# Dissecting Efficient Architectures for Wake-Word Detection

**Cody Berger** [*1] **Juncheng B Li** [*1] **Yiyuan Li** [2] **Aaron Berger** [3] **Dmitri Berger** [3] **Karthik Ganesan** [1]
**Emma Strubell** [1] **Florian Metze** [1]

## Abstract

Wake-word detection models on edge devices have stringent efficiency requirements (Sigtia et al., 2021). We observe that over-the-air test accuracy of models trained on parallel devices (GPU/TPU) usually degrades when deployed on edge devices using a CPU for over-the-air, real-time evaluation (Rybakov et al., 2020). Further, differing inference time when migrating between GPU and CPU varies across models. This drop is due to hardware latency and acoustic impulse response, while non-uniform growth of inference time results from models' varying exploitation of hardware acceleration. Although many neural architectures have been applied to wake-word detection tasks, such latency or accuracy drops have not been studied at granular, layer matrix multiplication levels. We compare five Convolutional Neural Network (CNN) architectures and one pure Transformer architecture, train them for wake-word detection on the Speech Commands dataset (Warden, 2018), and quantize two representative models. We seek to quantify their accuracy-efficiency tradeoffs to inform researchers and practitioners about the key components in models influencing this tradeoff.

## 1. Introduction

Many neural architectures have been successfully applied to wake-word detection (Rybakov et al., 2020; Zhang et al., 2017; Wu et al., 2018; Coucke et al., 2019; Berg et al., 2021), with existing works noting the tradeoff between accuracy, efficiency, and over-the-air robustness (Rybakov et al., 2020; Guo et al., 2018; Choi et al., 2019). While there is plenty of theory surrounding how architectures likely perform on sequential versus parallel devices, prior works have never analyzed how this bears out in practice. Further, many papers provide experimental performance findings only on either GPU or CPU (Rybakov et al., 2020). Yet we find in this paper that *efficiency and accuracy often do not linearly translate between GPU and CPU.* In lieu of experimental validation for layer and model performance on both GPU and CPU, researchers and developers risk their work being affected by incorrect assumptions about how performance transfers between devices. Prior works such as (Rybakov et al., 2020; Berg et al., 2021) point out the efficiency advantage of CNN-based architectures over other architectures such as Deep Neural Networks (DNNs), Recurrent Neural Networks (RNNs), or Transformers. But within the CNN family itself, differences in efficiency, accuracy, and over-the-air robustness are not clearly addressed. To shed light on this problem, we compare five different efficient CNN architectures and their performances on the wake-word task between GPU and CPU devices, as well as one Transformer architecture (ViT style, (Berg et al., 2021)) for completeness. We analyze how models' structural differences affect performance on GPU vs CPU without pretraining or fine-tuning. We apply static Post Training Quantization (PTQ) to two families of models, and interpret how those families' structural differences impact resulting speedup. With practical relevance in mind, we completed over-the-air accuracy and efficiency testing for all models to pinpoint where performance diverged across platforms. Lastly, we broke models down to identify specific factors which hinder model efficiency on CPU compared to GPU. [1]

## 2. Related Works

**Wake-Word Detection:** Rybakov et al. (2020) improved upon the baseline for wake-word detection set by Zhang et al. (2017), which empirically demonstrated the efficiency advantages of CNN-family models with depth-wise connection: DSCNN (MobileNet (Howard et al., 2017) equivalent). Concurrent works such as Wu et al. (2018); Coucke et al. (2019) explored other types of CNNs that involve dilation, and also acknowledged CNNs' inherent efficiency. Berg et al. (2021) reported that Transformers achieve 1 per-

[*]Equal contribution [1]Carnegie Mellon University, Pittsburgh, USA [2]University of North Carolina, Chapel Hill, USA [3]Independent Researcher. Correspondence to: Cody Berger, Juncheng B Li <codyberger@cmu.edu, junchenl@cs.cmu.edu>.

*Work presented at the ES-FoMo Workshop at ICML 2023*

[1]Our code and appendix (includes more details, results, and demos) can be found at: https://github.com/Wakeword2023/EsFoMo2023

cent better accuracy than DSCNN on the OnePlus 6 at the cost of a 4x increase in latency. Peter et al. (2022) used Quantization Aware Training (QAT), which requires retraining. Mittermaier et al. (2020) performed PTQ but used raw audio input, which is not in this work's scope.[2]

**Efficient Architectures:** Following MobileNet (Howard et al., 2017) this line of research saw huge developments benefiting from Neural Architecture Search (NAS), which injects efficiency metrics into the training objective. MobileNetV2 (Sandler et al., 2018), EfficientNet (Tan and Le, 2019) and EfficientNetV2 (Tan and Le, 2021) demonstrated an ever-improving efficiency-accuracy tradeoff. While existing wake-word literature has followed up on NAS search by layers (Peter et al., 2022; Mo et al., 2020), it has not yet followed up on the newer NAS search by blocks used by more recent architectures like EfficientNetV2.

## 3. Experiments

**Dataset and Setup:** We trained eight wake-word detection models using six different architectures on wake-word audio samples from the Google Speech Commands dataset (Warden, 2018) with PyTorch: VGG19_bn, DSCNN, EfficientNet_b1, EfficientNet_b7, EfficientNetV2_m, EfficientNetV2_xl, ResNet50, and Transformer (Berg et al., 2021). All architectures were non-streaming[3]. We also used PyTorch Eager Quantization (Contributors, 2022) to perform static PTQ on VGG19_bn and DSCNN. Here, we compared quantization in families of models - VGG based models, represented by VGG19_bn (VGG19_bn, ResNet), and MobileNet based models, represented by DSCNN (DSCNN, EfficientNetV1, EfficientNetV2). Digital testing analyzed models' performance on 807 audio files from the Speech Commands dataset, of which 187 contained the wake-word.

**Over-the-air Test:** We performed over-the-air testing to analyze overall model performance and layer-wise latencies. Three subjects conducted testing, two with lower voices, and one with a higher voice. All trials took place at the same location in the same room, ensuring a consistent room impulse response across trials. During over-the-air testing, subjects completed five trials for each model on a GPU platform and then a CPU platform. Table 1 details the results from analyzing digital versus over-the-air performance for all trained models on GPU and CPU devices. Digital Test F1 reflects the F1 score calculated by checking the correctness of models' predictions for whether or not an input contained the wake-word. Over-the-Air Test F1 reflects the F1 scores calculated from individual subjects' analysis of model prediction correctness for each trial. Table 3 (Appendix) breaks model leaf node layers into nine operational categories across the eight examined models: Conv, BN, ReLu, MaxPool, Linear, Dropout, AvgPool, LayerNorm, and GELU. We used PyTorch hooks to calculate the average latency for each layer category, and the THOP library (Zhu, 2022) to calculate the average FLOPS for each layer category.

