Model Acceleration for Efficient Deep Learning Computing

by

Han Cai

B.Eng, Shanghai Jiao Tong University (2016)
M.Eng, Shanghai Jiao Tong University (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Han Cai Department of Electrical Engineering and Computer Science May 17, 2024
Certified by:	Song Han Associate Professor in Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Model Acceleration for Efficient Deep Learning Computing

by

Han Cai

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Large foundation models play a central role in achieving recent fundamental breakthroughs in artificial intelligence. By simultaneously scaling up the dataset and model size to an unprecedented level, these foundation models demonstrate remarkable performances in many areas such as protein structure prediction, image/video synthesis, code generation, ChatBot, etc. However, their computation and memory costs grow dramatically. It makes deploying these foundation models on real-world applications difficult, especially for resource-constrained edge devices. In addition, their prohibitive training cost also significantly hinders the development of new foundation models and raises concerns about the enormous energy consumption and CO_2 emission. To address these concerns, building effective model acceleration techniques is critical to closing the gap between supply and demand for computing.

This thesis will cover three important aspects of model acceleration. First, we will discuss efficient representation learning, including EfficientViT (a new vision transformer architecture) for high-resolution vision and condition-aware neural networks (a new control module) for conditional image generation. Second, we will present hardware-aware acceleration techniques to create specialized neural networks for different hardware platforms and efficiency constraints. Third, we will introduce TinyTL, a memory-efficient transfer learning technique to enable on-device model customization. Through our design, we can significantly boost deep neural networks' efficiency on hardware without losing accuracy, making them more accessible and reducing their serving cost. For example, our model delivers 48.9× higher throughput on A100 GPU while achieving slightly better zero-shot instance segmentation performance than the state-of-the-art model. For conditional image generation, our approach achieves 52× computational cost reduction without performance degradation.

Thesis supervisor: Song Han
Title: Associate Professor in Electrical Engineering and Computer Science

Acknowledgments

First, I want to thank my PhD advisor, Professor Song Han. I have been very fortunate to receive his guidance for the five years of the PhD. He always provides the best support for us and helps us pursue our research interests. His enthusiasm for impactful research greatly motivated me. His deep connection with the industry made his advice insightful beyond the academic context. I also thank my friends and lab mates at MIT: Hanrui Wang, Zhijian Liu, Yujun Lin, Ji Lin, Haotian Tang, Ligeng Zhu, Zhekai Zhang, Guangxuan Xiao, Muyang Li, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Nicole Stiles, and Tianhao Huang. It was a great pleasure to have you all during my PhD journey.

I also want to thank my advisors during my master's and undergraduate years, Professor Yong Yu and Professor Weinan Zhang. I started to understand what is research and how to do research work, under their supervision. It is my fortune to have them to guide me into the world of research.

I thank my PhD thesis committee members, Professor Vincent Sitzmann and Professor Vijay Janapa Reddi, for their time, help, and advice during my thesis preparation and defense.

I want to thank my mom and dad for their continued support, even though I haven't been home for four years due to various reasons.

Finally, I thank the Qualcomm Innovation Fellowship and Analog Devices Graduate Fellowship for the funding support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large foundation models have revolutionized many AI areas, including natural language processing [1], [2], computer vision [3]–[5], AI for science [6], etc. By scaling up the model size and training these models on web-scale datasets, these foundation models demonstrate astounding few-shot/zero-shot learning abilities in solving complicated tasks. These remarkable performances have driven a surge of interest in using these foundation models in real-world applications, bringing AI to our work and daily lives.

However, these foundation models have prohibitive training and inference costs due to the increased model size and computation cost. For example, a GPT-3 [7] model has 175B parameters. Storing it alone already exceeds the capacity of the current most powerful GPUs (e.g., NVIDIA H100 GPU). It poses big challenges for serving these models on cloud platforms or deploying them on edge devices. In addition, prohibitive training cost also leads to enormous energy consumption and CO_2 emission, raising sustainability concerns about these AI foundation models.

In this dissertation, we aim to investigate model acceleration techniques to improve the efficiency of deep neural networks to tackle this challenge. Our approach accelerates deep neural networks from three aspects. First, we will discuss efficient representation learning, targeting to build efficient building blocks / neural network architectures to extract useful information from the raw data. Second, we will discuss hardware-aware acceleration, aiming to get specialized neural networks for different hardware platforms and efficiency constraints to get the best trade-off between accuracy and hardware efficiency. Third, we will discuss efficient model customization that enables memory-efficient on-device learning to offer customized AI services without sacrificing privacy. We summarize the main content of this thesis as follows:

1.1 Thesis Outline

- **Chapter 2** describes techniques for efficient representation learning. The content is based on [8] and [9]. First, the transformer architecture is the core component of current large foundation models. However, the transformer architecture is bad at handling long sequences since its computational cost grows quadratically as the input sequence length increases. We propose EfficientViT, a new vision transformer architecture for high-resolution vision. It achieves global receptive field and strong capacity with only hardware-efficient operations.

EfficientViT delivers remarkable performance gains over previous models with speedup on diverse hardware platforms. Second, adding control is a critical step to convert image/video generative models to productive tools for humans. We propose condition-aware neural network (CAN), a new method for adding control to image generative models. In parallel to prior conditional control methods, CAN controls the image generation process by dynamically manipulating the weight of the neural network. CAN consistently delivers significant improvements for diffusion transformer models.

- **Chapter 3** presents hardware-aware AutoML techniques to efficiently get specialized deep neural networks for different hardware platforms and efficiency constraints. The content is based on [10] and [11]. Different hardware platforms have different properties (e.g., degree of parallelism, cache size, bandwidth, etc.). Given different target hardware platforms and different efficiency constraints, we need specialized neural networks to achieve the best trade-off between performance and efficiency. However, manually customizing neural networks for each case is unscalable. Thus, we propose hardware-aware AutoML techniques to tackle this challenge. Our approach delivers significant speedup on diverse hardware platforms, including mobile phones, CPU, GPU, FPGA, etc. In addition, our approach has won first place in multiple low-power computer vision challenges.

- **Chapter 4** presents TinyTL [12] for memory-efficient on-device learning. TinyTL freezes the weights while only learns the memory-efficient bias modules, thus no need to store the intermediate activations. To maintain the adaptation capacity, we introduce a new memory-efficient bias module, the lite residual module, to refine the feature extractor by learning small residual feature maps adding only 3.8% memory overhead. Extensive experiments show that TinyTL significantly saves the memory with little accuracy loss compared to fine-tuning the full network.

Chapter 2

Efficient Representation Learning

2.1 Efficient Vision Transformer for High-Resolution Vision

2.1.1 Introduction

High-resolution dense prediction is a fundamental task in computer vision and has broad applications in the real world, including autonomous driving, medical image processing, computational photography, etc. Therefore, deploying state-of-the-art (SOTA) high-resolution dense prediction models on hardware devices can benefit many use cases.

However, there is a large gap between the computational cost required by SOTA high-resolution dense prediction models and the limited resources of hardware devices. It makes using these models in real-world applications impractical. In particular, high-resolution dense prediction models require high-resolution images and strong context information extraction ability to work well [13]–[18]. Therefore, directly porting efficient model architectures from image classification is unsuitable for high-resolution dense prediction.

This work introduces **EfficientViT**, a new family of vision transformer models for efficient high-resolution dense prediction. The core of EfficientViT is a new multi-scale linear attention module that enables the global receptive field and multi-scale learning with hardware-efficient operations. Our module is motivated by prior SOTA high-resolution dense prediction models. They demonstrate that the multi-scale learning [15], [16] and global receptive field [19] are critical in improving models' performances. However, they do not consider hardware efficiency when designing their models, which is essential for real-world applications. For example, SegFormer [19] introduces softmax attention [20] into the backbone to have a global receptive field. However, its computational complexity is quadratic to the input resolution, making it unable to handle high-resolution images efficiently. SegNeXt [21] proposes a multi-branch module with large-kernel convolutions (kernel size up to 21) to enable a large receptive field and multi-scale learning. However, large-kernel convolution requires exceptional support on hardware to achieve good efficiency [22], [23], which is usually unavailable on hardware devices.

Hence, the design principle of our module is to enable these two critical features while avoiding hardware-inefficient operations. Specifically, we propose substituting the inefficient

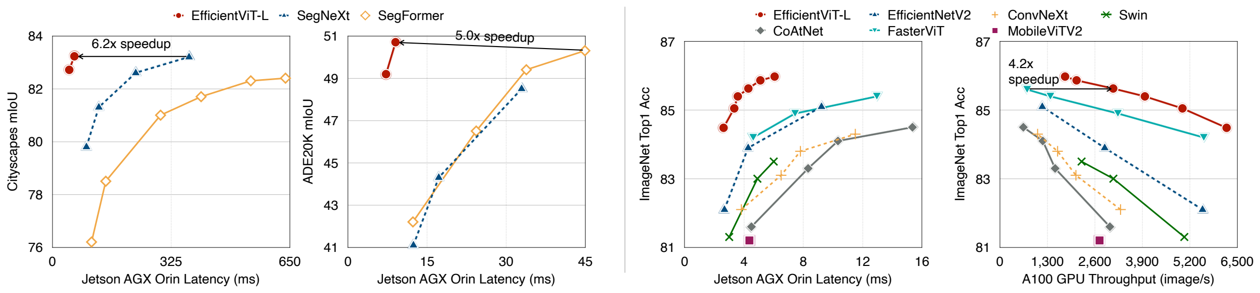


Figure 2.1: **Latency/Throughput vs. Performance.** All performance results are obtained with the single model and single-scale inference. The GPU latency/throughput results are obtained on one edge GPU (Jetson AGX Orin) and one cloud GPU (A100) using TensorRT and fp16. EfficientViT consistently achieves a remarkable boost in speed on diverse hardware platforms while providing the same/higher performances on Cityscapes, ADE20K, and ImageNet than prior segmentation/classification models.

softmax attention with lightweight ReLU linear attention [24] to have the global receptive field. By leveraging the associative property of matrix multiplication, ReLU linear attention can reduce the computational complexity from quadratic to linear while preserving functionality. In addition, it avoids hardware-inefficient operations like softmax, making it more suitable for hardware deployment (Figure 2.4).

However, ReLU linear attention alone has limited capacity due to the lack of local information extraction and multi-scale learning ability. Therefore, we propose to enhance ReLU linear attention with convolution and introduce the multi-scale linear attention module to address the capacity limitation of ReLU linear attention. Specifically, we aggregate nearby tokens with small-kernel convolutions to generate multi-scale tokens. We perform ReLU linear attention on multi-scale tokens (Figure 2.2) to combine the global receptive field with multi-scale learning. We also insert depthwise convolutions into FFN layers to further improve the local feature extraction capacity.

We extensively evaluate EfficientViT on two popular high-resolution dense prediction tasks: semantic segmentation and super-resolution. EfficientViT provides significant performance boosts over prior SOTA high-resolution dense prediction models. More importantly, EfficientViT does not involve hardware-inefficient operations, so our #FLOPs reduction can easily translate to latency reduction on hardware devices (Figure 2.1).

In addition to these conventional high-resolution dense prediction tasks, we apply EfficientViT to Segment Anything [4], an emerging promptable segmentation task that allows zero-shot transfer to many vision tasks. EfficientViT achieves 48.9× acceleration on A100 GPU than SAM-ViT-Huge [4] without performance loss.

2.1.2 Related Work

High-Resolution Dense Prediction. Dense prediction targets producing predictions for each pixel given the input image. It can be viewed as an extension of image classification from per-image prediction to per-pixel predictions. Extensive studies have been done to improve the performance of CNN-based high-resolution dense prediction models [13]–[18].

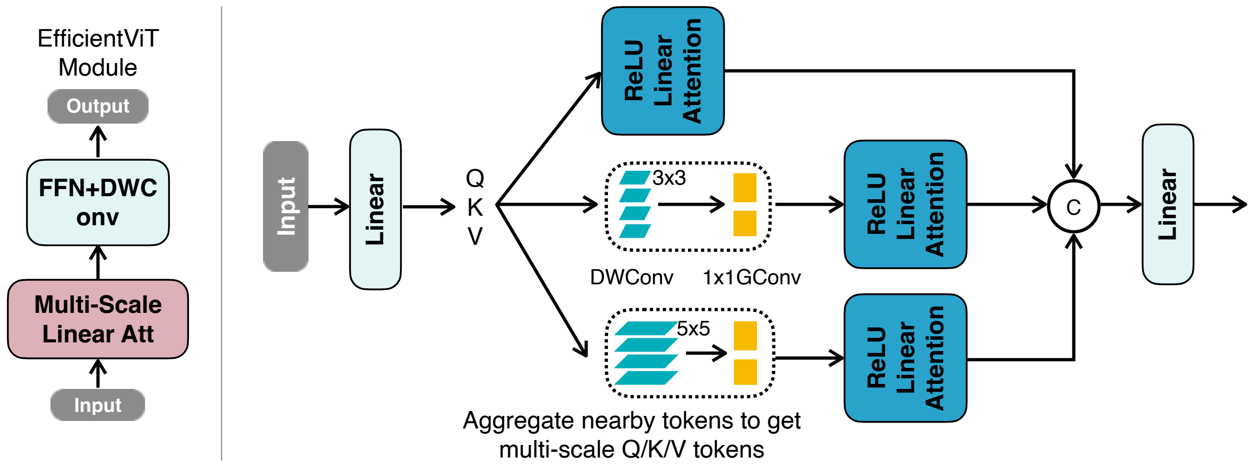


Figure 2.2: **EfficientViT's Building Block (left) and Multi-Scale Linear Attention (right).** *Left*: EfficientViT's building block consists of a multi-scale linear attention module and an FFN with depthwise convolution (FFN+DWConv). Multi-scale linear attention is responsible for capturing context information, while FFN+DWConv captures local information. *Right*: After getting Q/K/V tokens via the linear projection layer, we generate multi-scale tokens by aggregating nearby tokens via lightweight small-kernel convolutions. ReLU linear attention is applied to multi-scale tokens, and the outputs are concatenated and fed to the final linear projection layer for feature fusing.

In addition, there are also some works targeting improving the efficiency of high-resolution dense prediction models [25]–[28]. While these models provide good efficiency, their performances are far behind SOTA high-resolution dense prediction models.

Compared to these works, our models provide a better trade-off between performance and efficiency by enabling a global receptive field and multi-scale learning with lightweight operations.

Efficient Vision Transformer. While ViT provides impressive performances in the high-computation region, it is usually inferior to previous efficient CNNs [11], [29]–[31] when targeting the low-computation region. To close the gap, MobileViT [32] proposes to combine the strength of CNN and ViT by replacing local processing in convolutions with global processing using transformers. MobileFormer [33] proposes to parallelize MobileNet and Transformer with a two-way bridge in between for feature fusing. NASViT [34] proposes to leverage neural architecture search to search for efficient ViT architectures.

However, these models mainly focus on image classification and still rely on softmax attention with quadratic computational complexity, thus unsuitable for high-resolution dense prediction.

Efficient Deep Learning. Our work is also related to efficient deep learning, which aims at improving the efficiency of deep neural networks so that we can deploy them on hardware platforms with limited resources, such as mobile phones and IoT devices. Typical technologies in efficient deep learning include network pruning [35]–[37], quantization [38], efficient model

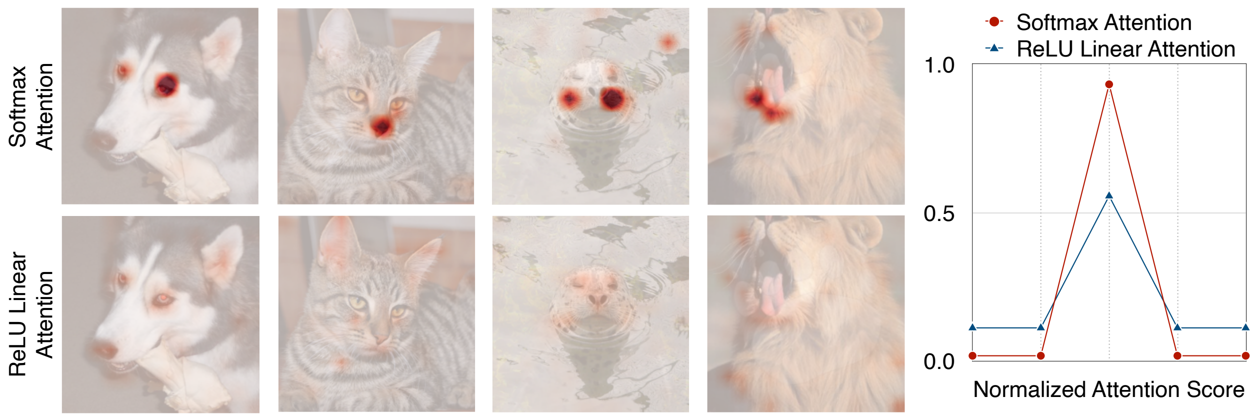


Figure 2.3: **Softmax Attention vs. ReLU Linear Attention.** Unlike softmax attention, ReLU linear attention cannot produce sharp attention distributions due to a lack of the non-linear similarity function. Thus, its local information extraction ability is weaker than the softmax attention.

architecture design [39], [40], and training techniques [12], [41], [42]. In addition to manual designs, many recent works use AutoML techniques [10], [43], [44] to automatically design [11], prune [45] and quantize [46] neural networks.

2.1.3 Method

This section first introduces the multi-scale linear attention module. Unlike prior works, our multi-scale linear attention simultaneously achieves the global receptive field and multi-scale learning with only hardware-efficient operations. Then, based on the multi-scale linear attention, we present a new family of vision transformer models named EfficientViT for high-resolution dense prediction.

Multi-Scale Linear Attention

Our multi-scale linear attention balances two crucial aspects of efficient high-resolution dense prediction, i.e., performance and efficiency. Specifically, the global receptive field and multi-scale learning are essential from the performance perspective. Previous SOTA high-resolution dense prediction models provide strong performances by enabling these features but fail to provide good efficiency. Our module tackles this issue by trading slight capacity loss for significant efficiency improvements.

An illustration of the proposed multi-scale linear attention module is provided in Figure 2.2 (right). In particular, we propose to use ReLU linear attention [24] to enable the global receptive field instead of the heavy softmax attention [20]. While ReLU linear attention [24] and other linear attention modules [47]–[50] have been explored in other domains, it has never been successfully applied to high-resolution dense prediction. To the best of our knowledge, EfficientViT is the first work demonstrating ReLU linear attention's effectiveness in high-resolution dense prediction. In addition, our work introduces novel designs to address its capacity limitation.

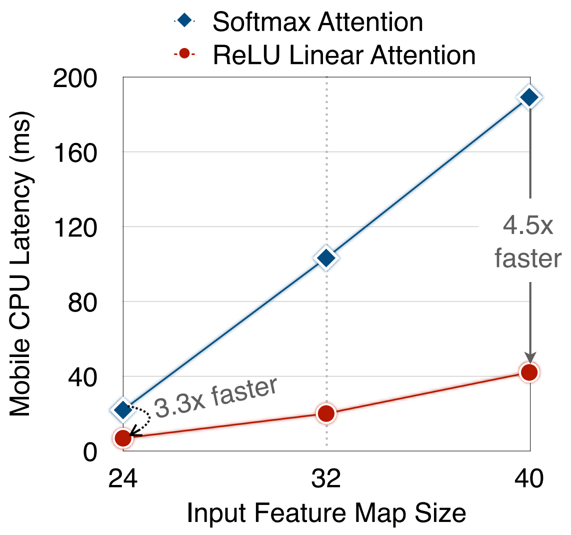


Figure 2.4: **Latency Comparison Between Softmax Attention and ReLU Linear Attention.** ReLU linear attention is 3.3-4.5× faster than softmax attention with similar computation, thanks to removing hardware-unfriendly operations (e.g., softmax). Latency is measured on the Qualcomm Snapdragon 855 CPU with TensorFlow-Lite, batch size 1, and fp32.

Enable Global Receptive Field with ReLU Linear Attention. Given input $x \in \mathbb{R}^{N \times f}$, the generalized form of softmax attention can be written as:

$$O_i = \sum_{j=1}^{N} \frac{Sim(Q_i, K_j)}{\sum_{j=1}^{N} Sim(Q_i, K_j)} V_j, \tag{2.1}$$

where $Q = xW_Q$, $K = xW_K$, $V = xW_V$ and $W_Q/W_K/W_V \in \mathbb{R}^{f \times d}$ is the learnable linear projection matrix. O_i represents the i-th row of matrix O. $Sim(\cdot, \cdot)$ is the similarity function. When using the similarity function $Sim(Q, K) = \exp(\frac{QK^T}{\sqrt{d}})$, Eq. (2.1) becomes the original softmax attention [20].

Apart from $\exp(\frac{QK^T}{\sqrt{d}})$, we can use other similarity functions. In this work, we use ReLU linear attention [24] to achieve both the global receptive field and linear computational complexity. In ReLU linear attention, the similarity function is defined as

$$Sim(Q, K) = \text{ReLU}(Q)\text{ReLU}(K)^T. \tag{2.2}$$

With $Sim(Q, K) = \text{ReLU}(Q)\text{ReLU}(K)^T$, Eq. (2.1) can be rewritten as:

$$O_i = \sum_{j=1}^{N} \frac{\text{ReLU}(Q_i)\text{ReLU}(K_j)^T}{\sum_{j=1}^{N} \text{ReLU}(Q_i)\text{ReLU}(K_j)^T} V_j = \frac{\sum_{j=1}^{N}(\text{ReLU}(Q_i)\text{ReLU}(K_j)^T)V_j}{\text{ReLU}(Q_i)\sum_{j=1}^{N}\text{ReLU}(K_j)^T}.$$

Then, we can leverage the associative property of matrix multiplication to reduce the computational complexity and memory footprint from quadratic to linear without changing

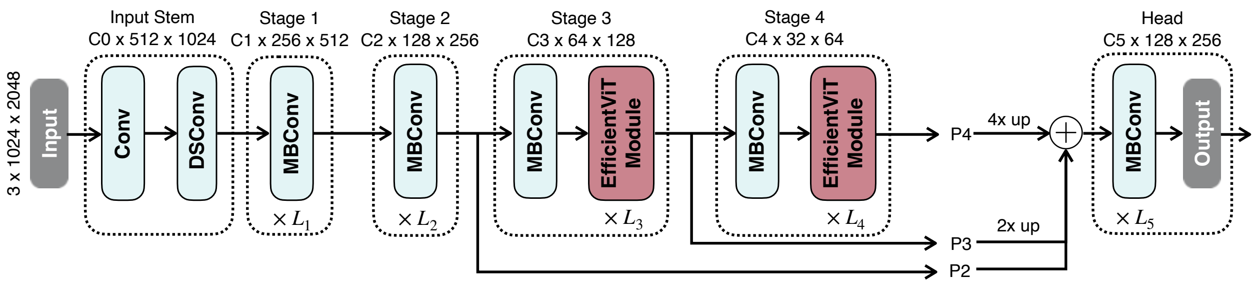


Figure 2.5: **Macro Architecture of EfficientViT.** We adopt the standard backbone-head/encoder-decoder design. We insert our EfficientViT modules in Stages 3 and 4 in the backbone. Following the common practice, we feed the features from the last three stages (P2, P3, and P4) to the head. We use addition to fuse these features for simplicity and efficiency. We adopt a simple head design that consists of several MBConv blocks and output layers.

its functionality:

$$O_i = \frac{\sum_{j=1}^{N} [\text{ReLU}(Q_i)\text{ReLU}(K_j)^T]V_j}{\text{ReLU}(Q_i)\sum_{j=1}^{N}\text{ReLU}(K_j)^T} = \frac{\sum_{j=1}^{N}\text{ReLU}(Q_i)[(\text{ReLU}(K_j)^T V_j)]}{\text{ReLU}(Q_i)\sum_{j=1}^{N}\text{ReLU}(K_j)^T}$$
$$= \frac{\text{ReLU}(Q_i)(\sum_{j=1}^{N}\text{ReLU}(K_j)^T V_j)}{\text{ReLU}(Q_i)(\sum_{j=1}^{N}\text{ReLU}(K_j)^T)}. \quad (2.3)$$

As demonstrated in Eq. (2.3), we only need to compute $(\sum_{j=1}^{N}\text{ReLU}(K_j)^T V_j) \in \mathbb{R}^{d\times d}$ and $(\sum_{j=1}^{N}\text{ReLU}(K_j)^T) \in \mathbb{R}^{d\times 1}$ once, then can reuse them for each query, thereby only requires $\mathcal{O}(N)$ computational cost and $\mathcal{O}(N)$ memory.

Another key merit of ReLU linear attention is that it does not involve hardware-unfriendly operations like softmax, making it more efficient on hardware. For example, Figure 2.4 shows the latency comparison between softmax attention and ReLU linear attention. With similar computation, ReLU linear attention is significantly faster than softmax attention on the mobile CPU.

Address ReLU Linear Attention's Limitations. Although ReLU linear attention is superior to softmax attention in terms of computational complexity and hardware latency, ReLU linear attention has limitations. Figure 2.3 visualizes the attention maps of softmax attention and ReLU linear attention. Because of the lack of the non-linear similarity function, ReLU linear attention cannot generate concentrated attention maps, making it weak at capturing local information.

To mitigate its limitation, we propose to enhance ReLU linear attention with convolution. Specifically, we insert a depthwise convolution in each FFN layer. An overview of the resulting building block is illustrated in Figure 2.2 (left), where the ReLU linear attention captures context information and the FFN+DWConv captures local information.

