Simplify: a Python library for optimizing pruned neural networks

Andrea Bragagnolo\textsuperscript{a,b,*}, Carlo Alberto Barbano\textsuperscript{a}

\textsuperscript{a}Computer Science Department, University of Turin, Corso Svizzera 185, Turin, 10149, Italy
\textsuperscript{b}Synesthesia s.r.l., Corso Dante 118, Turin, 10126, Italy

\begin{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 actual advantages provided by sparsification methods. We propose \textit{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.
\end{abstract}

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 [3, 6, 18, 19, 22].

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 FLOPs or inference speed [1], this does not always reflect the actual achievable performance gain and it is usually overestimated.

To solve this issue, we propose \textit{Simplify}, a PyTorch [14] 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, \textit{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 Figure 1.

In the related literature it is possible to encounter two class of pruning procedures: \textit{unstructured} and \textit{structured}. Unstructured pruning approaches remove single parameters from the network, independently from one another [2, 4, 9, 13, 20, 21]. When employing this kind of techniques, one can obtain high degree of sparsity, but the pruning of entire neurons is not a guarantee. 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 \[10, 19, 23\]. 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 \[11, 15, 25, 26\]. 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 \[16\], 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 \[5\], DenseNet \[7\] 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 know

\(^2\)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
Table 1

| Architecture       | Dense   | Inference Time (ms) | Simplified |
|--------------------|---------|---------------------|------------|
|                    | Inference Time | Pruned          |            |
| AlexNet [8]        | 13.8±5.0 | 12.0±0.4           | 8.0±2.0   |
| VGG-19 [16]        | 96.5±3.8 | 87.3±1.6           | 34.4±0.6  |
| ResNet-50 [5]      | 55.2±1.2 | 54.3±0.4           | 34.1±0.4  |
| WideResNet-101 [24]| 95.9±0.1 | 95.8±0.6           | 53.9±0.4  |
| InceptionV3 [17]   | 45.0±1.0 | 42.6±0.4           | 30.4±0.3  |
| ShuffleNetV2 [12]  | 20.9±0.3 | 20.5±0.5           | 15.0±0.2  |

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 Sec. 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 Sec. 1, namely optimization for model deployment and optimization during training.

