ONCE-FOR-ALL: TRAIN ONE NETWORK AND SPECIALIZE IT FOR EFFICIENT DEPLOYMENT ON DIVERSE HARDWARE PLATFORMS

Han Cai¹, Chuang Gan², Tianzhe Wang¹, Zhekai Zhang¹, Song Han¹
¹Massachusetts Institute of Technology, ²MIT-IBM Watson AI Lab
{hancai, chuangg, songhan}@mit.edu

ABSTRACT

We address the challenging problem of efficient deep learning model deployment across many devices and diverse constraints, from general-purpose hardware to specialized accelerators. Conventional approaches either manually design or use neural architecture search (NAS) to find a specialized neural network and train it from scratch for each case, which is computationally prohibitive (causing CO₂ emission as much as 5 cars’ lifetime Strubell et al. (2019)) thus unscalable. To reduce the cost, our key idea is to decouple model training from architecture search. To this end, we propose to train a once-for-all network (OFA) that supports diverse architectural settings (depth, width, kernel size, and resolution). Given a deployment scenario, we can then quickly get a specialized sub-network by selecting from the OFA network without additional training. To prevent interference between many sub-networks during training, we also propose a novel progressive shrinking algorithm, which can train a surprisingly large number of sub-networks (> $10^{19}$) simultaneously. Extensive experiments on various hardware platforms (CPU, GPU, mCPU, mGPU, FPGA accelerator) show that OFA consistently outperforms SOTA NAS methods (up to 4.0% ImageNet top1 accuracy improvement over MobileNetV3) while reducing orders of magnitude GPU hours and CO₂ emission. In particular, OFA achieves a new SOTA 80.0% ImageNet top1 accuracy under the mobile setting (<600M FLOPs). Code and pre-trained models are released at https://github.com/mit-han-lab/once-for-all.

1 INTRODUCTION

Deep Neural Networks (DNNs) deliver state-of-the-art accuracy in many machine learning applications. However, the explosive growth in model size and computation cost gives rise to new challenges on how to efficiently deploy these deep learning models on diverse hardware platforms, since they have to meet different hardware efficiency constraints (e.g., latency, energy). For instance, one mobile application on App Store has to support a diverse range of hardware devices, from a high-end Samsung Note10 with a dedicated neural network accelerator to a 5-year-old Samsung S6 with a much slower processor. With different hardware resources (e.g., on-chip memory size, #arithmetic units), the optimal neural network architecture varies significantly. Even running on the same hardware, under different battery conditions or workloads, the best model architecture also differs a lot.

Given different hardware platforms and efficiency constraints (defined as deployment scenarios), researchers either design compact models specialized for mobile (Howard et al., 2017; Sandler et al., 2018; Zhang et al., 2018) or accelerate the existing models by compression (Han et al., 2016; He et al., 2018) for efficient deployment. However, designing specialized DNNs for every scenario is engineer-expensive and computationally expensive, either with human-based methods or NAS. Since such methods need to repeat the network design process and retrain the designed network from scratch for each case. Their total cost grows linearly as the number of deployment scenarios increases, which will result in excessive energy consumption and CO₂ emission (Strubell et al., 2019). It makes them unable to handle the vast amount of hardware devices (23.14 billion IoT devices till...
We extensively evaluated the effectiveness of OFA on ImageNet with many hardware platforms (CPU, GPU, mCPU, mGPU, FPGA accelerator) and efficiency constraints. Under all deployment scenarios, OFA consistently improves the ImageNet accuracy by a significant margin compared to SOTA hardware-aware NAS methods while saving the GPU hours, dollars, and CO₂ emission by orders of magnitude. On the ImageNet mobile setting (less than 600M FLOPs), OFA achieves a new SOTA 80.0% top1 accuracy with 595M FLOPs. To the best of our knowledge, this is the first time that the SOTA ImageNet top1 accuracy reaches 80% under the mobile setting.

