Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Simplify: A Python library for optimizing pruned neural networks

Andrea Bragagnolo [a,b,*], Carlo Alberto Barbano [a]

[a] Computer Science Department, University of Turin, Turin, Italy
[b] Synesthesia s.r.l., Turin, Italy

ARTICLE INFO

Article history:
Received 30 July 2021
Received in revised form 17 November 2021
Accepted 18 November 2021

Keywords:
Deep learning
Simplification
Optimization
Pruning
PyTorch

ABSTRACT

Neural network pruning allows for impressive theoretical reduction of models sizes and complexity. However it usually offers little practical benefits as it is most often limited to just zeroing out weights, without actually removing the pruned parameters. This precludes from the actual advantages provided by sparsification methods. We propose **Simplify**, a PyTorch compatible library for achieving effective model simplification. Simplified models benefit of both a smaller memory footprint and a lower inference time, making their deployment to embedded or mobile devices much more efficient.

Code metadata

Current code version	v1.0
Permanent link to code/repository used of this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00143
Legal Code License	BSD-3-Clause
Code versioning system used	Git
Software code languages, tools, and services used	Python
Compilation requirements, operating environments & dependencies	PyTorch
If available Link to developer documentation/manual	https://github.com/EIDOSlab/simplify
Support email for questions	eidoslab@di.unito.it

1. Motivation and significance

Over the last few years, neural network pruning (i.e. the reduction of the size and complexity of a model through the removal of a set of parameters) has been the subject of extensive research in the scientific community [1–8].

Modern pruning techniques allow for impressive theoretical reduction in both memory requirements and inference time for state-of-the-art neural network architectures. However, most procedures are limited to only identifying which portion of the weights can be set to zero, offering little to no practical advantages when the model is deployed to resource-constrained devices such as mobile phones or embedded systems. While most of the pruning-related works report some form of theoretical speedup, either in terms of Floating Point Operations (FLOPs) or inference speed [9], this does not always reflect the actual achievable performance gain and it is usually overestimated.

To solve this issue, we propose *Simplify*,[1] a PyTorch [10] compatible simplification library that, allows to obtain an actually smaller model in which the pruned neurons are removed and do not weight on the size and inference time of the network. This technique can be used to correctly evaluate the actual impact of a pruning procedure when applied to a given network architecture. Moreover, *Simplify* allows to apply the simplification process even at training time, in conjunction with pruning techniques, thus reducing the required time for pruning and fine-tuning neural networks. A high level representation of the pruning and simplification pipeline is given in Fig. 1

In the related literature it is possible to encounter two class of pruning procedures: *unstructured* and *structured*. Unstructured pruning approaches remove single parameters from the network, independently from one another [11–16]. When employing this

* Correspondence to: Dipartimento di Informatica, Corso Svizzera 185, 10049, Torino

E-mail addresses: andrea.bragagnolo@unito.it (Andrea Bragagnolo), carlo.barbano@unito.it (Carlo Alberto Barbano).

1 https://github.com/EIDOSlab/simplify.

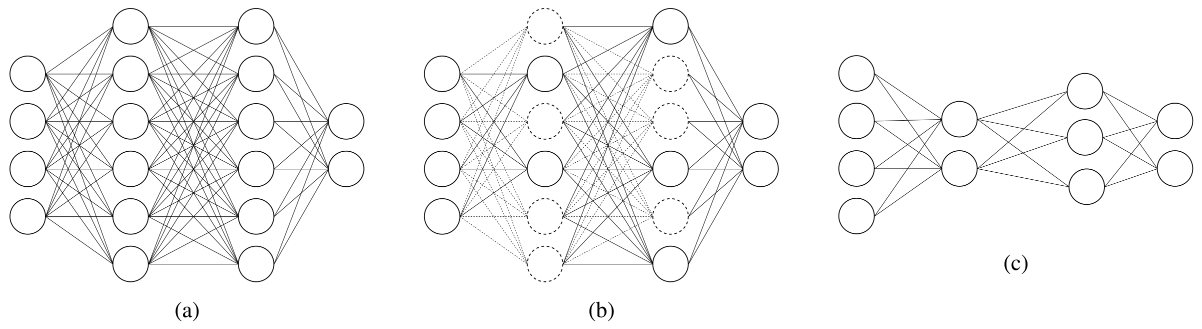


Fig. 1. Overview of the simplification procedure: *(a)* dense network *(b)* pruned network (dotted lines represent pruned neurons and connections) *(c)* simplified network in which the pruned neurons are actually removed from the architecture.

kind of techniques, one can obtain a high degree of sparsity, but the pruning of entire neurons is not guaranteed. Structured approaches, on the other hand, focus on the removal of whole neurons, leading to the imposition of some kind of structure over the pruned topology [7,17,18]. Since our proposed library removes the pruned neurons from the network, we will focus on models pruned using structured techniques.

Various accelerators, both hardware and software, for sparse neural networks have been proposed [19–22]. The main downside of this kind of solutions is the requirement for specific hardware or software, that can be hardly applied to standard consumer devices. Furthermore, they are designed to apply inference-time acceleration using the zero-filled model instead of building an optimized structure, thus precluding the ability to train a pruned neural network.

Simplify solves these issues by extracting the remaining structure from a pruned model, and removing all the zeroed-out neurons from the network. This allows to obtain a model that can be saved, shared and used without any special hardware or software. While, at a first glance, this may seem a straightforward procedure, the removal of zeroed neurons poses some hidden challenges like the presence of bias in said neurons or some constraints in the output's dimensions due to skip or residual connections. Even though the interest of the deep learning community on the matter seems to be quite strong,[2] very few approaches and libraries for simplifying pruned models have been proposed.[3] Moreover, they are usually limited to simpler architectures such as VGG [23], and their usage is restricted to the deployment of an already pruned model. On the other hand, with *Simplify*, we provide a way to:

(1) Optimize more complex network architectures (e.g. ResNet [24], DenseNet [25] and so on), and, in general, custom architectures, without constraints given by the connectivity patterns (i.e. residual connections);
(2) Optimize models during training: this allows to obtain speed-ups in the time required for training a model and reduce the memory occupation, when applied together with an iterative pruning technique.

2. Software description

The *Simplify* library leverages on the main PyTorch packages and is composed of three main modules that, even if designed to function in a predefined order, can be used independently based on the user requirements. We now provide a brief overview of each module functionalities and purpose. A more detailed explanation of the maths involved in each module is provided in the Appendix.

Fuse First, we have the *fuse* module. Here we perform a non-mandatory optimization of the model by merging, in a single Convolutional layer, pairs of consecutive Convolutional and Batch Normalization layers. This process is known as Batch Normalization fusion or folding. This step can be ignored if the presence of Batch Normalization layers in the network is required, i.e. for further training of the simplified model. This step is not needed to define the simplified model, but provides inference-time and memory usage advantages, especially when deploying a trained model to production, thanks to an optimization of the model architecture.

Propagate The second module is called *propagate*. With this module we solve the problem of non-zero bias in zeroed neurons mentioned in Section 1. It is possible that some pruned neuron retain non-zero bias; in such situation it would be impossible to remove the neuron without losing the bias contribution. To solve this problem, in the *propagate* module, we essentially treat such neurons as a constant signal that can then be absorbed by the next layer, making the zeroed neuron removable.

Remove Lastly, with the *remove* module we perform the actual simplification of the model, removing the zeroed-out neurons. Here we make sure that the output and input dimensions of adjacent layers correspond, while also taking into account architecture constraints such as the presence of skip connections.