**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.

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

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

Tab. 1 shows the inference times (in milliseconds) of different standard PyTorch dense models, the resulting pruned models (random, structured pruning with 50% probability) and the simplified model obtained with our proposed library. The benchmarks are run on an 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’s 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. Additional results for all the torchvision architectures can be found in the repository README file.
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 Sec. C.1. Below, we provide a sample code snippet.

```python
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 shows the percentage of time reduction for the forward and backward step (i.e. a training iteration) for some simplified PyTorch models with respect to the amount of pruned neurons.

4. Impact

Current SOTA 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.
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.

6. Conflict of Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

CRediT authorship contribution statement

Andrea Bragagnolo: Conceptualization of this study, Methodology, Software. Carlo Alberto Barbano: Conceptualization of this study, Methodology, Software.

References

[1] Blalock, D., Ortiz, J.J.G., Frankle, J., Guttag, J., 2020. What is the state of neural network pruning? arXiv preprint arXiv:2003.03033.
[2] Frankle, J., Carbin, M., 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks. 7th International Conference on Learning Representations, ICLR 2019.
[3] Gale, T., Elsen, E., Hooker, S., 2019. The state of sparsity in deep neural networks. arXiv preprint arXiv:1902.09574.
[4] Han, S., Pool, J., Tran, J., Dally, W., 2015. Learning both weights and connections for efficient neural network, in: Advances in neural information processing systems, pp. 1135–1143.
[5] He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778.
[6] He, Y., Liu, P., Wang, Z., Hu, Z., Yang, Y., 2019. Filter pruning via geometric median for deep convolutional neural networks acceleration, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4340–4349.
[7] Huang, G., Liu, Z., Pleiss, G., van der Maaten, L., Weinberger, K., 2019. Convolutional networks with dense connectivity. IEEE Transactions on Pattern Analysis and Machine Intelligence PP, 1–1. doi:10.1109/TPAMI.2019.2918284.
[8] Krizhevsky, A., 2014. One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997.
[9] Liu, Z., Sun, M., Zhou, T., Huang, G., Darrell, T., 2018. Rethinking the value of network pruning. arXiv preprint arXiv:1810.05270.
[10] Louizos, C., Welling, M., Kingma, D.P., 2018. Learning sparse neural networks through variational dropout sparsifies deep neural networks, in: 34th International Conference on Machine Learning, ICML2017, pp. 3854–3863. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85048506601&partnerID=40&md5=dc00c36113f775ff4a6978b86543814d. cited By 29.
[11] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. Pytorch: An imperative style, high-performance deep learning library, in: NeurIPS.
[12] PyTorch, 2021. PyTorch sparse tensor. URL: https://pytorch.org/docs/stable/sparse.html.
[13] Simonany, K., Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.
[14] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z., 2016. Rethinking the inception architecture for computer vision, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2818–2826.
[15] Tang, Y., Wang, Y., Xu, Y., Tao, D., Xu, C., Xu, C., Xu, C., 2020. Scop: Scientific control for reliable neural network pruning. arXiv preprint arXiv:2010.10732.
[16] Tartaglione, E., Bragagnolo, A., Odiera, F., Fiandrotti, A., Grangetto, M., 2021. Serene: Sensitivity-based regularization of neurons for structured sparsity in neural networks. IEEE Transactions on Neural Networks and Learning Systems, 1–14doi:10.1109/TNNLS.2021.3084527.
[17] Tartaglione, E., Lepsy, S., Fiandrotti, A., Francini, G., 2018. Learning sparse neural networks via sensitivity-driven regularization, in: Advances in neural information processing systems, pp. 3878–3888.
[18] Ulrich, K., Welling, M., Meeds, E., 2019. Soft weight-sharing for neural network compression, in: 5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85071003624&partnerID=40&md5=dc00c36113f775ff4a6978b86543814d. cited By 2.
[22] Wang, Y., Zhang, X., Xie, L., Zhou, J., Su, H., Zhang, B., Hu, X., 2020. Pruning from scratch., in: AAAI, pp. 12273–12280.
[23] Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H., 2016. Learning structured sparsity in deep neural networks, in: Advances in Neural Information Processing Systems, pp. 2074–2082.
[24] Zagoruyko, S., Komodakis, N., 2016. Wide residual networks. arXiv preprint arXiv:1605.07146.
[25] Zhou, X., Du, Z., Zhang, S., Zhang, L., Lan, H., Liu, S., Li, L., Guo, Q., Chen, T., Chen, Y., 2019. Addressing sparsity in deep neural networks. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38, 1858–1871. doi:10.1109/TCAD.2018.2864289.
[26] Zhu, C., Huang, K., Yang, S., Zhu, Z., Zhang, H., Shen, H., 2020. An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 28, 1953–1965.
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:

\[
y = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta
\]

where \( \gamma \) and \( \beta \) represent, respectively, the weights and bias of the layer and are learned using standard backpropagation procedures; \( \mu \) and \( \sigma^2 \) 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 \( \hat{\mu} \) and \( \hat{\sigma}^2 \). Notice that each parameter is defined for each channel of the input feature map; we will denote them as \( \gamma_c, \beta_c, \hat{\mu}_c \) and \( \hat{\sigma}^2_c \) 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

\[
y = \gamma_c \frac{x - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} + \beta
\]

this can be rewritten as

\[
y = \frac{\gamma}{\sqrt{\hat{\sigma}^2 + \epsilon}} x - \frac{\gamma}{\sqrt{\hat{\sigma}^2 + \epsilon}} \hat{\mu} + \beta
\]

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

\[
x = W \cdot z + b
\]

