{I}(8,8)$; (b) the new produced element $\mathcal{O}(8,8)$ (red colored)

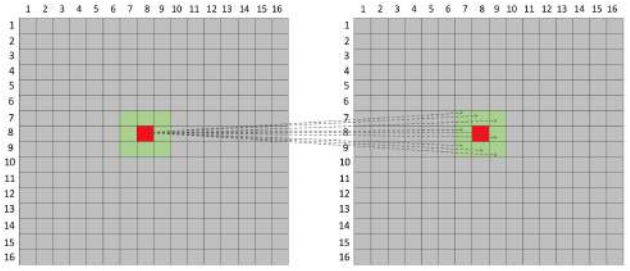


Fig. 3 Direct connections (dashed lines) between $\mathcal{I}(8,8)$ and the elements $\mathcal{O}(8+a, 8+b)$, $-1 \leq a \leq 1$ and $-1 \leq b \leq 1$ (red and green colored)

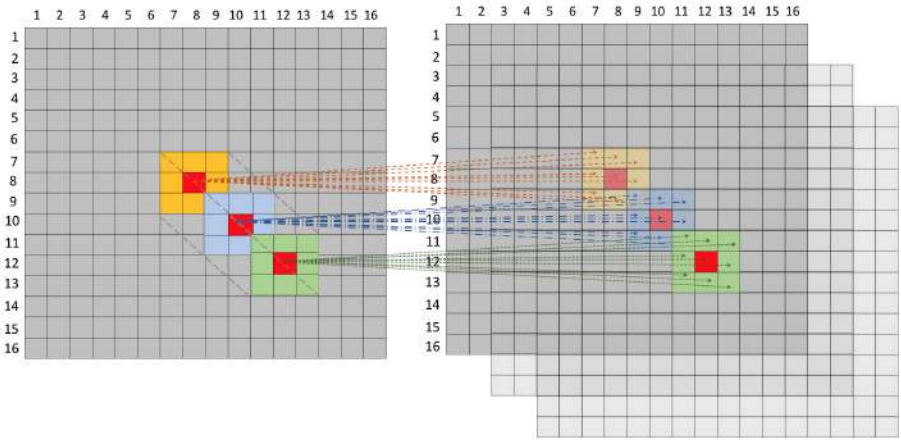


Fig. 4 Direct connections (dashed lines) between $\mathcal{I}(8,8)$ and $\mathcal{O}_h(8+a,8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$ and $1 \leq h \leq 3$, obtained by applying three filters (yellow, blue and green colored)

As for the pooling layer, it shrinks the input feature maps and, in our approach, leads to an increase in the number of connections between the input and the output. This is accomplished by sliding the filter over the input with a given stride and determining an aggregated value for each filter window[1]. Since aggregated values are anyway elements of the feature map provided in input, the next application of a convolutional layer to the feature map returned by the pooling layer generates connections between the aggregated values and the elements of the feature map returned by the convolutional layer. Specifically, direct connections exist between the aggregated values of the feature map provided in input to the pooling layer and the adjacent elements of the feature map generated by the next convolutional layer.

Figure 5 shows a sample procedure of direct connection generation for a pooling layer. On the left, a pooling filter of size 2×2 and stride 2 is applied to the input; the selected aggregated values are visible in the feature map as colored elements. On the right, the next application of a convolutional filter of size 3×3 at position $\mathcal{I}(2,2)$ generates 9 direct connections between $\mathcal{I}(2,2)$ and $\mathcal{O}(2+a,2+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$.

3.1.3 Weight definition

The application of a filter to the element $\mathcal{I}(i,j)$ generates a new element $\mathcal{O}(i,j)$, whose value is given by the following convolution operation:

$$g(i,j) = f(i,j) * \mathcal{I}(i,j) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} f(s,t)\mathcal{I}(i+s,j+t), \tag{1}$$

where f is the filter of size $(2a+1) \times (2b+1)$.

[1] An example of aggregated value could be the maximum.

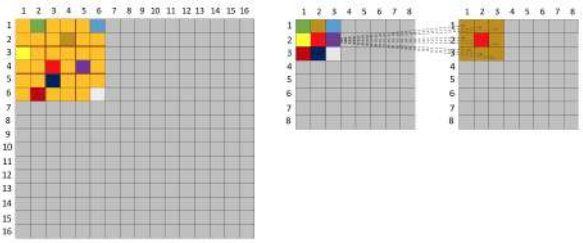


Fig. 5 Example of direct connection generation for a pooling layer. The filter is of size 2×2 with stride 2. For each filter application, a colored element corresponds to the aggregated value. At right, the 9 direct connections between the aggregated value at position $\mathcal{I}(2,2)$ and the elements $\mathcal{O}(2+a, 2+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$ are represented by dashed lines.

Accordingly, the direct connections generated between $\mathcal{I}(i,j)$ and $\mathcal{O}(i+s, j+t)$, $-a \leq s \leq a$, $-b \leq t \leq b$, are labeled with the weight $\mathcal{A}(g(i,j))$. This represents the result of the application of the activation function $\mathcal{A}()$ to the output $g(i,j)$ of the convolution operation. The activation function represents the last step in the creation of our feature map. Its application to the convolution result returns the final weights of the arcs. In the rest of this section, in order to not burden the examples, we will use the identity activation function, whose output is identical to the input, and thus to the convolution result, i.e. $\mathcal{A}(g(i,j)) = g(i,j)$. Figure 6 shows that the application of a filter of size 3×3 to $\mathcal{I}(8,8)$ returns $\mathcal{O}(8+a, 8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$. The weight is the convolution result $g(8,8) = f(8,8) * \mathcal{I}(8,8) = (0 \cdot 7) + (-1 \cdot 6) + (0 \cdot 6) + (-1 \cdot 4) + (5 \cdot 4) + (-1 \cdot 2) + (0 \cdot 3) + (-1 \cdot 3) + (0 \cdot 7) = 5$.

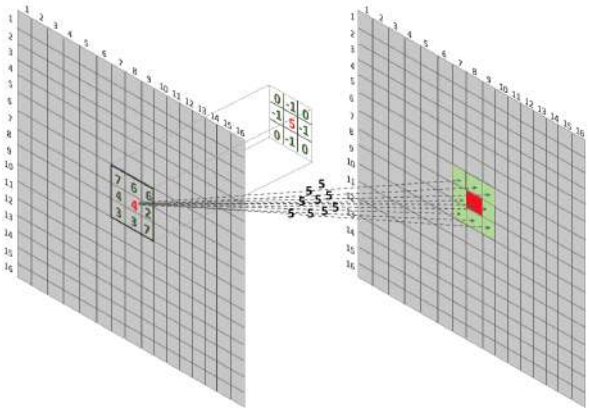


Fig. 6 Application of a filter of size 3×3 to $\mathcal{I}(8,8)$ (red colored) and computation of the weights for the direct connections between $\mathcal{I}(8,8)$ and $\mathcal{O}(8+a, 8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$

For a number x of filters, the direct connections between $\mathcal{I}(i,j)$ and $\mathcal{O}_h(i+a, j+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$, $1 \leq h \leq x$, are weighted with the results obtained by applying the activation function $\mathcal{A}()$ to the outputs of the corresponding convolution operation, i.e. $\mathcal{A}(g_1(i,j)), \mathcal{A}(g_2(i,j)), \ldots, \mathcal{A}(g_x(i,j))$.

Figure 7 shows the application of three filters of size 3×3 (green, blue and yellow colored, respectively) to $\mathcal{I}(8,8)$. It also reports, for each filter f_h, the weighted direct connections generated between $\mathcal{I}(8,8)$ and $\mathcal{O}_h(8+a, 8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$, $1 \leq h \leq 3$. The three weights $g_1(8,8)$, $g_2(8,8)$ and $g_3(8,8)$ are obtained as follows:

- $g_1(8,8) = f_1(8,8) * \mathcal{I}(8,8) = (0 \times 7) + (-1 \times 6) + (0 \times 6) + (-1 \times 4) + (5 \times 4) + (-1 \times 2) + (0 \times 3) + (-1 \times 3) + (0 \times 7) = 5$;
- $g_2(8,8) = f_2(8,8) * \mathcal{I}(8,8) = (-1 \times 7) + (-1 \times 6) + (0 \times 6) + (1 \times 4) + (3 \times 4) + (1 \times 2) + (0 \times 3) + (0 \times 3) + (-1 \times 7) = -2$;
- $g_3(8,8) = f_3(8,8) * \mathcal{I}(8,8) = (0 \times 7) + (0 \times 6) + (-1 \times 6) + (-1 \times 4) + (4 \times 4) + (-1 \times 2) + (1 \times 3) + (-1 \times 3) + (1 \times 7) = 11$.

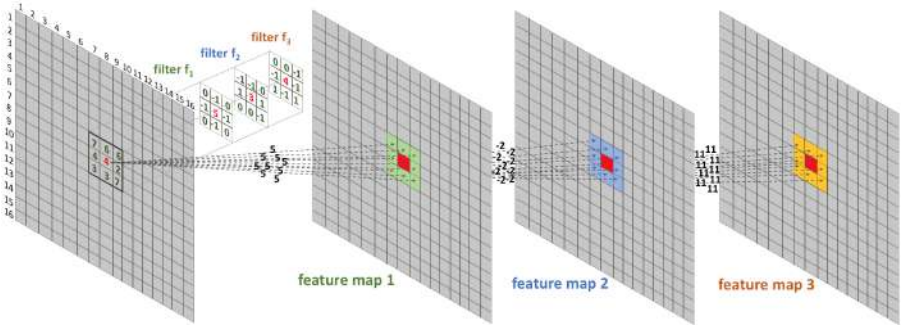


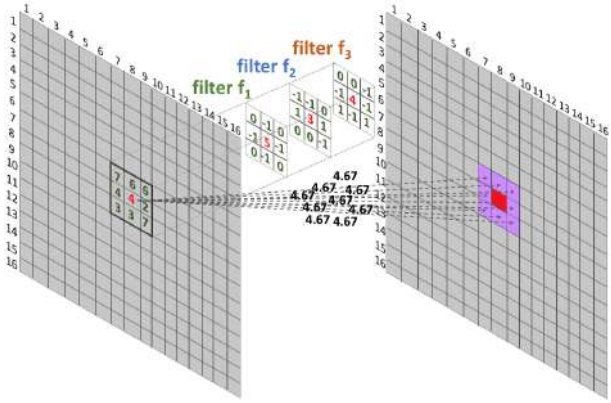
Fig. 7 Application of three filters of size 3×3 (green, blue and yellow colored, respectively) to $\mathcal{I}(8,8)$ and computation, for each filter, of the weights of the direct connections between $\mathcal{I}(8,8)$ and $\mathcal{O}_h(8+a, 8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$, $1 \leq h \leq 3$. More specifically, 5 is the result of the convolution between $\mathcal{I}(8,8)$ and the filter f_1, i.e. $5 = g_1(8,8) = f_1(8,8) * \mathcal{I}(8,8)$. Also, -2 is the result of the convolution between $\mathcal{I}(8,8)$ and the filter f_2, i.e. $-2 = g_2(8,8) = f_2(8,8) * \mathcal{I}(8,8)$. Finally, 11 is the result of the convolution between $\mathcal{I}(8,8)$ and the filter f_3, i.e. $11 = g_3(8,8) = f_3(8,8) * \mathcal{I}(8,8)$

In order to generate the arcs of the graph G from the weights of the direct connections of the x filters, we adopt some statistical descriptors. The goal is to have only one set of arcs from the node corresponding to $\mathcal{I}(i,j)$ to the node corresponding to $\mathcal{O}(i+s, j+t)$, $-a \leq s \leq a$, $-b \leq t \leq b$[2]. The weight of an arc from $\mathcal{I}(i,j)$ to $\mathcal{O}(i+s, j+t)$ is obtained by applying a suitable descriptor parameter to the weights of $\mathcal{A}(g_h(i,j))$, $1 \leq h \leq x$.

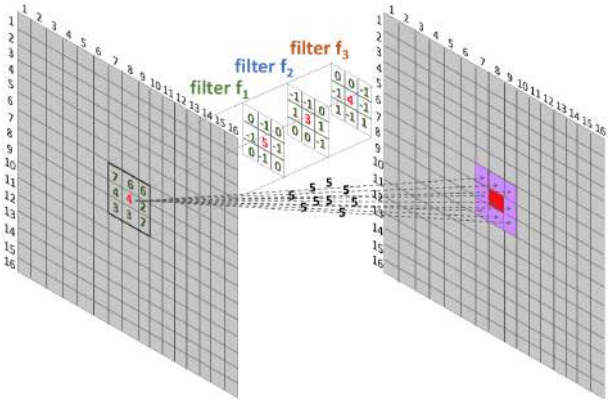
The two statistical descriptors used in this context are: *(i)* the mean, and *(ii)* the median, which revealed to achieve the best performance results. Accordingly, the weight of the direct connections from $\mathcal{I}(i,j)$ to $\mathcal{O}(i+s, j+t)$ can be obtained as:

$$g_{mean}(i,j) = \frac{\sum_{h=1}^{x} \mathcal{A}(g_h(i,j))}{x}, \tag{2}$$

[2]Here and in the following, we will use the symbols $\mathcal{I}(i,j)$ and $\mathcal{O}(i,j)$ to denote both the elements of the feature maps and the corresponding nodes of the class network.



