Reproducing MobileNets on CIFAR-10

ECE472 Midterm Project
Richard Lee & Jonathan Lam

October 29, 2020

1 Introduction

1.1 Project Goal

We aim to independently reproduce some of the results from the original MobileNets paper [1]. The assignment states to recreate the results from the paper alone, without using guidance from an existing implementation of MobileNets. The goal is not to reproduce the exact results, or to achieve the highest possible performance, but to realize similar patterns of results as the Mobilenets authors in their original research.

Much of our time spent on this project was in adapting the architecture from the original ImageNet-centric architecture to one designed for CIFAR-10, and those design challenges are discussed in this report. Following the original setup, we were able to reproduce most of the desired results (Figures 4 and 5 from [1]).

1.2 Reproduced work

In our project, we:

1. Used the principles of MobileNets to produce a more feasible smaller-scale version to test on CIFAR-10
2. Explained the equations for parameter count and computational cost presented in the MobileNets paper
3. Reproduced Figures 4 and 5 from the original MobileNets paper

Most of the results and architecture in the original MobileNets paper was based on ImageNet, which was outside the limits of our hardware (see Hardware and time constraints).

2 Overview of MobileNets

MobileNets is a deep neural network architecture that is aimed toward reducing computational cost and parameter count, which is highly desirable in the world of IoT and embedded devices. It does this by using “depthwise separable convolutions” (first introduced in [2]), which combines a "depthwise convolution" layer and an ordinary 1x1 convolutional layer; this mimicks an ordinary convolutional layer but requires many fewer computations without compromising much cost.

The MobileNets authors also created two high-level hyperparameters to generally tune the network, which can be useful for matching hardware requirements. The first hyperparameter, α, is the width multiplier, which is a multiplicative factor of the number of input/output channels of the layers. The second hyperparameter, ρ, is the resolution multiplier; by changing the resolution of the input images, the number of computations can also be reduced. (This hyperparameter is already implicitly set by the input dimensions, relative to some set baseline resolution; i.e., changing the input resolution of the network naturally affects the number of computations.)

The authors show that α is proportional to the square of the number of parameters and computational cost, and ρ is proportional to the square of the computational cost only. They state the general relationship between these two hyperparameters and the number of weights, as well as the relationship between these hyperparameters and the computational cost, and they provide a high-level intuition as justification. In Appendix I, we delve deeper into how the number of weights and computational cost are quantified, calculated theoretically, and calculated empirically.

3 Methodology

3.1 Hardware and time constraints

Our results were run on a combination of Google Colab with GPU accelerators as well as a local machine with a single GPU. The original MobileNets paper focused on large models such as ImageNet, but given the time constraints for this midterm project, we chose to focus on CIFAR-10 for our testing, as it is another commonly used baseline and can train in a reasonable amount of time on our limited computational power. (For example, even with CIFAR-10 we ran out of free GPU usage on Colab, so training on CIFAR-100 or larger problems would likely be infeasible.)

YOU WERE SUPPOSED TO BE COMPUTATIONALLY EFFICIENT
NOT CONSUME ALL OF THE FREE COLAB GPU ACCELERATION
imgflip.com

"Jonathan Lam, circa 3AM some days before project deadline, 2020, colorized." Richard Lee, 2020.

3.2 Designing the baseline model

The MobileNets authors listed the structure of their MobileNets architecture for ImageNet in the paper. However, ImageNet images have a resolution of 224×224, while CIFAR-10 has a resolution of 32×32: there is a huge difference in the scale of the problem. ImageNet also has 1000 output classes; CIFAR-10 only has 10.

We cannot use this architecture as-is for CIFAR; if we were to keep the depth of the network (and $\rho = 1$), we would have to modify the strides because of the difference in dimensions. The MobileNets architecture for ImageNet, through the use of strides, halves the image dimensions multiple times ($224 \times 224 \to 112 \times 112 \to 56 \times 56 \to 28 \times 28 \to 14 \times 14 \to 7 \times 7$) before doing a final 7×7 average pooling layer. Each time it halves the image dimensions in the depthwise convolution layer by using stride 2, it also doubles the number of feature channels in the corresponding 1×1 convolution.

We implemented this model and called it `full_mobilenet`. It is still based around 224×224 resolution images. The only major change is that dropout was added to every MobileNet block (detailed in the regularization section) and before the final dense layer, and that the final dense layer was set to have 10 outputs (because CIFAR-10 has 10 classes). Since the final dense layer has 1024 input features, this reduces the number of weights from the original model greatly (1024×1000 to 1024×10 means a reduction of over one million weights). With 1000 classes, our model has 4,179,920 trainable weights, which matches the reported value of 4.2 million weights. Reducing this to 10 classes reduces the number of trainable parameters to 3,188,930 trainable weights.

Using the vanilla MobileNet model for ImageNet, there are two straightforward ways to adapt this to our model. The first way is to more-or-less keep the same network, but only perform strides in the latter part of the network. This means that the image feature map stays at 32×32 until the ImageNet-centric architecture reduces the image dimensions to a smaller value. While viable, this method seems to contradict the design principles of creating an "efficient and streamlined" architecture for images classification. Because of this, and due to time constraints, we did not perform testing with this modification to the architecture.

An alternative way to use the vanilla model is to set the resolution multiplier ρ to 1/7 (i.e., $32 = 224/7$). This reduces the input image resolution as well as the internal layer resolutions by 7, which aligns the ImageNet-centric architecture to 32x32 input images. We tested this model briefly, and present the results in Section 4.3.

However, it is probably clear that both of these methods, which only slightly modify a model designed for ImageNet, are probably overkill for CIFAR-10. Thus, we tried to design a simpler model from the using the building blocks and starting the ground up. The MobileNets authors do not specify how they decided on the details of their architecture (e.g., number of layers, widths, when to perform stride 2, etc.) so we attempt to mimic the overall design principles, namely utilizing the depthwise convolutions to significantly reduce required computational power and time.

We began by implementing the basic MobileNet block, which is shown in Figure 1. This block

3×3 2D regular convolution
Batch normalization
Activation (ReLU)

(a) Convolutional block

3×3 2D depthwise-separable convolution, stride 1 or 2
Batch normalization
Activation (ReLU)
1×1 2D regular convolution, stride 1
Batch normalization
Activation (ReLU)
Dropout

(b) MobileNet block

Figure 1: Comparison between our implementations of regular convolutional and MobileNet blocks

follows the structure presented by the authors of the MobileNet paper, with the exception of an additional dropout layer that we added to prevent overfitting (see 3.2.2). Using this basic MobileNet block, we created two models, CIFAR-MobileNet-v1 and CIFAR-MobileNet-v2.

Both models have a regular convolution as the first layer, followed by 5 MobileNet blocks, followed by an average pooling, a dropout, and a final softmax layer for classification. What differs between V1 and V2 is the number filters (width) of each MobileNet block. Specifically, V1 uses, in order, 64, 128, 256, 512, 1024 filters for the 5 MobileNet Blocks, while V2 uses, 32, 32, 64, 128, 256 filters. V1 is slightly closer to using the widths of the original MobileNets architecture, while V2 is much smaller (number of parameters and computational cost) while not performing significantly worse in accuracy. V2 also follows the original design a little closer, in which the number of output channels is only increased iff the depthwise convolution has stride 2, which keeps the number of parameters low. Note that the strides in a MobileNet block refer to the depthwise convolution layer (the 1×1 convolutions always have stride 1). See Appendix II for the source code implementations.

Regular convolution 3×3, stride 1, 32 filters
MobileNet block, stride 1, 64 filters
MobileNet block, stride 2, 128 filters
MobileNet block, stride 1, 128 filters
MobileNet block, stride 2, 256 filters
MobileNet block, stride 1, 256 filters
MobileNet block, stride 2, 512 filters
MobileNet block, stride 1, 512 filters
MobileNet block, stride 1, 512 filters
MobileNet block, stride 1, 512 filters
MobileNet block, stride 1, 512 filters
MobileNet block, stride 1, 512 filters
MobileNet block, stride 2, 1024 filters
MobileNet block, stride 1, 1024 filters
Average pooling 7×7
Flatten
Dropout
Fully connected
Softmax

(a) MobileNet (our implementation)

Regular convolution 3×3, strode 1, 32 filters
MobileNet block, stride 1, 64 filters
MobileNet block, stride 1, 128 filters
MobileNet block, stride 2, 256 filters
MobileNet block, stride 2, 512 filters
MobileNet block, stride 2, 1024 filters
Average pooling 4×4
Flatten
Dropout
Fully connected
Softmax

(b) CIFAR-MobileNet-v1

Regular convolution 3×3, strode 1, 32 filters
MobileNet block, stride 1, 32 filters
MobileNet block, stride 1, 32 filters
MobileNet block, stride 2, 64 filters
MobileNet block, stride 2, 128 filters
MobileNet block, stride 2, 256 filters
Average pooling 4×4
Flatten
Dropout
Fully connected
Softmax

(c) CIFAR-MobileNet-v2

Figure 2: Comparison between the different architectures we implemented

3.2.1 Validation

We randomly split a portion of the training set (20%) for a validation set to provide early stopping, but we did not do any hyperparameter tuning through validation.