Furthermore, we propose to aggregate the information from nearby Q/K/V tokens to get multi-scale tokens to enhance the multi-scale learning ability of ReLU linear attention. This information aggregation process is independent for each Q, K, and V in each head. We only use small-kernel depthwise-separable convolutions [39] for information aggregation to avoid hurting hardware efficiency. In the practical implementation, independently executing these aggregation operations is inefficient on GPU. Therefore, we take advantage of the group convolution to reduce the number of total operations. Specifically, all DWConvs are fused into a single DWConv while all 1x1 Convs are combined into a single 1x1 group convolution (Figure 2.2 right) where the number of groups is $3 \times$ #heads and the number of channels in each group is d. After getting multi-scale tokens, we perform ReLU linear attention upon them to extract multi-scale global features. Finally, we concatenate the features along the head dimension and feed them to the final linear projection layer to fuse the features.

EfficientViT Architecture

We build a new family of vision transformer models based on the proposed multi-scale linear attention module. The core building block (denoted as 'EfficientViT Module') is illustrated in Figure 2.2 (left). The macro architecture of EfficientViT is demonstrated in Figure 2.5. We use the standard backbone-head/encoder-decoder architecture design.

- **Backbone.** The backbone of EfficientViT also follows the standard design, which consists of the input stem and four stages with gradually decreased feature map size and gradually increased channel number. We insert the EfficientViT module in Stages 3 and 4. For downsampling, we use an MBConv with stride 2.

- **Head.** P2, P3, and P4 denote the outputs of Stages 2, 3, and 4, forming a pyramid of feature maps. For simplicity and efficiency, we use 1x1 convolution and standard upsampling operation (e.g., bilinear/bicubic upsampling) to match their spatial and channel size and fuse them via addition. Since our backbone already has a strong context information extraction capacity, we adopt a simple head design that comprises several MBConv blocks and the output layers (i.e., prediction and upsample). In the experiments, we empirically find this simple head design is sufficient for achieving SOTA performances.

  In addition to dense prediction, our model can be applied to other vision tasks, such as image classification, by combining the backbone with task-specific heads.

Following the same macro architecture, we design a series of models with different sizes to satisfy various efficiency constraints. We name these models EfficientViT-B0, EfficientViT-B1, EfficientViT-B2, and EfficientViT-B3, respectively. In addition, we designed the EfficientViT-L series for the cloud platforms. Detailed configurations of these models are provided in our official GitHub repository[1].

2.1.4 Experiments

Setups

Datasets. We evaluate the effectiveness of EfficientViT on three representative high-resolution dense prediction tasks, including semantic segmentation, super-resolution, and Segment Anything.

[1] https://github.com/mit-han-lab/efficientvit

Components		mIoU ↑	Params ↓	MACs ↓
Multi-scale	Global att.			
		68.1	0.7M	4.4G
✓		72.3	0.7M	4.4G
	✓	72.2	0.7M	4.4G
✓	✓	**74.5**	0.7M	4.4G

Table 2.1: **Ablation Study.** The mIoU and MACs are measured on Cityscapes with 1024x2048 input resolution. We rescale the width of the models so that they have the same MACs. Multi-scale learning and the global receptive field are essential for obtaining good semantic segmentation performance.

For semantic segmentation, we use two popular benchmark datasets: Cityscapes [56] and ADE20K [57]. In addition, we evaluate EfficientViT under two settings for super-resolution: lightweight super-resolution (SR) and high-resolution SR. We train models on DIV2K [58] for lightweight SR and test on BSD100 [59]. For high-resolution SR, we train models on the first 3000 training images of FFHQ [60] and test on the first 500 validation images of FFHQ[2].

Apart from dense prediction, we also study the effectiveness of EfficientViT for image classification using the ImageNet dataset [61].

Latency Measurement. We measure the mobile latency on Qualcomm Snapdragon 8Gen1 CPU with Tensorflow-Lite[3], batch size 1 and fp32. We use TensorRT[4] and fp16 to measure the latency on edge GPU and cloud GPU. The data transfer time is included in the reported latency/throughput results.

Implementation Details. We implement our models using Pytorch [62] and train them on GPUs. We use the AdamW optimizer with cosine learning rate decay for training our models. For multi-scale linear attention, we use a two-branch design for the best trade-off between performance and efficiency, where 5x5 nearby tokens are aggregated to generate multi-scale tokens.

For semantic segmentation experiments, we use the mean Intersection over Union (mIoU) as our evaluation metric. The backbone is initialized with weights pretrained on ImageNet and the head is initialized randomly, following the common practice.

For super-resolution, we use PSNR and SSIM on the Y channel as the evaluation metrics, same as previous work [63]. The models are trained with random initialization.

Ablation Study

Effectiveness of EfficientViT Module. We conduct ablation study experiments on Cityscapes to study the effectiveness of two key design components of our EfficientViT

[2]https://rb.gy/7je1a
[3]https://www.tensorflow.org/lite
[4]https://docs.nvidia.com/deeplearning/tensorrt/

Models	Top1 Acc ↑	Top5 Acc ↑	Params ↓	MACs ↓	Latency ↓ Nano(bs1)	Latency ↓ Orin(bs1)	Throughput ↑ A100 (image/s)
CoAtNet-0 [51]	81.6	-	25M	4.2G	95.8ms	4.5ms	3011
ConvNeXt-T [52]	82.1	-	29M	4.5G	87.9ms	3.8ms	3303
EfficientViT-B2 (r256)	82.7	96.1	24M	2.1G	**58.5ms**	**2.8ms**	**5325**
Swin-B [53]	83.5	-	88M	15G	240ms	6.0ms	2236
CoAtNet-1 [51]	83.3	-	42M	8.4G	171ms	8.3ms	1512
ConvNeXt-S [52]	83.1	-	50M	8.7G	146ms	6.5ms	2081
EfficientViT-B3 (r224)	83.5	96.4	49M	4.0G	**101ms**	**4.4ms**	**3797**
CoAtNet-2 [51]	84.1	-	75M	16G	254ms	10.3ms	1174
ConvNeXt-B [52]	83.8	-	89M	15G	211ms	7.8ms	1579
EfficientViT-B3 (r288)	84.2	96.7	49M	6.5G	**141ms**	**5.6ms**	**2372**
CoAtNet-3 [51]	84.5	-	168M	35G	-	15.4ms	642
ConvNeXt-L [52]	84.3	-	198M	34G	-	11.5ms	1032
EfficientNetV2-S [54]	83.9	-	22M	8.8G	-	4.3ms	2869
EfficientViT-L1 (r224)	84.5	96.9	53M	5.3G	-	**2.6ms**	**6207**
EfficientNetV2-M [54]	85.2	-	54M	25G	-	9.2ms	1160
FasterViT-4 [55]	85.4	97.3	425M	37G	-	13.0ms	1382
EfficientViT-L2 (r288)	85.6	97.4	64M	11G	-	**4.3ms**	**3102**
FasterViT-6 [55]	85.8	97.4	1360M	142G	-	-	594
EfficientNetV2-L [54]	85.7	-	120M	53G	-	-	696
EfficientViT-L2 (r384)	86.0	97.5	64M	20G	-	-	**1784**

Table 2.2: **Backbone Performance on ImageNet Classification.** 'r224' means the input resolution is 224x224. 'bs1' represents that the latency is measured with batch size 1.

module, i.e., multi-scale learning and global attention. To eliminate the impact of pre-training, we train all models from random initialization. In addition, we rescale the width of the models so that they have the same #MACs. The results are summarized in Table 2.1. We can see that removing either global attention or multi-scale learning will significantly hurt the performances. It shows that all of them are essential for achieving a better trade-off between performance and efficiency.

Backbone Performance on ImageNet. To understand the effectiveness of EfficientViT's backbone in image classification, we train our models on ImageNet following the standard training strategy. We summarize the results and compare our models with SOTA image classification models in Table 2.2.

Though EfficientViT is designed for high-resolution dense prediction, it achieves highly competitive performances on ImageNet classification. In particular, EfficientViT-L2-r384 obtains 86.0 top1 accuracy on ImageNet, providing +0.3 accuracy gain over EfficientNetV2-L and 2.6x speedup on A100 GPU.

Semantic Segmentation

Cityscapes. Table 2.3 reports the comparison between EfficientViT and SOTA semantic segmentation models on Cityscapes. EfficientViT achieves remarkable efficiency improvements over prior SOTA semantic segmentation models without sacrificing performances. Specifically,

Models	mIoU ↑	Params ↓	MACs ↓	Latency ↓ Nano(bs1)	Latency ↓ Orin(bs1)	Throughput ↑ A100(image/s)
DeepLabV3plus-Mbv2 [64]	75.2	15M	555G	-	83.5ms	102
EfficientViT-B0	75.7	0.7M	4.4G	**0.28s**	**9.9ms**	**263**
SegFormer-B1 [19]	78.5	14M	244G	5.6s	146ms	49
SegNeXt-T [21]	79.8	4.3M	51G	2.2s	93.2ms	95
EfficientViT-B1	80.5	4.8M	25G	**0.82s**	**24.3ms**	**175**
SegFormer-B3 [19]	81.7	47M	963G	-	407ms	18
SegNeXt-S [21]	81.3	14M	125G	3.4s	127ms	70
EfficientViT-B2	82.1	15M	74G	**1.7s**	**46.5ms**	**112**
SegFormer-B5 [19]	82.4	85M	1460G	-	638ms	12
SegNeXt-B [21]	82.6	28M	276G	-	228ms	41
EfficientViT-B3	83.0	40M	179G	-	81.8ms	70
EfficientViT-L1	82.7	40M	282G	-	**45.9ms**	**122**
SegNeXt-L [21]	83.2	49M	578G	-	374ms	26
EfficientViT-L2	83.2	53M	396G	-	**60.0ms**	**102**

Table 2.3: **Comparison with SOTA Semantic Segmentation Models on Cityscapes.** The input resolution is 1024x2048 for all models. Models with similar mIoU are grouped for efficiency comparison.

compared with SegFormer, EfficientViT obtains up to 13x #MACs saving and up to 8.8x latency reduction on the edge GPU (Jetson AGX Orin) with higher mIoU. Compared with SegNeXt, EfficientViT provides up to 2.0x MACs reduction and 3.8x speedup on the edge GPU (Jetson AGX Orin) while maintaining higher mIoU. On A100 GPU, EfficientViT delivers up to 3.9x higher throughput than SegNeXt and 10.2x higher throughput than SegFormer while achieving the same or higher mIoU. Having similar computational cost, EfficientViT also yields significant performance gains over previous SOTA models. For example, EfficientViT-B3 delivers +4.5 mIoU gain over SegFormer-B1 with lower MACs.

In addition to the quantitative results, we visualize EfficientViT and the baseline models qualitatively on Cityscapes. The results are shown in Figure 2.6. We can find that EfficientViT can better recognize boundaries and small objects than the baseline models while achieving lower latency on GPU.

ADE20K. Table 2.4 summarizes the comparison between EfficientViT and SOTA semantic segmentation models on ADE20K. Like Cityscapes, we can see that EfficientViT also achieves significant efficiency improvements on ADE20K. For example, with +0.6 mIoU gain, EfficientViT-B1 provides 5.2x MACs reduction and up to 3.5x GPU latency reduction than SegFormer-B1. With +1.6 mIoU gain, EfficientViT-B2 requires 1.8x fewer computational costs and runs 2.4x faster on Jetson AGX Orin GPU than SegNeXt-S.

Models	mIoU ↑	Params ↓	MACs ↓	Latency ↓ Nano(bs1)	Latency ↓ Orin(bs1)	Throughput ↑ A100(image/s)
SegFormer-B1 [19]	42.2	14M	16G	389ms	12.3ms	542
SegNeXt-T [21]	41.1	4.3M	6.6G	281ms	12.4ms	842
EfficientViT-B1	42.8	4.8M	3.1G	**110ms**	**4.0ms**	**1142**
SegNeXt-S [21]	44.3	14M	16G	428ms	17.2ms	592
EfficientViT-B2	45.9	15M	9.1G	**212ms**	**7.3ms**	**846**
Mask2Former [65]	47.7	47M	74G	-	-	-
MaskFormer [66]	46.7	42M	55G	-	-	-
SegFormer-B2 [19]	46.5	28M	62G	920ms	24.3ms	345
SegNeXt-B [21]	48.5	28M	35G	806ms	32.9ms	347
EfficientViT-B3	49.0	39M	22G	411ms	12.5ms	555
EfficientViT-L1	49.2	40M	36G	-	**7.2ms**	**947**
SegFormer-B4 [19]	50.3	64M	96G	-	44.9ms	212
EfficientViT-L2	50.7	51M	45G	-	**9.0ms**	**758**

Table 2.4: **Comparison with SOTA Semantic Segmentation Models on ADE20K.** The shorter side of the image is resized to 512, following the common practice.

Model	FFHQ (512x512 → 1024x1024)				BSD100 (160x240 → 320x480)			
	PSNR ↑	SSIM ↑	A100(bs1) ↓	Speedup ↑	PSNR ↑	SSIM ↑	A100(bs1) ↓	Speedup ↑
Restormer [67]	43.43	0.9806	92.0ms	1x	32.31	**0.9021**	15.1ms	1x
SwinIR [63]	43.49	0.9807	61.2ms	1.5x	32.31	0.9012	9.7ms	1.6x
VapSR [68]	-	-	-	-	32.27	0.9011	4.8ms	3.1x
BSRN [69]	-	-	-	-	32.24	0.9006	4.5ms	3.4x
EfficientViT w0.75	43.54	0.9809	**14.3ms**	**6.4x**	32.31	0.9016	**2.8ms**	**5.4x**
EfficientViT	**43.58**	**0.9810**	17.8ms	5.2x	**32.33**	0.9019	3.2ms	4.7x

Table 2.5: **Comparison with SOTA super-resolution models.**

Super-Resolution

Table 2.5 presents the comparison of EfficientViT with SOTA ViT-based SR methods (SwinIR [63] and Restormer [67]) and SOTA CNN-based SR methods (VapSR [68] and BSRN [69]). EfficientViT provides a better latency-performance trade-off than all compared methods.

On lightweight SR, EfficientViT provides up to 0.09dB gain in PSNR on BSD100 while maintaining the same or lower GPU latency compared with SOTA CNN-based SR methods. Compared with SOTA ViT-based SR methods, EfficientViT provides up to 5.4× speedup on GPU and maintains the same PSNR on BSD100.

On high-resolution SR, the advantage of EfficientViT over previous ViT-based SR methods becomes more significant. Compared with Restormer, EfficientViT achieves up to 6.4× speedup on GPU and provides 0.11dB gain in PSNR on FFHQ.

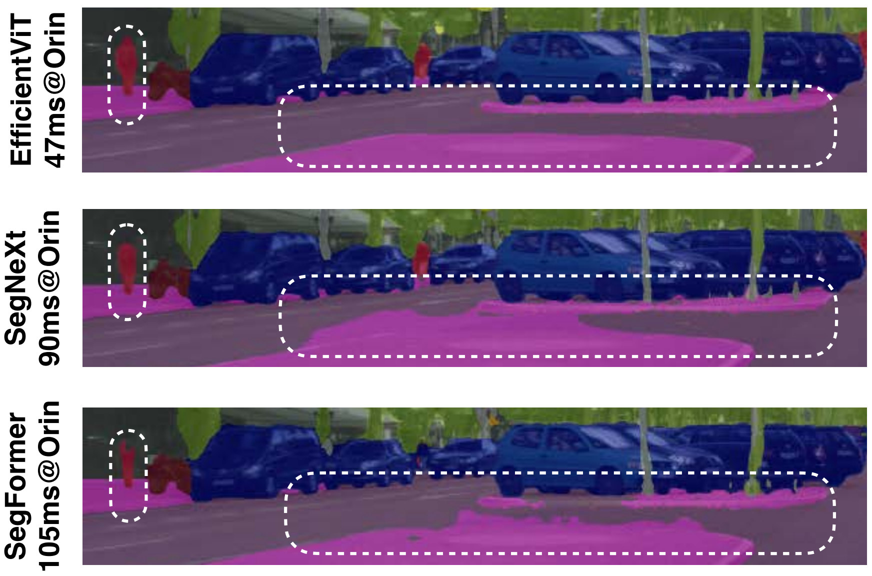


Figure 2.6: **Qualitative results on Cityscapes.**

	Params ↓	MACs ↓	Throughput ↑	COCO				LVIS			
			A100(image/s)	mAP	AP^S	AP^M	AP^L	mAP	AP^S	AP^M	AP^L
SAM-ViT-H [4]	641M	2973G	11	46.5	30.8	51.0	61.7	44.2	31.8	57.1	65.3
EfficientViT-SAM-L0	35M	35G	762	45.7	28.2	49.5	63.4	41.8	28.8	53.4	64.7
EfficientViT-SAM-L1	48M	49G	638	46.2	28.7	50.4	64.0	42.1	29.1	54.3	65.0
EfficientViT-SAM-L2	61M	69G	538	46.6	28.9	50.8	64.2	42.7	29.4	55.1	65.5
EfficientViT-SAM-XL0	117M	185G	278	47.5	30.0	51.5	64.6	43.9	31.2	56.2	65.9
EfficientViT-SAM-XL1	203M	322G	182	47.8	30.5	51.8	64.7	44.4	31.6	57.0	66.4

Table 2.6: **Zero-Shot Instance Segmentation Results, Prompted with ViTDet Boxes.** Throughput is profiled on A100 GPU with TensorRT and fp16, including the image encoder and SAM head.

Segment Anything

We build EfficientViT-SAM [70], a new family of accelerated segment anything models, by leveraging EfficientViT to replace SAM's image encoder. Meanwhile, we retain SAM's lightweight prompt encoder and mask decoder. The training process consists of two phases. First, we train the image encoder of EfficientViT-SAM using SAM's image encoder as the teacher. Second, we train EfficientViT-SAM end-to-end using the whole SA-1B dataset [4].

We thoroughly test EfficientViT-SAM on various zero-shot benchmarks to verify its effectiveness. Table 2.6 demonstrates the zero-shot instance segmentation results on COCO [71] and LVIS [72], prompted with the predicted bounding boxes from ViTDet [73]. EfficientViT-SAM provides superior performance/efficiency compared with SAM-ViT-H [4]. In particular, EfficientViT-SAM-XL1 outperforms SAM-ViT-H on COCO and LVIS while having 16.5× higher throughput on A100 GPU.

Figure 2.7 shows the comparison between EfficientViT-SAM and prior SAM models.

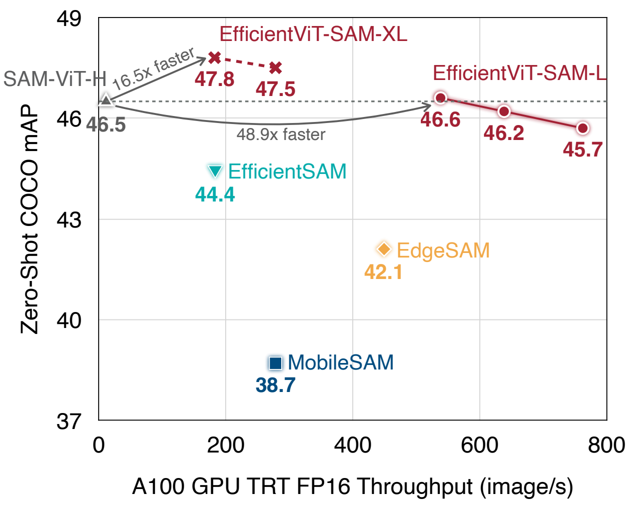


Figure 2.7: **Throughput vs. COCO Zero-Shot Instance Segmentation mAP.** EfficientViT-SAM is the first accelerated SAM model that matches/outperforms SAM-ViT-H's [4] zero-shot performance, delivering the SOTA performance-efficiency trade-off.

	COCO			LVIS		
	1 click	3 click	5 click	1 click	3 click	5 click
SAM-ViT-H [4]	58.4	69.6	71.4	59.2	66.0	66.8
EfficientViT-SAM-XL1	59.8	71.3	75.3	56.6	67.0	71.7

Table 2.7: **Zero-Shot Point-Prompted Segmentation Results.**

EfficientViT-SAM is the first accelerated SAM model that matches/outperforms SAM-ViT-H's [4] zero-shot performance, delivering the SOTA performance-efficiency trade-off.

Apart from box-prompted instance segmentation, we also evaluate EfficientViT-SAM on point-prompted segmentation. The results are summarized in Table 2.7. EfficientViT-SAM-XL1 outperforms SAM-ViT-H in most cases, especially when more points are given. On LVIS, when given a single point, we find SAM-ViT-H performs better than EfficientViT-SAM-XL1. This might be because we do not have the interactive segmentation setup during the end-to-end training phase. Further investigation is needed to improve the performance of the single-point setting.

2.1.5 Conclusion

In this section, we studied efficient architecture design for high-resolution dense prediction. We introduced a lightweight multi-scale attention module that simultaneously achieves a global receptive field, and multi-scale learning with lightweight and hardware-efficient operations, thus providing significant speedup on diverse hardware devices without performance loss than SOTA high-resolution dense prediction models. For future work, we will explore applying EfficientViT to other vision tasks and further scaling up our EfficientViT models.

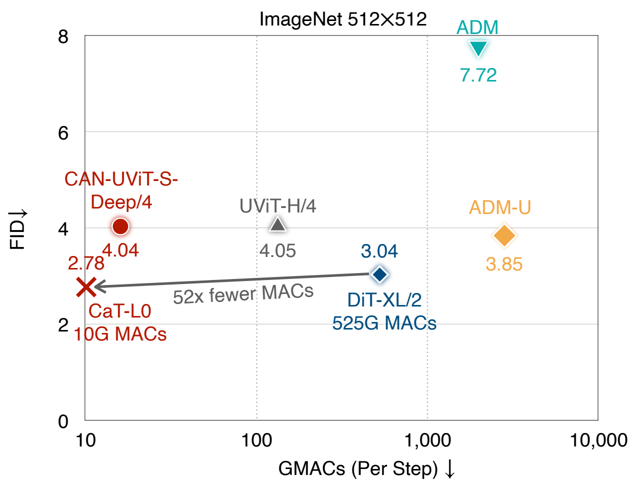


Figure 2.8: **Comparing CAN Models and Prior Image Generative Models on ImageNet 512×512.** With the new conditional control method, we significantly improve the performance of controlled image generative models. Combining CAN and EfficientViT [8], our CaT model provides 52× MACs reduction per sampling step than DiT-XL/2 [83] without performance loss.

2.2 Efficient Control Module for Conditional Image Generation

2.2.1 Introduction

Large-scale image [3], [74]–[76] and video generative models [5], [77] have demonstrated astounding capacity in synthesizing photorealistic images and videos. To convert these models into productive tools for humans, a critical step is adding control. Instead of letting the model randomly generate data samples, we want the generative model to follow our instructions (e.g., class label, text, pose) [78].

Extensive studies have been conducted to achieve this goal. For example, in GANs [60], [79], a widespread solution is to use adaptive normalization [80], [81] that dynamically scales and shifts the intermediate feature maps according to the input condition. In addition, another widely used technique is to use cross-attention [3] or self-attention [82] to fuse the condition feature with the image feature. Though differing in the used operations, these methods share the same underlying mechanism, i.e., adding control by feature space manipulation. Meanwhile, the neural network weight (convolution/linear layers) remains the same for different conditions.

This work aims to answer the following questions: *i) Can we control image generative models by manipulating their weight? ii) Can controlled image generative models benefit from this new conditional control method?*

To this end, we introduce **Condition-Aware Neural Network (CAN)**, a new conditional control method based on weight space manipulation. Differentiating from a regular neural network, CAN introduces an additional weight generation module (Figure 2.9). The input to this module is the condition embedding, which consists of the user instruction

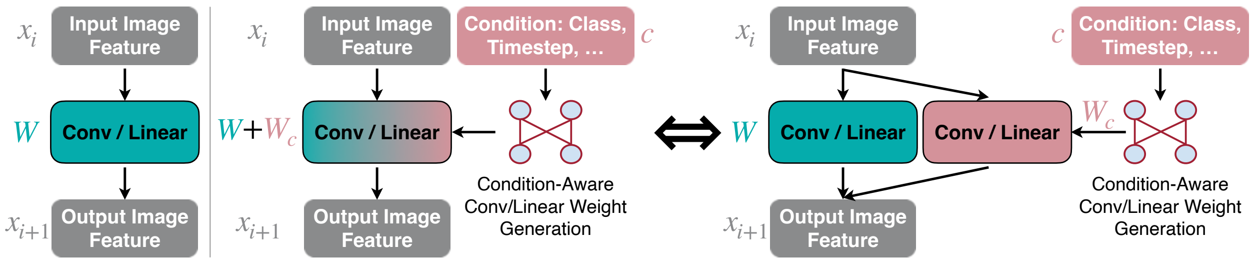


Figure 2.9: **Illustration of Condition-Aware Neural Network.** *Left:* A regular neural network with static convolution/linear layers. *Right:* A condition-aware neural network and its equivalent form.

(e.g., class label) and the timestep for diffusion models [84]. The module's output is the conditional weight used to adapt the static weight of the convolution/linear layer. We conduct extensive ablation study experiments investigating the practical use of CAN on diffusion transformers. Our study reveals two critical insights for CAN. First, rather than making all layers condition-aware, we find carefully choosing a subset of modules to be condition-aware (Figure 2.10) is beneficial for both efficiency and performance (Table 2.8). Second, we find directly generating the conditional weight is much more effective than adaptively merging a set of base static layers [85] for conditional control (Figure 2.11).

We evaluate CAN on two representative diffusion transformer models, including DiT [83], and UViT [82]. CAN achieves significant performance boosts for all these diffusion transformer models while incurring negligible computational cost increase (Figure 2.14). We also find that CAN alone provides effective conditional control for image generative models, delivering lower FID and higher CLIP scores than prior conditional control methods (Table 2.10). Apart from applying CAN to existing diffusion transformer models, we further build a new family of diffusion transformer models called **CaT** by marrying CAN and EfficientViT [8] (Figure 2.13).

