DISSECTING EFFICIENT CONVOLUTION NETWORKS FOR WAKE WORD DETECTION

Cody Berger, Juncheng B Li, Karthik Ganesan, Aaron Berger, Dmitri Berger

Carnegie Mellon University

ABSTRACT

Wake-word detection models running on edge devices have stringent requirements for efficiency besides the requirements for accuracy. [1] We observe the over-the-air test accuracy of trained models on parallel devices (GPU/TPU) usually degrades when deployed on an edge device that only has a CPU for over-the-air real-time evaluation [2], and the inference time of different models expands differently when migrating from GPU to CPU. Such accuracy drop is due to hardware latency and acoustic impulse response, while the nonuniform expansion of inference time is due to different neural architectures' varying ability to take advantage of hardware acceleration for matrix multiplication across various platforms. Although many neural architectures have been applied to wake-word detection tasks, the aforementioned latency or accuracy drops have never been broken down to matrix multiplication levels and thoroughly studied. In this paper, we compare 6 CNN-variant architectures that were optimized for edge deployment, train them for the wake-word detection task on the Speech Command dataset [3] and seek to accurately quantify the trade-off between their accuracy, efficiency, and over-the-air robustness. Our findings should be directly useful to practitioners and could inform researchers about the key components in CNN architectures that influence the trade-off.

Index Terms— wake-word detection, neural architecture, edge computing

1. INTRODUCTION

Many neural architectures have been successfully applied to the wake-word detection task [4, 5, 6, 2, 7], with many existing works noting the trade-off between accuracy, efficiency, and over-the-air robustness. This trade-off has critical relevance for developers. Wake-word detection models are liable to perform differently across devices and operating systems, making it challenging to identify the best-performing model given a specific platform. Without clarity on how architecture performance varies across different platforms, developers are in the dark when attempting to select the best wake-word model for their specific task.

Prior works such as [2, 7] pointed out the efficiency advantage of Convolutional Neural Network (CNN) variants over other architectures such as DNN, RNN or Transformers. In pursuit of low latency, we focus on studying CNN variants models in this paper. Within the CNN-family, the difference in their efficiency VS. accuracy VS. over-the-air robustness is not clearly quantified. On top of this, there are new developments of CNN such as [8, 9] that were specifically optimized for efficiency. Without a fine-grained comparison down to the matrix multiplication (tensor operation) level, it is difficult to identify the key ingredient of a "winning" architecture that strikes a balance between this 3-way tradeoff.

To shed light on this issue, we compare 6 different efficient CNN architectures and their corresponding performances on the wake-word task between multi-threaded and single-threaded devices. In keeping with the theme of practical relevance, we completed real-time accuracy and efficiency testing for all models to pinpoint areas where performance diverged across platforms. We identify specific factors which may harm model efficiency on CPU as compared to GPU.

2. RELATED WORKS

Wake Word Detection: [2] improved upon the baseline for wake-word detection task set up by [4], where they empirically showed the efficiency advantages of CNN-family model with depth-wise connection: DS-CNN (MobileNet[10] equivalent). Concurrent works such as [5, 6] explored other types of CNNs that involve dilation and also acknowledged CNN's inherent efficiency. [7] shows that Transformers achieve 1 percent better accuracy for a trade-off of 4x increase in latency showing that CNN-family models have a much superior trade-off for efficiency vs accuracy.

Efficient Architectures: Following MobileNet [10], this line of research saw huge developments benefiting from Network Architecture Search that injects efficiency metrics into the training objective. MobileNet-V2 [11], mnasNet [12] EfficientNet [8] and Efficient-v2 [9] demonstrated ever improving efficiency accuracy tradeoff. However, existing wake-word literature has not followed up on this trend yet. In this work, we will dive into these efficient CNN architectures and identify the best candidate for wake-word detection task.

3. EXPERIMENT

3.1. Dataset and Setup

We trained 8 wake-word architectures on wake-word audio samples from the Google Speech Commands database, selecting 'Sheila' as the wake-word. For both EfficientNet and EfficientNetV2, two models with different scaling were trained (in order to cover the full spectrum of model sizes). For all other architectures, only one model was trained per architecture. All models were uniformly trained for 75 epochs and 5 runs, optimizing for stochastic gradient descent. Each model was trained with batch size 64, learning rate $1e^{-4}$ and weight decay $1e^{-2}$. Four dataload workers were used for each training. A plateau learning rate scheduler was used with patience 5 and gamma 0.5. All architectures were non-streaming, meaning that they take the entirety of an input and classify it at once.[2]

Digital testing analyzed the models' performance on 807 audio files, of which 187 contained the wake-word and 620 did not.