The goal of this project is not to produce the highest-scoring results on CIFAR-10, but rather to more-or-less recreate the general patterns that the authors of MobileNets observed. Our baseline model just needs to produce sufficiently decent results on CIFAR-10 so that we know it works. Finding optimal hyperparameters (e.g., number of MobileNets blocks, best width multipliers, best depth multipliers, best layers to perform stride 2 convolutions, etc.) for a MobileNets-based architecture on CIFAR-10 was beyond the scope of this project and open for further research.

3.2.2 Regularization

The MobileNets authors mention that explicit regularization (e.g., through dropout and L1/L2-regularization) is not necessary because a simpler model tends not to overfit. While MobileNets may be considered a relatively simple model for ImageNet, this is not the case for CIFAR-10, and we observed serious overfitting. Thus, our MobileNet implementations include several dropout layers.

While our professor notes that it is usually strange to include both batch normalization and dropout layers, we included batch normalization to keep in line with the original model, and dropout to enforce stronger regularization.

3.2.3 Initializers and generic model hyperparameters

We let the dense and convolutional layers (both regular and depthwise separable) use the default initializers. At the time of writing, the defaults for all of these layers is the Glorot uniform initialization method.

For all of the models, we used the Adam optimizer (the original paper uses RMSprop) and mostly used the default learning rate 0.001 (except for CIFAR-MobileNet-v2, in which we saw a lot of instability in the training, so the learning rate was reduced to 0.0001).

The dropout values were not chosen by any strict validation scheme; the dropout rate of 0.2 was somewhat-arbitrarily chosen, and it seemed to work well regularizing most of our models.

3.2.4 Image preprocessing

As with regularization, the authors of MobileNets stated that image augmentation was unnecessary because the smaller model architecture did not have trouble with overfitting. To simplify the process of training many models, we did not perform image augmentation. This provided decent accuracy, and it did not seem to introduce problems with overfitting (as omitting regularization might).

The only preprocessing we performed was a min-max scaling to $[0, 1]$ (i.e., dividing pixel values by 255).

3.3 Implementing the MobileNet hyperparameters

The MobileNet authors use the following notation to indicate a given model: "α MobileNet-*resolution*". (The default implementation, with $\alpha = 1$ and $\rho = 1$ (full 224×224 pixel resolution is denoted 1.0 MobileNet-224). We follow this convention as well. To be overly explicit, we will use the name "CIFAR-MobileNet" when referring to our reduced MobileNet structure, although it should be clear from the smaller resolution.

To implement α, we had to multiply the 1.0 CIFAR-MobileNet-32 layer widths by α (i.e. the number filters/output channels). This was relatively straightforward, so long as we were careful that the number of filters rounded to an integral number. We tested α values 0.5, 0.75, 0.8, 0.9, 1, and 2.

To implement ρ, the changes that have to be made are:

- The images have to be rescaled during preprocessing (we used `tensorflow.image.resize` to accomplish this).
- The input layer of the network has to expect this new image size.
- The final average pooling layer pool size has to be scaled down by ρ. This value is rounded to an integer, and for any meaningful output, must be at least one.

With that in mind, we chose ρ values of 16/32, 20/32, 24/32, 28/32, 30/32, and 1. The common denominator of 32 was chosen to align with the 32 pixel image size of CIFAR-10, so the numerators (16, 20, 24, 28, 30, 32) describe the input image sizes.

4 Results and discussion

4.1 Verifying the paper's results for the full-size model

Since our goal is to reproduce some of the results of the original MobileNets paper, we attempted to provide some verification that our model's properties matched those stated in the paper. In particular, Table 4 in the original paper give a summary of the mult-adds and millions of parameters (weights) for the 1.0-MobileNet-224 model and the equivalent model with regular convolutional blocks. To have a fair comparison, we ran the profiler to get the number of million mult-adds and trainable weights from `full_mobilenet` with a 1000-width final dense layer (which was intended to be exactly the same model as what the authors described); that same network with regular convolutional blocks rather than MobileNet blocks; and the provided implementation of MobileNets in Keras. `tf.keras.applications.MobileNet`. The results are displayed in Table 1. Explanation of the calculations of mult-adds and parameters, as well as the profiling code is given in Appendix I. We note that our value of "mult-adds" is given by the approximation that the number of mult-adds is half the number of FLOPs (Appendix I), which means that we are slightly overcounting number of mult-adds in our empirical profiling calculations. We get complete agreement with the number

	Million Mult-Adds		Million Parameters	
	Reported	Empirical	Reported	Empirical
Conv MobileNet	4866	4898	22.9	22.9
MobileNet (Keras)	569	576	4.2	4.2
MobileNet		573		4.2

Table 1: Comparison of MobileNet empirically-calculated mult-adds and parameters to the values reported in the paper. Conv MobileNet is `full_mobilenet` with `use_mobilenet_blocks=False, alpha=1, rho=1`; MobileNet is `full_mobilenet` with `use_mobilenet_blocks=True, alpha=1, rho=1`; and Keras MobileNet is the `tf.keras.applications.MobileNet` implementation. Empirical values were computed using the `tf.compat.v1.profiler` profilers.

of reported and empirical million parameters (to the number of significant figures reported by the authors).

4.1.1 Fidelity of our model to the original model

We also compared our results against the Keras MobileNet implementation. While our assignment explicitly instructs us not to look at existing implementations of the work published in the research paper for guidance, we thought that using this would be an interesting way to gauge our model's fidelity to what the authors described. This is the only place we used an existing MobileNet implementation. We only used it to look at profiling and layer summary details and did not look at the source code nor use it to influence our model.

When examining the model summaries of our model versus the builtin implementation's model summary (i.e., by examining `tf.keras.Model::summary()`), we noticed a few differences. These were only an after-the-fact realization and we didn't incorporate these changes into our model.

- The initial convolutional layer in the Keras implementation also includes batch normalization and an activation. We misinterpreted the original paper and only have a convolutional layer there, since it is not described like a regular MobileNet block in the MobileNet architecture described in the original paper.
- The keras implementation's final fully-connected (FC) layer uses a 1×1 convolution operation, while ours uses a dense layer. We don't believe that there's any real difference in the operation here, but it's interesting to see that a fully-connected layer can be done in multiple ways.
- As stated before, we use dropout in every convolutional block. The keras implementation only has a single dropout layer before the final classification step.
- The convolutional layers (both depthwise and regular) in the keras implementation do not have bias terms (`use_bias=False`), whiles ours do (using the default `use_bias=True`). This slightly reduces the number of weights and computations. We believe that the bias is omitted because the convolutional layers are immediately followed by batch normalization layers, which would introduce a learned bias already. We didn't think of this when concocting our models.

Despite these (relatively minor) differences, we believe that our implementation of the full MobileNet, and thus the ideas transmitted to the reduced CIFAR MobileNets, are sound.

4.2 Reproducing the effect of hyperparameters

In the original MobileNets paper, the authors had two primary results: comparing MobileNets to existing deep convolutional layers models with regular convolutional layers (e.g., VGG and AlexNet); and varying α and ρ and viewing the relationships between those hyperparameters, their test accuracies, number of parameters, and computational cost. We do not explore the first goal; our primary focus is on the latter.

We tested 72 total models formed from the Cartesian product of $model \in \{V1, V2\}$, $\alpha \in \{0.5, 0.75, 0.8, 0.9, 1, 2\}$, and $\rho \in \{16/32, 20/32, 24/32, 28/32, 30/32, 1\}$. (In contrast, the MobileNets paper uses $\alpha \in \{0.25, 0.5, 0.75, 1\}$ and $\rho \in \{128/228, 160/228, 192/228, 1\}$.) These parameters values were arbitrarily chosen. We chose to test $\alpha = 2$ to see if the additional filters in each MobileNet block would provide any improvement in test accuracy; the original paper only went up to $\alpha = 1$.

(On the other hand, increasing ρ past 1 is not meaningful, as you cannot simply increase resolution to gain new information.)

The following plots and tables describe the relationships between hyperparameters, accuracy, number of trainable weights (i.e., size of the model in memory), and computational cost (multiplied in mult-adds).