(a) mean



(b) median

Fig. 8 Weights of the arcs from $\mathcal{I}(8,8)$ to $\mathcal{O}(8+a,8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$, obtained by applying the mean (on the left) and the median (on the right) as statistical descriptors of the corresponding direct connections

$$g_{median}(i,j) = \begin{cases} \mathcal{A}(g_{\lceil \frac{x}{2} \rceil}(i,j)) & \text{if } x \text{ is even} \\ \frac{\mathcal{A}(g_{\lceil \frac{x-1}{2} \rceil}(i,j)) + \mathcal{A}(g_{\lceil \frac{x+1}{2} \rceil}(i,j))}{2} & \text{if } x \text{ is odd} \end{cases} \tag{3}$$

where $g_{mean}(i,j)$ is the mean value, and $g_{median}(i,j)$ is the median value.

Figure 8 shows the direct connections between $\mathcal{I}(8,8)$ and $\mathcal{O}(8+a,8+b)$, $-1 \leq a \leq 1$, $-1 \leq b \leq 1$, whose weights are the mean value (see Figure 8(a)) and the median value (see Figure 8(b)) of the connections for the three filters depicted in Figure 7. The mean value is 4.67, whereas the median value is 5.

From all aforementioned, the set E of the arcs of G is a set of subsets $E = \{E_1, E_2, ..., E_{M-1}\}$. Here, E_k denotes the set of arcs connecting nodes of V_k to nodes of V_{k+1}. Analogously, the set W of the weights of G consists

of a set of subsets $W = \{W_1, W_2, ..., W_{M-1}\}$, where W_k is the set of weights associated with the arcs of E_k.

3.2 Mapping a CNN into a multilayer network

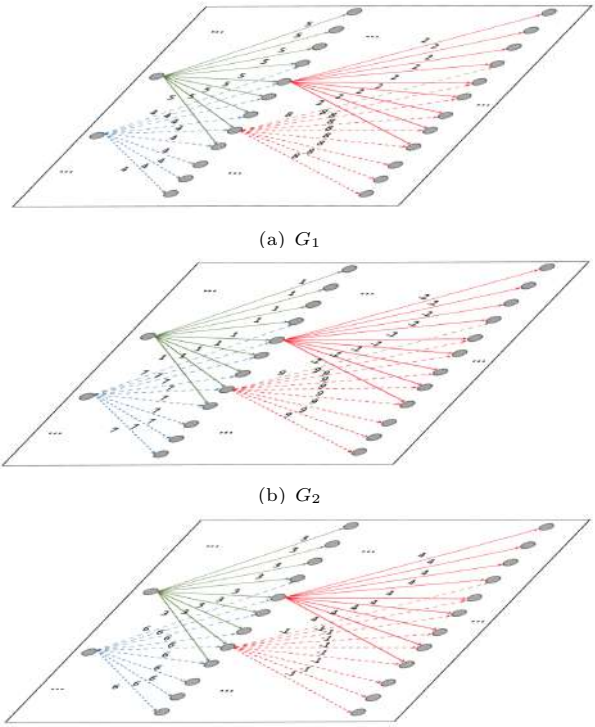
In the previous subsection, we have illustrated how it is possible to construct a class network representing a CNN. In this section, we show how, starting from the class network and a dataset on which the corresponding CNN is trained and tested, it is possible to construct a more complex structure called multilayer network.

Roughly speaking, a multilayer network is a set of t class networks, one for each target class of the dataset. Formally speaking, given a dataset D consisting of t target classes $Cl_1, Cl_2, \ldots, Cl_t$, and given a Convolutional Neural Network *cnn*, the multilayer network $\mathcal{G} = \{G^1, G^2, ...G^t\}$ corresponding to *cnn* is a set of t class networks. The class network G^h, $1 \leq h \leq t$, corresponds to the h^{th} target class of D. Figure 9 shows a sample multilayer network $\mathcal{G}$ characterized by three layers, each corresponding to a generated class network. The top (resp., middle, bottom) class network is denoted as G^1 (resp., G^2, G^3). Figure 10 shows the generation of a portion of the multilayer network $\mathcal{G}$ shown in Figure 9. It is obtained by extracting the feature maps of three target classes from *cnn*. In this last network a set of filters of size 3×3 with stride 1, sliding over positions $(3, 2)$ and $(4, 2)$ of the feature maps, generates some arcs for the class networks of $\mathcal{G}$. The weights of these arcs are computed as described in Section 3.1.3. In particular, the mean statistical descriptor is applied. Specifically, for the class network G^1 (resp., G^2, G^3), two sets of arcs of weights 5 (resp., 1, 3) and 4 (resp., 7, 6) are generated between the first and second feature maps, and the set of arcs of weights 2 (resp., -2, 4) and 8 (resp., 9, -1) are generated between the second and the third feature maps.

3.2.1 Algorithm for building the multilayer network

In this section, we illustrate our algorithm to construct a multilayer network from a CNN and a dataset D consisting of a set of t target classes. It consists of two steps, namely: *(i)* creation of a list of patch lists, which represents a support data structure for the next step; *(ii)* creation of the multilayer network from the list of patch lists. We have defined a suitable function for each of these steps.

The function corresponding to the first step, called CREATE_PATCHES, is reported in Algorithm 1. It receives a Convolutional Neural Network *cnn* and a target class Cl_h, $1 \leq h \leq t$, and constructs a list of patch lists. CREATE_PATCHES operates on the feature maps provided in input to each convolutional layer of *cnn*. A patch is a part of a feature map; in particular, it has the same size as the filters applied by the next convolutional layer and gives rise to a node in the multilayer network.



(a) G_1

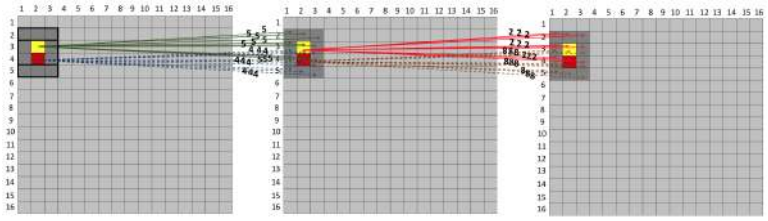
(b) G_2

(c) G_3

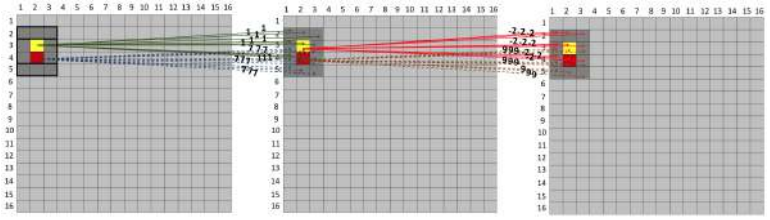
Fig. 9 Sample multilayer network $\mathcal{G}$ composed of three layers corresponding to the class networks G^1, G^2, and G^3.

CREATE_PATCHES proceeds as follows. It uses a list *conv_layers* that initially contains the sequence of the convolutional layers present in the reference CNN. It iterates over all elements of *conv_layers*, providing in input the images of Cl_h. During each iteration, it takes the current element as the *source* and the next element as the *target*. The feature map returned as output from *source* represents the input to *target*; the latter receives this input and processes it as specified below.

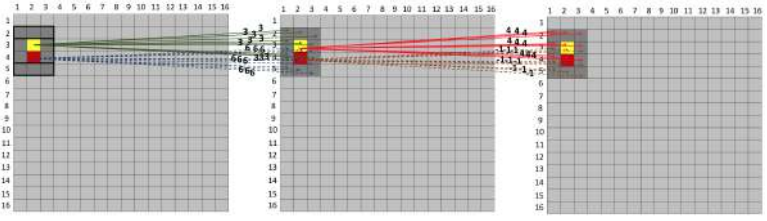
At the beginning of each iteration, CREATE_PATCHES determines the starting and ending points of the output of *source*. In particular, the starting (resp., ending) points img_{ws} (resp., img_{we}) and img_{hs} (resp., img_{he}) represent the center of the first (resp., last) application of the convolutional filters activated by *target* on the output of *source*. After this, CREATE_PATCHES iterates on the output of *source* and creates a patch for each application of the convolutional filter on an element. For each patch, it stores its identifier, the coordinates of its center, its width and height expressed in pixels, and the corresponding source and target. At the end of the iteration, it stores the patches corresponding to a convolutional layer in a list called *patch_list*.



(a) Class network G^1



(b) Class network G^2



(c) Class network G^3

Fig. 10 Generation of a portion of the class networks G^1, G^2 and G^3 corresponding to the layers of the multilayer network $\mathcal{G}$ depicted in Figure 9

The lists corresponding to all the convolutional layers of *cnn* are stored in a list of lists called *list_of_patch_lists*, which represents the output of CREATE_PATCHES.

The function corresponding to the second step, called CREATE_LAYER_NETWORK, is shown in Algorithm 2. It receives a Convolutional Neural Network *cnn* and a target class Cl_h, $1 \leq h \leq t$, and returns a layer (specifically, the h^{th} layer G^h) of the multilayer network $\mathcal{G}$.

First, CREATE_LAYER_NETWORK calls the function CREATE_PATCHES, described in Algorithm 1, which returns the list of patch lists corresponding to *cnn* when trained with the images of the target class Cl_h. Then, it creates an initially empty network G^h. Afterward, it iterates over the list of patch lists returned by CREATE_PATCHES considering two lists at a time. In particular, it considers the current list as *source* and the next one as *target*.

At the beginning of each iteration, CREATE_LAYER_NETWORK adds to G^h a node for each patch present in *source* and a node for each patch present in *target*, if they are not already present in G^h. Then, it computes a pair of

Algorithm 1 Function CREATE_PATCHES

Input

- *cnn*: a Convolutional Neural Network
- Cl_h: a target class
- *get_convolutional_layers*: a function that receives a CNN and returns the list of its convolutional layers

Output

- *list_of_patch_lists*: the list of the patch lists

```
1: function CREATE_PATCHES()
2:     list_of_patch_lists = ∅
3:     conv_layers = get_convolutional_layers(cnn)
4:     for i = 0 to len(conv_layers)-1 do
5:         patch_list = ∅
6:         source = conv_layers[i]
7:         target = conv_layers[i + 1]
8:         img_ws = ⌊target["filter_width"] / 2⌋
9:         img_hs = ⌊target["filter_height"] / 2⌋
10:        img_we = source["width"] − img_ws
11:        img_he = source["height"] − img_hs
12:        for i = img_ws to img_we do
13:            for j = img_hs to img_he do
14:                patch = (id, center, width, height, source, target)
15:                Add patch to patch_list
16:            end for
17:        end for
18:        Add patch_list to list_of_patch_lists
19:    end for
20:    return list_of_patch_lists
21: end function
```

parameters called w_{ratio} and h_{ratio}. In fact, as we have seen in Section 3.1.2, a pooling layer can exist between two consecutive convolutional layers, which reduces the image size. If this happens, the area covered by a filter in *target* is greater than the one covered in *source* (see Figure 5). The parameters w_{ratio} and h_{ratio} allow us to model this phenomenon, as we will see below.

At this point, CREATE_LAYER_NETWORK stores in w_{bound} (resp., h_{bound}) half the width (resp., height) in pixels of the filter associated with the convolutional layer. Indeed, as we have seen in Section 3.1.2, the application of convolutional filters is done with reference to the center of the filter, and therefore of the patch. For this reason, CREATE_LAYER_NETWORK iterates over the source and target nodes on which the filter acts and whose coordinates are determined from those of the center of the filter, its width, and its height. More specifically, given a node $node_t$ with coordinates $(node_{t_x}, node_{t_y})$, CREATE_LAYER_NETWORK considers all the nodes of *source* that can be processed by the filter whose center falls into the rectangle defined by the coordinates of the patch of $node_t$ (this rectangle is determined thanks to w_{bound} and h_{bound}) and, for each of them, adds an arc from it to $node_t$ in G^h.