2.2.2 Related Work

Controlled Image Generation. Controlled image generation requires the models to incorporate the condition information into the computation process to generate related images. Various techniques have been developed in the community for controlled image generation. One typical example is adaptive normalization [81] that regresses scale and shift parameters from the condition information and applies the feature-wise affine transformation to influence the output. Apart from adaptive normalization, another typical approach is to treat condition information as tokens and use either cross-attention [3] or self-attention [82] to fuse the condition information. ControlNet [78] is another representative technique that uses feature-wise addition to add extra control to pre-trained text-to-image diffusion models. In parallel to these techniques, this work explores another mechanism for adding conditional control to image generative models, i.e., making the weight of neural network layers (conv/linear) condition-aware.

Dynamic Neural Network. Our work can be viewed as a new type of dynamic neural network. Apart from adding conditional control explored in this work, dynamically adapting

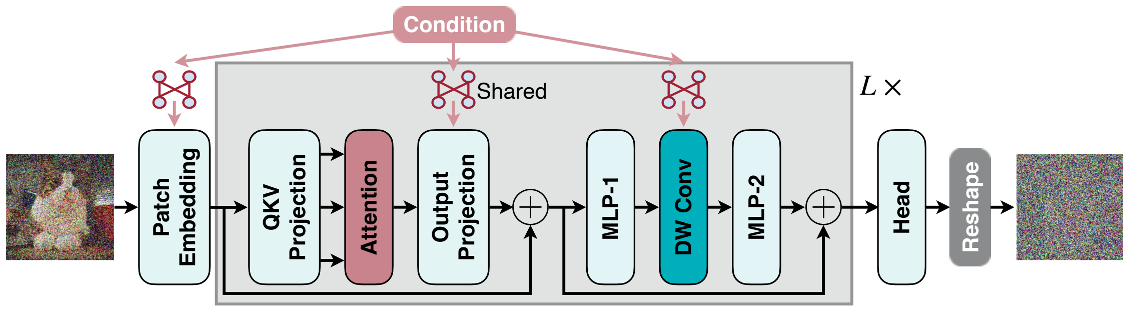


Figure 2.10: **Overview of Applying CAN to Diffusion Transformer.** The patch embedding layer, the output projection layers in self-attention, and the depthwise convolution (DW Conv) layers are condition-aware. The other layers are static. All output projection layers share the same conditional weight while still having their own static weights.

the neural network can be applied to many deep learning applications. For example, CondConv [85] proposes to dynamically combine a set of base convolution kernels based on the input image feature to increase the model capacity. Similarly, the mixture-of-expert [86] technique uses a gating network to route the input to different experts dynamically. For efficient deployment, once-for-all network [87] and slimmable neural network [88] dynamically adjust the neural network architecture according to the given efficiency constraint to achieve better tradeoff between efficiency and accuracy.

Weight Generating Networks. Our conditional weight generation module can be viewed as a new kind of weight generation network specially designed for adding conditional control to generative models. There are some prior works exploiting weight generation networks in other scenarios. For example, [89] proposes to use a small network to generate weights for a larger network. These weights are the same for every example in the dataset for better parameter efficiency. Additionally, weight generation networks have been applied to neural architecture search to predict the weight of a neural network given its architecture [90] to reduce the training and search cost [91] of neural architecture search.

Efficient Deep Learning Computing. Our work is also connected to efficient deep learning computing [42], [92] that aims to improve the efficiency of deep learning models to make them friendly for deployment on hardware. State-of-the-art image generative models [3], [74], [76], [93] have enormous computation and memory costs, which makes it challenging to deploy them on resource-constrained edge devices while maintaining high quality. Our work can improve the efficiency of the controlled generative models by delivering the same performance with fewer diffusion steps and lower-cost models. For future work, we will explore combining our work and efficient deep learning computing techniques [87], [94] to futher boost efficiency.

2.2.3 Method

Condition-Aware Neural Network

The image generation process can be viewed as a mapping from the source domain (noise or noisy image) to the target domain (real image). For controlled image generation, the target data distribution is different given different conditions (e.g., cat images' data distribution vs. castle images' data distribution). In addition, the input data distribution is also different for diffusion models [84] at different timesteps. Despite these differences, prior models use the same static convolution/linear layers for all cases, limiting the overall performance due to negative transfer between different sub-tasks [95]. To alleviate this issue, one possible solution is to have an expert model [86] for each sub-task. However, this approach is infeasible for practical use because of the enormous cost. Our condition-aware neural network (CAN) tackles this issue by enabling the neural network to adjust its weight dynamically according to the given condition, instead of explicitly having the expert models.

Figure 2.9 demonstrates the general idea of CAN. The key difference from a regular neural network is that CAN has an extra conditional weight generation module. This module takes the condition embedding c as the input and outputs the conditional weight W_c. In addition to the conditional weight W_c, each layer has the static weight W. During training and inference, W_c and W are fused into a single kernel call by summing the weight values. This is equivalent to applying W_c and W independently on the input image feature and then adding their outputs.

Practical Design

Which Modules to be Condition-Aware? Theoretically, we can make all layers in the neural network condition-aware. However, in practice, this might not be a good design. First, from the performance perspective, having too many condition-aware layers might make the model optimization challenging. Second, from the efficiency perspective, while the computational overhead of generating the conditional weight for all layers is negligible[15], it will incur a significant parameter overhead. For example, let's denote the dimension of the condition embedding as d (e.g., 384, 512, 1024, etc) and the model's static parameter size as #params. Using a single linear layer to map from the condition embedding to the conditional weight requires #params $\times d$ parameters, which is impractical for real-world use. In this work, we carefully choose a subset of modules to apply CAN to solve this issue.

An overview of applying CAN to diffusion transformer [82], [83] is provided in Figure 2.10. Depthwise convolution [96] has a much smaller parameter size than regular convolution, making it a low-cost candidate to be condition-aware. Therefore, we add a depthwise convolution in the middle of FFN following the previous design [8]. We conduct ablation study experiments on ImageNet 256×256 using UViT-S/2 [82] to determine the set of modules to be condition-aware. *All the models, including the baseline model, have the same architecture. The only distinction is the set of condition-aware modules is different.*

[51]It is because the sequence length (or spatial size) of the condition embedding is much smaller than the image feature.

ImageNet 256×256, UViT-S/2		
Models	FID ↓	CLIP Score ↑
1. Baseline (Static Conv/Linear)	28.32	30.09
Making Modules Condition-Aware:		
2. DW Conv	11.18	31.54
3. + Patch Embedding	10.23	31.61
4. or + Head (✗)	12.29	31.40
5. + Output Projection	8.82	31.74
6. or + QKV Projection (✗)	9.71	31.66
7. or + MLP (✗)	10.06	31.62

Table 2.8: **Ablation Study on Making Which Modules Condition-Aware.**

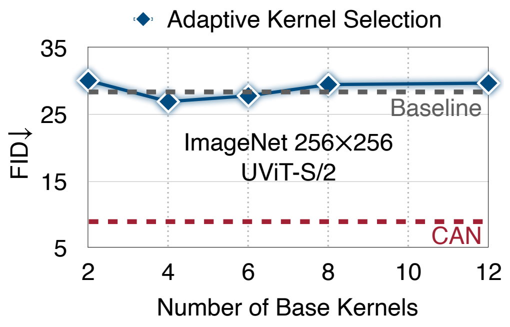


Figure 2.11: **CAN is More Effective than Adaptive Kernel Selection.**

We summarize the results in Table 2.8. We have the following observations in our ablation study experiments:

- Making a module condition-aware does not always improve the performance. For example, using a static head gives a lower FID and a higher CLIP score than using a condition-aware head (row #2 vs. row #4 in Table 2.8).
- Making depthwise convolution layers, the patch embedding layer, and the output projection layers condition-aware brings a significant performance boost. It improves the FID from 28.32 to 8.82 and the CLIP score from 30.09 to 31.74.

Based on these results, we chose this design for CAN. Details are illustrated in Figure 2.10. For the depthwise convolution layers and the patch embedding layer, we use a separate conditional weight generation module for each layer, as their parameter size is small. In contrast, we use a shared conditional weight generation module for the output projection layers, as their parameter size is large. Since different output projection layers have different static weights, we still have different weights for different output projection layers.

CAN vs. Adaptive Kernel Selection. Instead of directly generating the conditional weight, another possible approach is maintaining a set of base convolution kernels and dynam-

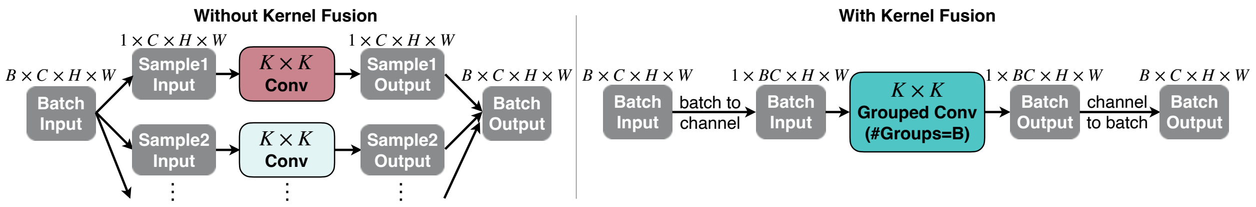


Figure 2.12: **Practical Implementation of CAN.** *Left:* The condition-aware layers have different weights for different samples. A naive implementation requires running the kernel call independently for each sample, which incurs a large overhead for training and batch inference. *Right:* An efficient implementation for CAN. We fuse all kernel calls into a grouped convolution. We insert a batch-to-channel transformation before the kernel call and add a channel-to-batch conversion after the kernel call to preserve the functionality.

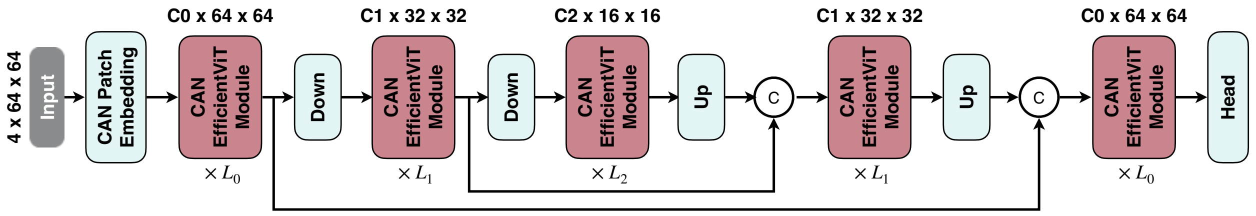


Figure 2.13: **Macro Architecture of CaT.** Benefiting from EfficientViT's linear computational complexity [8], we can keep the high-resolution stages without efficiency concerns.

ically generating scaling parameters to combine these base kernels [74], [85]. This approach's parameter overhead is smaller than CAN. However, this adaptive kernel selection strategy cannot match CAN's performance (Figure 2.11). It suggests that dynamic parameterization alone is not the key to better performances; better condition-aware adaptation capacity is critical.

Implementation. Since the condition-aware layers have different weights given different samples, we cannot do the batch training and inference. Instead, we must run the kernel calls independently for each sample, as shown in Figure 2.12 (left). This will significantly slow down the training process on GPU. To address this issue, we employ an efficient implementation for CAN (Figure 2.12 right). The core insight is to fuse all convolution kernel calls [97] into a grouped convolution where #Groups is the batch size B. We do the batch-to-channel conversion before running the grouped convolution to preserve the functionality. After the operation, we add the channel-to-batch transformation to convert the feature map to the original format.

Theoretically, with this efficient implementation, there will be negligible training overhead compared to running a static model. In practice, as NVIDIA GPU supports regular convolution much better than grouped convolution, we still observe 30%-40% training overhead. This issue can be addressed by writing customized CUDA kernels. We leave it to future work.

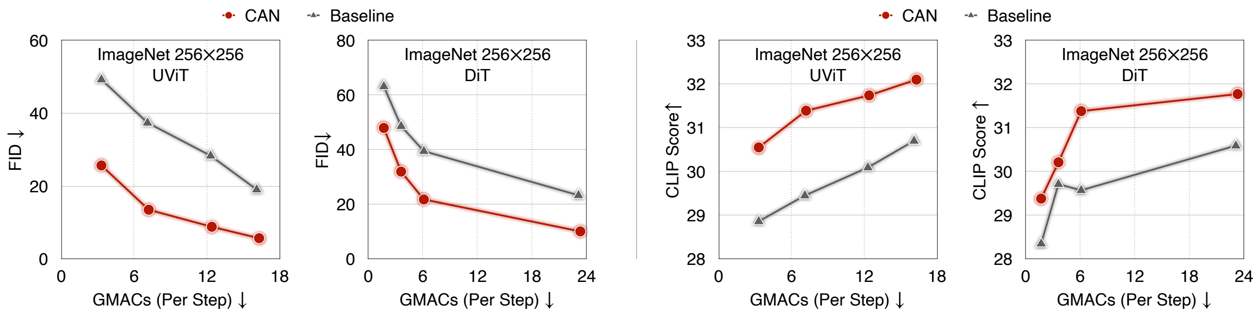


Figure 2.14: **CAN Results on Different UViT and DiT Variants.** CAN consistently delivers lower FID and higher CLIP score for UViT and DiT variants.

2.2.4 Experiments

Setups

Datasets. Due to resource constraints, we conduct class-conditional image generation experiments using the ImageNet dataset and use COCO for text-to-image generation experiments. For large-scale text-to-image experiments [93], we leave them to future work.

Evaluation Metric. Following the common practice, we use FID [98] as the evaluation metric for image quality. In addition, we use the CLIP score [99] as the metric for controllability. We use the public CLIP ViT-B/32 [100] for measuring the CLIP score, following [93]. The text prompts are constructed following CLIP's zero-shot image classification setting [100].

Implementation Details. We apply CAN to recent diffusion transformer models, including DiT [83] and UViT [82]. We follow the training setting suggested in the official paper or GitHub repository. By default, classifier-free guidance [101] is used for all models unless explicitly stated. The baseline models' architectures are the same as the CAN models', having depthwise convolution in FFN layers. We implement our models using Pytorch and train them using A6000 GPUs. Automatic mixed-precision is used during training. In addition to applying CAN to existing models, we also build a new family of diffusion transformers called **CaT** by marrying CAN and EfficientViT [8]. The macro architecture of CaT is illustrated in Figure 2.13.

Ablation Study

We train all models for 80 epochs with batch size 1024 (around 100K iterations) for ablation study experiments unless stated explicitly. All models use DPM-Solver [102] with 50 steps for sampling images.

Effectiveness of CAN. Figure 2.14 summarizes the results of CAN on various UViT and DiT variants. CAN significantly improves the image quality and controllability over the baseline for all variants. Additionally, these improvements come with negligible computational

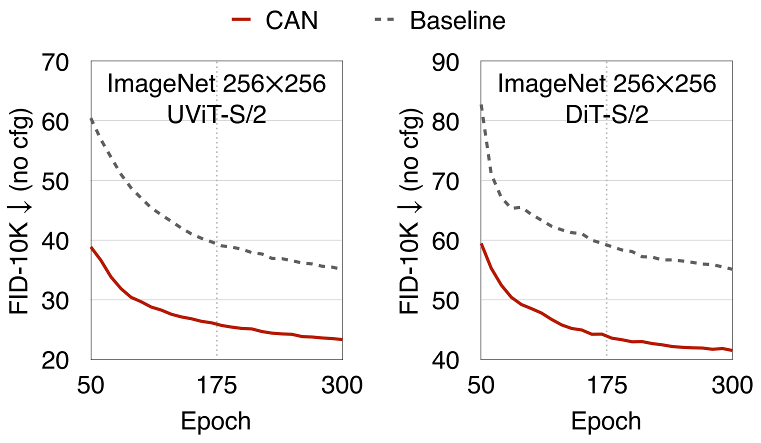


Figure 2.15: **Training Curve.** CAN's improvements are not due to faster convergence. We observe consistent FID improvements when trained longer.

ImageNet 256×256, UViT-S/2		
Models	FID ↓	CLIP Score ↑
Baseline	28.32	30.09
CAN (Timestep Only)	15.16	31.26
CAN (Class Label Only)	10.01	31.59
CAN (All)	8.82	31.74

Table 2.9: **Ablation Study on the Effect of Each Condition for CAN.**

cost overhead. Therefore, CAN also enhances efficiency by delivering the same FID and CLIP score with lower-cost models.

Figure 2.15 compares the training curves of CAN and baseline on UViT-S/2 and DiT-S/2. We can see that the absolute improvement remains significant when trained longer for both models. It shows that the improvements are not due to faster convergence. Instead, adding CAN improves the performance upper bound of the models.

Analysis. For diffusion models, the condition embedding contains both the class label and timestep. To dissect which one is more important for the conditional weight generation process, we conduct the ablation study experiments using UViT-S/2, and summarize the results in Table 2.9. We find that:

- The class label information is more important than the timestep information in the weight generation process. Adding class label alone provides 5.15 lower FID and 0.33 higher CLIP score than adding timestep alone.
- Including the class label and the timestep in the condition embedding delivers the best results. Therefore, we stick to this design in the following experiments.

ImageNet 256×256, DiT-S/2		
Models	FID ↓	CLIP Score ↑
Adaptive Normalization	39.44	29.57
CAN Only	26.44	30.54
CAN + Adaptive Normalization	21.76	30.86
ImageNet 256×256, UViT-S/2		
Models	FID ↓	CLIP Score ↑
Attention (Condition as Tokens)	28.32	30.09
CAN Only	8.79	31.75
CAN + Attention (Condition as Tokens)	8.82	31.74

Table 2.10: **Comparison with Prior Conditional Control Methods.** CAN can work alone without adding other conditional control methods.

Comparison with Prior Conditional Control Methods. In prior experiments, we kept the original conditional control methods of DiT (adaptive normalization) and UViT (attention with condition as tokens) unchanged while adding CAN. To see if CAN can work alone and the comparison between CAN and previous conditional control methods, we conduct experiments and provide the results in Table 2.10. We have the following findings:

- CAN alone can work as an effective conditional control method. For example, CAN alone achieves a 13.00 better FID and 0.97 higher CLIP score than adaptive normalization on DiT-S/2. In addition, CAN alone achieves a 19.53 lower FID and 1.66 higher CLIP score than attention (condition as tokens) on UViT-S/2.
- CAN can be combined with other conditional control methods to achieve better results. For instance, combining CAN with adaptive normalization provides the best results for DiT-S/2.
- For UViT models, combining CAN with attention (condition as tokens) slightly hurts the performance. Therefore, we switch to using CAN alone on UViT models in the following experiments.

Comparison with State-of-the-Art Models

We compare our final models with other diffusion models on ImageNet and COCO. The results are summarized in Table 2.11 and Table 2.13. For CaT models, we use UniPC [104] for sampling images to reduce the number of sampling steps.

Class-Conditional Generation on ImageNet 256×256. As shown in Table 2.11 (bottom), with the classifier-free guidance (cfg), our CaT-B0 achieves 2.09 FID on ImageNet 256×256, outperforming DiT-XL/2 and UViT-H/2. More importantly, our CaT-B0 is much more compute-efficient than these models: 9.9× fewer MACs than DiT-XL/2 and 11.1×

ImageNet 512×512					
Models	FID-50K (no cfg) ↓	FID-50K ↓	#MACs (Per Step) ↓	#Steps ↓	#Params ↓
ADM [103]	23.24	7.72	1983G	250	559M
ADM-U [103]	**9.96**	3.85	2813G	250	730M
UViT-L/4 [82]	-	4.67	77G	50	287M
UViT-H/4 [82]	-	4.05	133G	50	501M
DiT-XL/2 [83]	12.03	3.04	525G	250	675M
CAN (UViT-S-Deep/4)	23.40	4.04	16G	50	185M
CaT-L0	14.25	2.78	10G	20	377M
CaT-L1	10.64	**2.48**	12G	20	486M

ImageNet 256×256					
Models	FID-50K (no cfg) ↓	FID-50K ↓	#MACs (Per Step) ↓	#Steps ↓	#Params ↓
LDM-4 [3]	10.56	3.60	-	250	400M
UViT-L/2 [82]	-	3.40	77G	50	287M
UViT-H/2 [82]	-	2.29	133G	50	501M
DiT-XL/2 [83]	9.62	2.27	119G	250	675M
CAN (UViT-S/2)	16.20	3.52	12G	50	147M
CAN (UViT-S-Deep/2)	11.89	2.78	16G	50	182M
CaT-B0	**8.81**	**2.09**	12G	30	475M

Table 2.11: **Class-Conditional Image Generation Results on ImageNet.**

ImageNet 512×512			
Models	#Steps ↓	Orin Latency ↓	FID ↓
DiT-XL/2 [83]	250	45.9s	3.04
CaT-L0	20	0.2s	2.78

Table 2.12: **NVIDIA Jetson AGX Orin Latency vs. FID.** Latency is profiled with TensorRT and fp16.

fewer MACs than UViT-H/2. Without the classifier-free guidance, our CaT-B0 also achieves the lowest FID among all compared models (8.81 vs. 9.62 vs. 10.56).

Class-Conditional Generation on ImageNet 512×512. On the more challenging 512×512 image generation task, we observe the merits of CAN become more significant. For example, our CAN (UViT-S-Deep/4) can match the performance of UViT-H (4.04 vs. 4.05) while only requiring 12% of UViT-H's computational cost per diffusion step. Additionally, our CaT-L0 delivers 2.78 FID on ImageNet 512×512, outperforming DiT-XL/2 (3.04 FID) that requires 52× higher computational cost per diffusion step. In addition, by slightly scaling up the model, our CaT-L1 further improves the FID from 2.78 to 2.48.

In addition to computational cost comparisons, Table 2.12 compares CaT-L0 and DiT-XL/2 on NVIDIA Jetson AGX Orin. The latency is measured with TensorRT, fp16. Delivering

COCO 256×256			
Models	FID-30K ↓	#MACs ↓	#Params ↓
Friro	8.97	-	512M
UViT-S/2	5.95	15G	45M
UViT-S-Deep/2	5.48	19G	58M
CaT-S0	5.49	3G	169M
CaT-S1	**5.22**	7G	307M

Table 2.13: **Text-to-Image Generation Results on COCO 256×256.**

a better FID on ImageNet 512×512, CaT-L0 combined with a training-free fast sampling method (UniPC) runs 229× faster than DiT-XL/2 on Orin. It is possible to further push the efficiency frontier by combining CaT with training-based few-step methods [105], [106], showing the potential of enabling real-time diffusion model applications on edge devices.

Apart from quantitative results, Figure 2.16 provides samples from the randomly generated images by CAN models, which demonstrate the capability of our models in generating high-quality images.

Text-to-Image Generation on COCO 256×256. For text-to-image generation experiments on COCO, we follow the same setting used in UViT [82]. Specifically, models are trained from scratch on the COCO 2014 training set. Following UViT [82], we randomly sample 30K text prompts from the COCO 2014 validation set to generate images and then compute FID. We use the same CLIP encoder as in UViT for encoding the text prompts. The results are summarized in Table 2.13. Our CaT-S0 achieves a similar FID as UViT-S-Deep/2 while having much less computational cost (19G MACs → 3G MACs). It justifies the generalization ability of our models.

2.2.5 Conclusion

In this section, we studied adding control to image generative models by manipulating their weight. We introduced a new conditional control method, called **Condition-Aware Neural Network (CAN)**, and provided efficient and practical designs for CAN to make it usable in practice. We conducted extensive experiments on class-conditional generation using ImageNet and text-to-image generation using COCO to evaluate CAN's effectiveness. CAN delivered consistent and significant improvements over prior conditional control methods. We also built a new family of diffusion transformer models by marrying CAN and EfficientViT. For future work, we will apply CAN to more challenging tasks like large-scale text-to-image generation, video generation, etc.

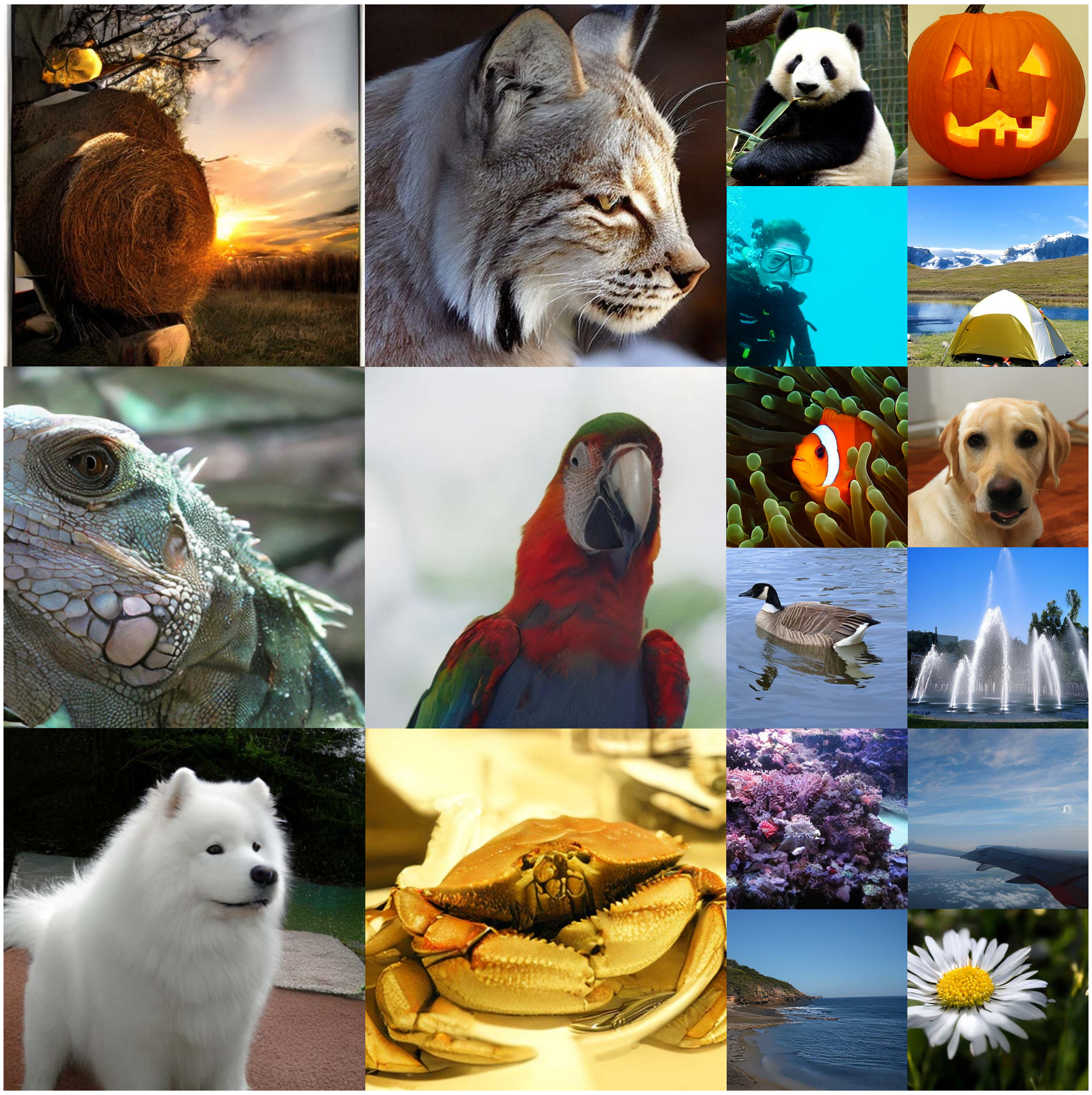


Figure 2.16: **Samples of Generated Images by CAN Models.**

Chapter 3

Hardware-Aware Acceleration

3.1 Direct Neural Architecture Search on Target Task and Hardware

3.1.1 Introduction

Neural architecture search (NAS) has demonstrated much success in automating neural network architecture design for various deep learning tasks, such as image recognition [44], [107]–[109] and language modeling [43]. Despite the remarkable results, conventional NAS algorithms are prohibitively computation-intensive, requiring to train thousands of models on the target task in a single experiment. Therefore, directly applying NAS to a large-scale task (e.g. ImageNet) is computationally expensive or impossible, which makes it difficult for making practical industry impact. As a trade-off, [107] propose to search for building blocks on proxy tasks, such as training for fewer epochs, starting with a smaller dataset (e.g. CIFAR-10), or learning with fewer blocks. Then top-performing blocks are stacked and transferred to the large-scale target task. This paradigm has been widely adopted in subsequent NAS algorithms [108], [110]–[115].