3. Illustrative examples

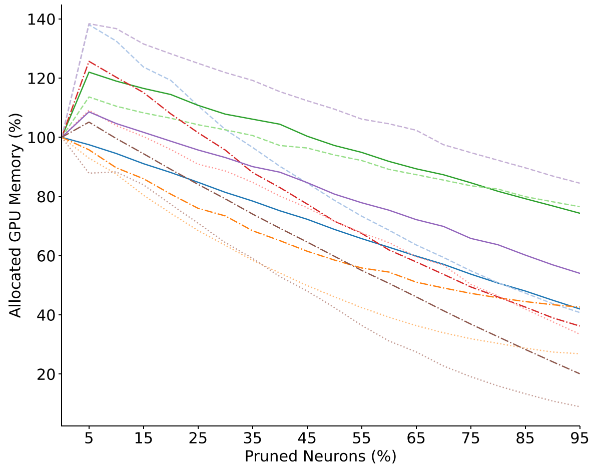
In this section we provide an usage overview for *Simplify*. We also illustrate the results obtained for the two different use cases discussed in Section 1, namely optimization for model deployment and optimization during training. Please note that our experiments were performed on different image classification networks, made available by the Torchvision library.[4] Such models are defined for the ImageNet [26] dataset and therefore expect an input of size $B \times 3 \times 224 \times 224$, where B represents the batch size.

Optimization for deployment This is the most common use case. Here, the simplification procedure is applied on an already trained model, on which a pruning criterion has been previously applied. In most cases, a one-line call to the `simplify` method is sufficient: the library performs all the three steps autonomously, and takes care of different architectural patterns such as residual connections. Below, we provide a sample code snippet.

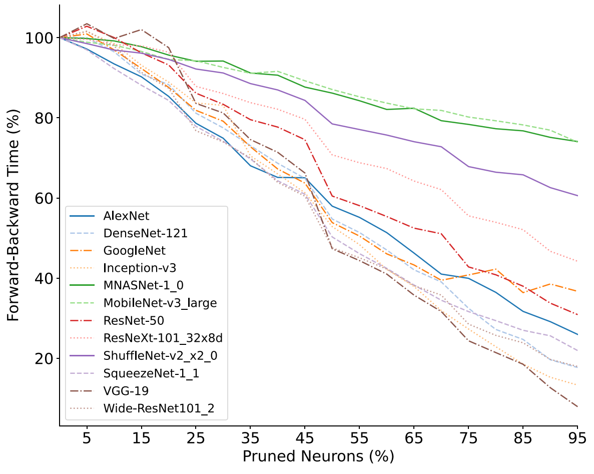
[2] Some examples can be found at: https://github.com/pytorch/pytorch/issues/47915, https://github.com/pytorch/pytorch/issues/43552, https://github.com/pytorch/pytorch/issues/36214, https://github.com/pytorch/pytorch/issues/32928, https://discuss.pytorch.org/t/pruning-doesnt-affect-speed-nor-memory-for-resnet-101/75814, https://discuss.pytorch.org/t/discussion-of-practical-speedup-for-pruning/75212, https://discuss.pytorch.org/t/save-pruned-model-after-pruning/103212.

[3] Some experimental implementations of simplification procedures: https://github.com/jacobgil/pytorch-pruning, https://github.com/microsoft/nni.

[4] https://pytorch.org/vision/stable/models.html#classification.



(a)



(b)

Fig. 2. Simplification during training: (a) Allocated GPU memory for a forward/backward pass of different pruned models (b) Total time for a forward and a backward pass of different pruned models.

Table 1
Inference time for different dense, pruned and simplified torchvision models.

Architecture	Inference time (ms)		
	Dense	Pruned	Simplified
AlexNet [27]	7.58 ± 0.29	7.55 ± 0.28	2.95 ± 0.02
DenseNet-121 [25]	36.41 ± 4.88	34.31 ± 3.85	21.87 ± 1.45
GoogLeNet [28]	15.44 ± 3.19	13.68 ± 0.09	10.31 ± 0.82
InceptionV3 [29]	25.29 ± 7.31	21.68 ± 2.90	13.22 ± 2.23
MNASNet [30]	17.66 ± 0.57	13.64 ± 0.13	11.59 ± 0.07
MobileNetV3 [31]	13.74 ± 0.67	12.18 ± 0.46	11.95 ± 0.21
ResNet-50 [24]	24.39 ± 4.48	26.19 ± 5.84	18.21 ± 1.98
ResNeXt-101 [32]	76.11 ± 15.79	77.35 ± 20.04	65.68 ± 16.41
ShuffleNetV2 [33]	18.07 ± 2.23	14.32 ± 0.21	13.06 ± 0.08
SqueezeNet [34]	4.50 ± 0.06	4.39 ± 0.05	4.09 ± 0.50
VGG-19 [23]	40.41 ± 12.13	38.56 ± 10.72	12.39 ± 0.19
WideResNet-101 [35]	79.40 ± 25.57	82.86 ± 22.47	60.16 ± 10.77

```
# Load a pruned model checkpoint
model = torch.load(..)

# Apply simplification.
model.eval()
simplify(model, torch.zeros(1, 3, 224, 224))
```

As show in the code snippet, the `simplify` function takes as argument the model to be simplified and a tensor representing a input image, filled with zeros, with batch size of 1 (in this case we are working with models build for ImageNet therefore is of size 1 × 3 × 224 × 224). Is important to note that the model has to be set to evaluation mode before the simplification procedure, so that the automatic update of parameters (such as for Batch Normalization) is disabled. The simplification of the model happens in-place.

Table 1 shows the inference times (in milliseconds) of different Torchvision models. In this table we compare the inference speed of the dense models (i.e. models that have not been pruned), the resulting pruned architectures (random, structured pruning with 50% probability) and the simplified model obtained with our proposed library. The benchmarks are run on a Intel(R) Core(TM) i9-9900K CPU, with a batch size of 1 in order to simulate a one-shot inference of a deployed model. The results are averaged across 1000 different runs for each architecture. It is easy to see that, thanks to *Simplify*, the resulting model is actually faster and able to leverage on the applied pruning while remaining a fully-fledged PyTorch network. We can see that, for most network families, *Simplify* allows for a decrease in the actual inference time. It is important to point out that for some architectures, like MobileNet or SqueezeNet, the library may not lead to great speed-up as they are already very optimized.

Optimization for training Most modern network architectures employ Batch Normalization as a way to improve generalization. To avoid losing the Batch Normalization contribution, we provide the ability to avoid the fusion step, so that these layers are retained. To further improve training time, it is possible to enable a *training* mode for `simplify`, which helps in decreasing inference time. More details are provided in Appendix C.1. Below, we provide a sample code snippet.

```
for step in range(epochs):
    model.train()
    model = ... # train model and apply pruning

    # Apply simplification
    model.eval()
    simplify(model,
             torch.zeros(1, 3, 224, 224),
             fuse_bn=False,
             training=True)
```

Fig. 2(a) shows the reduction in allocated GPU memory for different pruning ratios. While an overhead is introduced at low pruning percentages (due to the steps described in Appendix B and Appendix B.3), a reduction in the required memory for performing an optimization step is achieved when pruning a sufficient amount of neurons. In Fig. 2(b) we show the decrease in time required by the forward and backward pass of a network training loop. Appendix D shows the plots for each architecture separately. The benchmarks are run on a NVIDIA RTX 2080Ti GPU, with a Intel(R) Core(TM) i9-9960X CPU, using a batch size of 64.

4. Impact

Current state-of-the-art pruning research base their results on theoretical estimation of the models improvement. They offer poor practical benefits due to the lack of removal of pruned neurons that still weight on the model computation, especially when deployed to resource constrained devices like mobile phones.

Simplify provides out of the box functionalities to translate the impressive theoretical results of pruning procedure to an actual shrinking of the neural network model, reducing both memory requirements and inference time. It allows for a more precise evaluation of pruning procedures, enabling systematic comparison within scientific research, and helps during deployment, allowing for the full exploitation of the pruned network without the need for ad hoc hardware platforms. In such regard the proposed library was used in different research works [7,36,37] to evaluate the advantages of structured pruning procedures.

5. Conclusions

We propose the PyTorch compatible library *Simplify*, with the aim of providing a simple-to-use set of procedures to remove zeroed neurons from a neural network architecture. The proposed library solves different issues in the creation of simplified models, such as the propagation of the bias of pruned neurons and the shape constraint of skip connections.

The proposed library is composed of three modules that, while designed to work together, can be used independently from one another according to the required functionality for a specific setting.