Once this arc has been inserted, CREATE_LAYER_NETWORK computes the corresponding weights by applying the formulas seen in Section 3.1.2. For

this purpose, it first considers the feature map of *source* and selects the portion of this map corresponding to the inserted arcs. This portion consists of a rectangle comprised between the top left corner $(node_{s_x} - w_{bound}, node_{s_y} - h_{bound})$ and the bottom right corner $(node_{s_x} + w_{bound}, node_{s_y} + h_{bound})$. These two pairs of coordinates correspond exactly to the ones of the application of a filter to the patch of $node_s$, whose output is connected to $node_t$. Note that, for each arc, both the weight based on the mean (see Equation (2)) and the one based on the median (see Equation (3)) are stored.

At the end of its iterations, CREATE_LAYER_NETWORK has created the h^{th} layer G^h of the multilayer network $\mathcal{G}$; G^h corresponds to the target class Cl_h. Applying CREATE_LAYER_NETWORK t times, once for each target class of the dataset D, we obtain the final multilayer network.

We end this section by pointing out that Algorithms 1 and 2 are general and can be applied to many kinds of CNN.

4 Applying the multilayer network model to compress a CNN

In the previous section, we proposed an approach to map a CNN in a multilayer network. The network thus obtained represents the tool we use to analyze and manipulate the corresponding CNN. The operations that can be performed in a CNN thanks to the multilayer network thus obtained are many and various. To give an idea of their potential, in this section, we illustrate one of them, namely the compression of a CNN, which represents the second main contribution of this paper. First, we provide an informal description of the proposed compression approach, along with an example of its behavior. Then, we present its formalization through a pseudocode algorithm.

4.1 Methodology

Consider a multilayer network $\mathcal{G}$ and let G^h be its h^{th} layer. G^h consists of a weighted directed graph. Therefore, given a node v of G^h, we can define: *(i)* the *indegree* of v, as the sum of the weights of the arcs of G^h incoming into v; *(ii)* the *outdegree* of v, as the sum of the weights of the arcs outgoing from v; *(iii)* the *degree* of v, as the sum of its indegree and its outdegree. In the following, we use the symbol $d^h(v)$ to denote the degree of v in G^h. Instead, we use the symbol $\delta(v)$ to indicate the overall (weighted) degree of v in $\mathcal{G}$. As we will see below, $\delta(v)$ is an important indicator of the effectiveness of the filter represented by v.

As we have seen above, a multilayer network $\mathcal{G} = \{G^1, G^2, \ldots, G^t\}$ has a layer for each target class. As a consequence, we can think that the overall degree $\delta(v)$ of the node v in $\mathcal{G}$ can be obtained by suitably aggregating the degrees $d^1(v), d^2(v), \cdots, d^t(v)$ of v in the t layers of $\mathcal{G}$. More specifically:

$$\delta(v) = \mathcal{F}(d^1(v), d^2(v), ..., d^t(v)) \tag{4}$$

where $\mathcal{F}$ is an aggregation function.

Algorithm 2 Function CREATE_LAYER_NETWORK

Input

- *cnn*: a Convolutional Neural Network
- Cl_h: a target class
- *get_feature_maps*: a function that receives a CNN and a target class Cl_h and returns the list of the feature maps for each layer of the CNN when trained with Cl_h

Output

- G^h: the h^{th} layer of the multilayer network $\mathcal{G}$

function CREATE_LAYER_NETWORK()
  $f\_maps = get\_feature\_maps(cnn, Cl_h)$
  $list\_of\_patch\_lists = CREATE\_PATCHES(cnn, Cl_h)$
  $G^h = \emptyset$
  for $i = 0$ to len($list\_of\_patch\_lists$)-1 **do**
    $source = list\_of\_patch\_lists[i]$
    $target = list\_of\_patch\_lists[i+1]$
    Add nodes from $source$ and $target$ to G^h if they are not present therein
    $w_{ratio} = \frac{source["width"]}{target["width"]}$
    $h_{ratio} = \frac{source["height"]}{target["height"]}$
    $w_{bound} = \left\lfloor \frac{target["filter_{width}"]}{2} \right\rfloor$
    $h_{bound} = \left\lfloor \frac{target["filter_{height}"]}{2} \right\rfloor$
    for $node_t$ in $target$ **do**
      $(node_{tx}, node_{ty}) = node_t["center\_coordinates"]$
      for $node_s$ in $source$ **do**
        $(node_{s_x}, node_{s_y}) = node_s["center\_coordinates"]$
        if $(node_{s_x} - w_{bound}) \cdot w_{ratio} \leq node_{tx} \leq (node_{s_x} + w_{bound}) \cdot w_{ratio}$ **then**
          if $(node_{s_y} - h_{bound}) \cdot h_{ratio} \leq node_{t_y} \leq (node_{s_y} + h_{bound}) \cdot h_{ratio}$ **then**
            Add an arc from $node_s$ to $node_t$ in G^h
            $map = f\_maps[source["name"]]$
            Select the portion of f_maps starting from the top left corner $(node_{s_x} - w_{bound}, node_{s_y} - h_{bound})$ to the bottom right corner $(node_{s_x} + w_{bound}, node_{s_y} + h_{bound})$ and store it in $node_{map}$
            Add the mean and the median of $node_{map}$ as the weights of the arc from $node_s$ to $node_t$
          end if
        end if
      end for
    end for
  end for
  return G^h
end function

Let $d^{tot}(v) = \sum_{h=1}^{t} d^h(v)$ be the sum of the degrees of v in the t layers of $\mathcal{G}$. Our approach adopts the following entropy-based aggregation function [62, 63] for determining the overall degree $\delta(v)$ of v in $\mathcal{G}$:

$$\delta(v) = -\sum_{h=1}^{t} \frac{d^h(v)}{d^{tot}(v)} log\left(\frac{d^h(v)}{d^{tot}(v)}\right), \tag{5}$$

This function refers to the well-known concept of *information entropy* introduced by Shannon in [64]. The definition of $\delta(v)$ in Equation (5) favors a uniform distribution of the degree of v in the different layers while it penalizes the presence of a high degree of v in few layers. The rationale underlying it is

favoring those nodes whose feature extraction is balanced for different target classes [65] and penalizing those nodes that make a significant contribution in only few target classes.

Our compression approach aims to select a subset of the nodes of $\mathcal{G}$ with the highest values of the overall degree δ. Selecting such a subset allows us to determine the best convolutional layers that will form the compressed CNN. In particular, our approach selects the nodes of $\mathcal{G}$ whose values of overall degree δ are higher than a certain threshold:

$$th_{\delta} = \gamma \cdot \overline{\delta}, \tag{6}$$

Here, $\overline{\delta}$ is a statistical aggregation of the values of the overall degree δ of all nodes in $\mathcal{G}$. Our approach allows the adoption of two statistical aggregators, namely mean and median. Our choice fell on these two operators because they are the ones that allowed us to achieve the best experimental results. γ is a scaling factor whose value belongs to the real interval $[0, +\infty)$. It allows us to tune the contribution of $\overline{\delta}$.

After the subset of the nodes of $\mathcal{G}$ having an overall degree δ higher than th_{δ} has been selected, our approach determines the set of the convolutional layers from which these nodes are extracted. Then, it creates the compressed CNN from these convolutional layers. Once this task is completed, a new training task must be performed to adjust the weights of the compressed network.

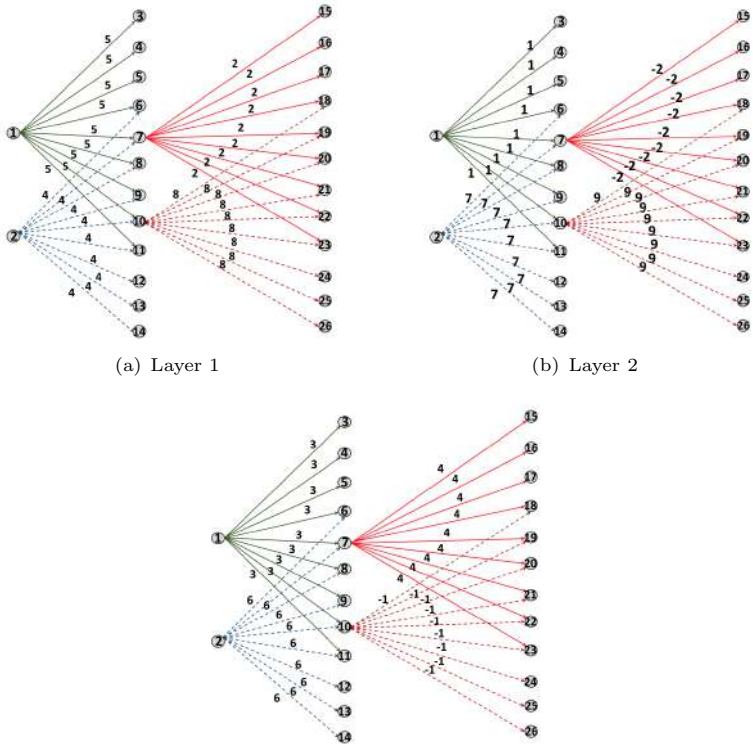
Figure 11 shows a flattened representation of a portion of the multilayer network $\mathcal{G} = \{G^1, G^2, G^3\}$ depicted in Figure 9. Here, all the nodes are numbered from 1 to 26.

In order to compute the threshold th_{δ}, $\overline{\delta}$ is first determined as the mean of the overall degrees δ of the nodes of $\mathcal{G}$. For a node v, $\delta(v)$ is computed according to Equation (5). For instance, $\delta(2)$ is computed as follows:

$$\begin{aligned}\delta(2) = &- \left[\frac{d^1(2)}{d^{tot}(2)} \cdot log\left(\frac{d^1(2)}{d^{tot}(2)}\right)\right] - \left[\frac{d^2(2)}{d^{tot}(2)} \cdot log\left(\frac{d^2(2)}{d^{tot}(2)}\right)\right] \\ &+ \left[\frac{d^3(2)}{d^{tot}(2)} \cdot log\left(\frac{d^3(2)}{d^{tot}(2)}\right)\right] = - \left[\frac{36}{153} \cdot log\left(\frac{36}{153}\right)\right] \\ &- \left[\frac{63}{153} \cdot log\left(\frac{63}{153}\right)\right] - \left[\frac{54}{153} \cdot log\left(\frac{54}{153}\right)\right] \\ &= 0.34 + 0.36 + 0.37 = 1.07\end{aligned} \tag{7}$$

where $d^1(2)$, $d^2(2)$ and $d^3(2)$ are the degrees of node 2 for G^1, G^2 and G^3, respectively, whereas $d^{tot}(2) = 153$ is the total degree of this node.

Analogously, the overall degree δ of the other nodes are: $\delta(1) = \delta(3) = \delta(4) = \delta(5) = 0.94$, $\delta(6) = \delta(8) = \delta(9) = \delta(11) = 1.09$, $\delta(12) = \delta(13) = \delta(14) = 1.07$, $\delta(18) = \delta(19) = \delta(20) = \delta(21) = \delta(22) = \delta(23) = 1.00$. Finally, nodes 7, 10, 15, 16, 17, 24, 25, and 26 do not give any contribution, since the *log* in Equation (5) has no meaning for negative numbers.



(a) Layer 1 (b) Layer 2

(c) Layer 3

Fig. 11 Flattened representation of a portion of the multilayer network $\mathcal{G}$ depicted in Figure 9

Observe that $\delta(2) = 1.07 < \delta(11) = 1.09$, because the degree distribution of node 11 (i.e., $d^1(11) = 9$, $d^2(11) = 8$, and $d^3(11) = 9$), is more balanced over the three network layers than the one of node 2 (i.e., $d^1(2) = 36$, $d^2(2) = 63$, and $d^3(2) = 54$) – see Figure 11.

The value of $\overline{\delta}$, adopting the mean as a statistical aggregator, is computed as follows:

$$\overline{\delta} = \frac{[(0.94 \cdot 4) + (1.07 \cdot 4) + (1.09 \cdot 4) + (1.00 \cdot 6)]}{26} = \frac{18.40}{26} = 0.71 \qquad (8)$$

Let the scaling factor γ be equal to 1.50. Then, the threshold th_δ will be computed as $th_\delta = \gamma \cdot \overline{\delta} = 1.50 \cdot 0.71 = 1.06$. The nodes with degree $\delta > th_\delta$ are 2, 6, 8, 9, 11, 12, 13, 14. They are located in the first and second feature maps (see Figures 10 and 11). Hence, the compressed CNN model consists of the first and second convolutional layers, while the third one is pruned from the model.

4.2 Approach formalization

In this subsection, we present the formalization of the CNN compression approach that we informally described in the previous subsection. The corresponding pseudocode can be found within the function COMPRESS_CNN shown in Algorithm 3.

