# Model Compression and Acceleration Analysis on Image Classification Task

Mingxi Chen (mc7805), Xuefei Zhou (xz2643)

## • Project Description

Since computer vision models start getting deeper and more complicated for achieving better performance, huge amount of trainable parameters are introduced which will obviously require more resources. However, a model with higher test accuracy doesn't mean that it is more efficient and applicable.

Therefore, in this project, we aim to evaluate model compression methods and find the most suitable and simplest configuration for our application. We choose two structure-pruning based techniques and do some necessary adaptations for applying them to both VGG [3], a plain CNN architecture, and a more complicated network, ResNet [4], which has cross layer connections. The first approach sets different prune ratio for filters from each layer while the second one directly uses BN layers as scaling factors and sets global threshold to remove unimportant channels. We analyze the performance of each model in 8 aspects such as total parameters, run-time memory, inference time, throughput and so on. We also build an image classification application. It is deployed on Google Cloud Platform to test online performance of each model. Our project demonstrates the practicality of model compression methods with different network architectures and offline/online modes.

## • Approach

The first approach is proposed by Liu, Zhuang, et al [1] in 2017. Channel level pruning means pruning all the incoming and outgoing connections associated with a channel. Their basic idea is to introduce a scaling factor $\gamma$ for each channel and jointly train network weights with scaling factors. Then the training objective of this approach becomes to Eq.(1):

$$L = \sum l\left(f\left(x, W\right), y\right) + \lambda \sum |\gamma| \tag{1}$$

where $x$ is the input image, $y$ is the corresponding label and $W$ is the model weights, so the first part is the sum of the normal training loss of CNN. $\lambda$ is the scaling sparse rate and the second part is the sparsity-induced L1 penalty term. When $\lambda$ is higher, more weights will be forced to be near zero.

Then how to combine these scaling factors with original network? Let's think about Batch Normalization (BN) layer. As shown in Eq.(2), it applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

$$\hat{z} = \frac{z_{in} - \mu B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2}$$

$$z_{out} = \gamma \hat{z} \tag{3}$$

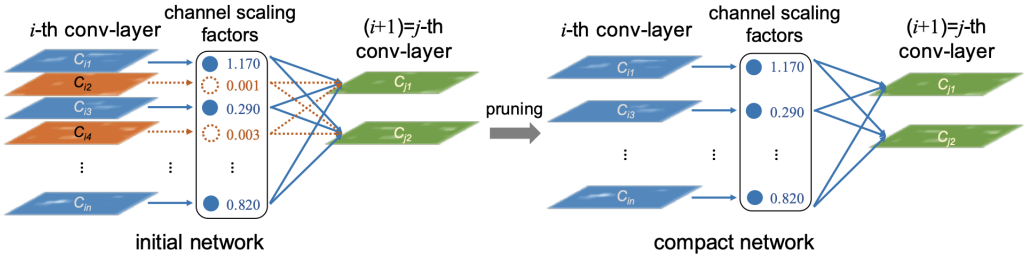


Figure 1: Channel-level pruning process

Here, the $\gamma$ parameter in BN layer is a trainable affine factor which makes it possible to directly leverage our scaling factor in BN layer.

As for other options, if we don't use BN layer and only add a scaling layer after a convolutional layer, then it's actually a linear transformation, the weights decreased in the scaling layer can be enlarged again in the convolutional layer. If the scaling layer is added before BN layer, then the scaling factors will be normalized by BN layer. If we add it behind BN layer, then each channel will have actually two constitute scaling factors. Therefore, Figure 1 shows the pruning process. First, initial network will jointly trained with sparsity regularization by reusing BN layers. Most of the weights are forced to be near zero which are marked with orange color. Depending on the prune ratio, part of those orange channels will be pruned to obtain a compact network. As shown in Figure 2, the pruned networks are then fine-tuned to achieve comparable or even higher accuracy as original models. We also repeatedly try different prune ratio setting to find best configuration.

