SSC22-P5-27

Evaluation of Parameter-Scaling for Efficient Deep Learning on Small Satellites

Calvin B. Gealy, Alan D. George
NSF Center for Space, High-performance, and Resilient Computing (SHREC) - University of Pittsburgh
4420 Bayard Street, Suite 560, Pittsburgh, PA; 412-624-9664
c.gealy@pitt.edu

ABSTRACT

Parameter-scaling techniques change the number of parameters in a machine-learning model in an effort to make the network more amenable to different device types or accuracy requirements. This research compares the performance of two such techniques. NeuralScale is a neural architecture search method which claims to generate deep neural networks for devices that are resource-constrained. It shrinks a network to a target number of parameters by adjusting the width of layers independently to achieve a higher accuracy than previous methods. The novel NeuralScale algorithm is compared to the baseline uniform scaling of MobileNet-style models, where the width of each layer in the model is scaled uniformly across the network. Measurements of the latency and runtime memory required for inference were gathered on the NVIDIA Jetson TX2 and Jetson AGX Xavier embedded GPUs using NVIDIA TensorRT. Measurements were also gathered on the Raspberry Pi 4 embedded CPU featuring ARM Cortex-A72 cores using ONNX Runtime. VGG-11, MobileNetV2, Pre-Activation ResNet-18, and ResNet-50 were all scaled to 0.25×, 0.50×, 0.75×, and 1.00× the original number of parameters. On embedded GPUs, this research finds that NeuralScale models do offer higher accuracy, but they run slower and consume much more runtime memory during inference than their equivalent uniform-scaling models. On average, NeuralScale is 40% as efficient as uniform scaling in terms of accuracy per megabyte of runtime memory, and NeuralScale uses 2.7× the runtime memory per parameter as uniform scaling. On the embedded CPU, NeuralScale is slightly more efficient than uniform scaling in terms of accuracy per megabyte of memory, using essentially the same amount of memory per parameter. However, there is on average an over 2.5× increase in the latency for inference. Importantly, parameter count does not guarantee performance in terms of runtime-memory usage between the scaling methods on embedded GPUs, while latency grows significantly on embedded CPUs.

INTRODUCTION

With machine learning and computer vision becoming more ubiquitous, the desire to run computer-vision models on resource-constrained systems such as small satellites has grown. These systems lack the significant computing resources that can be found in larger-scale systems, such as high-performance CPUs and GPUs. Therefore, extremely large and deep networks that are present in current research must be scaled for efficient and effective inference on less powerful systems. Two such methods are NeuralScale[1] and the uniform scaling introduced in MobileNets.[2] This research analyzes the performance of networks created using NeuralScale and uniform scaling on embedded GPUs and an embedded CPU.

The NeuralScale method aims to optimize the accuracy of a network for a target number of parameters. The uniform-scaling method is simpler. It scales the width of each layer uniformly across the network, as is done in MobileNets.[2] CIFAR-100 is used as the target dataset in these comparisons.[3] To compare these two techniques, measurements of general model statistics like floating-point operations (FLOPs) and the Critical Datapath Length (CDL)[4] were taken. This research presents:

1. Results when performing NeuralScale on ResNet-50, a larger network than any previously tested
2. An analysis of the FLOPs and CDL model metrics on both NeuralScale and uniform-scale models
3. Latency and runtime memory measurements and analysis on two embedded GPUs and one embedded CPU

In Section 2, the broad trends in the research are discussed. Next, details about NeuralScale are presented in Section 3. Section 4 explains specifics of

the methods used. The results are presented in Section 5. Finally, a discussion of this research, conclusions, and future research directions are presented in Section 6, Section 7, and Section 8, respectively.

RELATED RESEARCH

This section is split into five subsections. First, an overview of several neural architecture search methods is given. Next is a brief introduction to MobileNets. Then, Critical Datapath Length, a newer metric used to quantify model computational complexity, is reviewed. In the final two sections, an overview of NVIDIA TensorRT and ONNX Runtime is given.

Neural Architecture Search

Designing neural networks that perform well on resource-constrained devices has become a research topic of interest in the machine-learning and computer-vision fields. Recently, there has been a significant amount of interest in neural architecture search (NAS), where a neural-network architecture is evolved algorithmically to adapt to a specific problem. Within NAS, there is research interest in creating networks optimized for specific devices. Li et al. introduced HW-NAS-Bench.[5] HW-NAS-Bench is a dataset assembled by gathering performance measurements on real devices. Researchers can then use the information in this dataset to guide their NAS algorithms. By using this dataset, Li et al. found that they could design efficient networks for their target platforms. Other methods like EfficientNet,[6] published by Tan and Le, begin with a baseline model determined algorithmically that can then be scaled up or down to improve accuracy or performance, respectively. The EfficientNet-B7 model, a scaled-up version of EfficientNet-B0, achieved state-of-the-art performance on ImageNet. The authors also show that scaling the network's depth, resolution, and width uniformly with a compounding coefficient allows the CNN to find more important regions in an image than if the components had been scaled individually.

Other forms of NAS focus on pruning larger networks. Cai et al. perform a proxyless NAS by generating an over-parameterized network and then conducting path-level pruning to create a network that is efficient for a given platform.[7] This method is done while avoiding simpler tasks that do not relate to given network, dataset, or target platform. Cai et al. demonstrate NAS by choosing a sub-network from a supernet for deployment on a given device.[8] They do this operation by training the supernet and then fine-tuning subsets of the supernet. With this method, they were able to achieve a new state of the art on ImageNet. Another pruning method called NeuralScale was proposed by Lee and Lee.[1] This method is the focus of this research. With NeuralScale, only the width component of the network is pruned and scaled. NeuralScale attempts to select the best number of filters for each individual layer at a given scaling factor rather than scaling uniformly across layers. More on the exact method will be explained in Section 3, but the authors showed accuracy improvements over other scaling methods.[1]

MobileNets and Uniform Scaling

MobileNets are a structure of networks aimed at making an efficient architecture for mobile devices.[2] MobileNetV1 and MobileNetV2 use a depthwise-separable convolution to reduce the computational complexity of the model.[2,9] Hyperparameters of the network include the width multiplier and image resolution. The effect of the width multiplier on the network is the same as the effect of the scaling ratio with uniform scaling in this research.

Critical Datapath Length

Critical Datapath Length (CDL) was proposed by Langerman et al.[4] As noted in their research, the standard metric of floating-point operations (FLOPs), often used to estimate the performance of a network, does not necessarily translate well to massively parallel devices like GPUs. CDL has been shown to be a much more useful metric for performance on parallel devices than FLOPs. CDL is included in this research as it is another metric that has been introduced to try to better understand and predict model performance. A larger CDL indicates that there is a longer serial path that the data must flow through and, therefore, likely longer execution time on a highly parallel system.