This function receives: *(i)* the Convolutional Neural Network *cnn* that we want to compress; *(ii)* the multilayer network $\mathcal{G}$ associated with *cnn* and computed by applying the approach described in Section 3.2; *(iii)* the scaling factor γ that we have seen in Equation (6); *(iv)* the type $aggr_{type}$ of statistical aggregation function used in the computation of $\overline{\delta}$ in Equation (6). At present, the possible types are "mean" and "median".

It also uses some support functions, namely:

- *compute_overall_degrees*, which computes the value of the overall degree δ of each node of $\mathcal{G}$.
- *get_convolutional_layers*, which receives a CNN and returns the list of its convolutional layers.
- *mean* (resp., *median*), which receives the set of overall degrees δ of the nodes of $\mathcal{G}$ and computes $\overline{\delta}$ as their mean (resp., median).
- *remove_convolutional_layers*, which receives a Convolutional Neural Network *cnn* and a list *removable_layers* of convolutional layers to be pruned from it and returns a compressed version $\overline{cnn}$ of *cnn*, in which the layers of *removable_layers* have been pruned.

First, COMPRESS_CNN creates an empty list *preserving_layers*; it will contain all the layers of *cnn* to be preserved in $\overline{cnn}$. Then, it calls the function *compute_overall_degrees* to compute the list δ of the overall degrees of the nodes of $\mathcal{G}$. Afterwards, it computes the threshold th_δ by applying Equation (6) and taking $aggr_{type}$ into account.

At this point, for each node v of $\mathcal{G}$, COMPRESS_CNN checks whether its overall degree $\delta(v)$ is greater than th_δ. In the affirmative case, it adds the convolutional layer associated with v to the list *preserving_layers*. This way of proceeding implies that a convolutional layer is preserved if it has at least one node that provides a significant contribution to the operation of *cnn*.

Once all preserving layers have been identified, COMPRESS_CNN obtains the prunable layers by calling the function *get_convolutional_layers* with *cnn* as input and subtracting from the layers returned by it those present in the list *preserving_layers*.

After determining the layers to be pruned, COMPRESS_CNN calls the function *remove_convolutional_ layers* giving it *cnn* and *removable_layers* as input. The latter function prunes all the layers of *removable_layers* from *cnn* thus obtaining $\overline{cnn}$, which is also the output of COMPRESS_CNN.

Algorithm 3 COMPRESS_CNN

Input

- *cnn*: a CNN to compress
- $\mathcal{G}$: the multilayer network associated with *cnn*
- γ: a real number representing the scaling factor in the computation of th_δ
- $aggr_{type}$: the statistical aggregation function chosen for the computation of $\overline{\delta}$
- *compute_overall_degrees*: a function that computes the overall degree δ of each node of $\mathcal{G}$
- *get_convolutional_layers*: a function that returns the list of the convolutional layers of a CNN
- *median*: a function that returns the median of a list of values
- *mean*: a function that returns the mean of a list of values
- *remove_convolutional_layers*: a function that prunes a list of convolutional layers from a CNN

Output

- $\overline{cnn}$: a compressed version of *cnn*

```
1: function COMPRESS_CNN()
2:     preserving_layers = ∅
3:     δ = compute_overall_degrees(G)
4:     if aggr_type = "mean" then
5:         δ̄ = mean(δ)
6:     else if aggr_type = "median" then
7:         δ̄ = median(δ)
8:     end if
9:     th_δ = γ · δ̄
10:    for each node v of G do
11:        if δ(v) > th_δ then
12:            Add the convolutional layer associated with v to preserving_layers
13:        end if
14:    end for
15:    removable_layers = get_convolutional_layers(cnn) \ preserving_layers
16:    cnn̄ = remove_convolutional_layers(cnn, removable_layers)
17:    return cnn̄
18: end function
```

5 Experiments

In this section, we evaluate the performance of our approach. We pointed out that it is general and can be applied to several kinds of CNN. The interested reader can find the corresponding source code at the GitHub address https://github.com/lucav48/cnn2multilayer. In our experiments, we decided to apply it to VGG16 [66]; it is a benchmark vision CNN that won the ILSVR (ImageNet) competition in 2014. We tested the effectiveness of our approach in two well-known computer vision tasks, namely: *(i)* handwriting character recognition, and *(ii)* object recognition. The image datasets used for the experiments are MNIST and CALTECH-101, which are considered well-suited benchmarks for testing CNN architectures in these tasks [67, 68].

MNIST[3] consists of 60,000 training and 10,000 test images representing binary handwritten digits categorized into 10 target classes, which correspond to the digits from 0 to 9. Images from MNIST represent a portion of the larger NIST[4], a well-known dataset for evaluating computer vision and pattern recognition approaches. Each image of MNIST is 28×28 pixels in bitmap (`.bmp`) format, centered and size normalized.

CALTECH-101[5] is a dataset with 9,146 colored pictures of different objects belonging to 101 target classes (e.g. chair, ant, watch, pizza). Each class comprises from 40 to 800 images in JPEG (`.jpg`) format. The original size of each image is about 300×200 pixels but, in our tests, we scaled it to 128×128 pixels.

In our campaign, we first trained the VGG16 model from scratch. Then, we generated the layers of the multilayer network from the target classes of the dataset. For each target class, we computed the average of the feature maps over the image set before proceeding with the compression, and use them as the weights of the arcs.

As for MNIST, we used the predefined training and test sets and the predefined classes, i.e. 60,000 training and 10,000 test images categorized into 10 target classes. As for CALTECH-101, we performed a holdout validation that used 80% of the images for training and the remaining 20% of them for testing. Furthermore, from the training set of each dataset, we selected 90% of the images for training our model and the remaining 10% of them for validating it.

For tuning the hyperparameters of the VGG16 model, we randomly picked up its hyperparameters for 50 times. Each hyperparameter ranged in a suitable set. Specifically, we picked up the batch size in the set [16, 32, 64, 128], the learning rate η in the set [0.0001, 0.001, 0.01, 0.1], the optimizer in the set [Adam, SGD], and the epoch number in the range [100, 1000]. For each trial, we trained the VGG16 model with the selected hyperparameters on the training set and computed the performance measures of the compression algorithm on the validation set. In the end, we selected the model which obtained the best values of the performance measures on the validation set. Accordingly, we set: *(i)* the batch size equal to 128 for MNIST and 64 for CALTECH-101; *(ii)* the learning rate η equal to 0.0001; *(iii)* the Adam optimizer; *(iv)* the epoch number equal to 100. For limiting overfitting, we monitored validation loss and stopped training when no more changes occurred for three iterations.

We first performed a sensitivity analysis for studying the impact of the scaling factor γ on the compression performances of our approach when also varying the statistical aggregation function (i.e., mean and median) used in the computation of $\overline{\delta}$ (see Section 4.1). Then, we compared the results obtained by our approach, set with the best combination of γ and statistical aggregation function, with the ones obtained by a single-layer network-based approach on the same datasets. For this analysis, we used the test set of both datasets.

[3] http://yann.lecun.com/exdb/mnist/
[4] https://www.nist.gov/srd/nist-special-database-19
[5] http://www.vision.caltech.edu/Image_Datasets/Caltech101/

We carried out our experiments on Google Colab, which provided a GPU NVIDIA Tesla K80, 12 GB RAM, and 2 Intel Xeon CPUs 2.30GHz. The programming environment consisted of Python 3.7, Tensorflow 2.6, and NetworkX 2.6.3.

5.1 Reference Convolutional Neural Network

The VGG16 [66, 69] network is characterized by five convolutional blocks, named as $Conv_i$, $1 \leq i \leq 5$, five max pooling layers (red colored), and three fully connected layers (green colored), as shown in Figure 12. Each convolutional block consists of two or three convolutional layers with a given number of filters of size 3×3 and stride 1. One of the VGG16 configurations also includes convolutional filters of size 1×1, which can be considered as a linear transformation of the input channels. Padding in the convolutional layers with filters of size 3×3 preserves the spatial resolution after the convolution and is fixed to 1 pixel. The convolutional blocks are interleaved with max pooling layers with filters of size 2×2 and stride 2. This configuration of convolutional and max pooling layers is consistent throughout the whole network structure. It is followed by three fully connected layers; the first two have the same number of channels, while the third one has a number of channels equal to the number of target classes. The final part corresponds to the softmax layer, which returns the output. All the hidden layers are characterized by the rectification non-linearity (ReLU) which is the activation function for this network.

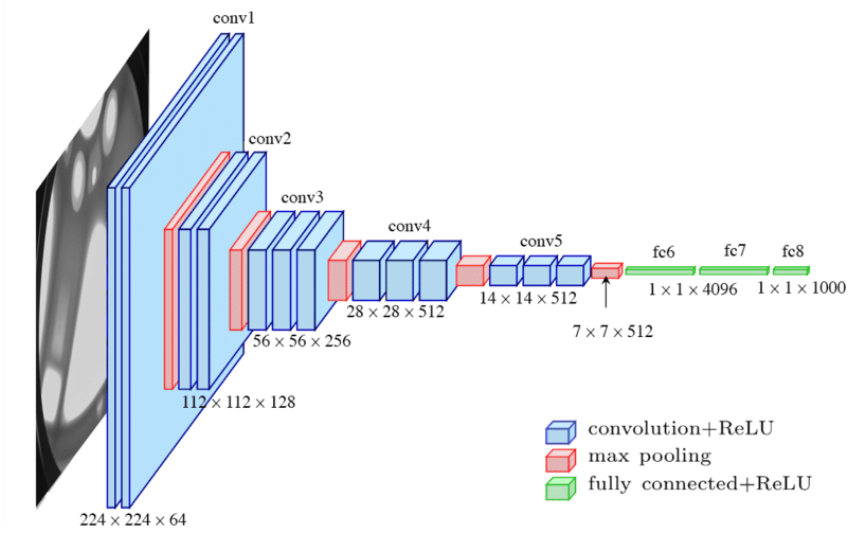


Fig. 12 The VGG16 network architecture

In the standard VGG16 network architecture, i.e., the one shown in Figure 12, the input is an RGB image of size 224×224. Also, the first two fully connected layers consist of 4,096 neurons, while the third one has 1,000 neurons, which correspond to the target classes of the ILSVRC-2012 dataset competition [66]. Finally, VGG16 has about 138 million parameters. In our specific case, the input to VGG16 can be an image with size different from 224×224. Thus, the number of parameters of our VGG16 is varied accordingly.

Similar to the VGG16 model presented above, another interesting Convolutional Neural Network is the VGG19 [66] model. In particular, VGG19 has three more convolutional layers than VGG16, which leads to more parameters to train and higher complexity of the corresponding architecture.

5.2 Obtained results

In this section, we describe the results obtained by performing several tests to validate our approach. Specifically, in Subsection 5.2.1, we describe the tuning activities performed on our approach, as well as the tasks carried out for computing its performance. Next, in Subsection 5.2.2, we compare our approach with an analogous one based on a single-layer network, instead of a multilayer one.

5.2.1 Tuning and performances

In Table 1 (resp., Table 2), we show some results obtained by VGG16 when the input is MNIST (resp., CALTECH-101). In particular, we report the values of accuracy, precision, recall, mean time per epoch (in seconds) of the CNN training phase, number of CNN parameters, and number of convolutional layers pruned by our compression method. The scaling factor γ varies from 0.25 to 2.00 with steps of 0.25. This revealed as the best working range for our approach. Recall that $\gamma = 0$ corresponds to no compression.

We report the values of the above metrics when the statistical aggregation function for computing $\overline{\delta}$ is the mean or the median and when the arc weight is based on the mean or the median. As a consequence, we have four possible combinations that we represent as $(\overline{\delta}_{mean}, g_{mean})$, $(\overline{\delta}_{median}, g_{mean})$, $(\overline{\delta}_{mean}, g_{median})$ and $(\overline{\delta}_{median}, g_{median})$, respectively.

Figure 13 (resp., Figure 14) shows the VGG16 convolutional layers preserved and pruned by our compression approach for the first (resp., last) two configurations when the input is MNIST. Figures 15 and 16 show the same data when the input is CALTECH-101. In these figures, green circles denote preserved layers whereas red crosses indicate pruned ones.