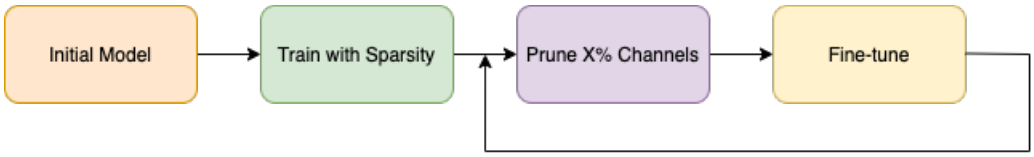


Figure 2: Channel-level whole process

The second approach is proposed by Li, Hao, et al [2] in 2016. Filter-level pruning measures the relative importance of a filter in each layer by calculating the sum of its absolute weights. When increasing the number of pruned filters, the number of matrix multiplications decreases thus making it easy to tune for a target speedup. Since filters in the same layer have identical input channels, the sum of absolute weights also represents the average magnitude of kernel weights. Filters with smaller kernel weights tend to produce feature maps with weak activation as compared to the other filters in that layer. Therefore, the sum of a filter's absolute kernel weights also indicates an expectation of the magnitude of the output feature map.

Figure 3 shows the filter pruning process. We first train original model and load the pre-trained model for pruning. Then we sort the filter of specified layers according to the absolute kernel weights and prune by customized ratios. The detailed procedure of pruning $m$ filters from the $i$th convolutional layer is as follows:

1. For each filter $F_{i,j}$, calculate the sum of its absolute kernel weights $s_j = \sum_{l=1}^{n_i} \sum |K_l|$. ($n_i$ denotes the number of input channels for the ith convolutional layer)
2. Sort the filters by $s_j$.
3. Prune $m$ filters with the smallest sum values and their corresponding feature maps. The

kernels in the next convolutional layer corresponding to the pruned feature maps are also removed.
4. A new kernel matrix is created for both the $i$th and $i+1$th layers, and the remaining kernel weights are copied to the new model.

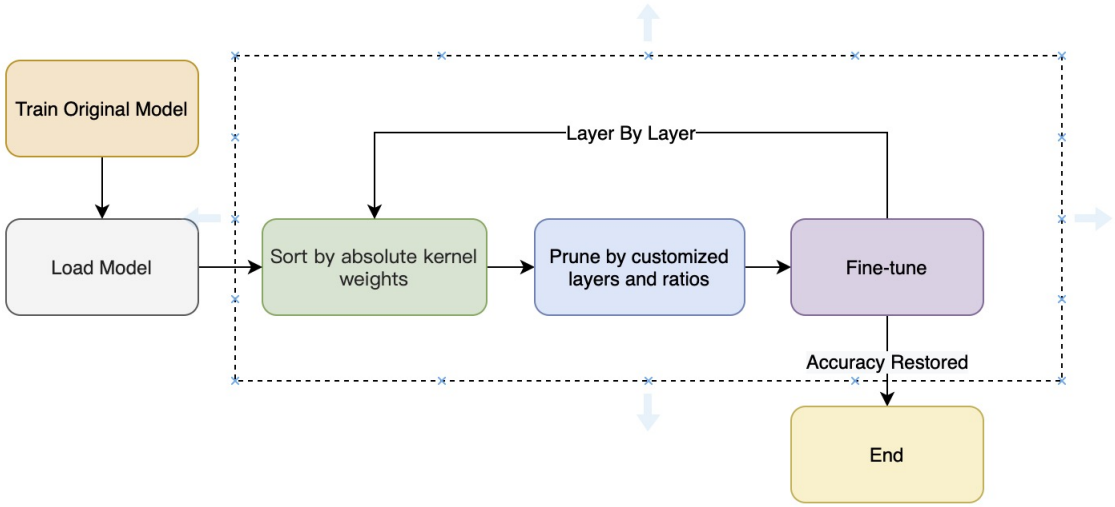


Figure 3: Filter-level whole process

After pruning the filters, the performance degradation should be compensated by retraining the network. There are two strategies to prune the filters across multiple layers:
1. Prune once and retrain: Prune filters of multiple layers at once and retrain them until the original accuracy is restored.
2. Prune and retrain iteratively: Prune filters layer by layer and then retrain iteratively. The model is retrained before pruning the next layer for the weights to adapt to the changes from the pruning process.

Instead of pruning with specific layer-wise hyper-parameters and time-consuming iterative re-training, we use the one-shot pruning and retraining strategy for simplicity and ease of implementation.

