Learn & drop

Fast learning of cnns based on layer dropping

Cruciata, Giorgio; Cruciata, Luca; Lo Presti, Liliana; van Gemert, Jan; La Cascia, Marco

DOI
10.1007/s00521-024-09592-3

Publication date
2024

Document Version
Final published version

Published in
Neural Computing and Applications

Neural Computing and Applications (2024) 36:10839–10851
https://doi.org/10.1007/s00521-024-09592-3

ORIGINAL ARTICLE

Learn & drop: fast learning of cnns based on layer dropping

Giorgio Cruciata[1] · Luca Cruciata[1] · Liliana Lo Presti[1] · Jan van Gemert[2] · Marco La Cascia[1]

Received: 17 May 2023 / Accepted: 6 February 2024 / Published online: 28 March 2024

Abstract
This paper proposes a new method to improve the training efficiency of deep convolutional neural networks. During training, the method evaluates scores to measure how much each layer's parameters change and whether the layer will continue learning or not. Based on these scores, the network is scaled down such that the number of parameters to be learned is reduced, yielding a speed-up in training. Unlike state-of-the-art methods that try to compress the network to be used in the inference phase or to limit the number of operations performed in the back-propagation phase, the proposed method is novel in that it focuses on reducing the number of operations performed by the network in the forward propagation during training. The proposed training strategy has been validated on two widely used architecture families: VGG and ResNet. Experiments on MNIST, CIFAR-10 and Imagenette show that, with the proposed method, the training time of the models is more than halved without significantly impacting accuracy. The FLOPs reduction in the forward propagation during training ranges from 17.83% for VGG-11 to 83.74% for ResNet-152. As for the accuracy, the impact depends on the depth of the model and the decrease is between 0.26% and 2.38% for VGGs and between 0.4 and 3.2% for ResNets. These results demonstrate the effectiveness of the proposed technique in speeding up learning of CNNs. The technique will be especially useful in applications where fine-tuning or online training of convolutional models is required, for instance because data arrive sequentially.

Keywords Fast learning · CNN · Fine-tuning · Layer dropping

1 Introduction

The use of deep learning models is increasing over time, since they perform well in various areas, such as computer vision, natural language processing, and speech recognition. In computer vision, a popular deep learning model is the Convolutional Neural Network (CNN), which consists of deep models with many different layers processing the input by mapping it to the expected output [1].

The training time of CNNs is sometimes very long and could last hours, days or even weeks depending on the task, the size of the dataset and the available hardware. Nowadays, the trend is to increase the depth and size of architectures [2]; this usually leads to better performance despite a heavy computational cost. Although modern deep CNNs are composed of a variety of layer types, convolutional layers are the building blocks of CNNs and contribute greatly to the overall computational load of the network.

In recent times, the deep learning community has focused on how to improve the efficiency of CNNs while saving time, computational costs, and energy without compromising the accuracy of the results [2]. We can divide the methods that improve the efficiency of CNN into two categories [2], depending on the stage in which they seek to achieve better efficiency:

✉ Giorgio Cruciata
giorgio.cruciata@unipa.it

Luca Cruciata
luca.cruciata@unipa.it

Liliana Lo Presti
liliana.lopresti@unipa.it

Jan van Gemert
j.c.vangemert@tudelft.nl

Marco La Cascia
marco.lacascia@unipa.it

[1] Engineering Department, University of Palermo, V.le delle Scienze, Ed. 6, Palermo 90128, Italy

[2] Computer Vision Lab, Delft University of Technology, Van Mourik Broekmanweg 6, Delft 2628 XE, The Netherlands

- methods for inference efficiency: this category includes methods that compress the network during the training phase to use a more compact model during inference [3–7];
- methods for training efficiency: fall into this category methods that improve the efficiency of model training only, for example by freezing some layers during training but continuing to use the entire model during the inference phase [8–10].

At first glance, one might think that methods for inference efficiency are the most relevant because a model is trained once and used to infer multiple times. However, there are applications where new data arrives sequentially and the model has to be adapted and retrained using this new data. An example is visual tracking where the model is often used to represent the target appearance. Since appearance changes over time, it is required that the model adapts to such changes. In that case, slow retraining of the network may affect the real-time capabilities of the tracker. Also, recommendation systems or large language models (LLM) must be regularly retrained, and the training can last several weeks

The goal of this work is to improve the training efficiency of a CNN. During training, the model is gradually compressed by dropping convolutional layers. The proposed strategy stores the feature maps produced by the layer to drop and feed the remaining layers with them, reducing the training time. At inference time, the entire model (with all its layers) is used.

To demonstrate the main idea, we apply the method to some of the most popular CNNs such as VGG [11] and ResNet [12] by observing their learning behavior through gradient monitoring. However, the method is general and applicable to any modular model, namely models that can be partitioned into convolutional sub-blocks.

The two main steps in neural network training are forward and backward propagation; both steps have a high computational cost, which increases according to the complexity of the network. In general, forward propagation is the process of passing data through the network from one layer to another. In each layer, the input data are processed taking into account the weight matrix of the layer itself and a suitable activation function to produce the layer output. The output of each layer becomes the input of the next layer repeating this process until the output of the final layer is produced. The back-propagation algorithm [1] is used to train a neural network by updating its weights to minimize the error between the predicted and expected output. The algorithm computes the gradient of the loss function with respect to each weight via the chain rule, starting at the output layer and propagating the gradient backwards through the network from one layer to another.

The magnitude of the gradient permits to understand how the parameters of the model vary during training: the closer it is an optimal point, the lower the value of the gradient. The variation of the gradient magnitude across the epochs provides a learning curve that can be calculated separately for each layer of the network. We have empirically found similarities among these learning curves when analyzing different CNNs and, in particular, we have found that the layers' parameters are learned sequentially from the first to the last layer. Thus, in this paper, we propose to sequentially eliminate the convolutional layers during the training phase. Layers to drop are selected based on a metric related to the layer gradient. The adopted metric gives us an indication of which layer has stopped learning and can, therefore, be temporally eliminated from the model. However, when we drop a layer, we have to find a way to feed the next layer to continue its training. This is done by feeding the remaining model (the one without the deleted layers) with the feature maps produced from the last dropped layer. We stress here that dropping the layers has a double consequence: the weights of the removed layers are no longer modified and, moreover, a compressed model is trained in the next epochs. During the test, the entire model is used and each layer will have as weights those obtained when the layer was selected for dropping. This method shows an huge acceleration of the training process. We can summarize the main contributions of this work as it follows:

- A new method to improve the efficiency of training CNNs by dropping layers based on the metric derived from gradients.
- A significant speed-up of the training process of CNNs compared to state-of-art methods, given by the possibility of processing directly feature maps extracted from dropped layers and calculating the back-propagation only for the remaining layers. We empirically show that, by using our technique, the training time of a CNN is more than halved.

The paper is organized as it follows. In Sect. 2, we summarize the main differences between methods for inference efficiency and methods for training efficiency. We also detail how our method relates to former works. In Sect. 3, we detail the way we score convolutional layers in a given architecture and how we select layers to be dropped. We illustrate how our approach can be used during the training process by highlighting its main steps. Sect. 4 presents the results of an extensive validation of our approach over different architectures belonging to the family of VGG and ResNet models demonstrating the effectiveness of the methods on CNNs of varying depth. Three different and well known, publicly available datasets have been used to demonstrate that the proposed approach does not affect

much the accuracy values of the trained models and contributes to greatly reduce the training time. In particular, a discussion about the time reduction is presented in Sect. 5 while Sect. 6 presents conclusions and future work.