2 Related Work

Efficient Deep Learning. Many efficient neural network architectures are proposed to improve the hardware efficiency, such as SqueezeNet (Iandola et al., 2016), MobileNets (Howard et al., 2017; 2018) and highly dynamic deployment environments (different battery conditions, different latency requirements, etc.).
Sandler et al., 2018), ShuffleNets (Ma et al., 2018; Zhang et al., 2018), etc. Orthogonal to architecting efficient neural networks, model compression (Han et al., 2016) is another very effective technique for efficient deep learning, including network pruning that removes redundant units (Han et al., 2015) or redundant channels (He et al., 2018; Liu et al., 2017), and quantization that reduces the bit width for the weights and activations (Han et al., 2016; Courbariaux et al., 2015; Zhu et al., 2017).

**Neural Architecture Search.** Neural architecture search (NAS) focuses on automating the architecture design process (Zoph & Le, 2017; Zoph et al., 2018; Real et al., 2019; Cai et al., 2018a; Liu et al., 2019). Early NAS methods (Zoph et al., 2018; Real et al., 2019; Cai et al., 2018b) search for high-accuracy architectures without taking hardware efficiency into consideration. Therefore, the produced architectures (e.g., NASNet, AmoebaNet) are not efficient for inference. Recent hardware-aware NAS methods (Cai et al., 2019; Tan et al., 2019; Wu et al., 2019) directly incorporate the hardware feedback into architecture search. As a result, they are able to improve inference efficiency. However, given new inference hardware platforms, these methods need to repeat the architecture search process and retrain the model, leading to prohibitive GPU hours, dollars and CO$_2$ emission. They are not scalable to a large number of deployment scenarios. The individually trained models do not share any weight, leading to a large total model size and high downloading bandwidth.

**Dynamic Neural Networks.** To improve the efficiency of a given neural network, some work explored skipping part of the model based on the input image. For example, Wu et al. (2018); Liu & Deng (2018); Wang et al. (2018) learn a controller or gating modules to adaptively drop layers; Huang et al. (2018) introduce early-exit branches in the computation graph; Lin et al. (2017) adaptively prune channels based on the input feature map; Kuen et al. (2018) introduce stochastic downsampling point to reduce the feature map size adaptively. Recently, Slimmable Nets (Yu et al., 2019; Yu & Huang, 2019b) propose to train a model to support multiple width multipliers (e.g., 4 different global width multipliers), building upon existing human-designed neural networks (e.g., MobileNetV2 0.35, 0.5, 0.75, 1.0). Such methods can adaptively fit different efficiency constraints at runtime, however, still inherit a pre-designed neural network (e.g., MobileNet-v2), which limits the degree of flexibility (e.g., only width multiplier can adapt) and the ability in handling new deployment scenarios where the pre-designed neural network is not optimal. In this work, in contrast, we enable a much more diverse architecture space (depth, width, kernel size, and resolution) and a significantly larger number of architectural settings ($10^{19}$ v.s. 4 (Yu et al., 2019)). Thanks to the diversity and the large design space, we can derive new specialized neural networks for many different deployment scenarios rather than working on top of an existing neural network that limits the optimization headroom. However, it is more challenging to train the network to achieve this flexibility, which motivates us to design the progressive shrinking algorithm to tackle this challenge.

### 3 Method

#### 3.1 Problem Formalization

Assuming the weights of the once-for-all network as $W_o$ and the architectural configurations as $\{arch_i\}$, we then can formalize the problem as

$$
\min_{W_o} \sum_{arch_i} L_{val}(C(W_o, arch_i)),
$$

where $C(W_o, arch_i)$ denotes a selection scheme that selects part of the model from the once-for-all network $W_o$ to form a sub-network with architectural configuration $arch_i$. The overall training objective is to optimize $W_o$ to make each supported sub-network maintain the same level of accuracy as independently training a network with the same architectural configuration.

#### 3.2 Architecture Space

Our once-for-all network provides one model but supports many sub-networks of different sizes, covering four important dimensions of the convolutional neural networks (CNNs) architectures, i.e., depth, width, kernel size, and resolution. Following the common practice of many CNN models (He et al., 2016; Sandler et al., 2018; Huang et al., 2017), we divide a CNN model into a sequence of units with gradually reduced feature map size and increased channel numbers. Each unit consists of a
sequence of layers where only the first layer has stride 2 if the feature map size decreases (Sandler et al., 2018). All the other layers in the units have stride 1.