## • Implementation Details

The programming language is Python. We choose Pytorch as our deep learning framework and use Flask as the web framework for our application. The dataset we used is CIFAR-10 which consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The baseline models are VGG-19 [3] and ResNet-50 [4]. We replace 3 fully connected layers used in original VGG network with 1 fully connected layer which maps 512 units to 1 unit for output. Our code is based on official implementations of two pruning approaches [5] [6] mentioned in the previous section. The training, pruning and fine-tuning tasks are completed on NYU Greene HPC Platform with V100 GPU.

As for channel-level approach, we train VGG with $\lambda = 0.0001$ and batch size $= 64$ for 160 epochs. We try several prune ratios which are 0.60, 0.65, 0.70, 0.75 and all pruned models are

further fine-tuned for 160 epochs. We train ResNet with $\lambda = 0.00001$ and the prune ratios are 0.45, 0.50, 0.55, 0.60. Other settings are the same as VGG.

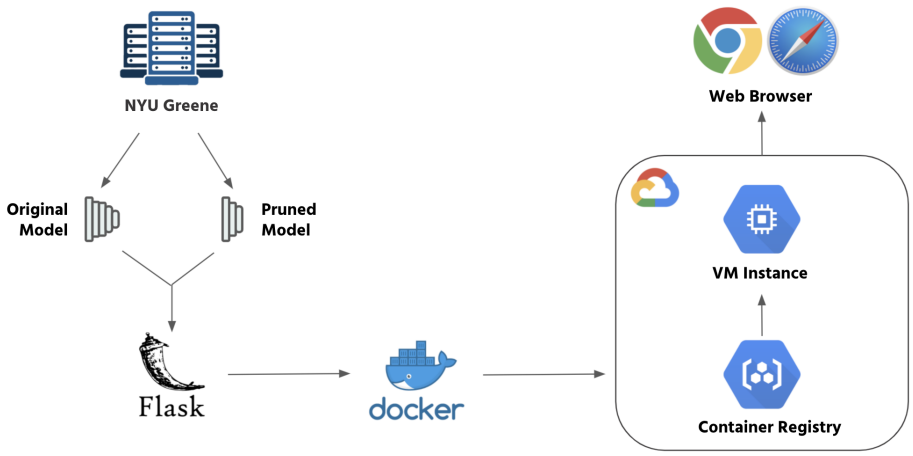


Figure 4: Design Flow

As for filter-level approach, we train VGG with batch size = 128 for 350 epochs. We prune each convolutional layer independently and evaluate the resulting pruned network's accuracy on the validation set to determine the sensitive layers. We then try to find the optimal pruning ratio without significant accuracy drop for each non-sensitive layer. We also retrain for 40 epochs with a constant learning rate 0.001 to compare the result with that before fine-tuning.

For simpler Convolutional Neural Networks like VGG, we can easily prune any of the filters in any convolutional layer. However, for more complex network architectures such as Residual networks, pruning filters may not be straightforward. The architecture of ResNet imposes restrictions and the filters need to be pruned carefully. We prune each residual block independently to observe sensitivity, which is similar to the approach applied to VGG. However, different layers of a residual block also possess different sensitivity. We conduct extensive experiments to find the optimal pruning ratios for each residual block while introducing less hyper-parameters (generalization purpose) and accuracy drop. Other settings are the same as VGG.

As shown in Figure 4, we select 4 models with/without pruning to integrate them with our image classification application. Then we containerize the application with Docker, build the image and push it to google container registry. Then we create a VM instance with V100 and deploy our image to this VM. After adding new firewall rule on Google Cloud VPC to allow port 5000, we are able to see our application through web browsers using external IP.