2 Related work

In recent years, many works have focused on how to reduce the parameter size of deep learning models. This is quite a challenge considering the energy impact of training large networks. The goal of compression techniques is to achieve a more efficient representation in a neural network, improving the generalization of the model if the model is overly parameterized. Model optimization is performed with respect to model size or training time in exchange for as little accuracy loss as possible. A popular method of reducing the complexity of neural networks is to permanently remove neurons, filters, or layers for training. This technique is called pruning. Pruning can be performed based on different aspects of neural networks, for example it may be possible to remove parameters with low saliency scores from a pre-trained network. These techniques are often referred to as Optimal Brain Damage or Optimal Brain Surgeon and were first introduced by LeCun et al. [13] and Hassibi et al. [14], respectively. The goal of this process is to minimize the impact of compressing the network on its performance, as measured by its validation loss. OBD approximates the saliency score by using a second-derivative of the parameters $(\frac{\partial^2 L}{\partial w_i^2})$, where L is the loss function, and w_i is the candidate parameter for removal. In [15], it is suggested that a greedy method be used to determine the minimum set of neurons needed to minimize the reconstruction loss, but this approach has a high computational cost. Other methods that prune by "saliency" consider the magnitude of the weights [16, 17]. Sparse Momentum [18] determines where to grow new weights in a sparse network by looking at the weighted average of recent gradients (momentum) to find weights and layers which reduce the error consistently. The method: (1) determines the importance of each layer according to the mean momentum magnitude; (2) for each layer, removes the 50% of the smallest weights; (3) redistributes the weights across layers according to layer importance. In particular, it grows weights of layers whose momentum magnitude is large. The previously described methods for removing sparse neurons are examples of unstructured pruning methods. On the other hand, structural pruning aims at reducing the number of filters as in [19] where the L_1-norm is used to select the filters to be removed without affecting the accuracy of the classification. A similar idea is presented in [20] where the feature map channels that do not contribute to the result are removed.

In addition to pruning filters and channels, there are also methods of pruning entire layers [21–23]. Using different criteria, the selected layers from the network are removed to obtain a compact model. These methods claim the model obtained from layer pruning require less inference time and memory usage at runtime with similar accuracy values than the model obtained from filter pruning methods. The work in [21] uses independently trained linear classifiers per layer to rank their importance. After ranking, they remove less important layers and fine-tune the remaining model. However, their method requires additional rank training. All the previously described methods fall into the category of inference efficiency methods because they produce compressed model to be used at inference time.

In other efficiency training approaches like in [24], the gradient computation through the chain rule is stopped.

To speed-up training and increase accuracy in very deep networks, AFNet [24] investigates a different use of back-propagation. Only a subset of layers is trained while the others are frozen. Frozen layers weights do not need to be updated during back-propagation. In [8], a metric F, named Freezing Rate, is defined as a function of the gradient values of the set of weights of a layer. This metric is used in [8] to decide which layer should be frozen during the training. The frozen layers must be subsequent to not break the layers chain, and therefore the freezing of the layers start from the first layer and advances to the subsequent ones. During training, the layers are not removed and therefore the computational advantage of the method concerns only limiting the calculation of the gradient and the modification of the weights of the frozen layers. Our approach is inspired by [8], in the sense that we adopt the same metric F. However, unlike the method in [8], our method consists in performing a layer elimination (drop) during the training phase. Once a layer is removed, the remaining model is fed through the feature maps produced by the last deleted layer. This speeds up both forward propagation, since there is no need to recalculate the feature maps of the deleted layers, and backward propagation, since the gradient is not calculated in the dropped layers and there is no need to update their weights. Our experiments demonstrate how this approach increases training efficiency by gradually reducing the number of FLOPs over the epochs. At the same time our method is different from classic pruning methods, as our method does not permanently remove the layers by performing a network compression but only temporarily drops them during the training phase. In fact, in the test phase, the model will contain all the layers of the one originally created.

3 Dropping layers for training efficiency

Our method is described in Algorithm 1 and its main steps are:

- Compute the "Layer Importance metric" based on the gradient of the layer's weights;
- Apply the "Fast Learning" algorithm, the core of our method. The algorithm consists of steps to: (1) select the layers to be dropped, (2) split the network into a "tail," composed of the dropped layers, and a "head," composed of the layers to still train, (3) compute and store output feature maps from the tail, (4) train the head by using output feature maps from the tail.

3.1 Layer importance

Our layer dropping method is based on the observation of the loss function gradients ∇g. Generally, the gradient values can be interpreted as the rate of change of the weights. The sign of the partial derivatives represents instead the inverse direction in which weights should be changed to reach a minimum.

Given a neural network with layers $L = \{l_0, l_1, ..., l_L\}$, the average absolute partial derivative value $\overline{g_l^{(k)}}$ corresponding to the weights of the l-th layer is calculated as:

$$\overline{g_l^{(k)}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} \mid g_{lij}^{(k)} \mid \tag{1}$$

where N is the number of weights in layer l, M is the number of iterations in epoch k, and $g_{lij}^{(k)}$ is the partial derivative of the loss function with respect to the i-th weight in layer l at the j-th iteration in epoch k. Figure 1 shows two graphs representing the average absolute partial derivative value $\overline{g_l^{(k)}}$ of each convolutional layer in the VGG-11 and ResNet-18 models. The layers closest to the input in both networks have a greater average partial derivative value than the layers closest to the output. In practice, the figure suggests that weights in the first layers undergo much higher changes than the weights in the layers closest to the output, especially at the beginning of the training.

Thus, partial derivative values can help understanding if the weights of a layer are still changing or not. However, using directly the average absolute partial derivative values could be misleading since the weights may have small magnitude but still change.

Hence, we decided to adopt the metric proposed in [8] and define for the l-th layer the score:

$$P_l^{(k)} = 1 - \frac{\sum_{i=1}^{N} \mid \sum_{j=1}^{M} g_{lij}^{(k)} \mid}{\sum_{i=1}^{N} \sum_{j=1}^{M} \mid g_{lij}^{(k)} \mid} \tag{2}$$

with $0 \leq P_l^{(k)} \leq 1$, where $P_l^{(k)}$ measures the degree of changes of the weights in layer l at the k-th epoch. $P_l^{(k)}$ will be 1 if the partial derivatives cancel each other across the M iterations. In such a case, within the epoch, the layer weights do not change much and, intuitively, the layer has stopped to learn. $P_l^{(k)}$ will tend to 0 if most of the partial derivatives are in the same direction across iterations. In this case, layer weights are changing during the epoch. Thus, the layer is learning something about the problem to solve. We note here that the normalization factors make the scores comparable across the layers despite the different magnitude of the weight's partial derivatives.