We allow each unit to use arbitrary numbers of layers (denoted as elastic depth); For each layer, we allow to use arbitrary numbers of channels (denoted as elastic width) and arbitrary kernel sizes (denoted as elastic kernel size). In addition, we also allow the CNN model to take arbitrary input image sizes (denoted as elastic resolution). For example, in our experiments, the input image size ranges from 128 to 224 with a stride 4; the depth of each unit is chosen from {2, 3, 4}; the width expansion ratio in each layer is chosen from {3, 4, 6}; the kernel size is chosen from {3, 5, 7}. Therefore, with 5 units, we have roughly \((3 \times 3)^2 + (3 \times 3)^3 + (3 \times 3)^4\) \( \approx 2 \times 10^{19}\) different neural network architectures and each of them can be used under 25 different input resolutions. Since all of these sub-networks share the same weights (i.e., \(W_o\) (Cheung et al., 2019), we only require 7.7M parameters to store all of them. Without sharing, the total model size will be prohibitive.

3.3 Training the Once-for-all Network

Naïve Approach. Training the once-for-all network can be cast as a multi-objective problem, where each objective corresponds to one sub-network. From this perspective, a naïve training approach is to directly optimize the once-for-all network from scratch using the exact gradient of the overall objective, which is derived by enumerating all sub-networks in each update step, as shown in Eq. (1).

The cost of this approach is linear to the number of sub-networks. Therefore, it is only applicable to scenarios where a limited number of sub-networks are supported (Yu et al., 2019), while in our case, it is computationally prohibitive to adopt this approach.

Another naïve training approach is to sample a few sub-networks in each update step rather than enumerate all of them, which does not have the issue of prohibitive cost. However, with such a large number of sub-networks that share weights thus interfere with each other, we find it suffers from significant accuracy drop. In the following section, we introduce a solution to address this challenge by adding a progressive shrinking training order to the training process. Correspondingly, we refer to the naïve training approach as random order.

Progressive Shrinking. The once-for-all network comprises many sub-networks of different sizes where small sub-networks are nested in large sub-networks. To prevent interference between the sub-networks, we propose to enforce a training order from large sub-networks to small sub-networks in a progressive manner. We name this training order as progressive shrinking (PS). An example of the training process with PS is provided in Figure 2, where we start with training the largest neural network with the maximum kernel size (i.e., 7), depth (i.e., 4), and width (i.e., 6). Next, we progressively fine-tune the network to support smaller sub-networks by gradually adding them into the sampling space (larger sub-networks may also be sampled). Specifically, after training the largest network, we first support elastic kernel size which can choose from \{3, 5, 7\} at each layer, while the depth and width remain the maximum values. Then, we support elastic depth and elastic width sequentially, as is shown in Figure 2. The resolution is elastic throughout the whole training process, which is implemented by sampling different image sizes for each batch of training data. We also use the knowledge distillation technique after training the largest neural network (Hinton et al., 2015; Ashok et al., 2018; Yu & Huang, 2019b). It combines two loss terms using both the soft labels given by the largest neural network and the real labels.
Published as a conference paper at ICLR 2020

Figure 3: Left: Kernel transformation matrix for elastic kernel size. Right: Progressive shrinking for elastic depth. Instead of skipping each layer independently, we keep the first $D$ layers and skip the last $(4 - D)$ layers. The weights of the early layers are shared.

Figure 4: Progressive shrinking for elastic width. In this example, we progressively support 4, 3, and 2 channel settings. We perform channel sorting and pick the most important channels (with large L1 norm) to initialize the smaller channel settings. The important channels’ weights are shared.