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.

\[
y = \frac{\gamma}{\sqrt{\hat{\sigma}^2 + \epsilon}} (W \cdot z + b) - \frac{\gamma}{\sqrt{\hat{\sigma}^2 + \epsilon}} \hat{\mu} + \beta
\]

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

\[
W_{\text{fuse}} = \frac{\gamma W}{\sqrt{\hat{\sigma}^2 + \epsilon}}
\]

\[
b_{\text{fuse}} = \gamma \frac{b - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} + \beta
\]

and the output \( y \) is therefore

\[
y = W_{\text{fuse}} \cdot z + b_{\text{fuse}}
\]

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:

$$y = f(xA^T + a)$$

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

$$z = yB^T + b$$

Focusing on Equation 9, we can visualize the vector-matrix product:

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

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

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

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 Equation 10 focusing on $B_0$ we obtain:

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

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 Equation 10 focusing on $B_0$ we obtain:

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

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 $\hat{b}$ for the layer can be written as:

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

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

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

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 $\hat{L}_1 = \langle A, \hat{a} \rangle$ and $\hat{L}_2 = \langle B, \hat{b} \rangle$ 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 Section B.1, let us consider two sequential convolutional layers \( L_1 = (A, a) \) and \( L_2 = (B, b) \). 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 channel (\( B \) 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:

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

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

\[
F^i + a_i = \begin{bmatrix} F^i_{0,0} & \cdots & F^i_{0,H_{out}} \\ \vdots & \ddots & \vdots \\ F^i_{W_{out},0} & \cdots & F^i_{W_{out},H_{out}} \end{bmatrix} + \begin{bmatrix} a_0 \\ \vdots \\ a_0 \end{bmatrix}
\]

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:

\[
F^1 + a_1 = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} + \begin{bmatrix} a_1 & \cdots & a_1 \\ \vdots & \ddots & \vdots \\ a_1 & \cdots & a_1 \end{bmatrix} = \begin{bmatrix} a_1 & \cdots & a_1 \\ \vdots & \ddots & \vdots \\ a_1 & \cdots & a_1 \end{bmatrix}
\]

Thus, \( y \) becomes:

\[
\begin{bmatrix} F^0 + a_0 \\ \frac{F^1}{a_1} \end{bmatrix}^T
\]

where \( \frac{\cdot}{\cdot} \) 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 (Section B.1). The forward pass of \( L_2 \) can be expressed as follows:

\[
z = f(F^0 + a_0) \ast B_{0,0} + f(\frac{a_1}{a_1}) \ast B_{0,1} + b_0
\]

(13)

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

\[
\hat{b}_0 = \begin{bmatrix} f(a_1) \\ f(a_1) \\ \frac{f(a_1)}{a_1} \end{bmatrix} \ast \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + b_0 = \begin{bmatrix} 4f(a_1) \\ 4f(a_1) \end{bmatrix} + b_0
\]

(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 \( \hat{b}_0 = 4f(a_1) + b_0 \) which will be used from now on in \( L_2 \).

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 Equation 14. The general rule to obtain the new bias vector \( \hat{b} \) can be expressed as follows:

\[\text{\footnotesize Notice that this can be applied for every choice of values for } B. \text{ Of course the resulting bias factor will change accordingly}\]
Bragagnolo and Barbano: Preprint submitted to SoftwareX

\[
\hat{b} = \begin{bmatrix}
  b_0 + \sum_i f(a_i) \ast B_{0,i} \\
  b_1 + \sum_i f(a_i) \ast B_{1,i} \\
  \vdots \\
  b_{C_{\text{out}}} + \sum_i f(a_i) \ast B_{C_{\text{out}},i}
\end{bmatrix}
\]

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

\[
\hat{b}_0 = \begin{bmatrix}
  0 & 0 & 0 & 0 & 0 \\
  0 & a' & a' & a' & 0 \\
  0 & a' & a' & a' & 0 \\
  0 & 0 & a' & a' & 0 \\
  0 & 0 & 0 & 0 & 0
\end{bmatrix} \ast \begin{bmatrix}
  1 \\
  1
\end{bmatrix} + b_0 = \begin{bmatrix}
  a' + b_0 \\
  2a' + b_0 \\
  2a' + b_0 \\
  a' + b_0 \\
  a' + b_0
\end{bmatrix} + b_0
\]

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

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

\[
\hat{b}_0 = \begin{bmatrix}
  a' + b_0 & 2a' + b_0 & 2a' + b_0 & a' + b_0 \\
  2a' + b_0 & 4a' + b_0 & 4a' + b_0 & 2a' + b_0 \\
  2a' + b_0 & 4a' + b_0 & 4a' + b_0 & 2a' + b_0 \\
  a' + b_0 & 2a' + b_0 & 2a' + b_0 & a' + b_0
\end{bmatrix}
\]

To obtain the updated biases in case of multiple neurons and multiple channels, the same rule of Equation 15 can be applied, keeping in mind that in this case it will result into a tensor of shape \( C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}} \) 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 \( L_3 = \langle B, b \rangle \):

\[
y + t = \begin{bmatrix}
  xA_T^0 + a_0 \\
  0 + a_1 \\
  \vdots \\
  xA_T^{N-1} + a_{N-1}
\end{bmatrix} + \begin{bmatrix}
  0 + c_0 \\
  \hat{x}C_{1}^T + c_1 \\
  \vdots \\
  \hat{x}C_{N}^T + c_N
\end{bmatrix}
\]

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 Equation 17 as Equation 12 and obtain:

\[
z_0 = f(xA_T^0 + a_0 + c_0)B_{0,0} + f(a_1 + \hat{x}C_{1}^T + c_1)B_{0,1} + \cdots + \\
+ f(a_{N-1} + c_{N-1})B_{0,N-1} + \\
+ f(xA_T^N + a_N + \hat{x}C_{N}^T + c_N)B_{0,N} + b_0
\]

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 (Sec. C.1) to achieve a speed-up in the convolution operations anyways.
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):

   \[
   \tilde{W}^L = \begin{bmatrix}
   W^L_{0,i}, W^L_{1,i}, \ldots, W^L_{N,i}
   \end{bmatrix}
   \]  
   (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:

   \[
   \tilde{W}^L = \begin{bmatrix}
   \tilde{W}^L_j
   \end{bmatrix}_{j \in J}
   \]  
   (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.

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 the Fig. 3. After the expansion step, we sum the layers biases (which were not
propagated as explained in Sec. 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.

Figure 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).