*Table 1.* Model performance on Speech Commands Dataset. F1 scaled to 100 and latencies in *ms*. Top 3 results are highlighted.

| GPU (Mac-M1) | Digital F1 | Over-the-Air F1 | Latency |
|---|---|---|---|
| VGG19_bn (Rybakov et al., 2020) | **93.37** | **96.69** ± 2.99 | **17.3** |
| DSCNN (Rybakov et al., 2020) | 92.15 | 93.78 ± 5.61 | **37.6** |
| EfficientNet_b1 (Tan and Le, 2019) | 92.75 | 93.16 ± 4.90 | 85.5 |
| EfficientNet_b7 (Tan and Le, 2019) | **93.87** | **96.55** ± 2.59 | 146.0 |
| EfficientNetV2_m (Tan and Le, 2021) | 92.88 | 92.14 ± 5.60 | 94.0 |
| EfficientNetV2_xl (Tan and Le, 2021) | 92.64 | **94.06** ± 2.08 | 146.3 |
| ResNet50 (He et al., 2016) | 91.86 | 90.52 ± 7.48 | **56.5** |
| Transformer (Berg et al., 2021) | **93.87** | 91.47 ± 4.56 | 65.8 |
| **CPU (Raspberry Pi 4B)** | **Digital F1** | **Over-Air F1** | **Latency** |
| VGG19_bn (Rybakov et al., 2020) | **93.37** | **90.80** ± 6.54 | 372.0 |
| DSCNN (Rybakov et al., 2020) | 92.15 | **90.28** ± 4.02 | **151.0** |
| EfficientNet_b1 (Tan and Le, 2019) | 92.75 | 89.83 ± 4.31 | **283.0** |
| EfficientNet_b7 (Tan and Le, 2019) | **93.87** | 89.20 ± 5.20 | 1044.0 |
| EfficientNetV2_m (Tan and Le, 2021) | 92.88 | **93.45** ± 6.61 | 808.0 |
| EfficientNetV2_xl (Tan and Le, 2021) | 92.64 | 71.85 ± 5.54 | 1673.0 |
| ResNet50 (He et al., 2016) | 91.86 | 90.18 ± 10.37 | 324.1 |
| Transformer (Berg et al., 2021) | **93.87** | 88.96 ± 11.30 | **182.6** |

## 4. Preliminary Observations

VGG19_bn and EfficientNet_b7 achieved the highest F1 scores during over-the-air GPU testing, surpassing the larger EfficientNetV2_xl model by about two points (Table 1). VGG19_bn also achieved the lowest latency. *VGG19_bn's success is likely linked to the efficiency of its convolution method on GPU.* Moving from GPU to CPU, over-the-air performance dropped for many architectures. All models except for EfficientNetV2_m saw their F1 scores slip below the digital baseline, consistent with previous research (Rybakov et al., 2020). The EfficientNetV2_xl model in particular showed a steep decline in performance, accompanied by a large increase in latency (Table 1). *EfficientNetV2_xl's drop in performance is likely a result of its increased latency, in turn caused by its convolution method's inefficiency on CPU.* DSCNN achieved the lowest latency, enduring only a small performance drop. The Transformer maintained a relatively low latency across both GPU and CPU, though its accuracy dropped moving to CPU. In the discussion below we drill into the fine-grained components of each architecture to elaborate on these italicized sections.

## 5. Further Analysis and Discussion

Table 3 breaks down individual network components, analyzing their Percentages of Aggregate Runtime (PAR), the relative time models spent computing each type of layer.

[2]We use logMel features, Mittermaier et al. (2020) also acknowledged systems using logMel features still prevail performance-wise.

[3]We used the same definition of streaming and non-streaming in (Rybakov et al., 2020).

We recommend readers juxtapose Table 3 in the appendix with the subsequent paragraph for better reading.

**Matrix Multiplication in Convolution:** 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 strongly affects the efficiency of convolution layers because most of convolution's computation comes from 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, resulting in a total cost of $O(h \cdot w \cdot c_{in} \cdot c_{out} \cdot k^2)$, where $r$ is a specific input channel and $s$ is a specific output channel:
$z[:,:,s] = \sum_{r=1}^{c_{in}} x[:,:,r] \times W[r,s,:,:]$
Note that only the forward pass is involved in wake-word detection; there is no back-propagation.

**Matrix Multiplication Optimization:** The parallelization abilities of CPUs are limited by their low numbers of cores and caches. GPUs, however, are equipped with larger array of cores and caches to support acceleration of parallellizable computations, including matrix multiplication. Matrix multiplication can be broken down via tiling (Nvidia, 2022), enabling one large matrix multiplication to be calculated as the multiplication of a series of much smaller, parallelizable chunks. This strategy helps avoid costly memory accesses. GPUs can take advantage of this parallel character to significantly speed up matrix multiplication. For most of the models we tested, convolution took a larger share of time to execute on CPU versus on GPU (See Table 3 Conv rows and PAR columns). Parallelization and tiling offer a good explanation for this pattern.

**MBConv Blocks (Sandler et al., 2018) versus Vanilla CNNs:** The cost of convolution on GPU versus CPU can also be affected by model architecture. Models with MBConv blocks (like DSCNN, EfficientNet, and EfficientNetV2) use pointwise convolutions, forcing depthwise convolutions to be done in sequential order. This method reduces the cost of convolution to $O(h \cdot w \cdot c_{in} \cdot (c_{out} + k^2))$ (Sandler et al., 2018). Notably, the sequential computation design of MBConv blocks reduces their parallelizability, and consequently MBConvs can only receive a minor latency boost on GPU. Yet moving to CPU, they show less slowdown and reap the benefits from requiring fewer operations (Table 3 shows about 10x slow down GPU versus CPU in Conv latency). Meanwhile, models using vanilla CNN blocks (like VGG19_bn) enable greater parallelization and therefore high GPU performance, but suffer a more severe slowdown transitioning to CPU. Table 3's latency columns show that the VGG19_bn model's vanilla convolution layers saw an exponentially larger latency increase (100x GPU versus CPU) compared to models using MBConv blocks (DSCNN, EfficientNet, EfficientNetV2). This difference clarifies that theoretical algorithm complexity alone is not sufficient to determine an algorithm's practical efficiency on GPU; parallel algorithm complexity must also be accounted for.

**Accelerating Convolution:** Convolution receives significant acceleration on GPU devices. As CPUs cannot take advantage of parallelism to accelerate matrix multiplication, convolution is often much slower on CPU devices, even with architecture optimization like MBConv blocks. This is corroborated by our findings in Table 3, which shows a greater latency increase moving from GPU to CPU in models which were more reliant on convolution blocks, like EfficientNet_b1 and EfficientNet_b7. Yet the value of architecture optimization cannot be ignored. Without a deeper understanding of how GPUs accelerate convolution and how different models optimize for GPU or CPU, it is hard to fully comprehend the performance tradeoff between parallel and sequential devices.

**Fused-MBConv Blocks in EfficientNetV2:** EfficientNetV2 was NAS trained on GPU to find the optimal combination of MBConv and Fused-MBConv blocks. In order to take better advantage of GPU acceleration, Fused-MBConv blocks remove the pointwise convolution, forcing depthwise convolutions to be sequentially applied for each individual channel (Tan and Le, 2021). This change echoes the structure of vanilla convolution blocks. In Table 3, EfficientNetV2 shows similar patterns to VGG19_bn, which used vanilla convolution. Specifically, convolution has a much lower PAR on GPU than on CPU for EfficientNetV2 (Table 3), a marked difference from EfficientNet where PAR for convolution was similar across CPU and GPU. This is consistent with its use of Fused-MBConv blocks. Also, EfficientNetV2 was designed using NAS to find the optimal block combination on GPU, without guaranteeing that the combination is optimal on CPU.