# • Experiment Results

## ◦ Channel-pruned VGG19

Figure 5 shows the number of parameters and test accuracy of each model. The blue line shows that after pruning, up to 80% parameters are reduced and the test accuracy of fine-tuned models can still be maintained. The results of VGG_pruned_70 and VGG_pruned_75 are even better than baseline model. This is possibly due to the regularization effect of the L1 penalty term on scaling factors. As shown in figure 6, the performance of the pruned models

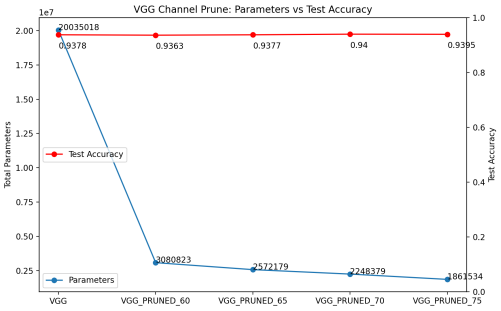


Figure 5: Parameters vs Test Accuracy.

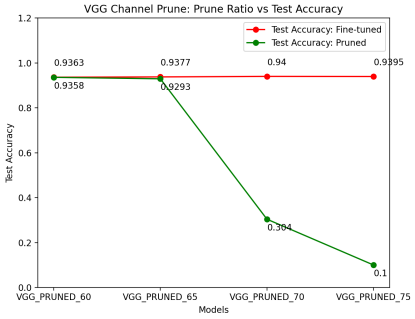


Figure 6: Prune Ratio vs Test Accuracy.

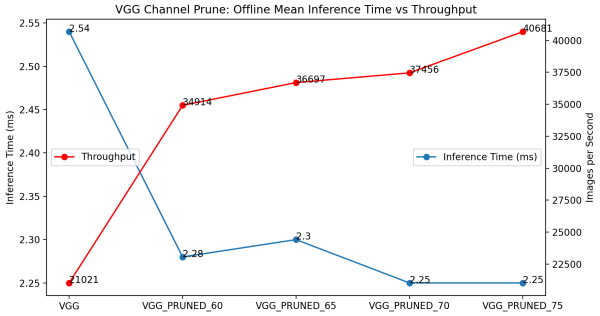


Figure 7: Offline Mean Inference Time vs Throughput

degrade only when the pruning ratio surpasses a threshold. The fine-tuning process can typically compensate the accuracy loss caused by pruning. In figure 7, we evaluate inference time and throughput of each model. Inference time is only measured on the feed-forward of network. Throughput here is defined as the maximal number of input instances the network can process in one second. We do a GPU warm up for both measuring processes in order to prevent GPU going into power-saving status. The result shows that the pruning approach can achieve up to 11% inference time speedup and significantly increase throughput.

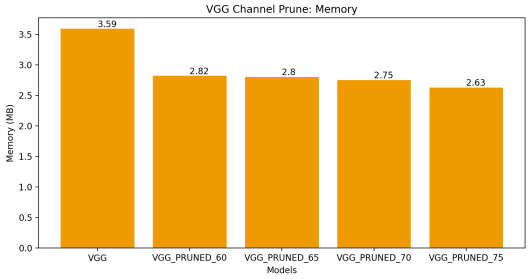


Figure 8: Prune Ratio vs Memory.

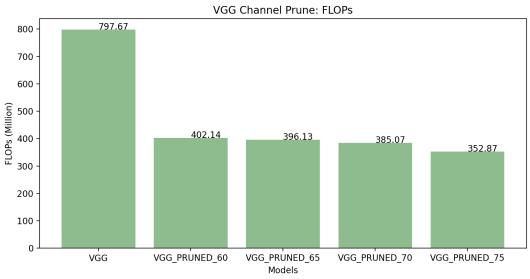


Figure 9: Prune Ratio vs FLOPs.

Figure 8 and figure 9 show run-time memory and FLOPs of each model. Compared to parameter saving, memory saving is relatively insignificant. It's because this pruning approach has a tendency to prune more channels in deeper layers which produce smaller activation maps. Flops saving is more significant than memory saving because convolutional layers are

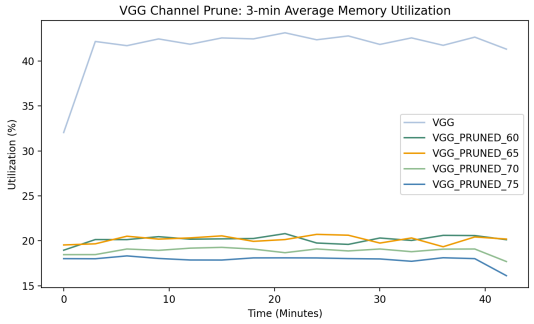


Figure 10: Average Memory Utilization.

