Improving Pruning Filters for Efficient ConvNets

Cenk Alkan

April 2024

1 Introduction

This text begins with a summary of *Pruning Filters for Efficient ConvNets* [1], followed by a presentation of my improvements to the program described in that paper.

Pruning Filters for Efficient ConvNets [1] was presented at *ICLR 2017* by Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf.

2 Original Paper

2.1 Research Question

The research question posed in [1] is: How can the computational and parameter storage costs of convolutional neural networks (CNNs) be reduced while preserving model performance as these networks grow deeper and more complex? This question is critical because reducing computational costs is essential for many CNN applications, including mobile and cloud services.

2.2 Challenge Analysis

Previous research has also focused on reducing computational costs by compressing models through various methods. Some approaches prune individual weights by selectively removing those that contribute the least to the output, often based on criteria such as the smallest magnitude. While this method is effective at reducing the parameter count in fully connected layers, it does not necessarily lead to significant reductions in computational time for convolutional layers, which dominate the computation in CNNs. Furthermore, this approach creates sparse matrices that require specialized computing libraries to process efficiently, and such libraries may not be universally available or fully optimized.

2.3 Philosophy

The philosophy of the paper was to implement pruning at the level of entire filters rather than individual weights. This approach aimed to achieve two key

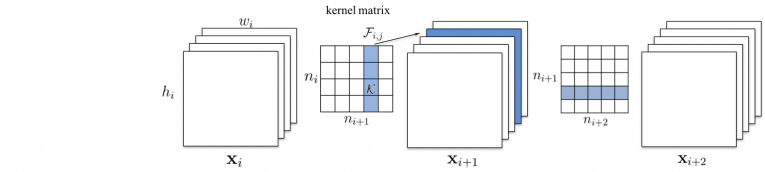


Figure 1: Filter pruning removes kernels not just on the next layer, but also the layer after that one.

objectives: preserving existing connectivity patterns and significantly reducing computational costs. By maintaining the existing connectivity patterns, the method offered a simpler solution with the potential to better preserve the program's accuracy. Reducing computational costs was particularly desirable as it decreased the total running time of processes, enhancing both speed and efficiency.

2.4 Solution

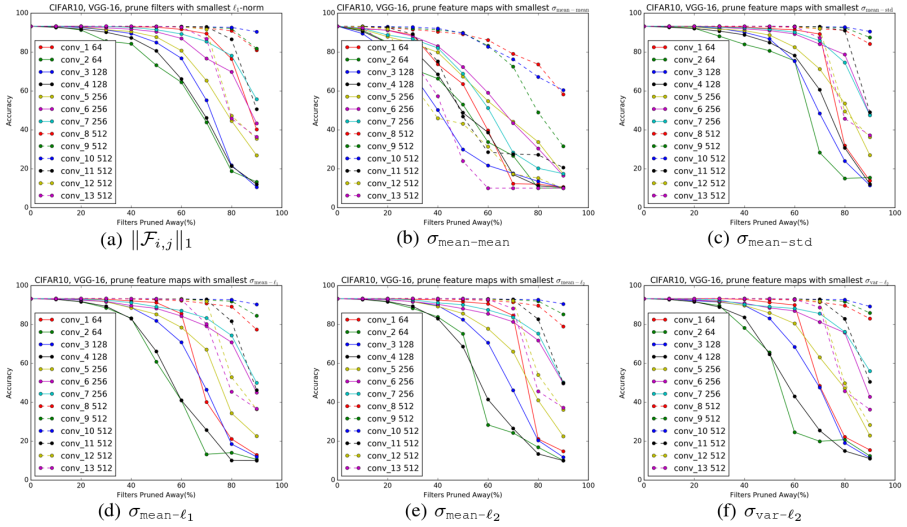
The solution to the challenge proposed above is to remove redundant filters, and retrain the model to recover from any accuracy loss caused by the removal of filters. The key idea is to select and prune filters which have a small effect on the output accuracy. The authors measure this effect of a filter in each layer by calculating the L1-norm. Filters are ranked based on their L1-norm, and this predetermined percentage of filters with the lowest L1-norm are pruned away. Layers with the same input sizes often have similar sensitivity to pruning, so the authors used the same pruning ratio for these layers. For layers that are sensitive to pruning, they used a small pruning rate or completely skipped pruning them.