Over-the-air Test We performed two sets of over-the-air experiments, one set (Experiment 1) analyzing model performance and one set (Experiment 2) with hooks embedded into the models to analyze the runtime of individual layers in each model. Three subjects conducted Experiment 1, two with lower voices, and one with a higher voice. Two subjects, both with lower voices, conducted Experiment 2. During over-the-air testing for both Experiments 1 and 2, each tester completed five trials on a GPU device and five trials on a CPU device for each model. All trials took place at the same location in the same room, ensuring a consistent room impulse response across trials. Each trial encompassed 20 seconds of real-time audio recording, during which testers uttered the phrase 'Sheila is the wake-word' ten times. Recorded audio was sent to the model in complete 0.5 second chunks to be analyzed, taking a non-streaming approach. After processing an audio chunk, 1 or 0 would be printed respectively depending on whether the model detected the wake-word or not. Testers individually determined the correctness for each of the model's predictions. To measure each layer's runtime during over-the-air testing in Experiment 2, forward hooks were added. The hooks performed a depth-first-search on the graph of a model, measuring runtime for each leaf node layer. Leaf node layers and their runtimes were logged and exported to a csv for each trial.

FLOP Count We measured the floating point operations per second (FLOP) and parameter count for each model using the THOP library's profiler.

Table 1. Experiment 1: Model Performance on Speech Commands Dataset

M1 MacBook Pro	Digital Test F1	Over-Air Test F1	Latency (s)	Parameters
DSCNN	92.15%	93.78%	0.0376	2.23E+06
efficientnet_b1	92.75%	93.16%	0.0855	6.46E+04
efficientnet_b7	93.87%	96.55%	0.1460	3.16E+05
effnetv2_m	92.88%	92.14%	0.0940	5.35E+07
effnetv2_xl	92.64%	94.06%	0.1463	2.07E+08
densenet_bc_190_40	93.37%	94.85%	0.3442	2.56E+07
mnasnet1_0	91.58%	82.59%	0.0353	3.10E+06
CNN	93.37%	96.69%	0.0173	3.89E+07
Raspberry Pi 4B	**Digital F1**	**Over-Air Test F1**	**Latency(s)**	**Parameters**
DSCNN	92.15%	90.28%	0.1507	2.23E+06
efficientnet_b1	92.75%	89.83%	0.2826	6.46E+04
efficientnet_b7	93.87%	89.20%	1.0435	3.16E+05
effnetv2_m	92.88%	93.45%	0.8079	5.35E+07
effnetv2_xl	92.64%	71.85%	1.6734	2.07E+08
densenet_bc_190_40	93.37%	42.22%	3.6742	2.56E+07
mnasnet1_0	91.58%	82.37%	0.1311	3.10E+06
CNN	93.37%	90.80%	0.3723	3.89E+07

3.2. Models

Table 1 details the results from Experiment 1, which analyzed digital vs. over-the-air performance for all eight trained models on a GPU device and on a CPU device. Parameters were calculated for each model using the THOP library. Digital testing analyzed the models' performance on 807 audio files from the Speech Commands dataset, of which 187 contained the wake-word and 620 did not. Digital Test F1 reflects the F1 score calculated from comparing the models' predictions for whether or not an input contained the wake-word versus whether that input actually contained the wake-word or not.

During over-the-air testing, three subjects completed five trials for each architecture on a parallel platform and five trials for each architecture on a CPU platform. Audio was recorded in 1-second increments during the course of each real-time trial, yielding a total of 20 seconds of audio per trial. As we took a non-streaming approach, each 1-second chunk of audio would be downsampled immediately after it was recorded, converted into a mel spectrogram, and sent as input to the model to be classified in its entirety. Following classification, the model's prediction for whether the wake-word was present in that 1-second audio chunk would be printed to the console. Subjects uttered the phrase 'Sheila is the wake-word' ten times during each trial, manually determining the prediction accuracy for each model output. For each trial, subjects recorded the overall number of true positives, false positives, true negatives, and false negatives that occurred. This data was used to compute an the average over-air F1 score for each model, according to the formula $\frac{1}{3}\sum_{s=1}^{3}\frac{1}{5}\sum_{t=1}^{5}\frac{TP_t}{TP_t+\frac{1}{2}(FP_t+FN_t)}$ where s is an unique subject and t is an unique trial. For each trial, latency was calculated as the time between when a model received input and when its corresponding prediction was returned.

