DEPTH PRUNING WITH AUXILIARY NETWORKS FOR TINYML

Josen Daniel De Leon[⋆†], *Rowel Atienza*[⋆]

[⋆]Electrical and Electronics Engineering Institute, University of the Philippines
[†]Samsung Research Philippines
{josen.daniel.de.leon, rowel}@eee.upd.edu.ph

ABSTRACT

Pruning is a neural network optimization technique that sacrifices accuracy in exchange for lower computational requirements. Pruning has been useful when working with extremely constrained environments in tinyML. Unfortunately, special hardware requirements and limited study on its effectiveness on already compact models prevent its wider adoption. Depth pruning is a form of pruning that requires no specialized hardware but suffers from a large accuracy falloff. To improve this, we propose a modification that utilizes a highly efficient auxiliary network as an effective interpreter of intermediate feature maps. Our results show a parameter reduction of 93% on the MLPerfTiny Visual Wakewords (VWW) task and 28% on the Keyword Spotting (KWS) task with accuracy cost of 0.65% and 1.06% respectively. When evaluated on a Cortex-M0 microcontroller, our proposed method reduces the VWW model size by 4.7× and latency by 1.6× while counter intuitively gaining 1% accuracy. KWS model size on Cortex-M0 was also reduced by 1.2× and latency by 1.2× at the cost of 2.21% accuracy.

Index Terms— pruning, optimization, tinyML

1. INTRODUCTION

There has been a recent growing interest to bring machine learning inference to commercial devices driven by the increasing consumer interest in privacy, energy efficiency and autonomy of edge devices. Market studies speculate that shipments of smart devices making use of these developments could grow from 15.2M in 2020 to 2.5B in 2030 [1].

New challenges have emerged from this movement towards more ubiquitous devices. These mainly stem from the extreme constraints on compute resources imposed by ultra-low power devices, typically under the mW range. As such, academe and industry leaders have established tinyML[2] as a field focused on the optimization of the different stages of on-device inference in extremely constrained environments. These may impose model sizes in the kilobytes (KB) range and operate with only MFLOPS of compute power.

We contribute to this space by exploring how effective depth pruning is on tinyML tasks. Depth pruning is a technique where entire layers are removed from the end of a trained model until a target model size is reached. We also introduce a modification to this method by using a small auxiliary network as a new head to improve accuracy with minimal overhead. This method is well structured and has no special hardware requirements. Our experiments show a parameter reduction of 93% with accuracy cost of 0.65% on the MLPerfTiny Visual Wakewords (VWW) task as shown in Figure 1. We validate our results by deploying the pruned Visual Wakewords (VWW) model on an inexpensive ($3 retail [3]) and widely used ARM Cortex-M0. This results to a 1.6× on-device inference speedup and a 4.7× smaller model size while counter intuitively gaining 1% in accuracy.

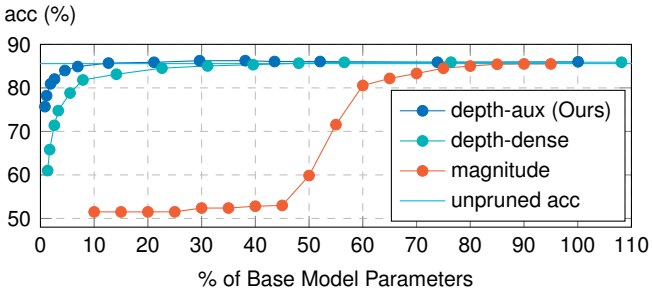

Fig. 1. Pruning Method Comparison on MobilenetV1 VWW task. Parameters are reduced by 93% for our method (frozen tail), 68.9% for depth pruning (dense), and 20% for magnitude pruning if maximum accuracy drop is set to 0.65%.

2. RELATED WORK

2.1. TinyML, Use Cases and Hardware

TinyML is an emerging field in machine learning with the goal of bringing machine learning to edge devices. Various benchmarks [4, 5] have been selected to represent tinyML applications and use cases. Among them is the Visual Wakewords (VWW) task which is a person presence detection problem using 115k train and 8k validation 96 × 96 RGB images [6]. Another is the Keyword Spotting (KWS) task which uses a dataset with 105,829 1sec word utterances such as

Platform	Processor	FLOPS	RAM
Desktop	RTX 3080Ti	34 T	128GB
Galaxy Note 20 (Mobile)	Exynos 990	1.1 T	8GB
RPi 4 Model B	Cortex-A72	48 G	4GB
Arducam Pico4ML	Cortex-M0	16 M	264KB

Table 1. Comparison of compute resources of Desktop, Mobile and Cortex Microcontrollers

"up", "down", "yes" and "no" to define a speech recognition problem with 12 classes [7, 8].