4.2.1 Accuracy, number of weights, computational cost vs. α, ρ

The direct effect of hyperparameters on the other three metrics (accuracy, number of parameters, and computational cost) are shown in Tables 2, 3, and 4, respectively. We can make a number of observations from these tables:

- The relationship between α and accuracy, or between ρ and accuracy (Table 2), is a positive direct relationship. This data is noisy relative to what the MobileNets authors observe, as neither relationship is monotonically increasing, but the general trends are still apparent. The monotonic relationship is more noticeable for CIFAR-MobileNet-v2 than for CIFAR-MobileNet-v1.
- The number of parameters is only affected by α (and not ρ). Intuitively, this makes sense as kernel size is independent of feature map resolution. The relationship between number of parameters and α is roughly a squared relationship.
- The computational cost is roughly proportional to the square of both α and ρ, as expected.

4.2.2 Accuracy vs. Number of Trainable Weights

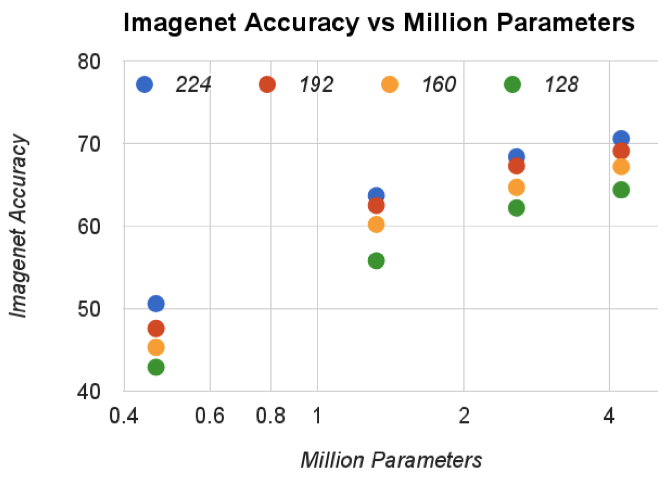


Figure 3: Figure 5 from the original MobileNets paper

Figure 5 (in our report) is a visual representation of Table 2, and plots test accuracies against number of trainable weights. These plots are intended to mirror Figure 5 in the original MobileNets paper (included here as Figure 3). For both CIFAR-MobileNet-v1 and CIFAR-MobileNet-v2, we notice a general increase in accuracy as the number of weights increased. As noted before, these results are "noisier" than those of the original paper. We attribute this to the inherent randomness in model training, as well as the fact that CIFAR-10 is a much smaller dataset with fewer classes, so we would expect there to be less variation than in that of ImageNet.

It is clear from Figure 5 that changing ρ while keeping α constant doesn't change the number of parameters, but still changes the accuracy. We also connected the points in the same ρ-series (i.e., points with the same ρ) in order to more easily see how changing α affects ρ-series. We observe that the slopes of the ρ-series in the v2 model are much more consistent and steep than the slopes in the v1 model.

We can also observe that for any fixed value of α, increasing ρ generally tends to increase the accuracy, except for higher α in v1. To address this anomaly, we posit that the v1 models are expressive enough in their vanilla forms ($\alpha \leq 1$ and $\rho \leq 1$) for the CIFAR-10 dataset (i.e. little additional information is added by the extra filters). This shows that for a MobileNet with a given depth, it is important to sweep over α to test what width works best without overfitting.

4.2.3 Accuracy vs. Number of Mult-Adds (Computational Cost)

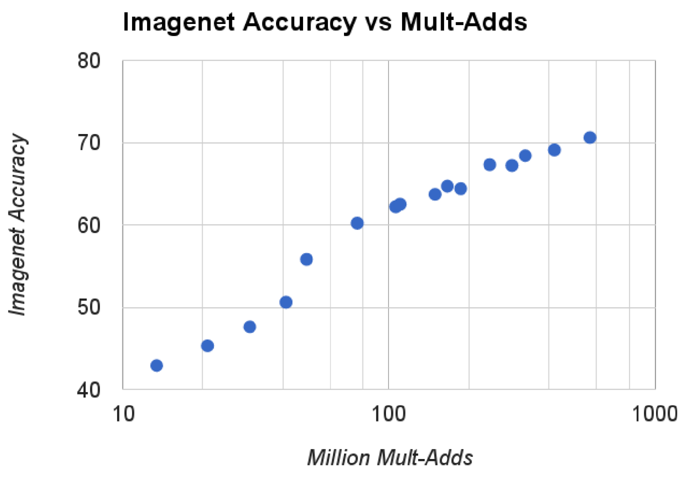


Figure 4: Figure 4 from the original MobileNets paper

In Figure 4 of the original MobileNets paper (included here as Figure 4), they describe the number of mult-adds that each of the models used to achieve the various test accuracies to measure the trade off between computational cost and test accuracy. They found a log-linear relationship between the number of mult-adds and test accuracy.

We created similar plots for CIFAR-MobileNet-v1 and CIFAR-MobileNet-v2 in Figure 6. Once again, the results are noisier than that of the original paper, but a log-linear relationship can still be drawn, especially for v2 - as the computational cost increases, a general increase in test accuracy can be expected. However, with v1, we only noticed a positive log-linear relationship for a majority of the models. We draw a second log-linear regression line, as the general trend shows that there are negative returns despite increasing the computational cost of the models.

Comparing against Table 4, we can observe that primarily models with the highest α of 2 and 1 also contain the highest computational cost (intuitively, more filters results in more computations). This result supports our previous conclusion that the models in v1 are expressive enough for CIFAR-10 and additional parameters do not provide additional useful information.

We also provide Figure 7, which shows the same data plotted on the same axes as in Figure 6, but with the ρ-series distinguished like in Figure 5. The $\alpha = 1$ and $\alpha = 2$ values in this representation are the two rightmost points on each ρ-series, which we can tell do not have higher accuracies than lower α values within the same ρ-series in the v1 model. Another interesting conclusion that can be drawn from this modified chart is which value of ρ to use to maximize accuracy efficiency (i.e., highest ratio of accuracy to number of mult-adds); this graphically would be indicated by which ρ-series is highest for any given number of mult-adds. However, it is unclear by the plots which ρ-series is best, because the lines overlap often. More research could be done on quantifying this effect.

4.3 Comparing Convolution Types

We noted in Section 3.2 that it would be possible to use the vanilla MobileNets model (designed for a 224×224 resolution image) by using a $\rho = 1/7$, and we present the results in Table **??** below. We did not test out how the hyperparameters affected this model, since it is larger than our CIFAR-specific models, but we ran it with regular convolution blocks and with MobileNet blocks to compare the two.

	MobileNet with regular convolution	MobileNet
Test dataset accuracies (%)	76.32	73.51
Test dataset evaluation time (s)	3.076	1.378
Millions of parameters	56.6	6.43
Millions of mult-adds	28.3	3.23

We note that the model using regular convolutions attains a slightly higher test accuracy than when using depthwise convolutions, but the depthwise convolutional model is able to evaluate the entire test set in less than half the time of the plain convolution model. This supports the authors' claim that depthwise convolutions (and thereby MobileNets) is more computationally efficient for image classification than other conventional models, while still achieving similar accuracy.

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	79.59	82.34	78.59	82.05	79.3	78.47
	1	84.24	79.13	81.83	81.81	78.53	77.54
	0.9	83.54	81.04	82.56	79.57	78.85	78.21
	0.8	84.6	80.05	80.01	80.75	79.05	76.15
	0.75	82.67	81.74	79.81	80.17	78.74	76.52
	0.5	80.42	79.58	79.65	78.56	76.56	72.35

(a) CIFAR-MobileNet-v1

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	76.87	73.39	72.58	76.06	76.69	71.68
	1	72.51	67.35	68.53	68.66	69.87	68.22
	0.9	68.68	71.16	68.28	68.72	66.26	65.96
	0.8	65.61	69.31	68.78	64.64	65.28	64.13
	0.75	66.89	66.24	66.81	61.7	63.93	65.75
	0.5	63.45	64.12	60.87	61.24	61.76	59.73

(b) CIFAR-MobileNet-v2

Table 2: Test accuracy (%) w.r.t. ρ and α.

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	2851466					
	1	727370					
	0.9	590014					
	0.8	468839					
	0.75	414586					
	0.5	189098					

(a) CIFAR-MobileNet-v1

		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	197130					
	1	53514					
	0.9	43709					
	0.8	35242					
	0.75	31690					
	0.5	15498					

(b) CIFAR-MobileNet-v2

Table 3: Number of model parameters w.r.t. ρ and α. (ρ is independent of model parameter count.) Values were empirically determined using the `tf.compat.v1.profiler` profiler.