CRediT authorship contribution statement

Andrea Bragagnolo: Conceptualization, Methodology, Software. **Carlo Alberto Barbano:** Conceptualization, Methodology, Software.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Batch normalization fusion

A vast amount of modern neural networks use Batch Normalization (from here on out BatchNorm) as a way to improve generalization. Given an input x, we can define the output of BatchNorm as:

$$y = \gamma \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} + \beta \tag{1}$$

where γ and β represent, respectively, the weights and bias of the layer and are learned using standard backpropagation procedures; μ and σ^2 represent the mean and variance computed over a batch. During training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. Let us denote this approximations as $\widehat{\mu}$ and $\widehat{\sigma}^2$. Notice that each parameter is defined for each channel of the input feature map; we will denote them as γ_c, β_c, $\widehat{\mu}_c$ and $\widehat{\sigma}_c^2$ for a given channel c.

Once a neural network is trained to completion, all the parameters of its layer can be considered frozen i.e. no longer update from further training. Also in standard network architectures, is possible to identify pairs of Convolutional and BatchNorm layers whose output is of the same size. In such conditions is possible to reduce the network complexity by fusing this two layers them into a single one. Note that this operation is only applicable if there is no non-linearity between the two layers.

Let us consider a generic Batch Normalization's output

$$y = \gamma_c \frac{x-\widehat{\mu}}{\sqrt{\widehat{\sigma}^2+\epsilon}} + \beta \tag{2}$$

this can be rewritten as

$$y = \frac{\gamma}{\sqrt{\widehat{\sigma}^2+\epsilon}} x - \frac{\gamma}{\sqrt{\widehat{\sigma}^2+\epsilon}} \widehat{\mu} + \beta \tag{3}$$

since this BatchNorm layer is preceded by a Convolutional layer, x_c can be defined as

$$x = W \cdot z + b \tag{4}$$

where z_c is the input of the Convolutional layer, W are its weights and b its bias.

We can now express the BatchNorm output as a function of the Convolutional layer, substituting Eq. (4) in Eq. (3).

$$y = \frac{\gamma}{\sqrt{\widehat{\sigma}^2+\epsilon}} (W \cdot z + b) - \frac{\gamma}{\sqrt{\widehat{\sigma}^2+\epsilon}} \widehat{\mu} + \beta \tag{5}$$

Leveraging on Eq. (5), we can finally fuse the Convolutional and the BatchNorm layer in a single Convolutional layer whose weights and bias are defined as

$$W_{fuse} = \gamma \frac{W}{\sqrt{\widehat{\sigma}^2+\epsilon}} \tag{6}$$

$$b_{fuse} = \gamma \frac{b-\widehat{\mu}}{\sqrt{\widehat{\sigma}^2+\epsilon}} + \beta \tag{7}$$

and the output y is therefore

$$y = W_{fuse} \cdot z + b_{fuse} \tag{8}$$

Appendix B. Bias propagation

This step is necessary if biases are presents in the model's hidden layers, or are introduced by the fusion of batch normalization layers. Neurons with zeroed-out channels might have non-zero bias, and so they will fire a constant output value. Hence, a neuron cannot immediately be removed if the corresponding bias is nonzero. These values, however, can be propagated and accumulated into the biases of the next layer. This operation can be repeated until all of the biases have been propagated to the last layer of the network. After a bias has been propagated, it can then be set to zero in the original neuron, which in turn allows the removal of the whole weight channel.

B.1. Linear layers

We denote as $L_1 = \langle A, a \rangle$ and $L_2 = \langle B, b \rangle$ two sequential linear layers. A and a denote the weight matrix and bias vector of L_1, of size $N \times M$ and N respectively. B and b denote the weight matrix and bias vector of L_2 of size $T \times N$ and T respectively. We also denote as f the activation function (e.g. ReLU). A forward pass for L_1 consists in:

$$y = f(xA^T + a) \tag{9}$$

(where x represents an input vector of size M) and for L_2:

$$z = yB^T + b \tag{10}$$

Focusing on Eq. (9), we can visualize the vector–matrix product:

$$y = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_M \end{bmatrix}^T \cdot \begin{bmatrix} A_{0,0} & \dots & A_{N,0} \\ A_{0,1} & \dots & A_{N,1} \\ \vdots & \ddots & \vdots \\ A_{0,M} & \dots & A_{N,M} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix}^T = \begin{bmatrix} xA_0^T + a_0 \\ xA_1^T + a_1 \\ \vdots \\ xA_N^T + a_N \end{bmatrix}^T$$

We now suppose that some output channel of A has been zeored-out following the application of some pruning criterion, e.g. every entry in A_1 is zero. The multiplication becomes:

$$y = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_M \end{bmatrix}^T \cdot \begin{bmatrix} A_{0,0} & 0 & \dots & A_{N,0} \\ A_{0,1} & 0 & \dots & A_{N,1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{0,M} & 0 & \dots & A_{N,M} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix}^T = \begin{bmatrix} xA_0^T + a_0 \\ a_1 \\ \vdots \\ xA_N^T + a_N \end{bmatrix}^T$$

We now focus on the forward pass of L_2. As example, we analyze what happens with the first neuron B_0. If we rewrite Eq. (10) focusing on B_0 we obtain:

$$z_0 = f(xA_0^T + a_0)B_{0,0} + \boldsymbol{f(a_1)B_{0,1}} + \dots + f(xA_N^T + a_N)B_{0,N} + b_0 \quad (11)$$

We now focus on the forward pass of L_2. As example, we analyze what happens with the first neuron B_0. If we rewrite Eq. (10) focusing on B_0 we obtain:

$$z_0 = f(xA_0^T + a_0)B_{0,0} + \boldsymbol{f(a_1)B_{0,1}} + \dots + f(xA_N^T + a_N)B_{0,N} + b_0 \quad (12)$$

The term $f(a_1)B_{0,1}$ is a constant which can be accumulated into b_0. The same reasoning can be extended to all neurons in L_2, by adding $f(a_1)$ multiplied with the respective incoming weight to the neuron bias. The new set of biases $\widehat{b}$ for the layer can be written as:

$$\widehat{b} = \begin{bmatrix} b_0 + f(a_1)B_{0,1} \\ b_1 + f(a_1)B_{1,1} \\ \vdots \\ b_T + f(a_1)B_{T,1} \end{bmatrix}^T$$

and the original bias a_1 can be set to zero in L_1, resulting in $\widehat{a} = [a_0, 0, a_1, \dots, a_N]$. This procedure can be applied when multiple neurons are pruned in L_1 and the general rule to obtain the updated biases $\widehat{b}$ is as follows:

$$\widehat{b} = \begin{bmatrix} b_0 + \sum_i f(a_i)B_{0,i} \\ b_1 + \sum_i f(a_i)B_{1,i} \\ \vdots \\ b_T + \sum_i f(a_i)B_{T,i} \end{bmatrix}^T$$

where i represents the indices of zeroed channels in L_1. After the bias propagation procedure, the layers L_1 and L_2 can be replaced by $\widehat{L_1} = \langle A, \widehat{a} \rangle$ and $\widehat{L_2} = \langle B, \widehat{b} \rangle$ respectively.

B.2. Convolutional layers

A similar reasoning can be applied for convolutional layers. However, the propagation process needs to take into account whether the convolution employs zero-padding on the input tensor or not.

For the sake of simplicity, using the same notation of Appendix B.1, let us consider two sequential convolutional layers $L_1 = \langle A, a \rangle$ and $L_2 = \langle B, b \rangle$. We also assume that L_1 has one input channel and two output channels (A has shape $2 \times 1 \times H_1 \times W_1$ and a is a vector of length 2), while L_2 has two input channels and one output channels (B has shape $1 \times 2 \times H_2 \times W_2$, and b is a vector of length 1).