However, these blocks optimized on proxy tasks are not guaranteed to be optimal on the target task, especially when taking hardware metrics such as latency into consideration. More importantly, to enable transferability, such methods need to search for only a few architectural motifs and then repeatedly stack the same pattern, which restricts the block diversity and thereby harms performance.

In this work, we propose a simple and effective solution to the aforementioned limitations, called *ProxylessNAS*, which directly learns the architectures on the target task and hardware instead of with proxy (Figure 3.1). We also remove the restriction of repeating blocks in previous NAS works [107], [113] and allow all of the blocks to be learned and specified. To achieve this, we reduce the computational cost (GPU hours and GPU memory) of architecture search to the same level of regular training in the following ways.

GPU hour-wise, inspired by recent works [113], [116], we formulate NAS as a path-level pruning process. Specifically, we directly train an over-parameterized network that contains all candidate paths (Figure 3.2). During training, we explicitly introduce architecture parameters to learn which paths are redundant, while these redundant paths are pruned at the end of

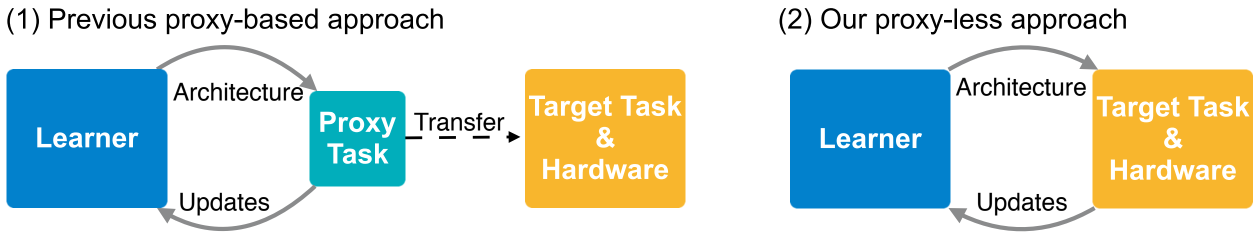


Figure 3.1: ProxylessNAS directly optimizes neural network architectures on target task and hardware. Benefiting from the directness and specialization, ProxylessNAS can achieve remarkably better results than previous proxy-based approaches. On ImageNet, with only 200 GPU hours (200 × fewer than MnasNet [114]), our searched CNN model for mobile achieves the same level of top-1 accuracy as MobileNetV2 1.4 while being 1.8× faster.

training to get a compact optimized architecture. In this way, we only need to train a single network without any meta-controller (or hypernetwork) during architecture search.

However, naively including all the candidate paths leads to GPU memory explosion [113], [116], as the memory consumption grows linearly w.r.t. the number of choices. Thus, GPU memory-wise, we binarize the architecture parameters (1 or 0) and force only one path to be active at run-time, which reduces the required memory to the same level of training a compact model. We propose a gradient-based approach to train these binarized parameters based on BinaryConnect [117]. Furthermore, to handle non-differentiable hardware objectives (using latency as an example) for learning specialized network architectures on target hardware, we model network latency as a continuous function and optimize it as regularization loss. Additionally, we also present a REINFORCE-based [118] algorithm as an alternative strategy to handle hardware metrics.

In our experiments on CIFAR-10 and ImageNet, benefiting from the directness and specialization, our method can achieve strong empirical results. On CIFAR-10, our model reaches 2.08% test error with only 5.7M parameters. On ImageNet, our model achieves 75.1% top-1 accuracy which is 3.1% higher than MobileNetV2 [119] while being 1.2× faster.

3.1.2 Related Work

The use of machine learning techniques, such as reinforcement learning or neuro-evolution, to replace human experts in designing neural network architectures, usually referred to as neural architecture search, has drawn an increasing interest [43], [44], [90], [108], [110], [112], [113], [116], [120]–[123]. In NAS, architecture search is typically considered as a meta-learning process, and a meta-controller (e.g. a recurrent neural network (RNN)), is introduced to explore a given architecture space with training a network in the inner loop to get an evaluation for guiding exploration. Consequently, such methods are computationally expensive to run, especially on large-scale tasks, e.g. ImageNet.

Some recent works [90], [120] try to improve the efficiency of this meta-learning process by reducing the cost of getting an evaluation. In [90], a hypernetwork is utilized to generate weights for each sampled network and hence can evaluate the architecture without training it. Similarly, [120] propose to share weights among all sampled networks under the standard NAS framework [43]. These methods speed up architecture search by orders of magnitude,

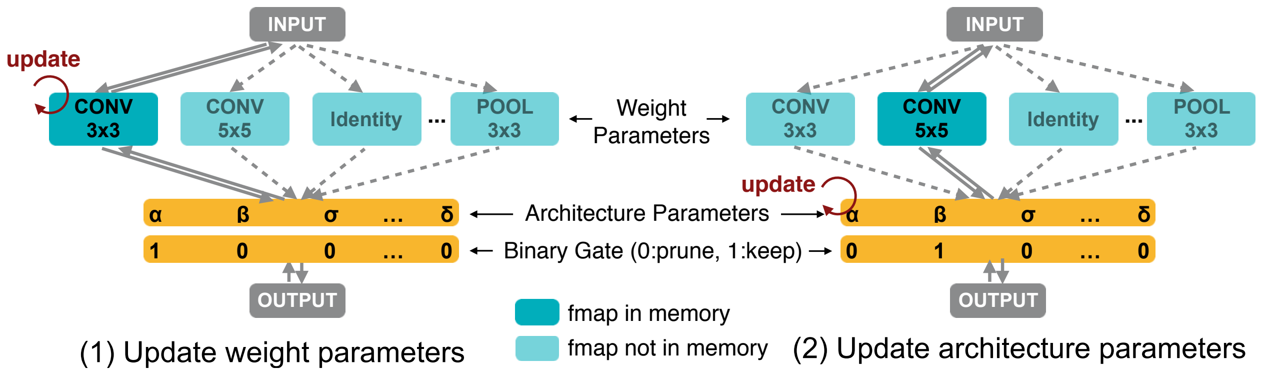


Figure 3.2: Learning both weight parameters and binarized architecture parameters.

however, they require a hypernetwork or an RNN controller and mainly focus on small-scale tasks (e.g. CIFAR) rather than large-scale tasks (e.g. ImageNet).

Our work is most closely related to One-Shot [116] and DARTS [113], both of which get rid of the meta-controller (or hypernetwork) by modeling NAS as a single training process of an over-parameterized network that comprises all candidate paths. Specifically, One-Shot trains the over-parameterized network with DropPath [107] that drops out each path with some fixed probability. Then they use the pre-trained over-parameterized network to evaluate architectures, which are sampled by randomly zeroing out paths. DARTS additionally introduces a real-valued architecture parameter for each path and jointly train weight parameters and architecture parameters via standard gradient descent. However, they suffer from the large GPU memory consumption issue and hence still need to utilize proxy tasks. In this work, we address the large memory issue in these two methods through path binarization.

Another relevant topic is network pruning [92] that aim to improve the efficiency of neural networks by removing insignificant neurons [35] or channels [37]. Similar to these works, we start with an over-parameterized network and then prune the redundant parts to derive the optimized architecture. The distinction is that they focus on layer-level pruning that only modifies the filter (or units) number of a layer but can not change the topology of the network, while we focus on learning effective network architectures through *path-level pruning*. We also allow both pruning and growing the number of layers.

3.1.3 Method

We first describe the construction of the over-parameterized network with all candidate paths, then introduce how we leverage binarized architecture parameters to reduce the memory consumption of training the over-parameterized network to the same level as regular training. We propose a gradient-based algorithm to train these binarized architecture parameters. Finally, we present two techniques to handle non-differentiable objectives (e.g. latency) for specializing neural networks on target hardware.

Construction of Over-Parameterized Network

Denote a neural network as $\mathcal{N}(e, \cdots, e_n)$ where e_i represents a certain edge in the directed acyclic graph (DAG). Let $\mathcal{O} = \{o_i\}$ be the set of N candidate primitive operations (e.g. convolution, pooling, identity, zero, etc). To construct the over-parameterized network that includes any architecture in the search space, instead of setting each edge to be a definite primitive operation, we set each edge to be a mixed operation that has N parallel paths (Figure 3.2), denoted as $m_{\mathcal{O}}$. As such, the over-parameterized network can be expressed as $\mathcal{N}(e = m_{\mathcal{O}}^1, \cdots, e_n = m_{\mathcal{O}}^n)$.

Given input x, the output of a mixed operation $m_{\mathcal{O}}$ is defined based on the outputs of its N paths. In One-Shot, $m_{\mathcal{O}}(x)$ is the sum of $\{o_i(x)\}$, while in DARTS, $m_{\mathcal{O}}(x)$ is weighted sum of $\{o_i(x)\}$ where the weights are calculated by applying softmax to N real-valued architecture parameters $\{\alpha_i\}$:

$$m_{\mathcal{O}}^{\text{One-Shot}}(x) = \sum_{i=1}^{N} o_i(x), \qquad m_{\mathcal{O}}^{\text{DARTS}}(x) = \sum_{i=1}^{N} p_i o_i(x) = \sum_{i=1}^{N} \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)} o_i(x). \tag{3.1}$$

As shown in Eq. (3.1), the output feature maps of all N paths are calculated and stored in the memory, while training a compact model only involves one path. Therefore, One-Shot and DARTS roughly need N times GPU memory and GPU hours compared to training a compact model. On large-scale dataset, this can easily exceed the memory limits of hardware with large design space. In the following section, we solve this memory issue based on the idea of path binarization.

Learning Binarized Path

To reduce memory footprint, we keep only one path when training the over-parameterized network. Unlike [117] which binarize individual weights, we binarize entire paths. We introduce N real-valued architecture parameters $\{\alpha_i\}$ and then transforms the real-valued path weights to binary gates:

$$g = \text{binarize}(p_1, \cdots, p_N) = \begin{cases} [1, 0, \cdots, 0] & \text{with probability } p_1, \\ \quad\cdots & \\ [0, 0, \cdots, 1] & \text{with probability } p_N. \end{cases} \tag{3.2}$$

Based on the binary gates g, the output of the mixed operation is given as:

$$m_{\mathcal{O}}^{\text{Binary}}(x) = \sum_{i=1}^{N} g_i o_i(x) = \begin{cases} o_1(x) & \text{with probability } p_1 \\ \quad\cdots & \\ o_N(x) & \text{with probability } p_N. \end{cases}. \tag{3.3}$$

As illustrated in Eq. (3.3) and Figure 3.2, by using the binary gates rather than real-valued path weights [113], only one path of activation is active in memory at run-time and the memory requirement of training the over-parameterized network is thus reduced to the same level of training a compact model. That's more than an order of magnitude memory saving.

Training Binarized Architecture Parameters. Figure 3.2 illustrates the training procedure of the weight parameters and binarized architecture parameters in the over-parameterized network. When training weight parameters, we first freeze the architecture parameters and stochastically sample binary gates according to Eq. (3.2) for each batch of input data. Then the weight parameters of active paths are updated via standard gradient descent on the training set (Figure 3.2 left). When training architecture parameters, the weight parameters are frozen, then we reset the binary gates and update the architecture parameters on the validation set (Figure 3.2 right). These two update steps are performed in an alternative manner. Once the training of architecture parameters is finished, we can then derive the compact architecture by pruning redundant paths. In this work, we simply choose the path with the highest path weight.

Unlike weight parameters, the architecture parameters are not directly involved in the computation graph and thereby cannot be updated using the standard gradient descent. In this section, we introduce a gradient-based approach to learn the architecture parameters.

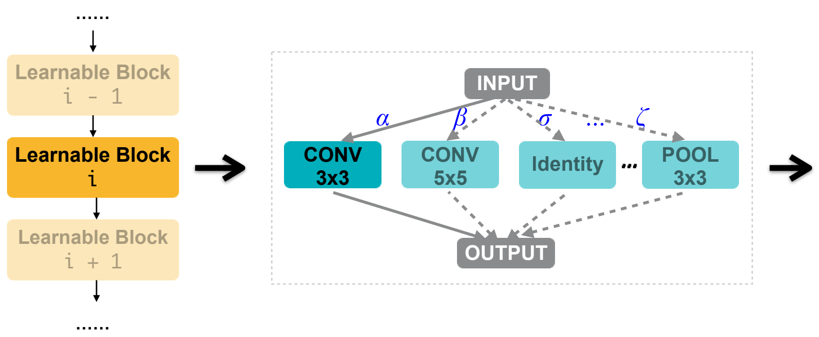
In BinaryConnect [117], the real-valued weight is updated using the gradient w.r.t. its corresponding binary gate. In our case, analogously, the gradient w.r.t. architecture parameters can be approximately estimated using $\partial L/\partial g_i$ in replace of $\partial L/\partial p_i$:

$$\frac{\partial L}{\partial \alpha_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial p_j}\frac{\partial p_j}{\partial \alpha_i} \approx \sum_{j=1}^{N} \frac{\partial L}{\partial g_j}\frac{\partial p_j}{\partial \alpha_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial g_j}\frac{\partial\left(\frac{\exp(\alpha_j)}{\sum_k \exp(\alpha_k)}\right)}{\partial \alpha_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial g_j} p_j(\delta_{ij} - p_i), \tag{3.4}$$

where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$. Since the binary gates g are involved in the computation graph, as shown in Eq. (3.3), $\partial L/\partial g_j$ can be calculated through backpropagation. However, computing $\partial L/\partial g_j$ requires to calculate and store $o_j(x)$. Therefore, directly using Eq. (3.4) to update the architecture parameters would also require roughly N times GPU memory compared to training a compact model.

To address this issue, we consider factorizing the task of choosing one path out of N candidates into multiple binary selection tasks. The intuition is that if a path is the best choice at a particular position, it should be the better choice when solely compared to any other path.

Following this idea, within an update step of the architecture parameters, we first sample two paths according to the multinomial distribution $(p_1, \cdots, p_N)$ and mask all the other paths as if they do not exist. As such the number of candidates temporarily decrease from N to 2, while the path weights $\{p_i\}$ and binary gates $\{g_i\}$ are reset accordingly. Then we update the architecture parameters of these two sampled paths using the gradients calculated via Eq. (3.4). Finally, as path weights are computed by applying softmax to the architecture parameters, we need to rescale the value of these two updated architecture parameters by multiplying a ratio to keep the path weights of unsampled paths unchanged. As such, in each update step, one of the sampled paths is enhanced (path weight increases) and the other sampled path is attenuated (path weight decreases) while all other paths keep unchanged. In this way, regardless of the value of N, only two paths are involved in each update step of the architecture parameters, and thereby the memory requirement is reduced to the same level of training a compact model.



$$\begin{aligned}\mathbb{E}[\text{Latency}] = {} & \alpha \times F(\text{conv\_3x3}) + \\ & \beta \times F(\text{conv\_5x5}) + \\ & \sigma \times F(\text{identity}) + \\ & \ldots\ldots \\ & \zeta \times F(\text{pool\_3x3})\end{aligned}$$

$$\mathbb{E}[\text{latency}] = \sum_i \mathbb{E}[\text{latency}_\text{i}]$$

$$Loss = Loss_{CE} + \lambda_1 ||w||_2^2 + \lambda_2 \mathbb{E}[\text{latency}]$$

Figure 3.3: Making latency differentiable by introducing latency regularization loss.

Handling Non-differentiable Hardware Metrics

Besides accuracy, latency (not FLOPs) is another very important objective when designing efficient neural network architectures for hardware. Unfortunately, unlike accuracy that can be optimized using the gradient of the loss function, latency is non-differentiable. In this section, we present two algorithms to handle the non-differentiable objectives.

Making Latency Differentiable. To make latency differentiable, we model the latency of a network as a continuous function of the neural network dimensions. Consider a mixed operation with a candidate set $\{o_j\}$ and each o_j is associated with a path weight p_j which represents the probability of choosing o_j. As such, we have the expected latency of a mixed operation (i.e. a learnable block) as:

$$\mathbb{E}[\text{latency}_i] = \sum_j p_j^i \times F(o_j^i), \tag{3.5}$$

where $\mathbb{E}[\text{latency}_i]$ is the expected latency of the i^{th} learnable block, $F(\cdot)$ denotes the latency prediction model and $F(o_j^i)$ is the predicted latency of o_j^i. The gradient of $\mathbb{E}[\text{latency}_i]$ w.r.t. architecture parameters can thereby be given as: $\partial \mathbb{E}[\text{latency}_i] \;/\; \partial p_j^i = F(o_j^i)$.

For the whole network with a sequence of mixed operations (Figure 3.3 left), since these operations are executed sequentially during inference, the expected latency of the network can be expressed with the sum of these mixed operations' expected latencies:

$$\mathbb{E}[\text{latency}] = \sum_i \mathbb{E}[\text{latency}_i], \tag{3.6}$$

We incorporate the expected latency of the network into the normal loss function by multiplying a scaling factor $\lambda_2 (> 0)$ which controls the trade-off between accuracy and latency. The final loss function is given as (also shown in Figure 3.3 right)

$$Loss = Loss_{CE} + \lambda_1 ||w||_2^2 + \lambda_2 \mathbb{E}[\text{latency}], \tag{3.7}$$

where $Loss_{CE}$ denotes the cross-entropy loss and $\lambda_1 ||w||_2^2$ is the weight decay term.

Model	Top-1	Top-5	Mobile Latency	Hardware -aware	No Proxy	No Repeat	Search cost (GPU hours)
MobileNetV1 [39]	70.6	89.5	113ms	-	-	✗	Manual
MobileNetV2 [119]	72.0	91.0	75ms	-	-	✗	Manual
NASNet-A [107]	74.0	91.3	183ms	✗	✗	✗	$48,000$
AmoebaNet-A [111]	74.5	92.0	190ms	✗	✗	✗	$75,600$
MnasNet [114]	74.0	91.8	76ms	✓	✗	✗	$40,000$
MnasNet (our impl.)	74.0	91.8	79ms	✓	✗	✗	$40,000$
Proxyless-G (mobile)	71.8	90.3	83ms	✗	✓	✓	200
Proxyless-G + LL	74.2	91.7	79ms	✓	✓	✓	200
Proxyless-R (mobile)	**74.6**	**92.2**	78ms	✓	✓	✓	200

Table 3.1: ProxylessNAS achieves state-of-the art accuracy (%) on ImageNet (under mobile latency constraint $\leq 80ms$) with 200× less search cost in GPU hours. "LL" indicates latency regularization loss.

REINFORCE-based Approach. As an alternative to BinaryConnect, we can utilize REINFORCE to train binarized weights as well. Consider a network that has binarized parameters α, the goal of updating binarized parameters is to find the optimal binary gates g that maximizes a certain reward, denoted as $R(\cdot)$. Here we assume the network only has one mixed operation for ease of illustration. Therefore, according to REINFORCE [118], we have the following updates for binarized parameters:

$$\begin{aligned} J(\alpha) &= \mathbb{E}_{g\sim\alpha}[R(\mathcal{N}_g)] = \sum_i p_i R(\mathcal{N}(e=o_i)), \\ \nabla_\alpha J(\alpha) &= \sum_i R(\mathcal{N}(e=o_i))\nabla_\alpha p_i = \sum_i R(\mathcal{N}(e=o_i)) p_i \nabla_\alpha \log(p_i), \\ &= \mathbb{E}_{g\sim\alpha}[R(\mathcal{N}_g)\nabla_\alpha \log(p(g))] \approx \frac{1}{M}\sum_{i=1}^{M} R(\mathcal{N}_{g^i})\nabla_\alpha \log(p(g^i)), \end{aligned} \tag{3.8}$$

where g^i denotes the i^{th} sampled binary gates, $p(g^i)$ denotes the probability of sampling g^i according to Eq. (3.2) and $\mathcal{N}_{g^i}$ is the compact network according to the binary gates g^i. Since Eq. (3.8) does not require $R(\mathcal{N}_g)$ to be differentiable w.r.t. g, it can thus handle non-differentiable objectives. An interesting observation is that Eq. (3.8) has a similar form to the standard NAS [43], while it is not a sequential decision-making process and no RNN meta-controller is used in our case. Furthermore, since both gradient-based updates and REINFORCE-based updates are essentially two different update rules to the same binarized architecture parameters, it is possible to combine them to form a new update rule for the architecture parameters.

3.1.4 Experiments

For ImageNet experiments, we focus on learning efficient CNN architectures [39], [119], [124] that have not only high accuracy but also low latency on specific hardware platforms. Therefore, it is a multi-objective NAS task [45], [114], [125]–[128], where one of the objectives

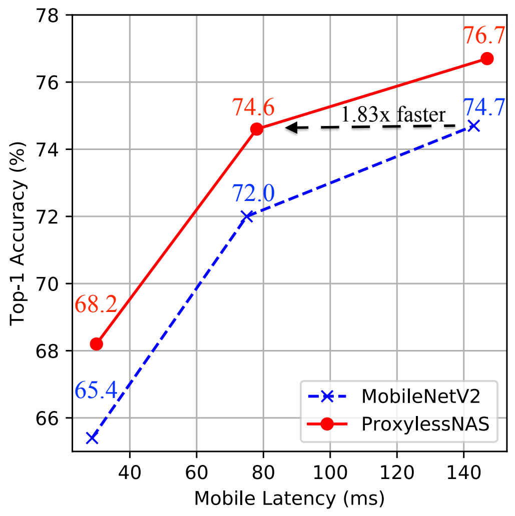


Figure 3.4: ProxylessNAS consistently outperforms MobileNetV2 under various latency settings.

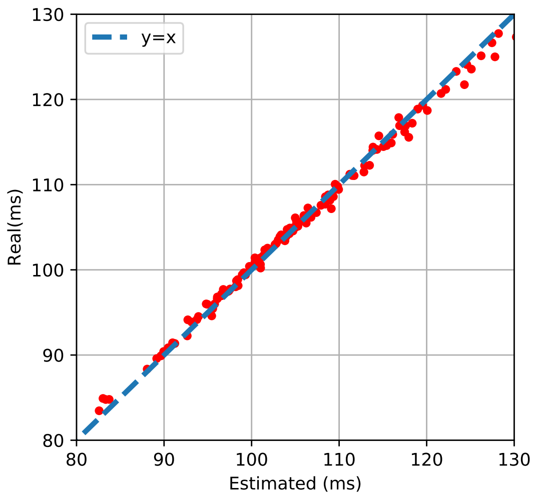


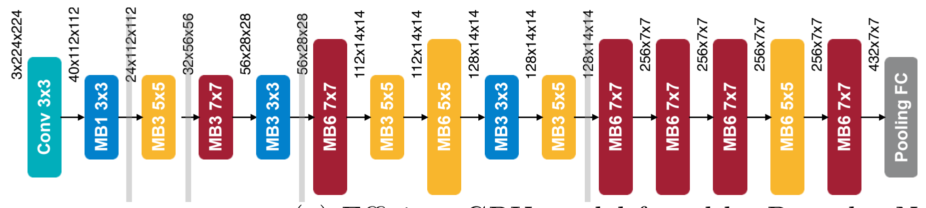
Figure 3.5: Our mobile latency model is close to $y = x$. The latency RMSE is 0.75ms.

is non-differentiable (i.e. latency). We use three different hardware platforms, including mobile phone, GPU and CPU, in our experiments. The GPU latency is measured on V100 GPU with a batch size of 8 (single batch makes GPU severely under-utilized). The CPU latency is measured under batch size 1 on a server with two 2.40GHz Intel(R) Xeon(R) CPU E5-2640 v4. The mobile latency is measured on Google Pixel 1 phone with a batch size of 1. For Proxyless-R, we use $ACC(m) \times [LAT(m)/T]^w$ as the optimization goal, where $ACC(m)$ denotes the accuracy of model m, $LAT(m)$ denotes the latency of m, T is the target latency and w is a hyperparameter for controlling the trade-off between accuracy and latency.