NVIDIA TensorRT

NVIDIA TensorRT is a tool used for high-performance inference on NVIDIA GPUs.[10] It allows for several performance optimizations such as mixed precision, fusing layers, automatically selecting the best kernel for a platform, and others. It can input an Open Neural Network Exchange (ONNX) model and create the appropriate TensorRT engine automatically.[11,12]

ONNX Runtime

ONNX Runtime is designed for efficient inference with ONNX models. It is an open-source project backed by Microsoft. It includes a Python API which allows for optimized execution of ONNX models on different accelerator types as well as standard CPU inference. For more information, the reader is directed to the ONNX Runtime documentation.[13]

BACKGROUND

This background section is broken into several subsections. First, the NeuralScale method is described. Then, embedded-device constraints are quantified. Finally, statistics about the test platforms are given.

NeuralScale

To target a specific number of parameters, NeuralScale first performs an iterative pruning method to determine the importance of each filter in the network.[1] The importance is defined as a measurement of the increase in error caused by the removal of that filter. When removing a filter causes a large decrease in the accuracy, the filter is considered to be more important. This metric of importance is explained in the research of Molchanov et al.[14]

As the pruning method is performed across many iterations, the change in the number of filters per layer compared to the number of parameters in the network is learned. With this knowledge, a model of the number of filters in a layer given the total number of parameters can be generated using power functions. Now, an approximation of the number of filters per layer can be set for a target number of parameters, which is further refined using stochastic gradient descent. An in-depth, mathematical explanation can be found in the NeuralScale paper.[1] The steps of iteratively pruning the model, searching for the parameters for the power function, and then generating the network with stochastic gradient descent represent one iteration of what the authors define as architecture descent. Architecture descent can then be run iteratively for a set number of steps or until convergence. The authors of NeuralScale find that multiple iterations of architecture descent generate a more accurate model than one single iteration.

Embedded-Device Constraints

With small satellites, the performance of computer vision applications is often limited by the Size, Weight, and Power - Cost (SWaP-C) constraints. These constraints can lead to heavy limitations on target apps in terms of energy consumption or memory usage. Therefore, it is critical to know the processing limitations when selecting a machine-learning model.

An example of a critical application for small satellites is space debris collision avoidance. If a collision avoidance system is based on a computer-vision model, the speed at which that model operates dictates how quickly the system can respond. Analyzing collision avoidance can be simplified by considering a self-driving car rather than a satellite orbiting the Earth. Consider a car moving at 80 mph (117 feet per second). If an obstruction suddenly enters the roadway 20 feet ahead of the vehicle, and it takes 100 milliseconds to process the first frame where the obstruction is in the field of view, the car will have already traversed half of the distance to the object before it even recognizes that there is an obstruction. Therefore, it can be important that low latency is achievable on embedded devices. Furthermore, the high-resolution vision necessary for detecting small obstructions will be even slower due to the added computational cost from the increase in the number of pixels and limited memory bandwidth on the embedded devices. This problem only becomes more challenging when considering a satellite moving in more dimensions and at higher speeds.

The runtime memory of a model is an additional constraint in embedded and space computing platforms. NVIDIA's Jetson embedded GPUs have memories in the range from 2 GB[15] to 32 GB.[16] However, on these systems, the memory is shared between the CPU cores and the GPU. Therefore, in a safety-critical situation like a self-driving car, the amount of memory required by all applications on the device must be optimized. Similarly, with a general-purpose CPU architecture like on the Raspberry Pi 4, memory is once again limited and shared by all the processes running on the system.

Test Platforms

The specifications for the two embedded GPU platforms tested in this research are listed in Table 1. For reference to a desktop-class GPU, the specifications of a GTX 1080 Ti are also listed. The GTX 1080 Ti was used in the original NeuralScale research.[1] Note the significant limits to the core count, thermal profile, and memory bandwidth on the embedded devices. Also note that while the Jetson AGX does have more memory than the GTX 1080 Ti, this memory must be shared across the

Table 1: Specifications of the embedded GPUs[17,18] and GTX 1080 Ti.[17,19]

Metric	Jetson TX2	Jetson AGX	GTX 1080 Ti
Architecture	Pascal	Volta	Pascal
Perf. (TFLOPs)	1.3	1.4	11.3
CUDA Cores	256	512	3584
TDP (W)	15	30	250
Memory (GB)	8	16	11
Mem. BW (GB/s)	59.7	136.5	484

GPU and CPU cores.

The Raspberry Pi 4 is a popular single-board computer which features a quad-core ARM Cortex-A72 processor.[20] It is used as an embedded CPU baseline as it is simple to use and has a lot of development support among the open-source and hobbyist communities as well as uses for rapid prototyping. The NXP i.MX 8QuadMax uses the A72 and is being considered for near-term space computing applications, thus performance results from this architecture are considered valuable in a small satellite context.[21] The board used in this research features 4 GB of RAM. A 64-bit version of the Ubuntu GNU/Linux operating system was used.

APPROACH

The PyTorch code distributed with the NeuralScale paper[1] was used for this research. The results of the architecture-descent operation are included for the models from the original paper. Therefore, testing was done with VGG-11,[22] PreAct ResNet-18,[23,24] and MobileNetV2.[9] PreAct ResNet-18 model[24] was used in the original NeuralScale paper.[1] For simplicity in labeling, this network will be referred to as PResNet-18.

To expand upon the originally published models of NeuralScale, a scaled version of ResNet-50 was also tested for this research. The models in the original NeuralScale paper are small compared to many modern models, potentially making it harder to optimize. Therefore, a larger model like ResNet-50 was chosen to provide insight to how the NeuralScale method performs on larger networks. The ResNet-50 model was scaled using PyTorch version 1.8.1. Due to the time-consuming step of pruning, a batch size of 64 was chosen, which allowed the parameters to be pruned in fewer epochs. CIFAR-100[3] was used as the dataset for comparing the models in this research. This dataset features 32×32-pixel RGB images. Measurements were taken at parameter scalings of $0.25\times$, $0.50\times$, $0.75\times$, and $1.00\times$. These values were chosen as they allow for interpolation between the points without generating so much data that analysis would be prohibitive. They also match the original scaling values from the original NerualScale paper.[1]

Uniform scaling, a different name for the operation of the width multiplier found in MobileNets, is the other parameter scaling method investigated in this research.[2] In uniform scaling, each layer is scaled by the same scaling factor. For example, if a layer normally has a width of 64, and the goal is to generate a network with 50% of the original number of parameters, then that layer will have 32 filters after being scaled.