Note that the accuracy, precision, and recall of MNIST are higher than 0.97 for all configurations. In particular, the highest values of these parameters are obtained for the configuration $(\overline{\delta}_{median}, g_{mean})$ and γ greater than 0.75. In this case, the accuracy is higher than 0.99, whereas precision and recall are up to 0.99 when γ ranges between 1.00 and 1.50. With these configurations, our compression approach prunes the first two layers of $Conv_3$ and the first layer

Table 1 Accuracy, precision, recall, mean epoch time (in seconds) of the training phase, number of CNN parameters, and number of pruned convolutional layers obtained by our compression approach when MNIST is given in input to VGG16

		γ								
		0.00	0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00
Accuracy	$\overline{\delta}_{mean}, g_{mean}$	0.9865	0.9865	0.9857	0.9893	0.9893	0.9905	0.987	0.987	0.987
	$\overline{\delta}_{median}, g_{mean}$	0.9865	0.9877	0.9821	0.9821	0.9821	0.9855	0.9855	0.9855	0.9855
	$\overline{\delta}_{mean}, g_{median}$	0.9865	0.9865	0.9865	0.9857	0.9921	0.9921	0.9921	0.9905	0.9905
	$\overline{\delta}_{median}, g_{median}$	0.9865	0.9873	0.9873	0.9873	0.9873	0.9873	0.99	0.99	0.99
Precision	$\overline{\delta}_{mean}, g_{mean}$	0.9839	0.9838	0.9818	0.9846	0.9844	0.9876	0.9838	0.9842	0.9835
	$\overline{\delta}_{median}, g_{mean}$	0.9817	0.9852	0.9776	0.9775	0.9776	0.9827	0.9819	0.9816	0.9813
	$\overline{\delta}_{mean}, g_{median}$	0.9832	0.9835	0.9831	0.9814	0.989	0.9886	0.9882	0.987	0.986
	$\overline{\delta}_{median}, g_{median}$	0.9828	0.9839	0.9839	0.9842	0.9836	0.9837	0.9859	0.9858	0.9851
Recall	$\overline{\delta}_{mean}, g_{mean}$	0.9839	0.984	0.9822	0.9849	0.9851	0.9867	0.9827	0.9843	0.9828
	$\overline{\delta}_{median}, g_{mean}$	0.9821	0.9838	0.9794	0.9778	0.979	0.9813	0.9825	0.9814	0.9825
	$\overline{\delta}_{mean}, g_{median}$	0.9832	0.9835	0.9836	0.9819	0.9873	0.9882	0.9895	0.9862	0.9873
	$\overline{\delta}_{median}, g_{median}$	0.9831	0.9845	0.9835	0.984	0.983	0.9833	0.9859	0.9874	0.9874
#parameters	$\overline{\delta}_{mean}, g_{mean}$	33,640,010	33,640,010	31,280,202	29,214,794	29,214,794	29,067,210	28,698,186	28,698,186	28,698,186
	$\overline{\delta}_{median}, g_{mean}$	33,640,010	25,011,402	23,978,570	23,978,570	23,978,570	21,617,866	21,617,866	21,617,866	21,617,866
	$\overline{\delta}_{mean}, g_{median}$	33,640,010	33,640,010	33,640,010	31,280,202	30,100,042	30,100,042	30,100,042	29,067,210	29,067,210
	$\overline{\delta}_{median}, g_{median}$	33,640,010	28,920,394	28,920,394	28,920,394	28,920,394	28,920,394	28,330,314	25,232,842	25,232,842
Mean epoch time (s)	$\overline{\delta}_{mean}, g_{mean}$	56	56	52	41	41	38	35	35	35
	$\overline{\delta}_{median}, g_{mean}$	56	29	26	26	26	20	20	20	20
	$\overline{\delta}_{mean}, g_{median}$	56	56	56	52	44	44	44	38	38
	$\overline{\delta}_{median}, g_{median}$	56	47	47	47	47	47	44	32	32
#pruned layers	$\overline{\delta}_{mean}, g_{mean}$	0	0	1	4	4	5	6	6	6
	$\overline{\delta}_{median}, g_{mean}$	0	7	8	8	8	9	9	9	9
	$\overline{\delta}_{mean}, g_{median}$	0	0	0	1	3	3	3	5	5
	$\overline{\delta}_{median}, g_{median}$	0	2	2	2	2	2	3	6	6

Table 2 Accuracy, precision, recall, mean epoch time (in seconds) of the training phase, number of CNN parameters, and number of pruned convolutional layers obtained by our compression approach when CALTECH-101 is given in input to VGG16

		γ								
		0.00	0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00
Accuracy	$\overline{\delta}_{mean}, g_{mean}$	0.595	0.595	0.595	0.61	0.642	0.641	0.613	0.165	0.165
	$\overline{\delta}_{median}, g_{mean}$	0.595	0.595	0.595	0.595	0.617	0.627	0.607	0.617	0.617
	$\overline{\delta}_{mean}, g_{median}$	0.595	0.595	0.595	0.625	0.661	0.641	0.613	0.165	0.165
	$\overline{\delta}_{median}, g_{median}$	0.595	0.595	0.595	0.595	0.595	0.595	0.595	0.595	0.595
Precision	$\overline{\delta}_{mean}, g_{mean}$	0.431	0.431	0.431	0.45	0.492	0.509	0.438	0.031	0.031
	$\overline{\delta}_{median}, g_{mean}$	0.431	0.431	0.431	0.431	0.469	0.487	0.467	0.468	0.468
	$\overline{\delta}_{mean}, g_{median}$	0.431	0.431	0.431	0.494	0.527	0.509	0.438	0.031	0.031
	$\overline{\delta}_{median}, g_{median}$	0.431	0.431	0.431	0.431	0.431	0.431	0.431	0.431	0.431
Recall	$\overline{\delta}_{mean}, g_{mean}$	0.432	0.432	0.432	0.534	0.555	0.546	0.589	0.031	0.031
	$\overline{\delta}_{median}, g_{mean}$	0.432	0.432	0.432	0.432	0.495	0.503	0.494	0.508	0.508
	$\overline{\delta}_{mean}, g_{median}$	0.432	0.432	0.432	0.509	0.599	0.546	0.589	0.031	0.031
	$\overline{\delta}_{median}, g_{median}$	0.432	0.432	0.432	0.432	0.432	0.432	0.432	0.432	0.432
#parameters	$\overline{\delta}_{mean}, g_{mean}$	65,472,422	65,472,422	65,472,422	64,144,678	60,531,110	55,811,494	51,054,950	17,399,910	17,399,910
	$\overline{\delta}_{median}, g_{mean}$	65,472,422	65,472,422	65,472,422	65,472,422	65,435,494	65,287,910	60,204,646	57,844,838	57,844,838
	$\overline{\delta}_{mean}, g_{median}$	65,472,422	65,472,422	65,472,422	63,923,366	58,171,302	55,811,494	51,054,950	17,399,910	17,399,910
	$\overline{\delta}_{median}, g_{median}$	65,472,422	65,472,422	65,472,422	65,472,422	65,472,422	65,472,422	65,472,422	65,472,422	65,472,422
Mean epoch time (s)	$\overline{\delta}_{mean}, g_{mean}$	72	72	72	51	38	33	21	17	17
	$\overline{\delta}_{median}, g_{mean}$	72	72	72	72	60	51	32	29	29
	$\overline{\delta}_{mean}, g_{median}$	72	72	72	46	38	33	21	17	17
	$\overline{\delta}_{median}, g_{median}$	72	72	72	72	72	72	72	72	72
#pruned layers	$\overline{\delta}_{mean}, g_{mean}$	0	0	0	3	6	8	11	13	13
	$\overline{\delta}_{median}, g_{mean}$	0	0	0	0	1	2	6	7	7
	$\overline{\delta}_{mean}, g_{median}$	0	0	0	4	7	8	11	13	13
	$\overline{\delta}_{median}, g_{median}$	0	0	0	0	0	0	0	0	0

Layer ↓		$\overline{\delta}_{mean}$								$\overline{\delta}_{median}$							
	γ →	0.25	0.50	0.75	1	1.25	1.50	1.75	2	0.25	0.50	0.75	1	1.25	1.50	1.75	2
Conv1_Layer1 (28x28x64)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv1_Layer2 (28x28x64)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv2_Layer1 (14x14x128)		O	O	O	O	X	X	X	X	O	O	O	O	O	O	X	X
Conv2_Layer2 (14x14x128)		O	O	O	O	O	X	X	X	O	O	O	O	O	O	O	O
Conv3_Layer1 (7x7x256)		O	O	X	X	X	X	X	X	O	O	O	X	X	X	X	X
Conv3_Layer2 (7x7x256)		O	O	X	X	X	X	X	X	O	O	O	X	X	X	X	X
Conv3_Layer3 (7x7x256)		O	O	X	X	X	X	X	X	O	O	O	O	O	O	X	X
Conv4_Layer1 (3x3x512)		O	X	X	X	X	X	X	X	O	O	X	X	X	X	X	X
Conv4_Layer2 (3x3x512)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv4_Layer3 (3x3x512)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv5_Layer1 (2x2x512)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv5_Layer2 (2x2x512)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv5_Layer3 (2x2x512)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O

Fig. 13 Convolutional layers of VGG16 preserved and pruned by our compression algorithm for MNIST (arc weights based on mean)

Layer ↓		$\overline{\delta}_{mean}$								$\overline{\delta}_{median}$							
	γ →	0.25	0.50	0.75	1	1.25	1.50	1.75	2	0.25	0.50	0.75	1	1.25	1.50	1.75	2
Conv1_Layer1 (28x28x64)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv1_Layer2 (28x28x64)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv2_Layer1 (14x14x128)		X	X	X	X	X	X	X	X	O	O	O	O	O	O	X	X
Conv2_Layer2 (14x14x128)		X	X	X	X	X	X	X	X	O	O	O	O	O	O	O	O
Conv3_Layer1 (7x7x256)		X	X	X	X	X	X	X	X	O	O	O	O	O	O	X	X
Conv3_Layer2 (7x7x256)		X	X	X	X	X	X	X	X	O	O	O	O	O	X	X	X
Conv3_Layer3 (7x7x256)		O	X	X	X	X	X	X	X	O	O	O	O	O	O	O	O
Conv4_Layer1 (3x3x512)		X	X	X	X	X	X	X	X	O	O	O	O	O	O	X	X
Conv4_Layer2 (3x3x512)		O	O	O	O	X	X	X	X	O	O	O	O	O	O	O	O
Conv4_Layer3 (3x3x512)		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Conv5_Layer1 (2x2x512)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv5_Layer2 (2x2x512)		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
Conv5_Layer3 (2x2x512)		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Fig. 14 Convolutional layers of VGG16 preserved and pruned by our compression algorithm for MNIST (arc weights based on median)

of $Conv_4$. Pruning also the third layer of $Conv_3$ and the first layer of $Conv_2$ leads to a slight decrease in accuracy, precision, and recall. The configuration $(\overline{\delta}_{median}, g_{mean})$ guarantees the best performances but it also requires the highest mean epoch time and the highest number of parameters. It also leads

Layer ↓		$\overline{\delta}_{mean}$								$\overline{\delta}_{median}$							
	γ →	0.25	0.50	0.75	1	1.25	1.50	1.75	2	0.25	0.50	0.75	1	1.25	1.50	1.75	2
Conv1_Layer1 (128x128x64)		o	o	o	o	o	o	x	x	o	o	o	o	o	o	x	x
Conv1_Layer2 (128x128x64)		o	o	o	o	o	x	x	x	o	o	o	o	o	x	x	x
Conv2_Layer1 (64x64x128)		o	o	o	x	x	x	x	x	o	o	x	x	x	x	x	x
Conv2_Layer2 (64x64x128)		o	o	x	x	x	x	x	x	o	o	x	x	x	x	x	x
Conv3_Layer1 (32x32x256)		o	o	x	x	x	x	x	x	o	o	x	x	x	x	x	x
Conv3_Layer2 (32x32x256)		o	o	x	x	x	x	x	x	o	o	x	x	x	x	x	x
Conv3_Layer3 (32x32x256)		o	o	o	x	x	x	x	x	o	o	o	x	x	x	x	x
Conv4_Layer1 (16x16x512)		o	o	o	o	x	x	x	x	o	o	o	o	x	x	x	x
Conv4_Layer2 (16x16x512)		o	o	o	o	o	x	x	x	o	o	o	o	o	x	x	x
Conv4_Layer3 (16x16x512)		o	o	o	o	o	o	x	x	o	o	o	o	o	o	x	x
Conv5_Layer1 (8x8x512)		o	o	o	o	o	x	x	x	o	o	o	o	o	x	x	x
Conv5_Layer2 (8x8x512)		o	o	o	o	x	x	x	x	o	o	o	x	x	x	x	x
Conv5_Layer3 (8x8x512)		o	o	o	x	x	x	x	x	o	o	o	x	x	x	x	x

Fig. 15 Convolutional layers of VGG16 preserved and pruned by our compression algorithm for CALTECH-101 (arc weights based on mean)

Layer ↓		$\overline{\delta}_{mean}$								$\overline{\delta}_{median}$							
	γ →	0.25	0.50	0.75	1	1.25	1.50	1.75	2	0.25	0.50	0.75	1	1.25	1.50	1.75	2
Conv1_Layer1 (128x128x64)		o	o	o	o	o	x	x	x	o	o	o	o	o	o	o	o
Conv1_Layer2 (128x128x64)		o	o	o	x	x	x	x	x	o	o	o	o	o	o	o	o
Conv2_Layer1 (64x64x128)		o	o	o	o	x	x	x	x	o	o	o	o	o	o	o	o
Conv2_Layer2 (64x64x128)		o	o	o	o	o	x	x	x	o	o	o	o	o	o	o	o
Conv3_Layer1 (32x32x256)		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
Conv3_Layer2 (32x32x256)		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
Conv3_Layer3 (32x32x256)		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
Conv4_Layer1 (16x16x512)		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
Conv4_Layer2 (16x16x512)		o	o	o	o	o	x	x	x	o	o	o	o	o	o	o	o
Conv4_Layer3 (16x16x512)		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
Conv5_Layer1 (8x8x512)		o	o	o	o	o	x	x	x	o	o	o	o	o	o	o	o
Conv5_Layer2 (8x8x512)		o	o	o	o	o	o	x	x	o	o	o	o	o	o	o	o
Conv5_Layer3 (8x8x512)		o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o

Fig. 16 Convolutional layers of VGG16 preserved and pruned by our compression algorithm for CALTECH-101 (arc weights based on median)

to the lowest number of pruned convolutional layers that gradually increases when γ increases.

