A Systematic Framework for Opportunistic Pruning of Deep Neural Networks on Edge Devices

Robert Viramontes
rviramontes@wisc.edu
Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, Wisconsin, USA

Azadeh Davoodi
adavoodi@wisc.edu
Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, Wisconsin, USA

ABSTRACT

Deep Neural Networks (DNNs) are getting increasingly complex and 'general purpose'. At the same time, there is great interest in using edge devices to infer very specific tasks, which allows for opportunistic DNN pruning. Many pruning techniques have been proposed for simplifying DNNs, but they are often developed in isolation from each other. In some cases, an edge device may not have architectural support to actually see the benefits of pruning for objectives such as latency. In this work, we propose `OppPrune` as a systematic framework to opportunistically prune a complex DNN into one targeted for more specific edge inference tasks. `OppPrune` hierarchically combines a set of structural pruning techniques which do not rely on assumptions about the architecture of the edge device in order to meet a target latency. In our experiments with two more-specific datasets, we show `OppPrune` can effectively combine two recent fine-grained and coarse-grained pruning techniques, and perform significantly better than each one individually.

KEYWORDS

neural networks, pruning, profiling, edge-based inference

1 INTRODUCTION

In recent years, deep neural networks (DNNs) have been rapidly deployed on edge devices. Compared to cloud inference, this can improve latency and privacy while reducing communication costs [13]. However, the computational complexity of these DNNs can limit the practical ability to store and execute them on edge devices.

At the same time, for many practical applications, the inference task that should be executed on an edge device is significantly simpler than the tasks solved by competition-grade DNNs. For example, the COCO dataset has 80 object classes and the ImageNet competition has 1000 classes. However, a typical edge application may require significantly fewer classes. For instance, a coffee roaster may only be interested in identifying under-, over-, and optimally-roasted beans [15]. Hence, a neural network for edge applications is not just smaller, but also *more specific* than a general-purpose DNN. Fortunately, these goals overlap as we can remove extraneous information to shrink the network while making it more specific. This extends the lottery ticket hypothesis [6] that observes that a DNN can be extensively pruned after training, introducing a reduced class dimension to aid pruning efforts.

Many DNN pruning techniques have been proposed including edge pruning, weight quantization, and channel, layer, and block pruning. *However, when using pruning on complex DNNs to run more specific tasks and on edge devices, three sets of problems must be specifically taken into account.*

- **First**, often times, the primary goal of pruning techniques is to compress the DNN as much as possible which is measured in terms of number of stored parameters. However, for many edge applications, an important goal is to ensure that the inference latency does not exceed a target time. This is relevant, for example, if the inference results trigger a time-sensitive action, impacting the user's experience.
- **Second**, some pruning techniques rely on specific architectural features to be present in the edge device to observe the benefits in the context of not exceeding a target inference latency. For example, weight pruning allows condensed storage of sparse weight matrices. But there may be little to no benefit in latency in the absence of zero-skip hardware. Similarly, weight quantization allows the storage requirement of the DNN to be reduced. However, quantization does not necessarily translate into latency improvement in the absence of specialized hardware.
- **Finally** and most importantly, different classes of pruning techniques are proposed in isolation from each other. To the best of our knowledge, there are no studies on how different techniques may be systematically combined to obtain a more efficient and/or effective pruning framework.

In this work, to address the above points, we propose `OppPrune` which is a systematic framework to Opportunistically Prune a 'general purpose' DNN for edge inference. Pruning is done to meet a target latency requirement with minimal loss in accuracy without relying on architectural features of the edge device. We group the pruning techniques into fine and coarse -grained and combine them in an iterative framework. For a more-specific inference task, `OppPrune` first applies the more aggressive coarse-grained pruning to bring the latency of the model much closer to the target latency, before it applies the much-more time-consuming fine-grained pruning scheme. In our experiments, we demonstrate that, under the same latency constraint that `OppPrune` finds a similar solution but with significantly faster runtime compared to `NetAdapt` [18] which is a relatively-recent fine-grained approach based on channel pruning. We also compare to a coarse-grained approach based on block pruning, and show `OppPrune` can achieve a significantly better accuracy with a reasonable runtime. These are using more-specific datasets, i.e., ImageNette (different from ImageNet) and ImageWoof.