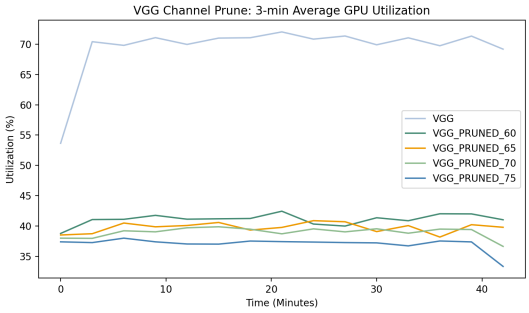


Figure 11: Average GPU Utilization.

compute-intensive. Figure 10 and figure 11 show that after pruning, both memory and GPU utilization are significantly lower than before.

- **Channel-pruned ResNet50**

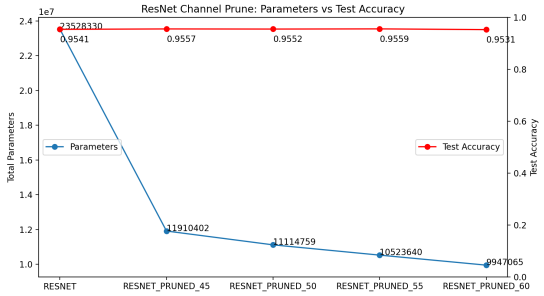


Figure 12: Parameters vs Test Accuracy.

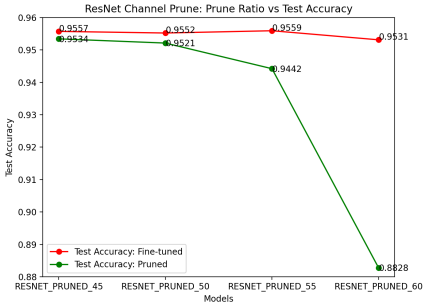


Figure 13: Prune Ratio vs Test Accuracy.

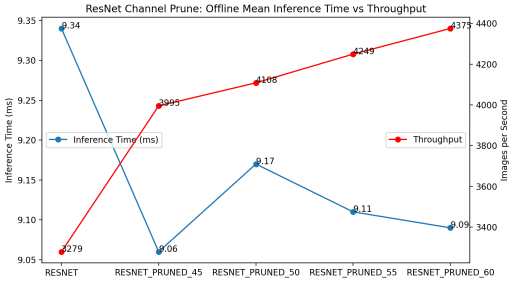


Figure 14: Offline Mean Inference Time vs Throughput

| | VGG | VGG_75 | ResNet | ResNet_60 |
|---|---|---|---|---|
| **Accuracy** | 0.9378 | **0.9395** | 0.9541 | 0.9531 |
| **Parameters** | 20,035,018 | **1,861,534 (-90%)** | 23,528,330 | 9,947,065 (-58%) |
| **Inference** | 2.54ms | **2.25ms** | 9.34ms | 9.09ms |
| **Throughput** | 21,021 | **40,681 (+94%)** | 3,279 | 4,375 (+33%) |

Figure 15: VGG vs ResNet.

Next let's see experiment results of ResNet. As shown in Figure 12 and Figure 13Similar to VGG, only when the pruning ratio is larger than a threshold, here is 55%, will significantly hurt model performance. And further fine-tuning process could compensate this loss or even increase test accuracy. However, as shown in Figure 15, although the test accuracy of original and pruned ResNet are better than VGG, both of parameter saving and throughput increasing are relatively insignificant. It's probably due to the bottleneck structure which

has already been designed for channel selection. Therefore, pruned VGG with 2ms inference time and reasonable accuracy is more suitable for real-time application. Also in Figure 18 and Figure 19, the decrease of memory and GPU utilization are insignificant too.

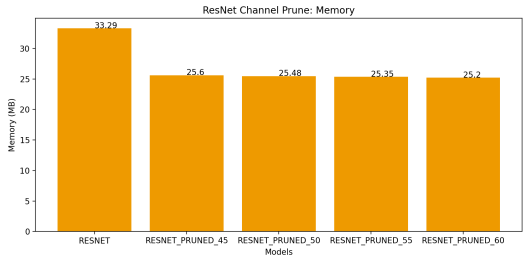


Figure 16: Prune Ratio vs Memory.