Under this point of view, the score $P_l^{(k)}$ is measuring the importance of the layer during training. Layers with a score close to 0 must be trained. Layer with a score approaching 1 are not learning much and probably can be dropped to speed-up the model training. Unlike [8], where this score is used to freeze the layer and stop the back-propagation

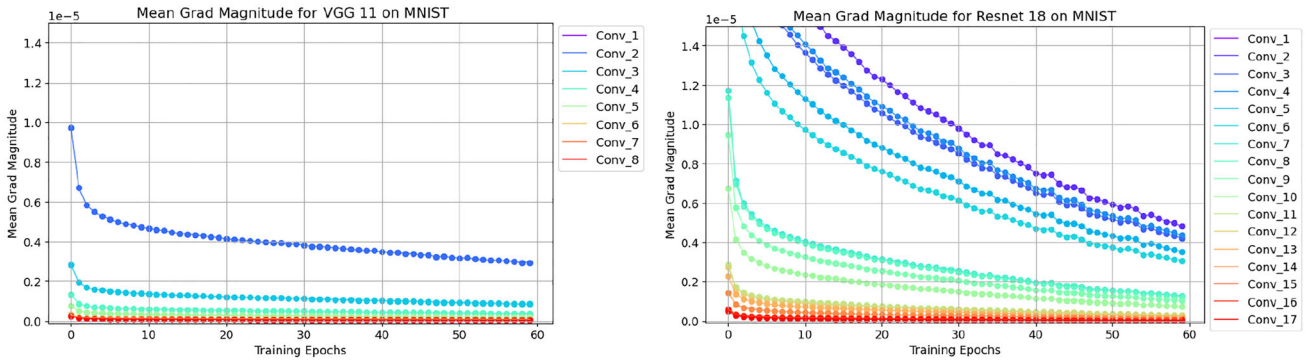

Fig. 1 The graphs represent the average absolute partial derivative (AAPD) value of each convolutional layer in the VGG-11 (left) and ResNet-18 (right) on the MNIST dataset (AAPD on the y-axis, epochs on the x-axis). Each curve represents a different layer (purple for layers close to the input, red for those close to the output). Weights in the first layers undergo higher changes than those in the layers closest to the output. AAPD help measuring if a layer is still learning or not

computation up to the l-th convolutional layer, our algorithm uses P_l to drop the l-th convolutional layer. The feature maps produced by the last dropped layer are used as input to the remaining model.

3.2 Improving training efficiency

In our approach, the removal of layers from the model to improve the training efficiency must take place in sequential order. At the k-th epoch, the layers to be dropped are selected based on the importance score $P_l^{(k)}$.

Thus, our fast learning algorithm works as it follows:

1. At the end of epoch k, the metric $P_l^{(k)}$ is calculated for each layer l. The score values are then standardized:

$$P'_l = \frac{P_l - \overline{P}}{\sigma_p} \tag{3}$$

where $\overline{P}$ and σ_p represent the average score over the layers and the standard deviation, respectively. We omitted the apex k for simplicity.

Ideally, we want to drop all subsequent layers for which the parameters do not change much anymore starting from the first layer of the network. The standardized scores P'_l can have positive and negative values. Positive values indicate that the layer's weights are changing less than the average (hence, the layer is likely not learning much), while negative values indicate that the layer's weights are changing more than the average (hence, the layer is still learning something).

The problem of selecting the subsequent layers to drop starting from the first layer turns into the problem of finding the sub-vector of maximum sum starting from the first element of an array. In our case, the array represents the list of scores P'_l with $l \in L$.

Let us assume that the current layers in the model are $L = \{l_z, l_{z+1}, \ldots, l_L\}$. Candidate layers to drop are $l_z ... l_{n^*}$ with n^* computed as:

$$n^* = \min_t \left\{ t \in [z, \ldots, L-1] : \; P'_{l_t} > 0 \wedge P'_{l_{t+1}} < 0 \right\}. \tag{4}$$

2. As soon the candidate layers to drop $l_z ... l_{n^*}$ are found, to avoid dropping them too early, we estimate the median M_c of the scores P'_{l_t} with $l_t \in l_z ... l_{n^*}$ (namely the scores of the candidate layers to drop) and compare it with the median M_d of the scores P'_{l_t} with $l_t \in l_0 ... l_{z-1}$ (namely the scores of the layers dropped in previous iterations and estimated when the decision of dropping the layers was taken). We perform layer dropping if $M_c \geq M_d$. In this way, we limit the effects that early layer dropping may have on the network accuracy value.

   Once the layers to drop are identified, the network is split into two parts: the "tail," composed of the layers in the network up to l_{n^*}, and the "head," composed of the layers from l_{n^*+1} to the network output.
3. In epoch $k+1$, the tail is used to extract feature maps. These feature maps are stored on a memory, such as a disk, and are also used to feed the head to continue its training.
4. In epoch $k+2$, the stored feature maps are retrieved from the memory and used to train the head.

These 4 steps are within an iterative procedure repeated until the maximum number of epochs is reached or the convolutional layers are exhausted, namely the head does not have any layer.

Our approach differs from the one in [8]. In the latter approach, layers with a high score are not "physically" removed from the network but their weights are not trained. The main limitation of the approach in [8] is that, during forward propagation, the data must be processed at each iteration even by layers for which the weights are not updated. Our approach overcomes this limitation by removing layers in order starting with the first. We experimentally demonstrate the advantages of this approach in significantly reducing the computational cost of the training process.

Furthermore, in [8], layers to exclude from the training are selected after a prefixed number of epochs based on the vector $f = [f_1, f_2, ..., f_n]$. Each value f_i indicates the number of epochs between one freezing and another at a specific learning rate.

Hyper-parameters in f are empirically defined and change over the adopted datasets. In our approach, the decision to drop a layer is fully automatic. After each epoch, the method analyzes the scores $P_l^{(k)}$ to detect candidate layer to be dropped and, as already described, the decision to remove the layers or not depends also on the median of the estimated scores.

Algorithm 1 Fast Learning by layer dropping

Require:
$model$, a model of L layers with randomly initialized parameters;
$e_1 > 0$, number of warm-up epochs;
$e_2 > 0$, number of training epochs;
L_0, number of dense layers + 1;
Ensure:
Trained $model$;
Initialize models $head$ and $tail$
Initialize $Data$ with the training images
$save\_features$ = False.
Train $model$ for e_1 epochs on $Data$
$L = model$.layers.length()
$head$.layers = $model$.layers$[0 : L-1]$
$tail$.layers = []
$features\_maps$ = []
Initialize P'_l with l in $head$
for $k = 0$ **to** e_2 **do**
  if $save\_features$ **then**
    $save\_features$ = False
    $features\_maps = tail(Data)$
    Train $head$ on $features\_maps$
    Update $model$ weights based on $head$
    store $features\_maps$ to memory
  else
    if $features\_maps$! = [] **then**
      Initialize $Data$ with $features\_map$
      $features\_maps$ = []
    end if
    Train $head$ for 1 epoch on $Data$
    Update $model$ weights based on $head$
    if $L > L_0$ **then**
      for $\forall$ conv. layer l in $head$ **do**
        update P'_l (Eqs. 2& 3)
      end for
      Find n^* (Eq. 1) for layer dropping
      Estimate median values M_c and M_d
      if $n^* > 1 \wedge M_c \geq M_d$ **then**
        $save\_features$ = True
        $tail$.layers = $head$.layers$[0 : n^*]$
        $head$.layers.pop$(0 : n^*)$
        $L = head$.layers.length()
      end if
    end if
  end if
  validate $model$ on validation set
end for

3.3 Fast-training algorithm

Our approach is described in Algorithm 1 and represented as diagram block in Fig. 2 It starts with a few warm-up epochs e_1 where the *model* is trained to move from the initial random weights. After this warm-up, the weights of *model* are copied to the *head* model, which is initially equal to *model*, while the *tail* model is empty. From now on, only the *head* model is trained. The *tail* model stores the dropped layers and is used to estimate the features maps needed to feed the *head* model. Each time the *head* model is trained, the corresponding weights in the *model* are updated accordingly. In practice, *model* always contain all layers, whose weights are iteratively updated based on the weights learned by the *head* model. The *save_features* flag is used to indicate whether the *tail* model should be used to estimate feature maps using the dropped layers. *Data* stores the data for training the model. Initially, *Data* stores the training images. When layers are dropped, *Data* stores the features maps produced by the dropped layers, i.e., the *tail* model. At each iteration, the layer importance P'_l is recomputed only for each layer of the *head* model as described in Eqs. 2 and 3.

