1 Introduction

In short, our approach to find an efficient neural network model includes compound scaling of a simple baseline model plus a subsequent quantization process. By scaling up the model's width and depth we find a net architecture just wide and deep enough that a given task can be solved to a specified quality level. Entropy controlled ternary quantization then compresses the neural network significantly, achieving that whole channels can be pruned and the resulting kernel matrices are not only very sparse but are also characterized by a range of just two discrete values per layer.

2 CIFAR-100 task

2.1 Compound Model Scaling

We utilize an approach that scales a baseline model in multiple dimensions uniformly. Compared to [1], where the baseline model is a MnasNet[1] [3] and the experiment's dataset an ImageNet, our baseline model is a simple ResNet-44 architecture, adjusted for the CIFAR dataset as prescribed in the original paper [4], and scaled by the most reasonable coefficients found by a small grid search. Instead of scaling depth, width and resolution (cf. [1]), we only scale depth and width as for CIFAR the image resolution is the same for all samples (32 x 32 pixels).

2.1.1 Grid search

In order to find the best scaling coefficients for depth d and width w we apply a small grid search

$$\begin{aligned} \text{s.t.} \quad & d \cdot w^2 \approx 2 \\ & d, w \geq 1. \end{aligned} \tag{1}$$

This approach simplifies the search space extremely, compared to other architecture search methods like MnasNet where each architecture search takes several days on multiple TPU devices ([5], [3]). Width factor w is squared because doubling network width increases FLOPs by four times. The constraint in (1), defined in [1], will ensure that the total number of FLOPs approximately increase by 2.
We tested six tuples of (depth, width) coefficients - satisfying equation (1) - which were multiplied by the original ResNet-44 architecture, namely by the number of convolutional layers and the number of channels per layer. 1).

2.1.2 Uniform scaling of the baseline model

We found that ResNet-44-4 with depth multiplier $d = 1.4$ and width multiplier $w = 1.2$ plus ResNet-44-5 with depth multiplier $d = 1.2$ and width multiplier $w = 1.3$ perform best on CIFAR (see Table 1). Furthermore Table 1 shows that scaling only depth (ResNet-110) does not improve error a lot or even increases it (ResNet-1202).
Inspired by [1], in a next step our baseline models were scaled exponentially, according to

$$\begin{aligned} & (d \cdot w^2)^{\phi} \\ \text{s.t.} \quad & d \cdot w^2 \approx 2, \quad d, w \geq 1 \end{aligned} \tag{2}$$

[1]MNAS: mobile neural architecture search with building block elements such as inverted residual bottlenecks or depthwise convolutions (MobileNet(V2) [2] inspired)

Model	#params C10/C100	C10 error (%)	C100 error* (%)	depth, width multiplier
ResNet-44 [4]	0.66M	7.17	-	d=0, w=0
ResNet-56 [4]	0.85M	6.97	-	d=0, w=0
ResNet-110 [4]	1.7M	6.43	-	d=0, w=0
ResNet-1202 [4]	19.46M	7.93	-	d=0, w=0
ResNet-44-0	0.66M/0.67M	6.86	25.90	d=0, w=0
ResNet-44-1	1.34M/1.35M	6.60	22.99	d=1.9, w=0
ResNet-44-2	1.38M/1.39M	6.36	23.02	d=1.7, w=1.1
ResNet-44-3	1.38M/1.39M	6.30	22.99	d=1.6, w=1.1
ResNet-44-4	1.52M/1.53M	6.06	**21.98**	**d=1.4, w=1.2**
ResNet-44-5	1.37M/1.38M	**5.69**	22.73	**d=1.2, w=1.3**
ResNet-44-6	1.31M/1.32M	5.96	22.74	d=1.0, w=1.4

Table 1: Results of the grid search on depth and width multipliers. Validation error values are mean values of the best five out of 200 epochs. *updated model including all improvements found in the following chapters (data augmentation techniques, Nesterov momentum, cosine annealing, change of residual block sequence)

such that for any ϕ the total number of FLOPs will increase by approximately 2^{ϕ}. For our experiments we choose $\phi \in \{1.5, 2, 2.5, 3, 4, 5\}$ and later on found an optimal ϕ in between 3.5 and 3.6. It turns out that all but the first scaled model (ϕ=1.5) outperform the baseline models. Moreover the error does not decrease linearly by increasing ϕ. It seems there is an upper bound where the model starts overfitting on the training data as its capacity becomes too large. Hence, we applied several regularization techniques to increase the model's generalization capability.