The forward pass for L_1 is:

$$y = \left(\begin{bmatrix} x * A_0 \\ x * A_1 \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \right)^T = \left(\begin{bmatrix} F^0 \\ F^1 \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \right)^T = \begin{bmatrix} F^0 + a_0 \\ F^1 + a_1 \end{bmatrix}^T$$

where $*$ represents the convolution operation and x is a properly sized input. In this context, the addition operation $+$ between the resulting feature map $F^i = x * A_i$ and the corresponding bias value a_i will perform a shape expansion of a_i to match the feature map shape, for example:

$$F^i + a_i = \begin{bmatrix} F^i_{0,0} & \dots & F^i_{0,H_{out}} \\ \vdots & \ddots & \vdots \\ F^i_{W_{out},0} & \dots & F^i_{W_{out},H_{out}} \end{bmatrix} + \begin{bmatrix} a_i & \dots & a_i \\ \vdots & \ddots & \vdots \\ a_i & \dots & a_i \end{bmatrix}$$

We now assume that the second channel A_1 of L_1 has been zeroed out after the application of some pruning criterion, hence if we consider $F^1 + a_1$ we obtain:

$$F^1 + a_1 = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} + \begin{bmatrix} a_1 & \dots & a_1 \\ \vdots & \ddots & \vdots \\ a_1 & \dots & a_1 \end{bmatrix} = \begin{bmatrix} a_1 & \dots & a_1 \\ \vdots & \ddots & \vdots \\ a_1 & \dots & a_1 \end{bmatrix}$$

Thus, y becomes:

$$\begin{bmatrix} F^0 + a_0 \\ \overline{a_1} \end{bmatrix}^T$$

where $\overline{\cdot}$ denotes that the element shape has been expanded.

We now analyze what happens with L_2. For the sake of simplicity, we assume that $W_{out} = H_{out} = 3$, that $W_2 = H_2 = 2$ and that every value of B is equal to 1. We also consider a stride value of 1 for L_2.

Convolution without padding (or "same" padding): This is the simpler case, and it is similar to the linear layers (Appendix B.1). The forward pass of L_2 can be expressed as follows:

$$z = f(F^0 + a_0) * B_{0,0} + \boldsymbol{f(\overline{a_1}) * B_{0,1}} + b_0 \quad (13)$$

The factor $f(\overline{a_1}) * B_{0,1}$ is constant and can be accumulated into b_0. Visualizing it, we obtain:

$$\begin{aligned} \widehat{b_0} &= \begin{bmatrix} f(a_1) & f(a_1) & f(a_1) \\ f(a_1) & f(a_1) & f(a_1) \\ f(a_1) & f(a_1) & f(a_1) \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + b_0 \\ &= \begin{bmatrix} 4f(a_1) & 4f(a_1) \\ 4f(a_1) & 4f(a_1) \end{bmatrix} + b_0 \end{aligned} \quad (14)$$

In this case, the updated bias can be converted as a scalar replacing the original value b_0: given that the resulting matrix is constant, we can directly factor out $4f(a_1)$ and set a_1 to 0 in L_1, obtaining a new bias $\widehat{b_0} = 4f(a_1) + b_0$ which will be used from now on in L_2.[5]

The same reasoning can be extended to the case of multiple neurons in the convolution layer and multiple pruned channel in the preceding layer: each bias value will be updated according to the rule in Eq. (14). The general rule to obtain the new bias vector $\widehat{b}$ can be expressed as follows:

$$\widehat{b} = \begin{bmatrix} b_0 + \sum_i f(\overline{a_i}) * B_{0,i} \\ b_1 + \sum_i f(\overline{a_i}) * B_{1,i} \\ \vdots \\ b_{C_{out}} + \sum_i f(\overline{a_i}) * B_{C_{out},i} \end{bmatrix} \quad (15)$$

where i represents the indices of the zeroed output channels in L_1.

Convolution with zero-padding If the convolution applies zero-padding to the input values, then the bias cannot be accumulated into a scalar, as the resulting matrix will not be constant.

[5] Notice that this can be applied for every choice of values for B. Of course the resulting bias factor will change accordingly.

To show this, we rewrite Eq. (14) applying a zero-padding of size 1 along each dimension of the input tensor:

$$\widehat{b}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a' & a' & a' & 0 \\ 0 & a' & a' & a' & 0 \\ 0 & a' & a' & a' & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + b_0 = \begin{bmatrix} a' & 2a' & 2a' & a' \\ 2a' & 4a' & 4a' & 2a' \\ 2a' & 4a' & 4a' & 2a' \\ a' & 2a' & 2a' & a' \end{bmatrix} + b_0 \quad (16)$$

where $a' = f(a_1)$ for brevity.

In this case, the new bias value need to be maintained in a matrix form, i.e.:

$$\widehat{b}_0 = \begin{bmatrix} a'+b_0 & 2a'+b_0 & 2a'+b_0 & a'+b_0 \\ 2a'+b_0 & 4a'+b_0 & 4a'+b_0 & 2a'+b_0 \\ 2a'+b_0 & 4a'+b_0 & 4a'+b_0 & 2a'+b_0 \\ a'+b_0 & 2a'+b_0 & 2a'+b_0 & a'+b_0 \end{bmatrix}$$

To obtain the updated biases in case of multiple neurons and multiple channels, the same rule of Eq. (15) can be applied, keeping in mind that in this case it will result into a tensor of shape $C_{out} \times H_{out} \times W_{out}$ instead of a vector. This introduces a constraint on the feature map size, hence the model can only ever be used at a fixed input size. However, given that the whole simplification procedure is executed on an already trained model, before deploying to production, it should not represent a major issue.

B.3. Residual connections

While the above process works fine for simple feed-forward models, special care must be taken to handle residual connections. As an example, let us consider the case of two linear layers $L_1 = \langle A, a \rangle$ and $L_2 = \langle C, c \rangle$, whose outputs y and t are summed together in a residual connection, followed by another layer $L_3 = \langle B, b \rangle$:

$$y + t = \left(\begin{bmatrix} xA_0^T & + & a_0 \\ 0 & + & a_1 \\ \vdots & & \\ 0 & + & a_{N-1} \\ xA_N^T & + & a_N \end{bmatrix} + \begin{bmatrix} 0 & + & c_0 \\ \hat{x}C_1^T & + & c_1 \\ \vdots & & \\ 0 & + & c_{N-1} \\ \hat{x}C_N^T & + & c_N \end{bmatrix} \right)^T \quad (17)$$

where 0 denotes that a channel was pruned. The residual (sum) operation introduces a new constraint: only biases corresponding to matching pruned channels in L_1 and L_2 can be propagated to the next layer. To see why, we can rewrite Eq. (17) as Eq. (12) and obtain:

$$\begin{aligned} z_0 =& f(xA_0^T + a_0 + c_0)B_{0,0} + f(a_1 + \hat{x}C_1^T + c_1)B_{0,1} + \cdots + \\ &+ \boldsymbol{f(a_{N-1} + c_{N-1})B_{0,N-1}} + \\ &+ f(xA_N^T + a_N + \hat{x}C_N^T + c_N)B_{0,N} + b_0 \end{aligned} \quad (18)$$

It is clear that even if multiple channels are pruned from L_1 and L_2, only the factor $f(a_{N-1} + c_{N-1})B_{0,N-1}$ becomes a constant. In this case, we opt not to propagate any bias and employ an expansion scheme (Appendix C.1) to achieve a speed-up in the convolution operations anyways.

Appendix C. Channels removal

Once the biases have been propagated and removed from the hidden layers, the weight matrices corresponding to zeroed channels can actually be removed. The process, which we call simplification, is actually quite simple. For each layer L, we denote with W^L the corresponding weight tensor, with shape $N \times I \times W \times H$ for convolutional layers and $N \times I$ for linear layers. The simplification consists of two steps:

1. Remove all the input channels corresponding to zeroed channels in the previous layer (none if it is the input layer):

$$\widehat{W}^L = \left[W^L_{0,i}, W^L_{1,i}, \ldots, W^L_{N,i} \right] \quad (19)$$

where i is the indices of the remaining output channels in W^{L-1}. The resulting weight tensor will be of shape $N \times I_s \times W \times H$ for convolutional layers and $N \times I_s$ for linear layers, where $I_s \leq I$ is the number of remaining output channels in the previous layer $L-1$.

2. Remove all the output channels corresponding to zeroed neurons:

$$\widetilde{W}^L = \left[\widehat{W}^L_j \right]_{\forall j \in J} \quad (20)$$

where J is the set of indices corresponding to the remaining output channels. The resulting tensor will be of shape $N_s \times I_s \times W \times H$ for convolutional layers and $N_s \times I_s$ for linear layers, where $N_s \leq N$ is the number of remaining (non-zero) output channels of L.