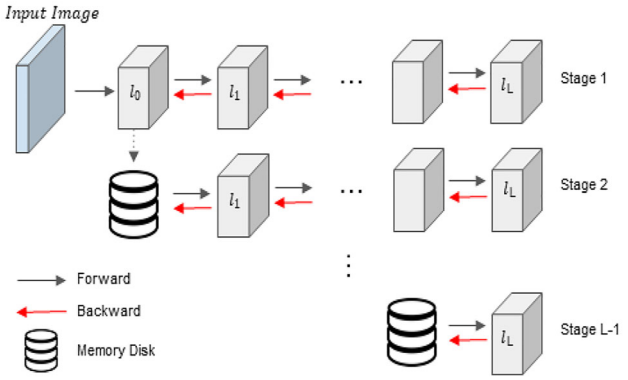

Fig. 2 The image shows how the process flows through the sequence of stages. At the first stage the input is the original image, subsequently the features maps stored from dropped layers are used as input for the remaining layers

Then, n^* is computed based in Eq. 4. Layer dropping is performed if it is found a maximum sum sub-sequence of scores P'_l starting from the first layer of the *head* that includes at least one layer and the median value of the scores in the found sub-sequence is greater than the median score of the previously dropped layers. The scores of the dropped layer are not recomputed every time but stored during the training process and kept updated till the layer is not dropped. The index n^* is also used to further compress the *head* model. In particular, the *tail* model stores the dropped layers, namely the first n^* layers of the *head* model. These same layers are pulled out from the *head* model, resulting in a reduced model.

The described process is iterated until only the last convolutional and dense layers remain; they continue to train till the maximal number of epochs e_2 is not reached.

We emphasize that the whole model (*model*) is tested on a validation set; this proves that removing the layers does

not affect the accuracy of the original model. The model is optimized using SGD method in order to maintain a fixed learning rate over the different iterations. This is not a limitation, and other optimizers might be used as well. Also, early stopping may be included in the algorithm to add more regularization. In our experiments, we did not use early stopping to compare different training strategies on equal terms of number of epochs.

4 Experimental results

In this section, we present results of the validation of our method. We first present the selected neural architectures used to assess our method. We also provide details about how to apply our method to improve the training efficiency of these architectures. Then, we detail the datasets selected to perform the experiments and the hyper-parameters adopted on each dataset.

The results of our experiments are reported in Tables 1 and 2. In each table, "Network" indicates the neural architecture; "Dataset" specifies the dataset used for training and testing the model; "SGD," "Freezing" and "Dropping" indicate the strategy used to train the network. In particular, SGD is the baseline method, namely the model is trained in a standard way without attempting any training efficiency. Freezing represents our implementation of the method in [8] where, however, layers to freeze are selected in the same way as we do in our method. In this case, weights of selected layers are excluded from the training (frozen) but the layers are not physically removed from the trained model. The column Dropping reports results of our layer dropping method. The columns indicated with "T" report the duration of the training and refers to the time in minutes required to complete the expected number of training epochs, including the warm-up epochs. Columns "A" report the test accuracy values, measuring the percentage of correct predictions made by the model on the test set. Finally, columns "ΔT" report the percentage of time saved by applying a training efficiency strategy (Freezing or Dropping) with respect to the baseline (SGD). In particular, we compute this metric as:

$$\Delta T = \frac{T_{\text{SGD}} - T}{T_{\text{SGD}}} \times 100. \tag{5}$$

All experiments have been carried on a machine equipped with: GPU RTX 3090 24GB, RAM 96 GB, processor Intel(R) Xeon(R) CPU E5-2403 1.80GHz.

In our implementation, feature maps produced by the dropped layers are stored on disk directly as PyTorch tensor using the "Pickle" Python package [25], that implements binary protocols for serializing and de-serializing a Python object. We experimentally noted that using Pickle is faster than writing and reading files on disk with the Numpy package [26] and PyTorch [27].

4.1 Neural architectures

In this paper we focused on improving training efficiency of CNNs. To assess our method, we considered two neural network architectures widely adopted in the computer vision field. VGG (Visual Geometry Group) is a convolutional neural network introduced in [11]. The VGG architecture is characterized by its depth and the use of small convolutional filters. It consists of a sequence of convolutional layers, followed by a sequence of fully-connected layers. There are several configurations of the VGG architecture. The smaller version is the VGG-11 with only

Table 1 Fast Training of VGG architectures

		SGD		Freezing			Dropping (Ours)		
Network	Dataset	T (min)	A (%)	T (min)	A (%)	ΔT (%)	T (min)	A (%)	ΔT (%)
VGG-11	MNIST	20.83	**98.64**	19.58	98.25	6.00	**8.74**	98.25	**58.04**
VGG-11	CIFAR-10	23.83	**92.02**	23.54	91.72	1.21	**8.21**	91.72	**65.54**
VGG-11	Imagenette	61.01	**75.33**	59.33	74.08	2.75	**18.32**	74.08	**69.97**
VGG-16	MNIST	22.54	**98.85**	21.45	98.26	4,24	**9.01**	98.26	**60.03**
VGG-16	CIFAR-10	26.54	**93.12**	24.94	92.84	6.03	**9.56**	92.84	**63.98**
VGG-16	Imagenette	74.73	**78.76**	71.21	77.83	4.71	**25.23**	77.83	**66.24**
VGG-19	MNIST	23.02	**98.52**	22.68	96.22	1.48	**9.45**	96.22	**58.95**
VGG-19	CIFAR-10	27.02	**93.10**	25.78	91.71	4.59	**11.53**	91.71	**57.33**
VGG-19	Imagenette	110.35	**80.32**	105.34	78.13	4.54	**37.76**	78.13	**65.78**

SGD refers to the standard training strategy of the entire model. Freezing refers to excluding the parameters of some layers from the training without removing the layers from the model. Dropping is our method where layers are deleted from the trained model. T is the training time in minutes. A is the test accuracy value. ΔT is the percentage of reduced training time with respect to the time of SGD

Bold values indicate the best metric value

Table 2 Fast Training of ResNet architectures

		SGD		Freezing			Dropping (Ours)		
Network	Dataset	T (min)	A (%)	T (min)	A (%)	ΔT (%)	T (min)	A (%)	ΔT (%)
ResNet-18	MNIST	23.67	**98.2**	23.10	97.78	2.41	**8.64**	97.78	**63.50**
ResNet-18	CIFAR-10	27.67	**92.25**	25.97	91.82	6.14	**11.90**	91.82	**56.99**
ResNet-18	Imagenette	253.07	**80.12**	242.32	79.07	4.25	**83.78**	79.07	**66.89**
ResNet-50	MNIST	35.43	**98.75**	35.02	96.85	1.16	**11.23**	96.85	**68.30**
ResNet-50	CIFAR-10	38.43	**94.40**	35.40	92.05	7.88	**13.05**	92.05	**66.04**
ResNet-50	Imagenette	336.00	**82.78**	315.34	80.34	6.15	**86.28**	80.34	**74.32**
ResNet-101	MNIST	53.12	**97.81**	51.64	95.45	2.79	**18.56**	95.45	**65.06**
ResNet-101	CIFAR-10	56.12	**93.98**	52.03	91.26	7.29	**19.53**	91.26	**65.20**
ResNet-101	Imagenette	402.34	**82.23**	380.23	80.75	5.29	**120.44**	80.75	**70.06**
ResNet-152	MNIST	70.76	**97.43**	65.30	95.12	7.72	**23.34**	95.12	**67.01**
ResNet-152	CIFAR-10	74.76	**93.45**	68.93	91.03	7.80	**25.75**	91.03	**65.56**
ResNet-152	Imagenette	540.76	**82.65**	504.72	79.45	6.66	**180.34**	79.45	**66.65**