**Batch Normalization (BN) VS Layer Normalization:** Data from Table 3's latency columns shows that for most models, batch normalization slows down from GPU to CPU, but less than convolution. BN layers normalize over mini-batches of an activation matrix and require cell-by-cell operations, so the work of normalizing one column is $O(m)$ where $m$ is batch size. Normalization occurs for all $d$ dimensions in the activation matrix, giving batch normalization work of $O(md)$. Batch normalization can be done in parallel on GPU, but we need the entire batch loaded for these calculations, and when $m$ is small, there would be not much parallelism. In contrast, in Layer Normalization, despite having to do the same total work $O(md)$ theoretically, the mean and variance are computed over the $d$ features of each individual instance (row) without batch dependency, so the computations for different instances are independent. This makes it easier to split the computations across GPU processing units, and would enjoy more speedup when inputs are contiguous. Both BN and LayerNorm would lose parallelization moving to CPU causing a large increase in latency as observed in Table 3.

**Multi-Head Attention Parallelizability:** The Multi-Head Attention layer computes the self-attention equation $\mathrm{Softmax}(\frac{KQ^T}{\sqrt{d}}) \cdot V$ where $K, Q, V \in \mathbf{R}^{n \times d}$. Ultimately, Multi-Head Attention takes $O(n^2 d)$ work, all resulting from matrix multiplication. While matrix multiplication receives acceleration boosts both from PyTorch and GPU, Multi-Head Attention takes more work than convolution by a factor of $n$, limiting the speedup potential. Though the latency of Multi-Head Attention decreases moving from CPU to GPU, it does not show a major decrease akin to convolution (Table 4). This is likely attributable in part to the limited optimization potential for attention layers.

## 6. Quantization

We investigated the effectiveness of quantization for the VGG and MobileNet families by quantizing VGG19_bn and DSCNN respectively. There are two main strategies for quantizing neural architectures: Quantization Aware Training (QAT) and Post Training Quantization (PTQ) (Krishnamoorthi et al., 2020). QAT requires retraining, which is sometimes impractical without the model's original training data and can be computationally expensive. Since PTQ is post-hoc quantization, it would be the most accessible option for developers lacking resources for NAS or other efficiency improvements. For these reasons, we chose to study PTQ as a complement to the previous QAT study in wake-word detection (Peter et al., 2022). We implemented quantization using PyTorch's open-source quantization library. However, it only has full support for CPU (Contributors, 2022). The results of using PyTorch Eager Quantization for static PTQ to quantize VGG19_bn and DSCNN from float32 to int8 are show in Table 2.

For both models, we found that quantization improved latency without harming F1 scores relative to unquantized models (Table 2). This suggests that PTQ is indeed a valuable strategy for developers seeking to increase model efficiency or to deploy architectures on edge devices.
However, we also noticed that the latency speedup for both models (2.3 times and 1.31 times, from Table 2) was lower than the maximum speedup that could be achieved theoretically from float32 to int8 (4 times). Convolution's interaction with caches provides one possible explanation. As shown in Section 5, convolution is a component with heavy computation and may lead to frequent reloading of model weights into caches. After quantization, model weights would not need to be loaded into the cache as frequently since some weights which were different with 32-bit precision would be the same when quantized to 8-bit precision. In contrast, components with lighter computation complexity like DSCNN and MBConv blocks sequentially apply convolution one channel at a time, which decreases how often data would need to be reloaded into the cache. Since there is less reloading to begin with, they would benefit less from quantization than VGG19_bn. However, DSCNN is better for the overall latency despite receiving less speedup from quantization when compared to VGG19_bn.

*Table 2.* Quantized Model Performance, VGG19_bn, DSCNN are identical to Rybakov et al. (2020)

| Quantized int8 models on CPU (Raspberry Pi 4B) | | | |
|---|---|---|---|
| CPU | Digital F1 (%) | Over-Air F1 (%) | Latency (ms) ($\times$ Speedup) |
| VGG19_bn | 92.63 | $97.57 \pm 4.07$ | 162.0 ($\times$ 2.30) |
| DSCNN | 91.86 | $95.13 \pm 4.28$ | 115.0 ($\times$ 1.31) |
| **Unquantized float32 models on CPU (Raspberry Pi 4B) from Table 1** | | | |
| CPU | Digital F1 (%) | Over-Air F1 (%) | Latency (ms) |
| VGG19_bn | 93.37 | $90.80 \pm 10.19$ | 372.0 |
| DSCNN | 92.15 | $90.28 \pm 7.25$ | 151.0 |

## 7. Conclusion

In this work, we compared the performance of six efficient neural architectures for the wake-word detection task. We observed that the best-performing architectures on GPU and CPU did not perfectly align: VGG19_bn was the highest-performing model on GPU and DSCNN the highest on CPU. We found that this corresponded to the categories of convolution layers they used. Our observations showed that MobileNet-related architectures were more efficient architecture choices for CPU deployment, as their use of MBConv blocks reduced the computational cost on CPU. However, the more parallelizable character of VGG19_bn allows it to outperform MobileNet-like architectures on GPU. Descendants of DSCNN which were optimized for ImageNet using NAS (EfficientNet, EfficientNetV2) tended to demonstrate worse efficiency on both CPU and GPU than other models examined, despite being more recent. This suggests that NAS optimization for image data does not directly benefit optimization for audio, and could make such architectures sub-optimal for wake-word detection on edge devices. We expect our results to benefit practitioners trying to choose the "right" efficient architecture for a platform. Our analysis of over-the-air performance and post-training quantization provides insight into the balance between accuracy and efficiency across various architectures.

## References

Axel Berg, Mark O'Connor, and Miguel Tairum Cruz. Keyword transformer: A self-attention model for keyword spotting. In *Interspeech 2021*, pages 4249–4253. ISCA, 2021.

Seungwoo Choi, Seokjun Seo, Beomjun Shin, Hyeongmin Byun, Martin Kersner, Beomsu Kim, Dongyoung Kim, and Sungjoo Ha. Temporal convolution for real-time keyword spotting on mobile devices. *Proc. Interspeech 2019*, pages 3372–3376, 2019.

PyTorch Contributors. Quantization. `https://pytorch.org/docs/stable/quantization.html`, 2022.

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)*, pages 6351–6355. IEEE, 2019.

CPU-monkey. Apple m1 vs raspberry pi 4 b (broadcom bcm2711). `https://www.cpu-monkey.com/en/compare_cpu-apple_m1-vs-raspberry_pi_4_b_broadcom_bcm2711`, 2023.

Jinxi Guo, Kenichi Kumatani, Ming Sun, Minhua Wu, Anirudh Raju, Nikko Ström, and Arindam Mandal. Time-delayed bottleneck highway networks using a dft feature for keyword spotting. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5489–5493. IEEE, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

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.

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*, pages 4700–4708, 2017.

Aleksandar Kostovic. Apple m1's 'small' icestorm cores benchmarked against 'big' firestorm cores. 2021.

Raghuraman Krishnamoorthi, James Reed, Min Ni, Chris Gottbrath, and Seth Weidman. Introduction to quantization on pytorch. `https://pytorch.org/blog/introduction-to-quantization-on-pytorch/`, 2020.