2 BACKGROUND AND RELATED WORK

In this section, we first give a brief overview of pruning and model compression techniques, and then review existing frameworks from the perspective of how they utilize these pruning techniques.

2.1 Overview of Pruning Techniques

We explain two structural pruning techniques which are suitable for opportunistic edge pruning, and are utilized in this work, as well as a quick review of other model compression techniques.

2.1.1 Channel Pruning. Channel pruning is a well-known method targeting convolution layers. Also called filter pruning, this technique removes entire channels from the convolution kernel. The convolution operation involves convolving the channels of a weight 'kernel' with the input data. For example, in a convolution with a 2D input, the kernel is 3D: height, width, and number of output channels (filters). Channel pruning removes some portion of convolution filters, reducing the size along this 3rd dimension.

2.1.2 Block Pruning. Block pruning (such as [16]) refers to structural pruning methods that alter the *architecture* of a DNN by entirely removing one or more layers. While the smallest granularity of these techniques is to remove a single layer, it may be better to consider a block of related layers which is a more recent technique. For instance, a convolution operation is often followed by an activation operation, such that they form a single conceptual block of operations. This concept extends to larger blocks of many operators, such as the residual blocks of ResNet-style networks [7].

2.1.3 Other Model Compression Techniques. Another pruning technique is weight pruning that sets particular edge weights of the DNN to zero. While this allows condensed storage of sparse weight matrices, there may be little to no latency benefit [2] because the number of computations may not be reduced.

Quantization is another technique that reduces the precision and/or changes the datatype of the network, which are typically originally trained and stored as `float32`. It reduces the storage requirements of the DNN, and in the case of datatype conversion, may enable use of faster hardware. However, a latency benefit is hardware-dependent, and *for instance NVIDIA Jetson Nano GPU does not provide support for accelerating integer networks. Since the focus of our study is hardware-agnostic latency reduction, quantization and weight pruning are not considered in* `OppPrune`.

2.2 Pruning Frameworks

We will discuss several pruning frameworks that consider *how* pruning techniques should be engineered to be most effective. We group these into iterative and one-shot frameworks.

2.2.1 Iterative Frameworks. One method for iterative pruning is proposed in the original `NetAdapt` [18] framework, which developed a methodology for pruning DNNs with a focus of running on mobile/low-compute devices. In each iteration, the framework applies a pruning technique at different points of the DNN to generate several candidate DNNs. The candidates are briefly trained, picking the candidate which suffers the least accuracy loss. This winning DNN is the input for the next iteration, and iterations continue until a latency constraint is met. The `NetAdapt` work only experiments with channel pruning approaches, but notes that its iterative framework is conceptually generic to other pruning approaches. However, it is designed to include only one pruning technique and it is unclear how it may be generalized to combine multiple techniques in one shot.

Discrimination-based block-level pruning [16] is another iterative framework which focuses only on pruning blocks for latency reduction. In each iteration, it sets a ratio of blocks to remove in the iteration and uses a discrimination criteria to select the blocks to remove. Between each iteration, the network is trained briefly, and iterations continue until a user-defined constraint is met, effectively reducing the latency.

In CAP'NN [8], neurons are iteratively pruned it is shown that the pruned model may *improve* accuracy on a subset of the classes in a more-specific dataset.

Iterative pruning allows for a clear separation of pruning techniques at the iteration boundaries. In addition, it provides opportunities for complex quality constraint assessments because complete models are generated at each step. This flexibility comes at the cost of a laborious pruning process due to discarding significant retraining computations when a single model is kept at each step.

2.2.2 One-Shot Frameworks. Another style of approach for pruning reduces the weights in "one-shot", without multiple rounds of pruning and retraining. For instance, Huang and Wang [10] utilized a technique that adds a scaling factor to the output of each block and then uses accelerated proximal gradient method to learn the scaling factors during training. Scaling factors which are close to zero, identify structures that can be pruned, and these structures can be removed simultaneously (in one-shot) at the end of training. Lin et al. [12] follow a similar approach to learn which elements may be pruned, but instead utilize a generative adversarial learning framework which allows identifying prunable structures without labels. After the mask is learned, the network is pruned in one-shot. While these techniques can effectively prune the network with only a single pruning pass, they still require epochs of learning to compute the pruning masks so their runtime benefit is unclear.