SGD refers to the standard training strategy of the entire model. Freezing refers to excluding the parameters of some layers from the training without removing the layers from the model. Dropping is our method where layers are deleted from the trained model. T is the training time in minutes. A is the test accuracy value. ΔT is the percentage of reduced training time with respect to the time of SGD

Bold values indicate the best metric value

11 layers. The VGG-16 has 16 layers and the VGG-19 has 19 layers. As the number of layers in VGG increases, so do the training time and memory requirements.

From the experiments carried out with VGG, we found out that the order of the curves representing the layer scores P'_l across the epochs depends on the inclusion in the model of the batch normalization layer. In fact, as it can be seen in Fig. 3, we note that the scores order of the layers of VGG+BN is inverse respect to the scores order of the VGG without BN. This is because batch normalization speeds up learning in neural networks by normalizing the inputs to each layer, which reduces the internal covariate shift. This makes the optimization more stable and allows the network to learn more quickly and with higher accuracy. Hence, based on our experiments, to use our technique with a VGG it is recommended to include the batch normalization layer (one after each convolutional layer).

ResNet (Residual Network) is a convolutional neural network introduced in [12]. ResNet is characterized by the use of residual blocks, which help to alleviate the vanishing gradient problem and allow for the creation of much deeper neural networks. The original ResNet architecture has several configurations, including ResNet-18, a relatively small version of the ResNet architecture with only 18

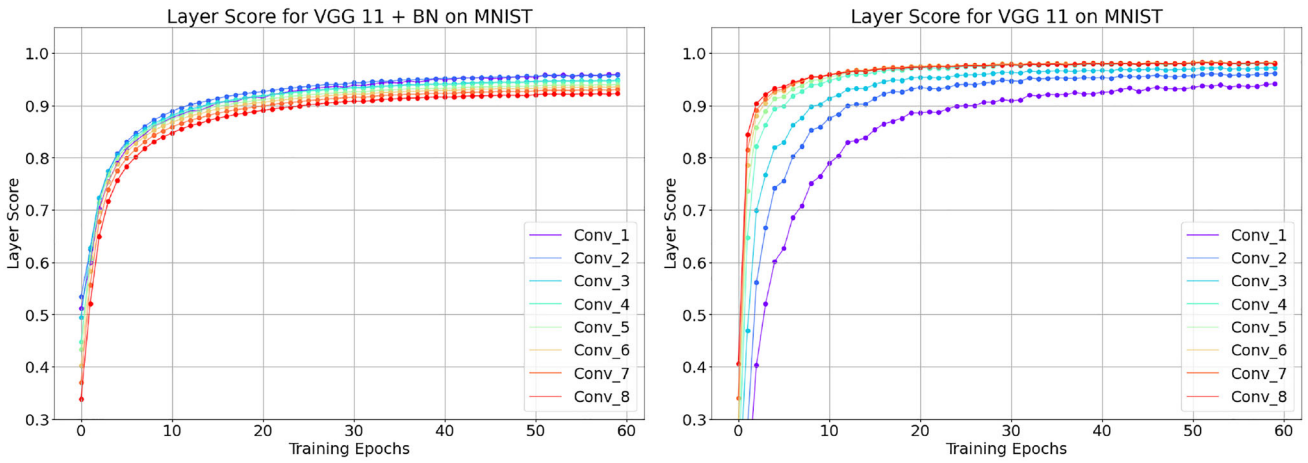

Fig. 3 The plots show the scores $P_l^{(k)}$ on the MNIST dataset for a VGG-11 trained with (on the left) and without batch normalization (on the right). Bath normalization reverses the order of the score curves and reduces the internal covariate shift making the optimization more stable and quick. As an effect, layers are learned sequentially from input to output

layers. ResNet-50, ResNet-101, and ResNet-152 are much deeper versions of the ResNet architecture with 50, 101, and 152 layers, respectively.

We point out that in the case of ResNet, we have to save not only the features maps but also the output of the skip connections to maintain the original behavior as shown in Fig. 4.

4.2 Dataset and hyper-parameters

To evaluate our algorithm, we use three popular classification datasets: MNIST [28], CIFAR-10 [29], and Imagenette [30].

The MNIST dataset contains 60,000 gray scale images, with 50,000 for training and 10,000 for test. It includes 10 classes, and each class is represented by 6,000 images.

The CIFAR-10 dataset contains 60,000 color images, with 50,000 for training and 10,000 for test. Overall, there are 10 classes, and, for each class, 6,000 images.

Imagenette [30] includes a subset of 10 classes from the larger Imagenet [31].

The classes are *tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute*. Overall, the adopted dataset includes about 1,000 color images per class with a resolution of 160 x 160 pixels. The images are obtained from the ones in the original Imagenet dataset by performing a resizing that preserves the original aspect ratio.

We opted to use MNIST and CIFAR-10 datasets to ensure the comparability of our work with [8]. In addition, we included the more complex Imagenette dataset to evaluate the performance of our work on a more challenging dataset.

In all our experiments, we use a batch size of 256 samples. We set the learning rates to values generally used in literature, while the number of epochs and warm-up are selected empirically. On the MNIST and CIFAR-10 datasets, the total number of epochs (including the model warm-up) is set to 60. On the MNIST dataset, the learning rate is fixed to 0.001. The warm-up epochs are 5. On the CIFAR-10 dataset, the learning rate is fixed to 0.1 and scaled x10 after 20 epochs. The warm-up epochs are 10. Finally, on the Imagenette, the number of epochs is set to 150 and the learning rate is fixed to 0.01 and scaled x10 after 50 epochs. The warm-up epochs are 25.