Compared to the naïve approach, PS prevents small sub-networks from interfering large sub-networks, since large sub-networks are already well-trained when the once-for-all network is fine-tuned to support small sub-networks. Additionally, during fine-tuning, the model is optimized in the local space around the well-trained large sub-networks by using a small learning rate and revisiting (i.e., sampling) well-trained large sub-networks. Regarding the small sub-networks, they share the weights with the large ones. Therefore, PS allows initializing small sub-networks with the most important weights of well-trained large sub-networks, which expedites the training process. We describe the details of the PS training flow as follows:

- **Elastic Kernel Size** (Figure 3 left). We let the center of a 7x7 convolution kernel also serve as a 5x5 kernel, the center of which can also be a 3x3 kernel. Therefore, the kernel size becomes elastic. The challenge is that the centering sub-kernels (e.g., 3x3 and 5x5) are shared and need to play multiple roles (independent kernel and part of a large kernel). The weights of centered sub-kernels may need to have different distribution or magnitude as different roles. Forcing them to be the same degrades the performance of some sub-networks. Therefore, we introduce kernel transformation matrices when sharing the kernel weights. We use separate kernel transformation matrices for different layers. Within each layer, the kernel transformation matrices are shared among different channels. As such, we only need $25 \times 25 + 9 \times 9 = 706$ extra parameters to store the kernel transformation matrices in each layer, which is negligible.

- **Elastic Depth** (Figure 3 right). To derive a sub-network that has $D$ layers in a unit that originally has $N$ layers, we keep the first $D$ layers and skip the last $N - D$ layers, rather than keeping any $D$ layers as done in current NAS methods (Cai et al., 2019; Wu et al., 2019). As such, one depth setting only corresponds to one combination of layers. In the end, the weights of the first $D$ layers are shared between large and small models.

- **Elastic Width** (Figure 4). Width means the number of channels. We give each layer the flexibility to choose different channel expansion ratios. Following the progressive shrinking scheme, we first train a full-width model. Then we introduce a channel sorting operation to support partial widths. It reorganizes the channels according to their importance, which is calculated based on the L1 norm of a channel’s weight. Larger L1 norm means more important. For example, when shrinking from a 4-channel-layer to a 3-channel-layer, we select the largest 3 channels, whose weights are shared with the 4-channel-layer (Figure 4 left and middle). Thereby, smaller sub-networks are initialized with the most important channels on the once-for-all network which is already well trained. This channel sorting operation preserves the accuracy of larger sub-networks.
## Table 1: ImageNet top1 accuracy (%) performances of sub-networks under resolution 224 × 224.

<table>
<thead>
<tr>
<th>Sub-networks</th>
<th>D = 2</th>
<th>D = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W = 3</td>
<td>W = 6</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K = 3 K = 7</td>
<td>3.4M</td>
<td>3.5M</td>
</tr>
<tr>
<td>K = 3 K = 7</td>
<td>121M</td>
<td>151M</td>
</tr>
<tr>
<td>Random Order</td>
<td>68.0</td>
<td>69.1</td>
</tr>
<tr>
<td>PS (ours)</td>
<td>70.5</td>
<td>71.9</td>
</tr>
<tr>
<td>∆ Acc.</td>
<td>+2.5</td>
<td>+2.8</td>
</tr>
</tbody>
</table>

Table 1: ImageNet top1 accuracy (%) performances of sub-networks under resolution 224 × 224. “(D = d, W = w, K = k)” denotes a sub-network with d layers in each unit, and each layer has an width expansion ratio w and kernel size k. “Mbv3-L” denotes “MobileNetV3-Large”.

### 3.4 Specialized Model Deployment with Once-for-all Network

Having trained a once-for-all network, the next stage is to derive the specialized sub-network for a given deployment scenario. The goal is to search for a neural network that satisfies the efficiency (e.g., latency, energy) constraints on the target hardware while optimizing the accuracy. Since OFA decouples model training from architecture search, we do not need any training cost in this stage. Furthermore, we build neural-network-twins to predict the latency and accuracy given a neural network architecture, providing a quick feedback for model quality. It eliminates the repeated search cost by substituting the measured accuracy/latency with predicted accuracy/latency (twins).

Specifically, we randomly sample 16K sub-networks with different architectures and input image sizes, then measure their accuracy on 10K validation images sampled from the original training set. These [architecture, accuracy] pairs are used to train an accuracy predictor to predict the accuracy of a model given its architecture and input image size\(^2\). We also build a latency lookup table (Cai et al., 2019) on each target hardware platform to predict the latency. Given the target hardware and latency constraint, we conduct an evolutionary search (Real et al., 2019) based on the neural-network-twins to get a specialized sub-network. Since the cost of searching with neural-network-twins is negligible, we only need 40 GPU hours to collect the data pairs, and the cost stays constant regardless of #deployment scenarios.