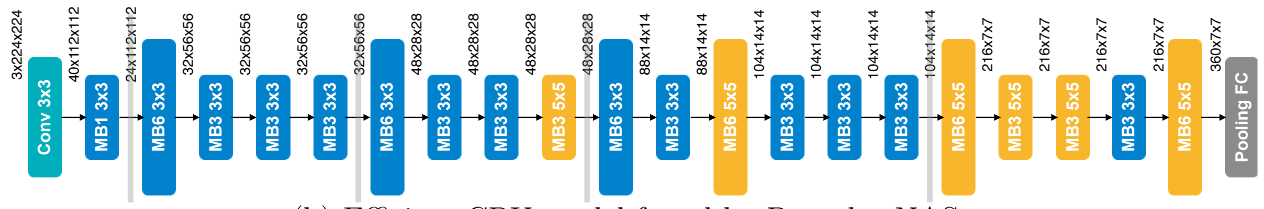
Additionally, on mobile phone, we use the latency prediction model during architecture search. As illustrated in Figure 3.5, we observe a strong correlation between the predicted latency and real measured latency on the test set, suggesting that the latency prediction model can be used to replace the expensive mobile farm infrastructure [114] with little error introduced.

Architecture Space

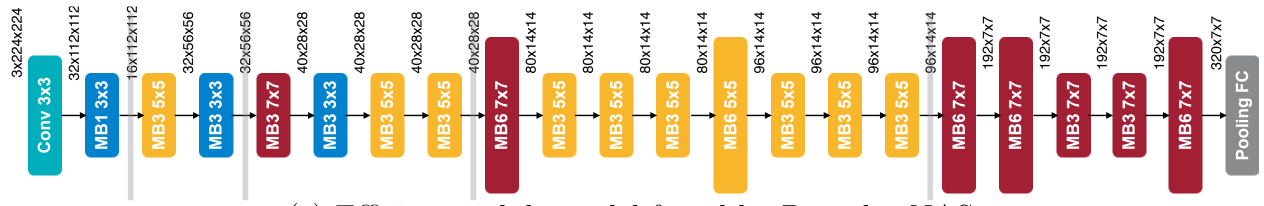
We use MobileNetV2 [119] as the backbone to build the architecture space. Specifically, rather than repeating the same mobile inverted bottleneck convolution (MBConv), we allow a set of MBConv layers with various kernel sizes $\{3, 5, 7\}$ and expansion ratios $\{3, 6\}$. To enable a direct trade-off between width and depth, we initiate a deeper over-parameterized network and allow a block with the residual connection to be skipped by adding the zero operation to the candidate set of its mixed operation. In this way, with a limited latency budget, the network can either choose to be shallower and wider by skipping more blocks and using larger MBConv layers or choose to be deeper and thinner by keeping more blocks and using smaller MBConv layers.



(a) Efficient GPU model found by ProxylessNAS.



(b) Efficient CPU model found by ProxylessNAS.



(c) Efficient mobile model found by ProxylessNAS.

Figure 3.6: Efficient models optimized for different hardware. "MBConv3" and "MBConv6" denote mobile inverted bottleneck convolution layer with an expansion ratio of 3 and 6 respectively. Insights: GPU prefers shallow and wide model with early pooling; CPU prefers deep and narrow model with late pooling. Pooling layers prefer large and wide kernel. Early layers prefer small kernel. Late layers prefer large kernel.

Training Details

We randomly sample 50,000 images from the training set as a validation set during the architecture search. The settings for updating architecture parameters are the same as CIFAR-10 experiments except the initial learning rate is 0.001. The over-parameterized network is trained on the remaining training images with batch size 256.

ImageNet Classification Results

We first apply our ProxylessNAS to learn specialized CNN models on the mobile phone. The summarized results are reported in Table 3.1. Compared to MobileNetV2, our model improves the top-1 accuracy by 2.6% while maintaining a similar latency on the mobile phone. Furthermore, by rescaling the width of the networks using a multiplier [114], [119], it is shown in Figure 3.4 that our model consistently outperforms MobileNetV2 by a significant margin under all latency settings. Specifically, to achieve the same level of top-1 accuracy performance (i.e. around 74.6%), **MobileNetV2 has 143ms latency while our model only needs 78ms (1.83× faster)**. While compared with MnasNet [114], our model can achieve 0.6% higher top-1 accuracy with slightly lower mobile latency. More importantly, we are much more resource efficient: the GPU-hour is 200× fewer than MnasNet (Table 3.1).

Model	Top-1	Top-5	GPU latency
MobileNetV2 [119]	72.0	91.0	6.1ms
ShuffleNetV2 (1.5) [40]	72.6	-	7.3ms
ResNet-34 [129]	73.3	91.4	8.0ms
NASNet-A [107]	74.0	91.3	38.3ms
DARTS [113]	73.1	91.0	-
MnasNet [114]	74.0	91.8	6.1ms
Proxyless (GPU)	**75.1**	**92.5**	**5.1ms**

Table 3.2: ImageNet Accuracy (%) and GPU latency (Tesla V100) on ImageNet.

Model	Top-1 (%)	GPU latency	CPU latency	Mobile latency
Proxyless (GPU)	75.1	5.1ms	204.9ms	124ms
Proxyless (CPU)	75.3	7.4ms	138.7ms	116ms
Proxyless (mobile)	74.6	7.2ms	164.1ms	78ms

Table 3.3: Hardware prefers specialized models. Models optimized for GPU does not run fast on CPU and mobile phone, vice versa. ProxylessNAS provides an efficient solution to search a specialized neural network architecture for a target hardware architecture, while cutting down the search cost by 200× compared with state-of-the-arts [43], [114].

Additionally, we also observe that Proxyless-G has no incentive to choose computation-cheap operations if were not for the latency regularization loss. Its resulting architecture initially has 158ms latency on Pixel 1. After rescaling the network using the multiplier, its latency reduces to 83ms. However, this model can only achieve 71.8% top-1 accuracy on ImageNet, which is 2.4% lower than the result given by Proxyless-G with latency regularization loss. Therefore, we conclude that it is essential to take latency as a direct objective when learning efficient neural networks.

Besides the mobile phone, we also apply our ProxylessNAS to learn specialized CNN models on GPU and CPU. Table 3.2 reports the results on GPU, where we find that our ProxylessNAS can still achieve superior performances compared to both human-designed and automatically searched architectures. Specifically, compared to MobileNetV2 and MnasNet, our model improves the top-1 accuracy by 3.1% and 1.1% respectively while being 1.2× faster. Table 3.3 shows the summarized results of our searched models on three different platforms. An interesting observation is that models optimized for GPU do not run fast on CPU and mobile phone, vice versa. Therefore, it is essential to learn specialized neural networks for different hardware architectures to achieve the best efficiency on different hardware.

Specialized Models for Different Hardware

Figure 3.6 demonstrates the detailed architectures of our searched CNN models on three hardware platforms: GPU/CPU/Mobile. We notice that the architecture shows different preferences when targeting different platforms: (i) The GPU model is shallower and wider, especially in early stages where the feature map has higher resolution; (ii) The GPU model prefers large MBConv operations (e.g. 7×7 MBConv6), while the CPU model would go for smaller MBConv operations. This is because GPU has much higher parallelism than

CPU so it can take advantage of large MBConv operations. Another interesting observation is that our searched models on all platforms prefer larger MBConv operations in the first block within each stage where the feature map is downsampled. We suppose it might because larger MBConv operations are beneficial for the network to preserve more information when downsampling. Notably, such kind of patterns cannot be captured in previous NAS methods as they force the blocks to share the same structure [107], [108].

3.1.5 Conclusion

We introduced ProxylessNAS that can directly learn neural network architectures on the target task and target hardware without any proxy. We also reduced the search cost (GPU-hours and GPU memory) of NAS to the same level of normal training using path binarization. Benefiting from the direct search, we achieve strong empirical results on CIFAR-10 and ImageNet. Furthermore, we allow specializing network architectures for different platforms by directly incorporating the measured hardware latency into optimization objectives. We compared the optimized models on CPU/GPU/mobile and raised the awareness of the needs of specializing neural network architecture for different hardware architectures.

3.2 Once-for-All Network for Diverse Deployment Scenarios

3.2.1 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 [39], [119], [130] or accelerate the existing models by compression [38], [45] 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 CO_2 emission [131]. It makes them unable to handle the vast amount of hardware devices (23.14 billion IoT devices till 2018[1]) and highly dynamic

[1]https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

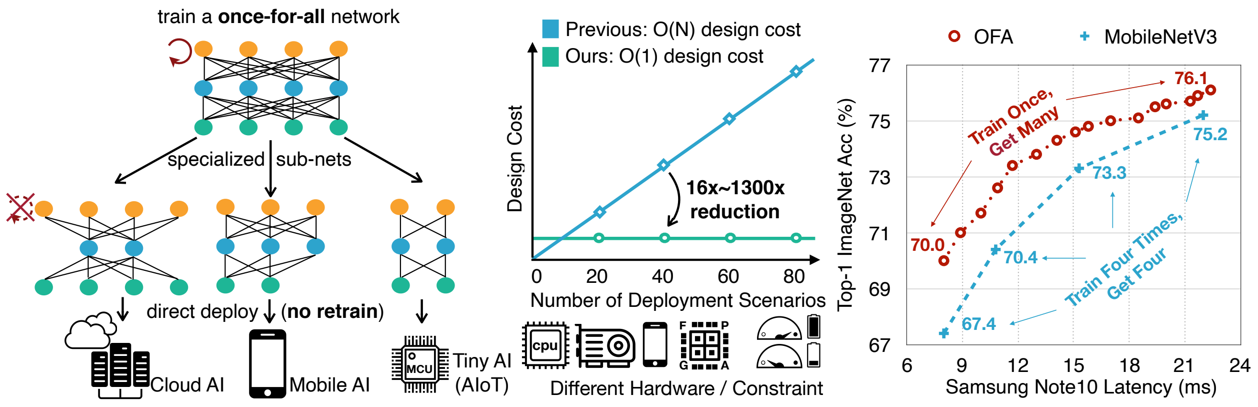


Figure 3.7: Left: a single once-for-all networkis trained to support versatile architectural configurations including depth, width, kernel size, and resolution. Given a deployment scenario, a specialized sub-network is directly selected from the once-for-all networkwithout training. Middle: this approach reduces the cost of specialized deep learning deployment from O(N) to O(1). Right: once-for-all network followed by model selection can derive many accuracy-latency trade-offs by training only once, compared to conventional methods that require repeated training.

deployment environments (different battery conditions, different latency requirements, etc.).

This paper introduces a new solution to tackle this challenge – designing a *once-for-all network* that can be directly deployed under diverse architectural configurations, amortizing the training cost. The inference is performed by selecting only part of the once-for-all network. It flexibly supports different depths, widths, kernel sizes, and resolutions without retraining. A simple example of *Once-for-All* (OFA) is illustrated in Figure 3.7 (left). Specifically, we decouple the model training stage and the neural architecture search stage. In the model training stage, we focus on improving the accuracy of all sub-networks that are derived by selecting different parts of the once-for-all network. In the model specialization stage, we sample a subset of sub-networks to train an accuracy predictor and latency predictors. Given the target hardware and constraint, a predictor-guided architecture search [132] is conducted to get a specialized sub-network, and the cost is negligible. As such, we reduce the total cost of specialized neural network design from O(N) to O(1) (Figure 3.7 middle).

However, training the once-for-all networkis a non-trivial task, since it requires joint optimization of the weights to maintain the accuracy of a large number of sub-networks (more than 10^{19} in our experiments). It is computationally prohibitive to enumerate all sub-networks to get the exact gradient in each update step, while randomly sampling a few sub-networks in each step will lead to significant accuracy drops. The challenge is that different sub-networks are interfering with each other, making the training process of the whole once-for-all networkinefficient. To address this challenge, we propose a *progressive shrinking* algorithm for training the once-for-all network. Instead of directly optimizing the once-for-all networkfrom scratch, we propose to first train the largest neural network with *maximum* depth, width, and kernel size, then progressively fine-tune the once-for-all networkto support *smaller* sub-networks that share weights with the larger ones. As such, it provides better initialization by selecting the most important weights of larger sub-networks,

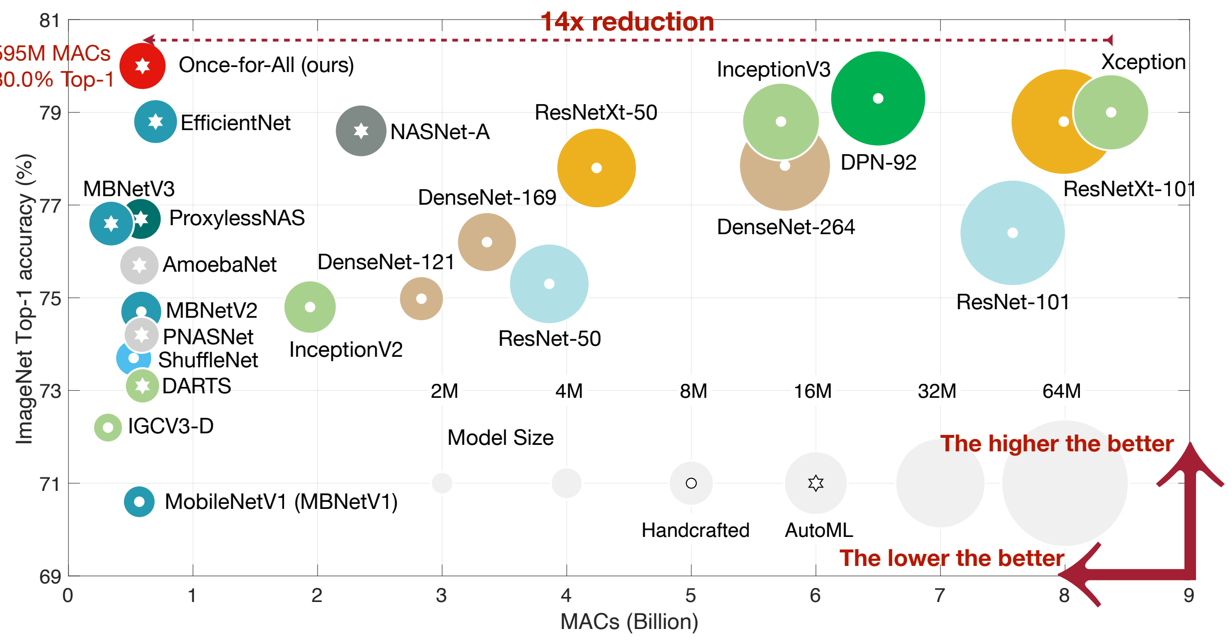


Figure 3.8: Comparison between OFA and state-of-the-art CNN models on ImageNet. OFA provides 80.0% ImageNet top1 accuracy under the mobile setting (< 600M MACs).

and the opportunity to distill smaller sub-networks, which greatly improves the training efficiency. From this perspective, progressive shrinking can be viewed as a generalized network pruning method that shrinks multiple dimensions (depth, width, kernel size, and resolution) of the full network rather than only the width dimension. Besides, it targets on maintaining the accuracy of all sub-networks rather than a single pruned network.

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 CO_2 emission by orders of magnitude. On the ImageNet mobile setting (less than 600M MACs), OFA achieves a new SOTA 80.0% top1 accuracy with 595M MACs (Figure 3.8). To the best of our knowledge, this is the first time that the SOTA ImageNet top1 accuracy reaches 80% under the mobile setting.

3.2.2 Related Work

Efficient Deep Learning. Many efficient neural network architectures are proposed to improve the hardware efficiency, such as SqueezeNet [124], MobileNets [39], [119], ShuffleNets [40], [130], etc. Orthogonal to architecting efficient neural networks, model compression [38], [133] is another very effective technique for efficient deep learning, including network pruning that removes redundant units [35] or redundant channels [37], [45], and quantization that reduces the bit width for the weights and activations [38], [117], [134].

Neural Architecture Search. Neural architecture search (NAS) focuses on automating the architecture design process [43], [44], [107], [111], [113], [135]. Early NAS methods [107], [111], [112] 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 [10], [114], [136] directly incorporate the hardware feedback into architecture search. As a result, they can 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 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, [137]–[139] learn a controller or gating modules to adaptively drop layers; [140] introduce early-exit branches in the computation graph; [141] adaptively prune channels based on the input feature map; [142] introduce stochastic downsampling point to reduce the feature map size adaptively. Recently, Slimmable Nets [88], [143] 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 [88]). 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.

3.2.3 Method

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

$$\min_{W_o} \sum_{arch_i} \mathcal{L}_{val}\big(C(W_o, arch_i)\big), \tag{3.9}$$

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.

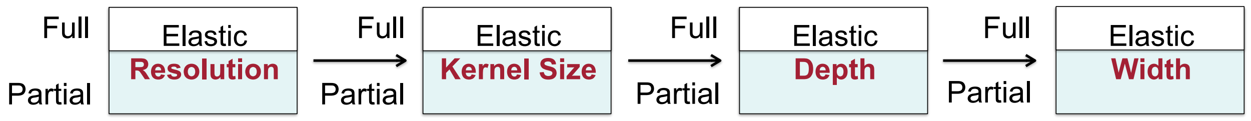


Figure 3.9: Illustration of the progressive shrinking process to support different depth D, width W, kernel size K and resolution R. It leads to a large space comprising diverse sub-networks ($> 10^{19}$).

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 [119], [129], [144], 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 [119]. 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 $((3 \times 3)^2 + (3 \times 3)^3 + (3 \times 3)^4)^5 \approx 2 \times 10^{19}$ 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) [145], we only require 7.7M parameters to store all of them. Without sharing, the total model size will be prohibitive.

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. (3.9). 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 [88], 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, i.e., *progressive shrinking*.

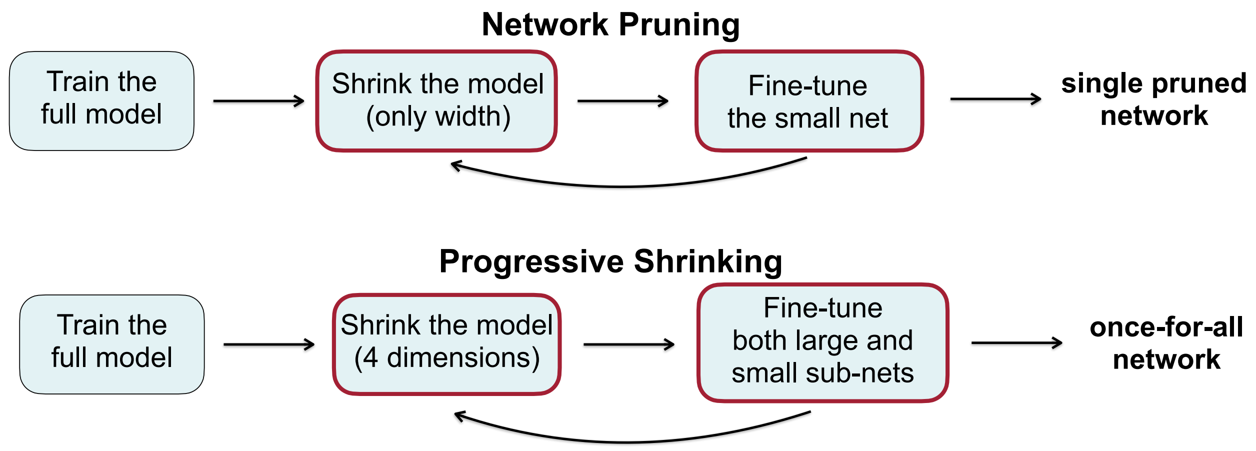


Figure 3.10: Progressive shrinking can be viewed as a generalized network pruning technique with much higher flexibility. Compared to network pruning, it shrinks more dimensions (not only width) and provides a much more powerful once-for-all network that can fit different deployment scenarios rather than a single pruned network.

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 scheme as *progressive shrinking* (PS). An example of the training process with PS is provided in Figure 3.9 and Figure 3.10, where we start with training the largest neural network with the maximum kernel size (e.g., 7), depth (e.g., 4), and width (e.g., 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 3.9. 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 [41], [143], [146]. It combines two loss terms using both the soft labels given by the largest neural network and the real labels.

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. 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. Compared to network pruning (Figure 3.10), PS also starts with training the full model, but it shrinks not only the width dimension but also the depth, kernel size, and resolution dimensions of the full model. Additionally, PS fine-tunes both large and small sub-networks rather than a single pruned network. As a result, PS provides a much more powerful once-for-all network that can fit diverse hardware platforms and efficiency constraints compared to network pruning. We describe the details of the PS training flow as follows:

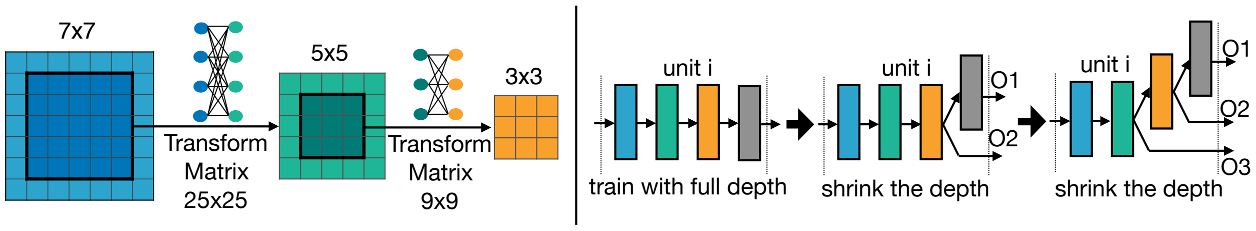


Figure 3.11: Left: Kernel transformation matrix for elastic kernel size. Right: Progressive shrinking for elastic depth. Instead of skipping each layer independently, we keep the first D layers and skip the last $(4 - D)$ layers. The weights of the early layers are shared.

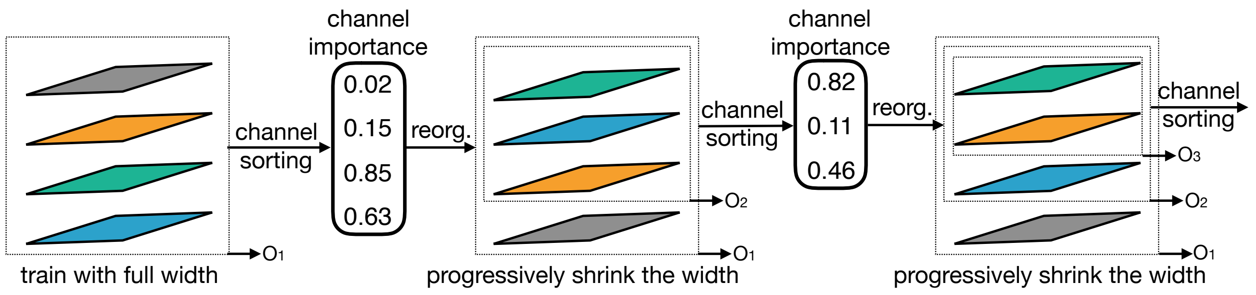


Figure 3.12: Progressive shrinking for elastic width. In this example, we progressively support 4, 3, and 2 channel settings. We perform channel sorting and pick the most important channels (with large L1 norm) to initialize the smaller channel settings. The important channels' weights are shared.

- **Elastic Kernel Size** (Figure 3.11 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 3.11 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 [10], [136]. 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 3.12). 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

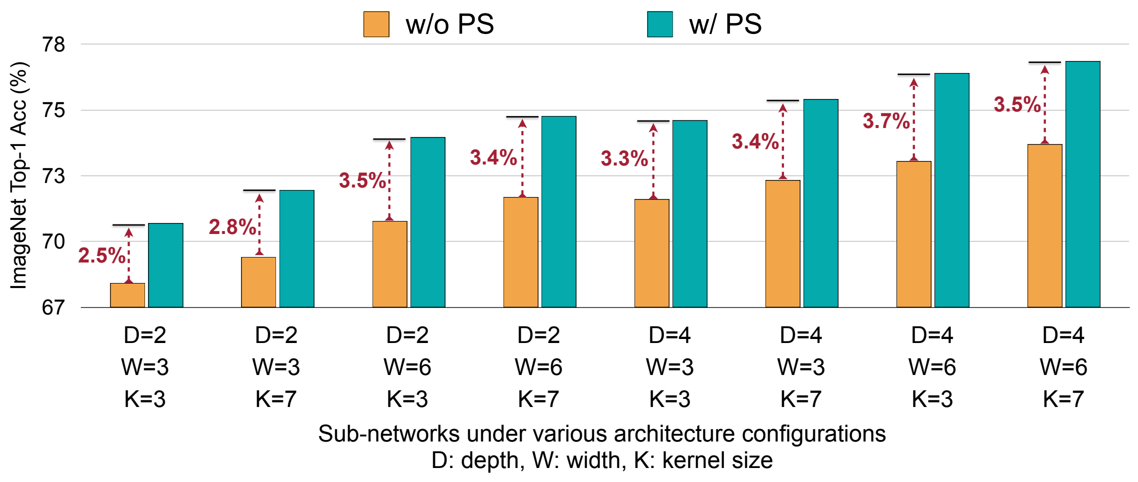


Figure 3.13: ImageNet top1 accuracy (%) performances of sub-networks under resolution 224×224. "($D = d$, $W = w$, $K = k$)" denotes a sub-network with d layers in each unit, and each layer has an width expansion ratio w and kernel size k.

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 3.12 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.

Specialized Model Deployment with once-for-all networkCap

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 neural 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. We also build a latency lookup table [10] on each target hardware platform to predict the latency. Given the target hardware and latency constraint, we conduct an evolutionary search [111] 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.