The configuration $(\overline{\delta}_{mean}, g_{mean})$ leads to lower values of accuracy, precision and recall than the previous configuration when γ is higher than 0.75. The highest values of these parameters are obtained for $\gamma = 1.25$. When γ

ranges between 0.75 and 1.25, $Conv_3$ and the first layer of $Conv_4$ are pruned. When $\gamma = 1.25$ also the first layer of $Conv_2$ can be pruned without a significant decrease in the performance values. Instead, if also the second layer of $Conv_2$ is pruned, the accuracy, precision, and recall start to decrease and become lower than 0.99. If compared with $(\overline{\delta}_{median}, g_{mean})$, this configuration leads to an average decrease in the mean epoch time of 4.9s and of the number of parameters of 1.06 million. It also leads to an increase in the number of pruned convolutional layers of 1.5.

Analogous trends can be observed for the configuration $(\overline{\delta}_{median}, g_{median})$. In this case, the values of accuracy, precision, and recall are up to 0.99 for γ between 1.50 and 2.00. This configuration leads to the pruning of the first layer of $Conv_2$ and $Conv_4$, the first two layers of $Conv_3$, and the third layer of $Conv_4$ and $Conv_5$. If compared with the configuration $(\overline{\delta}_{mean}, g_{mean})$, this one leads to an average increase in the mean epoch time of 1.25s, to an average decrease in: *(i)* the number of parameters of 1.89 million, and *(ii)* the number of pruned convolutional layers of 0.87.

Finally, as for the configuration $(\overline{\delta}_{mean}, g_{median})$, the accuracy, precision and recall reached the lowest values. For instance, the accuracy value is below 0.985 whereas precision and recall are up to 0.98 when γ ranges between 0.50 and 1.00. On the other side, this configuration leads to a rapid decrease in the mean epoch time below 30s and of the number of CNN parameters below 25 million; at the same time, there is an increase in the pruned layers up to 8. In this case, the compressed CNN model only preserves all the layers of $Conv_1$, the second layer of $Conv_4$, and the first two layers of $Conv_5$. Interestingly, pruning also the second layer of $Conv_4$ does not lead to a significant decrease in accuracy, precision and recall if γ is higher than 1. At the same time, this choice further reduces the mean epoch time and the number of CNN parameters.

As far as CALTECH-101 is concerned, the highest values of the performance measures are obtained when γ ranges from 0.75 to 1.50. More specifically, the highest value of accuracy (resp., precision, recall) is 0.661 (resp., 0.527, 0.599) and is reached when $\gamma = 1$ for the configuration $(\overline{\delta}_{median}, g_{mean})$. With this configuration, when γ ranges between 0.75 and 1.50, the mean epoch time and the number of CNN parameters gradually decrease, whereas the number of pruned convolutional layers rapidly increases. This trend can be observed in Figure 15, where the convolutional layers are rapidly pruned until the CNN model consists of the first layer of $Conv_1$ and the third layer of $Conv_4$. Interestingly, in spite of the presence of only two layers, the performance measures keep the same or higher values than the ones obtained with no compression (i.e., with $\gamma = 0$).

The configuration $(\overline{\delta}_{mean}, g_{mean})$ shows closely related trends with $(\overline{\delta}_{median}, g_{mean})$ in all measures.

By contrast, the performance measures of $(\overline{\delta}_{mean}, g_{median})$ diverge from the ones obtained by the two previous configurations. In fact, they show an increasing trend against γ and their value becomes even higher than the ones obtained with no compression when γ is high. On the other side, the mean

epoch time is on average 12.6s higher, the number of CNN parameters is 13.5 million higher and the number of pruned convolutional layers is 4.12 less than the previous two configurations. This is justified by the reduced number of convolutional layers pruned with this configuration, which gradually comprises the two blocks $Conv_1$ and $Conv_2$, the second layer of $Conv_4$ and the first two layers of $Conv_5$.

Finally, the configuration $(\overline{\delta}_{median}, g_{median})$ does not provide any compression, and all the six measures into consideration do not show any change against γ.

From all previous reasoning, we can conclude that the configuration guaranteeing the best tradeoff between costs and benefits is $(\overline{\delta}_{mean}, g_{mean})$ with $\gamma = 1.25$. In fact, in this case, the values of the performance measures are higher than the corresponding ones without compression for both MNIST and CALTECH-101. At the same time, the mean epoch time and the number of CNN parameters are acceptably low, whereas the number of pruned convolutional layers is high.

From a cross-comparison of the layer distributions, we can say that the best tradeoff between costs and benefits corresponds to pruning $Conv_2$, $Conv_3$, and the first layer of $Conv_4$.

Clearly, if we are not interested in the best tradeoff, but we are willing to sacrifice costs (i.e., to accept a high mean epoch time and a high number of CNN parameters) for maximizing benefits (i.e., high values of accuracy, precision, and recall) the best configuration is $(\overline{\delta}_{median}, g_{mean})$.

As a further analysis, we verified the effectiveness of our approach on VGG19 [66], trained on MNIST and CALTECH-101. The performance results we obtained are shown in Table 3. Here, for the sake of space, we report only the results obtained by our approach with the configuration $(\overline{\delta}_{mean}, g_{mean})$.

Table 3 Performance results obtained by our approach applied on VGG19, trained on MNIST and CALTECH-101- adopted configuration: $(\overline{\delta}_{mean}, g_{mean})$

Dataset	Model	γ	Accuracy	Precision	Recall	F1-Score	Mean epoch time (s)	Parameters
MNIST	VGG19	0	0.988	0.986	0.986	0.986	32	38,952,650
	Compressed VGG19	1	0.991	0.991	0.991	0.991	24	33,789,514
CALTECH-101	VGG19	0	0.565	0.412	0.45	0.430	44	70,782,118
	Compressed VGG19	1.25	0.619	0.483	0.547	0.513	21	56,180,262

From the analysis of this table, we can observe that our compression approach works well also in this case. As for the MNIST dataset, we obtained a compressed VGG19 that has slightly higher accuracy, precision and recall than the original one, while having fewer parameters (-13.2%) and a lower training time (-25.0%) than the latter. Similar reasoning can be made for the CALTECH-101 case; here, the compressed VGG19 shows a higher accuracy, precision, and recall than the original one, while being a smaller and faster model (-20.6% in the number of parameters and -52.2% in the mean epoch time) than it.

Finally, we measured the computation time needed to compress VGG16 and VGG19, as well as the inference time for a single prediction. We computed all these parameters for both the original and compressed models. The results obtained are reported in Table 4.

Table 4 Computation time needed by our approach to compress VGG16 and VGG19 when it is trained on MNIST and CALTECH-101; inference time required for a single prediction by the original model and the compressed one

	CNN	*Compression time (s)*	*Inference time of the original model (ms)*	*Inference time of the compressed model (ms)*
MNIST	VGG16	10.28	25.2	14.9
	VGG19	11.78	32.0	20.5
CALTECH-101	VGG16	412.45	44.1	24.1
	VGG19	437.67	61.2	35.3

From the analysis of this table we can observe that, as for the MNIST case, the compression time for both VGG16 and VGG19 is low and acceptable. Furthermore, as for the inference time, we obtain compressed models that are 35-40% faster than the original ones. As for the CALTECH-101 case, the compression time is higher because this dataset has many more classes than MNIST (102 against 10), and its images have a higher resolution ($128 \times 128 \times 3$ against $28 \times 28 \times 1$). However, we point out that the compression time includes the building of the multilayer network corresponding to the CNN, which is a one-time operation for any value of the threshold γ. Instead, as for the inference time, the compressed models are 42-45% faster than the original ones.

5.2.2 Comparison results

In order to highlight the importance of adopting a multilayer network for supporting the representation and manipulation of CNNs, we compare our compression method with an analogous one based on a single-layer network. This approach considers each target class as a single contribution to the CNN compression. As a consequence, given a class network G^h, it first computes $\overline{\delta^h}$, $1 \leq h \leq t$, as a statistical aggregation of the values of the degree δ of all the nodes of G^h. Then, it calculates $th_\delta^h = \gamma \cdot \overline{\delta^h}$, $1 \leq h \leq t$. Afterwards, it selects the subset of the nodes of G^h, $1 \leq h \leq t$, having a degree δ higher than th_δ^h. Finally, it determines the set of nodes from which the convolutional layers of the compressed CNN are extracted by computing the intersection of these t subsets.

Table 5 shows the performance results obtained by VGG16 when the input datasets are MNIST and CALTECH-101. In particular, we report the values of accuracy, precision, recall, F1-Score, mean time per epoch (in seconds) of the CNN training phase, number of CNN parameters, and number of convolutional layers pruned obtained by applying our multilayer network-based (indicated by "m") compression approach and the single layer network-based one (denoted by "s") described above. In order to make the comparison as objective as possible, we identified the best configuration also for the single-layer network-based approach and adopted it in the comparison. This configuration states

that the statistical aggregation function adopted in the computation of $\overline{\delta^h}$, $1 \leq h \leq t$, is the mean, the arc weight is based on the mean (g_{mean}), and $\gamma = 1.25$.

Table 5 Performance results obtained by our approach based on a multilayer network (m) and a single-layer one (s) when applied on VGG16, and MNIST and CALTECH-101 are provided in input

Performance measures	Accuracy		Precision		Recall		F1-Score		Mean epoch time (s)		#parameters		#pruned layers	
multilayer (m) single-layer (s)	m	s	m	s	m	s	m	s	m	s	m	s	m	s
MNIST	0.990	0.987	0.988	0.984	0.987	0.983	0.987	0.983	38.0	44.1	$2.9 \cdot 10^7$	$3.1 \cdot 10^7$	5	3
CALTECH-101	0.641	0.51	0.509	0.315	0.546	0.326	0.527	0.320	33.1	19.0	$5.6 \cdot 10^7$	$2.5 \cdot 10^7$	8	12

As for MNIST, we observe that the compression method based on a multilayer network obtains the highest values of accuracy, precision, recall, and F1-Score, whereas the mean epoch time is 6s lower, the number of CNN parameters is 2 million lower, and the number of pruned convolutional layers is 2 more than the ones obtained by adopting a single-layer network.

As for CALTECH-101, we observe that the compression method based on a single-layer network obtains better values of mean epoch time, number of CNN parameters, and number of pruned convolutional layers than the one based on a multilayer network. However, the former leads to much lower results than the latter for accuracy (0.641 against 0.510), precision (0.509 against 0.315), recall (0.546 against 0.326), and F1-Score (0.527 against 0.320).

After comparing our compression approach with an analogous one based on a single-layer network, we went on to perform a theoretical analysis and a qualitative comparison between our approach and others that have been shown to work very well in the area of pruning-based compression. We have chosen this category of compression approaches because it is the one to which our approach belongs. Table 6 reports the results of our theoretical analysis considering four comparison features, namely pruning type, re-training type, pruning rate, and training after pruning, as suggested in [3].