The number of FLOPs and the CDL of the network were measured using the models from both uniform scaling and the models generated after 15 iterations of the NeuralScale architecture-descent method. Then, for each model, five fine-tuned versions were created to determine an average test accuracy of the networks. FLOPs were found by measuring the multiply accumulates (MACs) using a PyTorch tool[25] and multiplying the MACs by two. While this method is not a perfect way to calculate FLOPs, it does provide an estimate.

After collecting these general model statistics, device-specific measurements were taken on a Jetson TX2 with a GPU maximum frequency of 1122 MHz and a Jetson AGX with a GPU maximum frequency of 1377 MHz. In the NeuralScale paper,[1] the authors note that NeuralScale is a tool designed for making models more efficient on "resource-constrained" systems. Therefore, this research tests this claim on the Jetson embedded GPUs and the Arm Cortex-A72 embedded CPU cores. To measure the performance on these systems, all of the networks were exported to ONNX from PyTorch with the ONNX opset version 9. In Python, each ONNX model was loaded into a temporary TensorRT engine. Next, an image from CIFAR-100 was input to both the original PyTorch model and the TensorRT engine. The outputs from the final layers of both models were then compared using the cosine distance. The cosine distance allows for the difference between the two high-dimensional tensors to be compared. The maximum cosine distance from 100 samples of CIFAR-100 was confirmed to be less than 10^{-4}. By checking that the distance is small, it is confirmed that the TensorRT engine is functionally the same as the original PyTorch network while allowing for small implementation discrepancies.

With the ONNX models saved, they were then

individually loaded into NVIDIA's *trtexec* program. This program is capable of loading an ONNX model, creating a TensorRT engine, and running sample inferences. Average latency measurements for inference were gathered for every model using this program. Inferences were performed with a batch size of one to represent the case where an embedded system would be processing one frame at a time from a camera.

Then, runtime-memory measurements were taken as memory usage on embedded devices can be critical. In the TensorRT documentation,[10] NVIDIA estimates the runtime memory required by the model to be the sum of the persistent memory, the size of the serial engine, and the memory needed for the activations. The host and device persistent memory usage can be found in the verbose outputs of the *trtexec* program. The size of the serial engine can be easily found by saving the engine and querying the operating system for the file size. Finally, a custom version of *trtexec* was created to print the activation memory required by the network as explained by the NVIDIA TensorRT documentation.[10] An average was taken across 50 different generations of the TensorRT engine for each model. This averaging was done because there are slight variations in the TensorRT engines since the tool may optimize the network differently depending on the device's resource utilization at the time of inference. An adapted version of the Docker container from an NVIDIA Jetson developer[26] with PyTorch version 1.9.0 and TensorRT version 7.1.3 was used on the embedded GPU platforms.

When measuring performance with ONNX Runtime on the Raspberry Pi 4, the generated ONNX models were directly loaded using the Python API for the runtime. ONNX Runtime version 1.9.0 was used for testing. The execution provider was set to be the CPU with ONNX Runtime being allowed to choose the optimal number of threads to use. The Raspberry Pi 4 default maximum clock rate of 1.5 GHz was used for testing. To find the latency of the model inference, 20 rounds of inference were run to prime the caches. Then, the latency was averaged over 80 single-image batches. This whole process was further averaged over another 50 executions of this sequence. To measure the runtime memory of the model, the Python memory profiler[27] was used. This profiler queries the OS to determine the amount of memory used by the code. It also can report the specific amount of memory used by a specific line of Python code. This was used to record the amount of memory used for the ONNX Runtime InferenceSession generation. This value was averaged over 50 separate InferenceSession generations to account for variances caused by the operating system.

RESULTS

The results section is split into three subsections. In the first subsection, purely model-based statistics are presented. In the second subsection, results based on running the models on the Jetson TX2 and Jetson AGX are shown. In the final subsection, results from running the models on the Raspberry Pi 4 are noted. Due to space constraints, the models VGG-11, MobileNetV2, PResNet-18, and ResNet-50 are abbreviated as VGG11, MNV2, PRN18, and RN50, respectively. Accuracy measurements for VGG-11, MobileNetV2, and PResNet-18 are from the original NeuralScale paper.[1] Note that comparisons should only be drawn between scaling methods and not between different model architectures. Due to constraints on the training of ResNet-50, a smaller batch size was used, which is suspected to cause the resulting lower accuracies. However, a fair comparison can still be drawn between NeuralScale and uniform scaling as the training hyperparameters were held constant between the two.

Model Analysis Results

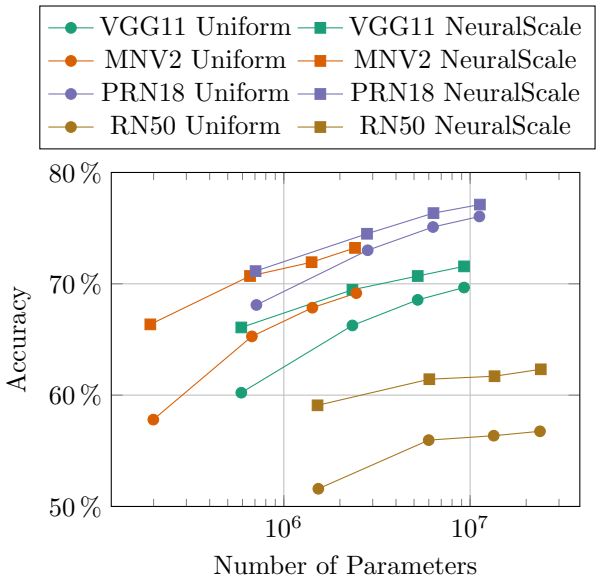

Figure 1: Accuracy versus number of parameters for all scaled models. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

NeuralScale is designed to increase the accuracy of a model while using the same number of parameters. The accuracy results from the NeuralScale paper[1] are presented along with the new results for ResNet-50 in Fig. 1. Note that the y-axis on the figure shows the accuracy range of 50% to 80% in order to make the differences more discernible. The series of four points for each line represents the four scaling values of 0.25×, 0.50×, 0.75×, and 1.00× in order. The NeuralScale models are able to achieve a better accuracy for the same scaling in all tests of this study. Each scaling has approximately the same number of parameters. Therefore, NeuralScale is achieving its goal of improving accuracy over the more naive uniform-scaling method.