Model	ImageNet Top1 (%)	MACs	Mobile latency	Search cost (GPU hours)	Training cost (GPU hours)	Total cost ($N = 40$)		
						GPU hours	CO_2e (lbs)	AWS cost
MobileNetV2 [119]	72.0	300M	66ms	0	150N	6k	1.7k	$18.4k
MobileNetV2 #1200	73.5	300M	66ms	0	1200N	48k	13.6k	$146.9k
NASNet-A [107]	74.0	564M	-	48,000N	-	1,920k	544.5k	$5875.2k
DARTS [113]	73.1	595M	-	96N	250N	14k	4.0k	$42.8k
MnasNet [114]	74.0	317M	70ms	40,000N	-	1,600k	453.8k	$4896.0k
FBNet-C [136]	74.9	375M	-	216N	360N	23k	6.5k	$70.4k
ProxylessNAS [10]	74.6	320M	71ms	200N	300N	20k	5.7k	$61.2k
SinglePathNAS [147]	74.7	328M	-	288 + 24N	384N	17k	4.8k	$52.0k
AutoSlim [148]	74.2	305M	63ms	180	300N	12k	3.4k	$36.7k
MobileNetV3-Large [30]	75.2	219M	58ms	-	180N	7.2k	1.8k	$22.2k
OFA w/o PS	72.4	235M	59ms	40	1200	1.2k	0.34k	$3.7k
OFA w/ PS	**76.0**	230M	58ms	40	1200	1.2k	0.34k	$3.7k
OFA w/ PS #25	**76.4**	230M	58ms	40	1200 + 25N	2.2k	0.62k	$6.7k
OFA w/ PS #75	**76.9**	230M	58ms	40	1200 + 75N	4.2k	1.2k	$13.0k
OFA$_{\text{Large}}$ w/ PS #75	**80.0**	595M	-	40	1200 + 75N	4.2k	1.2k	$13.0k

Table 3.4: Comparison with SOTA hardware-aware NAS methods on Pixel1 phone. OFA decouples model training from neural 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. "CO_2e" denotes CO_2 emission which is calculated based on [131]. AWS cost is calculated based on the price of on-demand P3.16xlarge instances.

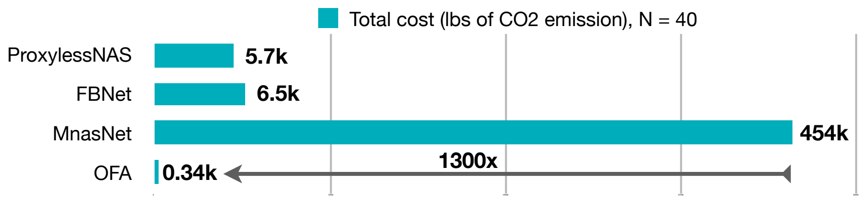


Figure 3.14: OFA saves orders of magnitude design cost compared to NAS methods.

3.2.4 Training Once-for-All Network on ImageNet

Training Details. We use the same architecture space as MobileNetV3 [30]. 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 [149] 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 3.9 to further fine-tune the full network. 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. Figure 3.13 reports the top1 accuracy of sub-networks derived from the once-for-all networks that are trained with our progressive shrinking (PS) algorithm and without PS 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. PS can improve the ImageNet accuracy of sub-networks by a significant margin under all architectural settings. Specifically, without architecture optimization, PS can achieve 74.8% top1 accuracy

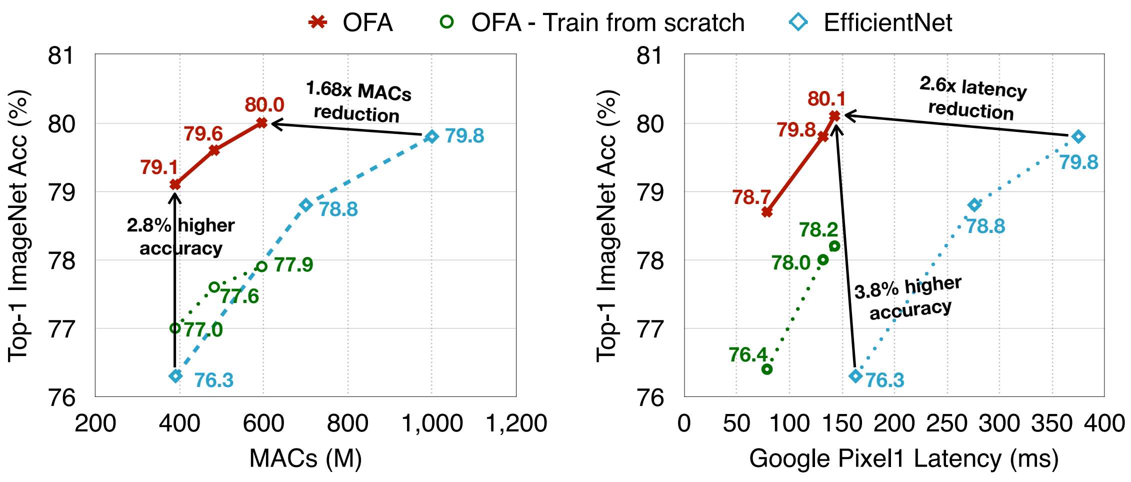


Figure 3.15: OFA achieves 80.0% top1 accuracy with 595M MACs and 80.1% top1 accuracy with 143ms Pixel1 latency, setting a new SOTA ImageNet top1 accuracy on the mobile setting.

using 226M MACs under the architecture setting (D=4, W=3, K=3), which is on par with MobileNetV3-Large. In contrast, without PS, it only achieves 71.5%, which is 3.3% lower.

3.2.5 Once-for-All Network Results 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**, including mobile phones, 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[2], batch size 1.

Comparison with NAS on Mobile Devices. Table 3.4 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 CO_2 emissions of OFA is 16× fewer than ProxylessNAS, 19× fewer than FBNet, and 1,300× fewer than MnasNet (Figure 3.14).** 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.

[2]https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html

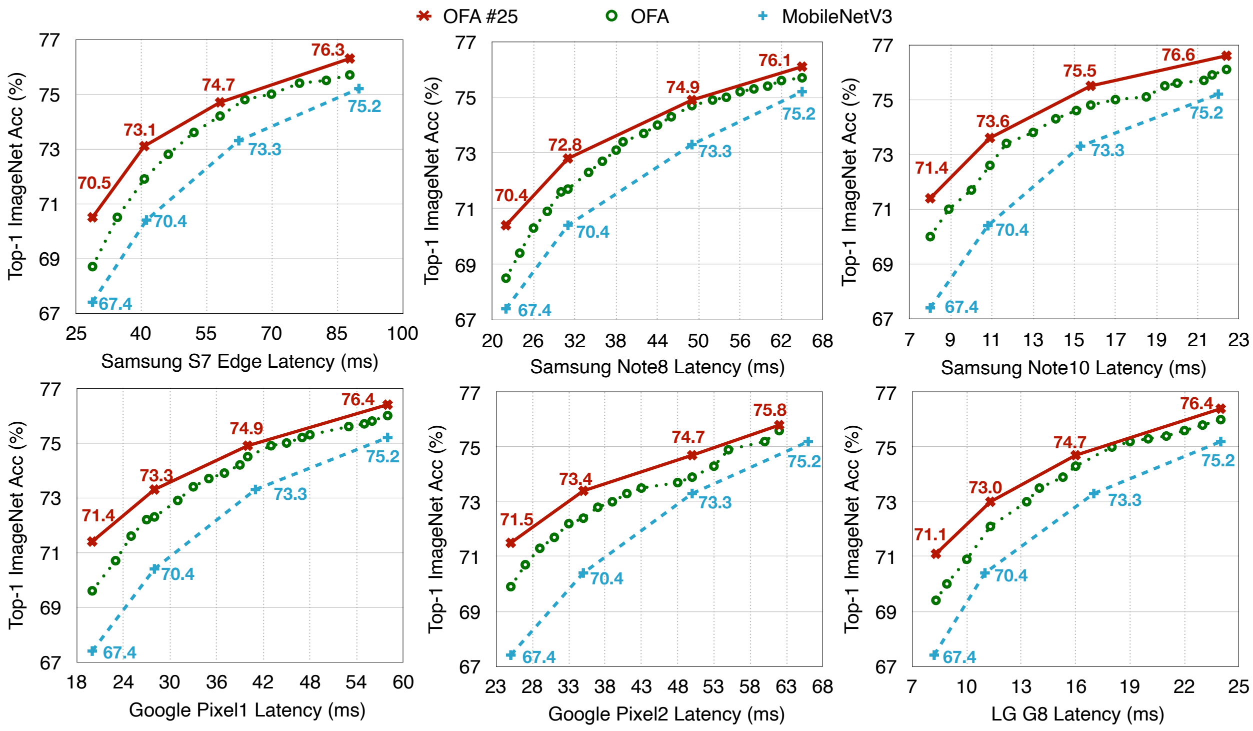


Figure 3.16: OFA consistently outperforms MobileNetV3 on mobile platforms.

OFA under Different Computational Resource Constraints. Figure 3.15 summarizes the results of OFA under different MACs and Pixel1 latency constraints. OFA achieves 79.1% ImageNet top1 accuracy with 389M MACs, being 2.8% more accurate than EfficientNet-B0 that has similar MACs. With 595M MACs, OFA reaches a new SOTA 80.0% ImageNet top1 accuracy under the mobile setting (<600M MACs), which is 0.2% higher than EfficientNet-B2 while using 1.68× fewer MACs. 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 2.6× faster than EfficientNet-B2. We also find that training the searched neural architectures from scratch cannot reach the same level of accuracy as OFA, suggesting that not only neural architectures but also pre-trained weights contribute to the superior performances of OFA.

Figure 3.16 reports detailed comparisons between OFA and MobileNetV3 on six mobile devices. Remarkably, **OFA can produce the entire trade-off curves with many points over a wide range of latency constraints by training only once** (green curve). It is impossible for previous NAS methods [10], [114] due to the prohibitive training cost.

OFA for Diverse Hardware Platforms. Besides the mobile platforms, we extensively studied the effectiveness of OFA on six additional hardware platforms (Figure 3.17) using the ProxylessNAS architecture space [10]. 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 3.20. It reveals the insight that

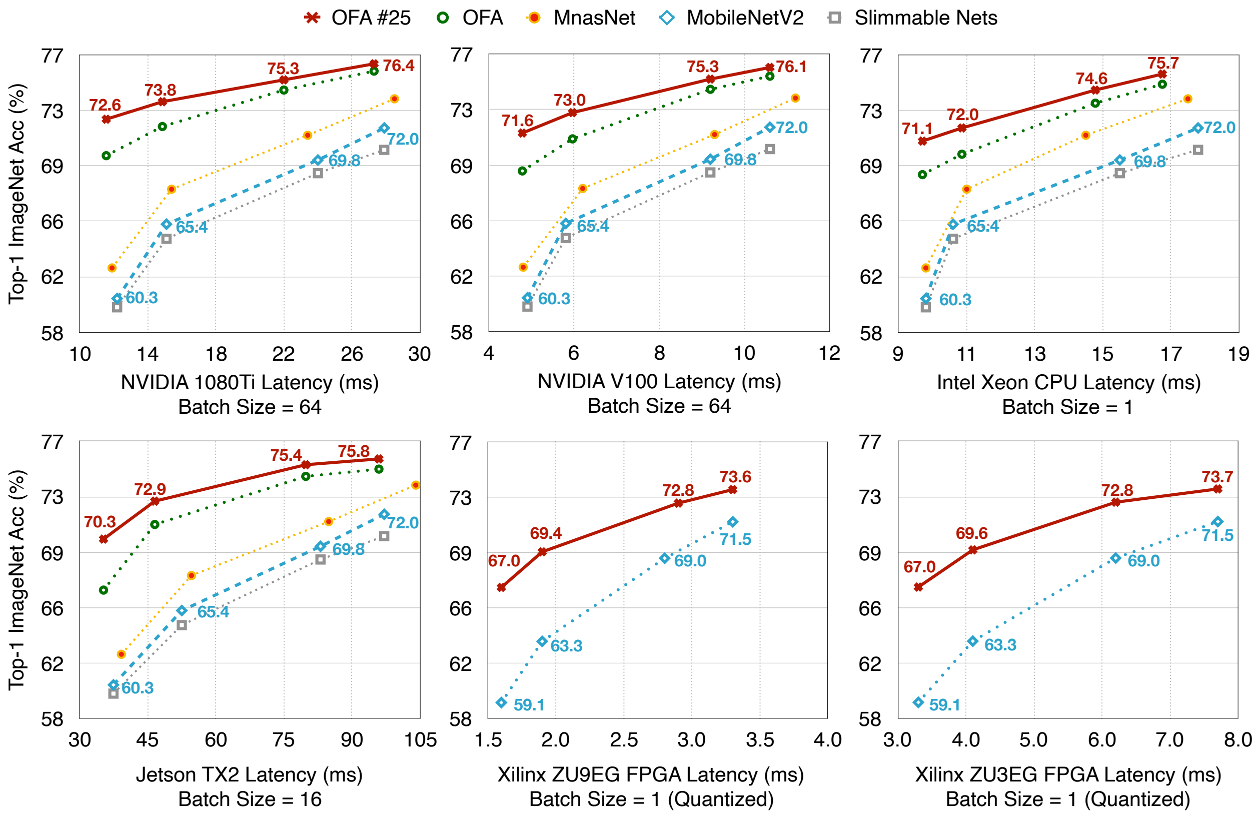


Figure 3.17: Specialized OFA models consistently achieve significantly higher ImageNet accuracy with similar latency than non-specialized neural networks on CPU, GPU, mGPU, and FPGA. More remarkably, specializing for a new hardware platform does not add training cost using OFA.

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 work 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^{19}) (Section 3.2.3), and the OFA model twin that can quickly give the accuracy/latency feedback (Section 3.2.3), the evolutionary search can automatically find a CNN architecture that has higher arithmetic intensity. As shown in Figure 3.18, OFA's arithmetic intensity is 48%/43% higher than MobileNetV2 and MnasNet (MobileNetV3 is not supported by Xilinx Vitis AI).

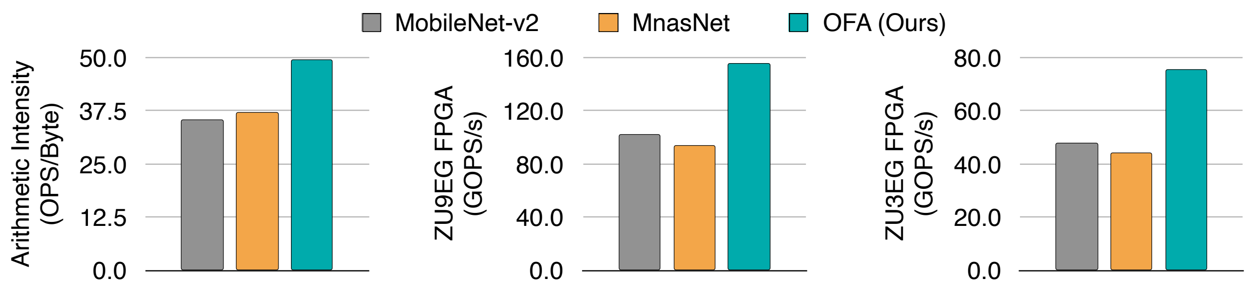


Figure 3.18: OFA models improve the arithmetic intensity (OPS/Byte) and utilization (GOPS/s) compared with the MobileNetV2 and MnasNet (measured results on Xilinx ZU9EG and ZU3EG FPGA).

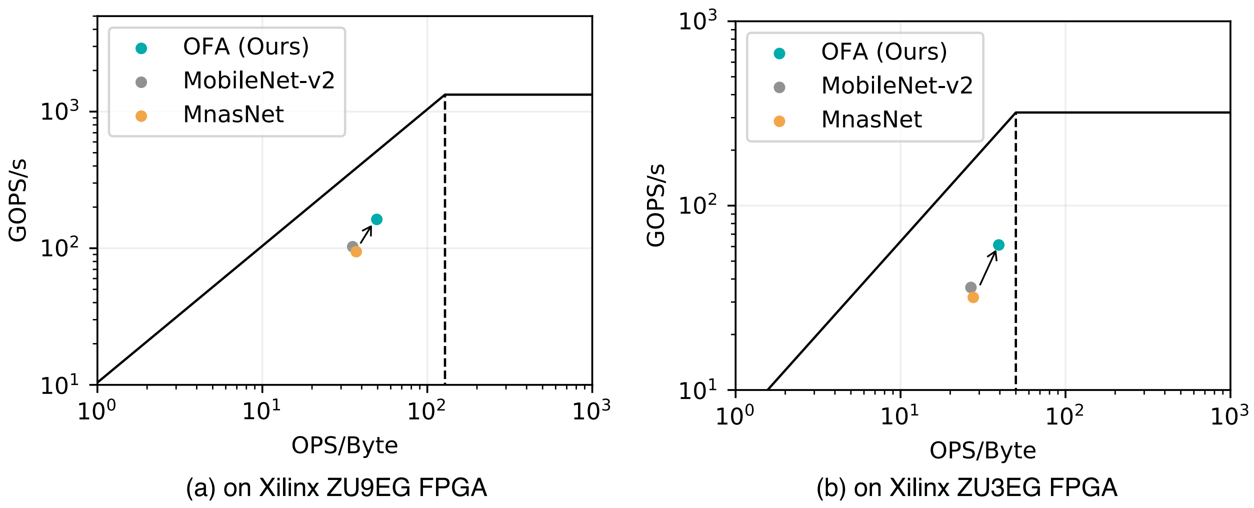
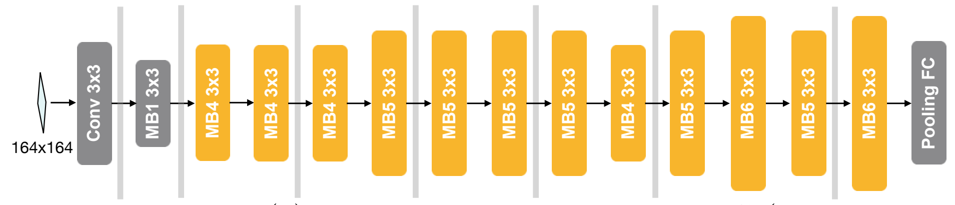


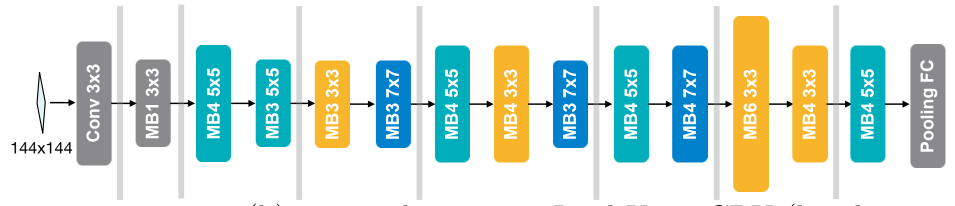
Figure 3.19: Quantative study of OFA's roofline model on Xilinx ZU9EG and ZU3EG FPGAs (log scale). OFA model increased the arithmetic intensity by 33%/43% and GOPS/s by 72%/92% on these two FPGAs compared with MnasNet.

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 [150], as shown in Figure 3.19. (70%-90% looks small in the log scale but that is significant).

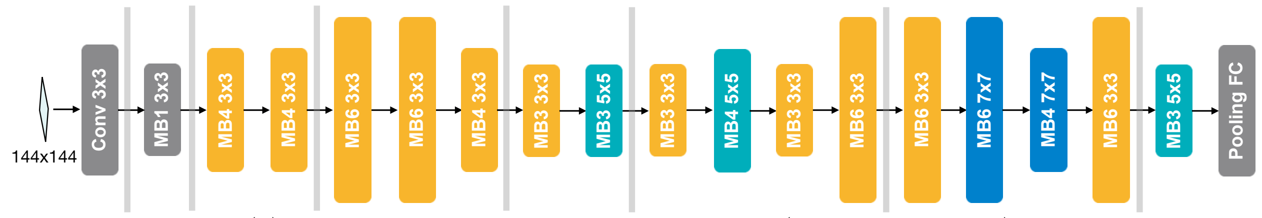
Principle 2: the CNN architecture should be co-designed with the hardware accelerator's cost model. The FPGA accelerator has a specialized depth-wise engine that is pipelined with the point-wise engine. The pipeline throughput is perfectly matched for 3x3 kernels. As a result, OFA's searched model only has 3x3 kernel (Figure 3.20, 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.



(a) 4.1ms latency on Xilinx ZU3EG (batch size = 1).



(b) 10.9ms latency on Intel Xeon CPU (batch size = 1).



(c) 14.9ms latency on NVIDIA 1080Ti (batch size = 64).

Figure 3.20: OFA can design specialized models for different hardware and different latency constraint. "MB4 3x3" 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.

3.2.6 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 CO_2 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-networks 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.

Chapter 4

Efficient Model Customization

4.1 Introduction

Intelligent edge devices with rich sensors (e.g., billions of mobile phones and IoT devices)[1] have been ubiquitous in our daily lives. These devices keep collecting *new* and *sensitive* data through the sensor every day while being expected to provide high-quality and customized services without sacrificing privacy[2]. These pose new challenges to efficient AI systems that could not only run inference but also continually fine-tune the pre-trained models on newly collected data (i.e., on-device learning).

Though on-device learning can enable many appealing applications, it is an extremely challenging problem. First, edge devices are *memory-constrained*. For example, a Raspberry Pi 1 Model A only has 256MB of memory, which is sufficient for inference, but by far insufficient for training (Figure 4.1 left), even using a lightweight neural network architecture (MobileNetV2 [119]). Furthermore, the memory is shared by various on-device applications (e.g., other deep learning models) and the operating system. A single application may only be allocated a small fraction of the total memory, which makes this challenge more critical. Second, edge devices are *energy-constrained*. DRAM access consumes two orders of magnitude more energy than on-chip SRAM access. The large memory footprint of activations cannot fit into the limited on-chip SRAM, thus has to access DRAM. For instance, the training memory of MobileNetV2, under batch size 16, is close to 1GB, which is by far larger than the SRAM size of an AMD EPYC CPU[3] (Figure 4.1 left), not to mention lower-end edge platforms. If the training memory can fit on-chip SRAM, it will drastically improve the speed and energy efficiency.

There is plenty of efficient inference techniques that reduce the number of trainable parameters and the computation FLOPs [11], [30], [35], [38], [39], [114], [119], [130], [136], [151], however, parameter-efficient or FLOPs-efficient techniques do not directly save the training memory. It is the activation that bottlenecks the training memory, not the parameters. For example, Figure 4.1 (right) compares ResNet-50 and MobileNetV2-1.4. In terms of parameter size, MobileNetV2-1.4 is 4.3× smaller than ResNet-50. However, for training activation

[1] https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/
[2] https://ec.europa.eu/info/law/law-topic/data-protection_en
[3] https://www.amd.com/en/products/cpu/amd-epyc-7302

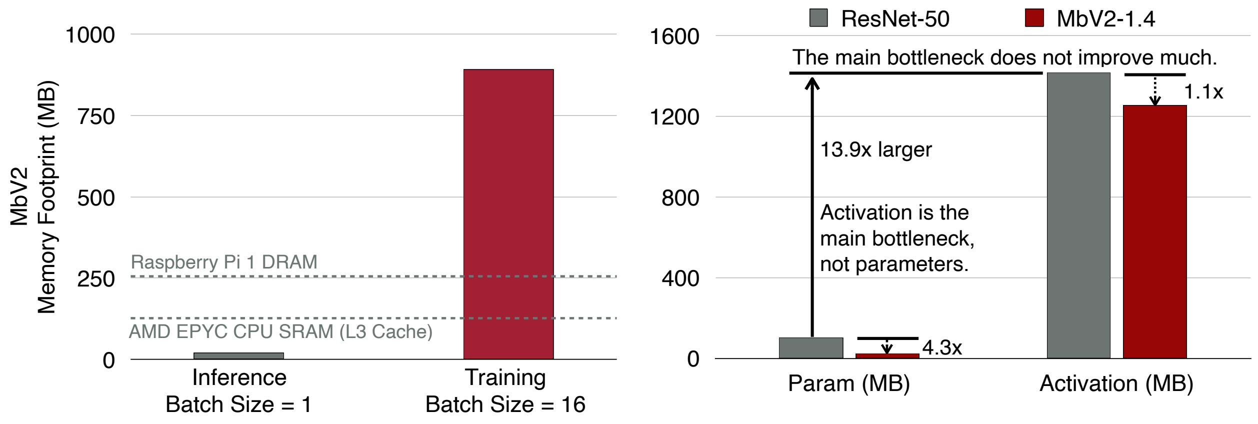


Figure 4.1: *Left*: The memory footprint required by training is much larger than inference. *Right*: Memory cost comparison between ResNet-50 and MobileNetV2-1.4 under batch size 16. Recent advances in efficient model design only reduce the size of parameters, but the activation size, which is the main bottleneck for training, does not improve much.

size, MobileNetV2-1.4 is almost the same as ResNet-50 (only 1.1× smaller), leading to little memory reduction. It is essential to reduce the size of intermediate activations required by back-propagation, which is the key memory bottleneck for efficient on-device training.

In this paper, we propose *Tiny-Transfer-Learning* (TinyTL) to address these challenges. By analyzing the memory footprint during the backward pass, we notice that the intermediate activations (the main bottleneck) are only needed when updating the weights, not the biases (Eq. 4.2). Inspired by this finding, we propose to freeze the weights of the pre-trained feature extractor and only update the biases to reduce the memory footprint (Figure 4.2b). To compensate for the capacity loss, we introduce a memory-efficient bias module, called *lite residual module*, which improves the model capacity by refining the intermediate feature maps of the feature extractor (Figure 4.2c). Meanwhile, we aggressively shrink the resolution and width of the lite residual module to have a small memory overhead (only 3.8%). Extensive experiments on 9 image classification datasets with the same pre-trained model (ProxylessNAS-Mobile [10]) demonstrate the effectiveness of TinyTL compared to previous transfer learning methods. Further, combined with a pre-trained once-for-all network [11], TinyTL can select a specialized sub-network as the feature extractor for each transfer dataset (i.e., feature extractor adaptation): given a more difficult dataset, a larger sub-network is selected, and vice versa. TinyTL achieves the same level of (or even higher) accuracy compared to fine-tuning the full Inception-V3 while reducing the training memory footprint by up to **12.9**×.