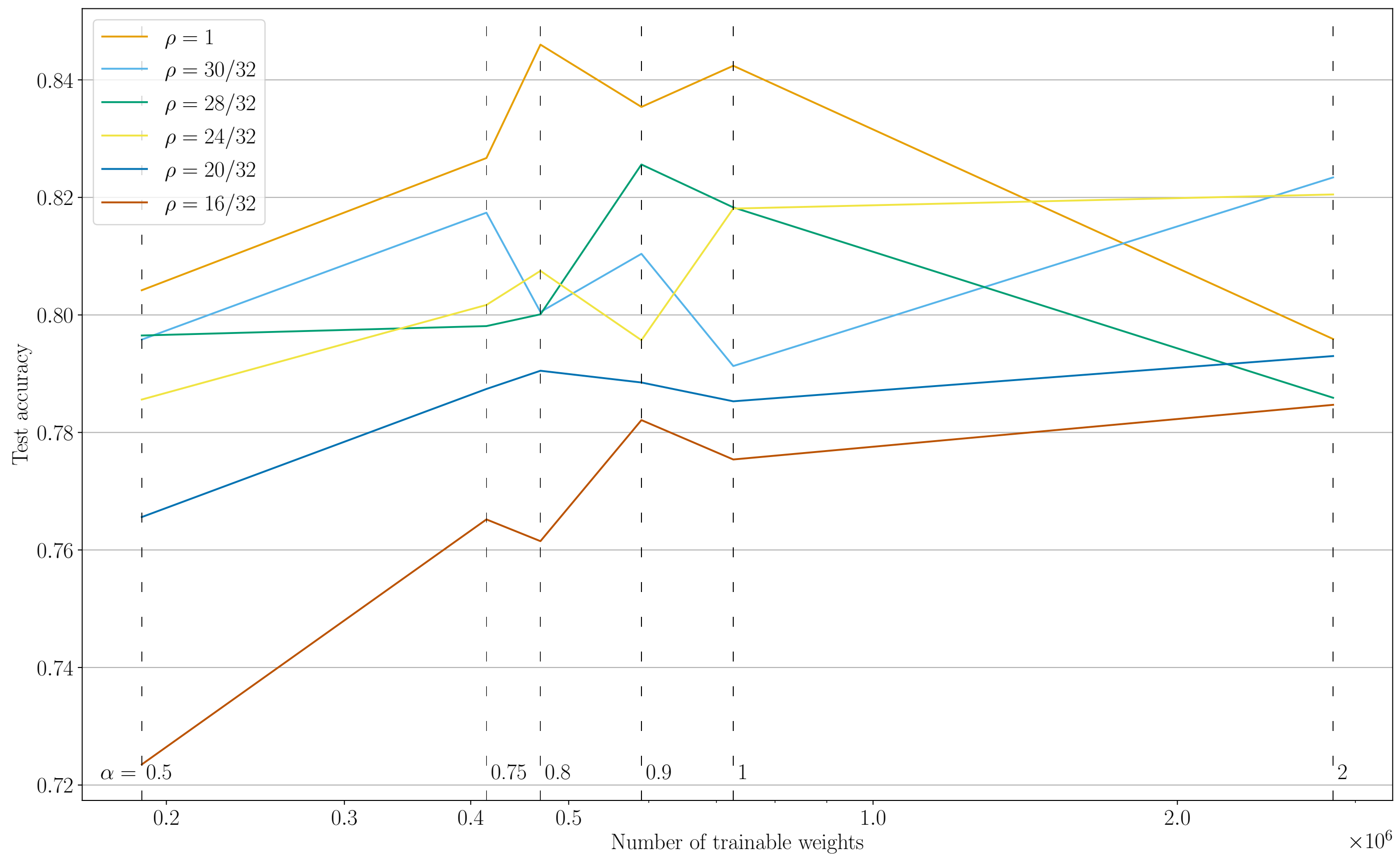
		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	154.64	144.93	127.92	90.02	70.43	43.86
	1	40.98	38.41	34.01	24.27	19.16	12.33
	0.9	33.36	31.28	27.72	19.83	15.7	10.17
	0.8	26.78	25.12	22.27	15.99	12.68	8.28
	0.75	23.92	22.43	19.9	14.31	11.36	7.45
	0.5	11.41	10.7	9.53	6.96	5.57	3.77

(a) CIFAR-MobileNet-v1

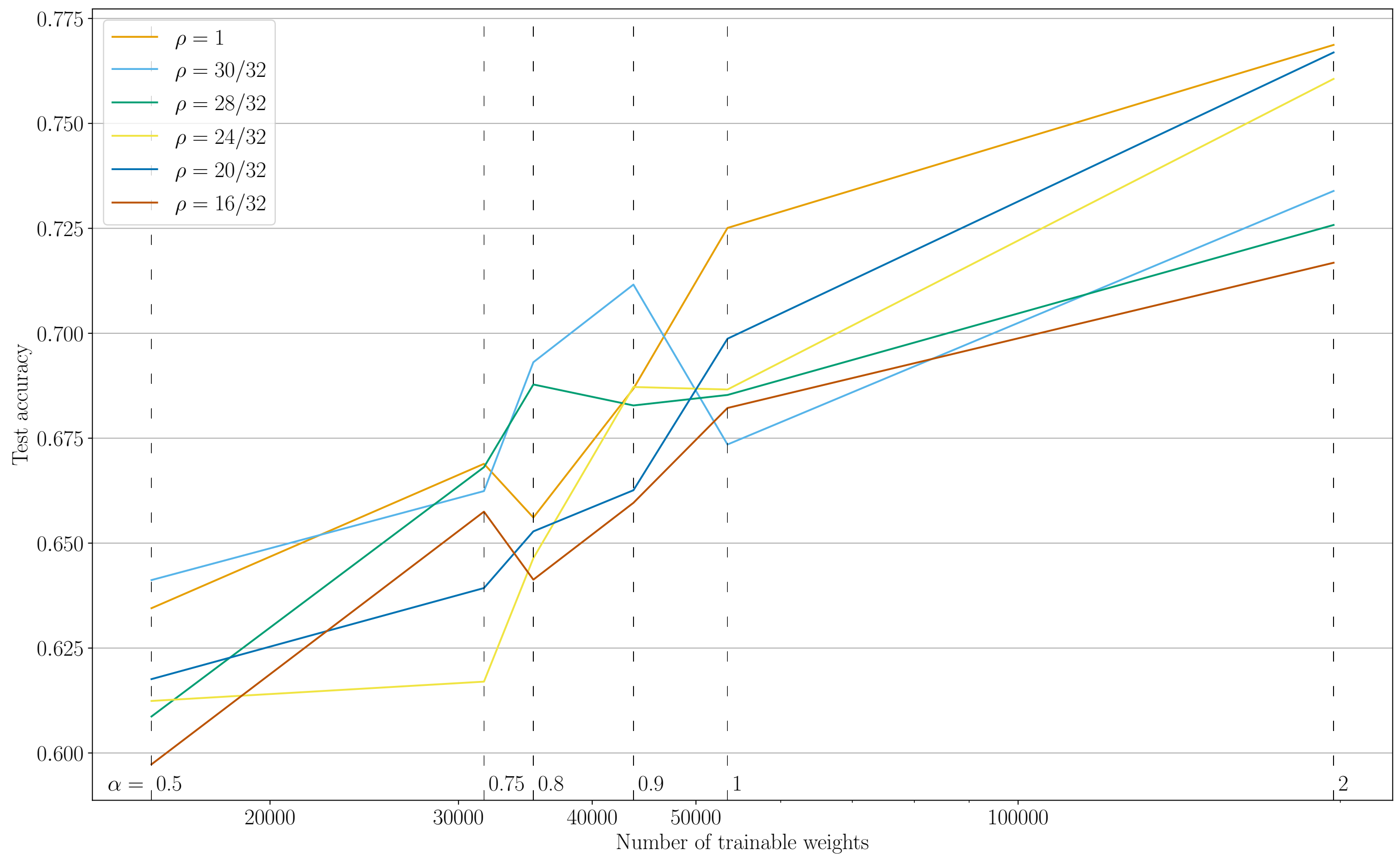
		ρ					
		1	30/32	28/32	24/32	20/32	16/32
α	2	18.3	16.63	14.56	10.38	7.64	4.73
	1	5.43	4.92	4.3	3.08	2.25	1.4
	0.9	4.4	3.98	3.48	2.49	1.82	1.13
	0.8	3.65	3.29	2.88	2.07	1.51	0.94
	0.75	3.38	3.05	2.67	1.92	1.39	0.87
	0.5	1.79	1.61	1.41	1.01	0.73	0.46

(b) CIFAR-MobileNet-v2

Table 4: Millions of mult-add operations w.r.t. ρ and α. Values were empirically determined using the `tf.compat.v1.profiler` profiler.