2.1.3 Regularization techniques

- Dropout as described in Wide Residual Networks [6]: one dropout layer into each residual block between the two convolutional layers
- Data augmentation:
  - standard data augmentation (mirror + crop) used in all experiments
  - *Cutout* as described in [7]: randomly masking out square regions of image input during training
  - in [8] an automatic search for data augmentation policies is described. For CIFAR they found that *AutoAugment* picks mostly color-based transformations such as Equalize, AutoContrast, Color and Brightness. So we implement the PyTorch ColorJitter transformation in our transform sequence.

Table 2 shows the results for applying different techniques and combinations of regularization techniques. Utilizing other activation functions such as swish, hard swish, ReLU6, RReLU or leaky ReLU did not improve the validation accuracy. Hence we apply ReLU.
We also altered the basic building block structure from [Conv1>BN1>ReLU>Conv2>BN2>ReLU] to [BN1>Conv1>BN2>ReLU>Conv2>BN3] as reported in [9]. Nesterov momentum and a cosine annealed learning rate schedule (250 epochs) further improved the accuracy of the ϕ=3.0 model in the

CIFAR-100 task to 19.10% error. With the mentioned improvements we found that the following hyperparameters worked best for the CIFAR task:

- ϕ=3.5, d =1.4, w =1.2, resulting in 8.06M model parameters and a validation error of 18.54%
- ϕ=3.6, d =1.2, w =1.3, resulting in 9.03M model parameters and a validation error of 18.53%

Model	#params	baseline error* (%)	dropout(0.2) cutout(8px) ColorJitter	dropout(0.2) cutout(16px) ColorJitter	altered training procedure**	altered block architecture***
ϕ=1.5	2.15M	26.53	24.48	23.35	22.67	21.97
ϕ=2.0	3.27M	26.27	24.03	21.66	21.90	20.83
ϕ=2.5	4.17M	25.96	23.93	22.27	21.34	20.36
ϕ=3.0	6.23M	25.28	22.99	21.80	20.47	19.98
ϕ=4.0	11.72M	24.82	23.76	20.63	19.48	18.93
ϕ=5.0	24.11M	26.47	23.84	21.45	18.51	18.43

Table 2: Results of regularization methods training on CIFAR-100. Validation error values are mean values of the best five out of 200 epoch; **training procedure as specified in [6]: weight decay = 5e-4, dropping initial lr = 0.1 by 0.2 at epoch 60, 120 and 160; ***as found in [9]: a basic building block with [bn1>conv1>bn2>relu>conv2>bn3] tends to work better than the original [conv1>bn1>relu>conv2>bn2>relu]

2.2 Entropy controlled ternary quantization

Inspired by [10], we implemented an optimizer for the ternary values of a layer. In other words, we train the cluster centers (centroids) to which the model weights are assigned layerwise. For inference we use the ternary valued model but we back-propagate the resulting loss only to the non-ternary full-resolution copy of the model and to the centroid values. By doing so we learn on the one hand the ternary assignment and on the other hand the ternary values. This is pretty much equivalent to the approach in [10]. We extended the approach by adding an entropy constraint to the assignment function such that as much model weights as possible would be pushed into the cluster with the lowest information content (here zero-cluster) while maintaining accuracy. The assignment cost function C is defined as

$$C_c^{(l)} = d(w_i^{(l)}, w_c^{(l)}) - \lambda_{div} \cdot \delta^{(l)} \cdot \lambda_{max}^{(l)} \cdot log_2 P_c^{(l)} \tag{3}$$

where l denotes the layer index, i the weight index of a layer's full-resolution weights, c the centroid index according to the three possible cluster centers $\{w_n, w_0, w_p\}$, $d(\cdot)$ calculates the squared distance and $-log_2 P_c^{(l)}$ denotes the information content of a layer's centroids, where P_c describes the probability of the given weights w_i being assigned to centroid c. After a preliminary assignment, which only takes into account the squared distance as cost, i.e. $\lambda = 0$, P_c is approximated by counting the number of assigned weights per centroid. The entropy constraint, more precisely the information content constraint, results in a shift of the decision boundaries. As most weight values are very close to zero, w_0 usually has the lowest information content, thus for λ_{max} (almost) all layer weights would be assigned to w_0. λ_{max} is

