REPRODUCIBILITY CHALLENGE - NEURON MERGING: COMPENSATING FOR PRUNED NEURONS

Bence Cserkuti, Olaf Lipinski, Matthew Pugh, Kian Spencer

ABSTRACT

We have looked at the paper proposing a new technique for reducing the number of parameters in neural networks called Neuron Merging (Kim et al., 2020). We have performed an in-depth analysis of the mathematics used in the paper and the code that the authors have provided. We also propose possible new avenues of exploration and extension to this technique. Our code is available on GitHub [1].

1 INTRODUCTION

Pruning neural networks is a common practice to remove parts of a model which have low salience. Ideally, the pruned network can achieve comparable accuracy to the original network with significantly fewer parameters. While pruning techniques have been published in the 1980s (LeCun et al., 1990), the need for compact networks has been driven by mobile phones and IoT devices recently.

The general method to prune a network is the following: first a model is trained to convergence. Then a pruning algorithm determines which parameters to cut based on a metric such as the magnitude of the weights. There are two main pruning approaches: in unstructured pruning individual weights are removed, while in structured pruning neurons or filters are discarded (Blalock et al., 2020). While the former method leads to smaller networks, the pruned model is often sparse and speedups might only be possible using specialised hardware (Blalock et al., 2020).

Our selected paper by (Kim et al., 2020) presents a structured pruning algorithm where the information of the pruned neurons is preserved by merging a scaled version into the layer following the pruned neurons. This method offers a data-free way to merge neurons and has been shown to produce more accurate results without fine-tuning than pruned networks.

2 MATHEMATICAL ANALYSIS

The approach is to maintain the activation vector of the $(i+1)$-th layer, $\mathbf{a}_{i+1} = \mathbf{W}_{i+1}^\top f(\mathbf{W}_i^\top \mathbf{x}_i)$ where $\mathbf{W}_i \in \mathbb{R}^{N_i \times N_{i+1}}$ are the weights of the i-th layer, $\mathbf{x}_i \in \mathbb{R}^{N_i}$ is the input, and $\mathbf{x}_{i+1} \in \mathbb{R}^{N_{i+1}}$ is the output. Decompose $\mathbf{W}_i$ as $\mathbf{W}_i \approx \mathbf{Y}_i \mathbf{Z}_i$ such that $\mathbf{Y}_i \in \mathbb{R}^{N_i \times P_{i+1}}$ and $\mathbf{Z}_i \in \mathbb{R}^{P_{i+1} \times N_{i+1}}$ for $0 < P_{i+1} \leq N_{i+1}$. Such a decomposition exists by rank factorization, and $\mathbf{Z}_i$ shoud be a sparse non-negative matrix with one strictly positive entry per column (condition (1)). The activation function f should be ReLU so that $f(\mathbf{Z}_i^\top) = \mathbf{Z}_i^\top$ and $f(\mathbf{Z}_i^\top \mathbf{Y}_i^\top) = \mathbf{Z}_i^\top f(\mathbf{Y}_i^\top)$, and for $a, b \in \mathbb{R}$, $a > 0$, we have $f(ab) = af(b)$. Hence,

$$\mathbf{a}_{i+1} \approx \mathbf{W}_{i+1}^\top f(\mathbf{Z}_i^\top \mathbf{Y}_i^\top \mathbf{x}_i) = \underbrace{\mathbf{W}_{i+1}^\top \mathbf{Z}_i^\top f}_{\text{ReLU}+(1)}(\mathbf{Y}_i^\top \mathbf{x}_i) = (\mathbf{Z}_i \mathbf{W}_{i+1})^\top f(\mathbf{Y}_i^\top \mathbf{x}_i)$$

The key insight of neuron merging is the final equality: $\mathbf{Z}_i$ is a scaling matrix for $\mathbf{W}_{i+1}$ and $\mathbf{Y}_i$ is the new i-th layer weight matrix.