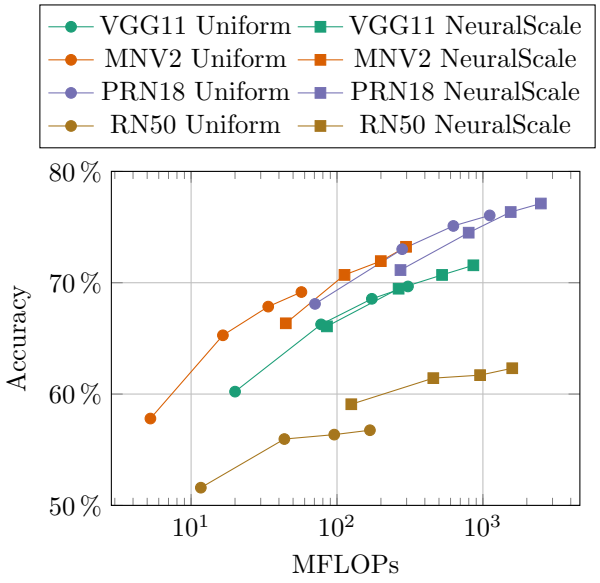

Figure 2: Accuracy versus million FLOPs for the models. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

Another insightful perspective on the scaling methods is to view the accuracy versus million FLOPs (MFLOPs) shown in Fig. 2. Here, NeuralScale's dominance in terms of accuracy is less clear. When the points are no longer constrained by the number of parameters, but rather the number of FLOPs, it can be seen that the gains in accuracy achieved by NeuralScale over uniform scaling are not necessarily surprising. For the same scaling ratios, the NeuralScale points use significantly more MFLOPs, which allows for the gains in accuracy.

The other model-based metric studied is CDL. As NeuralScale only adjusts the widths of the networks and not the number of layers, the CDL of every network within the same model class is the same. The CDL values are displayed in Table 2. As noted in the paper proposing CDL,[4] CDL is a more appropriate metric for memory-bound devices, while FLOPs are more appropriate on compute-bound devices. An analysis of these results will follow in Section 6.

Table 2: CDL of the models tested compared with the range of MFLOPs for the scaling methods.

Model	CDL	Uniform Scaling MFLOPs range	NeuralScale MFLOPs range
VGG-11	39	20.01-306.54	85.33-860.13
MobileNetV2	162	5.26-57.03	44.57-297.25
PResNet-18	61	70.68-1113.18	272.38-2495.02
ResNet-50	167	11.66-168.15	125.31-1589.02

GPU-Specific Results

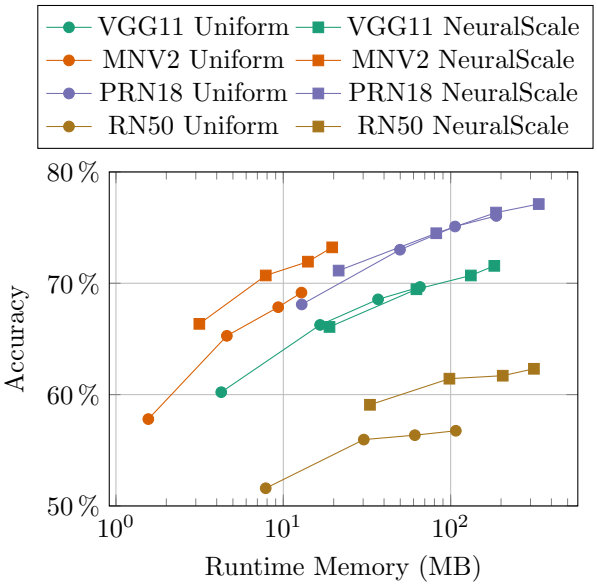

Figure 3: Accuracy versus MB of runtime memory for Jetson TX2. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

One of the important additions of this research is the analysis of uniform scaling and, specifically, NeuralScale on resource-constrained devices. As such, it is important to consider the amount of runtime memory required to perform inference on the device. A valuable perspective is to view the accuracy of the models versus the amount of runtime memory required. While networks of the same model and scaling ratio may use a similar number of parameters, the arrangement of the parameters has a cost in

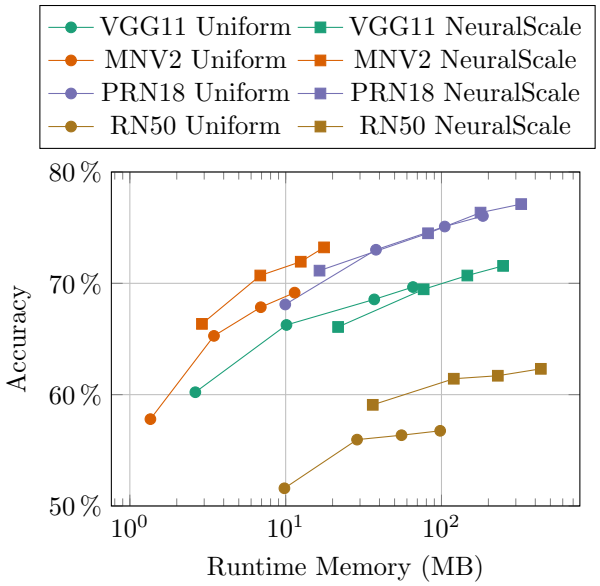

Figure 4: Accuracy versus MB of runtime memory for Jetson AGX. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

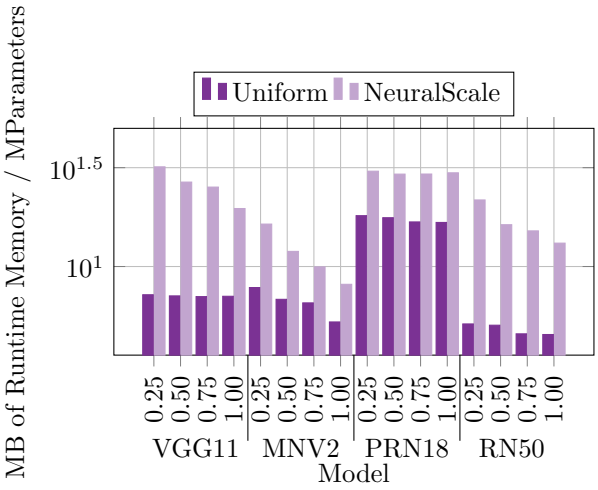

Figure 5: MB of runtime memory per million parameters for Jetson TX2. Log scale on y-axis.

terms of runtime-memory usage and computational complexity. Fig. 3 presents the results collected on the Jetson TX2, and Fig. 4 depicts the results collected on the Jetson AGX.

The results are similar on both the Jetson TX2 and the Jetson AGX. For the same scaling ratio, NeuralScale does increase the accuracy but also causes a very large increase in the runtime memory (note the log scale of the x-axes). When the results are plotted as a line, it is evident that NeuralScale produces models than often do have a higher accuracy for the same amount of memory as is shown by the MobileNetV2 and ResNet-50 models. However, for the VGG-11 and PResNet-18 models, the NeuralScale results show that for essentially the same amount of runtime memory, the same result is achieved, though they still do use fewer parameters.