Microcontrollers provide an attractive and low cost hardware environment to achieve the goals of tinyML [8, 9]. Unfortunately, they also have significantly low compute capabilities as seen in Table 1. This influences neural network design for microcontrollers to become a multi-objective optimization problem [10] between performance and computate requirements. As such, methods which allow us to tweak models towards Pareto-optimal allocations between these metrics become important to tinyML research.

2.2. Network Pruning

One avenue of optimization in tinyML involves the modification of trained model parameters to reduce or simplify the computations during inference. Pruning [11, 12, 13] is one such method wherein unimportant parameters from a trained model are removed to reduce model size. It relies on a hypothesis that there exists a "winning lottery ticket" subnetwork [14, 15] that allows us to reach the same accuracy as the original network using only a subset of its parameters. Despite progress in this research, identifying an effective pruning scheme for a specific model can still be considered as an unstructured process requiring iteration and intuition.

Among pruning schemes that aim to identify this subnetwork, magnitude pruning[13, 16] can be seen as the most popular . This method involves using a parameter's magnitude as a heuristic to determine which weights can be zeroed out to produce a sparse model. Sparse matrix computation and compression then lowers compute requirements during inference. Unfortunately, this feature is rarely found in microcontrollers limiting the adoption of pruning in commercial applications.

2.3. Depth Pruning

Depth pruning is a form of pruning where layers at the end of a trained model are removed to reduce model size. We define depth pruning as a technique that generates a subnetwork from a trained base model with N layers. We retain the first D layers from the base model and attach a new classifier head, typically a single dense layer, after D.

A prior work in intermediate feature map interpretability [17] shows that this method is able to produce an accuracy curve that is monotonically increasing with model depth. This property, along with most neural networks designs having parameters concentrated at deeper layers, gives us an easy-to-use and interpretable knob to adjust the trade-off between accuracy and compute requirements.

While this approach shares similarities with transfer learning, differences lie in their objectives. Whereas transfer learning uses stored knowledge on a certain task to improve performance on a different downstream task, depth pruning uses it to produce smaller models on the same task. In addition, transfer learning usually removes only the final layer while depth pruning aims to discard as many as possible.

While depth pruning has been shown to function well with Neural Architecture Search [18] and requires no special hardware, it is seldom used as a standalone solution due to the large falloff in accuracy. We hypothesize that this can be improved by using a highly efficient auxiliary network instead of just a single dense layer.

3. DEPTH PRUNING WITH AUXILIARY NETWORKS

Our work improves the accuracy of models produced by depth pruning through the use of an auxiliary network as the new layer head as illustrated in Figure 2. This acts as a powerful interpreter of intermediate feature maps from the trained layers to the output labels. Furthermore, this auxiliary network incurs minimal overhead and is simple to train and apply. Model training and pruning code is located at https://github.com/jd-deleon/depth-pruning-auxnets

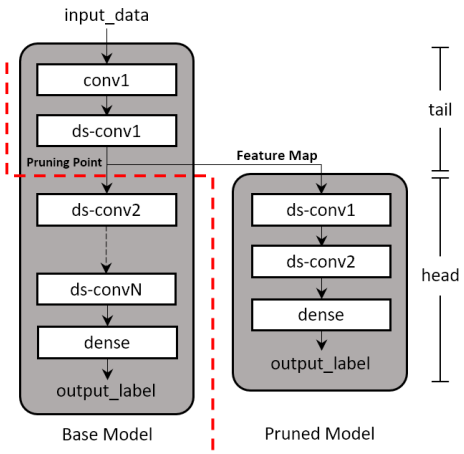

Fig. 2. Depth Pruning with Auxiliary Networks. Our pruning method applied to an arbitrary trained model.

Table 2 outlines the architecture of the auxiliary network selected for this study. We make use of depthwise separable convolutions which have been shown to be highly efficient building blocks for compact neural networks [19]. Hyperparameters are selected such that the auxiliary network incurs minimal overhead while still being able to preserve accuracy.