When a filter $F_{i,j}$ is pruned, its corresponding feature map $x_{i+1,j}$ is removed, which reduces the number of operations needed to be made. The kernels that apply on the removed feature maps from the filters of the next convolutional layers are also removed, which saves additional operations. Pruning m filters of layer i will reduce the computation cost for both layers i and $i + 1$. This significantly reduces the computation cost for the model.

The authors then adopted a one-shot pruning and retraining strategy, which is to prune filters of multiple layers at once and retrain them. This ensures quick regaining of accuracy.

2.5 Experiments and Results

The authors conducted several experiments to evaluate the overall performance of their pruning method by applying it to various CNN architectures on different datasets. They also tested its effectiveness by comparing it with other pruning methods, such as random pruning or pruning based on activation feature maps. The figure above demonstrates that the L1 norm is indeed the most effective



(a) $\|\mathcal{F}_{i,j}\|_1$ (b) $\sigma_{\texttt{mean-mean}}$ (c) $\sigma_{\texttt{mean-std}}$

(d) $\sigma_{\texttt{mean-}\ell_1}$ (e) $\sigma_{\texttt{mean-}\ell_2}$ (f) $\sigma_{\texttt{var-}\ell_2}$

Figure 2: Comparison of activation-based feature map pruning

indicator for determining which filters to prune.

FLOP stands for floating-point operations, with fewer FLOPs translating to lower computational load and faster computation time. When applying VGG-16 to CIFAR-10, the authors pruned 50% of the filters in layer 1 and layers 8 to 13, achieving a 34% reduction in FLOPs while slightly improving accuracy. Additionally, they trained ResNet-56 and ResNet-110 using CIFAR-10. For ResNet-56, pruning resulted in a 27.6% reduction in FLOPs with minimal accuracy loss. For ResNet-110, a more aggressive pruning strategy achieved a 38.6% reduction in FLOPs. Finally, when training ResNet-34 on ImageNet, pruning led to a 24.2% reduction in FLOPs, maintaining competitive accuracy with only about a 1% performance loss.

The paper also includes a comparative analysis of pruning strategies, including randomly pruning filters or pruning the largest filters (by weight sum). Consistently, pruning the smallest filters outperformed random pruning and proved more effective than pruning the largest filters, which caused a significant drop in accuracy. They also compared their method with another pruning strategy that removes feature maps with low activation levels. The L1 norm pruning method generally provided better or comparable results compared to activation-based pruning, particularly at higher pruning ratios.

The experiments highlight the effectiveness of using L1 norm-based metrics to identify and remove redundant filters across various CNN architectures. The results demonstrate that this approach significantly reduces computational load and model size without degrading—and, in some cases, even improving—model accuracy.

Table 1: Overall results. The best test/validation accuracy during the retraining process is reported. Training a pruned model from scratch performs worse than retraining a pruned model, which may indicate the difficulty of training a network with a small capacity.

Model	Error(%)	FLOP	Pruned %	Parameters	Pruned %
VGG-16	6.75	3.13×10^8		1.5×10^7	
VGG-16-pruned-A	**6.60**	2.06×10^8	34.2%	5.4×10^6	64.0%
VGG-16-pruned-A scratch-train	6.88				
ResNet-56	6.96	1.25×10^8		8.5×10^5	
ResNet-56-pruned-A	6.90	1.12×10^8	10.4%	7.7×10^5	9.4%
ResNet-56-pruned-B	**6.94**	9.09×10^7	27.6%	7.3×10^5	13.7%
ResNet-56-pruned-B scratch-train	8.69				
ResNet-110	6.47	2.53×10^8		1.72×10^6	
ResNet-110-pruned-A	**6.45**	2.13×10^8	15.9%	1.68×10^6	2.3%
ResNet-110-pruned-B	6.70	1.55×10^8	38.6%	1.16×10^6	32.4%
ResNet-110-pruned-B scratch-train	7.06				
ResNet-34	26.77	3.64×10^9		2.16×10^7	
ResNet-34-pruned-A	27.44	3.08×10^9	15.5%	1.99×10^7	7.6%
ResNet-34-pruned-B	27.83	2.76×10^9	24.2%	1.93×10^7	10.8%
ResNet-34-pruned-C	27.52	3.37×10^9	7.5%	2.01×10^7	7.2%