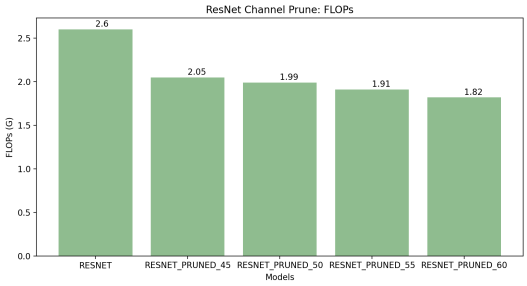


Figure 17: Prune Ratio vs FLOPs.

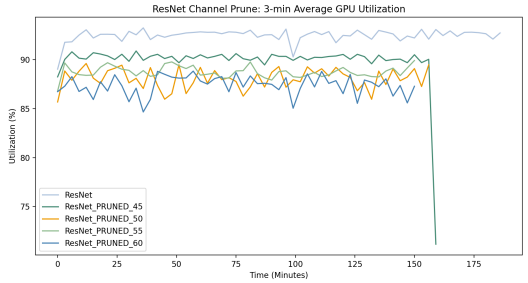


Figure 18: Average Memory Utilization.

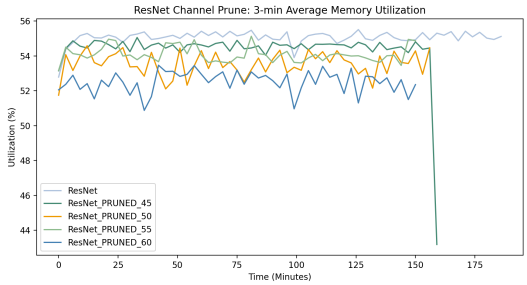


Figure 19: Average GPU Utilization.

## ◦ Filter-pruned VGG19

We observe that the first layer and each of the convolutional layers with 512 feature maps is robust to pruning as compared to the other layers. With 70% of the filters being pruned in layer 1 and 50% of the filters being pruned from 9 to 16, we achieve 38% FLOP reduction for similar accuracy.

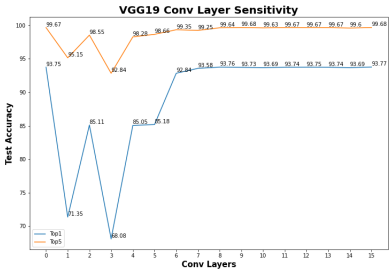


Figure 20: VGG19 Convolutional Layer Sensitivity

## ◦ Filter-pruned ResNet50

As shown in Figure 22, most of the residual blocks are robust to pruning. In addition, we find that the residual blocks close to the layers where the number of feature maps changes tend to

| Layer Type | $w_i \times h_i$ | Ori #Maps | #Maps | Reduced FLOPS% |
|---|---|---|---|---|
| Conv_1 | $32 \times 32$ | 64 | 18 | 70% |
| Conv_2 | $32 \times 32$ | 64 | 64 | 70% |
| Conv_3 | $16 \times 16$ | 128 | 128 | 0% |
| Conv_4 | $16 \times 16$ | 128 | 128 | 0% |
| Conv_5 | $8 \times 8$ | 256 | 256 | 0% |
| Conv_6 | $8 \times 8$ | 256 | 256 | 0% |
| Conv_7 | $8 \times 8$ | 256 | 256 | 0% |
| Conv_8 | $8 \times 8$ | 256 | 256 | 0% |
| Conv_9 | $4 \times 4$ | 512 | 256 | 50% |
| Conv_10 | $4 \times 4$ | 512 | 256 | 75% |
| Conv_11 | $4 \times 4$ | 512 | 256 | 75% |
| Conv_12 | $4 \times 4$ | 512 | 256 | 75% |
| Conv_13 | $2 \times 2$ | 512 | 256 | 75% |
| Conv_14 | $2 \times 2$ | 512 | 256 | 75% |
| Conv_15 | $2 \times 2$ | 512 | 256 | 75% |
| Conv_16 | $2 \times 2$ | 512 | 256 | 75% |
| Linear | 1 | 10 | 10 | 50% |
| Total | | | | 38% |