Type / Stride	Filter Shape	Parameters
dw-Conv / s1	3×3×[64] dw	[896]
pw-Conv / s1	1×1×[64]×32	[2,208]
dw-Conv / s1	3×3×32 dw	[448]
pw-Conv / s1	1×1×32×16	[592]
globalAvePool	-	0
dense	32×[2]	[34]

Table 2. Auxiliary Network Architecture. Bracketed values are dependent on the input shape. Presented values are with a 12×12×64 tensor from MobilenetV1-VWW block5.

We make use of ReLU activation and batch normalization after each convolution to stabilize training.

Algorithm 1 outlines how our pruning method is conducted in a trained model. In summary, we take the first p layers of our trained model W as our tail and freeze it as an optional step. We then attach our auxiliary network as the new head and train it to produce the pruned model.

Algorithm 1 Depth Pruning with Auxiliary Networks

Require: W, the trained model layers, p, the pruning point, and X, the dataset on which to finetune

$T \leftarrow W[0:p]$
$T \leftarrow freeze(T)$
$A \leftarrow initializeAux()$
$W' \leftarrow append(T, A)$
$W' \leftarrow trainToConvergence(f(X; W'))$
return W'

Freezing the model tail is an optional step which produces a pruned model having shared weights with the base network. This can be used in dynamic architectures wherein a smaller network can signal the activation of a larger high performance branch. We can also let the tail remain unfrozen allowing the model to converge to a better accuracy.

We retain all the data preprocessing, optimizer settings, loss functions and labelling conventions used to train the base network during the pruning process. This takes advantage of any prior knowledge already learned by the tail of the model. While training the auxiliary network usually takes only a few epochs, we train for the same number of epochs as in base model training to ensure convergence in the reported data.

4. EXPERIMENTS AND RESULTS

We use TensorFlow2 [20] model definitions and code samples in the MLPerfTiny benchmarking suite [4] to train and evaluate the base networks for both VWW and KWS tasks. We use step learning rates $[0.001, 0.0005, 0.00025]$ for $[20, 10, 20]$ epochs on VWW and $[0.0005, 0.0001, 0.00002]$ for $[12, 12, 6]$ epochs on KWS in the Adam [21] optimizer. All figures include overheads from the auxiliary network which may result to some configurations exceeding base model properties.

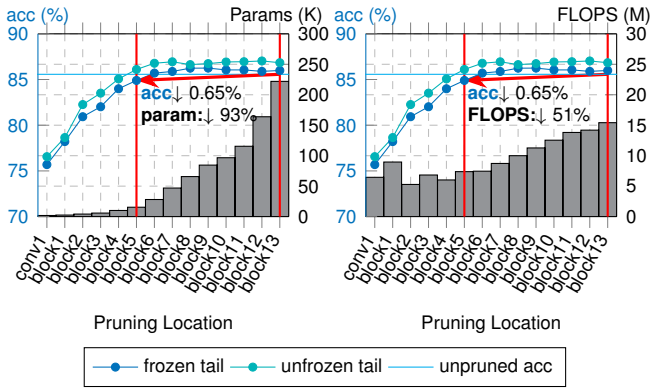

Fig. 3. Depth Pruning on MobilenetV1 VWW task. Pruning at block5 reduces parameters by 93% and FLOPS by 51% at a cost of 0.65% accuracy.

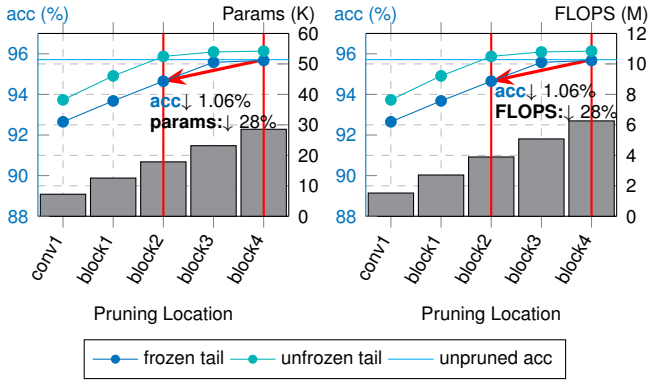

Fig. 4. Depth Pruning on DSCNN KWS task. Pruning at block2 reduces parameters by 28% and FLOPS by 28% at a cost of 1.06% accuracy.