Table 2. Experiment 2: Runtime of Specific Layer Classes on Speech Commands Dataset
Average time measures the time it took for a layer class to run on average across all trials. Average FLOPS measures the average floating point operations per second for a layer class across all trials. Percentage of aggregate runtime (PAR) measures the relative amount of computation time that was spent running layers of a certain class.

	DSCNN						**efficientnet_b1**					
	Average Time (ms)		Average FLOPS		PAR		Average Time (ms)		Average FLOPS		PAR	
	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi
Conv2d	0.236	1.411	2.40E+05	2.30E+05	39.53%	53.09%	0.340	1.348	2.13E+03	2.13E+03	67.38%	60.05%
BatchNorm2d	0.252	1.149	1.19E+04	7.70E+03	43.23%	43.22%	0.220	1.437	1.15E+04	1.15E+04	26.25%	38.43%
ReLU	0.151	0.134	0	0	17.03%	3.40%	-	-	-	-	-	-
MaxPool2d	-	-	-	-	-	-	-	-	-	-	-	-
Linear	0.347	0.249	5.12E+03	5.12E+03	1.12%	0.18%	0.300	0.139	5.12E+03	5.12E+03	0.52%	0.05%
Dropout	0.029	0.137	0	0	0.09%	0.10%	0.020	0.072	0	0	0.03%	0.03%
AveragePool2d	-	-	-	-	-	-	0.141	0.155	2.13E+02	2.13E+02	5.82%	1.44%
	efficientnet_b7						**effnetv2_m**					
	Average Time (ms)		Average FLOPS		PAR		Average Time (ms)		Average FLOPS		PAR	
	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi
Conv2d	0.240	2.478	?	?	68.29%	71.06%	0.189	2.865	1.37E+06	1.37E+06	38.45%	60.52%
BatchNorm2d	0.177	1.621	4.26E+03	4.26E+03	30.03%	27.77%	0.201	1.570	1.38E+04	1.38E+04	41.02%	33.18%
ReLU	-	-	-	-	-	-	-	-	-	-	-	-
MaxPool2d	-	-	-	-	-	-	-	-	-	-	-	-
Linear	0.347	0.592	5.12E+03	5.12E+03	0.36%	0.06%	0.121	0.434	2.30E+05	2.30E+05	13.75%	5.11%
Dropout	0.022	0.068	0	0	0.02%	0.01%	-	-	-	-	-	-
AveragePool2d	0.022	0.188	9.14E+01	9.14E+01	1.29%	1.11%	0.118	0.201	1.22E+04	1.22E+04	6.76%	1.19%
	effnetv2_xl						**densenet_bc_190_40**					
	Average Time (ms)		Average FLOPS		PAR		Average Time (ms)		Average FLOPS		PAR	
	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi
Conv2d	0.149	3.921	2.58E+06	2.58E+06	36.28%	69.63%	0.130	10.523	7.66E+07	1.92E+07	32.68%	80.58%
BatchNorm2d	0.175	1.277	1.79E+04	1.79E+04	42.42%	22.20%	0.190	1.731	1.58E+06	4.51E+05	47.89%	12.82%
ReLU	-	-	-	-	-	-	0.074	0.816	0	0	18.48%	6.07%
MaxPool2d	-	-	-	-	-	-	-	-	-	-	-	-
Linear	0.104	0.687	4.94E+05	4.94E+05	14.47%	7.09%	0.267	0.254	8.76E+03	8.76E+03	0.47%	0.04%
Dropout	-	-	-	-	-	-	-	-	-	-	-	-
AveragePool2d	0.098	0.216	1.53E+04	1.53E+04	6.83%	1.12%	0.58	3.283	4.38E+03	4.38E+03	0.77%	0.49%
	mnasnet1_0						**CNN**					
	Average Time (ms)		Average FLOPS		PAR		Average Time (ms)		Average FLOPS		PAR	
	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi	Mac	Pi
Conv2d	0.197	1.160	2.40E+05	2.40E+05	39.41%	53.18%	0.369	12.938	3.90E+07	3.90E+07	37.56%	68.67%
BatchNorm2d	0.228	0.943	1.19E+04	1.19E+04	45.68%	43.14%	0.334	3.656	1.39E+05	1.39E+05	34.07%	19.41%
ReLU	0.096	0.094	0	0	12.90%	2.95%	0.136	0.543	0	0	15.53%	3.24%
MaxPool2d	-	-	-	-	-	-	0.235	3.345	0	0	7.49%	5.55%
Linear	0.294	0.322	5.12E+03	5.12E+03	1.19%	0.30%	0.274	24.692	1.26E+07	1.26E+07	0.95%	24.58%
Dropout	0.202	0.473	0	0	0.81%	0.44%	0.017	0.215	0	0	0.21%	0.06%
AveragePool2d	-	-	-	-	-	-	-	-	-	-	-	-