calculated as follows

$$\lambda_{max_n} = \frac{d(w_{min}^{(l)}, w_0) - d(w_{min}^{(l)}, w_n^{(l)})}{log_2 P_0^{(l)} - log_2 P_n^{(l)}} \tag{4}$$

$$\lambda_{max_p} = \frac{d(w_{max}^{(l)}, w_0) - d(w_{max}^{(l)}, w_p^{(l)})}{log_2 P_0^{(l)} - log_2 P_p^{(l)}} \tag{5}$$

$$\lambda_{max} = \min\left(\lambda_{max_n}, \lambda_{max_p}\right). \tag{6}$$

Having λ_{max} we try to find the largest λ_{div} possible which is in a range of $[0, 1]$. With $\delta^{(l)}$, also in a range of $[0, 1]$, we take into account the number of weights per layer, having larger values for larger layer sizes and smaller values for smaller layer sizes.
For a given λ_{div} we execute the assignment A by solving

$$A^{(l)} = \underset{\forall c}{\operatorname{argmin}} \; C_c^{(l)}. \tag{7}$$

The assignment is executed per batch and layerwise. After 20 epochs of quantization the best assignment, in terms of validation accuracy, is loaded and frozen. For further 15 epochs we back-propagate to the ternary values only, discarding the update of the model's full-resolution copy and thus avoiding new cluster assignments. Only the centroids and unquantized layer parameters, namely the first convolutional layer, batch norm layers, shortcut layers, the final (and only) fully connected layer and its bias layer, will be trained for final fine-tuning.
The second hyperparameter of the quantizing procedure is a scaling coefficient s for the initial ternary values (centroids):

$$w_{n_{ini}}^{(l)} = |\min(w_i^{(l)})| \cdot s \tag{8}$$

$$w_{p_{ini}}^{(l)} = |\max(w_i^{(l)})| \cdot s. \tag{9}$$

With s=0.45 and λ_{div}=0.15 we quantized the above-mentioned model (with ϕ=3.5, d =1.4, w =1.2, resulting in 8.06M parameters) ternary with a resulting sparsity of 90.42%. The accuracy drop caused by ternarization amounts to 1.33%, while still accomplishing the specified quality level of the given task. Figure 1 shows an exemplary training process of a deep sparse layer.

2.2.1 Eliminating almost all multiplication operations in convolutions due to ternary matrix formats

A big advantage of ternary networks is that the number of multiplications while inference is reduced drastically. A kernel application can be thought as a dot product between the kernel weights and the patches of the input image. Dot products of two vectors of size n usually require n multiplication operations and $n-1$ addition operations. For $n = 9$, i_{hw} being the image pixels and k_{yx} being the applied kernel we get in a flattened representation:

$$\begin{matrix} i_{00} & i_{01} & i_{02} & \cdots \\ i_{10} & i_{11} & i_{12} & \cdots \\ i_{20} & i_{21} & i_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix} \bullet \begin{bmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \\ k_{20} & k_{21} & k_{22} \end{bmatrix} = i_{00} \cdot k_{00} + i_{01} \cdot k_{01} + \cdots + i_{22} \cdot k_2 \tag{10}$$