Figure 21: VGG19 optimal configuration

be more sensitive to pruning, e.g., the first and the last residual blocks for each stage. We skip those layers and prune the remaining layers at each stage equally. We compare three optimal configurations of pruning percentages for the four stages: (A) p1=30%, p2=30%, p3=30%, p4=30%; (B) p1=50%, p2=40%, p3=60%, p4=30% (C) p1=50%, p2=60%, p3=40%, p4=30%. From the result we can see that VGG19 can be used directly after pruning the pre-trained model without further fine-tuning, however, test accuracy drops significantly without re-training ResNet50. However, these non-sensitive layers are resilient to pruning, the prune and retrain once strategy can be used to prune away significant portions of the network and any loss in accuracy can be regained by retraining for a short period of time (less than the original training time and in our case just 40 epochs)

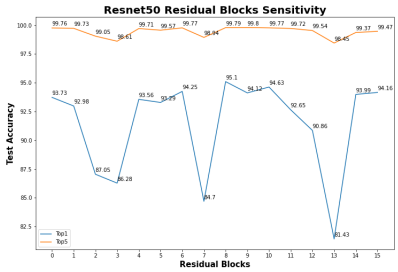


Figure 22: ResNet50 Residual Block Sensitivity

## ◦ Online Testing

Figure 23 shows online testing performance of each model. We deployed 3 VGG-based models, one is network with full structure, the others are pruned using channel-level approach and filter-level approach separately. The same configuration is also applied to ResNet-based

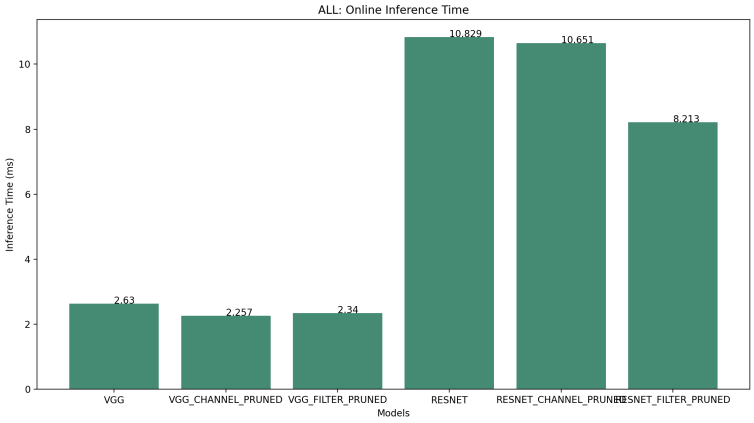


Figure 23: Online Inference Time

models. We can see that for each model, online inference time slightly increases compared to offline inference time. Also, channel-level pruning is able to achieve lowest inference time when applying to VGG and filter-level pruning performs best when applying to ResNet. However, VGG_CHANNEL_PRUNED is almost 3.6x faster than RESNET_FILTER_PRUNED which means it is more suitable for this real-time application.

# • Conclusion

Compared to setting different pruning ratio to each layer, global threshold channel-level pruning is more flexible and easier to implement, which can also achieve higher parameter saving. When using channel-level pruning, removing the same percentage of channels, larger amount of redundant parameters can be removed from VGG. On ResNet50, parameter saving is relatively insignificant. Within a threshold, test accuracy of pruned and fine-tuned model only decreases slightly or even achieves higher accuracy than original model. However, too aggressive pruning strategy will hurt the model structure and lead to bad performance. For relatively small dataset, pruned VGG is a better choice for online application which has better tradeoff among accuracy, inference time and memory.

# • References

1 Liu, Zhuang, et al. "Learning efficient convolutional networks through network slimming." Proceedings of the IEEE International Conference on Computer Vision. 2017.

2 Li, Hao, et al. "Pruning filters for efficient convnets." arXiv preprint arXiv:1608.08710 (2016).

3 K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015

4 K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In CVPR, 2016.

5 https://github.com/Eric-mingjie/network-slimming.git

6 https://github.com/tyui592/Pruning_filters_for_efficient_convnets.git