Continuing, viewing the amount of runtime memory required per parameter helps for better analysis of how different the runtime-memory usage is between NeuralScale and uniform scaling. Fig. 5 shows the results from the Jetson TX2, and Fig. 6 gives the results on the Jetson AGX. The x-axes of the bar plots contain the scaling values of 0.25×, 0.50×, 0.75×, and 1.00× for each of the different models. These plots show how vastly different the runtime-memory usage of the models is when scaled to the same number of parameters. NeuralScale consistently uses more runtime memory per parameter than uniform scaling.

Next, the drop in latency for inference, or relative speed, is shown in Fig. 7 and Fig. 8 for the Jetson TX2 and Jetson AGX, respectively. The relative speed is calculated by dividing the uniform-scaling latency by the NeuralScale latency. A value closer to 100% represents a NerualScale model who's latency is closer to the uniform-scaling model. There is always a decrease in relative speed when using NeuralScale versus the equivalent uniform-scaling network. However, all relative speeds are around 60% or larger.

Viewing the results as accuracy versus latency shows that for all but the ResNet-50 models, the NeuralScale and uniform-scaling lines essentially overlap. Therefore, with the same base model, they achieve close to the same accuracy for a set latency. Fig. 9 and Fig. 10 display this metric on the Jetson TX2 and Jetson AGX, respectively.

CPU-Specific Results

Again, the first perspective on the CPU results from the ARM Cortex-A72 is to view the accuracy versus megabyte of runtime memory as seen in Fig. 11. Here, there is essentially no memory increase for the gain in accuracy. NeuralScale models consistently perform better with the same memory utilization as their uniform-scaling counterparts.

On the ARM Cortex-A72, there is not a large change in the memory usage per parameter between

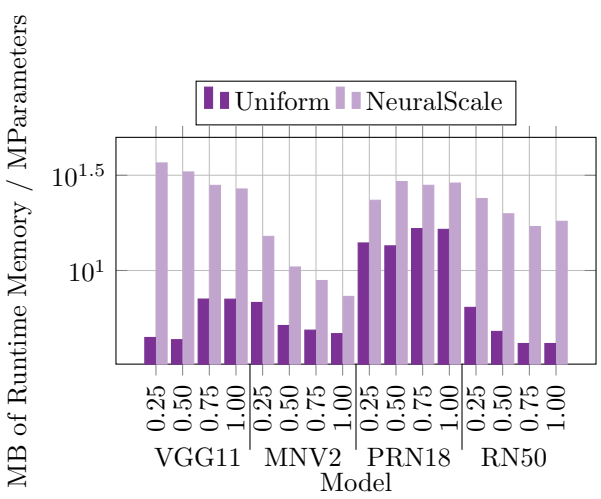

Figure 6: MB of runtime memory per million parameters for Jetson AGX. Log scale on y-axis.

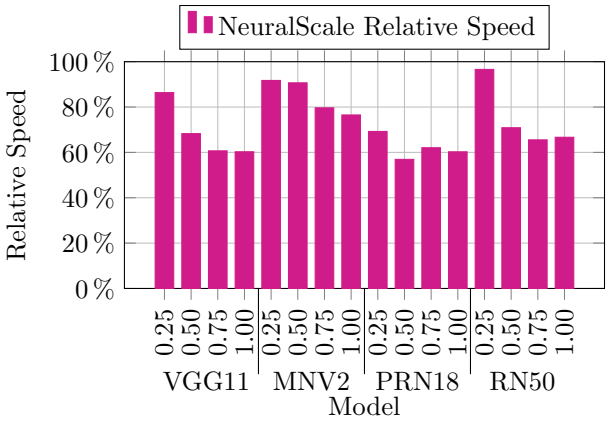

Figure 7: NeuralScale relative speed (uniform latency / NeuralScale latency) for Jetson TX2. Higher is better.

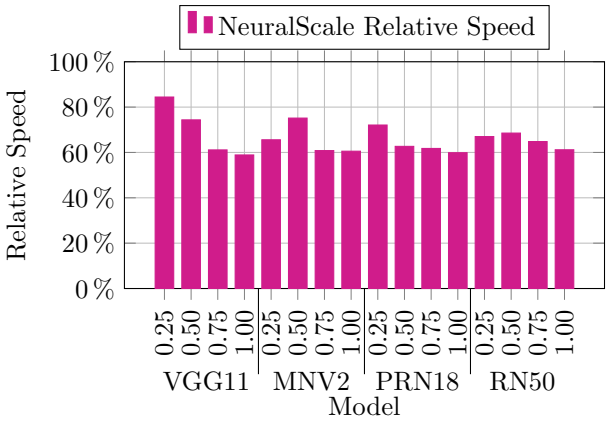

Figure 8: NeuralScale relative speed (uniform latency / NeuralScale latency) for Jetson AGX. Higher is better.

uniform scaling and NeuralScale. Fig. 12 shows how many megabytes of runtime memory per million parameters are used. This trend differs from that seen on the GPUs where there was a larger difference in memory efficiency between the scaling methods.

The relative speed of the NeuralScale models compared to the uniform scaling models on the ARM Cortex-A72 is shown in Fig. 13. Here, all values are below 60%. Again, this differs from the GPU where all values were around or above 60%. This demonstrates a large latency penalty for using NeuralScale models instead of uniform-scaling models on general-purpose CPU architectures.

The final view of the data for the ARM Cortex-A72 is in terms of the accuracy versus millisecond of latency. Here, for all but ResNet-50, the NeuralScale and uniform-scaling lines essentially lie on top of each other for the same base model. This indicates that with a set latency requirement, the accuracy will be relatively consistent when using the same model type. This trend is different for ResNet-50 where NeuralScale consistently provides better accuracy for the same latency.

DISCUSSION

This section is broken into several subsections. First, a presentation of the effect of NeuralScale on the network layout is given. Next, an analysis of latency compared to the predictive metrics of FLOPs and CDL is performed. Finally, an analysis of why parameters are a poor metric for scaling a model for a resource-constrained device is presented.

Model Layout in NeuralScale

One of the interesting aspects of the NeuralScale method is how it adjusts the widths of the layers. The widths of the layers in the networks are represented in Fig. 15. In this figure, the y-axis represents the width of a layer, while the x-axis represents the layer number in the network. The layer widths are shown for a scaling ratio of 1.00× as it shows model layout with the same number of parameters as chosen by the original model designers. As the parameter scaling is 1.00×, the uniform-scaling method simply returns the original network layout.