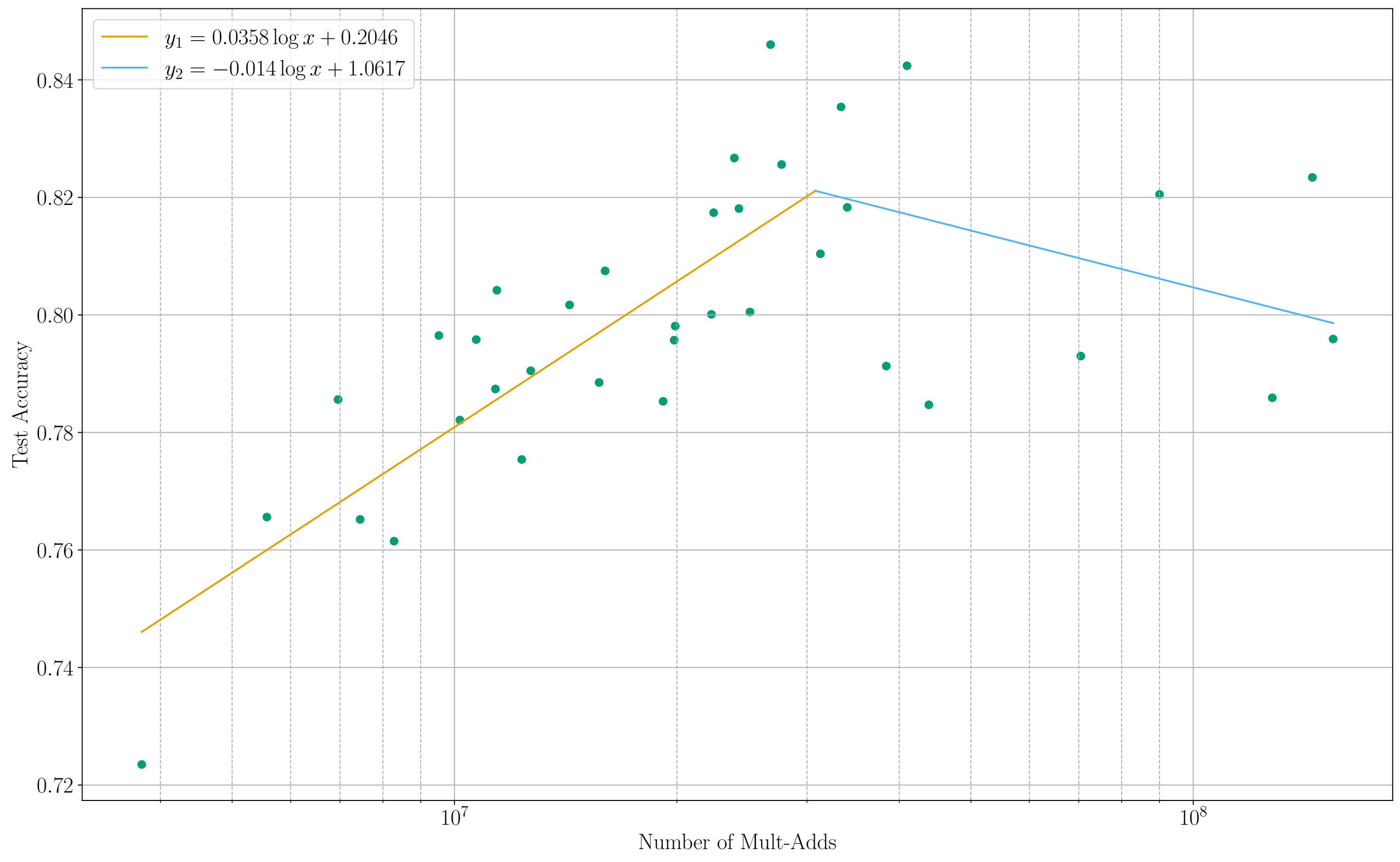


(a) CIFAR-MobileNet-v1

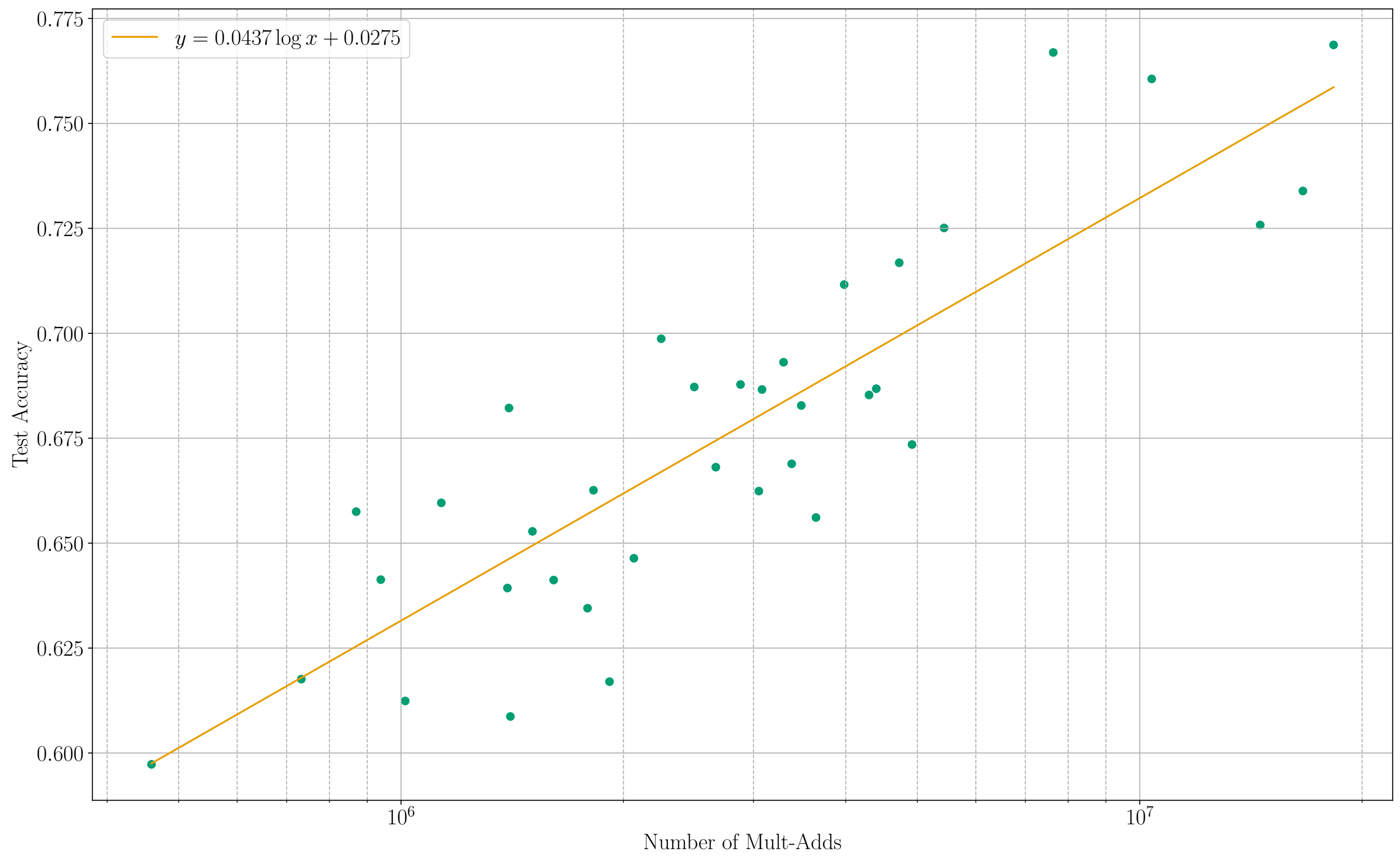


(b) CIFAR-MobileNet-v2

Figure 5: Accuracy vs. number of trainable weights for CIFAR-MobileNet models. ρ-series (with varying α) are shown as lines. Models with the same α share the same number of weights and lie on the same vertical dotted line.