We want to maintain the activation feature map of the $(i+1)$-th layer, $\mathcal{A}_{i+1} = \mathcal{W}_{i+1} \circledast f(\mathcal{W}_i \circledast \mathcal{X}_i)$ where $\mathcal{X}_i \in \mathbb{R}^{N_i \times H_i \times W_i}$ and $\mathcal{W}_i \in \mathbb{R}^{N_{i+1} \times N_i \times K \times K}$ is the filter weights of the i-th layer with N_{i+1} filters. Decompose $\mathcal{W}_i$ as $\mathcal{W}_i \approx \mathcal{Y}_i \times_1 \mathbf{Z}_i^\top$ such that $\mathcal{Y}_i \in \mathbb{R}^{P_{i+1} \times N_i \times K \times K}$ and $\mathbf{Z}_i \in \mathbb{R}^{P_{i+1} \times N_{i+1}}$.

[1] https://github.com/COMP6248-Reproducability-Challenge/neuron-merging/

The 1-mode product $\mathcal{Y}_i \times_1 \mathbf{Z}_i^\top$ can be expressed as $\mathbf{W}_{i(1)} \approx \mathbf{Z}_i^\top \mathbf{Y}_{i(1)}$ where $\mathbf{Y}_{i(1)}$ and $\mathbf{W}_{i(1)}$ are tensor unfoldings of $\mathcal{Y}_i$ and $\mathcal{W}_i$ with respect to the column fibres (Kolda & Bader, 2009). Then,

$$\mathcal{A}_{i+1} \approx \mathcal{W}_{i+1} \circledast f((\mathcal{Y}_i \circledast \mathcal{X}_i) \times_1 \mathbf{Z}_i^\top) = \mathcal{W}_{i+1} \circledast \underbrace{(f(\mathcal{Y}_i \circledast \mathcal{X}_i) \times_1 \mathbf{Z}_i^\top)}_{\text{ReLU}+(1)} = (\mathcal{W}_{i+1} \times_2 \mathbf{Z}_i) \circledast f(\mathcal{Y}_i \circledast \mathcal{X}_i)$$

where the 2-mode product $\mathcal{W}_{i+1} \times_2 \mathbf{Z}_i$ can be expressed in terms of tensor unfoldings with respect to the row fibres. Full proofs with the appended bias can be found in the appendix of the paper. For the definition of Tensor-wise convolution $\circledast$, see the paper.

Complexity. In general, tensor decomposition is NP-hard (Hillar & Lim, 2013). However, cosine similarity has low complexity, and $\mathbf{Z}_i$ is necessarily sparse. Therefore we expect the proposed algorithm to have polynomial time complexity. The reviews suggest comparing against Non-negative Matrix Factorisation type algorithms, however, this is not possible due to negative entries in $\mathbf{W}_i$.

Advantages. Generality: the core ideas of the paper show promise for going beyond ReLU, the paper demonstrates that pruning is a special case of merging. There is no need for retraining or fine-tuning. The proposed algorithm is fast and inexpensive as it does not retain decomposed tensors.

Disadvantages. We are limited by ReLU and condition (1). The models tested did not merge both the fully connected layer and the convolution layer within the same model.

3 Reproducibility

As the authors made their code publicly available, first we confirmed that the results quoted in their paper match the output of their code. Then the reproducibility of the paper was tested in two ways: by re-implementing algorithms 1-3 and by evaluating their approach on AlexNet, a network which is not supported by their code.

3.1 Re-implementing the Algorithms

Overall the notation in the paper has been clear, and it was fairly straightforward to re-implement the three algorithms. Using assert statements we confirmed that the tensors and evaluation accuracies returned by the original implementation and our re-implementation were identical. However, the performance of the re-implemented algorithms is drastically inferior. Algorithms 1-2 (for fully connected layers) ran 346 times slower on average, while algorithms 1-3 (for convolutional layers) ran 18 times slower. This is due to the fact that we strictly followed the description of the algorithms without optimising it. For example, the cosine similarity in algorithms 2-3 is calculated between $\mathbf{w}_n$ and every $\mathbf{w}$ in $\mathbf{Y}_i$, for every $\mathbf{w}_n$. In the original code this was optimised with a matrix operation (`pairwise_distances()` from `sklearn.metrics`). One inconsistency we discovered was the definition of s, which is the l_2-norm ratio of two convolutional feature maps $\mathcal{X}_{1,:,:}$ and $\mathcal{X}_{2,:,:}$. The authors' notation and description is not consistent and it is unclear whether $\mathcal{X}_{1,:,:}$ or $\mathcal{X}_{2,:,:}$ should be in the denominator. Using their code, we confirmed that the correct l_2-norm ratio is $\frac{x_1}{x_2}$. When the ratio was reversed, the accuracy was 4% lower on average.