Fig. 15 is insightful because it shows the structure for a more efficient architecture, in terms of accuracy per parameters, as determined by NeuralScale. NeuralScale tends to shift the parameters from the end of the network to the beginning or middle. In

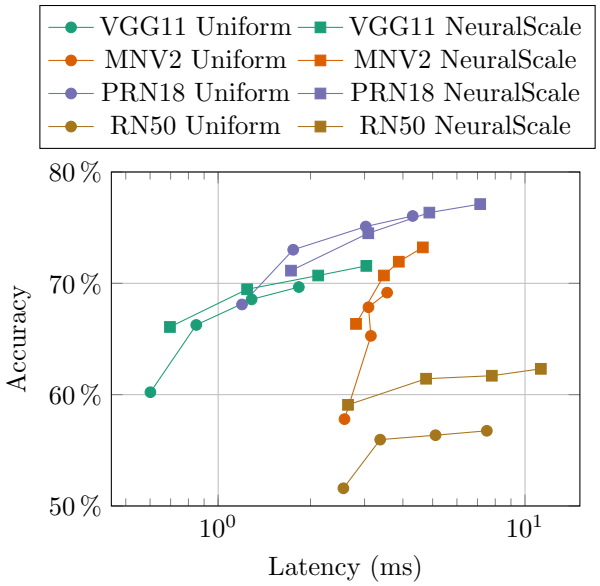

Figure 9: Accuracy versus latency (ms) for Jetson TX2. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

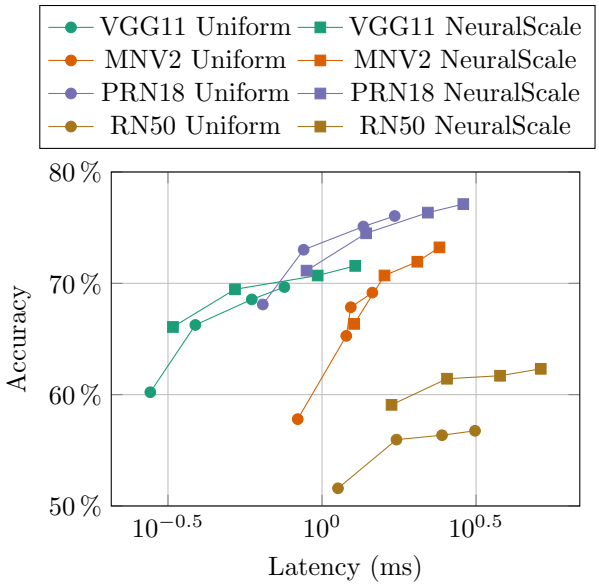

Figure 10: Accuracy versus latency (ms) for Jetson AGX. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

CNNs, earlier layers capture more general features like textures while later layers capture more specific details.[28] Fig. 15 displays how NeuralScale infers that general features are more useful on CIFAR-100. This result is interesting because most of the models as designed by the original sources tend to place more parameters near the end of the network.

For the NeuralScale models, there is an observed large increase in FLOPs for the same parameter count as uniform scaling. This result can be explained by the relationship between FLOPs and the model layout. As shown by Fig. 15, NeuralScale shifts parameters earlier in the network. Earlier in the network, there is also a larger spatial dimension. The relationship of FLOPs is on the order of $O(n^2)$ to the spatial dimension due to the height and width of the image factoring into the FLOPs calculation. Therefore, there is a large increase in the number of floating-point operations to be performed when more of the parameters are used for these earlier filters.

Runtime and Predictive Metrics

FLOPs and CDL are compared against the latency measurements on the devices. The CDL is the same for all networks with a common base model. Therefore, on a highly parallel device like a GPU, the runtime of the models should be about the same no matter how the width of the networks is scaled. The latencies between networks with the same base model and same scaling method but different scaling ratios are vastly different even though the CDL is the same. From this research and as is noted by the original CDL paper,[4] it is evident that CDL alone is not a perfect prediction metric, especially when focusing on these parameter scaling. For example, VGG-11 at 0.25× scaling with uniform scaling has 15× fewer MFLOPs than VGG-11 at 1.00× scaling, but it is only 3× faster on the Jetson TX2, while still having the same CDL. It is a balance between these two metrics that can be used to understand and predict the performance.

FLOPs, on the other hand, predicts that NeuralScale networks will perform significantly worse than uniform-scaling networks. However, this prediction is not accurate. The FLOP count on ResNet-50 at ratio 0.25× was over 10.74× higher for NeuralScale than for uniform scaling. However, the increase in latency on the TX2 was only around 4% and around 49% on the AGX.

The ARM Cortex-A72 is able to realize much less parallelism than that of the embedded GPUs. Therefore, the more serial nature of the FLOPs measurement is more indicative of latency performance than CDL. With more filters earlier in the network, and therefore more FLOPs, the serial throughput of the NeuralScale models is more limited than the uniform-scaling models on the CPU. In the worst

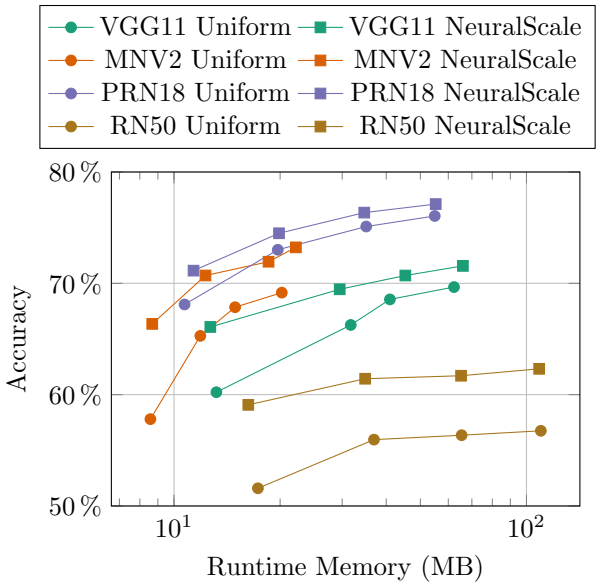

Figure 11: Accuracy versus MB of runtime memory for ARM Cortex-A72. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

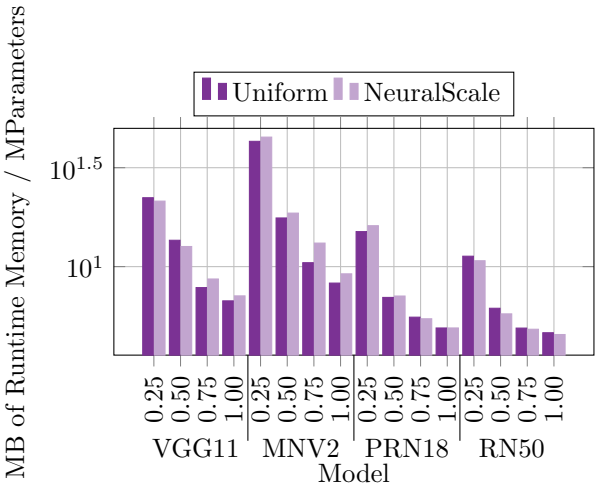

Figure 12: MB of runtime memory per million parameters for ARM Cortex-A72. Log scale on y-axis.

case, the MobileNetV2 model at 0.25× scaling is over 3.2× slower for NeuralScale than for uniform scaling while having 3.9× the number of FLOPs.