3 APPROACH

Figure 1 shows a high-level flow of `OppPrune`. The input is an unpruned DNN which is *initially pre-trained on the more-specific dataset.* The left-side of the flow chart shows how the coarse-grained and fine-grained pruning techniques are combined in the hybrid platform. The figure is drawn showing block and channel pruning as the coarse and fine -grained techniques, respectively. Both of these pruning schemes result in latency improvements without relying on specific architectural features of the edge device.

As shown in the left figure, `OppPrune` first applies the more aggressive coarse-grained pruning, to quickly prune the model to bring its latency much closer to the target latency, before it applies the much-more time-consuming fine-grained pruning scheme. More specifically, with the assumption that the initial DNN is violating the latency constraint, `OppPrune` first applies coarse-grained pruning to aggressively prune the model. This continues until the latency constraint is satisfied. Next, it switches to fine-grained pruning, using the last model which was *not* satisfying the latency constraint as the starting point. Fine-grained pruning is then applied until the timing constraint is satisfied. This is the final pruned model which is retrained (fine tuned) at the last step.

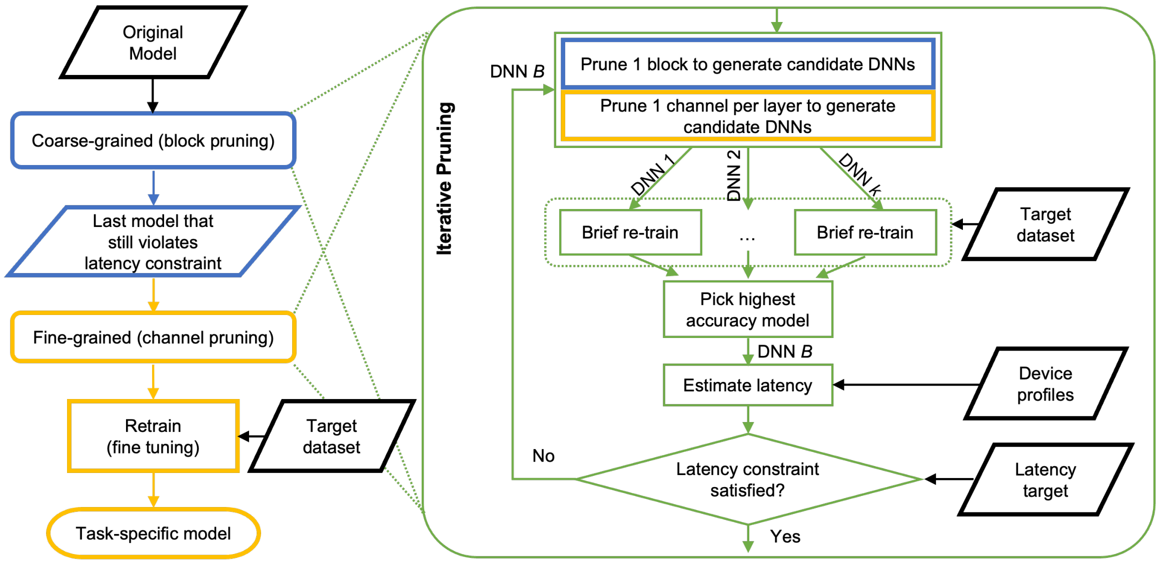
Original Model
Coarse-grained (block pruning)
Last model that still violates latency constraint
Fine-grained (channel pruning)
Retrain (fine tuning)
Target dataset
Task-specific model
Iterative Pruning
DNN B
Prune 1 block to generate candidate DNNs
Prune 1 channel per layer to generate candidate DNNs
DNN 1
DNN 2
DNN k
Brief re-train
…
Brief re-train
Target dataset
Pick highest accuracy model
DNN B
Estimate latency
Device profiles
No
Latency constraint satisfied?
Latency target
Yes

Figure 1: Flow chart of `OppPrune`: left-side shows the hybrid approach (block pruning followed by channel pruning), and right-side shows the iterative process. Input is unpruned DNN model that is already pre-trained on the more-specific dataset.