4.1. Depth Pruning on tinyML Tasks

From our VWW results in Figure 3, we observe that our method can significantly reduce operational requirements with minimal cost to accuracy. We also notice that some pruning locations are able to produce smaller models with higher accuracy than the base model. This may be attributed to overparametrization in the base model leading to overfitting which is reduced by our pruning process.

Our results with the much smaller models for the KWS task in Figure 4 is able to yield less improvement due to higher information density in compact models. In spite of this, we are still able to produce smaller models with minimal loss in accuracy. Unfreezing our tail is able to produce models with higher accuracy on both tasks. Latency measurements were conducted on an Intel-i9 CPU but produced no noticeable improvement from 5ms due to the already compact model sizes prior to pruning.

We compare our work to other pruning methods on VWW

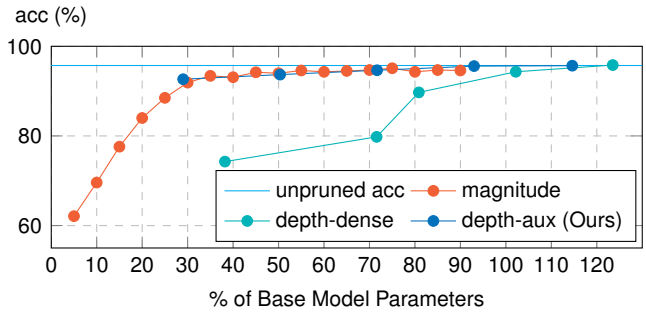

Fig. 5. Pruning Method Comparison on DSCNN KWS task. Parameters are reduced by 28% for our method (frozen tail) and 30% for magnitude pruning if maximum accuracy drop is set to 1.06%. Shallow networks limit pruning locations.

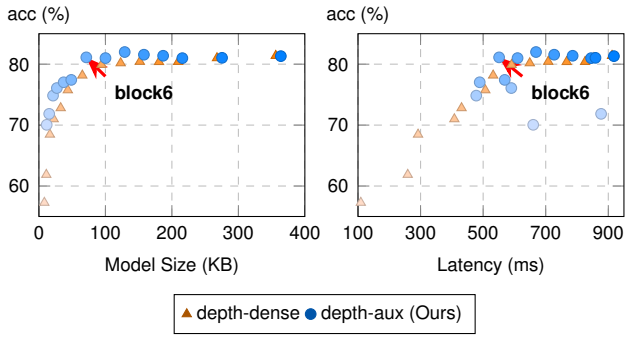

Fig. 6. VWW Depth Pruning on Cortex-M0. Darker marks denote deeper pruning points. Pruning at block6 of MobilenetV1-VWW reduces model size from 336KB to 71KB and latency from 904ms to 551ms. Accuracy also counterintuitively increases from 80% to 81%.

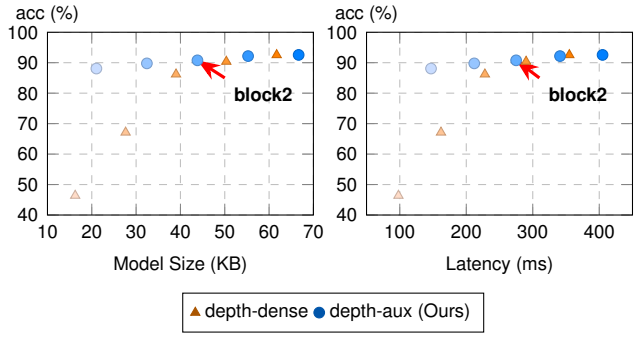

Fig. 7. KWS Depth Pruning Results on Cortex-M0. Darker marks denote deeper pruning points. Pruning at block2 of DSCNN reduces model size from 54KB to 43KB and latency from 340ms to 275ms at a cost of 2.21% accuracy.

and KWS by visualizing the Pareto frontier in Figure 1 and Figure 5 respectively. In both tasks, we are able to improve upon depth pruning with two dense layers using filter counts of $[64, 32]$. On VWW, our method performs significantly better than global magnitude pruning with constant sparsity implemented using the TensorFlow Optimization Toolkit. Despite being at par with magnitude pruning on the KWS task, our method does not require sparse matrix support making it a better choice for on-device inference.