C.1. Residual connections

Residual or skip connections introduce a constraint on the output size of a layer. A residual connection consists in the sum of the output of two layers L_1 and L_2. As an example, we assume L_1 and L_2 to be convolutional layers, with their respective output Z_1 and Z_2 being of size $C_1 \times H \times W$ and $C_2 \times H \times W$ (ignoring the batch size). To be able to compute $Z_1 + Z_2$, C_1 must be equal to C_2. However, after the simplification step, it is possible that the output sizes differ, depending on whether some output channels in L_1 and/or L_2 were removed.

Many works that propose a solution to channel removal, face some issues when applied to residual connections. For example He et al. [3] or Kruglov [5] apply some kind of approximation and need to resort to finetuning to recover the lost performance. Others, like Hu et al. [4], instead, just ignore residual networks, without proposing any solution.

To address this issue, we perform an expansion operation on the output tensors. Assuming the original size (before the simplification) was C, then Z_1 and Z_2 are expanded to the original number of channels before performing the addition. The process is illustrated in Fig. 3. After the expansion step, we sum the layers biases (which were not propagated as explained in Appendix B.3).

While it is true that the expansion operations introduce a computational overhead in the model inference, the speedups achieved by the simplified convolutions compensate for it when using the model for inference. However, given that the time required by the indexing operations employed in the expansion scheme is actually dependent on the given batch size, we opt not to adopt this scheme when using *Simplify* in training mode. In this case, we do not remove any output channel in the weight tensor.

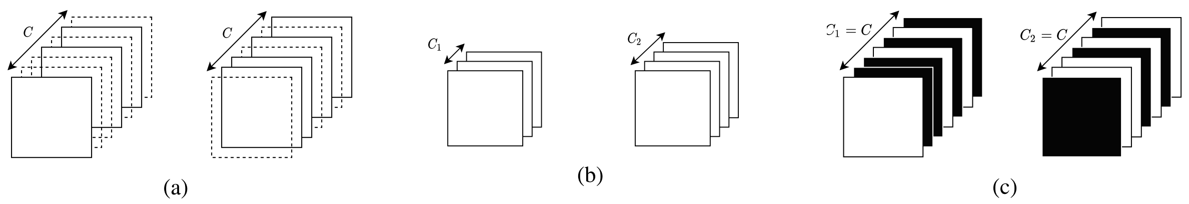
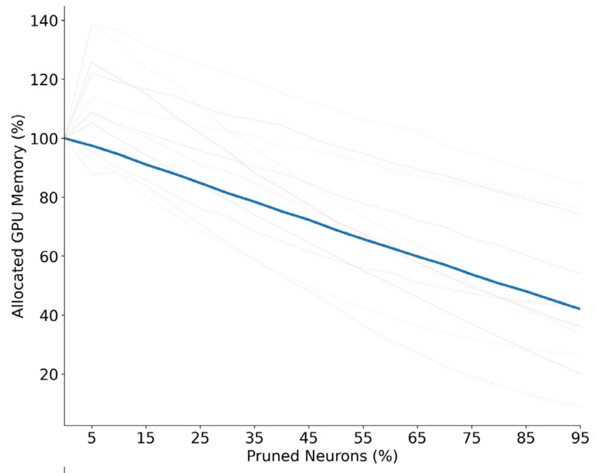
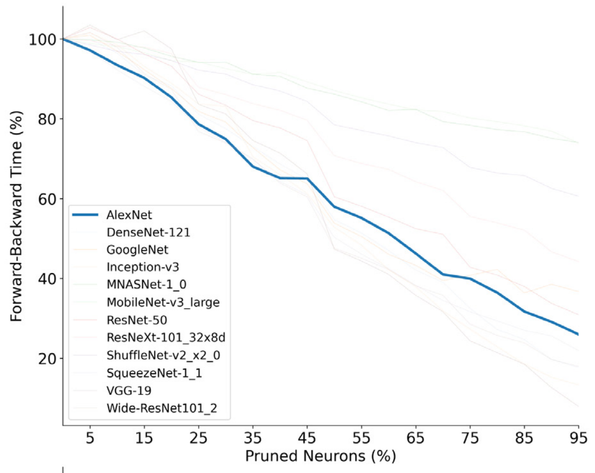
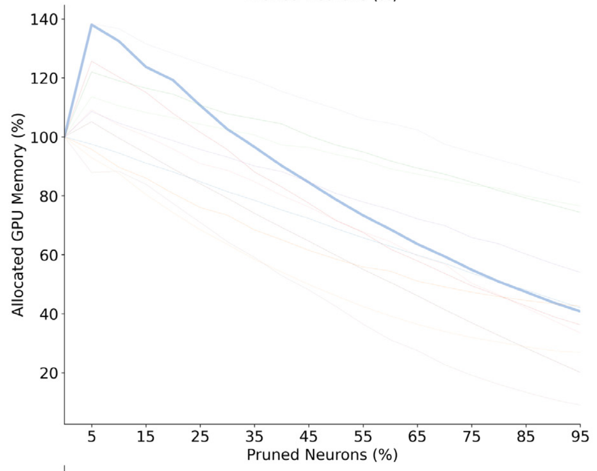


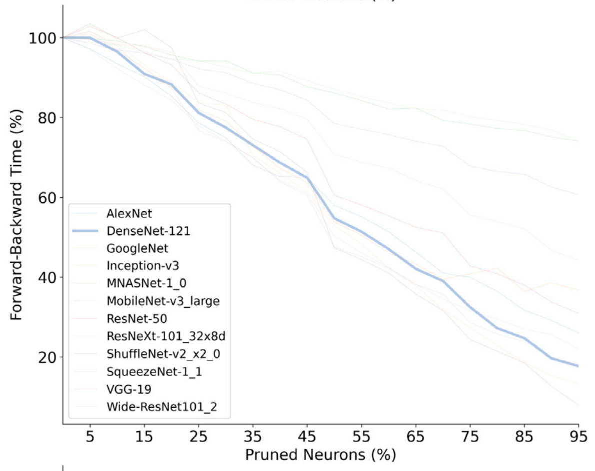
Fig. 3. Pruned weight matrices: a dotted line indicates a zeroed channel *(a)* simplified weight matrices *(b)* expanded weight matrices: black slices mean that the channel is a zero matrix *(c)*.