Each level of pruning (i.e., block or channel) is itself an iterative process which is shown on the right-side of the figure. One flow chart is drawn for both because they follow the same steps. The only difference is in the first step of an iteration which is shown with different colors corresponding to block versus channel pruning.

This iterative pruning process also requires target dataset (for brief retraining of pruned DNNs), device profiles and target latency (for generating latency estimates to evaluate if the latency constraint is satisfied). The goal of `OppPrune` is to minimize the loss in the inference accuracy subject to satisfying the target latency.

Next, we discuss how the iterative process works, for each of the coarse and fine -grained schemes. We also discuss our proposed latency estimation scheme which is driven by device profiling data and is utilized during pruning.

3.1 Iterative Pruning

We discuss more details about the two distinct stages of iterative pruning: coarse-grained pruning, shown in blue, and fine-grained pruning, shown in yellow, in Figure 1.

The current implementation of `OppPrune` uses block-pruning as the main technique to drive the coarse-grained step. We implement our own block-pruning scheme. A block is a set of linked operations, such as convolution followed by activation or an entire residual block, as in ResNets. When a block is selected for pruning, we trace the dependencies of the block and discover any conflict in dimensions of the remaining blocks. If a conflict is discovered, an adjacent block is removed and replaced with a block that fixes the dimension conflict. By removing and fixing only at a block-level, we stick completely to the coarse-grained regime.

In the block pruning stage, we start by generating candidate networks with one block removed. That is, if the input network has B blocks, we will generate B candidate networks each with a unique $B-1$ subset of the input blocks. These candidate networks are briefly re-trained on the training dataset. Because the re-training at this step is not intended to provide an *absolute* measure of each candidate DNN's accuracy, but simply *rank* the candidate networks, the training does not need to be for many epochs. While this helps to limit the computation required in each iteration, training is a computationally expensive process. Observe that the training of each candidate model is independent and can be run in parallel.

After the candidates have gone through brief re-training, they are all evaluated for accuracy on the validation split of the dataset. The candidate DNN with maximum accuracy, in other words with the minimal accuracy loss, is selected and the remaining candidate networks are discarded. In this way, we can consider this a greedy approach to pruning because the best candidate at each iteration is selected. The latency of the winning DNN is estimated and compared to the latency target given as input. If the latency target is not satisfied, the winning DNN is used as the start for the next round of block pruning. If the constraint is satisfied, we finish the block pruning and move on to channel pruning.

At the end of block pruning, the last model already satisfies the latency requirement. However, because of the relative ‘coarseness’ of block pruning, the selected model tends to *overshoot* the latency target. As a result, this model suffers from additional, unnecessary accuracy loss to meet the latency target under this coarse regime. To prevent this unnecessary loss, the second stage is done with a ‘finer’ channel pruning using the last model from block pruning that did *not* satisfy the latency constraint.

From this point, the channel pruning stage operates in much the same way as block pruning. It first generates a set of candidate networks by applying channel pruning one layer at a time. At each layer, a specific number of channels is pruned (which we clarify in our experiments). Therefore, if L layers are in the model provided as input to the fine-tuning phase, L candidate models are generated at each iteration. Channel pruning proceeds until the latency constraint is satisfied, at which point the final model configuration is determined. After channel pruning, the DNN is trained for a longer term (more epochs) to recover the most accuracy possible. After this step, we obtain the final pruned model that has been re-trained on the new, more specific task.

In channel pruning, we use the original `NetAdapt` implementation [18] to explore how much pruning to apply to each layer. The actual pruning is implemented with TorchPruning [5] using L2-norm, which removes the filters and performs operations to fix any connected layers whose dimensions must also change.

By designing the flow of pruning techniques in this manner, we use coarse block pruning to efficiently reach the neighborhood of the quality target. Then, by switching to the channel pruning, we can finely hone in on the latency target while preserving accuracy. Note that, according to [3], by starting with block pruning, we may even be able to prune the model to a latency not possible with channel pruning alone.

3.2 Latency Estimation During Pruning

We first discuss our proposed block-level latency estimation, utilizing device-level profiling data. For channel pruning, we use the existing device-level latency estimation technique in `NetAdapt`. In both cases we rely on device profiling data, as several works have established its benefit instead of analytical or statistical alternatives, to accurately estimate the inference time of a DNN [4, 11].