4.2 Tiny Transfer Learning

4.2.1 Understanding the Memory Footprint of Back-propagation

Without loss of generality, we consider a neural network $\mathcal{M}$ that consists of a sequence of layers:

$$\mathcal{M}(\cdot) = \mathcal{F}_{\mathbf{w}_n}(\mathcal{F}_{\mathbf{w}_{n-1}}(\cdots \mathcal{F}_{\mathbf{w}_2}(\mathcal{F}_{\mathbf{w}_1}(\cdot)) \cdots)), \tag{4.1}$$

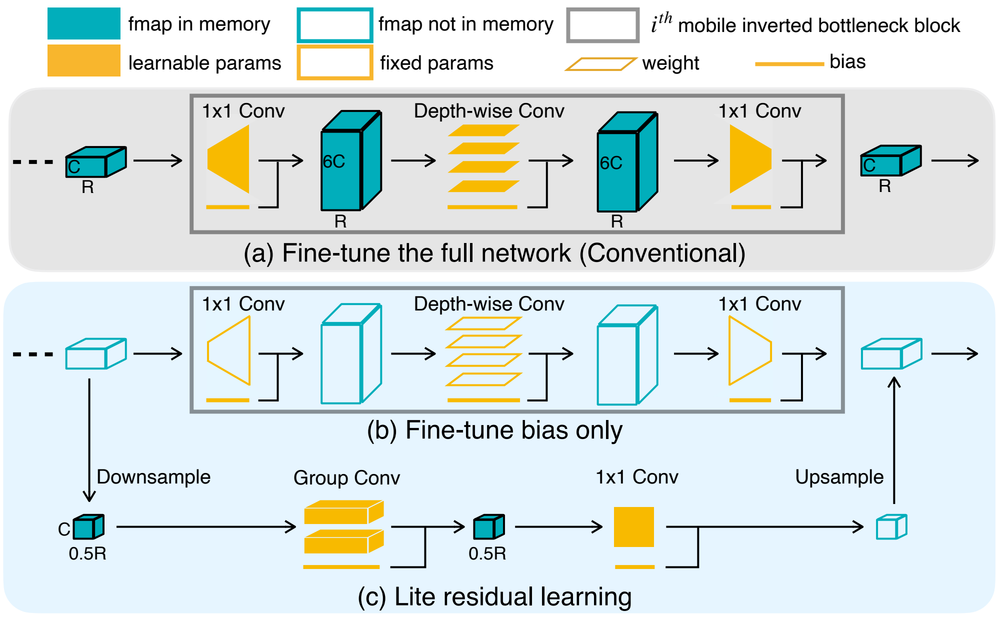


Figure 4.2: TinyTL overview ("C" denotes the width and "R" denote the resolution). Conventional transfer learning relies on fine-tuning the weights to adapt the model (Fig.a), which requires a large amount of activation memory (in blue) for back-propagation. TinyTL reduces the memory usage by fixing the weights (Fig.b) while only fine-tuning the bias. (Fig.c) exploit *lite residual learning* to compensate for the capacity loss, using group convolution and avoiding inverted bottleneck to achieve high arithmetic intensity and small memory footprint. The skip connection remains unchanged (omitted for simplicity).

where $\mathbf{w}_i$ denotes the parameters of the i^{th} layer. Let $\mathbf{a}_i$ and $\mathbf{a}_{i+1}$ be the input and output activations of the i^{th} layer, respectively, and $\mathcal{L}$ be the loss. In the backward pass, given $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}$, there are two goals for the i^{th} layer: computing $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i}$.

Assuming the i^{th} layer is a linear layer whose forward process is given as: $\mathbf{a}_{i+1} = \mathbf{a}_i\mathbf{W} + \mathbf{b}$, then its backward process under batch size 1 is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \frac{\partial \mathbf{a}_{i+1}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \mathbf{W}^T, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{a}_i^T \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}}. \tag{4.2}$$

According to Eq. (4.2), the intermediate activations (i.e., $\{\mathbf{a}_i\}$) that dominate the memory footprint are only required to compute the gradient of the weights (i.e., $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$), not the bias. If we only update the bias, training memory can be greatly saved. This property is also applicable to convolution layers and normalization layers (e.g., batch normalization [152], group normalization[153], etc) since they can be considered as special types of linear layers.

Regarding non-linear activation layers (e.g., ReLU, sigmoid, h-swish), sigmoid and h-swish require to store $\mathbf{a}_i$ to compute $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i}$ (Table 4.1), hence they are not memory-efficient. Activation layers that build upon them are also not memory-efficient consequently, such as tanh, swish [154], etc. In contrast, ReLU and other ReLU-styled activation layers (e.g., LeakyReLU [155]) only requires to store a binary mask representing whether the value is smaller than 0, which is 32× smaller than storing $\mathbf{a}_i$.

Table 4.1: Detailed forward and backward processes of non-linear activation layers. $|\mathbf{a}_i|$ denotes the number of elements of $\mathbf{a}_i$. "$\circ$" denotes the element-wise product. $(\mathbf{1}_{\mathbf{a}_i \geq 0})_j = 0$ if $(\mathbf{a}_i)_j < 0$ and $(\mathbf{1}_{\mathbf{a}_i \geq 0})_j = 1$ otherwise. $\text{ReLU6}(\mathbf{a}_i) = \min(6, \max(0, \mathbf{a}_i))$.

Layer Type	Forward	Backward	Memory Cost
ReLU	$\mathbf{a}_{i+1} = \max(0, \mathbf{a}_i)$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ \mathbf{1}_{\mathbf{a}_i \geq 0}$	$\|\mathbf{a}_i\|$ bits
sigmoid	$\mathbf{a}_{i+1} = \sigma(\mathbf{a}_i) = \frac{1}{1+\exp(-\mathbf{a}_i)}$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ \sigma(\mathbf{a}_i) \circ (1 - \sigma(\mathbf{a}_i))$	32 $\|\mathbf{a}_i\|$ bits
h-swish [30]	$\mathbf{a}_{i+1} = \mathbf{a}_i \circ \frac{\text{ReLU6}(\mathbf{a}_i+3)}{6}$	$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_{i+1}} \circ (\frac{\text{ReLU6}(\mathbf{a}_i+3)}{6} + \mathbf{a}_i \circ \frac{\mathbf{1}_{-3 \leq \mathbf{a}_i \leq 3}}{6})$	32 $\|\mathbf{a}_i\|$ bits

4.2.2 Lite Residual Learning

Based on the memory footprint analysis, one possible solution of reducing the memory cost is to freeze the weights of the pre-trained feature extractor while only update the biases (Figure 4.2b). However, only updating biases has limited adaptation capacity. Therefore, we introduce lite residual learning that exploits a new class of generalized memory-efficient bias modules to refine the intermediate feature maps (Figure 4.2c).

Formally, a layer with frozen weights and learnable biases can be represented as:

$$\mathbf{a}_{i+1} = \mathcal{F}_{\mathbf{W}}(\mathbf{a}_i) + \mathbf{b}. \tag{4.3}$$

To improve the model capacity while keeping a small memory footprint, we propose to add a lite residual module that generates a residual feature map to refine the output:

$$\mathbf{a}_{i+1} = \mathcal{F}_{\mathbf{W}}(\mathbf{a}_i) + \mathbf{b} + \mathcal{F}_{\mathbf{w}_r}(\mathbf{a}'_i = \text{reduce}(\mathbf{a}_i)), \tag{4.4}$$

where $\mathbf{a}'_i = \text{reduce}(\mathbf{a}_i)$ is the reduced activation. According to Eq. (4.2), learning these lite residual modules only requires to store the reduced activations $\{\mathbf{a}'_i\}$ rather than the full activations $\{\mathbf{a}_i\}$.

Implementation (Figure 4.2c). We apply Eq. (4.4) to mobile inverted bottleneck blocks (MB-block) [119]. The key principle is to keep the activation small. Following this principle, we explore two design dimensions to reduce the activation size:

- **Width.** The widely-used inverted bottleneck requires a huge number of channels (6×) to compensate for the small capacity of a depthwise convolution, which is parameter-efficient but highly activation-inefficient. Even worse, converting 1× channels to 6× channels back and forth requires two 1×1 projection layers, which doubles the total activation to 12×. Depthwise convolution also has a very low arithmetic intensity (its OPs/Byte is less than 4% of 1×1 convolution's OPs/Byte if with 256 channels), thus highly memory in-efficient with little reuse. To solve these limitations, our lite residual module employs the group convolution that has much higher arithmetic intensity than depthwise convolution, providing a good trade-off between FLOPs and memory. That also removes the 1×1 projection layer, reducing the total channel number by $\frac{6\times2+1}{1+1} = 6.5\times$.

- **Resolution.** The activation size grows quadratically with the resolution. Therefore, we shrink the resolution in the lite residual module by employing a 2×2 average pooling to downsample the input feature map. The output of the lite residual module is then upsampled to match the size of the main branch's output feature map via bilinear upsampling. Combining resolution and width optimizations, the activation of our lite residual module is roughly $2^2 \times 6.5 = 26\times$ smaller than the inverted bottleneck.

4.2.3 Discussions

Normalization Layers. As discussed in Section 4.2.1, TinyTL flexibly supports different normalization layers, including batch normalization (BN), group normalization (GN), layer normalization (LN), and so on. In particular, BN is the most widely used one in vision tasks. However, BN requires a large batch size to have accurate running statistics estimation during training, which is not suitable for on-device learning where we want a small training batch size to reduce the memory footprint. Moreover, the data may come in a streaming fashion in on-device learning, which requires a training batch size of 1. In contrast to BN, GN can handle a small training batch size as the running statistics in GN are computed independently for different inputs. In our experiments, GN with a small training batch size (e.g., 8) performs slightly worse than BN with a large training batch size (e.g., 256). However, as we target at on-device learning, we choose GN in our models.

Feature Extractor Adaptation. TinyTL can be applied to different backbone neural networks, such as MobileNetV2 [119], ProxylessNASNets [10], EfficientNets [29], etc. However, since the weights of the feature extractor are frozen in TinyTL, we find using the same backbone neural network for all transfer tasks is sub-optimal. Therefore, we choose the backbone of TinyTL using a pre-trained once-for-all network [11] to adaptively select the specialized feature extractor that best fits the target transfer dataset. Specifically, a once-for-all network is a special kind of neural network that is sparsely activated, from which many different sub-networks can be derived without retraining by sparsely activating parts of the model according to the architecture configuration (i.e., depth, width, kernel size, resolution), while the weights are shared. This allows us to efficiently evaluate the effectiveness of a backbone neural network on the target transfer dataset without the expensive pre-training process.

4.3 Experiments

4.3.1 Setups

Datasets. Following the common practice [156]–[158], we use ImageNet [61] as the pre-training dataset, and then transfer the models to 8 downstream object classification tasks, including Cars [159], Flowers [160], Aircraft [161], CUB [162], Pets [163], Food [164], CIFAR10 [165], and CIFAR100 [165]. Besides object classification, we also evaluate our TinyTL on human facial attribute classification tasks, where CelebA [166] is the transfer dataset and VGGFace2 [167] is the pre-training dataset.

Table 4.2: Comparison between TinyTL and conventional transfer learning methods (training memory footprint is calculated assuming the batch size is 8 and the classifier head for Flowers is used). For object classification datasets, we report the top1 accuracy (%) while for CelebA we report the average top1 accuracy (%) over 40 facial attribute classification tasks. 'B' represents Bias while 'L' represents LiteResidual. *FT-Last* represents only the last layer is fine-tuned. *FT-Norm+Last* represents normalization layers and the last layer are fine-tuned. *FT-Full* represents the full network is fine-tuned. The backbone neural network is ProxylessNAS-Mobile, and the resolution is 224 except for 'TinyTL-L+B@320' whose resolution is 320. TinyTL consistently outperforms *FT-Last* and *FT-Norm+Last* by a large margin with a similar or lower training memory footprint. By increasing the resolution to 320, TinyTL can reach the same level of accuracy as *FT-Full* while being 6× memory efficient.

Method	Train. Mem.	Flowers	Cars	CUB	Food	Pets	Aircraft	CIFAR10	CIFAR100	CelebA
FT-Last	**31MB**	90.1	50.9	73.3	68.7	91.3	44.9	85.9	68.8	88.7
TinyTL-B	**32MB**	93.5	73.4	75.3	75.5	92.1	63.2	93.7	78.8	90.4
TinyTL-L	**37MB**	95.3	84.2	76.8	79.2	91.7	76.4	96.1	80.9	91.2
TinyTL-L+B	**37MB**	95.5	85.0	77.1	79.7	91.8	75.4	95.9	81.4	91.2
TinyTL-L+B@320	65MB	96.8	88.8	81.0	82.9	92.9	82.3	96.1	81.5	-
FT-Norm+Last	192MB	94.3	77.9	76.3	77.0	92.2	68.1	94.8	80.2	90.4
FT-Full	391MB	96.8	90.2	81.0	84.6	93.0	86.0	97.1	84.1	91.4

Model Architecture. To justify the effectiveness of TinyTL, we first apply TinyTL and previous transfer learning methods to the same backbone neural network, ProxylessNAS-Mobile [10]. For each MB-block in ProxylessNAS-Mobile, we insert a lite residual module as described in Section 4.2.2 and Figure 4.2 (c). The group number is 2, and the kernel size is 5. We use the ReLU activation since it is more memory-efficient according to Section 4.2.1. We replace all BN layers with GN layers to better support small training batch sizes. We set the number of channels per group to 8 for all GN layers. Following [168], we apply weight standardization [169] to convolution layers that are followed by GN.

For feature extractor adaptation, we build the once-for-all network using the MobileNetV2 design space [10], [11] that contains five stages with a gradually decreased resolution, and each stage consists of a sequence of MB-blocks. In the stage-level, it supports elastic depth (i.e., 2, 3, 4). In the block-level, it supports elastic kernel size (i.e., 3, 5, 7) and elastic width expansion ratio (i.e., 3, 4, 6). Similarly, for each MB-block in the once-for-all network, we insert a lite residual module that supports elastic group number (i.e., 2, 4) and elastic kernel size (i.e., 3, 5).

Training Details. We freeze the memory-heavy modules (weights of the feature extractor) and only update memory-efficient modules (bias, lite residual, classifier head) during transfer learning. The models are fine-tuned for 50 epochs using the Adam optimizer [170] with batch size 8 on a single GPU. The initial learning rate is tuned for each dataset while cosine schedule [149] is adopted for learning rate decay. We apply 8bits weight quantization [38] on the frozen weights to reduce the parameter size, which causes a negligible accuracy drop in our experiments. For all compared methods, we also assume the 8bits weight quantization is applied if eligible when calculating their training memory footprint. Additionally, as PyTorch does not support explicit fine-grained memory management, we use the theoretically calculated training memory footprint for comparison in our experiments. For simplicity, we assume the batch size is 8 for all compared methods throughout the experiment section.

4.3.2 Main Results

Effectiveness of TinyTL. Table 4.2 reports the comparison between TinyTL and previous transfer learning methods including: i) fine-tuning the last linear layer [171]–[173] (referred to as *FT-Last*); ii) fine-tuning the normalization layers (e.g., BN, GN) and the last linear layer [174] (referred to as *FT-Norm+Last*) ; iii) fine-tuning the full network [156], [158] (referred to as *FT-Full*). We also study several variants of TinyTL including: i) TinyTL-B that fine-tunes biases and the last linear layer; ii) TinyTL-L that fine-tunes lite residual modules and the last linear layer; iii) TinyTL-L+B that fine-tunes lite residual modules, biases, and the last linear layer. All compared methods use the same pre-trained model but fine-tune different parts of the model as discussed above. We report the average accuracy across five runs.

Compared to *FT-Last*, TinyTL maintains a similar training memory footprint while improving the top1 accuracy by a significant margin. In particular, TinyTL-L+B improves the top1 accuracy by **34.1%** on Cars, by **30.5%** on Aircraft, by **12.6%** on CIFAR100, by **11.0%** on Food, etc. It shows the improved adaptation capacity of our method over *FT-Last*. Compared to *FT-Norm+Last*, TinyTL-L+B improves the training memory efficiency by **5.2**× while providing up to **7.3%** higher top1 accuracy, which shows that our method is not only more memory-efficient but also more effective than *FT-Norm+Last*. Compared to *FT-Full*, TinyTL-L+B@320 can achieve the same level of accuracy while providing **6.0**× training memory saving.

Regarding the comparison between different variants of TinyTL, both TinyTL-L and TinyTL-L+B have clearly better accuracy than TinyTL-B while incurring little memory overhead. It shows that the lite residual modules are essential in TinyTL. Besides, we find that TinyTL-L+B is slightly better than TinyTL-L on most of the datasets while maintaining the same memory footprint. Therefore, we choose TinyTL-L+B as the default.

Figure 4.3 demonstrates the results under different input resolutions. We can observe that simply reducing the input resolution will result in significant accuracy drops for *FT-Full*. In contrast, TinyTL can reduce the memory footprint by **3.9-6.5**× while having the same or even higher accuracy compared to fine-tuning the full network.

Combining TinyTL and Feature Extractor Adaptation. Table 4.3 summarizes the results of TinyTL and previously reported transfer learning results, where different backbone neural networks are used as the feature extractor. Combined with feature extractor adaptation, TinyTL achieves **7.5-12.9**× memory saving compared to fine-tuning the full Inception-V3, reducing from 850MB to 66-114MB while providing the same level of accuracy. Additionally, we try updating the last two layers besides biases and lite residual modules (indicated by †), which results in 2MB of extra training memory footprint. This slightly improves the accuracy performances, from 90.7% to 91.5% on Cars, from 85.0% to 86.0% on Food, and from 84.8% to 85.4% on Aircraft.

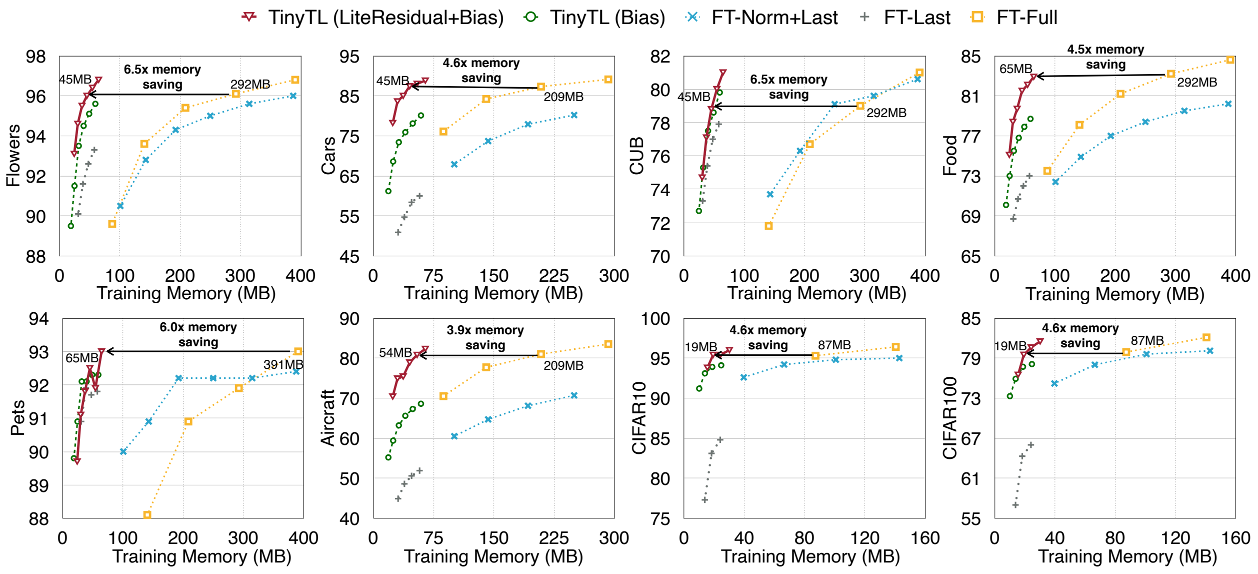


Figure 4.3: Top1 accuracy results of different transfer learning methods under varied resolutions using the same pre-trained neural network (ProxylessNAS-Mobile). With the same level of accuracy, TinyTL achieves 3.9-6.5× memory saving compared to fine-tuning the full network.

4.3.3 Ablation Studies and Discussions

Comparison with Dynamic Activation Pruning. The comparison between TinyTL and dynamic activation pruning [175] is summarized in Figure 4.4. TinyTL is more effective because it re-designed the transfer learning framework (lite residual module, feature extractor adaptation) rather than prune an existing architecture. The transfer accuracy drops quickly when the pruning ratio increases beyond 50% (only 2× memory saving). In contrast, TinyTL can achieve much higher memory reduction without loss of accuracy.

Initialization for Lite Residual Modules. By default, we use the pre-trained weights on the pre-training dataset to initialize the lite residual modules. It requires to have lite residual modules during both the pre-training phase and transfer learning phase. When applying TinyTL to existing pre-trained neural networks that do not have lite residual modules during the pre-training phase, we need to use another initialization strategy for the lite residual modules during transfer learning. To verify the effectiveness of TinyTL under this setting, we also evaluate the performances of TinyTL when using random weights [129] to initialize the lite residual modules except for the scaling parameter of the final normalization layer in each lite residual module. These scaling parameters are initialized with zeros.

Table 4.4 reports the summarized results. We find using the pre-trained weights to initialize the lite residual modules consistently outperforms using random weights. Besides, we also find that using TinyTL-RandomL+B still provides highly competitive results on Cars, Food, Aircraft, CIFAR10, CIFAR100, and CelebA. Therefore, if having the budget, it is better to use pre-trained weights to initialize the lite residual modules. If not, TinyTL can still be applied and provides competitive results on datasets whose distribution is far from

Table 4.3: Comparison with previous transfer learning results under different backbone neural networks. 'I-V3' is Inception-V3; 'N-A' is NASNet-A Mobile; 'M2-1.4' is MobileNetV2-1.4; 'R-50' is ResNet-50; 'PM' is ProxylessNAS-Mobile; 'FA' represents feature extractor adaptation. † indicates the last two layers are updated besides biases and lite residual modules in TinyTL. TinyTL+FA reduces the training memory by **7.5-12.9**$\times$ without sacrificing accuracy compared to fine-tuning the widely used Inception-V3.

Method	Net	Train. mem.	Reduce ratio	Flowers	Cars	CUB	Food	Pets	Aircraft	CIFAR10	CIFAR100
FT-Full	I-V3 [156]	850MB	1.0×	96.3	91.3	82.8	88.7	-	85.5	-	-
	R-50 [158]	802MB	1.1×	97.5	91.7	-	87.8	92.5	86.6	96.8	84.5
	M2-1.4 [158]	644MB	1.3×	97.5	91.8	-	87.7	91.0	86.8	96.1	82.5
	N-A [158]	566MB	1.5×	96.8	88.5	-	85.5	89.4	72.8	96.8	83.9
FT-Norm+Last	I-V3 [174]	326MB	2.6×	90.4	81.0	-	-	-	70.7	-	-
FT-Last	I-V3 [174]	94MB	9.0×	84.5	55.0	-	-	-	45.9	-	-
TinyTL	PM@320	**65MB**	**13.1**×	96.8	88.8	81.0	82.9	92.9	82.3	96.1	81.5
	FA@256	**66MB**	**12.9**×	96.8	89.6	80.8	83.4	93.0	82.4	96.8	82.7
	FA@352	114MB	7.5×	97.4	90.7	82.4	85.0	93.4	84.8	-	-
	FA@352†	116MB	7.3×	-	91.5	-	86.0	-	85.4	-	-

Table 4.4: Results of TinyTL under different initialization strategies for lite residual modules. TinyTL-L+B adds lite residual modules starting from the pre-training phase and uses the pre-trained weights to initialize the lite residual modules during transfer learning. In contrast, TinyTL-RandomL+B uses random weights to initialize the lite residual modules. Using random weights for initialization hurts the performances of TinyTL. But on datasets whose distribution is far from the pre-training dataset, TinyTL-RandomL+B still provides competitive results.

Method	Train. Mem.	Flowers	Cars	CUB	Food	Pets	Aircraft	CIFAR10	CIFAR100	CelebA
FT-Last	**31MB**	90.1	50.9	73.3	68.7	91.3	44.9	85.9	68.8	88.7
TinyTL-RandomL+B	**37MB**	88.0	82.4	72.9	79.3	84.3	73.6	95.7	81.4	91.2
TinyTL-L+B	**37MB**	95.5	85.0	77.1	79.7	91.8	75.4	95.9	81.4	91.2
FT-Norm+Last	192MB	94.3	77.9	76.3	77.0	92.2	68.1	94.8	80.2	90.4
FT-Full	391MB	96.8	90.2	81.0	84.6	93.0	86.0	97.1	84.1	91.4

the pre-training dataset.

Results of TinyTL under Batch Size 1. Figure 4.5 demonstrates the results of TinyTL when using a training batch size of 1. We tune the initial learning rate for each dataset while keeping the other training settings unchanged. As our model employs group normalization rather than batch normalization (Section 4.2.3), we observe little/no loss of accuracy than training with batch size 8. Meanwhile, the training memory footprint is further reduced to around 16MB, a typical L3 cache size. This makes it much easier to train on the cache (SRAM), which can greatly reduce energy consumption than DRAM training.