Simon Mittermaier, Ludwig Kürzinger, Bernd Waschneck, and Gerhard Rigoll. Small-footprint keyword spotting on raw audio data with sinc-convolutions. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7454–7458. IEEE, 2020.

Tong Mo, Yakun Yu, Mohammad Salameh, Di Niu, and Shangling Jui. Neural architecture search for keyword spotting. *arXiv preprint arXiv:2009.00165*, 2020.

Nvidia. Nvida cuda programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2022.

David Peter, Wolfgang Roth, and Franz Pernkopf. End-to-end keyword spotting using neural architecture search and quantization. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3423–3427. IEEE, 2022.

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

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*, pages 4510–4520, 2018.

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)*, pages 6843–6847, 2021. doi: 10.1109/ICASSP39728.2021.9414218.

Shyam A Tailor, Milad Alizadeh, and Nicholas D Lane. Torchquant: A hackable quantization library for researchers, by reseachers, 2021.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *International Conference on Machine Learning*, pages 10096–10106. PMLR, 2021.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile, 2019.

Luyu Wang, Pauline Luc, Yan Wu, Adria Recasens, Lucas Smaira, Andrew Brock, Andrew Jaegle, Jean-Baptiste Alayrac, Sander Dieleman, Joao Carreira, and Aaron van den Oord. Towards learning universal audio representations, 2021. URL https://arxiv.org/abs/2111.12124.

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

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)*, pages 5494–5498. IEEE, 2018.

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

Ligeng Zhu. Thop. https://github.com/Lyken17/pytorch-OpCounter, 2022.

## A. Appendix

*Table 3.* Analysis of Specific Layer Classes

**Average Latency**: Latencies of individual layers were averaged within their layer category for each trial
**PAR** (Percentage of Aggregate Runtime): Measures percentage of a model's total runtime spent computing layers in a specific category across trials. Percentages for categories taking up the most relative computation time are bolded
**Average FLOPS**: Measures the average floating point operations per second across all trials for a layer class
**Conv**: Convolution layers; **BN**: Batch normalization layers
**Note**: If a model does not have the layer in a row, it is filled with "-". 0 FLOPS indicates an existing layer has no FLOPS.

| Model (#Param) | **VGG19_bn** (38.9M) (Rybakov et al., 2020) | | | | | **DSCNN** (2.23M) (Rybakov et al., 2020) | | | | | **EfficientNet_b1** (0.06M) (Tan and Le, 2019) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Avg Latency (ms) | | Avg FLOPS | PAR (%) | | Avg Latency (ms) | | Avg FLOPS | PAR (%) | | Avg Latency (ms) | | Avg FLOPS | PAR (%) | |
| | GPU | CPU | | GPU | CPU | GPU | CPU | | GPU | CPU | GPU | CPU | | GPU | CPU |
| Conv | 0.369 | 12.938 | 3.90E+07 | **37.56** | **56.57** | 0.235 | 1.411 | 2.40E+05 | 39.51 | **53.09** | 0.340 | 1.348 | 5.63E+08 | **67.38** | **60.09** |
| BN | 0.334 | 3.656 | 1.39E+05 | 34.07 | 15.94 | 0.252 | 1.149 | 1.19E+04 | **42.25** | 43.22 | 0.220 | 1.438 | 1.15E+04 | 26.25 | 38.49 |
| ReLU | 0.136 | 0.542 | 0 | 15.53 | 2.67 | 0.151 | 0.134 | 0 | 17.04 | 3.41 | - | - | - | - | - |
| SiLU | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| MaxPool | 0.235 | 3.346 | 0 | 7.49 | 4.55 | - | - | - | - | - | - | - | - | - | - |
| Linear | 0.274 | 24.691 | 1.26E+07 | 5.14 | 20.22 | 0.347 | 0.249 | 5.12E+03 | 1.12 | 0.18 | 0.300 | 0.139 | 5.12E+03 | 0.52 | 0.05 |
| Dropout | 0.017 | 0.084 | 0 | 0.22 | 0.05 | 0.029 | 0.137 | 0 | 0.10 | 0.10 | 0.020 | 0.072 | 0 | 0.03 | 0.03 |
| AvgPool | - | - | - | - | - | - | - | - | - | - | 0.141 | 0.155 | 2.13E+02 | 5.82 | 1.44 |

| Model (#Param) | **EfficientNet_b7** (0.32M) (Tan and Le, 2019) | | | | | **EfficientNetV2_m** (53.5M) (Tan and Le, 2021) | | | | | **EfficientNetV2_xl** (207M) (Tan and Le, 2021) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Avg Latency (ms) | | Avg FLOPS | PAR (%) | | Avg Latency (ms) | | Avg FLOPS | PAR (%) | | Avg Latency (ms) | | Avg FLOPS | PAR (%) | |
| | GPU | CPU | | GPU | CPU | GPU | CPU | | GPU | CPU | GPU | CPU | | GPU | CPU |
| Conv | 0.240 | 2.501 | 7.13E+10 | **65.21** | **69.84** | 0.189 | 2.865 | 1.37E+06 | 32.91 | **59.10** | 0.149 | 3.921 | 2.58E+06 | 31.02 | **68.11** |
| BN | 0.177 | 1.748 | 4.26E+03 | 28.69 | 29.14 | 0.201 | 1.570 | 1.38E+04 | **35.09** | 32.52 | 0.175 | 1.277 | 1.79E+04 | **36.26** | 21.18 |
| ReLU | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| SiLU | - | - | - | - | - | 0.090 | 0.123 | 0 | 14.45 | 2.34 | 0.075 | 0.146 | 0 | 14.52 | 2.46 |
| MaxPool | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| Linear | 0.347 | 0.147 | 4.94E+05 | 0.345 | 0.02 | 0.121 | 0.434 | 8.76E+03 | 11.77 | 4.88 | 0.104 | 0.687 | 4.93E+05 | 12.37 | 7.14 |
| Dropout | 0.022 | 0.073 | 0 | 0.02 | 0.01 | - | - | - | - | - | - | - | - | - | - |
| AvgPool | 0.103 | 0.174 | 9.14E+01 | 5.73 | 1.00 | 0.118 | 0.201 | 1.22E+04 | 5.79 | 1.15 | 0.098 | 0.216 | 1.53E+04 | 5.84 | 1.12 |

| Model (#Param) | **ResNet** (23.5M) (He et al., 2016) | | | | |
|---|---|---|---|---|---|
| | Avg Latency (ms) | | Avg FLOPS | PAR (%) | |
| | GPU | CPU | | GPU | CPU |
| Conv | 0.383 | 3.286 | 3.09E+6 | **44.79** | **57.08** |
| BN | 0.299 | 2.248 | 1.57E+04 | 34.98 | 39.03 |
| ReLU | 0.156 | 0.087 | 0 | 16.90 | 1.40 |
| SiLU | - | - | - | - | - |
| MaxPool | 0.476 | 5.350 | 0 | 1.05 | 1.75 |
| Linear | 0.684 | 0.153 | 8.19E+03 | 1.51 | 0.05 |
| Dropout | - | - | - | - | - |
| AvgPool | 0.344 | 2.125 | 4.10E+03 | 0.76 | 0.70 |
| LayerNorm | - | - | - | - | - |
| GELU | - | - | - | - | - |

*Table 4.* Transformer Layers and Operations

| Model (#Param) | **Transformer** (0.60M) (Berg et al., 2021) | | | | |
|---|---|---|---|---|---|
| | Avg Latency (ms) | | Avg FLOPS | PAR (%) | |
| | GPU | CPU | | GPU | CPU |
| Multi-Head Attention | 2.631 | 3.082 | * | 54.19 | 22.71 |
| Linear | 0.432 | 1.096 | 4.89E+2 | 19.26 | 17.43 |
| LayerNorm | 0.566 | 2.570 | 1.69E+4 | 23.29 | 37.96 |
| GELU | 0.134 | 2.804 | 0 | 2.75 | 20.70 |
| Dropout | 0.012 | 0.081 | 0 | 0.51 | 1.20 |

*The THOP library could not measure FLOPS for the Multi-Head Attention layer since it is implemented using the einops library. We are current finding another way to measure FLOPS for this layer.