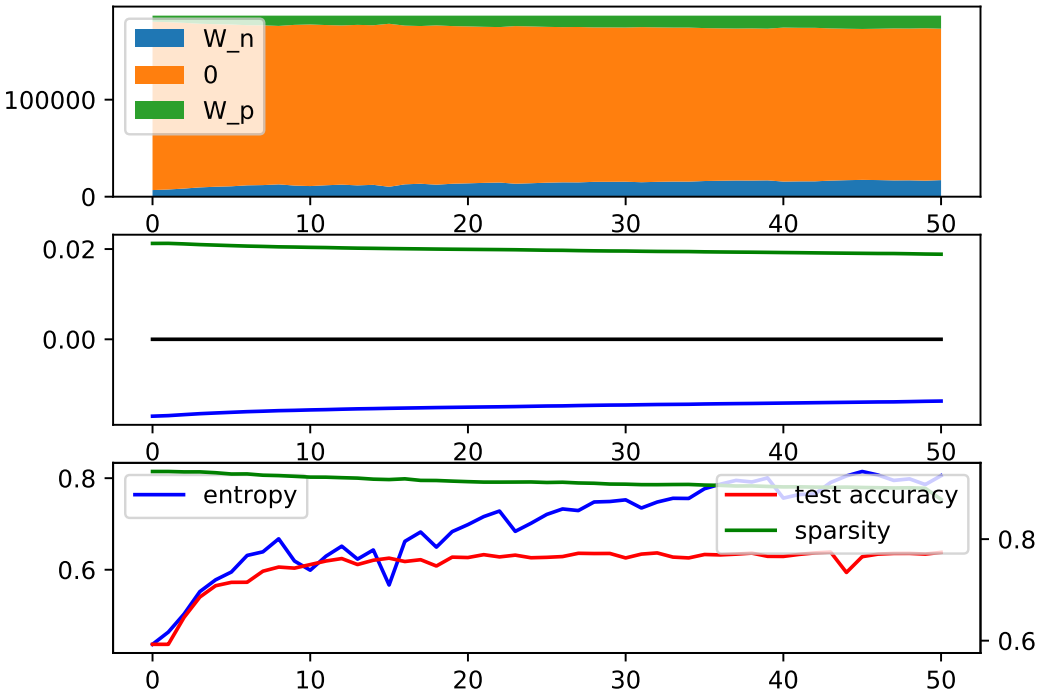

Figure 1: Quantization process. *Top*: number of weights assigned to the 3 clusters in conv layer #64; *middle*: change of cluster center values during training; *bottom*: overall training statistic.

which results in 17 floating point operations for a single application of a kernel of size 3x3.
In the ternary case two binary kernel matrices exist, one for locating the negative valued centroids w_n and one for the positive valued centroids w_p. The matrices are applied in two subsequent forward passes:

$$\begin{bmatrix} i_{00} & i_{01} & i_{02} & \cdots \\ i_{10} & i_{11} & i_{12} & \cdots \\ i_{20} & i_{21} & i_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \bullet \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}_{w_n} = (i_{01} + i_{20}) \cdot w_n \tag{11}$$

$$\begin{bmatrix} i_{00} & i_{01} & i_{02} & \cdots \\ i_{10} & i_{11} & i_{12} & \cdots \\ i_{20} & i_{21} & i_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}_{w_p} = (i_{00} + i_{12} + i_{22}) \cdot w_p \tag{12}$$

resulting in 2 multiplication FLOPs and $(n \cdot (1 - sparsity)) - 1$ addition FLOPs (3 in the above example). In summary, the above example shows that a ternary kernel can decrease FLOPs very effectively. Zooming out from the kernel level to the layer level the efficacy of ternary valued kernels in terms of saving multiplication FLOPs is even more significant. Having a convolutional layer with 32 input channels, 32 output channels and an output image size equal to the input image size equal to

32x32 (e.g. CIFAR), the number of floating point operations scales up to

$$\begin{aligned} & \text{kernel size}^2 \cdot \text{in channels} \cdot \text{out image size}^2 \cdot \text{out channels} \\ = \; & 9 \cdot 32 \cdot 1024 \cdot 32 \quad \approx \quad 9.44 \quad \text{MFLOPs for multiplication and} \qquad (13) \\ & [(\text{kernel size}^2 \cdot \text{in channels}) - 1] \cdot \text{out image size}^2 \cdot \text{out channels} \\ = \; & ((9 \cdot 32) - 1) \cdot 1024 \cdot 32 \quad \approx \quad 9.40 \quad \text{MFLOPs for addition.} \qquad (14) \end{aligned}$$

The ternary sparse matrix format allows to simplyfy the above equations to

$$\begin{aligned} & 2 \cdot \text{out image size}^2 \cdot \text{out channels} \\ = \; & 2 \cdot 1024 \cdot 32 \quad \approx \quad 65.54 \quad \text{kFLOPs for multiplication and} \qquad (15) \\ & \{[\text{kernel size}^2 \cdot \text{in channels} \cdot (1 - \text{sparsity})] - 1\} \cdot \text{out image size}^2 \cdot \text{out channels} \\ = \; & ((9 \cdot 32 \cdot 0.1) - 1) \cdot 1024 \cdot 32 \quad \approx \quad 910.95 \quad \text{kFLOPs for addition.} \qquad (16) \end{aligned}$$

The example shows that a ternary convolutional layer with 90 % sparsity is 19 times more computationally effective than the full-resolution copy of the layer. The benefit increases linearly with the number of channels and the amount of sparsity.