### 4 Experiments

In this section, we first apply the progressive shrinking algorithm to train the once-for-all network on ImageNet (Deng et al., 2009). Then we demonstrate the effectiveness of our trained once-for-all network on various hardware platforms (Samsung S7 Edge, Note8, Note10, Google Pixel1, Pixel2, LG G8, NVIDIA 1080Ti, V100 GPUs, Jetson TX2, Intel Xeon CPU, Xilinx ZU9EG, and ZU3EG FPGAs) with different latency constraints.

#### 4.1 Training the Once-for-all Network on ImageNet

**Training Details.** We use the same architecture space as MobileNetV3 (Howard et al., 2019). For training the full network, we use the standard SGD optimizer with Nesterov momentum 0.9 and weight decay \(3e^{-5}\). The initial learning rate is 2.6, and we use the cosine schedule (Loshchilov & Hutter, 2016) for learning rate decay. The full network is trained for 180 epochs with batch size 2048 on 32 GPUs. Then we follow the schedule described in Figure 2 to further fine-tune the full network\(^3\). The whole training process takes around 1,200 GPU hours on V100 GPUs. This is a one-time training cost that can be amortized by many deployment scenarios.

**Results.** Table 1 reports the top1 accuracy of sub-networks derived from the once-for-all networks that are trained with our progressive shrinking (PS) algorithm and random order respectively. Due to space limits, we take 8 sub-networks for comparison, and each of them is denoted as “(D = d, W = w, K = k)”. It represents a sub-network that has d layers for all units while the expansion ratio and kernel size are set to w and k for all layers. Compared to random order, PS can improve

\(^2\)Details of the accuracy predictor is provided in Appendix A.

\(^3\)Implementation details can be found in Appendix B.
Table 2: Comparison with SOTA hardware-aware NAS methods on Pixel1 phone. OFA decouples model training from architecture search. The search cost and training cost both stay constant as the number of deployment scenarios grows. “#25” denotes the specialized sub-networks are fine-tuned for 25 epochs after grabbing weights from the once-for-all network. “CO2e” denotes CO2 emission which is calculated based on Strubell et al. (2019). AWS cost is calculated based on the price of on-demand P3.16xlarge instances.

4.2 Specialized Sub-networks for Different Hardware and Constraints

We apply our trained once-for-all network to get different specialized sub-networks for diverse hardware platforms: from the cloud to the edge. On cloud devices, the latency for GPU is measured with batch size 64 on NVIDIA 1080Ti and V100 with Pytorch 1.0+cuDNN. The CPU latency is measured with batch size 1 on Intel Xeon E5-2690 v4+MKL-DNN. On edge devices, we use Samsung, Google and LG phones with TF-Lite, batch size 1; for mobile GPU, we use Jetson TX2 with Pytorch 1.0+cuDNN, batch size of 16; for embedded FPGA, we use Xilinx ZU9EG and ZU3EG FPGAs with Vitis AI.

Comparison with NAS on Mobile Devices. Table 2 reports the comparison between OFA and state-of-the-art hardware-aware NAS methods on the mobile phone (Pixel1). OFA is much more efficient than NAS when handling multiple deployment scenarios since the cost of OFA is constant while others are linear to the number of deployment scenarios (N). With N = 40, the total CO2 emissions of OFA is 16× fewer than ProxylessNAS, 19× fewer than FBNet, and 1,300× fewer than MnasNet (Figure 5). Without retraining, OFA achieves 76.0% top1 accuracy on ImageNet, which is 0.8% higher than MobileNetV3-Large while maintaining similar mobile latency. We can further improve the top1 accuracy to 76.4% by fine-tuning the specialized sub-network for 25 epochs and to 76.9% by fine-tuning for 75 epochs. Besides, we also observe that OFA with PS can achieve 3.6% better accuracy than without PS, showing the effectiveness of PS.

OFA under Different Computational Resource Constraints. Figure 6 summarizes the results of OFA under different FLOPs and Pixel1 latency constraints. OFA achieves 79.1% ImageNet top1...
With 595M FLOPs, OFA reaches a new SOTA 80.0% ImageNet top1 accuracy under the mobile scenarios with 72.6% (+12.3% improvement) on the 1080Ti GPU. Detailed architectures of our specialized models latency by a significant margin, especially on GPUs which have more parallelism. With similar latency OFA for Diverse Hardware Platforms.