During experiment 2, subjects conducted five over-air trials on a GPU device and five over-air trials on a CPU device for each model. As in experiment 1, 20 seconds of audio was recorded during each trial and 1-second increments of audio were individually passed in to a model. Also consistent with experiment 1, subjects repeated the statement 'Sheila is the wake-word' ten times during each trial and monitored accuracy of the models' predictions. In addition to these procedures, forward hooks were inserted into the models using depth first search to identify each time a 'leaf node layer' ran. Here, we define 'leaf node layer' as a layer which only has one edge in the graph of the model. In particular, whenever a leaf node layer ran, the hooks logged that layer and its runtime. Finally, the THOP library was used to calculate FLOPS for each model's leaf node layers.

4. OBSERVATION

Most of the architectures displayed no performance drop from digital testing to over-the-air testing. GPU over-the-air testing in fact revealed a slight increase in experimental F1 score relative to digital F1 for most other models run on GPU. However, MnasNet's performance degraded severely between digital testing versus over-the-air testing. Digital testing yielded an F1 score of 91.58%, whereas over-the-air testing on devices using GPU and CPU yielded much lower F1 scores of 82.59% and 82.37%. where This suggests that while most of the models were able to handle the audio transformations generated by room-impulse-response, MnasNet may have struggled.

The CNN and EfficientNet_b7 achieved the first and second highest F1 scores respectively amongst the architectures during over-the-air GPU testing, surpassing the heftier, more parameter-heavy DenseNet (densenet_bc_190_40) and EfficientNetV2_xl (effnetv2_xl) models by about two percentage points.

Moving from GPU to CPU, over-the-air performance dropped for many architectures. All models except for EfficientNetV2_m (effnetv2_m) saw their F1 scores slip below the digital baseline, the majority settling around an F1 score of 90%. This finding is consistent with previous scholarship [2].

The DenseNet and EfficientNetV2_xl models in particular showed a steep decline in performance. This performance cliff was accompanied by large increases in latency for both models. DenseNet's latency jumped from a slightly sluggish 0.3442 seconds on GPU to a whopping 3.6742 seconds on CPU, in conjunction with a drop to the extremely low F1 score of 42.22%. Meanwhile, EfficientNetV2_xls speedy 0.1463 second latency on GPU increased to 1.6734 seconds, accompanied by a drop in F1 score from an excellent 94.08% on GPU to just 71.85% on CPU.

Both of these architectures are parameter heavy, but parameter count alone cannot explain their drastic performance decrease. For example, EfficientNetV2_m and CNN architectures did not display anything close to DenseNet's dramatic slowdown despite having more parameters.

Latency does not provide a clear window into the cause behind performance decrease on CPU either. Efficientnet_b7 also suffered a slowdown on CPU, latency increasing from 0.1460 seconds to 1.0435 seconds. However, its F1 score did not experience a similarly sharp dip, instead floating down to 89.20%. This shallow decrease in F1 score would seem at home among speedier architectures such as DSCNN and the lighter-weight EfficientNet_b1, which saw their latencies hover between 0.1 and 0.3 seconds on CPU.

In order to explain these discrepancies, we drill into the fine-grained components of each architecture and analyze the FLOPS.

5. FURTHER ANALYSIS AND DISCUSSION

Taking one step further, we analyze the individual network components in Table 2.

Convolution layers in particular often take longer to compute on CPU as opposed to GPU. This divergence stems from hardware optimization differences for matrix multiplication.