Although the re-implementation of the proposed algorithms produced the same results as the original code, implementing the paper from scratch would have been really difficult. This is explained by how neural networks are saved in PyTorch. Developers have their own conventions when naming and structuring layers, which makes it harder to generalise the decomposition of the weight matrices. Accordingly, the authors had to create a custom implementation for each network which merged the neurons on a layer by layer basis using the internal reference names of the layers of the exported models. This makes it less straightforward to use neuron merging on new architectures.

3.2 AlexNet Neuron Merging Evaluation

We performed a test to see how reproducible neuron merging results would be on architectures other than those used by the authors. To run this test we chose AlexNet, because of its relative simplicity when compared to more complex networks such as Inception V3.

We also implemented an AlexNet variant to work with the CIFAR100 dataset. As seen in Fig. 1 the new network performs well with merging ratio up to 0.8. At that level of pruning we are left with only 19% of the original number of parameters, while still achieving only 8% lower accuracy. The accuracy drop is considerably higher above 0.8 pruning ratio. We can see that for 0.9 ratio the accuracy drops by more than half, compared to the original network. At very high pruning ratios, the performance between the approaches diverges greatly.

We include the results for the ”vanilla” AlexNet that was trained on ImageNET in Fig. 1. Interestingly, merging achieves a lower accuracy for this network, which could be considered counter-intuitive. The accuracies only diverge slightly, and then converge exactly at 0.95 pruning ratio. This result could signify that merging and compensating for all neurons is not a surefire way to increase accuracy and decrease parameters. Therefore, the neuron merging technique must be applied with rigorous testing of network accuracy vs merging parameters.

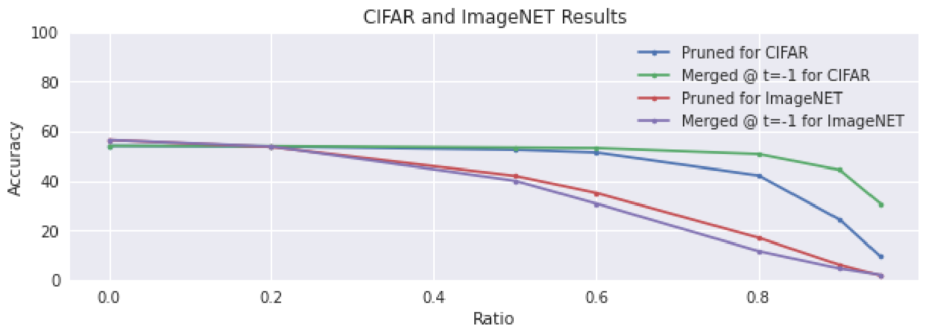

Figure 1: Pruning and Merging Accuracy for AlexNet on the CIFAR100 and ImageNET datasets

4 The Next Step: Generalisation

The current methodology lacks flexibility in two key areas: Model Architectures, and Activation Functions. These two issues dominate the remarks made by reviewing academics. This section outlines the progress made in the next stage of re-implementation; extension. The new techniques address the dominant issues, and potentially increase the chance that other researchers may incorporate these techniques into future projects.

4.1 Model Architecture

To successfully merge layers within a pre-trained model, the architecture must be known. As discussed in Section 3.1, this necessitates hard coded cases for each supported model, preventing generalisation, and requiring future researchers to re-implement the neuron merging code-base for each new model.

Considering user experience, in the optimal case: neuron merging would be imported, the model instance would be passed to the neuron merge function, and the function would return the merged model. The standard torch library provides the method `modules` as part of the `torch.nn.Module` class. This method recursively polls the children of the class instance which inherit from `torch.nn.Module` in order to produce a generator object of the model layers (e.g. `torch.nn.ReLU`, or `torch.nn.Linear`). However, it confers no information about the implementation of the forward method of the model, or even the order of the model layers. In order to overcome this, a hook function may be temporarily applied to the given model. Passing a dummy input of the correct size through the model allows the hook function to return an `OrderedDict` containing information about each layer: such as the layer object instance, layer type, input size, and output size. For the purposes of this report, the implementation utilised sections of code from the python library `pytorch-summary` from Chandel (2021). With information about the structure and layer types of the model, sections of the implementation which previously required numerous `if` statements and hard-coded model handling can be achieved automatically and generally.