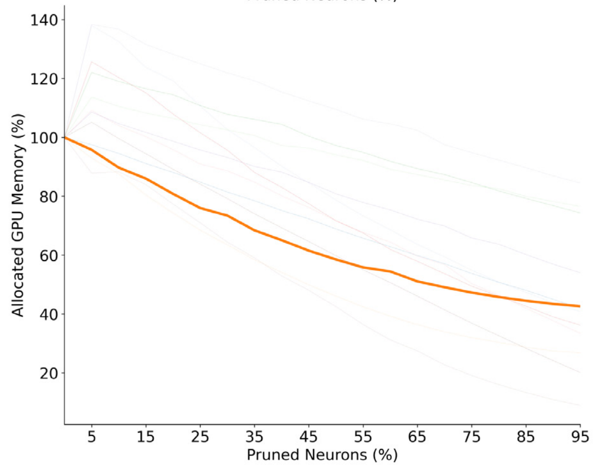
Appendix D. Additional results

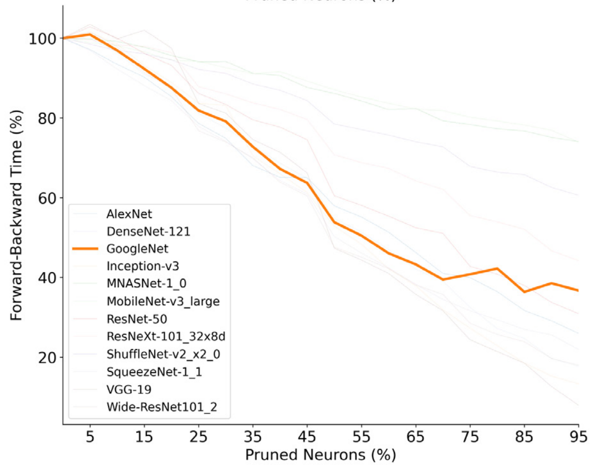


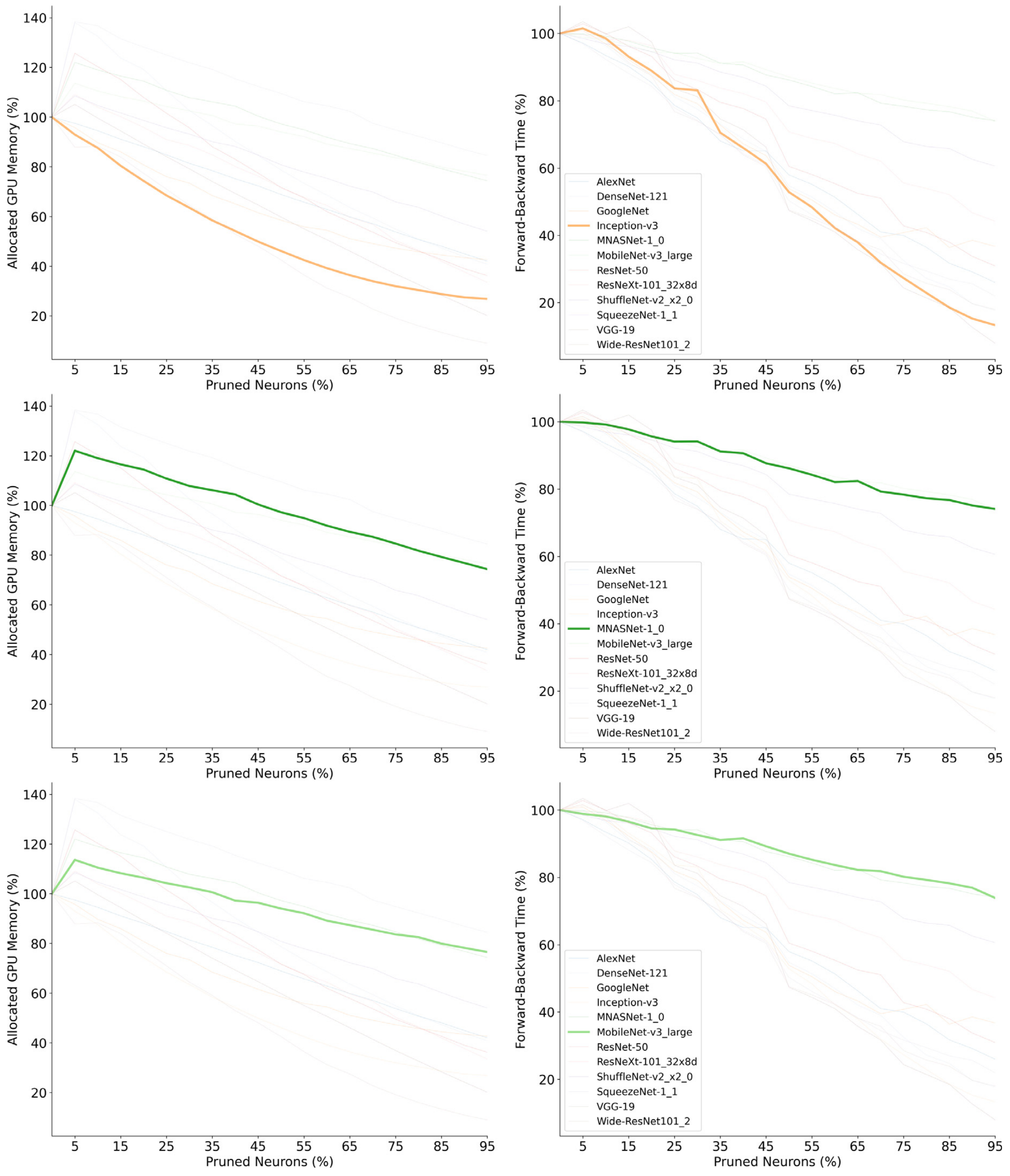


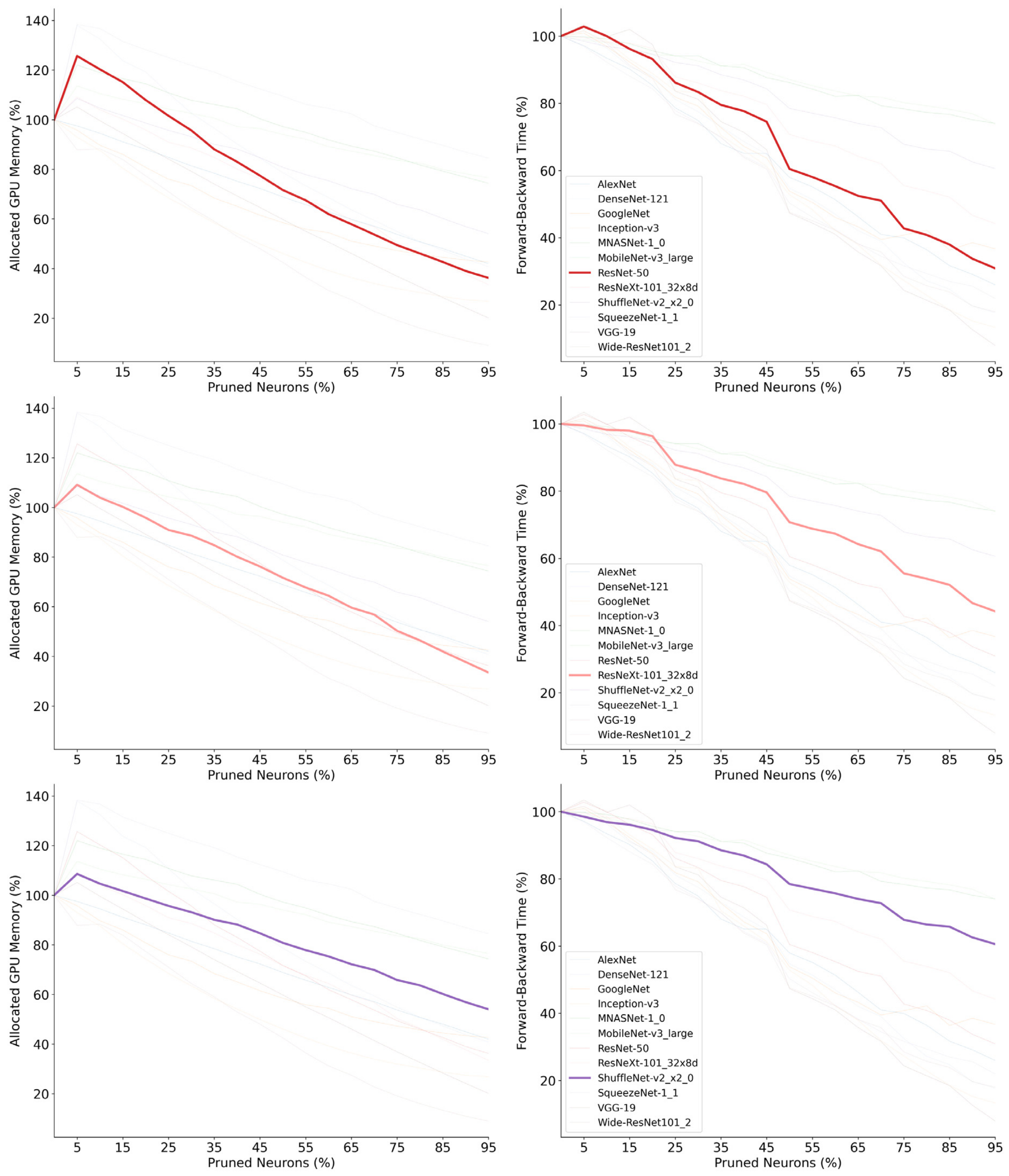


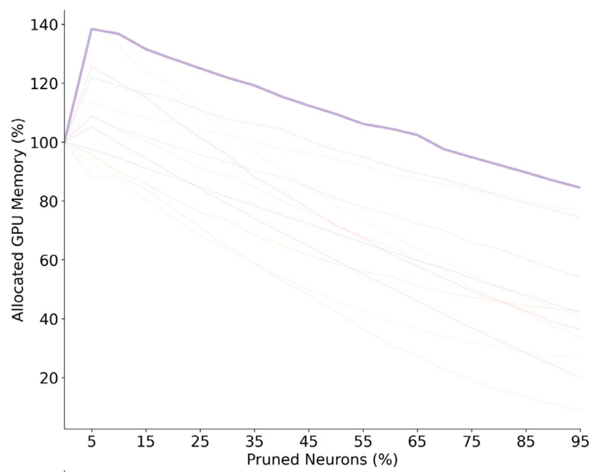


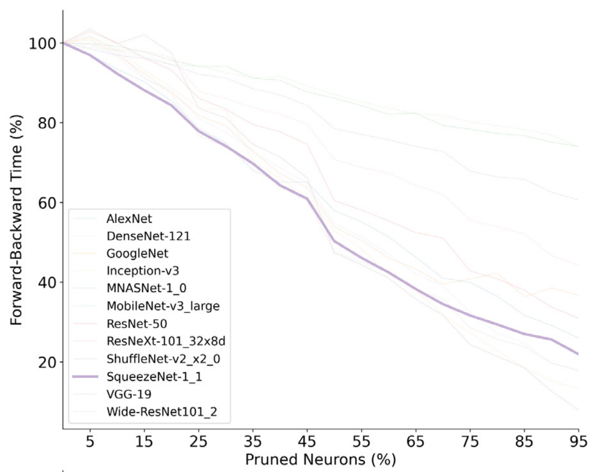


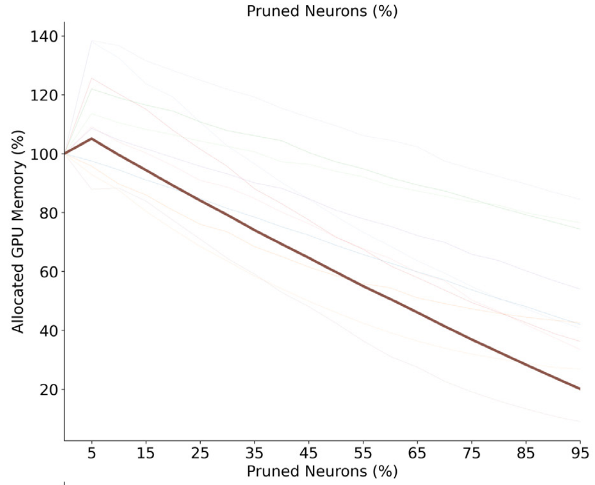


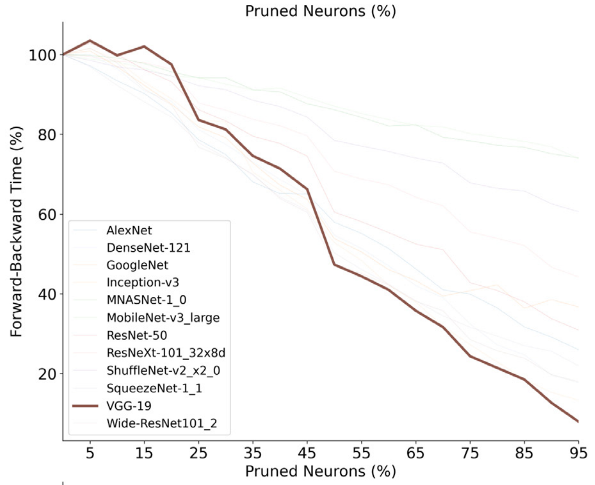

Allocated GPU Memory (%)
Forward-Backward Time (%)
Pruned Neurons (%)
AlexNet
DenseNet-121
GoogleNet
Inception-v3
MNASNet-1_0
MobileNet-v3_large
ResNet-50
ResNeXt-101_32x8d
ShuffleNet-v2_x2_0
SqueezeNet-1_1
VGG-19
Wide-ResNet101_2

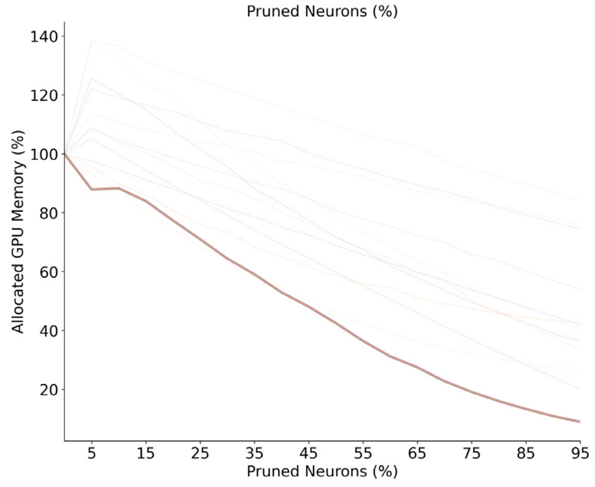

Allocated GPU Memory (%)
Forward-Backward Time (%)
Pruned Neurons (%)
5
15
25
35
45
55
65
75
85
95
20
40
60
80
100
120
140
AlexNet
DenseNet-121
GoogleNet
Inception-v3
MNASNet-1_0
MobileNet-v3_large
ResNet-50
ResNeXt-101_32x8d
ShuffleNet-v2_x2_0
SqueezeNet-1_1
VGG-19
Wide-ResNet101_2

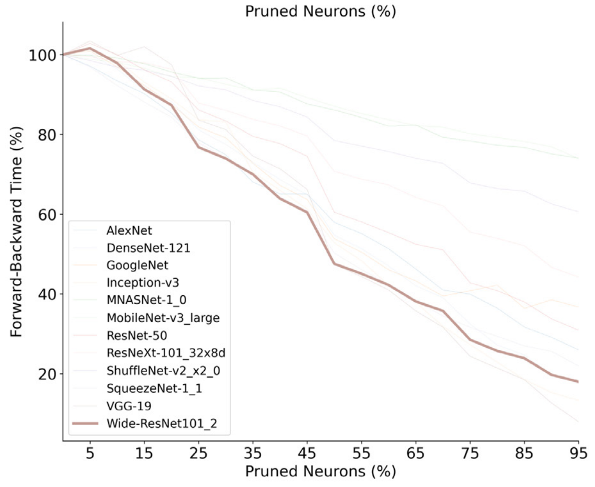

Allocated GPU Memory (%)
Pruned Neurons (%)