Hardware optimization has such an effect on the efficiency of convolution layers because convolution is fundamentally matrix multiplication. Convolution is computed as follows:

Let x be the input matrix of dimensions $h \times w \times c_{in}$, where h is image height, w is image width, and c_{in} is the number of input channels. Let W be a matrix of weights with dimensions $c_{in} \times c_{out} \times k \times k$ where c_{out} is the number of output channels and k is the kernel size. For each input channel, these matrices x and W are multiplied together. The sum of these multiplications yields an output matrix z of dimensions $h \times w \times c_{out}$. The exact formula is shown below, where r represents a specific input channel and s represents a specific output channel:

$$z[:, :, s] = \sum_{r=1}^{c_{in}} x[:, :, r] * W[r, s, :, :]$$

For the wake-word detection task, only a forward pass is necessary. Weights have already been trained in the forward pass, so we do not need to worry about the complexity of differentiating.

CPUs are restricted in their ability to parallelize by their number of cores and caches. GPUs on the other hand boast a much larger array of cores and caches, allowing them to accelerate processes that can be parallelized. Matrix multiplication is one such process. Matrix multiplication can be broken down via tiling, enabling one large matrix multiplication to be calculated as a series of much smaller ones. With tiling techniques, matrix accesses and operations can be broken down

into parallelizable chunks and costly memory accesses can be avoided. The GPU can then take advantage of this parallel character to significantly speed up matrix multiplication.

Pytorch stores matrices in memory using the strides format to decrease the cost of accessing data within them. With this storage method and acceleration, the load cost of matrix multiplication becomes $\mathrm{l1speed} * (\frac{n^3}{v2} + \frac{n^3}{v1}) + \mathrm{dramspeed} * (n^2 + \frac{n^3}{b1})$ where $\mathrm{l1speed}$ is the speed of accessing the l1cache, $\mathrm{dramspeed}$ is the speed of accessing the DRAM, n is ???, $v1$ is the cost of accessing a row, $v2$ is, and $b1$ is

Reword our understanding, take their complexity analysis and match up with our FLOP and time reported in table 2

In Pytorch, convolution is implemented using the im2col operation which

Discuss why xxx architectures that relied on convnets a lot got penalty moving to CPU, and explain the trade off. Densenet

Convolution receives significant acceleration on GPU devices. As CPUs cannot take full advantage of parallelism to accelerate matrix multiplication, convolution is much slower on CPU devices. Without understanding how GPUs accelerate convolution, speedy performance of convolution-reliant wake-word architectures on GPU can obscure the degree to which those same models are liable to slow down on CPU.

Table 2 analyzed percentage of aggregate runtime (PAR), the relative amount of time a model spent computing each type of layer across trials. For most models, there was a noticeable increase from GPU to CPU in the share of time taken up by convolution layers. In particular, DenseNet's PAR for convolution layers jumped from 32.68% on GPU to 80.58% on CPU, suggesting that convolution layers are the culprit behind its drastic slowdown.

point out efficientNet was searched on ImageNet but not audio, might be flawed.

Come up with caveats for practitioners if they know their device was going to be CPU only, should not rely on matrix-multiplication heavy architectures, but to pick maybe more light-weight models for better trade off.

6. CONCLUSION

In this work, we compared the performance 6 of different efficient neural architectures for the wake-word detection task. We observed that the best-performing architectures on GPU vs CPU did not always line up perfectly, and that architectures overly reliant on expensive convolution operations especially tended to take a blow to efficiency on CPU. We recommend that practicioners who need to use wake-word detection on CPU devices choose models which are less heavily reliant on convolution for the best efficiency.

7. REFERENCES

[1] Siddharth Sigtia, John Bridle, Hywel Richards, Pascal Clark, Erik Marchi, and Vineet Garg, "Progressive voice trigger detection: Accuracy vs latency," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 6843–6847.

[2] Oleg Rybakov, Natasha Kononenko, Niranjan Subrahmanya, Mirkó Visontai, and Stella Laurenzo, "Streaming keyword spotting on mobile devices," *Proc. Interspeech 2020*, pp. 2277–2281, 2020.

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

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

[5] Minhua Wu, Sankaran Panchapagesan, Ming Sun, Jiacheng Gu, Ryan Thomas, Shiv Naga Prasad Vitaladevuni, Bjorn Hoffmeister, and Arindam Mandal, "Monophone-based background modeling for two-stage on-device wake word detection," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 5494–5498.

[6] Alice Coucke, Mohammed Chlieh, Thibault Gisselbrecht, David Leroy, Mathieu Poumeyrol, and Thibaut Lavril, "Efficient keyword spotting using dilated convolutions and gating," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 6351–6355.

[7] Axel Berg, Mark O'Connor, and Miguel Tairum Cruz, "Keyword transformer: A self-attention model for keyword spotting," in *Interspeech 2021*. ISCA, 2021, pp. 4249–4253.