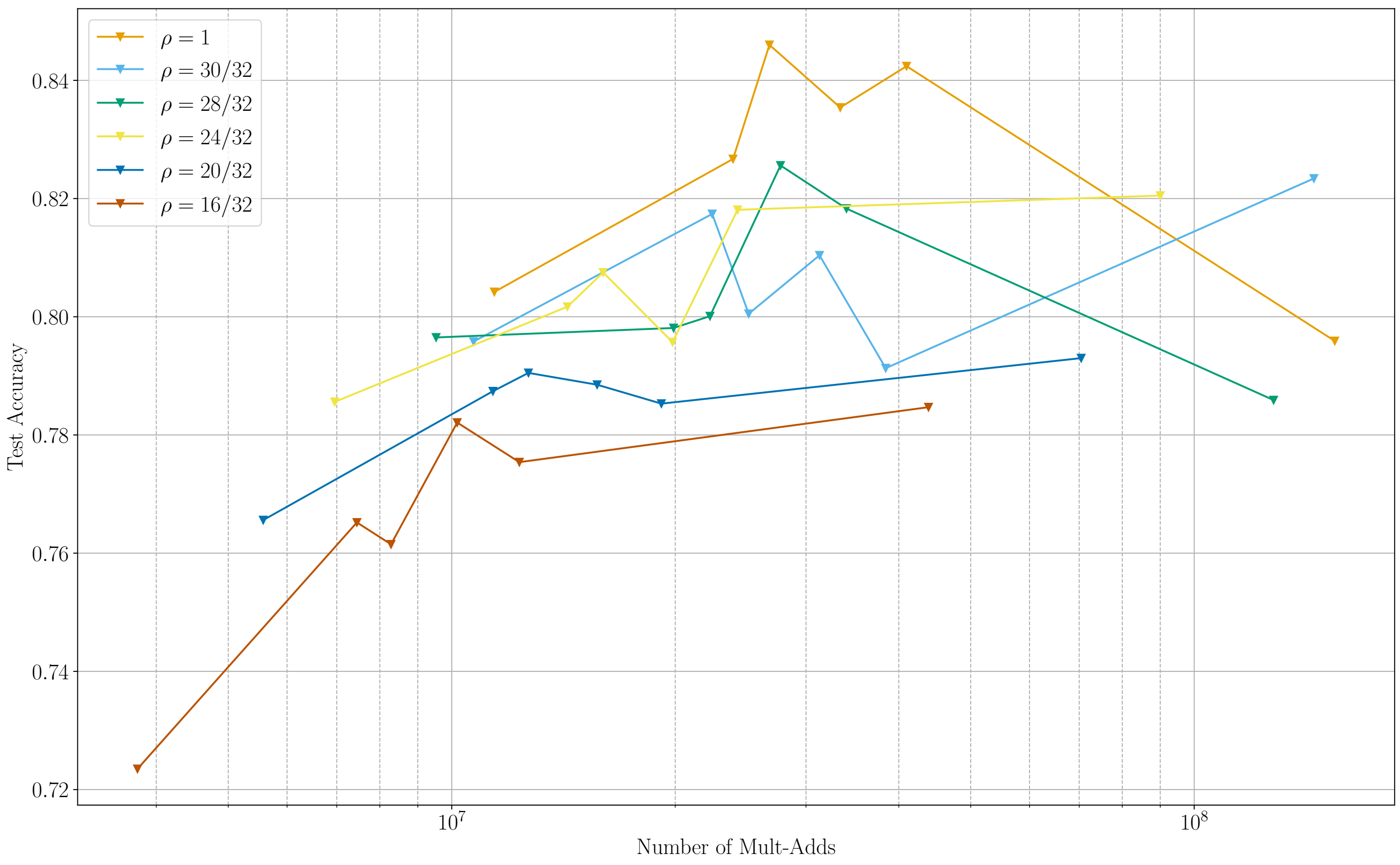


(a) CIFAR-MobileNet-v1

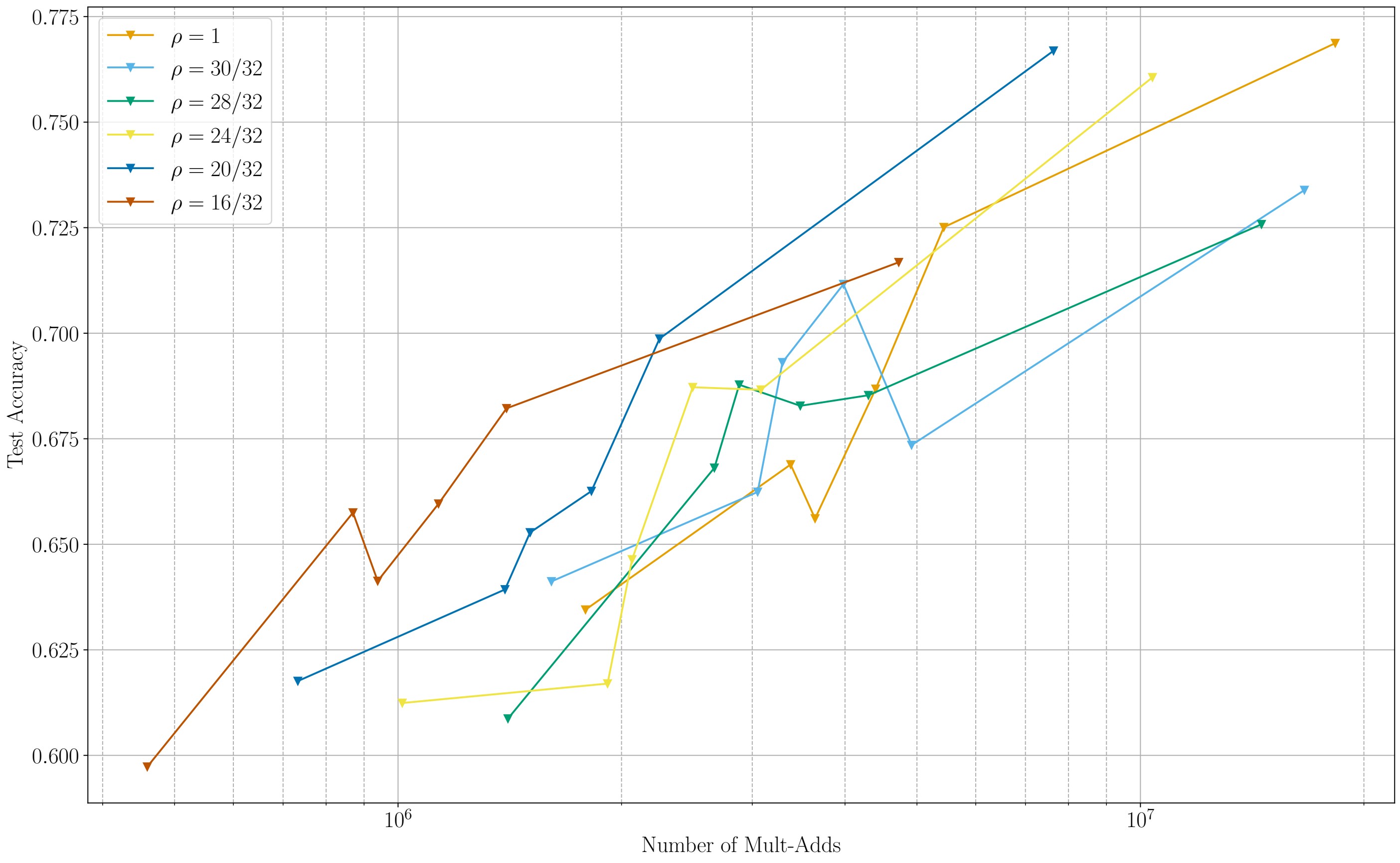


(b) CIFAR-MobileNet-v2

Figure 6: Accuracy vs. number of mult-adds for CIFAR-MobileNet models. Logarithmic regressions (logarithmic w.r.t. the number of mult-adds) are shown. (For v1, the logarithmic regressions were performed on the data left and right of the highest-accuracy point and were plotted to intersect.)



(a) CIFAR-MobileNet-v1



(b) CIFAR-MobileNet-v2

Figure 7: Accuracy vs. number of mult-adds for CIFAR-MobileNet models. ρ-series (with varying α) are shown as lines.

5 Conclusions and further inquiry

We were able to design a MobileNet implementation for CIFAR-10 incorporating the design features present in the original model designed for ImageNet. Our work is mainly concerned with comparing different versions (i.e., by varying the MobileNet parameters) of the same base MobileNet implementation, rather than being concerned with comparing MobileNets to other existing deep convolutional networks. We rederived the equations that the authors proposed for number of parameters and computational cost (as a function of the hyperparameters α and ρ) and verified their correctness against profiling data. We were able to train 72 different models (36 of CIFAR-MobileNet-v1 and 36 of CIFAR-MobileNet-v2), and used this to reproduce two figures from the original paper. While our plots are noisier than the results reported in original paper, the reproductions show the same general patterns.

Our original plan for this project also included running our MobileNet implementation on embedded hardware (as this would demonstrate MobileNet's intended purpose), but we were not able to complete this on top of the other goals we had. Doing so would be a good continuation of this model.

Further research into time to train and time to infer using this model could be very fruitful. Neither we nor the authors of MobileNet provide statistics on either of these. Even better, examining how efficient depthwise convolution may be (as opposed to regular convolutions) based on how they utilize low-level linear algebra frameworks (e.g., BLAS) would be an interesting exploration.

References

[1] Howard, Andrew G., et al. "MobileNets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861* (2017).

[2] Sifre, Laurent, and Stéphane Mallat. "Rigid-motion scattering for image classification." *Ph. D. thesis* (2014).

[3] Malcolm. "Answer to how to calculate a Mobilenet FLOPs in Keras." *Stack Overflow,* Web, 15 Aug 2018. https://stackoverflow.com/a/51866343/.