Figure 3: Approximately 30 percent reduction in FLOPs for VGG-16 and ResNets without significant loss in accuracy.

3 Our Improvements

3.1 Philosophy

This implementation builds upon the *Pruning Filters for Efficient ConvNets* paper by introducing significant advancements that enhance efficiency and maintain accuracy. Traditional pruning techniques are static, pruning filters post-training to reduce redundancy without affecting performance. While effective, this approach requires extensive retraining and limits its adaptability to the evolving dynamics of training. Our improvements center on dynamically clustering filters during training based on feature similarities, enabling real-time identification and pruning of redundant filters. This proactive methodology ensures the network continuously refines its architecture as training progresses, minimizing computational overhead.

Additionally, our improvements incorporate multi-GPU training, which is uniquely suited to our dynamic approach. The clustering and pruning processes involve computationally intensive tasks such as k-means clustering and cosine similarity calculations. Multi-GPU support allows these tasks to be distributed across devices, accelerating the process and enabling efficient scaling to larger models and datasets. Together, these enhancements ensure that the network maintains or even improves its predictive capabilities while significantly reducing computational costs.

3.2 Solution

Our solution integrates dynamic online clustering and pruning directly into the training process, allowing for real-time architectural adjustments. By clustering filters during training using cosine similarity metrics and k-means clustering, the approach identifies and removes filters with minimal contributions to output accuracy. This strategy reduces computational load during training and eliminates the need for extensive post-training pruning and retraining, addressing a key limitation of traditional methods.

To implement this approach effectively, we optimized the training process with a learning rate schedule, starting at 0.02 and reducing it by a factor of five every 60 epochs. This schedule aligns with the evolving complexity of the network, ensuring stability as pruning adjusts the architecture. Data augmentation techniques, including random cropping and horizontal flipping, further enhance model generalization and robustness, while data normalization ensures consistent training inputs.

The methodology includes two key strategies for filter management: `update_clusters()` and `cluster_and_prune()`. The `update_clusters()` method dynamically groups filters based on functionality and similarity without altering the network's weights. By leveraging cosine similarity metrics and k-means clustering, this method is particularly effective during the early and middle phases of training when the network's patterns are still evolving. In contrast, the `cluster_and_prune()` method directly prunes filters at predetermined milestones. Once the network stabilizes and predictions plateau, this method refines the architecture by removing redundant filters, reducing complexity and enhancing efficiency.

By combining these strategies, we achieve a streamlined and adaptive pruning process that continuously optimizes the network's structure throughout training. The integration of multi-GPU training further accelerates these improvements, distributing the computational load of clustering and pruning across multiple devices to handle large-scale tasks effectively.

3.3 Experiments

I compared my algorithm with the algorithm presented in the original paper. To accomplish this, I conducted five trials of my algorithm using different numbers of clusters for the k-means algorithm. Specifically, I ran five trials with $k = 5$ followed by five trials with $k - 10$, continuing this pattern up to $k = 30$ Additionally, I performed five trials of the original algorithm as described in [1]. The comparison focused on both accuracy and inference time, examining top-1 and top-5 accuracy metrics.

Top-5 accuracy measured the percentage of cases where any of the model's five highest-probability predictions matched the target output, while top-1 accuracy evaluated the percentage of cases where the model's highest-probability prediction matched the target output.