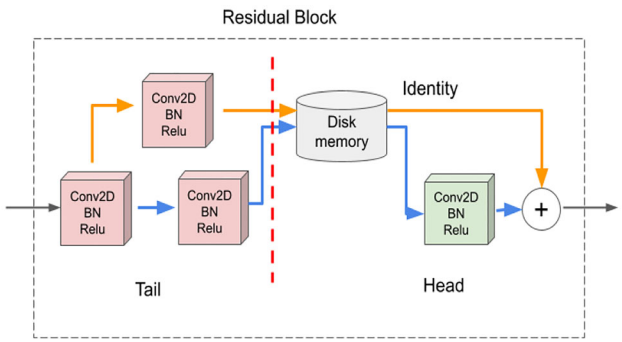

Fig. 4 The image shows how dropping takes place in the ResNet at any residual block. The feature maps saved to the memory come from the layers in red inside the residual block and on the skip connection. The layers on the left of the vertical dotted line are dropped and belong to the tail, the ones on the right belong to the head model and are trained based on the stored feature maps

4.3 Results and comparison

Table 1 reports the results obtained by applying our method to models of various depth (11, 16, and 19) in the family of VGG architectures. All models include batch normalization. First, we note that, across the models and datasets, the impact on accuracy values of freezing or dropping layers to increase training efficiency is negligible. The differences in the accuracy values vary, for both techniques, in the range 0.26 (for VGG-16 trained on CIFAR-10) and 2.38 (for VGG-19 trained on MNIST). These differences increase slightly with network depth and are generally higher for the Imagenette dataset. However, these small decreases in accuracy values come with a reduction in training time. As shown in the table, for VGG models, while the training time reduction with the freezing layer technique varies in the range 0.40% (for VGG-16 on the MNIST dataset) and 6.03% (for VGG-16 on the CIFAR-10 dataset), with our layer dropping technique the training time reduction varies in the range of 58.04% (for VGG-11 on the MNIST dataset) and 69.97% (for VGG-11 on the Imagenette dataset). While on average these percentages are 3.52% for the freezing layer method, they are 62.87% with our technique.

These results are not limited to VGG architectures. Indeed, Table 2 reports similar achievements for ResNet architectures of various depth (18, 50, 101, and 152). In particular, analyzing the results in a similar way to what was done for the VGG architectures, the differences in the accuracy values vary, for the layer freezing and dropping techniques, in the range 0.4% (for ResNet-18 on the MNIST dataset) and 3.2% (for ResNet-152 on the Imagenette dataset). For both techniques, the impact on accuracy values generally increases with network depth. In terms of training time reduction, with the layer freezing technique, the percentages vary in the range 1.16% (for ResNet-50 on the MNIST dataset) and 7.8% (for Resnet-152 on the CIFAR-10 dataset). Our approach achieves training time reduction percentages in the range 56.99% (for ResNet-18 on the CIFAR-10 dataset) and 74.32% (for ResNet-50 on the Imagenette dataset). While, on average, layer freezing accounts for a 5.46% of training time

reduction; our method results in average training time reduction in approximately 66.30%.

Overall, the training time for the VGG and ResNet architectures is more than halved with our technique despite the loss in accuracy values being comparable to that obtained by freezing the layers.

5 Training time and parameter reduction

The results discussed in Sect. 4 refer to the impact of our technique on model accuracy and the training time reduction achieved on our machine. To further analyze and explain the performance of our technique, this section refers to the effect it has on the number of parameters and operations performed during forward propagation at training time. Indeed, our technique not only reduces the number of weights for which it is necessary to estimate partial derivatives during gradient back-propagation, but also affects the number of operations performed during forward propagation.

A potential bottleneck in our method is the feature map saving to disk whenever layer dropping occurs. In some networks, the size of the features maps produced by a layer may be greater than the size of the input images; thus, reading and writing feature maps with large dimensions can cause a slowdown of the training. However, we have observed empirically that layer dropping never takes place layer by layer but generally several subsequent layers are dropped together. Figure 5 shows (on the left) the test

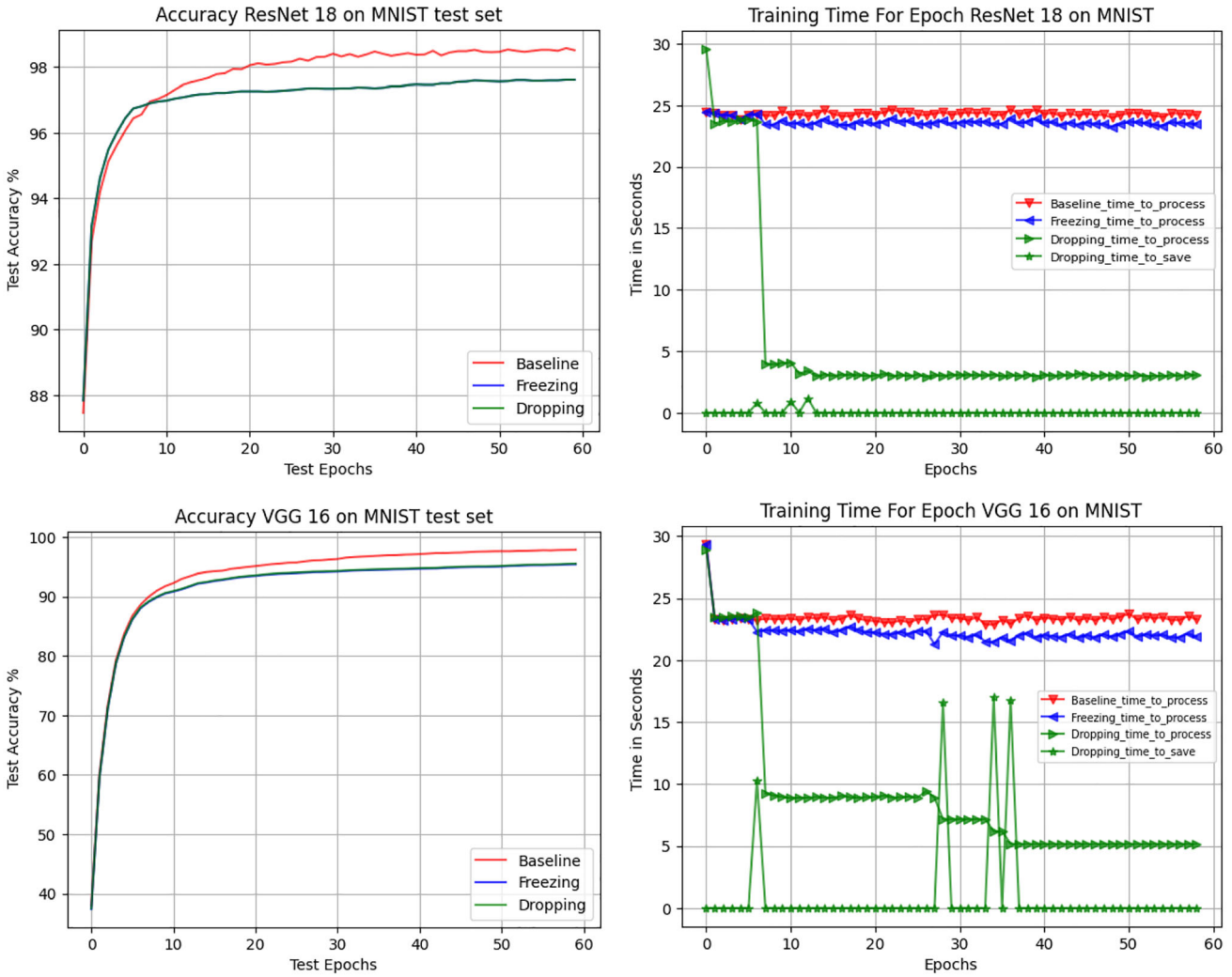

Fig. 5 The plots on the left show the test accuracy values of a ResNet-18 (top) and VGG-16 (bottom) trained on the MNIST dataset with different strategies: SGD (red curves), layer freezing (blue curves), and layer dropping (green curves). The experiments were repeated 10 times with different starting weights and data randomization. Freezing and dropping layers achieve nearly equivalent test accuracy values, and the values are slightly lower than those achieved by training the entire model. On the right, the plots show training time per epoch. Starred curves show the time required to store the feature maps to disk, while the other curves show the training time which decreases over the epochs due to the lower cost of forward propagation in our method