## B. Training Details

All models were trained using our open-source code, which can be found at https://github.com/Wakeword2023/EsFoMo2023. The models were trained on the Google Speech Commands train dataset (Warden, 2018) for the keyword 'Sheila.' The Speech Commands dataset contains 1372 utterances of the keyword. Before training, we augmented the dataset by applying transformations to the keyword utterances. Five copies of each transformed keyword utterance were put into the training dataset. In total, this gave us 6,860 keyword utterances. We added 16,558 non-keyword audio files to the training dataset, 712 of which were silence and 15,846 of which contained randomly chosen non-keyword utterances from the Speech Commands dataset, which underwent the same transformations as the keyword utterances. The valid dataset was computed by applying transformations to audio files in the Speech Commands valid dataset for the keyword 'Sheila'. Only one copy of each transformed keyword utterance was added to the valid dataset, for a total of 188 valid keyword utterances. We added 620 non-keyword audio files to the valid dataset, 18 of which were silence and 602 of which contained randomly chosen non-keyword utterances from the Speech Commands dataset, which underwent the same transformations as the keyword utterances. We used the same training hyperparameters for all models except the Transformer, for which we referenced the training parameters described in Berg et al. (2021). Table 5 provides a breakdown of the training hyperparameters.

*Table 5.* Training Hyperparameters

| | Non-Transformer | Transformer (Berg et al., 2021) |
|---|---|---|
| Batch Size | 64 | 512 |
| Epochs | 75 | 75 |
| Warmup Epochs | 0 | 10 |
| Weight Decay | 0.01 | 0.1 |
| Optimization | SGD | AdamW |
| Learning Rate | 0.0001 | 0.001 |
| Learning Rate Scheduler | Plateau | Cosine Annealing |

### B.1. Digital Test

Digital testing was conducted by inputting audio samples into the model, and comparing the model's predictions with samples' labels. The test dataset was computed by applying transformations to audio files in the Speech Commands test dataset for the keyword 'Sheila'. Only one copy of each transformed keyword utterance was added to the test dataset, for a total of 188 test keyword utterances. We added 465 non-keyword audio files to the test dataset, 18 of which were silence and 447 of which contained randomly chosen non-keyword utterances from the Speech Commands dataset, which underwent the same transformations as the keyword utterances. The F1 score resulting from digital testing on the test dataset is the 'Digital F1' score that appears in Table 3.

### B.2. Over-the-air Test

As described in Table 3, for each model five over-the-air trials were conducted both on GPU and on CPU. Audio was recorded and processed in 1-second chunks during each over-the-air trial. Each trial lasted 20 seconds, yielding 20 audio chunks. Immediately after being recorded, the 1 second chunk was downsampled to 16kHz and converted into a logMel spectrogram before being sent as input to the classifier. While there are several available approaches to wake-word detection (Mittermaier et al., 2020; Wang et al., 2021; Rybakov et al., 2020; Coucke et al., 2019), it has been shown that audio preprocessing using logMel spectrograms yields the best performance (Peter et al., 2022), and so this is the approach that we followed. After classification, the model's prediction for whether the wake-word was present in that chunk printed to the console. Subjects uttered a phrase including the wake-word ten times per trial, and manually determined each model's prediction accuracy. This data was used to compute an average over-the-air F1 score for each model by $\text{Average} = \frac{1}{|S|}\sum_s \frac{1}{|T|}\sum_t \frac{\text{TP}_t}{\text{TP}_t + \frac{1}{2}(\text{FP}_t + \text{FN}_t)}$ where $s \in S$ is a unique subject and $t \in T$ is a unique trial. For each trial, latency was calculated as the time between a model receiving input and returning its corresponding prediction.

## C. FLOPS Count & Hooks:

We measured the total floating point operations per second (FLOPS) and parameter count for each model using the THOP library (Zhu, 2022)'s profiler. Since THOP measures MACS, we used the widely accepted estimation $\text{FLOPS} = \text{MACS} \times 2$. In addition, timing functionality was added to each model in order to find the runtime for individual layers. PyTorch forward hooks collected and logged the timing results for all layers. We timed the following layers: Conv2d, Batchnorm2d, ReLU, Maxpool2d, Avgpool2d/AdaptiveAvgpool2d, Dropout, Linear, LayerNorm, and GELU. Avgpool2d and AdaptiveAvgpool2d were treated as the same layer for conciseness of presentation in Table 3. We did not time Sigmoid, Softmax, or Swish/MemoryEfficientSwish layers. Finally, we modified the THOP library's profiler to log MACS for individual layers by having its dfs_count function keep track of MACS for individual leaf-node layers in the model graph. THOP automatically zeroes the MACS for Dropout, ReLU, GELU, LayerNorm, Sigmoid, Softmax, and Swish/MemoryEfficientSwish layers.

### C.1. Batch Normalization Additional Analysis:

The effects of Batch Normalization's limited parallelizability are visible in Table 3, where BN layers accounted for the most computation on GPU but not on CPU for three out of the six models. Since BN layers receive less relative GPU acceleration than convolution, it follows that its GPU and CPU latencies will be less disparate. The only model to see PAR for convolution layers decrease moving from GPU to CPU was EfficientNet_b1, the smallest model among those tested (Table 3). This occurred in conjunction with an increase of time spent in BN layers, suggesting that EfficientNet_b1 has few enough parameters for convolution to be relatively low cost. This indicates that while most of the computation cost in these models comes from convolution, the cost of batch normalization is non-negligible and should be considered in model selection.

### C.2. Images versus Audio Non-Transferability:

The results of Table 1 show that EfficientNet_b1 (Tan and Le, 2019) and EfficientNet_b7 (Tan and Le, 2019), optimized on both accuracy and FLOPS for computer vision tasks, do not transfer the same accuracy versus latency tradeoff to the audio wake-word task. Compared to DSCNN (MobileNet based architecture) (Rybakov et al., 2020), EfficientNetV1 architectures have better accuracy but inferior latency. DSCNN precedes EfficientNetV1 and EfficientNetV2, which were trained using NAS (Neural Architecture Search) on CIFAR or other vision datasets. That DSCNN outperforms its successors in the wake-word detection task should motivate future works to train CNN-based audio models jointly optimized for FLOPS and accuracy.