140
120
100
80
60
40
20
5 15 25 35 45 55 65 75 85 95


Forward-Backward Time (%)
Pruned Neurons (%)
100
80
60
40
20
5 15 25 35 45 55 65 75 85 95
AlexNet
DenseNet-121
GoogleNet
Inception-v3
MNASNet-1_0
MobileNet-v3_large
ResNet-50
ResNeXt-101_32x8d
ShuffleNet-v2_x2_0
SqueezeNet-1_1
VGG-19
Wide-ResNet101_2


Allocated GPU Memory (%)
Pruned Neurons (%)
140
120
100
80
60
40
20
5 15 25 35 45 55 65 75 85 95


Forward-Backward Time (%)
Pruned Neurons (%)
100
80
60
40
20
5 15 25 35 45 55 65 75 85 95
AlexNet
DenseNet-121
GoogleNet
Inception-v3
MNASNet-1_0
MobileNet-v3_large
ResNet-50
ResNeXt-101_32x8d
ShuffleNet-v2_x2_0
SqueezeNet-1_1
VGG-19
Wide-ResNet101_2


Allocated GPU Memory (%)
Pruned Neurons (%)
140
120
100
80
60
40
20
5 15 25 35 45 55 65 75 85 95


Forward-Backward Time (%)
Pruned Neurons (%)
100
80
60
40
20
5 15 25 35 45 55 65 75 85 95
AlexNet
DenseNet-121
GoogleNet
Inception-v3
MNASNet-1_0
MobileNet-v3_large
ResNet-50
ResNeXt-101_32x8d
ShuffleNet-v2_x2_0
SqueezeNet-1_1
VGG-19
Wide-ResNet101_2