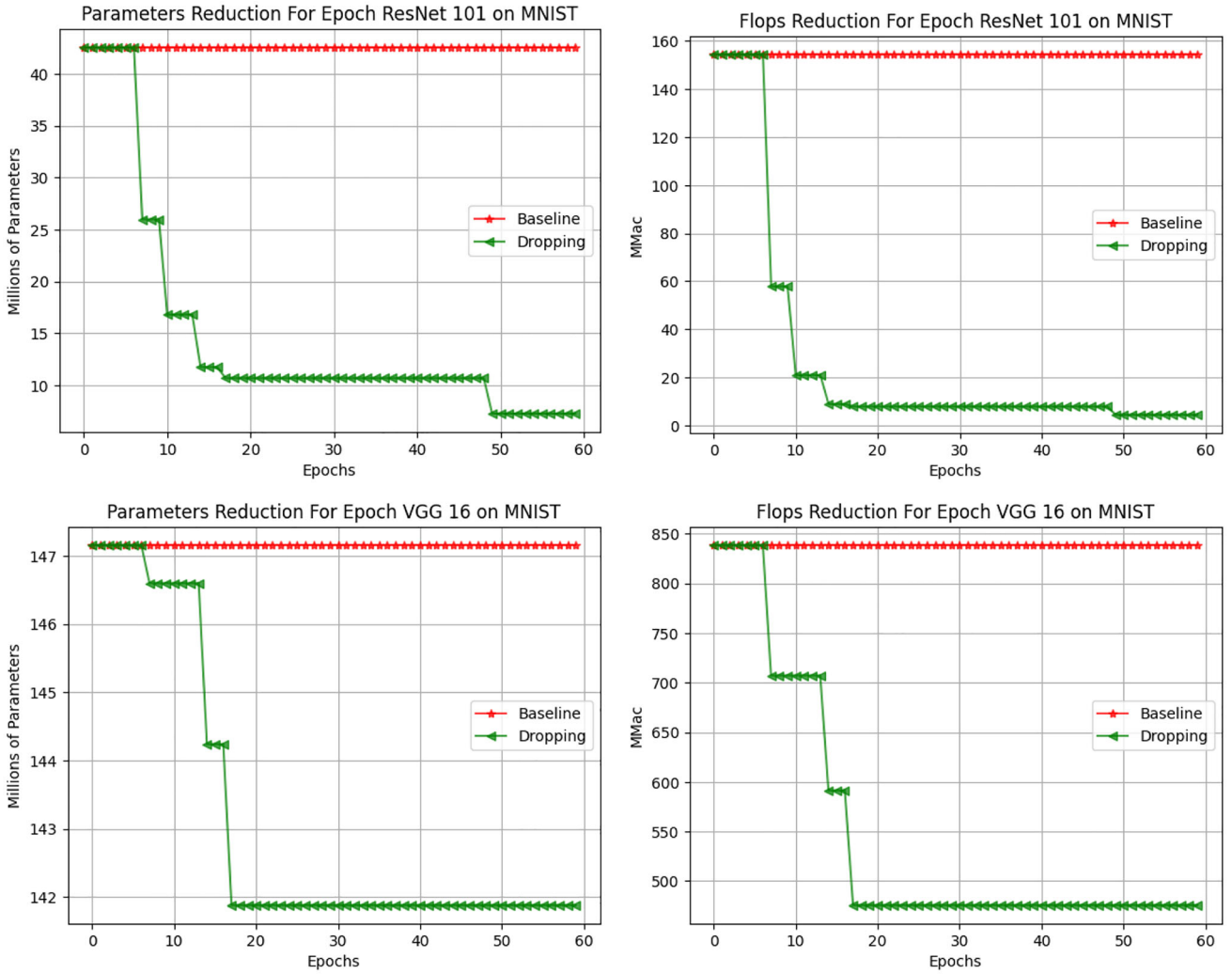

Fig. 6 Left plots show the number of network parameters in each epoch for the ResNet-101 (top) and VGG-16 (bottom). The number of parameters remains constant when training or freezing the layers (red curves); it decreases with our approach (green curves). This parameter reduction is correlated with the MMAC reduction, shown in the plots on the right, because our method reduces the number of operations during forward propagation

accuracy values of ResNet-18 (top row) and VGG-16 (bottom row) on the MNIST dataset over the epochs for SGD, the layer freezing method and our technique. As already discussed, our technique and the freezing layers one has a limited impact on the final model accuracy. On the right, the figure shows the training time per epoch of each technique. In particular, the time spent by our method refers to: the time to store feature maps on the disk whenever layer dropping arises, and the time to train the model. As shown in the plot, the time to store feature maps on the drive is concentrated only in few epochs since, as already said, our method does not constantly drop layers. The time to store the maps to disk depends on their size and, thus, is higher for the VGG model. The time to train the model includes, for all methods, the time for computing the gradients, updating the layer weights, loading the training data and performing forward propagation. The plot shows how, in our technique, this time decreases over time and becomes much lower than that of the other methods.

To further investigate this result, we discuss the FLOPs (floating point operations per second) of our method versus the baseline method (SGD) and the layer freezing technique we are comparing. More specifically, we measured the MMAC (Mega Multiply-Accumulate), a FLOPs metric that counts the number of matrix multiplications and accumulations (MACs) a neural network performs in one second. The metric is expressed in millions (mega) of operations and is useful for evaluating the computational complexity of a neural network and comparing the performance of different architectures. Figure 6 shows, on the left, the number of parameters per each epoch for the ResNet-101 (top) and VGG-16 (bottom). Red curves

Table 3 FLOPs reduction across architectures

	SGD	Dropping (Ours)	
Network	FLOPs	FLOPs	ΔFLOPs (%)
VGG-11	31,203.60	**25,847.02**	17.17
VGG-16	50,311.19	**33,049.44**	34.31
VGG-19	66,000.00	**44,079.31**	33.21
ResNet-18	1,987.80	**640.53**	67.78
ResNet-50	4,704.59	**2,193.45**	53.38
ResNet-101	9,262.2	**1,667.27**	82.00
ResNet-152	13,823.39	**2,247.46**	83.74

SGD refers to the standard training strategy of the entire model. Dropping is our method where layers are deleted from the trained model. FLOPs are measured during the forward propagation. ΔFLOPs is the percentage of reduced FLOPs with respect to SGD. Our approach reduces the FLOPs of all architectures, especially of the largest ones

Bold values indicate the best metric value

represent the number of model parameters when using the baseline and the layer freezing techniques; Green curves are the number of parameters when using our method. Since our method compresses the model at training time, there is a strong reduction in the number of parameters corresponding to the epochs when layer dropping arises. This reduction in parameters is correlated with the MMAC reduction during forward propagation, shown in the plots on the right. While the MMAC is constant for the baseline and the layer freezing techniques, in our method, it keeps decreasing due to the shrinkage in the number of parameters.

Table 3 reports the FLOPS required for the forward propagation during training by using SGD versus our approach (Dropping). The FLOPs refer to various deep learning models, including VGG-11, VGG-16, VGG-19, ResNet-18, ResNet-50, ResNet-101, and ResNet-152. The second column shows the FLOPs required when training the entire model, the third column shows the FLOPs required when using our approach. The final column ΔFLOPs show the percentage difference between the FLOPs of the two approaches.