6 Appendix I: Convolutional Layer Analyses

Because of the relevance of the fundamental depthwise-separable convolution to this paper, and for our own academic purposes, we decide to go more in-depth with an analysis of the convolutional layer types. The MobileNets paper does not go into detail about how they calculate number of weights and number of mult-adds, and since we attempt to reproduce their results, we describe how we understand and calculate these metrics.

We use the same notation as the book, i.e., for any given layer: $D_f :=$ input feature map size (length); $D_k :=$ kernel filter size (length); $M :=$ the number of input channels; $N :=$ the number of output channels. (The filters and feature maps are assumed to be square to make the calculations a little simpler, but this can be extended to any rectangular map.) I.e., if the input to a layer were a $32 \times 32 \times 256$ feature map, and number of output channels is 512, and the kernel filter size is 3×3, then $D_f = 32$, $D_k = 3$, $M = 256$, and $N = 512$. In the case of depthwise convolution, $M = N$. In the case of a 1×1 convolution, $D_k = 1$.

A “mult-add” is not defined in the original paper, but we assume that it is some computational-efficient multiplication-addition combination. The authors of the original paper mention that it is important to not only be able to measure the number of FLOPS or similar measure, but also take into consideration how efficiently those operations can be carried out in hardware, which lends itself to the idea of general matrix multiplies (GEMMs) and mult-adds.

6.1 Regular convolution

```
pixels ← input feature map, dimensions (Df, Df, M)
filters ← array of convolutional kernel filters, dimensions (N, M, Dk, Dk)
biases ← array of biases for each output channel, dimensions (N)
out ← zero-initialized output feature map, dimensions (Df, Df, N)
for i, j ← pixels
    for n ← 1..N
        for m ← 1..M
            out[i, j, n] += conv2d(filters[N, M, :, :], pixels[i:i+Dk, j:j+Dk, m])
        endfor
        out[i, j, n] += biases[n]
    endfor
endfor
return out
```

Figure 8: Pseudocode for a regular convolution

From Figure 8, we can see that there are NMD_k^2+N learnable weights in the kernel and biases. Each regular convolution takes D_k^2 multiplications and D_k^2-1 additions. It is added to `out[i, j, n]` so this adds one more addition; this is $2D_k^2$ FLOPs or D_k^2 mult-adds. This gets multiplied by the number of input channels, number of output channels, and the number of pixels for a total of $D_f^2NMD_k^2$ mult-adds. If the convolutional layer uses a bias term (i.e., toggled with `tf.keras.layers.Conv2D`'s `use_bias` parameter), then there are an additional D_f^2N additions. This latter term is dominated by the former if M or D_k is large (in our case, M gets very large), so we can safely ignore it from the overall mult-add calculations. (Also, while the default value for `use_bias` is `True`, and this is the value we use, in the MobileNets keras implementation there is no bias added. This is likely due to the fact that the layer is followed immediately by a batchnorm layer, which will provide a learned bias. We did not consider this when originally constructing our model design.)

6.2 Depthwise convolution

From Figure 9, there are $MD_k^2 + M$ learnable weights in the kernel and biases. As with regular convolution, a single convolution takes D_k^2 mult-adds. However, they are not combined in a sum to make an output channel; instead, each output channel is an input channel with one convolutional filter applied to it. Thus this makes $D_f^2MD_k^2$ mult-adds. Again, the number of operations from addition of a bias term are not important (and the keras implementation doesn't use the bias in depthwise convolution).

```
pixels ← input feature map, dimensions (Df, Df, M)
filters ← array of convolutional kernel filters, dimensions (M, Dk, Dk)
biases ← array of biases for each channel, dimensions (M)
out ← output feature map, dimensions (Df, Df, M)
for i, j ← pixels
    for m ← 1..M
        out[i, j, m] ← conv2d(filters[M, :, :], pixels[i:i+Dk, j:j+Dk, m])
        out[i, j, m] += biases[m]
    endfor
endfor
return out
```

Figure 9: Psuedocode for a depthwise-separable convolution

6.3 Including hyperparameters in calculations

This is fairly self-explanatory and is explained in the original paper. The width multiplier α scales the input and output feature sizes. The resolution multiplier ρ scales the feature map size. Since the feature map does not affect the number of weights, ρ is independent of the number of weights in the model (i.e., the memory requirement of the model). The computational cost of the regular convolutional layers (which account for most of the calculations in the MobileNet model) is jointly proportional to α^2 and ρ^2.

6.4 Summary of equations and example calculation

Layer type	Input shape	Weight count
2D depthwise convolution	$112 \times 112 \times 64$	640
Batch Normalization	$56 \times 56 \times 64$	256
ReLU	$56 \times 56 \times 64$	0
2D convolution	$56 \times 56 \times 64$	8320
Batch Normalization	$56 \times 56 \times 128$	512
ReLU	$56 \times 56 \times 128$	0

Table 5: Sample MobileNet block. This block has a 3×3 2D depthwise convolution with stride 2, and a 1×1 convolution that doubles the number of output channels, a typical pattern in the MobileNet blocks. (This is the second MobileNet block in `full_mobilenet` with $\alpha = \rho = 1$.) Note that the majority of the calculations and parameters lie in the 1×1 convolutions.

The calculations for number of weights and computational cost are summarized in the table below. (S is the stride length.)

$$\begin{aligned}
|W_{\text{regular conv.}}| &= D_k^2 \times \alpha M \times \alpha N + \alpha N \\
C_{\text{regular conv.}} &\approx \rho^2 D_f^2 (\div S^2) \times \alpha M \times \alpha N \times D_k^2 \\
|W_{\text{depthwise conv.}}| &= D_k^2 \times \alpha M + \alpha M \\
C_{\text{depthwise conv.}} &\approx \rho^2 D_f^2 (\div S^2) \times \alpha M \times D_k^2
\end{aligned}$$

Using these equations, we can estimate that the cost of the MobileNet block in Table 5 is:

$$C \approx (1)^2(112)^2 \times (1)(64) \times (3)^2 \div (2)^2 + (1)^2(56)^2 \times (1)(64) \times (1)(128) \times (1)^2 = 27496448$$

The value given by the profiler is 27806660 mult-adds (55613319 FLOPs $\div 2$). There is only a 1% error between theory and practice (even with some estimates in the theoretical calculation), which is not bad. It is also not hard to check that the number of weights match the values given by the equations. Now that we have shown by example that these equations are correct, we defer the rest of the calculations to the profiler.

6.5 Empirical calculations

Rather than calculating all of the models' weight counts and complexities by hand, we used the profiler from Tensorflow's v1 (compatibility module). This not only saved us a lot of trouble, but gets closer to the actual cost complexities of the models; our calculations only accounted for the convolutional layers, which form the bulk of both the weights and the computational cost, but there are still many calculations and weights in the batchnorm and dense layers.

The code used for profiling came from an answer on Stack Overflow [3]. (This source also gave us the estimate for mult-adds based on FLOPs.) It uses Tensorflow v1 code, so we have to use the compatibility module.

```
session = tf.compat.v1.Session()
graph = tf.compat.v1.get_default_graph()

with graph.as_default():
    with session.as_default():
        model = model_class(**model_parms,
            input_tensor=tf.compat.v1.placeholder('float32',
                shape=(1, int(model_parms['rho']*32), int(model_parms['rho']*32), 3)))

        run_meta = tf.compat.v1.RunMetadata()
        opts = tf.compat.v1.profiler.ProfileOptionBuilder.float_operation()
        flops = tf.compat.v1.profiler.profile(graph=graph, run_meta=run_meta,
            cmd='op', options=opts)

        opts = tf.compat.v1.profiler.ProfileOptionBuilder.trainable_variables_parameter()
        params = tf.compat.v1.profiler.profile(graph=graph,
            run_meta=run_meta, cmd='op', options=opts)

        flops, num_params = flops.total_float_ops, params.total_parameters
```

where `model_class` is a factory function for the model we want to analyze. To be consistent with the original authors, we report our results in mult-adds. Since the profiler outputs FLOPs, we make the approximation that one mult-add is approximately two FLOPs; this is justified by the fact that the vast majority of calculations occur in the regular convolution layers and dense layer, and in those layers most operations can be modeled as a multiplication followed by an addition. This gives us results almost exactly equal to the reported values in the original paper, so this is likely a sound assumption.