References

[1] Gale T, Elsen E, Hooker S. The state of sparsity in deep neural networks. 2019, arXiv preprint arXiv:1902.09574.
[2] He Y, Liu P, Wang Z, Hu Z, Yang Y. Filter pruning via geometric median for deep convolutional neural networks acceleration. In: 2019 IEEE/CVF conference on computer vision and pattern recognition. 2019, p. 4335–44. http://dx.doi.org/10.1109/CVPR.2019.00447.
[3] He Y, Zhang X, Sun J. Channel pruning for accelerating very deep neural networks. In: 2017 IEEE international conference on computer vision. 2017, p. 1398–406. http://dx.doi.org/10.1109/ICCV.2017.155.
[4] Hu H, Peng R, Tai Y-W, Tang C-K. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. 2016, arXiv preprint arXiv:1607.03250.
[5] Kruglov A. Channel-wise pruning of neural networks with tapering resource constraint. 2018, arXiv preprint arXiv:1812.07060.
[6] Tang Y, Wang Y, Xu Y, Tao D, Xu C, Xu C, et al. SCOP: Scientific control for reliable neural network pruning. Adv Neural Inf Process Syst 2020;2020-December.
[7] Tartaglione E, Bragagnolo A, Odierna F, Fiandrotti A, Grangetto M. SeReNe: Sensitivity-based regularization of neurons for structured sparsity in neural networks. IEEE Trans Neural Netw Learn Syst 2021;1–14. http://dx.doi.org/10.1109/TNNLS.2021.3084527.
[8] Wang Y, Zhang X, Xie L, Zhou J, Su H, Zhang B, et al. Pruning from scratch. In: AAAI. 2020, p. 12273–80. http://dx.doi.org/10.1609/aaai.v34i07.6910.
[9] Blalock D, Gonzalez Ortiz JJ, Frankle J, Guttag J. What is the state of neural network pruning? In: Dhillon I, Papailiopoulos D, Sze V, editors. Proceedings of machine learning and systems, Vol. 2. 2020, p. 129–46, URL https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf.
[10] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. PyTorch: An imperative style, high-performance deep learning library. In: NeurIPS. 2019.
[11] Frankle J, Carbin M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In: International conference on learning representations. 2019, URL https://openreview.net/forum?id=rJl-b3RcF7.
[12] Han S, Pool J, Tran J, Dally W. Learning both weights and connections for efficient neural network. In: Cortes C, Lawrence N, Lee D, Sugiyama M, Garnett R, editors. Advances in neural information processing systems, Vol. 28. Curran Associates, Inc.; 2015, URL https://proceedings.neurips.cc/paper/2015/file/ae0eb3eed39d2bcef4622b2499a05fe6-Paper.pdf.
[13] Liu Z, Sun M, Zhou T, Huang G, Darrell T. Rethinking the value of network pruning. In: International conference on learning representations. 2019, URL https://openreview.net/forum?id=rJlnB3C5Ym.
[14] Molchanov D, Ashukha A, Vetrov D. Variational dropout sparsifies deep neural networks. In: Precup D, Teh YW, editors. Proceedings of the 34th international conference on machine learning. Proceedings of machine learning research, vol. 70, PMLR; 2017, p. 2498–507, URL https://proceedings.mlr.press/v70/molchanov17a.html.
[15] Tartaglione E, Lepsøy S, Fiandrotti A, Francini G. Learning sparse neural networks via sensitivity-driven regularization. In: Proceedings of the 32nd international conference on neural information processing systems. 2018. p. 3882–92.
[16] Ullrich K, Meeds E, Welling M. Soft weight-sharing for neural network compression. In: ICLR (Poster). OpenReview.net; 2017.
[17] Louizos C, Welling M, Kingma DP. Learning sparse neural networks through L_0 regularization. In: International conference on learning representations. 2018, URL https://openreview.net/forum?id=H1Y8hhg0b.
[18] Wen W, Wu C, Wang Y, Chen Y, Li H. Learning structured sparsity in deep neural networks. In: Advances in neural information processing Systems. 2016, p. 2074–82.
[19] Lu L, Xie J, Huang R, Zhang J, Lin W, Liang Y. An efficient hardware accelerator for sparse convolutional neural networks on FPGAs. In: 2019 IEEE 27th annual international symposium on field-programmable custom computing machines. IEEE; 2019, p. 17–25. http://dx.doi.org/10.1109/FCCM.2019.00013.
[20] PyTorch. Pytorch sparse tensor. 2021, URL https://pytorch.org/docs/stable/sparse.html.
[21] Zhou X, Du Z, Zhang S, Zhang L, Lan H, Liu S, et al. Addressing sparsity in deep neural networks. IEEE Trans Comput-Aided Des Integr Circuits Syst 2019;38(10):1858–71. http://dx.doi.org/10.1109/TCAD.2018.2864289.
[22] Zhu C, Huang K, Yang S, Zhu Z, Zhang H, Shen H. An efficient hardware accelerator for structured sparse convolutional neural networks on FPGAs. IEEE Trans Very Large Scale Integr (VLSI) Syst 2020;28(9):1953–65. http://dx.doi.org/10.1109/TVLSI.2020.3002779.
[23] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. In: ICLR 2015 : International conference on learning representations 2015. 2015.
[24] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: 2016 IEEE conference on computer vision and pattern recognition. 2016, p. 770–8. http://dx.doi.org/10.1109/CVPR.2016.90.
[25] Huang G, Liu Z, Van Der Maaten L, Weinberger KQ. Densely connected convolutional networks. In: 2017 IEEE conference on computer vision and pattern recognition. 2017, p. 2261–9. http://dx.doi.org/10.1109/CVPR.2017.243.
[26] Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, et al. Imagenet large scale visual recognition challenge. Int J Comput Vis 2015;115(3):211–52. http://dx.doi.org/10.1007/s11263-015-0816-y.
[27] Krizhevsky A. One weird trick for parallelizing convolutional neural networks. 2014, arXiv preprint arXiv:1404.5997.
[28] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, et al. Going deeper with convolutions. In: 2015 IEEE conference on computer vision and pattern recognition. 2015, p. 1–9. http://dx.doi.org/10.1109/CVPR.2015.7298594.
[29] Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z. Rethinking the inception architecture for computer vision. In: 2016 IEEE conference on computer vision and pattern recognition. 2016, p. 2818–26. http://dx.doi.org/10.1109/CVPR.2016.308.
[30] Tan M, Chen B, Pang R, Vasudevan V, Sandler M, Howard A, et al. Mnasnet: Platform-aware neural architecture search for mobile. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019, p. 2820–8. http://dx.doi.org/10.1109/CVPR.2019.00293.
[31] Howard A, Sandler M, Chen B, Wang W, Chen L-C, Tan M, et al. Searching for MobileNetV3. In: 2019 IEEE/CVF international conference on computer vision. 2019, p. 1314–24. http://dx.doi.org/10.1109/ICCV.2019.00140.
[32] Xie S, Girshick R, Dollár P, Tu Z, He K. Aggregated residual transformations for deep neural networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2017, p. 1492–500. http://dx.doi.org/10.1109/CVPR.2017.634.
[33] Ma N, Zhang X, Zheng H-T, Sun J. Shufflenet V2: Practical guidelines for efficient CNN architecture design. In: Ferrari V, Hebert M, Sminchisescu C, Weiss Y, editors. Computer vision – ECCV 2018. Cham: Springer International Publishing; 2018, p. 122–38. http://dx.doi.org/10.1007/978-3-030-01264-9_8.
[34] Iandola FN, Han S, Moskewicz MW, Ashraf K, Dally WJ, Keutzer K. SqueezeNet: AlexNet-Level accuracy with 50x fewer parameters and < 0.5 MB model size. 2016, arXiv preprint arXiv:1602.07360.
[35] Zagoruyko S, Komodakis N. Wide residual networks. In: Wilson RC, Hancock ER, Smith WAP, editors. Proceedings of the British machine vision conference. BMVA Press; 2016, p. 87.1–87.12. http://dx.doi.org/10.5244/C.30.87.
[36] Bragagnolo A, Tartaglione E, Fiandrotti A, Grangetto M. On the role of structured pruning for neural network compression. In: 2021 IEEE International conference on image processing. 2021, p. 3527–31. http://dx.doi.org/10.1109/ICIP42928.2021.9506708.
[37] Tartaglione E, Nuzzarello G, Bragagnolo A, Grangetto M. Structured sparsity on embedded devices.