To estimate the latency of a candidate DNN during block-pruning, we first profile the inference time of the complete, unpruned network with a small, low-overhead modification to record the latency of each block 'along the way'. Using the per-block latency estimates, we then estimate the inference latency of a candidate DNN by subtracting the latencies of the corresponding removed blocks from the latency of the unpruned model. Because during pruning, the configurations of the blocks change very little, if at all, profiling a single unpruned model is all that is needed.

These profiles must be generated on the target hardware, but then can be exported to another device. This allows the pruning process, which includes computationally demanding retraining, to be offloaded to a more performant device while still reaping the accuracy benefits of profiling. In our work, we utilize NVIDIA's TensorRT [1] tool which performs device-specific optimizations to the DNN under profiling. In practice, a model deployed to the device would tend to be optimized with TensorRT, so we find it prudent to use during the profiling process as well.

In Figure 2, we show the actual latency and estimated latency of a DNN as one block is pruned per iteration for 7 consecutive iterations, with profiling on an NVIDIA Jetson Nano device. As the figure shows the estimated and actual latency closely track until removing more than half of the blocks. In practice, the number of pruned blocks are just a few before switching to fine-grained scheme, making our block-level estimation accurate in that range.

Finally, for channel pruning, we use the same latency estimation as in `NetAdapt`. Here, the inference latency of a candidate pruned network is taken as the sum of latencies of its layers which are found from a lookup table. To generate the lookup table, device profiling is used. We generate a set of possible "configurations" for each layer based on number of its pruned channels, and measure each configuration of each layer independently. This latency estimation technique used by `NetAdapt` requires many subnetworks to profile. As a result, it takes significantly longer time than our proposed block-level estimation technique.

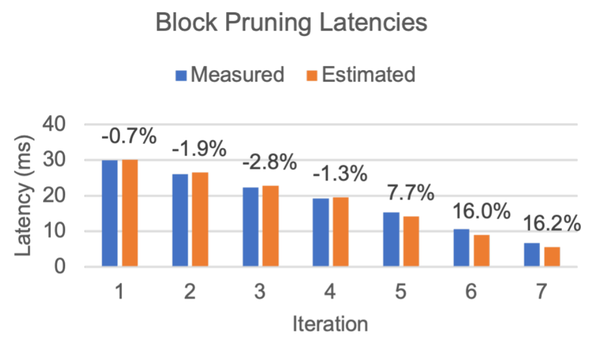

Figure 2: Comparison of our proposed estimated latency scheme with the actual one as blocks are iteratively pruned (in x-axis) in a DNN. Our coarse-grained model is obtained by collecting latency profiles of each block of a DNN (only once) from the NVIDIA Jetson Nano device. The actual model measures the latency of the pruned DNN as a whole. When fewer blocks are pruned, the accuracy remains relatively low.

4 EXPERIMENTAL RESULTS

To validate `OppPrune`, in our experiments, we focus on the image classification task, as there are ample resources and pre-trained models applicable for this task. We start with networks that have been trained on and have achieved notable performance on the ImageNet [14] dataset. We then use the more specific ImageNette and ImageWoof [9] datasets, which are subsets of the ImageNet dataset to define the specific task. ImageNette contains diverse classes (e.g. golf ball and parachute) while ImageWoof contains similar classes: different breeds of dogs. The characteristics of these datasets are described in Table 2. ImageNette and ImageWoof have only 10 classes, a 99% decrease in classes representing a much more specific task. The images are set for a (3, 224, 224) input size and the standard ImageNet transformations are applied for the training and validation sets.

For our target device, we select the NVIDIA Jetson Nano 4GB module. This is a low-cost edge device with an embedded GPU module that allows accelerating inference of DNNs. Profiles are collected utilizing NVIDIA's TensorRT utility, utilizing a 5 second warm up period and averaging over 100 runs of the network. During the iterative pruning process, we set the 'brief re-training' (shown in right side of Fig. 1) to run for 5 epochs and the final 'retrain (fine tuning)' step for 25 epochs.