### C.3. Calculations:

For all models except Transformer, we broke model layers into seven categories: Conv2d, BatchNorm2d, ReLU, MaxPool2d, Linear, Dropout, and AveragePool2d. The AveragePool2d category represented both AveragePool2d and AdaptiveAveragePool2d layers. For Transformer, we broke model layers into x categories: Multi-Head Attention, Linear, LayerNorm, and GELU. For each trial, a model's 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 layers in the same category, and by counting every time a 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 layer for 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 $n$th (and last) 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 $c$:

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

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

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

## D. Quantization Details:

We quantized both VGG19_bn and DSCNN using PyTorch Eager Mode quantization (Contributors, 2022). For DSCNN, we adapted our existing code for quantization following the 'Model Architecture' and 'Post-training Static Quantization' steps in the available tutorial at `https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html`. For VGG19_bn, we used similar code following the 'Post-training Static Quantization' step in this tutorial. Both models were quantized from float32 to int8. Our quantization implementation is available in our repository. Both of these models were neither difficult nor time-consuming to quantize with the publicly available PyTorch tutorial, and the effects of quantization on latency (Table 2) were readily apparent during testing.

**The Post-Training Quantization Landscape:** After quantizing VGG19_bn and DSCNN as representatives of the CNN family (VGG19_bn, ResNet50) and the DSCNN family (DSCNN, EfficientNet, EfficientNetV2), we sought to quantize the other models we examined in order to provide more detailed, granular analyses of quantization for these models. However, it soon became apparent that this is not as simple using the widely available tools for PTQ.

**Pytorch Eager Mode Quantization (Contributors, 2022)**: Pytorch Eager Mode quantization is still in beta. We successfully used it to quantize VGG19_bn and DSCNN, following PyTorch tutorials. It is not always very simple to use: bugs can be particularly confusing, and documentation is somewhat sparse. Nonetheless, it was far and away the best PTQ library that we experimented with, and it is the library that we plan on using in the future as we work on quantizing the rest of the models examined in this paper.

**PyTorch FX Graph Mode Quantization (Contributors, 2022)**: PyTorch introduced this mode of quantization, which is currently a prototype, as a simpler and more automatic alternative to PyTorch Eager Mode quantization. We found that it was more difficult to use than PyTorch Eager Mode quantization for VGG19_bn and DSCNN, both of which were already symbolically traceable. We could not get either of them to work within a reasonable amount of time following the official PyTorch FX quantization tutorials. Likewise, some of our other models like Transformer rely heavily on libraries and operations which PyTorch FX Graph Mode cannot quantize. To quantize our Transformer implementation using PyTorch FX Graph Mode, we would have to rewrite it entirely, which goes against the goal of the PyTorch FX library.

**TorchQuant Library and the PTQ landscape:** We also investigated third-party quantization libraries that advertised PTQ capabilities. In particular, we focused on the TorchQuant library (Tailor et al., 2021) which is geared towards quantization researchers, and has both QAT and PTQ modes. It also has fused implementations of EfficientNet, MobileNet, and ResNet designed to work with its own model of quantization. Without too much trouble, we were able to successfully integrate our own model implementations and input with the TorchQuant fused models and quantization. However we discovered that TorchQuant cannot actually do PTQ since it is not integrated with the PyTorch native ATen library. Given the instruction to quantize weights to eight bits, TorchQuant's quantization takes the float32 weights and zeroes out all information beyond eight bits. However, the weights remain in float32 format. This kind of 'simulated quantization' works well for QAT, and allows researchers to investigate how quantization impacts model accuracy without worrying about how it interacts with the PyTorch backend. But the goal of PTQ is to actually quantize the weights sent to the device to a smaller data representation in order to study the efficiency as well as the accuracy of quantized models, and the TorchQuant library cannot achieve this.

**Takeaways:** Because PTQ requires interaction with lower-level PyTorch, it is extremely challenging to do PTQ using PyTorch without using PyTorch's own quantization library. For this reason, current open-source options for PTQ are very limited, which creates challenges for developers. Part of the reason we chose to investigate PTQ rather than QAT was because we percieved it to be more practical: both because it doesn't require the computing power and technical background necessary to train a model as QAT does, and because it has a direct impact on a model's efficiency in addition to its theoretical accuracy. For researchers and developers looking to run neural networks on edge, PTQ could be a way to improve efficiency of 'off-the-shelf' pre-trained models. Yet as we discovered, PTQ is more difficult that QAT, and although PyTorch's quantization libraries are not fully developed, there are few alternatives. The open-source tools available for implementing PTQ do not live up to its potential.

**Future Work:** We are still working on quantization for the other models we examined. In particular, there are several promising open-source fused transformer implementations which are designed to work with PyTorch eager quantization. We will update our repository as we quantize more models.

## E. Hardware and Software:

In all of our experiments, our GPU was a 2020 13-inch MacBook Pro with an Apple M1 chip running macOS Ventura 13.1. We chose to use an M1 Mac since it has support through Pytorch MPS, and because it is a good example of a common, fairly portable GPU. Our CPU in all experiments was a Raspberry Pi 4B running 64-bit Raspberry Pi OS with Desktop, Debian version 11 (bullseye). We chose Raspberry Pi 4B as our CPU in this work because it is a common, open-source, portable CPU, and it is cheaper than many other CPU examples and thus more financially feasible for researchers and developers. The Raspberry Pi does not come with a built-in microphone, so we equipped it with the Seeed ReSpeaker 4-Mic Array for Raspberry Pi for all of our experiments. While the PyTorch device can be set to CPU on M1 Mac, we chose not to use it because it is not a good representative for a typical CPU. A 13-inch M1 MacBook Pro has eight cores and eight threads (CPU-monkey, 2023), some of which are optimized for performance and others which are optimized for efficiency (Kostovic, 2021). This means that the M1 Mac CPU has greater potential for parallelization and can sometimes behave more like a GPU, as shown in Table 6. We ran tests to obtain over-the-air latencies for all of our models, finding that parallelizable models (CNN family, Transformer) had relatively low latencies but models optimized for highly sequential devices (DSCNN family) had poorer latencies.

Based on our understanding of M1 architecture, the Mac M1 CPU does not seem like a good representation of a portable, edge CPU. Thus, we chose not to use it in our experiments.

**Audio Acquisition Time:** For both GPU and CPU, audio

*Table 6.* Model Latency on Speech Commands Dataset

| CPU (Mac-M1) | Latency |
|---|---|
| VGG19_bn (Rybakov et al., 2020) | 69.5 |
| DSCNN (Rybakov et al., 2020) | 350.6 |
| EfficientNet_b1 (Tan and Le, 2019) | 566.1 |
| EfficientNet_b7 (Tan and Le, 2019) | 2562.6 |
| EfficientNetV2_m (Tan and Le, 2021) | 1711.3 |
| EfficientNetV2_xl (Tan and Le, 2021) | 4401.1 |
| ResNet50 (He et al., 2016) | 54.5 |
| Transformer (Berg et al., 2021) | 23.8 |

acquisition time is minimal. We measured that it was 1.5ms on M1 Mac and 1.0ms on Raspberry Pi. We calculated audio acquisition time as the difference between the time of an utterance of audio and the time when it is received by the device memory. This was done by simultaneously starting a stopwatch and a recording, observing the time of an utterance on the stopwatch, and then comparing this time to the time when the utterance was recorded.