Besides the mobile platforms, we extensively studied methods (Tan et al., 2019; Cai et al., 2019) due to the prohibitive training cost. Figure 7 reports detailed comparisons between OFA and MobileNetV3 on six mobile devices. More importantly, OFA runs much faster than EfficientNets on hardware. Specifically, with 143ms Pixel1 latency, OFA achieves 80.1% ImageNet top1 accuracy, being 0.3% more accurate and 2.6x faster than EfficientNet-B2.

Figure 7: OFA consistently outperforms MobileNetV3 on mobile platforms.

OFA for Diverse Hardware Platforms. Besides the mobile platforms, we extensively studied the effectiveness of OFA on six additional hardware platforms (Figure 8) using the ProxylessNAS architecture space (Cai et al., 2019). OFA consistently improves the trade-off between accuracy and latency by a significant margin, especially on GPUs which have more parallelism. With similar latency as MobileNetV2 0.35, “OFA #25” improves the ImageNet top1 accuracy from MobileNetV2’s 60.3% to 72.6% (+12.3% improvement) on the 1080Ti GPU. Detailed architectures of our specialized models are shown in Figure 11. It reveals the insight that using the same model for different deployment scenarios with only the width multiplier modified has a limited impact on efficiency improvement: the accuracy drops quickly as the latency constraint gets tighter.
OFA for Specialized Hardware Accelerators. There has been plenty of work on NAS for general-purpose hardware, but little has been focused on specialized hardware accelerators. We quantitatively analyzed the performance of OFA on two FPGAs accelerators (ZU3EG and ZU9EG) using Xilinx Vitis AI with 8-bit quantization, and discuss two design principles.

**Principle 1:** memory access is expensive, computation is cheap. An efficient CNN should do as much as computation with a small amount of memory footprint. The ratio is defined as the arithmetic intensity (OPs/Byte). The higher OPs/Byte, the less memory bounded, the easier to parallelize. Thanks to OFA’s diverse choices of sub-network architectures (10\textsuperscript{19}) (Section 3.3), and the OFA model twin that can quickly give the accuracy/latency feedback (Section 3.4), the evolutionary search can automatically find a CNN architecture that has higher arithmetic intensity. As shown in Figure 9, OFA’s arithmetic intensity is 48%/43% higher than MobileNetV2 and MnasNet (MobileNetV3 is not supported by Xilinx Vitis AI). Removing the memory bottleneck results in higher utilization and GOPS/s by 70%-90%, pushing the operation point to the upper-right in the roofline model (Williams et al., 2009), as shown in Figure 10. (70%-90% looks small in the log scale but that is significant).

**Principle 2:** the CNN architecture should be co-designed with the hardware accelerator’s cost model. The FPGA accelerator has a specialized depth-wise engine that is pipelined with the point-wise engine. The pipeline throughput is perfectly matched for 3x3 kernels. As a result, OFA’s searched model only has 3x3 kernel (Figure 11, a) on FPGA, despite 5x5 and 7x7 kernels are also in the search space. Additionally, large kernels sometimes cause “out of BRAM” error on FPGA, giving high cost. On Intel Xeon CPU, however, more than 50% operations are large kernels. Both FPGA and GPU models are wider than CPU, due to the large parallelism of the computation array.
FPGA Arithmetic Intensity (op/B)
MobileNetV2 MnasNet OFA
0.35 27 27.6 39.4
0.5 35.3 37.1 49.4
0.75 51.6 51.9...

Table 1: Comparison of OFA, MobileNet-v2, and MnasNet on ZU3EG, ZU9EG, and UltraZed-EG FPGAs. OFA provides a higher arithmetic intensity and lower latency than other models. (OFA) is a new methodology that decouples model training from architecture search for efficient deep learning deployment under a large number of hardware platforms.

Figure 10: Quantitative study of OFA’s roofline model on Xilinx ZU9EG and ZU3EG FPGAs (log scale). OFA model increased the arithmetic intensity by 33%/43% and GOPS/s by 72%/92% on these two FPGAs compared with MnasNet.