4.1 Results of the ResNet-18

We first compare pruning approaches by experimenting with the ResNet-18 network, using weights for ImageNet. The ResNet-18 network has 20 convolutional layers that are organized into 8 different blocks. We replace the final layer (the "classification head") to reduce the number of classes from 1000 to 10, and the entire network is briefly trained on the target dataset. This network is fed into `OppPrune`. In addition, the channel-based and block-based latency profiles of the Jetson Nano device are collected and also provided as input to `OppPrune`. These are used as shown in the flow chart in Figure 1.

Table 1: Results for pruning ResNet-18 on two more-specific datasets for a latency target of 15ms

	ImageNette				**ImageWoof**			
	Latency (ms)	Runtime (hrs)	Accuracy	#Iterations	Latency (ms)	Runtime (hrs)	Accuracy	#Iterations
Original	19.06	n/a	99.00	n/a	19.06	n/a	99.00	n/a
Coarse-Only	13.24	0.41	97.61	2	13.24	0.23	92.59	2
Fine-Only (`NetAdapt`)	14.90	1.64	98.09	8	14.81	1.11	93.54	9
Corase-to-Fine (`OppPrune`)	14.69	1.26	98.09	1⟶5	14.77	1.05	93.23	1⟶6

Table 2: Description of the datasets

	Training	**Validation**	**#Classes**
ImageNet (generic)	1,281,000	50,000	1000
ImageNette (specific)	9,469	3,925	10
ImageWoof (specific)	9,025	3,929	10

The unpruned version of ResNet-18 network takes 19.06 ms to execute on the Jetson Nano, and based on this number, we specify 15 ms as the latency target for `OppPrune`. This is approximately 25% reduction in latency.

The results are shown in Table 1. We compare coarse-grained-only, fine-grained-only and `OppPrune` which is our hybrid coarse-to-fine pruning scheme. For fine-grained-only approach we use the `NetAdapt` code [18], implementing our own extension for ResNet-18, and provide it with the channel pruning profiles of the Jetson Nano device. This is the same fine-pruning-scheme utilized inside `OppPrune`. For coarse-grained-only we use the same block-pruning scheme as used in `OppPrune` which was explained in Section 3.1.

In the table, the latency of the original (unpruned) model is obtained by executing the DNN on the Jetson Nano device completely. For the remaining techniques, we used our latency model proposed in Section 3.2 which utilizes the channel-based and block-based profiles of the Jetson Nano device.

Besides latency, we also report the overall runtime of the algorithm[1], accuracy of the pruned DNN model, and the number of iterations the algorithm took to reach the exit condition. For `OppPrune` we report the number of iterations for the fine-grained phase and coarse-grained phase separately. For the accuracy, we evaluate the Top-1 accuracy on the validation dataset after the additional fine-tuning step.

We make several key observations from these results:

- We observe that coarse-only scheme 'overshoots' the latency target, finishing with a latency of 13.24ms in both datasets which is over 1.76ms below the target latency. `OppPrune` and fine-only schemes finish with latency of at most 0.31ms of the target latency which is much closer to the target, allowing for achieving a higher accuracy.
- The fine-only and `OppPrune` schemes have identical accuracy results for ImageNette and very similar results for ImageWoof. This indicates that the coarse-to-fine scheme is able to mitigate the accuracy loss of coarse-only pruning.
- `OppPrune` has significantly better runtime than `NetAdapt`. This reduction is primarily the result of reducing the number of time-consuming channel-pruning iterations. For example in ImageWoof, `NetAdapt` has 9 iterations of channel pruning while `OppPrune` has 6 iterations of channel pruning (after performing 1 iteration of fast block pruning).

[1]Evaluated on Ubuntu 20.04 with Intel i9 9900K CPU and NVIDIA 2080Ti GPU.

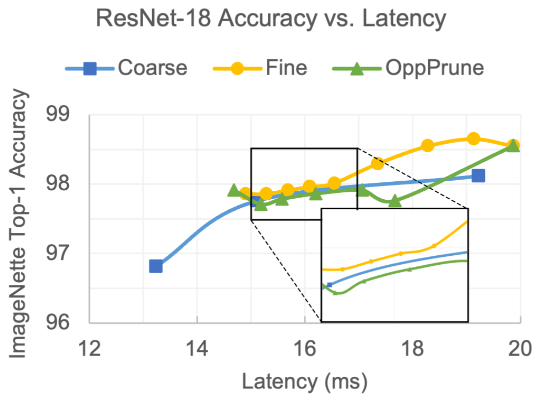

Figure 3: The evolution of accuracy and trade off for each step of the iterative pruning process (from right to left), for each of the 3 pruning techniques.