This new methodology alters only the model handling while retaining the original layer merging algorithms, leaving the pruning performance unaffected. Furthermore, these modifications allow the neuron merging algorithm to be applied naively, if the given model meets the following conditions: There are no branches in the forward method; it is implemented using `torch.nn.Module`; And the model only utilises layers of the types `nn.Identity`, `nn.ReLU`, `nn.Linear`, `nn.Conv2d`, `nn.MaxPool2d`, `nn.Dropout`. Removing the necessity to understand the specific implementation, introduces the potential for open source release with relatively minor, user-comfort, modifications. Greater accessibility to academics may not be directly novel itself, but it distinctly enhances impact of neuron merging.

4.2 Activation Functions

Though `ReLU` is an exceptionally popular activation function, this and condition (1) prevents the neuron merging algorithm being implemented on model layers with other activations. Though Theorem 1 was proven directly, considering why it works may allow for generalisation.

Activation functions are commonly defined on $\mathbb{R}$, but are extended to $\mathbb{R}^n$ through element-wise application $T_i(\vec{v}) = T(\vec{v}_i)$. The original theorem arises due to the symmetries of the `ReLU` transfer function. For the purposes of this analysis we define a symmetry on the extended activation function as a tuple of functions, $S = [S_i, S_o]$, for which $T(\vec{v}) = S_o T(S_i \vec{v})$. Because a densely connected layer is composed of two linear maps (the weight matrices) and the activation function, to find the relevant symmetries we constrain S_i and S_o to be linear maps, or matrices of arbitrary shape with bias values. By inspection it is clear the uni-variate `ReLU` function has symmetries of the form $S = [a, a^{-1}],\ a \in \mathbb{R}^+$ as multiplying the input by $\alpha \in \mathbb{R}^+$ leaves the output of `ReLU` the same. In fact, these symmetries would apply to any uni-variate piece-wise linear function of x, with domains $c > 0$ and $c \leq x$ for constant $c \in \mathbb{R}$, as the bias values allow us to shift our scaled input such that it stays in the same domain. In the extended case these symmetries can be composed in order to find higher dimensional symmetries. To find an input symmetry of $v \in \mathbb{R}^n$, the uni-variate symmetries may be combined into diagonal matrices $S = [diag([a_0, ..., a_n]),\ diag([a_0^{-1}, ..., a_n^{-1}])]$. As each element of $v \in \mathbb{R}^n$ is independent when transformed by the diagonal matrix, the lower dimensional symmetries still apply. The paper interprets Theorem 1 as conditions for the extended `ReLU` to act as the identity on a matrix. More generally, it may be better to consider it as an understanding of `ReLU` symmetries applied to modify the matrices inside and outside the function. In this way, with little extra work, it becomes apparent that neuron merging may be extended to other transfer functions, such as leaky `ReLU`. The extended version of this report uses these methods to find the linear symmetries of finite polynomial activation functions, demonstrating the potential for neuron merging to be applied to an approximation of any activation function.

5 Conclusion

In this review of neuron merging, we have presented an analysis of the methodology, reproduced and re-implemented the algorithms with matching results, performed testing on a new dataset, and suggested an idea for generalisation to a range of activations. Reviewers cited 3 main weakness of the paper: it is unclear how to extend beyond ReLU, limited datasets and benefit compared to vanilla pruning, and lack of comparison to NMF type algorithms. We have given a future direction for extending the activation, tested neuron merging on AlexNet, and highlighted why there is no comparison to NMF.

References

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

Shubham Chandel. sksq96/pytorch-summary, May 2021. URL `https://github.com/sksq96/pytorch-summary`. original-date: 2018-04-23T13:58:04Z.

Christopher J Hillar and Lek-Heng Lim. Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):1–39, 2013.

Woojeong Kim, Suhyun Kim, Mincheol Park, and Geonseok Jeon. Neuron merging: Compensating for pruned neurons, 2020. URL `https://github.com/friendshipkim/neuron-merging`.

Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky (ed.), *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1990.