## F. Other Models

We also trained and tested MnasNet and DenseNet models. Since DenseNet is not an efficient architecture and MnasNet was optimized for object detection, they are not the focus of our main investigation, but we include their experimental results here.

*Table 7.* Additional Model performance on Speech Commands Dataset. F1 scores are scaled to 100 and latencies are in *ms*.

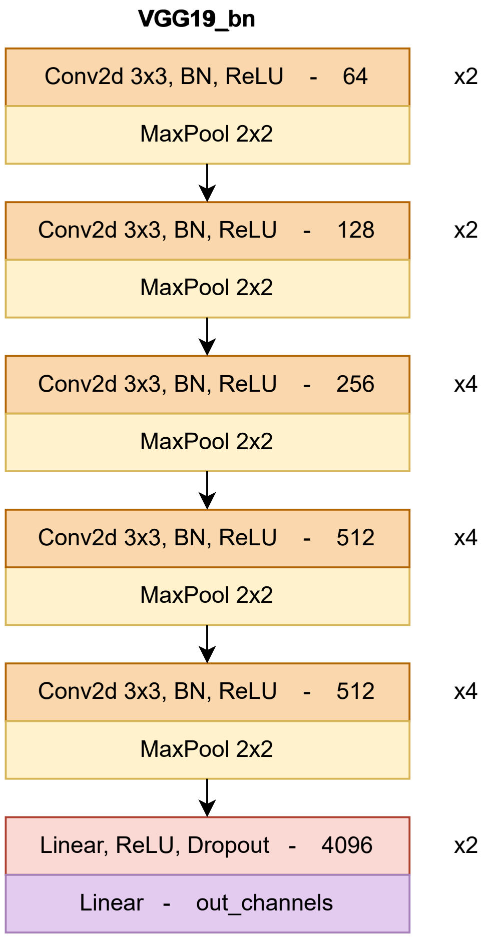
| GPU (Mac-M1) | Digital F1 | Over-the-Air F1 | Latency |
|---|---|---|---|
| MnasNet (Tan et al., 2019) | 91.58 | 82.59 ± 12.54 | 35.3 |
| DenseNet (Huang et al., 2017) | 93.37 | 94.85 ± 4.49 | 344.2 |
| **CPU (Raspberry Pi 4B)** | **Digital F1** | **Over-Air F1** | **Latency** |
| MnasNet (Tan et al., 2019) | 91.58 | 82.37 ± 14.16 | 131.1 |
| DenseNet (Huang et al., 2017) | 93.37 | 42.22 ± 6.81 | 3674.0 |

*Table 8.* Other Models: Analysis of Specific Layer Classes

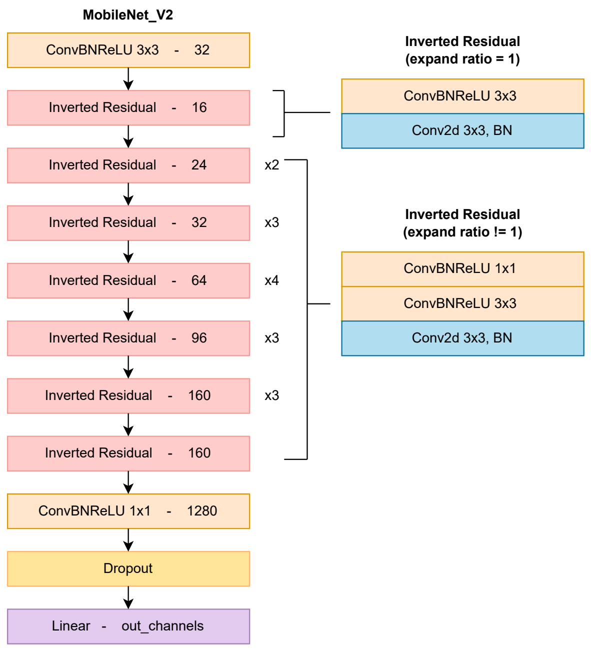
| Model (#Param) | **MnasNet** (3.10M) (Tan et al., 2019) | | | | |
|---|---|---|---|---|---|
| | Avg Latency (ms) | | Avg FLOPS | PAR (%) | |
| | GPU | CPU | | GPU | CPU |
| Conv | 0.197 | 1.160 | 2.41E+5 | 39.41 | 53.18 |
| BN | 0.228 | 0.943 | 1.20E+4 | 45.68 | 43.14 |
| ReLU | 0.096 | 0.094 | 6.54E+2 | 12.90 | 2.95 |
| MaxPool | - | - | - | - | - |
| Linear | 0.294 | 0.322 | 5.12E+3 | 1.19 | 0.30 |
| Dropout | 0.202 | 0.473 | 0 | 0.81 | 0.44 |
| AvgPool | - | - | - | - | - |
| **Model (#Param)** | **DenseNet (25.6M) (Huang et al., 2017)** | | | | |
| | Avg Latency (ms) | | Avg FLOPS | PAR (%) | |
| | GPU | CPU | | GPU | CPU |
| Conv | 0.130 | 10.583 | 7.66E+7 | 33.68 | 80.58 |
| BN | 0.190 | 1.731 | 1.58E+6 | 47.89 | 12.82 |
| ReLU | 0.074 | 0.816 | - | 18.48 | 6.07 |
| MaxPool | - | - | - | - | - |
| Linear | 0.267 | 0.254 | 8.76E+3 | 0.37 | 0.04 |
| Dropout | - | - | - | - | - |
| AvgPool | 0.143 | 3.283 | 1.51E+3 | 0.58 | 0.49 |

## G. Additional Model Descriptions

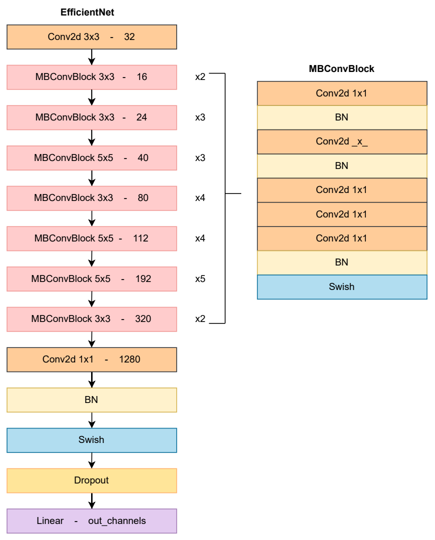
**Very Deep Convolutional Network (VGG19_bn)** (Zhang et al., 2017) We utilize a repeating sequence of stride 1 convolution, batch normalization, and ReLU layers followed by three linear layers. We initialized VGG19_bn with 2 classes and 1 input channel.