7 Appendix II: Source Code

```python
# define a mobilenet block
depthwise_conv = tf.keras.layers.DepthwiseConv2D
regular_conv = tf.keras.layers.Conv2D
batchnorm = tf.keras.layers.BatchNormalization
activation = tf.keras.layers.ReLU
average_pooling = tf.keras.layers.AveragePooling2D
flatten = tf.keras.layers.Flatten
dropout = tf.keras.layers.Dropout
dense = tf.keras.layers.Dense

# original convolutional block, can be used when calculating the
# difference in the number of parameters or comparing accuracy;
# returns a list that can be dropped into a keras Model.Sequential
def regular_conv_block(stride_2, out_channels, with_dropout=True):
    return [
        regular_conv(int(out_channels), (3, 3), padding='same',
                     strides=((2, 2) if stride_2 else (1, 1))),
        batchnorm(),
        activation()
    ]

# mobilenet block, as described in the original paper;
# also returns a list that can be used in a keras Model.Sequential
def mobilenet_block(stride_2, out_channels, with_dropout=True):
    return [
        depthwise_conv((3, 3), padding='same',
                       strides=((2, 2) if stride_2 else (1, 1))),
        batchnorm(),
        activation(),
        regular_conv(int(out_channels), (1, 1)),
        batchnorm(),
        activation(),

        # not part of the original model
        dropout(0.2)
    ]
```

Figure 10: Convolutional and MobileNet blocks

```python
# full MobileNet model; run this with rho=1/7 to work with CIFAR-10 without
# further modification (designed for 224x224 image);
# input_tensor necessary when profiling model
def full_mobilenet(alpha=1, rho=1, use_mobilenet_block=True, input_tensor=None):
    block = mobilenet_block if use_mobilenet_block else regular_conv_block

    model = tf.keras.Sequential([
        *([tf.keras.Input((int(rho * 224), int(rho * 224), 3))] \
            if input_tensor is None else [tf.keras.Input(tensor=input_tensor)]),

        regular_conv(int(alpha * 32), (3, 3), padding='same', strides=(2, 2)),

        *block(False, int(alpha * 64)),
        *block(True, int(alpha * 128)),
        *block(False, int(alpha * 128)),
        *block(True, int(alpha * 256)),
        *block(False, int(alpha * 256)),
        *block(True, int(alpha * 512)),

        *block(False, int(alpha * 512)),
        *block(False, int(alpha * 512)),
        *block(False, int(alpha * 512)),
        *block(False, int(alpha * 512)),
        *block(False, int(alpha * 512)),

        *block(True, int(alpha * 1024)),
        *block(False, int(alpha * 1024)),

        average_pooling((int(rho * 7), int(rho * 7))),
        flatten(),
        dropout(0.2),

        # this differs from ImageNet because of number of classes;
        # change to 1000 if want to measure params/cost on 224x224 image
        # (i.e., to compare results with original paper)
        dense(10)
    ])

    model.compile(
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        optimizer=tf.keras.optimizers.Adam(),
        metrics=['accuracy']
    )

    return model
```

Figure 11: Full MobileNet factory function (designed for $224 \times 224 \times 3$ images)

```python
# CIFAR-MobileNet-v1: designed for 32x32 images
# input_tensor necessary when profiling model
def cifar_mobilenet_v1(alpha=1, rho=1, use_mobilenet_block=True, input_tensor=None):
    block = mobilenet_block if use_mobilenet_block else regular_conv_block

    model = tf.keras.Sequential([
        *([tf.keras.Input((int(rho * 32), int(rho * 32), 3))] \
            if input_tensor is None else [tf.keras.Input(tensor=input_tensor)]),

        regular_conv(int(alpha * 32), (3, 3), padding='same'),

        *block(False, alpha * 64),
        *block(False, alpha * 128),
        *block(True, alpha * 256),
        *block(True, alpha * 512),
        *block(True, alpha * 1024),

        average_pooling((int(rho * 4), int(rho * 4))),
        flatten(),
        dropout(0.2),
        dense(1000),
        dense(10)
    ])

    model.compile(
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        optimizer=tf.keras.optimizers.Adam(),
        metrics=['accuracy']
    )

    return model
```

Figure 12: CIFAR-MobileNet-v1 factory function (designed for $32 \times 32 \times 3$ images)

```python
# CIFAR-MobileNet-v2: designed for 32x32 images
# input_tensor necessary when profiling model
def cifar_mobilenet_v2(alpha=1, rho=1, use_mobilenet_block=True, input_tensor=None):
    block = mobilenet_block if use_mobilenet_block else regular_conv_block

    model = tf.keras.Sequential([
        *([tf.keras.Input((int(rho * 32), int(rho * 32), 3))] \
            if input_tensor is None else [tf.keras.Input(tensor=input_tensor)]),

        regular_conv(int(alpha * 32), (3, 3), padding='same'),

        *block(False, alpha * 32),
        *block(False, alpha * 32),
        *block(True, alpha * 64),
        *block(True, alpha * 128),
        *block(True, alpha * 256),

        average_pooling((int(rho * 4), int(rho * 4))),
        flatten(),
        dropout(0.2),
        dense(10)
    ])

    model.compile(
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        optimizer=tf.keras.optimizers.Adam(1e-4),
        metrics=['accuracy']
    )

    return model
```

Figure 13: CIFAR-MobileNet-v2 factory function (designed for $32 \times 32 \times 3$ images). Note that the learning rate is smaller (due to training instabilities we noticed). This is essentially a thinner CIFAR-MobileNet-v1.

```
# session flops and parm count are cumulative, so we need to subtract old cumulative value
# in order to get current model's flops and parm count
cum_flops, cum_parms = 0, 0

# run model, collect summary and history
# - model_class: model factory function
# - num_epochs: number of epochs to train for
# - run_model: if true, will actually train the model; if false, will still generate
#              model metadata and profiling details without training
# - model_parms: parameters to forward to the model factory function
def run_model(model_class, num_epochs, run_model=True, model_parms={}):
    global cum_flops, cum_parms

    # counting flops and number of trainable parms/weights in model
    # see: https://stackoverflow.com/a/59862883/2397327
    session = tf.compat.v1.Session()
    graph = tf.compat.v1.get_default_graph()

    with graph.as_default():
        with session.as_default():
            # create and compile model
            model = model_class(**model_parms,
                input_tensor=tf.compat.v1.placeholder('float32',
                    shape=(1, int(model_parms['rho']*32), int(model_parms['rho']*32), 3)))

            # profiling
            run_meta = tf.compat.v1.RunMetadata()
            opts = tf.compat.v1.profiler.ProfileOptionBuilder.float_operation()
            flops = tf.compat.v1.profiler.profile(graph=graph, run_meta=run_meta, cmd='op', options=opts)

            opts = tf.compat.v1.profiler.ProfileOptionBuilder.trainable_variables_parameter()
            params = tf.compat.v1.profiler.profile(graph=graph, run_meta=run_meta, cmd='op', options=opts)

            flops, num_params = flops.total_float_ops - cum_flops, params.total_parameters - cum_parms
            cum_flops += flops
            cum_parms += num_params

    metadata = {
        'model': 'v1' if model_class == cifar_mobilenet_v1 else 'v2',
        'epochs': num_epochs,
        'params': model_parms,
        'summary': '',
        'flops': flops,
        'num_params': num_params
    }

    # save model summary to a string (not the default)
    def save_model_summary_to_string(summary_line):
        nonlocal metadata
        metadata['summary'] = metadata['summary'] + summary_line + '\n'
    model.summary(print_fn=save_model_summary_to_string)

    # run model
    if run_model:
        # don't resize feature sets
        resized_train_images = train_images
        resized_test_images = test_images
        # resize images based on rho; note this also depends on the model's
        # default image size (32 for our model, 224 for original model)
        rho = model_parms['rho'] if 'rho' in model_parms else 1
        if rho != 1:
            default_size = 224 if model_class == full_mobilenet else 32
            resized_train_images = tf.image.resize(
                resized_train_images,
                (int(rho * default_size), int(rho * default_size))
            )
            resized_test_images = tf.image.resize(
                resized_test_images,
                (int(rho * default_size), int(rho * default_size))
            )

        # train
        history = model.fit(
            resized_train_images, train_labels,
            epochs=num_epochs, validation_split=0.2,
            callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy',
                patience=10, min_delta=0.005)]).history

        # final test accuracy
        test_accuracy = model.evaluate(resized_test_images, test_labels,
            verbose=1, return_dict=True)
    else:
        history = None
        test_accuracy = None

    return {
        'metadata': metadata,
        'history': history,
        'test_accuracy': test_accuracy
    }

models = [cifar_mobilenet_v1, cifar_mobilenet_v2]
epochs = [100]
alphas = [2, 1, 0.9, 0.8, 0.75, 0.5]
rhos = [1, 30/32, 28/32, 24/32, 20/32, 16/32]

results = []
grid_count = len(models) * len(epochs) * len(alphas) * len(rhos)
for model in models:
    for epoch in epochs:
        for alpha in alphas:
            for rho in rhos:
                results.append(run_model(model, epoch, run_model=True, model_parms={
                    'alpha': alpha,
                    'rho': rho,
                    # 'use_mobilenet_block': False,
                }))
```

Figure 14: Test rig and profiling code. (Note that this code does not provide any of the analysis or plotting of the data. It only generates the data.)