While our method has been shown to work on tinyML models, we believe it can be extended to architectures with a generally linear design such as plain ConvNets and ResNets. This is supported by work in intermediate layer analysis[17] where a dense probe shows an increasing trend in separability in linear networks.

4.2. Hardware Experiments

We evaluate our work by deploying our pruned models on the Arducam Pico4ML running an ARM Cortex-M0 [22] processor. Due to device limitations, the MobilenetV1 model was retrained using grayscale inputs. TFLite-Micro [23] was used as the on-device inference framework and integer quantization was applied after pruning.

As shown in Figure 6, our approach using frozen tail training is able to produce a `tflite` model that is $4.7\times$ smaller in size which includes both parameters and architecture metadata. On-device inference latency is $1.6\times$ faster on pruned models with a slight accuracy boost of 1%. Pruning at block6 also produces a model that has 3% higher accuracy compared to depth pruning using dense layers with a small size increase of 7KB. Note that magnitude pruning cannot be deployed on-device due to lack of sparse matrix support.

We also validate our results on the KWS task with the DSCNN model [8] on the Cortex-M0. We first retrain the base model using Log Mel-filterbank Energies (LFBE) instead of Mel-frequency cepstral coefficients (MFCC) as the input preprocessing in order to use optimized on-device libraries. As shown in Figure 7, our results improve model size by by $1.2\times$ and latency by $1.2\times$ at the cost of 2.21% accuracy when pruned at block2.

5. CONCLUSIONS AND FUTURE WORK

In this paper we introduce our depth pruning method which uses an auxiliary network as a new head of the pruned model. This technique is easily interpretable and requires no special hardware support during inference. Results show that we are able to significantly reduce model size with minimal accuracy loss in tinyML tasks. On-device experiments validate our work with significant speedup and reduction to memory footprint which translates to energy savings in IoT devices.

These results open up new avenues for optimization to satisfy extreme hardware constraints. In future work, we aim to apply this technique to dynamic architectures opening up new opportunities for efficient on-device inference.

6. REFERENCES

[1] ABI Research, "TinyML : The next big opportunity in tech," Tech. Rep., 2021.

[2] TinyML Foundation, "TinyML," https://www.tinyml.org/. (accessed: 08.24.2021).

[3] Mouser, "Cortex-M0 retail pricing," https://www.mouser.com. (accessed: 10.07.2021).

[4] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al., "Mlperf tiny benchmark," *arXiv preprint arXiv:2106.07597*, 2021.

[5] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al., "Benchmarking tinyml systems: Challenges and direction," *arXiv preprint arXiv:2003.04821*, 2020.

[6] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes, "Visual wake words dataset," *arXiv preprint arXiv:1906.05721*, 2019.

[7] Pete Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[8] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.

[9] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, and Urmish Thakker et al., "MicroNets: Neural network architectures for deploying TinyML applications on commodity microcontrollers," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[10] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough, "SpArSe: Sparse architecture search for cnns on resource-constrained microcontrollers," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[11] Yann LeCun, John Denker, and Sara Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed. 1990, vol. 2, Morgan-Kaufmann.

[12] Song Han, Jeff Pool, John Tran, and William J Dally, "Learning both weights and connections for efficient neural network," in *Neural Information Processing Systems*, 2015.

[13] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag, "What is the state of neural network pruning?," *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.

[14] Jonathan Frankle and Michael Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations*, 2019.

[15] Ibrahim Alabdulmohsin, Larisa Markeeva, Daniel Keysers, and Ilya Tolstikhin, "A generalized lottery ticket hypothesis," *arXiv preprint arXiv:2107.06825*, 2021.

[16] Michael H Zhu and Suyog Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," in *International Conference on Learning Representations*, 2018.

[17] Guillaume Alain and Yoshua Bengio, "Understanding intermediate layers using linear classifier probes," in *International Conference on Learning Representations*, 2017.

[18] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han, "Once-for-all: Train one network and specialize it for efficient deployment," in *International Conference on Learning Representations*, 2020.

[19] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[20] Google, "TensorFlow2," https://www.tensorflow.org/. (accessed: 08.24.2021).

[21] Diederik P Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[22] ARM, "Cortex-M0," https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0. (accessed: 10.01.2021).

[23] Google, "TensorFlowLite Micro," https://www.tensorflow.org/lite/microcontrollers. (accessed: 10.5.2021).