3 ImageNet task

For the ImageNet task many highly efficient convolutional networks already exist. EfficientNets, published in ICML 2019 [1], achieved state-of-the-art and better benchmarks on ImageNet while having a smaller storage footprint than existing networks. That is why we chose EfficientNet as a baseline to further minimize FLOPs and storage in terms of accomplishing the ImageNet task.
We focus on EfficientNet-B1 because inference FLOPs increase fast by scaling up the input image size (from 224x224 pixels in EfficientNet-B0 to 600x600 pixels in EfficientNet-B7). As EfficientNets inherently exhibit a small amount of redundancy it's more difficult to ternarize them without causing a huge drop in accuracy. Especially depthwise convolutions, which allow no inter-channel information processing, seem to be super sensitive to ternarization and pruning. Whereas depthwise convolutions save parameters and math operations in the channel space, the remaining convolutions of an EfficientNet block make parameter savings in the kernel space by utilizing convolutions with 1x1 kernels. This is meant by inherently avoiding redundancy. Finding redundant or less important weights in such a trimmed parameter space is challenging.
The EfficientNet-B1 consists of a *stem* convolution with 3x3 kernels which is the very first layer fed with the input images. After the *stem* convolution 23 mobile inverted bottleneck *MBConv* blocks follow, to which a squeeze-and-excitation (SE) optimization is added. Each (but the first two) block consists of 5 convolutional layers: *expand*, *depthwise*, *SE-reduce*, *SE-expand* and *project* convolution. They all but the *depthwise* convolution have kernel sizes of 1x1. A *head* convolution with kernel size 1x1, global average pooling and a fully connected layer form the network's back-end.
In a first step we apply our entropy controlled trained ternary approach but only to the *expand*, *project* and *head* convolutional layers. The first step results in a model which is approximately 60% ternary.
In a second step we apply entropy controlled pruning which is equivalent to the latter algorithm but updates only weights which were assigned to the zero valued cluster. The remaining weights are updated according to the updates of the full-resolution background model. This pruning technique allows us to sparsify the more sensitive layers (*SE-reduce*, *SE-expand* and the fully connected layer) while keeping the ternary layers unchanged.

In a final step we slightly prune the *depthwise* layers by simple thresholding. We empirically increased the threshold factor until validation accuary started dropping. To achieve higher accuracy we upscaled the input image size for EfficientNet-B1 from 240x240 pixels to 256x256 pixels.
The aforementioned steps resulted in a 46% sparse EfficientNet-B1 counting 1.3M scoring parameters and achieving an ImageNet accuracy of 75.03%. For the first step we used scaling coefficients s=0.2 (eq. 9) and λ_{div}=0.125, for the second step we increased them to s=0.25 and λ_{div}=0.15. The pruning threshold for depthwise layers in step three was set to $0.01 \cdot w_{max}^{(l)}$.

4 MicroNet Challenge Scoring

4.1 Parameter Storage

- Sparse matrices are assumed to be stored as a set of nonzero values and a bitmask of the full tensor shape that denotes the location of the nonzero values.
  - In the ternary case kernel matrices are stored as two half precision centroid values and two binary bit masks of the full tensor shape.
- For block sparsity, a single bit can be used to denote a block of values (e.g. a 512x128 matrix with 4x4 block sparsity would require a bitmask with 4096 bits).
  - We asked if we could apply a hierarchical bitmask system: one outer bitmask selecting non-zero kernels and several inner bitmasks to locate non-zero kernel elements. In order to keep scoring more comparable this approach was denied.
  - However, we're allowed to ignore weights that belong to dead output channels and the according weights of the next layer's appropriate input channel weights
- A 32-bit parameter counts as one parameter. Quantized parameters of less than 32-bits will be counted as a fraction of one parameter.