Table 6 Theoretical analysis of our compression approach and other related ones based on pruning

Approach	Pruning type	Re-training type	Pruning rate	Pruning after training
Our approach	Layers	Fine-tune	Fixed	Yes
Li et al. (2016) [23]	Filters	Fine-tune	Fixed	Yes (iteratively)
Han et al. (2015) [70]	Connections	Fine-tune	Fixed	Yes (iteratively)
Liu et al. (2017) [71]	Filters	Fine-tune	Fixed	Yes (iteratively)

From the analysis of Table 6, we can observe that our approach is the only one capable of removing convolutional layers and not individual filters or connections. Another difference concerns the training of the CNN after pruning. In fact, our approach trains the CNN only once after pruning; this is achieved by selecting the most relevant nodes in the multilayer representation of a CNN. The approach of [70] trains the CNN to learn which connections

are relevant and then prunes the irrelevant ones. The approach of [23] is based on an acceleration method for a CNN, in which filters having a small effect on the accuracy of the output are pruned. Finally, the approach of [71] detects insignificant filters during the training task and then prunes them.

Furthermore, we highlight that our compression approach represents only one possible application of the idea of modeling a CNN through a multilayer network. In fact, this new representation of a CNN allows the application of concepts and approaches typical of Complex Network Analysis to the context of deep learning. Thanks to this way of proceeding, it is possible to perform a variety of analyses and applications on CNNs with the advantage of not having to treat them as black boxes. In fact, unlike many other CNN investigation approaches already proposed in the literature, our approach's capability of mapping all the constructs of a CNN while preserving all their features allows for explanations of how information flows through the various layers of the CNN, shedding a light on the black box characterizing it.

In Table 7, we report a quantitative comparison between our compression approach and the other three ones already examined in Table 6. We adopted Top-1 accuracy as performance measure and MNIST [3] as the dataset for training and testing. As for our approach, we used the best configuration, namely $(\overline{\delta}_{mean}, g_{mean})$, and we set $\gamma = 1$ for VGG19, and $\gamma = 1.25$ for VGG16.

Table 7 Quantitative comparison between our compression approach and other related ones based on pruning

Approach	CNN	Top-1 accuracy	Note
Our approach	VGG16	0.990	$(\overline{\delta}_{mean}, g_{mean})$, $\gamma = 1.25$
	VGG19	0.991	$(\overline{\delta}_{mean}, g_{mean})$, $\gamma = 1$
Li et al. (2016) [23]	AlexNet	0.985	Filters pruned in the first layer: 0%, in the other layers: 50%
		0.986	Filters pruned in the first layer: 30%, in the other layers: 50%
		0.987	Filters pruned in the first layer: 0%, in the other layers: 60%
Han et al. (2015) [70]	AlexNet	0.985	Conv. pruned: 60%, FC pruned: 70% (L1-norm)
		0.984	Conv. pruned: 60%, FC pruned: 70% (L1-norm, BND)
Liu et al. (2017) [71]	AlexNet	0.986	Filters pruned: 40%
		0.988	Filters pruned: 50%

From the analysis of Table 7, we can observe that our approach achieves the highest Top-1 accuracy for both VGG16 and VGG19 (the corresponding values are 0.990 for VGG16 and 0.991 for VGG19). As for the approach of [23], the best configuration prunes all filters in the first layer and 60% of filters in the next layers. It achieves a Top-1 accuracy equal to 0.987. As for the approach of [70], the authors propose two configurations. The first prunes 60% of the convolutional layers connections and 70% of the fully connected (FC) layers connections. It then re-trains the compressed model with an L1-norm. The second configuration performs the same pruning as the first one but trains the model with an L1-norm using Batch Normalization and Dropout (BND). The value of Top-1 accuracy obtained with these two configurations is very similar; in fact, it is 0.985 with the first configuration and 0.984 with the second one. Finally, the approach of [71] obtains a Top-1 accuracy value equal to 0.988 with 50% of pruned filters, and a Top-1 accuracy value equal to 0.986 with 40% of pruned filters.

5.3 Discussion

In this section, we draw some considerations on the approaches presented in this paper. First, we observe that our approach to map a CNN into a multilayer network is general and can be applied to most classical CNN architectures, such as LeNet, AlexNet, GoogLeNet, VGG19, and so forth.

Another advantage of our multilayer network-based representation is that it provides important insights about what is happening under the hood of a CNN. In fact, it allows us to identify the best performing nodes and describe how they interact with their neighbors. We can observe how information flows through the arcs of the multilayer network, which is a representation of how the corresponding CNN filters process images. The ability of our approach to identify the most important convolutional layers of a CNN derives exactly from this observation capability. In turn, the ability to identify the most important convolutional layers represents the starting point for several tasks, one of which is the compression of a CNN that we have seen in detail in this paper.

Our compression approach prunes entire layers; this reduces the number of parameters to be trained and speeds up both training and inference tasks. Interestingly, pruning a whole layer does not disrupt the nature of a CNN model, as it could happen with the cutting of redundant connections (see Section 2).

As for compression performance, we have already seen in Table 7 that our approach is able to provide very satisfactory results in terms of Top-1 accuracy. In fact, the value of this parameter is 0.990 for VGG16 and 0.991 for VGG19. As we have seen in Section 5.2.2, this value is comparable and, actually, slightly higher than that obtained by the other related pruning-based compression approaches. Indeed, as seen in Table 7, the latter are able to guarantee a Top-1 accuracy ranging from 0.984, in the case of the approach described in [70], to 0.988, in the case of the approach described in [71]. Another interesting strength of our compression approach concerns the difference between the value of Top-1 accuracy before and after compression. Specifically, with our approach, Top-1 accuracy increases by 0.4% after compression. In other words, if compression is performed by means of our approach, it results in an improvement in classification results. In contrast, the other three approaches examined above lead to a decrease in Top-1 accuracy after compression. Specifically, in the approach described in [23] (resp., [70], [71]), Top-1 accuracy decreases by 0.02% (resp., 0.15%, 0.04%) after compression.

We can draw further considerations by observing which layers are typically pruned by our approach. Regarding this, we must preliminarily recall an important behavior typical of CNNs, i.e., the fact that the first convolutional layers extract high-level patterns from images (e.g., the shape of a dog), while the last ones focus on in-depth patterns (e.g., details of the dog, like its ears or its nose). From Figures 13 - 16, we can see that our compression approach does not generally prune the first convolutional layers, which are the ones close to the input; in fact, this pruning activity happens only when γ is high. One reason for this behavior concerns the fact that these layers probably extract

generic patterns that are always useful for classifying input images. This characteristic makes them essential, and so they are hardly pruned by our approach. Instead, the most pruned layers are the middle ones. This could depend on the excessive number of convolution operations applied on a fine feature map. Indeed, it could happen that the first 3-4 convolutional layers extract a meaningful pattern from images, which then undergoes other 7-8 convolutions and loses its significance. Finally, the last layers of VGG16, i.e., the ones close to the classification output, are pruned less than the middle layers but more than the initial ones. The reason for this behavior is not so obvious and requires further study in the future.

Although our multilayer network-based representation has several interesting properties, it still has some limitations. For instance, it does not consider residual connections, typical of ResNet architectures; therefore, ResNet cannot be represented through it. However, this kind of connection can be easily added to our model, and we plan to make this task in the next future. Another limit concerns the view of the filters of a convolutional layer as a single unit. Indeed, our mapping approach aggregates all the computation results of the convolutional filters through a mean or a median. This aggregation keeps the multilayer network size low, but it loses information about single filters. Of course, this is a tradeoff between the computation time required to process a larger multilayer network and the need for an in-depth analysis. However, this implies that we cannot prune any filter from the convolutional layers because we do not have the corresponding data within the multilayer network model. Actually, this issue can be addressed by making the multilayer network bigger, and then by developing a compression algorithm (similar to the one proposed here) to identify the most performing filters from the convolutional layers of a CNN.

Another limit concerns the datasets we employed here. MNIST and CALTECH-101 are surely two important benchmarks, heavily used by the research community. However, both of them are not as big as ImageNet, CALTECH-256, and CIFAR100, which are much more complex and require much more computational power. We are confident that our approach can show good performances also with datasets like these, but it surely will be interesting to verify if this conjecture is true.

6 Conclusion

In this paper, we have proposed an approach for representing, exploring and handling a CNN thanks to the possibility of using concepts and techniques derived from graph theory. To achieve this goal, our approach employs two techniques. The first aims to map a CNN into a multilayer network. In particular, it maps each element of a CNN into the constructs of a multilayer network, such as nodes, arcs, arc weights, and layers. Then, starting from the multilayer network representation thus obtained, the second technique determines the redundant convolutional layers of a CNN and removes them in such

a way as to obtain a pruned version of it. Thanks to this way of proceeding, we can reduce the complexity of a CNN model while preserving good performances. Finally, we presented an extensive experimental campaign aimed to show the suitability of the proposed approach and to evaluate its performance. In particular, we tested our multilayer network representation on VGG16 and VGG19, trained on MNIST and CALTECH-101 datasets. The obtained results are interesting since we were able to reduce the number of parameters, as well as the training and inference times of the CNN models while keeping good performances.

The possibility to compress CNNs and, more generally, deep learning systems can be extremely important in areas related to biology and cognitive computing. In fact, these are two areas that can benefit greatly from deep learning systems and techniques, as evidenced, for instance, in [4, 5].

The concepts and techniques proposed in this paper certainly represent a point of arrival in our research. However, at the same time, they are a starting point for further efforts in this research field. For instance, we plan to extend our approach in such a way as to represent the residual connections typical of ResNet architectures. Following a similar reasoning, we can think of mapping a Recurrent Neural Network (i.e., RNN) into a multilayer network. Specifically, in this case, each layer could represent a dataset class, while a node in a layer could represent an RNN cell (think, for instance, of a Long-Short Term Memory unit). Then, node connections are defined by applying the same reasoning considered in this paper for CNNs. Specifically, a connection between two nodes is created if the output of one node represents the input of the other one. In this way, connections are defined by the output passed to the units of the next layer, as well as by the hidden states passed to the units that are found to be adjacent to the ones of the current layer based on the input sequence. Going forward with this way of proceeding, we might think of mapping the Attention mechanism into a multilayer network. This would require the definition of a way to map the hidden states, the context vector, and the resulting output. These future works could lead our representation and compression approach to be applied in many other scenarios.

Finally, as further future work, we might consider overcoming the current limitation of aggregating convolutional filters through a mean or median operator. In fact, we currently chose such form of aggregation because it keeps the dimension of the multilayer network low. However, it causes us to lose information about the single filters. In the future, we would like to find ways to proceed that overcome this limitation.

Declarations

Data Availability All datasets used in our experiments are public datasets. They are available online. The source code is stored at the GitHub address https://github.com/lucav48/cnn2multilayer.

Ethics Approval This article does not contain any studies with human participants or animals performed by any of the authors.
Conflicts of interest The authors are declaring that there are no conflicts of interests.

References

[1] Dargan, S., Kumar, M., Ayyagari, M.R., Kumar, G.: A survey of deep learning and its applications: A new paradigm to machine learning. Archives of Computational Methods in Engineering **27**, 1071–1092 (2020)

[2] Merzoug, M.A., Mostefaoui, A., Kechout, M.H., Tamraoui, S.: Deep learning for resource-limited devices. In: Proc. of the ACM Symposium on QoS and Security for Wireless and Mobile Networks, New York, NY, USA, pp. 81–87 (2020). Association for Computing Machinery

[3] Choudhary, T., Mishra, V., Goswami, A., Sarangapani, J.: A comprehensive survey on model compression and acceleration. Artificial Intelligence Review, 1–43 (2020)

[4] Angermueller, C., Pärnamaa, T., Parts, L., Stegle, O.: Deep learning for computational biology. Molecular systems biology **12**(7), 878 (2016)

[5] Mahmud, M., Kaiser, M.S., McGinnity, T.M., Hussain, A.: Deep learning in mining biological data. Cognitive computation **13**(1), 1–33 (2021). Springer

[6] Chen, Y., Zheng, B., Zhang, Z., Wang, Q., Shen, C., Zhang, Q.: Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions. ACM **53**(4) (2020). Association for Computing Machinery

[7] Chen, Z., Chen, Z., Lin, J., Liu, S., Li, W.: Deep neural network acceleration based on low-rank approximated channel pruning. IEEE Transactions on Circuits and Systems I: Regular Papers **67**(4), 1232–1244 (2020)

[8] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. CoRR **abs/1704.04861** (2017) https://arxiv.org/abs/1704.04861

[9] Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. arXiv preprint arXiv:1602.07360 (2016)

[10] Zhang, X., Zhou, X., Lin, M., Sun, J.: Shufflenet: An extremely efficient convolutional neural network for mobile devices. In: Proc. of the IEEE

Conference on Computer Vision and Pattern Recognition (CVPR'18), Salt Lake City, Utah, USA, pp. 6848–6856 (2018). IEEE

[11] Mehta, S., Rastegari, M., Caspi, A., Shapiro, L., Hajishirzi, H.: Espnet: Efficient spatial pyramid of dilated convolutions for semantic segmentation. In: Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18), Salt Lake City, Utah, USA, pp. 6848–6856 (2018). IEEE