Clearly, neither FLOPs nor CDL tell the whole story. However, by seeing that there is an increase in FLOPs for NeuralScale and by assuming that no parallelization is perfect, it can be correctly predicted that NeuralScale will be slower than uniform scaling. This latency increase may be acceptable on the GPU depending on the application, especially since the accuracy versus latency can be more efficient in some cases using NeuralScale as shown in Fig. 9 and Fig. 10. However, with already long runtimes, the relative speed on the CPU for NeuralScale models has a potentially larger effect on model selection. It too has similar performance in terms of accuracy versus latency as seen in Fig. 14, but the latency values are simply much larger than those on the embedded GPUs.

Parameter Scaling versus Runtime-Memory Usage

One of the results found in this research is that the number of parameters in a network does not necessarily represent an important metric during inference for embedded GPUs using TensorRT. The NeuralScale method assumes that parameters are a useful scaling metric.[1] However, the number of parameters in a network does not scale with the runtime-memory usage of the models. As shown by Fig. 5 and Fig. 6, even with a similar number of parameters between two models, the NeuralScale method uses significantly more runtime memory during inference than the uniform-scaling method. As the NeuralScale models contain more parallelism throughout the network shown by Fig. 15, it likely needs more runtime memory to handle processing of the increased parallelism on the GPUs. With large variations in runtime-memory usage across the models tested on the embedded GPUs using TensorRT, this research concludes that the assumption that scaling the parameters makes a model more efficient for a resource-constrained system does not hold true.

Continuing, the points between NeuralScale and uniform scaling tend to be close in terms of accuracy for a similar amount of runtime memory. This indicates that while the parameter selection in NeuralScale may be more intelligent, it does not result in gains in accuracy. The accuracy gains are counteracted by extra memory requirements for a similar number of parameters. Essentially, parameter scaling can help reduce the model size, but its overall effect on memory requirements is not straightforward. Therefore, choosing a scaling ratio is non-trivial and would require testing with the desired device.

It could be useful to scale the network based on the amount of runtime memory used for inference rather than the number of parameters. However, this change would add significant cost to the NeuralScale architecture descent method. Since the Ten-

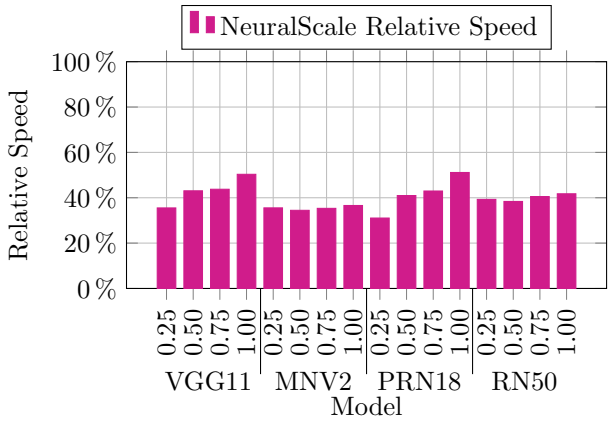

Figure 13: NeuralScale relative speed (uniform latency / NeuralScale latency) for ARM Cortex-A72.

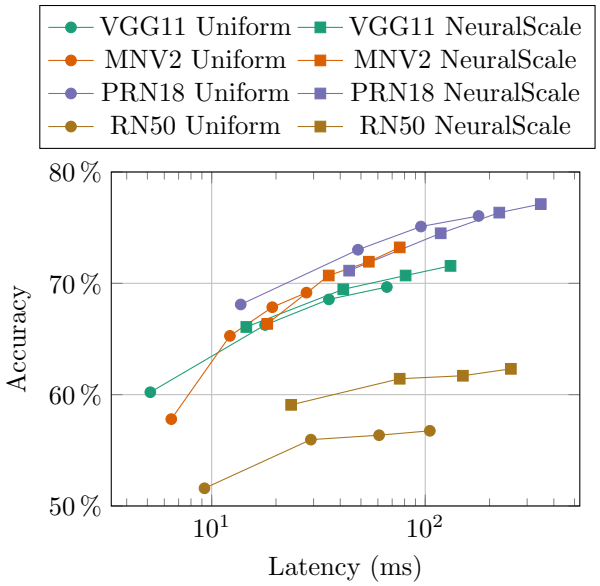

Figure 14: Accuracy versus latency (ms) for ARM Cortex-A72. Log scale on x-axis. Accuracy measurements for VGG11, MNV2, and PRN18 are from NeuralScale paper.[1]

sorRT engines used are created on-device, the measurement of runtime memory is not an easy metric to collect during model design in NAS without running every model on the target device. Of course, TensorRT has a significant effect on how these models perform, but it represents a common framework that is likely to be used on these embedded GPUs and is recommended by NVIDIA due to its inference speed.[29]

On the CPU side, when using ONNX Runtime with the ARM Cortex-A72, there was essentially no difference between the runtime memory required to operate a NeuralScale model versus its uniform-scaling model equivalent. This is interesting since the trend contrasts from that of the GPUs with TensorRT. The main TensorRT code is proprietary, so it is challenging to pick out the specific cause of the increase in memory usage on the embedded GPUs.

CONCLUSION

As computer-vision applications move towards embedded systems while models continue to grow, creating networks appropriate for resource-constrained systems is becoming increasingly important. Much research has been done into using NAS to create models rather than developing purely human-designed networks. Within NAS, methods like NeuralScale aim to take a pre-designed network and scale it for a target system. NeuralScale does this scaling by changing the width of network layers to optimize accuracy for a given number of parameters. NeuralScale can be compared to more simple methods like uniform scaling.

When using an embedded GPU with TensorRT, this research finds that parameter scaling does not have a strong analog in terms of model performance. Models scaled with NeuralScale use significantly more runtime memory than their uniform-scaling counterparts. They also run slower and the gains in accuracy are smaller than the growth in runtime-memory usage. When averaged over the tested models on the GPUs, NeuralScale creates models that are only 40% as efficient in terms of accuracy per megabyte of runtime memory as the uniform-scaling method. This drop in efficiency is mostly caused by the fact that NeuralScale uses on average 2.7× the runtime memory per parameter as uniform scaling, a major increase for a method targeting resource-constrained systems. NeuralScale can be used when runtime-memory usage is not a concern. Uniform scaling is a safer choice when runtime memory is a limiting factor.