All algorithms were run with a batch size of 128 and trained for 100 epochs. The experiments were conducted on the Brandeis HPCC, with each trial exe-

Trial	Top1 Accuracy	Top5 Accuracy	Inference Time (Seconds)
1	89.24	99.41	19.140482
2	89.44	99.48	19.794001
3	90.12	99.58	19.055294
4	88.27	99.39	18.983667
5	89.44	99.53	19.675458

Figure 4: The results for running the original algorithm presented in [1] for five trials.

cuted independently on compute nodes equipped with an NVIDIA TitanX GPU and an Intel Xeon X5670 CPU.

3.4 Results

I observed that the inference time for the original algorithm (see Figure 4) was consistently about five times larger than the inference time for my algorithm (see Figure 5). Additionally, the Top-1 and Top-5 accuracy appeared to be approximately the same for both the original algorithm and my version. These results held true across all cluster sizes.

To verify that these results were statistically significant, I first partitioned the results from my algorithm based on cluster size. Then, I ran a Kruskal-Wallis test on the partitions, including the trials from the original algorithm. The Kruskal-Wallis test was appropriate because each trial was independent, and the data distribution was unknown. I performed this analysis separately for inference time, Top-1 accuracy, and Top-5 accuracy, using an alpha value of 0.05.

For inference time, the Kruskal-Wallis test returned a p-value of approximately 0.033. This provided statistically significant evidence to suggest that at least one of the partitions, including the trials from the original algorithm, was different. To identify which were different, I conducted a post-hoc Dunn test with a Holm correction. The p-values for all comparisons between the original algorithm partition and the partitions based on cluster size were less than 0.02, demonstrating statistically significant evidence that my version of the algorithm ran faster for every cluster size.

For Top-1 accuracy, the Kruskal-Wallis test returned a p-value of approximately 0.19. For Top-5 accuracy, the p-value was about 0.87. These results provided statistically significant evidence to conclude that the performance was not significantly degraded.

From this analysis, I concluded that my version of the algorithm achieved an average of a fivefold speedup in model inference times while maintaining accuracy. This confirmed that integrating clustering and pruning into the training process effectively reduced inference times.

Trial	Top1 Accuracy	Top5 Accuracy	Inference Time (Seconds)	Number Clusters
1	88.53	99.45	4.741202	5
2	88.34	99.52	5.303724	5
3	88.71	99.53	5.060603	5
4	89.03	99.55	4.508155	5
5	88.76	99.61	4.585092	5
6	88.32	99.51	4.981002	10
7	88.05	99.57	4.942312	10
8	88.39	99.49	4.896819	10
9	89.12	99.65	4.574597	10
10	88.52	99.53	4.847188	10
11	88.97	99.57	4.731148	15
12	88.55	99.39	5.422887	15
13	88.95	99.58	4.752984	15
14	88.75	99.51	4.712664	15
15	88.58	99.64	4.753515	15
16	89.26	99.44	4.968990	20
17	88.67	99.56	4.616431	20
18	89.17	99.53	4.973336	20
19	87.98	99.60	5.659925	20
20	88.07	99.45	4.622275	20
21	88.50	99.50	4.789872	25
22	88.46	99.48	4.547283	25
23	88.81	99.66	4.895320	25
24	87.71	99.46	4.754891	25
25	88.49	99.64	4.744316	25
26	88.99	99.56	4.759562	30
27	88.70	99.46	4.845712	30
28	88.90	99.61	4.888407	30
29	87.96	99.40	4.540003	30
30	88.87	99.55	6.697844	30

Figure 5: The results for running our version of the algorithm with k clusters, where k is between 5 and 30.

4 Conclusions

I demonstrated that a modified version of the original algorithm could achieve significantly faster performance without sacrificing much accuracy. However, future work could address certain limitations in my experiments. Due to time constraints, I was only able to run each experiment for five trials. It is possible that different results might emerge with a larger number of trials. Additionally, I tested the algorithm solely on the CIFAR10 dataset and the VGG architecture, whereas the original paper evaluated its approach on multiple datasets and the ResNet architecture.

References

[1] Hao Li et al. *Pruning Filters for Efficient ConvNets*. 2017. arXiv: 1608.08710 [cs.CV].