Figure 11: OFA can design specialized models for different hardware and different latency constraint. “MB 4x3” means “mobile block with expansion ratio 4, kernel size 3x3”. FPGA and GPU models are wider than CPU model due to larger parallelism. Different hardware has different cost model, leading to different optimal CNN architectures. OFA provides a unified and efficient design methodology.

5 Conclusion

We proposed Once-for-All (OFA), a new methodology that decouples model training from architecture search for efficient deep learning deployment under a large number of hardware platforms. Unlike previous approaches that design and train a neural network for each deployment scenario, we designed a once-for-all network that supports different architectural configurations, including elastic depth, width, kernel size, and resolution. It reduces the training cost (GPU hours, energy consumption, and CO2 emission) by orders of magnitude compared to conventional methods. To prevent sub-networks of different sizes from interference, we proposed a progressive shrinking algorithm that enables a large number of sub-network to achieve the same level of accuracy compared to training them independently. Experiments on a diverse range of hardware platforms and efficiency constraints demonstrated the effectiveness of our approach. OFA provides an automated ecosystem to efficiently design efficient neural networks with the hardware cost model in the loop.
ACKNOWLEDGMENTS

We thank NSF Career Award #1943349, MIT-IBM Watson AI Lab, Google-Daydream Research Award, Samsung, Intel, Xilinx, SONY, AWS Machine Learning Research Award for supporting this research. We thank Samsung, Google and LG for donating mobile phones.

REFERENCES


Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In AAAI, 2018a. 3

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In ICML, 2018b. 3


Brian Cheung, Alex Terekhov, Yubei Chen, Pulkit Agrawal, and Bruno Olshausen. Superposition of many models into one. In NeurIPS, 2019. 4


Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In CVPR, 2009. 6


Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In NeurIPS, 2015. 3


Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In ECCV, 2018. 1, 3


Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018. 2


Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009. 9


Jiahui Yu and Thomas Huang. Universally slimmable networks and improved training techniques. In *ICCV*, 2019b. 3, 4


Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. In ICLR, 2017. 3


A DETAILS OF THE ACCURACY PREDICTOR

We use a three-layer feedforward neural network that has 400 hidden units in each layer as the accuracy predictor. Given a model, we encode each layer in the neural network into a one-hot vector based on its kernel size and expand ratio, and we assign zero vectors to layers that are skipped. Besides, we have an additional one-hot vector that represents the input image size. We concatenate these vectors into a large vector that represents the whole neural network architecture and input image size, which is then fed to the three-layer feedforward neural network to get the predicted accuracy. In our experiments, this simple accuracy prediction model can provide very accurate predictions. At convergence, the root-mean-square error (RMSE) between predicted accuracy and estimated accuracy on the test set is only 0.21%. Figure 12 shows the relationship between the RMSE of the accuracy prediction model and the final results (i.e., the accuracy of selected sub-networks). We can find that lower RMSE typically leads to better final results.

![Figure 12: Performances of selected sub-networks using different accuracy prediction model.](image)

B IMPLEMENTATION DETAILS OF PROGRESSIVE SHRINKING

After training the full network, we first have one stage of fine-tuning to incorporate elastic kernel size. In this stage (i.e., $K \in [7, 5, 3]$), we sample one sub-network in each update step. The network is fine-tuned for 125 epochs with an initial learning rate of 0.96. All other training settings are the same as training the full network.

Next, we have two stages of fine-tuning to incorporate elastic depth. We sample two sub-networks and aggregate their gradients in each update step. The first stage (i.e., $D \in [4, 3]$) takes 25 epochs with an initial learning rate of 0.08 while the second stage (i.e., $D \in [4, 3, 2]$) takes 125 epochs with an initial learning rate of 0.24.

Finally, we have two stages of fine-tuning to incorporate elastic width. We sample four sub-networks and aggregate their gradients in each update step. The first stage (i.e., $W \in [6, 4]$) takes 25 epochs with an initial learning rate of 0.08 while the second stage (i.e., $W \in [6, 4, 3]$) takes 125 epochs with an initial learning rate of 0.24.