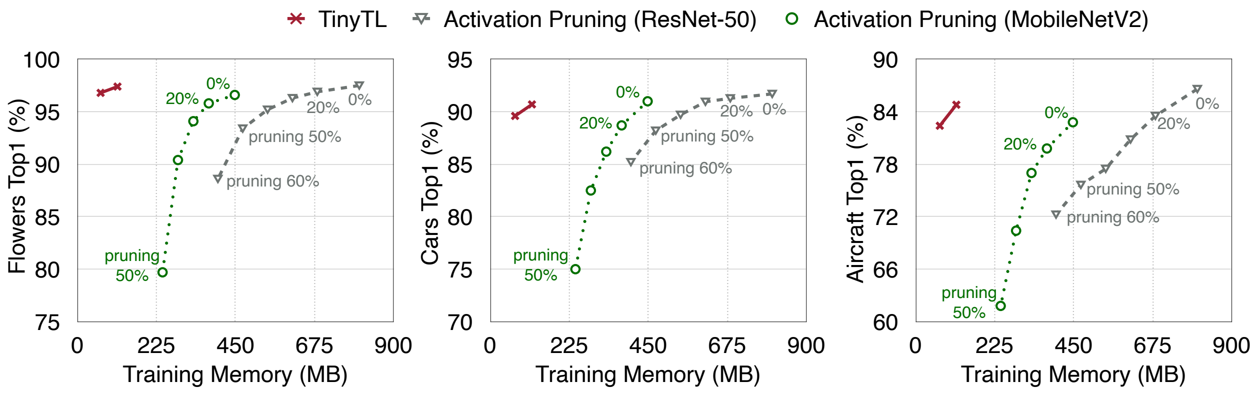


Figure 4.4: Compared with the dynamic activation pruning [175], TinyTL saves the memory more effectively.

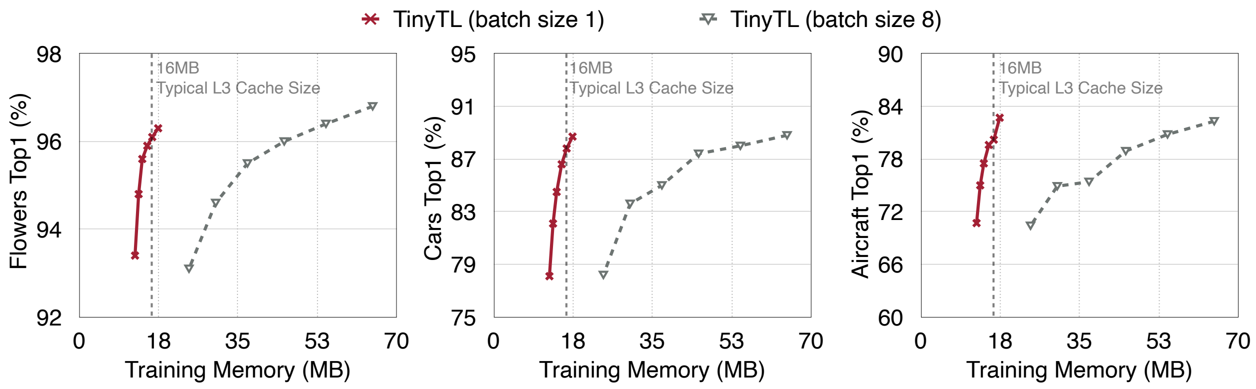


Figure 4.5: Results of TinyTL when trained with batch size 1. It further reduces the training memory footprint to around 16MB (typical L3 cache size), making it possible to train on the cache (SRAM) instead of DRAM.

4.4 Conclusion

This chapter presented Tiny-Transfer-Learning (TinyTL) for memory-efficient on-device learning that aims to adapt pre-trained models to newly collected data on edge devices. Unlike previous methods that focus on reducing the number of parameters or FLOPs, TinyTL directly optimizes the training memory footprint by fixing the memory-heavy modules (i.e., weights) while learning memory-efficient bias modules. We further introduce lite residual modules that significantly improve the adaptation capacity of the model with little memory overhead. Extensive experiments on benchmark datasets consistently show the effectiveness and memory-efficiency of TinyTL, paving the way for efficient on-device machine learning.

The proposed efficient on-device learning technique greatly reduces the training memory footprint of deep neural networks, enabling adapting pre-trained models to new data locally on edge devices without leaking them to the cloud. It can democratize AI to people in the rural areas where the Internet is unavailable or the network condition is poor. They can not only inference but also fine-tune AI models on their local devices without connections to the cloud servers. This can also benefit privacy-sensitive AI applications, such as health care, smart home, and so on.

Chapter 5

Conclusion

Deep neural networks have demonstrated remarkable performances in many areas, such as computer vision, natural language processing, speech recognition, etc. In particular, recent large foundation models, such as GPT and diffusion models, have shown an astounding capacity for generating high-quality content and tackling zero-shot or few-shot learning tasks. However, the performance breakthroughs come at the cost of significantly increased computational and memory costs. It makes deploying these deep learning models on real-world applications challenging and costly. To make them more accessible and reduce the serving cost of these models, it is crucial to investigate techniques for improving their efficiency on hardware while maintaining their remarkable performances. In this dissertation, we focus on tackling this challenge from three perspectives: efficient representation learning, hardware-aware acceleration, and efficient model customization. Extensive experiments on diverse tasks and hardware platforms demonstrate the effectiveness of our approach.

5.1 Impact

My research interests lie in machine learning, particularly efficient foundation models (diffusion models, LLMs, etc), EdgeAI, and AutoML, resulting in multiple impactful publications across leading conferences in machine learning (ICLR, ICML, NeurIPS), computer vision (ICCV, CVPR), and natural language processing (ACL). In particular, my works on hardware-aware neural architecture search (ProxylessNAS and Once-for-All) have received 2062 and 1272 citations and have been listed as top influential papers in ICLR 2019 and ICLR 2020[1]. My research has been honored by the Qualcomm Innovation Fellowship. In addition, I have been awarded 1st Place in multiple prestigious competitions, including 2020 IEEE Low-Power Computer Vision Challenge, 2019 IEEE Low-Power Image Recognition Challenge, and Low-Power Computer Vision Workshop at ICCV 2019.

My research also has significant industry impacts. My research in hardware-aware model optimization has landed in many industry projects, such as Meta PytorchHub, AWS AutoGluon, Microsoft NNI, Sony nnabla, etc. It has also been commercialized by OmniML (acquired by NVIDIA) and has generated real-world revenue. My research in EfficientViT has

[1]https://www.paperdigest.org/2021/05/most-influential-iclr-papers-2021-05/

been used in NVIDIA Picasso, AMD Low-Level Vision, NVIDIA Generative AI, Huggingface Pytorch Image Models, etc.

References

[1] OpenAI, "Gpt-4 technical report," *ArXiv*, vol. abs/2303.08774, 2023. URL: https://api.semanticscholar.org/CorpusID:257532815.

[2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[3] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10 684–10 695.

[4] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, *et al.*, "Segment anything," *arXiv preprint arXiv:2304.02643*, 2023.

[5] T. Brooks, B. Peebles, C. Holmes, *et al.*, "Video generation models as world simulators," 2024. URL: https://openai.com/research/video-generation-models-as-world-simulators.

[6] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," in *Advances in neural information processing systems*, vol. 33, 2020, pp. 1877–1901.

[8] H. Cai, J. Li, M. Hu, C. Gan, and S. Han, "Efficientvit: Lightweight multi-scale attention for high-resolution dense prediction," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 17 302–17 313.

[9] H. Cai, M. Li, Z. Zhang, Q. Zhang, M.-Y. Liu, and S. Han, "Condition-aware neural network for controlled image generation," in *CVPR*, 2024.

[10] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *ICLR*, 2019. URL: https://arxiv.org/pdf/1812.00332.pdf.

[11] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once for all: Train one network and specialize it for efficient deployment," in *ICLR*, 2020. URL: https://arxiv.org/pdf/1908.09791.pdf.

[12] H. Cai, C. Gan, L. Zhu, and S. Han, "Tinytl: Reduce memory, not parameters for efficient on-device learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 285–11 297, 2020.

[13] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[14] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, Springer, 2015, pp. 234–241.

[15] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[16] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2881–2890.

[17] Y. Yuan, X. Chen, and J. Wang, "Object-contextual representations for semantic segmentation," in *European conference on computer vision*, Springer, 2020, pp. 173–190.

[18] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, *et al.*, "Deep high-resolution representation learning for visual recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, no. 10, pp. 3349–3364, 2020.

[19] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, "Segformer: Simple and efficient design for semantic segmentation with transformers," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017.

[21] M.-H. Guo, C.-Z. Lu, Q. Hou, Z.-N. Liu, M.-M. Cheng, and S.-m. Hu, "Segnext: Rethinking convolutional attention design for semantic segmentation," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022. URL: https://openreview.net/forum?id=VgOw1pUPh97.

[22] X. Ding, X. Zhang, Y. Zhou, J. Han, G. Ding, and J. Sun, "Scaling up your kernels to 31x31: Revisiting large kernel design in cnns," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2022.

[23] Y. Wang, M. Li, H. Cai, W.-M. Chen, and S. Han, "Lite pose: Efficient architecture design for 2d human pose estimation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2022.

[24] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International Conference on Machine Learning*, PMLR, 2020, pp. 5156–5165.

[25] H. Zhao, X. Qi, X. Shen, J. Shi, and J. Jia, "Icnet for real-time semantic segmentation on high-resolution images," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 405–420.

[26] R. P. Poudel, S. Liwicki, and R. Cipolla, "Fast-scnn: Fast semantic segmentation network," *arXiv preprint arXiv:1902.04502*, 2019.

[27] H. Li, P. Xiong, H. Fan, and J. Sun, "Dfanet: Deep feature aggregation for real-time semantic segmentation," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 9522–9531.

[28] C. Yu, J. Wang, C. Peng, C. Gao, G. Yu, and N. Sang, "Bisenet: Bilateral segmentation network for real-time semantic segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 325–341.

[29] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *ICML*, 2019.

[30] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, *et al.*, "Searching for mobilenetv3," in *ICCV*, 2019.

[31] K. Han, Y. Wang, Q. Tian, J. Guo, C. Xu, and C. Xu, "Ghostnet: More features from cheap operations," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 1580–1589.

[32] S. Mehta and M. Rastegari, "Mobilevit: Light-weight, general-purpose, and mobile-friendly vision transformer," in *International Conference on Learning Representations*, 2022. URL: https://openreview.net/forum?id=vh-0sUt8HlG.

[33] Y. Chen, X. Dai, D. Chen, M. Liu, X. Dong, L. Yuan, and Z. Liu, "Mobile-former: Bridging mobilenet and transformer," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2022.

[34] C. Gong, D. Wang, M. Li, X. Chen, Z. Yan, Y. Tian, qiang liu, and V. Chandra, "NASVit: Neural architecture search for efficient vision transformers with gradient conflict aware supernet training," in *International Conference on Learning Representations*, 2022. URL: https://openreview.net/forum?id=Qaw16njk6L.

[35] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NeurIPS*, 2015.

[36] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *ICCV*, 2017.

[37] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *ICCV*, 2017.

[38] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[39] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[40] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *ECCV*, 2018.

[41] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[42] H. Cai, C. Gan, J. Lin, and S. Han, "Network augmentation for tiny deep learning," *arXiv preprint arXiv:2110.08890*, 2021.

[43] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017.

[44] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *AAAI*, 2018.

[45] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *ECCV*, 2018.

[46] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han, "Apq: Joint search for network architecture, pruning and quantization policy," in *CVPR*, 2020.

[47] D. Bolya, C.-Y. Fu, X. Dai, P. Zhang, and J. Hoffman, "Hydra attention: Efficient attention with many heads," in *Computer Vision–ECCV 2022 Workshops: Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part VII*, Springer, 2023, pp. 35–49.

[48] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, *et al.*, "Rethinking attention with performers," *arXiv preprint arXiv:2009.14794*, 2020.

[49] Z. Shen, M. Zhang, H. Zhao, S. Yi, and H. Li, "Efficient attention: Attention with linear complexities," in *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, 2021, pp. 3531–3539.

[50] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv preprint arXiv:2006.04768*, 2020.

[51] Z. Dai, H. Liu, Q. V. Le, and M. Tan, "Coatnet: Marrying convolution and attention for all data sizes," *Advances in Neural Information Processing Systems*, vol. 34, pp. 3965–3977, 2021.

[52] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2022.

[53] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[54] M. Tan and Q. Le, "Efficientnetv2: Smaller models and faster training," in *International Conference on Machine Learning*, PMLR, 2021, pp. 10 096–10 106.

[55] A. Hatamizadeh, G. Heinrich, H. Yin, A. Tao, J. M. Alvarez, J. Kautz, and P. Molchanov, "Fastervit: Fast vision transformers with hierarchical attention," *arXiv preprint arXiv:2306.06189*, 2023.

[56] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223.

[57] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, "Scene parsing through ade20k dataset," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 633–641.

[58] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 126–135.

[59] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, IEEE, vol. 2, 2001, pp. 416–423.

[60] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 4401–4410.

[61] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[62] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[63] J. Liang, J. Cao, G. Sun, K. Zhang, L. Van Gool, and R. Timofte, "Swinir: Image restoration using swin transformer," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 1833–1844.

[64] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.

[65] B. Cheng, I. Misra, A. G. Schwing, A. Kirillov, and R. Girdhar, "Masked-attention mask transformer for universal image segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 1290–1299.

[66] B. Cheng, A. Schwing, and A. Kirillov, "Per-pixel classification is not all you need for semantic segmentation," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[67] S. W. Zamir, A. Arora, S. Khan, M. Hayat, F. S. Khan, and M.-H. Yang, "Restormer: Efficient transformer for high-resolution image restoration," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 5728–5739.

[68] L. Zhou, H. Cai, J. Gu, Z. Li, Y. Liu, X. Chen, Y. Qiao, and C. Dong, "Efficient image super-resolution using vast-receptive-field attention," *arXiv preprint arXiv:2210.05960*, 2022.

[69] Z. Li, Y. Liu, X. Chen, H. Cai, J. Gu, Y. Qiao, and C. Dong, “Blueprint separable residual network for efficient image super-resolution,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 833–843.

[70] Z. Zhang, H. Cai, and S. Han, “Efficientvit-sam: Accelerated segment anything model without performance loss,” *arXiv preprint arXiv:2402.05008*, 2024.

[71] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft coco: Common objects in context,” in *European conference on computer vision*, Springer, 2014, pp. 740–755.

[72] A. Gupta, P. Dollar, and R. Girshick, “Lvis: A dataset for large vocabulary instance segmentation,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 5356–5364.

[73] Y. Li, H. Mao, R. Girshick, and K. He, “Exploring plain vision transformer backbones for object detection,” *arXiv preprint arXiv:2203.16527*, 2022.

[74] M. Kang, J.-Y. Zhu, R. Zhang, J. Park, E. Shechtman, S. Paris, and T. Park, “Scaling up gans for text-to-image synthesis,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 10 124–10 134.

[75] Y. Balaji, S. Nah, X. Huang, A. Vahdat, J. Song, K. Kreis, M. Aittala, T. Aila, S. Laine, B. Catanzaro, *et al.*, “Ediffi: Text-to-image diffusion models with an ensemble of expert denoisers,” *arXiv preprint arXiv:2211.01324*, 2022.

[76] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. L. Denton, K. Ghasemipour, R. Gontijo Lopes, B. Karagol Ayan, T. Salimans, *et al.*, “Photorealistic text-to-image diffusion models with deep language understanding,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 36 479–36 494, 2022.

[77] A. Blattmann, T. Dockhorn, S. Kulal, D. Mendelevitch, M. Kilian, D. Lorenz, Y. Levi, Z. English, V. Voleti, A. Letts, *et al.*, “Stable video diffusion: Scaling latent video diffusion models to large datasets,” *arXiv preprint arXiv:2311.15127*, 2023.

[78] L. Zhang, A. Rao, and M. Agrawala, “Adding conditional control to text-to-image diffusion models,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 3836–3847.

[79] A. Brock, J. Donahue, and K. Simonyan, “Large scale gan training for high fidelity natural image synthesis,” *arXiv preprint arXiv:1809.11096*, 2018.

[80] X. Huang and S. Belongie, “Arbitrary style transfer in real-time with adaptive instance normalization,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1501–1510.

[81] E. Perez, F. Strub, H. De Vries, V. Dumoulin, and A. Courville, “Film: Visual reasoning with a general conditioning layer,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[82] F. Bao, C. Li, Y. Cao, and J. Zhu, “All are worth words: A vit backbone for score-based diffusion models,” in *NeurIPS 2022 Workshop on Score-Based Methods*, 2022.

[83] W. Peebles and S. Xie, "Scalable diffusion models with transformers," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 4195–4205.

[84] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in neural information processing systems*, vol. 33, pp. 6840–6851, 2020.

[85] B. Yang, G. Bender, Q. V. Le, and J. Ngiam, "Condconv: Conditionally parameterized convolutions for efficient inference," *Advances in neural information processing systems*, vol. 32, 2019.

[86] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.

[87] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," *arXiv preprint arXiv:1908.09791*, 2019.

[88] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, "Slimmable neural networks," in *ICLR*, 2019.

[89] D. Ha, A. M. Dai, and Q. V. Le, "Hypernetworks," in *International Conference on Learning Representations*, 2017. URL: https://openreview.net/forum?id=rkpACe1lx.

[90] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Smash: One-shot model architecture search through hypernetworks," *arXiv preprint arXiv:1708.05344*, 2017.

[91] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[92] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[93] Z. Shi, X. Zhou, X. Qiu, and X. Zhu, "Improving image captioning with better use of captions," *arXiv preprint arXiv:2006.11807*, 2020.

[94] X. Li, Y. Liu, L. Lian, H. Yang, Z. Dong, D. Kang, S. Zhang, and K. Keutzer, "Q-diffusion: Quantizing diffusion models," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 17 535–17 545.

[95] S. Wu, H. R. Zhang, and C. Ré, "Understanding and improving information transfer in multi-task learning," *arXiv preprint arXiv:2005.00944*, 2020.

[96] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[97] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of stylegan," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 8110–8119.

[98] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," *Advances in neural information processing systems*, vol. 30, 2017.

[99] J. Hessel, A. Holtzman, M. Forbes, R. L. Bras, and Y. Choi, "Clipscore: A reference-free evaluation metric for image captioning," *arXiv preprint arXiv:2104.08718*, 2021.

[100] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, *et al.*, "Learning transferable visual models from natural language supervision," in *International conference on machine learning*, PMLR, 2021, pp. 8748–8763.

[101] J. Ho and T. Salimans, "Classifier-free diffusion guidance," *arXiv preprint arXiv:2207.12598*, 2022.

[102] C. Lu, Y. Zhou, F. Bao, J. Chen, C. Li, and J. Zhu, "Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps," *Advances in Neural Information Processing Systems*, vol. 35, pp. 5775–5787, 2022.

[103] P. Dhariwal and A. Nichol, "Diffusion models beat gans on image synthesis," *Advances in neural information processing systems*, vol. 34, pp. 8780–8794, 2021.

[104] W. Zhao, L. Bai, Y. Rao, J. Zhou, and J. Lu, "Unipc: A unified predictor-corrector framework for fast sampling of diffusion models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[105] T. Salimans and J. Ho, "Progressive distillation for fast sampling of diffusion models," *arXiv preprint arXiv:2202.00512*, 2022.

[106] Y. Song, P. Dhariwal, M. Chen, and I. Sutskever, "Consistency models," *arXiv preprint arXiv:2303.01469*, 2023.

[107] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018.

[108] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *ECCV*, 2018.

[109] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation," in *CVPR*, 2018.

[110] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," in *ICLR*, 2018.

[111] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI*, 2019.

[112] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," in *ICML*, 2018.

[113] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *ICLR*, 2019.

[114] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *CVPR*, 2019.

[115] R. Luo, F. Tian, T. Qin, and T.-Y. Liu, "Neural architecture optimization," in *NeurIPS*, 2018.

[116] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in *ICML*, 2018.

[117] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *NeurIPS*, 2015.

[118] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," in *Reinforcement Learning*, 1992.

[119] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.

[120] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," in *ICML*, 2018.

[121] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," *arXiv preprint arXiv:1711.04528*, 2017.

[122] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *JMLR*, 2019.

[123] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "Hat: Hardware-aware transformers for efficient natural language processing," *arXiv preprint arXiv:2005.14187*, 2020.

[124] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and$<$ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[125] C.-H. Hsu, S.-H. Chang, D.-C. Juan, J.-Y. Pan, Y.-T. Chen, W. Wei, and S.-C. Chang, "Monas: Multi-objective neural architecture search using reinforcement learning," *arXiv preprint arXiv:1806.10332*, 2018.

[126] J.-D. Dong, A.-C. Cheng, D.-C. Juan, W. Wei, and M. Sun, "Dpp-net: Device-aware progressive search for pareto-optimal neural architectures," in *ECCV*, 2018.

[127] T. Elsken, J. H. Metzen, and F. Hutter, "Multi-objective architecture search for cnns," *arXiv preprint arXiv:1804.09081*, 2018.

[128] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization," in *CVPR*, 2019.

[129] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[130] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *CVPR*, 2018.

[131] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," in *ACL*, 2019.

[132] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *ECCV*, 2018.

[133] H. Cai, J. Lin, Y. Lin, Z. Liu, H. Tang, H. Wang, L. Zhu, and S. Han, "Enable deep learning on mobile devices: Methods, systems, and applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 3, pp. 1–50, 2022.

[134] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," in *ICLR*, 2017.

[135] H. Cai, T. Wang, Z. Wu, K. Wang, J. Lin, and S. Han, "On-device image classification with proxyless neural architecture search and quantization-aware fine-tuning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019, pp. 0–0.

[136] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *CVPR*, 2019.

[137] Z. Wu, T. Nagarajan, A. Kumar, S. Rennie, L. S. Davis, K. Grauman, and R. Feris, "Blockdrop: Dynamic inference paths in residual networks," in *CVPR*, 2018.

[138] L. Liu and J. Deng, "Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution," in *AAAI*, 2018.

[139] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, "Skipnet: Learning dynamic routing in convolutional networks," in *ECCV*, 2018.

[140] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger, "Multi-scale dense networks for resource efficient image classification," in *ICLR*, 2018.

[141] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *NeurIPS*, 2017.

[142] J. Kuen, X. Kong, Z. Lin, G. Wang, J. Yin, S. See, and Y.-P. Tan, "Stochastic downsampling for cost-adjustable inference and improved regularization in convolutional networks," in *CVPR*, 2018.

[143] J. Yu and T. Huang, "Universally slimmable networks and improved training techniques," in *ICCV*, 2019.

[144] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks.," in *CVPR*, 2017.

[145] B. Cheung, A. Terekhov, Y. Chen, P. Agrawal, and B. Olshausen, "Superposition of many models into one," in *NeurIPS*, 2019.

[146] A. Ashok, N. Rhinehart, F. Beainy, and K. M. Kitani, "N2n learning: Network to network compression via policy gradient reinforcement learning," in *ICLR*, 2018.

[147] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," *arXiv preprint arXiv:1904.00420*, 2019.

[148] J. Yu and T. Huang, "Autoslim: Towards one-shot architecture search for channel numbers," *arXiv preprint arXiv:1903.11728*, 2019.

[149] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[150] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2009.

[151] H. Cai, J. Lin, Y. Lin, Z. Liu, K. Wang, T. Wang, L. Zhu, and S. Han, "Automl for architecting efficient and specialized neural networks," *IEEE Micro*, 2019.

[152] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015.

[153] Y. Wu and K. He, "Group normalization," in *ECCV*, 2018.

[154] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," in *ICLR Workshop*, 2018.

[155] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[156] Y. Cui, Y. Song, C. Sun, A. Howard, and S. Belongie, "Large scale fine-grained categorization and domain-specific transfer learning," in *CVPR*, 2018.

[157] P. K. Mudrakarta, M. Sandler, A. Zhmoginov, and A. Howard, "K for the price of 1: Parameter-efficient multi-task and transfer learning," in *ICLR*, 2019.

[158] S. Kornblith, J. Shlens, and Q. V. Le, "Do better imagenet models transfer better?" In *CVPR*, 2019.

[159] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3d object representations for fine-grained categorization," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2013.

[160] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, 2008.

[161] S. Maji, E. Rahtu, J. Kannala, M. Blaschko, and A. Vedaldi, "Fine-grained visual classification of aircraft," *arXiv preprint arXiv:1306.5151*, 2013.

[162] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "The caltech-ucsd birds-200-2011 dataset," 2011.

[163] O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. Jawahar, "Cats and dogs," in *CVPR*, 2012.

[164] L. Bossard, M. Guillaumin, and L. Van Gool, "Food-101–mining discriminative components with random forests," in *ECCV*, 2014.

[165] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[166] Z. Liu, P. Luo, X. Wang, and X. Tang, "Large-scale celebfaces attributes (celeba) dataset," *Retrieved August*, 2018.

[167] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman, "Vggface2: A dataset for recognising faces across pose and age," in *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*, 2018.

[168] A. Kolesnikov, L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby, "Big transfer (bit): General visual representation learning," *arXiv preprint arXiv:1912.11370*, 2019.

[169] S. Qiao, H. Wang, C. Liu, W. Shen, and A. Yuille, "Weight standardization," *arXiv preprint arXiv:1903.10520*, 2019.

[170] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[171] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, "Return of the devil in the details: Delving deep into convolutional nets," in *BMVC*, 2014.

[172] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "Decaf: A deep convolutional activation feature for generic visual recognition," in *ICML*, 2014.

[173] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "Cnn features off-the-shelf: An astounding baseline for recognition," in *CVPR Workshops*, 2014.

[174] P. K. Mudrakarta, M. Sandler, A. Zhmoginov, and A. Howard, "K for the price of 1: Parameter efficient multi-task and transfer learning," in *ICLR*, 2019.

[175] L. Liu, L. Deng, X. Hu, M. Zhu, G. Li, Y. Ding, and Y. Xie, "Dynamic sparse graph for efficient deep learning," in *ICLR*, 2019.