[8] Mingxing Tan and Quoc Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[9] Mingxing Tan and Quoc Le, "Efficientnetv2: Smaller models and faster training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 10096–10106.

[10] 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.

[11] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[12] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[13] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

A. APPENDIX

We broke model leaf node layers into seven categories across the six examined architectures: Conv2d, BatchNorm2d, ReLu, MaxPool2d, Linear, Dropout, and AveragePool2d. For each trial, a model's leaf node layers were first sorted into these categories regardless of parameters. The aggregate runtime and run count were calculated for each category by summing the individual runtimes of leaf node layers in the same category, and by counting every time a leaf node layer of a specific category was run. Finally, aggregate FLOPS per category were calculated using the results from the THOP library, summing the FLOPS per leaf node layer for leaf node layers in the same category. After finding aggregate runtime, run count, and FLOPS per category for each trial, these values were averaged across all trials, as summarized by the following equations, where t is trials, n_t is the nth (and last) leaf node layer to run in trial t, c is a layer category, r_c is average runtime for category c, k_c is average run count for category c, and m_c is average FLOPS for category c:

$$r_c = \frac{1}{5}\sum_{t=1}^{5}\sum_{l=1}^{n_t}\text{runtime}(l)\{l \in C\}$$

$$k_c = \frac{1}{5}\sum_{t=1}^{5}\sum_{l=1}^{n_t}\{l \in C\}$$

$$m_c = \frac{1}{5}\sum_{t=1}^{5}\sum_{l=1}^{n_t}\text{FLOPS}(l)\{l \in C\}$$

A.1. Additional Model descriptions

Depthwise Seperable Convolutional Neural Network (DSCNN) [11] DSCNN consists of MBConvReLU blocks where each MBConvReLU block contains one pointwise MBConvReLU block, one depthwise MBConvReLU block, and one pointwise linear convolution layer. The DSCNN structue as a whole begins with a stride-2 convolution layer, goes into a sequence of inverted residual bottleneck layers, and exits into a stride-1 convolution layer and final linear layer.

DenseNet [13] Densenet consists of 1x1 bottleneck convolution block is formed by a 1x1 bottleneck convolution layer with stride 1 sandwiched between two batch normalizations, followed by a 3x3 convolution layer of stride 1 and finally one last batch normalization and a ReLU layer . Transition convolution blocks are formed by a batch normalization - ReLU - 1x1 stride 1 convolution - average pooling sequence. The alternating bottleneck and transition layers are preceded by a 3x3 convolution layer with stride 1, and succeeded by a linear layer.

Convolutional Neural Network (CNN) [4] We utilizing a repeating sequence of stride 1 convolution, batch normalization, and ReLU layers followed by three linear layers.

MobileNet (DSCNN) [10] MobileNet was designed with efficiency in mind, as an answer to the practical challenge of implementing neural networks on platforms with less computation power. It is structured by blocks consisting of one depthwise convolution and one pointwise convolution, each followed by a batch normalization and ReLU layer.

Depthwise Seperable Convolutional Neural Network (DSCNN) [11] Our DSCNN was MobileNetV2, which leverages inverted residual layers connecting depthwise seperable convolutions and bottleneck convolutions. Inverted residual blocks are made up of MBConvReLU blocks similar to the implementation in MobileNetV1.

DenseNet [13] Densenet consists of 1x1 bottleneck convolution block formed by a 1x1 bottleneck convolution layer with batch normalization followed by a 3x3 convolution layer of stride 1 with ReLU activation . Transition convolution blocks are formed by a batch normalization - ReLU - 1x1 stride 1 convolution - average pooling sequence. The alternating bottleneck and transition layers are preceded by a 3x3 convolution layer with stride 1, and succeeded by a linear layer.

mnasNet [12] MNasNet uses training-aware neural architecture search (NAS), cnvolutions, depthwise separable convolutions, sequences of inverted residuals (pointwise, depthwise, linear pointwise), and finally a linear layer

EfficientNet [8] EfficientNet merges sequences of 'MBConv' blocks consisting of convolution and batchnorm layers with NAS, which is designed to select the best configuration. Ends with a linear layer We use scalings 1 and 7.

EfficientNet-V2 [9] These models were searched from the search space enriched with new ops such as Fused-MBConv. A combination of training-aware neural architecture search (NAS) and scaling was used to improve both training speed and parameter efficiency