To understand how the network transforms over time, we plot the Latency-Accuracy tradeoff for each step of the iterative process when pruning for ImageNette in Figure 3. We observe that `OppPrune` and fine-only schemes follow similar trends in the fine-tuning phase (approximately below 16ms), but `OppPrune` skips several iterations by employing block pruning first. This also illustrates the "overshoot" problem of coarse-only pruning, which ends quite far to the left and below the points found by the other two methods, because it can only satisfy the latency exit condition by removal of an entire block, leading to unnecessary additional accuracy loss.

Overall, ResNet-18 demonstrates that a coarse-then-fine pruning methodology provides a good balance between algorithm runtime, accuracy degradation, and latency targets.

4.2 Results of the ResNext-50

We also experimented with the larger ResNext-50 network [17]. This network has a similar residual block architecture to the ResNet networks, but extends them by utilizing grouped convolutions. It includes many more layers than the ResNet-18. It has 53 convolution layers organized into 16 blocks instead of 20 layers in 8 blocks.

On our target Jetson Nano device, the original ResNext-50 network takes approximately 83ms to execute. We set a 33ms target latency in this experiment. This is to satisfy a 30 frames-per-second constraint, a common framerate for commodity camera hardware.

For fine-only case, we set the `NetAdapt` algorithm to prune more aggressively. This is done by setting the input parameter `reduction ratio = 0.10` in `NetAdapt` which forces it to complete pruning in a more reasonable amount of time, given that the target is a 60% decrease from the original latency. We keep the brief re-training set at 5 epochs and final re-train at 25 epochs.

Table 3: Results for pruning ResNext-50 on two more-specific datasets for a latency target of 33ms

	ImageNette				ImageWoof			
	Latency (ms)	Runtime (hrs)	Accuracy	#Iterations	Latency (ms)	Runtime (hrs)	Accuracy	#Iterations
Original	83.94	n/a	99.59	n/a	83.94	n/a	96.61	n/a
Coarse-Only	31.09	8.00	94.19	11	31.09	7.65	86.89	11
Fine-Only (NetAdapt)	32.51	20.30	95.67	11	29.76	29.07	91.65	16
Coarse-to-Fine (OppPrune)	30.31	9.03	95.95	10 → 2	30.14	9.15	89.18	10 → 2

The results for pruning and specifying ResNext-50 for our two datasets are shown in Table 3. Similar to the ResNet-18 network, we observe that the coarse-only result performs the worst in terms of accuracy. Fine-only and OppPrune achieve a similar accuracy under the same latency constraint. Here, the runtime difference is much more pronounced, with NetAdapt taking over twice as long as OppPrune. This is because the *bulk* of the latency reduction is accomplished with the relatively less demanding block pruning algorithm, and only 2 iterations of channel pruning are required to fine-tune the block-pruned DNN, compared to 16 time-consuming channel-pruning iterations (in ImageWoof) in NetAdapt.

Note that runtime is a function of user parameters set for channel pruning. A user could set a larger granularity for channel pruning to more rapidly reduce latency, and control a decay factor implemented in [18] to allow the granularity to reduce as the latency target is approached. However, we argue that this kind of parameter exploration could require several expensive trials to tune and a deeper understanding of the pruning process than the general application builder may have. Hence, mixing techniques and allowing block pruning to be coarse and channel pruning to stay fine achieves similar results while eliminating expensive tuning efforts.

Overall, our experiments confirm the validity and benefits of OppPrune that a coarse-to-fine approach to pruning allows blending these techniques to achieve runtime-efficient methods to significantly reduce latency while mitigating accuracy loss.

4.3 Importance of Pre-Training

In OppPrune, we first pre-train the model on the desired, more-specific dataset *prior* to starting the pruning process. To highlight the importance of this step, we compare to an alternative where the network is still initialized with the ImageNet weights, but the replacement "classification head" is randomly initialized. For ResNet-18, when using the randomly-initialized head, the DNN pruned by OppPrune achieves a Top-1 accuracy of 97.15% on ImageNette. When using a model trained for 25 epochs on ImageNette prior to pruning, the pruned DNN achieves a higher 97.91% accuracy.