[12] Khan, A., Sohail, A., Zahoora, U., Qureshi, A.S.: A survey of the recent architectures of deep convolutional neural networks. Artif. Intell. Rev. **53**(8), 5455–5516 (2020). https://doi.org/10.1007/s10462-020-09825-6

[13] Kivela, M., Arenas, A., Barthelemy, M., Gleeson, J.P., Moreno, Y., Porter, M.A.: Multilayer networks. Journal of Complex Networks **2**(3), 203–271 (2014). https://doi.org/10.1093/comnet/cnu016

[14] Chen, S., Zhao, Q.: Shallowing deep networks: Layer-wise pruning based on feature representations. IEEE Transactions on Pattern Analysis and Machine Intelligence **41**(12), 3048–3056 (2019)

[15] Suzuki, K., Horiba, I., Sugie, N.: A simple neural network pruning algorithm with application to filter synthesis. Neural Process. Lett. **13**(1), 43–53 (2001). https://doi.org/10.1023/A:1009639214138

[16] Srinivas, S., Babu, R.V.: Data-free parameter pruning for deep neural networks. CoRR **abs/1507.06149** (2015)

[17] Ardakani, A., Condo, C., Gross, W.J.: Sparsely-connected neural networks: Towards efficient VLSI implementation of deep neural networks. CoRR **abs/1611.01427** (2016)

[18] Babaeizadeh, M., Smaragdis, P., Campbell, R.H.: A simple yet effective method to prune dense layers of neural networks. In: Proc. of the International Conference on Learning Representations (ICLR'17), Toulon, France (2017). ICLR

[19] Yang, Z., Moczulski, M., Denil, M., De Freitas, N., Song, L., Wang, Z.: Deep fried convnets. In: Proc. of the IEEE International Conference on Computer Vision (ICCV'15), pp. 1476–1483 (2015). https://doi.org/10.1109/ICCV.2015.173

[20] Lin, M., Chen, Q., Yan, S.: Network In Network (2014)

[21] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proc. of the IEEE Conference on Computer Vision and Pattern

Recognition (CVPR'15), pp. 1–9 (2015)

[22] Guo, Y., Yao, A., Chen, Y.: Dynamic network surgery for efficient dnns. In: Proc. of the International Conference on Neural Information Processing Systems (NIPS'16). NIPS'16, pp. 1387–1395. Curran Associates Inc., Red Hook, NY, USA (2016)

[23] Li, H., Kadav, A., Durdanovic, I., Samet, H., Graf, H.P.: Pruning filters for efficient convnets. CoRR **abs/1608.08710** (2016)

[24] Molchanov, P., Tyree, S., Karras, T., Aila, T., Kautz, J.: Pruning convolutional neural networks for resource efficient inference. In: Proc. of the International Conference on Learning Representations (ICLR'17), Toulon, France (2017). ICLR

[25] He, Y., Zhang, X., Sun, J.: Channel pruning for accelerating very deep neural networks. In: Proc. of the IEEE International Conference on Computer Vision (ICCV'17), pp. 1398–1406 (2017). https://doi.org/10.1109/ICCV.2017.155

[26] Liu, B., Wang, M., Foroosh, H., Tappen, M., Penksy, M.: Sparse convolutional neural networks. In: Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15), pp. 806–814 (2015). https://doi.org/10.1109/CVPR.2015.7298681

[27] Zhu, M., Gupta, S.: To prune, or not to prune: exploring the efficacy of pruning for model compression (2017)

[28] Albu, F., Mateescu, A., Dumitriu, N.: Architecture selection for a multilayer feedforward network. In: Proc. of International Conference on Microelectronics and Computer Science (ICMCS'97), pp. 131–134 (1997)

[29] Czernichow, T., Germond, A., Dorizzi, B., Caire, P.: Improving recurrent network load forecasting. In: Proc. of International Conference on Neural Networks (ICNN'95), vol. 2. Perth, WA, Australia, pp. 899–904 (1995). IEEE

[30] Chen, W., Wilson, J.T., Tyree, S., Weinberger, K.Q., Chen, Y.: Compressing neural networks with the hashing trick. In: Proc. of the International Conference on Machine Learning (ICML'15), pp. 2285–2294. JMLR.org, Lille, France (2015)

[31] Courbariaux, M., Bengio, Y., David, J.-P.: Binaryconnect: Training deep neural networks with binary weights during propagations. In: Proc. of the International Conference on Neural Information Processing Systems (NIPS'15). NIPS'15, pp. 3123–3131. MIT Press, Cambridge, MA, USA (2015)

[32] Lin, Z., Courbariaux, M., Memisevic, R., Bengio, Y.: Neural networks with few multiplications. In: Bengio, Y., LeCun, Y. (eds.) Proc. of the International Conference on Learning Representations (ICLR'16), San Juan, Puerto Rico (2016)

[33] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Proc. of the International Conference on Neural Information Processing Systems (NIPS'16), Red Hook, NY, USA, pp. 4114–4122 (2016). Curran Associates Inc.

[34] Hou, L., Yao, Q., Kwok, J.T.: Loss-aware binarization of deep networks. In: Proc. of the International Conference on Learning Representations (ICLR'17), Toulon, France (2017). ICLR

[35] Hou, L., Kwok, J.T.: Loss-aware weight quantization of deep networks. In: Proc. of the International Conference on Learning Representations (ICLR'18). ICLR, Vancouver, BC, Canada (2018)

[36] Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., Zou, Y.: Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. CoRR **abs/1606.06160** (2016) https://arxiv.org/abs/1606.06160

[37] Lin, J.-H., Xing, T., Zhao, R., Zhang, Z., Srivastava, M., Tu, Z., Gupta, R.K.: Binarized convolutional neural networks with separable filters for efficient hardware acceleration. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 344–352 (2017). https://doi.org/10.1109/CVPRW.2017.48

[38] Han, S., Mao, H., Dally, W.J.: Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding (2016)

[39] Anwar, S., Hwang, K., Sung, W.: Structured pruning of deep convolutional neural networks. J. Emerg. Technol. Comput. Syst. **13**(3) (2017). https://doi.org/10.1145/3005348

[40] Hinton, G., Vinyals, O., Dean, J.: Distilling the Knowledge in a Neural Network (2015)

[41] Romero, A., Ballas, N., Kahou, S.E., Chassang, A., Gatta, C., Bengio, Y.: Fitnets: Hints for thin deep nets. In: Proc. of the International Conference on Learning Representations (ICLR'15) (2015)

[42] Kim, J., Park, S., Kwak, N.: Paraphrasing complex network: Network compression via factor transfer. In: Advances in Neural Information Processing Systems, vol. 31 (2018). Curran Associates, Inc.

[43] Srinivas, S., Fleuret, F.: Knowledge transfer with Jacobian matching. In: Proc. of the International Conference on Machine Learning (ICLR'18), vol. 80, pp. 4723–4731 (2018). PMLR

[44] Polino, A., Pascanu, R., Alistarh, D.: Model compression via distillation and quantization. In: Proc. of the International Conference on Learning Representations (ICLR'18). ICLR, Vancouver, BC, Canada (2018)

[45] Lan, X., Zhu, X., Gong, S.: Knowledge distillation by on-the-fly native ensemble. In: Advances in Neural Information Processing Systems, vol. 31 (2018). Curran Associates, Inc.

[46] You, J., Leskovec, J., He, K., Xie, S.: Graph structure of neural networks. In: Proc. of the International Conference on Machine Learning (ICML'20), vol. 119, pp. 10881–10891 (2020). PMLR

[47] Altas, D., Cilingirturk, A.M., Gulpinar, V.: Analyzing the process of the artificial neural networks by the help of the social network analysis. New Knowledge Journal of Science **2**, 80–91 (2013)

[48] Sainath, T.N., Kingsbury, B., Sindhwani, V., Arisoy, E., Ramabhadran, B.: Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 6655–6659 (2013). https://doi.org/10.1109/ICASSP.2013.6638949

[49] Denton, E.L., Zaremba, W., Bruna, J., LeCun, Y., Fergus, R.: Exploiting linear structure within convolutional networks for efficient evaluation. In: Advances in Neural Information Processing Systems, vol. 27 (2014). Curran Associates, Inc.

[50] Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. In: Proc. of British Machine Vision Conference (BMVC'14) (2014). BMVA Press

[51] Kim, Y., Park, E., Yoo, S., Choi, T., Yang, L., Shin, D.: Compression of deep convolutional neural networks for fast and low power mobile applications. In: Bengio, Y., LeCun, Y. (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings (2016)

[52] Ioannou, Y., Robertson, D.P., Shotton, J., Cipolla, R., Criminisi, A.: Training cnns with low-rank filters for efficient image classification. In: Bengio, Y., LeCun, Y. (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings (2016)

[53] Alvarez, J.M., Salzmann, M.: Compression-aware training of deep networks. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS'17, pp. 856–867. Curran Associates Inc., Red Hook, NY, USA (2017)

[54] Zhang, X., Zou, J., He, K., Sun, J.: Accelerating very deep convolutional networks for classification and detection. IEEE Transactions on Pattern Analysis and Machine Intelligence **38**(10), 1943–1955 (2016). https://doi.org/10.1109/TPAMI.2015.2502579

[55] Li, C., Shi, C.-R.: Constrained optimization based low-rank approximation of deep neural networks. In: Proc. of the European Conference Computer (ECCV'18), vol. 11214, pp. 746–761. Springer, Munich, Germany (2018)

[56] Yao, K., Cao, F., Leung, Y., Liang, J.: Deep Neural Network Compression through Interpretability-Based Filter Pruning. Pattern Recognition, 108056 (2021). Elsevier

[57] Kahng, M., Andrews, P.Y., Kalro, A., Chau, D.H.: Activis: Visual exploration of industry-scale deep neural network models. IEEE Transactions on Visualization and Computer Graphics **24**(1), 88–97 (2018). https://doi.org/10.1109/TVCG.2017.2744718

[58] Hohman, F., Park, H., Robinson, C., Polo Chau, D.H.: Summit: Scaling deep learning interpretability by visualizing activation and attribution summarizations. IEEE Transactions on Visualization and Computer Graphics **26**(1), 1096–1106 (2020). https://doi.org/10.1109/TVCG.2019.2934659

[59] Zhang, Q., Cao, R., Shi, F., Wu, Y.N., Zhu, S.-C.: Interpreting cnn knowledge via an explanatory graph. Proceedings of the AAAI Conference on Artificial Intelligence **32**(1) (2018)

[60] Zhang, Q., Yang, Y., Ma, H., Wu, Y.N.: Interpreting cnns via decision trees. In: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6254–6263 (2019). https://doi.org/10.1109/CVPR.2019.00642

[61] Zhang, Q., Cao, R., Wu, Y.N., Zhu, S.-C.: Mining object parts from cnns via active question-answering. In: Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17), Los Alamitos, CA, USA, pp. 3890–3899 (2017). IEEE

[62] Manel Hmimida, R.K.: Community detection in multiplex networks: A seed-centric approach. Networks & Heterogeneous Media **10**(1), 71–85 (2015)

[63] Battiston, F., Nicosia, V., Latora, V.: Structural measures for multiplex networks. Phys. Rev. E **89**, 032804 (2014). https://doi.org/10.1103/PhysRevE.89.032804

[64] Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal **27**(3), 379–423 (1948). https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[65] Gowdra, N., Sinha, R., MacDonell, S., Yan, W.Q.: Mitigating severe over-parameterization in deep convolutional neural networks through forced feature abstraction and compression with an entropy-based heuristic. Pattern Recognition, 108057 (2021). Elsevier

[66] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: Bengio, Y., LeCun, Y. (eds.) Proc. of the International Conference on Learning Representations (ICLR'15) (2015)

[67] Alvear-Sandoval, R.F., Sancho-Gomez, J.L., Figueiras-Vidal, A.R.: On improving cnns performance: The case of mnist. Information Fusion **52**, 106–109 (2019). https://doi.org/10.1016/j.inffus.2018.12.005

[68] Angelov, P., Soares, E.: Towards explainable deep neural networks (xdnn). Neural Networks **130**, 185–194 (2020). https://doi.org/10.1016/j.neunet.2020.07.010

[69] Ferguson, M., Ak, R., Lee, Y.-T.T., Law, K.H.: Automatic localization of casting defects with convolutional neural networks. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 1726–1735 (2017). https://doi.org/10.1109/BigData.2017.8258115

[70] Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural networks. In: Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1. NIPS'15, pp. 1135–1143. MIT Press, Cambridge, MA, USA (2015)

[71] Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., Zhang, C.: Learning Efficient Convolutional Networks through Network Slimming. arXiv (2017). https://doi.org/10.48550/ARXIV.1708.06519