VGG19_bn
Conv2d 3x3, BN, ReLU - 64
x2
MaxPool 2x2
Conv2d 3x3, BN, ReLU - 128
x2
MaxPool 2x2
Conv2d 3x3, BN, ReLU - 256
x4
MaxPool 2x2
Conv2d 3x3, BN, ReLU - 512
x4
MaxPool 2x2
Conv2d 3x3, BN, ReLU - 512
x4
MaxPool 2x2
Linear, ReLU, Dropout - 4096
x2
Linear - out_channels

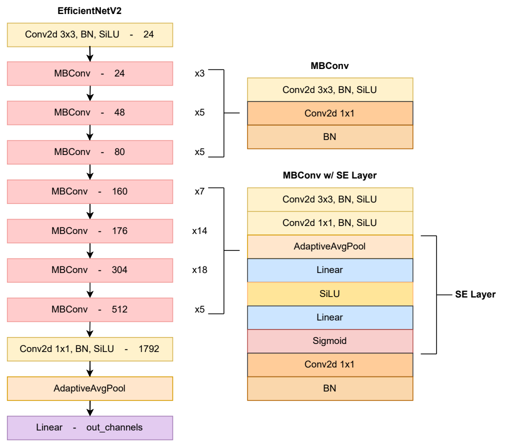
**Depthwise Seperable Convolutional Neural Network (DSCNN)** (Sandler et al., 2018) DSCNN consists of MBConvReLU blocks where each MBConvReLU block contains one pointwise MBConvReLU block, one depthwise MBConvReLU block, and one pointwise linear convolution layer. The DSCNN structure as a whole begins 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. 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. We initialized DSCNN with 2 classes and 1 input channel.



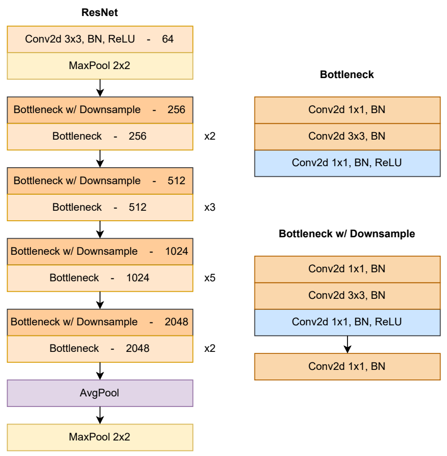
**EfficientNet** (Tan and Le, 2019) EfficientNet merges sequences of 'MBConv' blocks consisting of convolution and batch normalization layers. MBConv block sequences are searched using NAS, optimizing for FLOPS. We examined EfficientNet models with scalings 1 and 7. We initialized both EfficientNet models with 2 classes and 1 input channel.



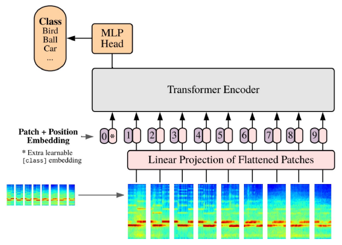
**EfficientNet-V2** (Tan and Le, 2021) These models were searched from a search space enriched with new operations, such as Fused-MBConv. A combination of training-aware neural architecture search (NAS) and scaling was used to improve both training speed and parameter efficiency. We trained and tested models with scalings 'm' and 'xl'. We initialized both EfficientNetV2 models with 2 classes and 1 input channel.



**ResNet** (He et al., 2016)
ResNet develops on CNN-type architecture by adding shortcut connections to combat vanishing gradients. We chose ResNet50 as our representative of ResNet. We initialized it with 2 classes and 1 input channel.

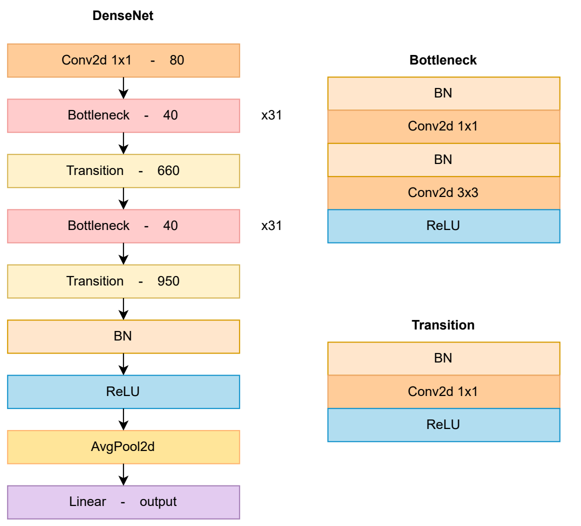


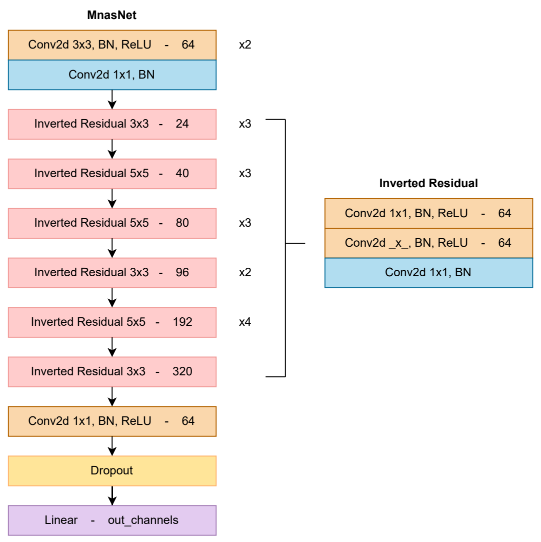
**Transformer** (Berg et al., 2021) We used the same Transformer model as in (Berg et al., 2021), specifically the implementation found at `https://github.com/wdjose/keyword-transformer/blob/master/models/kwt.py`. We integrated the Transformer model witho our codebase, and initialized it with img_x=32, img_y=32, patch_x=1, patch_y=32 to mirror the patch shape used in (Berg et al., 2021). We also initialized it with 2 classes, dim=64, depth=12 (since we used Speech Commands V1), heads=1, mlp_dim=256, pool="cls", channels=1, dim_head=64, dropout=0., and emb_dropout=0.. At the time of writing, there are reports of a bug with PyTorch GELU on mps `https://github.com/pytorch/pytorch/issues/98212`. We do not think that we were affected by this bug since Transformer performance in both digital and over-the-air testing was reasonable (albeit below the performance described by (Berg et al., 2021), but we will make note of it anyways for a potential followup in the future once the bug is resolved.


Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0 *
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

**DenseNet** (Huang et al., 2017) Densenet consists of alternating bottleneck and transition layers. We trained DenseNet with gradient accumulation where the accumulator was 8 and batch size was 8 to mirror the batch size 64 we used for all other models except the Transformer. We initialized DenseNet with depth=190, growthRate=40, compressionRate=2, 2 classes, and 1 input channel.

**MnasNet** (Tan et al., 2019) We used MnasNet1_0 as our MnasNet representative. MnasNet is designed for both image classification and object detection. We initialized it with 2 classes and 1 input channel. MnasNet uses (NAS), convolutions, depthwise separable convolutions, sequences of inverted residuals (pointwise, depthwise, linear pointwise), and finally a linear layer.




MnasNet
Conv2d 3x3, BN, ReLU - 64
x2
Conv2d 1x1, BN
Inverted Residual 3x3 - 24
x3
Inverted Residual 5x5 - 40
x3
Inverted Residual 5x5 - 80
x3
Inverted Residual 3x3 - 96
x2
Inverted Residual 5x5 - 192
x4
Inverted Residual 3x3 - 320
Conv2d 1x1, BN, ReLU - 64
Dropout
Linear - out_channels
Inverted Residual
Conv2d 1x1, BN, ReLU - 64
Conv2d _x_, BN, ReLU - 64
Conv2d 1x1, BN