5 CONCLUSIONS

In this work, we demonstrated OppPrune, an opportunistic DNN pruning framework that targets latency reductions for edge devices. Compared to prior works, we utilize a *coarse to fine* workflow. We reap the benefits of rapid latency reduction from block pruning with the accuracy preservation of channel pruning. Our experiments with ImageNette and ImageWoof more-specific datasets demonstrate that our approach achieves these goals, with similar accuracy to only channel pruning while keeping runtime similar to block pruning. In addition, our method does not require expert parameter tuning or development, minimizing the barrier to entry for deploying a DNN to the edge.

6 ACKNOWLEDGEMENT

GitHub Copilot was enabled for a portion of code development.

This material is based upon work supported by the National Science Foundation under Grant No. 200639.

REFERENCES

[1] 2023. TensorRT SDK | NVIDIA Developer. https://developer.nvidia.com/tensorrt. Accessed: 2023-11-15.

[2] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A Survey of Model Compression and Acceleration for Deep Neural Networks. (2017). arXiv:1710.09282 arXiv:1710.09282[cs.LG].

[3] Sara Elkerdawy, Mostafa Elhoushi, Abhineet Singh, Hong Zhang, and Nilanjan Ray. 2021. To Filter Prune, or to Layer Prune, That Is the Question. In *Computer Vision – ACCV 2020*, Hiroshi Ishikawa, Cheng-Lin Liu, Tomas Pajdla, and Jianbo Shi (Eds.). Springer International Publishing, Cham, 737–753.

[4] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. 2021. JointDNN: An Efficient Training and Inference Engine for Intelligent Mobile Cloud Computing Services. *IEEE Trans. on Mobile Computing* 20, 2 (2021), 565–576. https://doi.org/10.1109/TMC.2019.2947893

[5] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. 2023. DepGraph: Towards any structural pruning. In *Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*. 16091–16101.

[6] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *Int. Conf. on Learning Representations*.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conf. on Computer Vision and Pattern Recognition*. 770–778.

[8] Maedeh Hemmat, Joshua San Miguel, and Azadeh Davoodi. 2022. CAP'NN: A Class-Aware Framework for Personalized Neural Network Inference. *ACM Trans. Embed. Comput. Syst.* 21, 5, Article 59 (Dec 2022), 24 pages. https://doi.org/10.1145/3520126

[9] Jeremy Howard. 2019. ImageNette. Online. https://github.com/fastai/imagenette

[10] Zehao Huang and Naiyan Wang. 2018. Data-Driven Sparse Structure Selection for Deep Neural Networks. In *Proc. European Conf. on Computer Vision (ECCV)*. 304–320.

[11] Martin Lechner and Axel Jantsch. 2021. Blackthorn: Latency Estimation Framework for CNNs on Embedded Nvidia Platforms. *IEEE Access* 9 (2021), 110074–110084. https://doi.org/10.1109/ACCESS.2021.3101936

[12] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. 2019. Towards optimal structured CNN pruning via generative adversarial learning. In *Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*. 2790–2799.

[13] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. 2021. Machine Learning at the Network Edge: A Survey. *ACM Comput. Surv.* 54, 8, Article 170 (Oct 2021), 37 pages. https://doi.org/10.1145/3469029

[14] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *Inter. Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252.

[15] Gerardo Vilcamiza, Nicolas Trelles, Leonardo Vinces, and José Oliden. 2022. A coffee bean classifier system by roast quality using convolutional neural networks and computer vision implemented in an NVIDIA Jetson Nano. In *2022 Congreso Internacional de Innovación y Tendencias en Ingeniería (CONIITI)*. 1–6. https://doi.org/10.1109/CONIITI57704.2022.9953636

[16] Wenxiao Wang, Shuai Zhao, Minghao Chen, Jinming Hu, Deng Cai, and Haifeng Liu. 2019. DBP: Discrimination Based Block-Level Pruning for Deep Model Acceleration. arXiv:1912.10178 [cs.CV] arXiv:1912.10178[cs.CV].

[17] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 5987–5995. https://doi.org/10.1109/CVPR.2017.634

[18] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In *Proc. European Conf. on Computer Vision (ECCV)*. 285–300.