4.2 Math Operations

- The mean number of arithmetic operations per example required to perform inference on test set.
- A 32-bit operation counts as one operation.
- An operation on data of less than 32-bits will be counted as a fraction of one operation, where the numerator is the max number of bits in the inputs of the operation and the denominator is 32.
  - A multiplication operation with one 3-bit and one 5-bit input, with a 7-bit output, will count as 5/32nd of an operation.
  - A multiplication operation where one input is 32-bits, the other input is 8-bits and the output is 8-bits will count as one whole op, as the first input is 32-bits.
- We allow multiplication of a value with a standalone sign-bit by a binary value in this format to be counted as a 1/32nd of an operation regardless of the resolution of the other input.
- If an FP32 addition is followed by a FP16 multiplication the conversion of the output of the addition to FP16 to perform the multiplication does not need to be taken into account.

- We count transcendental functions (cannot be expressed in terms of a finite sequence of the algebraic operations of addition, multiplication, and root extraction, e.g. $\sin(x)$, c^x) as a single operation.
- We do not take dynamic activation sparsity into account because we do not take activation storage size into account.
- We allow static activation sparsity (e.g., sparse attention) because the sparsity pattern is known ahead of time and we can avoid expensive dynamic compression of layer outputs and potentially mitigate load-balancing issues.
- We allow "freebie" quantization of math operations to 16-bits to strike a balance between rewarding quantization approaches and minimizing the complexity of entering the competition. However, it is typically necessary to perform full 32-bit accumulation to maintain model quality. Thus, we do not allow "freebie" quantization for addition operations.

References

1. Tan, M. & Le, Q. V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *arXiv:1905.11946 [cs, stat]*. arXiv: 1905.11946. http://arxiv.org/abs/1905.11946 (2019) (May 28, 2019).
2. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. & Chen, L.-C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv:1801.04381 [cs]*. arXiv: 1801.04381. http://arxiv.org/abs/1801.04381 (2019) (Jan. 12, 2018).
3. Tan, M. *et al.* MnasNet: Platform-Aware Neural Architecture Search for Mobile. *arXiv:1807.11626 [cs]*. arXiv: 1807.11626. http://arxiv.org/abs/1807.11626 (2019) (July 30, 2018).
4. He, K., Zhang, X., Ren, S. & Sun, J. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*. arXiv: 1512.03385. http://arxiv.org/abs/1512.03385 (2019) (Dec. 10, 2015).
5. Wistuba, M., Rawat, A. & Pedapati, T. A Survey on Neural Architecture Search. *arXiv:1905.01392 [cs, stat]*. arXiv: 1905.01392. http://arxiv.org/abs/1905.01392 (2019) (May 3, 2019).
6. Zagoruyko, S. & Komodakis, N. Wide Residual Networks. *arXiv:1605.07146 [cs]*. arXiv: 1605.07146. http://arxiv.org/abs/1605.07146 (2019) (May 23, 2016).
7. DeVries, T. & Taylor, G. W. Improved Regularization of Convolutional Neural Networks with Cutout. *arXiv:1708.04552 [cs]*. arXiv: 1708.04552. http://arxiv.org/abs/1708.04552 (2019) (Aug. 15, 2017).
8. Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V. & Le, Q. V. AutoAugment: Learning Augmentation Policies from Data. *arXiv:1805.09501 [cs, stat]*. arXiv: 1805.09501. http://arxiv.org/abs/1805.09501 (2019) (May 24, 2018).
9. Han, D., Kim, J. & Kim, J. Deep Pyramidal Residual Networks. *arXiv:1610.02915 [cs]*. arXiv: 1610.02915. http://arxiv.org/abs/1610.02915 (2019) (Oct. 10, 2016).
10. Zhu, C., Han, S., Mao, H. & Dally, W. J. Trained Ternary Quantization. *arXiv:1612.01064 [cs]*. arXiv: 1612.01064. http://arxiv.org/abs/1612.01064 (2019) (Dec. 4, 2016).
11. Melas-Kyriazi, L. 2019 (accessed September 30, 2019). https://github.com/lukemelas/EfficientNet-PyTorch.
12. Antoshchenko, D. 2017 (accessed August 10, 2019). https://github.com/TropComplique/trained-ternary-quantization.
13. Thangarasa, V. 2017 (accessed June 20, 2019). https://github.com/uoguelph-mlrg/Cutout.
14. Idelbayev, Y. 2017 (accessed May 22, 2019). https://github.com/akamaster/pytorch_resnet_cifar10.
15. GoogleResearch. 2019 (accessed October 10, 2019). https://github.com/google-research/google-research/tree/master/micronet_challenge.