On the embedded CPU, scaling the number of parameters does not have an adverse effect on the runtime memory. However, the models run significantly slower. Averaged over the tested models, NeuralScale models take over 2.5× longer than their uniform scaling counterpart. For the CPU, NeuralScale can be used when latency is not a concern. Uniform scaling is a safer choice when latency is a limiting factor. However, as demonstrated by this research, simply designing or scaling a model for a set number of parameters does not bring perfor-

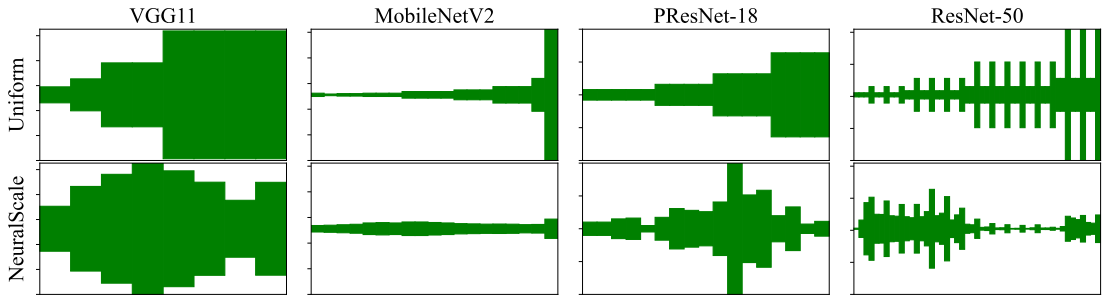

Figure 15: Layer widths at 1.00× scaling.

mance guarantees in terms of runtime memory for embedded GPUs with TensorRT and in terms of latency for embedded CPUs with ONNX Runtime.

FUTURE RESEARCH

This research is limited by the number of models tested and by the comparison of scaling methods. Further comparisons should be performed to other scaling methods. Additionally, further research can be done into the runtime-memory usage and memory-access trends of the networks. A dataset with larger images should be tested to measure possible effects due to limited memory bandwidth. Finally, these methods should be tested with other inference tools to determine their effect on performance.

ACKNOWLEDGMENT

This research was supported by the NSF Center for Space, High-performance, and Resilient Computing (SHREC) industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783. This research was supported in part by the University of Pittsburgh Center for Research Computing through GPU cluster resources provided for model training. We would like to thank the students of NSF-SHREC. Specifically, we thank David Langerman and Evan Gretok for their guidance and assistance in this research.

REFERENCES

[1] Eugene Lee and Chen Yi Lee. NeuralScale: Efficient scaling of neurons for resource-constrained deep neural networks. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1475–1484. IEEE Computer Society, 2020.

[2] 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:1704.04861 [cs]*, April 2017. URL `http://arxiv.org/abs/1704.04861`. arXiv: 1704.04861.

[3] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *Technical Report, University of Toronto*, 2009. ISSN 1098-6596.

[4] David Langerman, Alex Johnson, Kyle Buettner, and Alan D. George. Beyond Floating-Point Ops: CNN Performance Prediction with Critical Datapath Length. In *Proceedings of the 2020 IEEE High Performance Extreme Computing Conference, HPEC 2020*. Institute of Electrical and Electronics Engineers Inc., September 2020.

[5] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan Lin. HW-NAS-Bench: Hardware-Aware Neural Architecture Search Benchmark. In *Proceedings of the International Conference on Learning Representations*. International Conference on Learning Representations, ICLR, 2021. URL `https://github.com/RICE-EIC/HW-NAS-Bench.`. arXiv: 2103.10584.

[6] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019*, volume 2019-June. ICML, Long Beach, California, USA, 2019.

[7] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. In *Proceedings of the 7th International Conference on Learning*

Representations, ICLR 2019, pages 1–13. International Conference on Learning Representations, ICLR, New Orleans, Louisiana, USA, 2019. arXiv: 1812.00332.

[8] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *Proceedings of the International Conference on Learning Representations*, pages 1–15. International Conference on Learning Representations, ICLR, Addis Ababa, Ethiopia, 2020. URL `https://github.com/mit-han-lab/once-for-all.`. arXiv: 1908.09791.

[9] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, Salt Lake City, Utah, USA, 2018. ISSN: 10636919.

[10] NVIDIA. Developer Guide :: NVIDIA Deep Learning TensorRT Documentation. URL `https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-723/developer-guide/index.html`.

[11] Junjie Bai, Fang Lu, Ke Zhang, and others. ONNX: Open Neural Network Exchange, 2019. URL `https://github.com/onnx/onnx`. Publication Title: GitHub repository.

[12] NVIDIA. ONNX-TensorRT: TensorRT Backend For ONNX. URL `https://github.com/onnx/onnx-tensorrt`. Publication Title: GitHub repository.

[13] ONNX Runtime developers. ONNX Runtime, 2021. URL `https://onnxruntime.ai/`.

[14] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2019-June, pages 11256–11264. IEEE Computer Society, Long Beach, California, USA, June 2019.

[15] NVIDIA. Jetson Nano 2GB Developer Kit. URL `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/education-projects/`.

[16] NVIDIA. Jetson Modules. URL `https://developer.nvidia.com/embedded/jetson-modules`.

[17] NVIDIA. GeForce GTX 1080 Ti Graphics Cards. URL `https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1080-ti/`.

[18] TechPowerUp. NVIDIA Jetson AGX Xavier GPU Specs. URL `https://www.techpowerup.com/gpu-specs/jetson-agx-xavier-gpu.c3232`.

[19] TechPowerUp. NVIDIA GeForce GTX 1080 Ti Specs. URL `https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877`.

[20] Raspberry Pi Foundation. Raspberry Pi 4 Model B specifications – Raspberry Pi. URL `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/`.

[21] NXP Semiconductors. NXP i.MX 8QuadMax MIMX8QM6AVUFFAB. URL `https://www.nxp.com/part/MIMX8QM6AVUFFAB`.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015*. International Conference on Learning Representations, ICLR, San Diego, California, USA, 2015.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-Decem, pages 770–778. IEEE Computer Society, Las Vegas, Nevada, USA, 2016. ISSN: 10636919.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9908 LNCS, pages 630–645. Springer International Publishing, Amsterdam, The Netherlands, 2016. ISBN 978-3-319-46492-3. arXiv: 1603.05027 ISSN: 16113349.

[25] Ligeng Zhu. PyTorch-OpCounter. URL `https://github.com/Lyken17/pytorch-OpCounter`.

[26] Dustin Franklin. jetson-containers: Machine Learning Containers for NVIDIA Jetson and JetPack-L4T, 2020. URL `https://github.com/dusty-nv/jetson-containers`.

[27] Fabian Pedregosa and Philippe Gervais. memory_profiler: Monitor Memory usage of Python code. URL `https://github.com/pythonprofilers/memory_profiler`.

[28] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In *Proceedings of the European Conference on Computer Vision*. Springer International Publishing, Zurich, Switzerland, 2014.

[29] NVIDIA. NVIDIA TensorRT. URL `https://developer.nvidia.com/tensorrt`.