Overall, Table 3 suggests that the Dropping approach is more efficient than the baseline method. The percentage difference between the two approaches ranges from 17.17% (for VGG-11) to 83.74% (for ResNet-152), indicating that Dropping can significantly reduce the computational burden of training deep neural networks, especially for very deep models like ResNet-101 and ResNet-152. Table 3 also shows that the reduction in the FLOPs increases with the depth of the model. Moreover, as also indicated by the magnitude of the FLOPs, the VGG family performs more operations than the ResNet due to the presence of dense layers applied to feature maps of greater size.

6 Conclusion and future works

This work proposed a method to improve the training efficiency of convolutional networks. The method reduces the computational cost of the forward propagation by gradually dropping subsequent layers from the model to train based on the computed parameter gradient. This is different from previous work, which freeze layers without removing them.

The method has been validated on three popular datasets to train models of various depth in the VGG and ResNet families. On average, our method achieves a time reduction in 62.87% on VGG architectures, and of 66.30% on ResNet models with a limited impact on the model accuracy that, in our experiments, never exceeds the 3.2% for very deep network (ResNet-152).

A typical limitation of fast training and network compression algorithms is that the network must be initialized through a warm-up training. Rather than using an empirical number of warm-up epochs, our method might use a threshold on the layer score to decide when the warm-up is ended. Since our method significantly reduces the training time, it might also alternate among training the whole model and the head model to refine the first dropped layers' weights. This strategy would account for the accuracy drop of the model.

In future work, we intend to investigate to what extent our approach can be applied to other more complex architectures such as multi-branches and recurrent memory cells. In multi-branch networks, one should check whether learning proceeds similarly along each of the branches. In recurrent memory cells, it can be interesting to study how the gradient-based scores of these memory cells change over the epochs. Furthermore, it will be useful to study whether similar techniques can be applied to visual transformers.

Finally, we will also investigate an adaptive method to estimate optimal batch size while layer dropping is applied. Indeed, as the model size is reduced over time, the memory usage for feature maps and gradients reduces accordingly making it possible to dynamically increase the batch size and reduces the number of iterations in an epoch.

Funding Open access funding provided by Università degli Studi di Palermo within the CRUI-CARE Agreement.

Data availability statement The data that support the findings of this study are openly available. The Minst dataset can be downloaded

from http://yann.lecun.com/exdb/mnist/. The Cifar-10 dataset can be downloaded from https://www.cs.toronto.edu/~kriz/cifar.html. The Imaginette dataset can be downloaded from https://github.com/fastai/imagenette/. No new data were created in this study.

Declarations

Conflicts of interest All authors declare that they have no conflicts of interest.

References

1. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444
2. Menghani G (2023) Efficient deep learning: a survey on making deep learning models smaller, faster, and better. ACM Comput Surv 55(12):1–37
3. Molchanov P, Mallya A, Tyree S, Frosio I, Kautz J (2019) Importance estimation for neural network pruning. In: proceedings of the IEEE/CVF conference on computer vision and pattern recognition, p. 11264–11272
4. Choudhary T, Mishra V, Goswami A, Sarangapani J (2022) Heuristic-based automatic pruning of deep neural networks. Neural Comput Appl 34(6):4889–4903
5. Zemouri R, Omri N, Fnaiech F, Zerhouni N, Fnaiech N (2020) A new growing pruning deep learning neural network algorithm (GP-DLNN). Neural Comput Appl 32:18143–18159
6. He Y, Zhang X, Sun J (2017) Channel pruning for accelerating very deep neural networks. In: proceedings of the IEEE international conference on computer vision, p 1389–1397
7. Xu S, Chen H, Gong X, Liu K, Lü J, Zhang B (2021) Efficient structured pruning based on deep feature stabilization. Neural Comput Appl 33(13):7409–7420
8. Xiao X, Mudiyanselage TB, Ji C, Hu J, Pan Y (2019) Fast deep learning training through intelligently freezing layers. In: 2019 international conference on internet of things (iThings) and IEEE green computing and communications (greencom) and IEEE cyber, physical and social computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, p 1225–1232
9. Liu S, Ni'mah I, Menkovski V, Mocanu DC, Pechenizkiy M (2021) Efficient and effective training of sparse recurrent neural networks. Neural Comput Appl 33:9625–9636
10. Zhang J, Chen X, Song M, Li T, (2019) Eager pruning: Algorithm and architecture support for fast training of deep neural networks. In: (2019) ACM/IEEE 46th annual international symposium on computer architecture (ISCA). IEEE pp 292–303
11. Simonyan K, Zisserman A (2015) Very deep convolutional networks for large-scale image recognition. In: 3rd international conference on learning representations (ICLR 2015). Computational and Biological Learning Society
12. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: proceedings of the IEEE conference on computer vision and pattern recognition; 2016. p. 770–778
13. LeCun Y, Denker J, Solla S (1989) Optimal brain damage. Adv Neural Inform Process Syst 2:89
14. Hassibi B, Stork DG, Wolff GJ (1993) Optimal brain surgeon and general network pruning. In: IEEE international conference on neural networks. IEEE p 293–299
15. Luo JH, Wu J, Lin W (2017) Thinet: A filter level pruning method for deep neural network compression. In: proceedings of the IEEE international conference on computer vision, pp 5058–5066
16. Han S, Mao H, Dally WJ. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149. 2015;
17. Han S, Pool J, Tran J, Dally W (2015) Learning both weights and connections for efficient neural network. Adv Neural Information Process Syst 28:e10246
18. Dettmers T, Zettlemoyer L (2019) Sparse networks from scratch: Faster training without losing performance. arXiv preprint arXiv:1907.04840
19. Veit A, Wilber MJ, Belongie S (2016) Residual networks behave like ensembles of relatively shallow networks. Adv Neural Information Process Syst 29:1
20. Polyak A, Wolf L (2015) Channel-level acceleration of deep face representations. IEEE Access 3:2163–2175
21. Chen S, Zhao Q (2018) Shallowing deep networks: layer-wise pruning based on feature representations. IEEE Trans Patt Anal Mach Intell 41(12):3048–3056
22. Xu P, Cao J, Shang F, Sun W, Li P (2020) Layer pruning via fusible residual convolutional block for deep neural networks. arXiv preprint arXiv:2011.14356
23. Elkerdawy S, Elhoushi M, Singh A, Zhang H, Ray N (2020) To filter prune, or to layer prune, that is the question. In: proceedings of the Asian conference on computer vision, p 1–17
24. Tan D, Zhong W, Peng X, Wang Q, Mahalec V (2020) Accurate and fast deep evolutionary networks structured representation through activating and freezing dense networks. IEEE Trans Cognit Develop Syst
25. Van Rossum G (2020) The python library reference, release 3.8.2. python software foundation. https://github.com/python/cpython/blob/3.11/Lib/pickle.py
26. Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D et al (2020) Array programming with NumPy. Nature 585(7825):357–362. https://doi.org/10.1038/s41586-020-2649-2
27. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al (2019) PyTorch: an imperative style, high-performance deep learning library. In: advances in neural information processing systems 32. Curran Associates, Inc. p. 8024–8035. https://pytorch.org/
28. LeCun Y, Bengio Y et al (1995) Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks 3361(10):1995
29. Krizhevsky A, Hinton G, et al (2009) Learning multiple layers of features from tiny images. Technical Report. p 32–33
30. Howard J. Imagenette;https://github.com/fastai/imagenette/
31. Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S et al (2015) Imagenet large scale visual recognition challenge. Int J Comput Vis 115:211–252