AdaptiveNet: Post-deployment Neural Architecture Adaptation for Diverse Edge Environments

Hao Wen
Institute for AI Industry Research (AIR), Tsinghua University

Yuanchun Li
Institute for AI Industry Research (AIR), Tsinghua University

Zunshuai Zhang
Shanghai University

Shiqi Jiang
Microsoft Research

Xiaozhou Ye
AsiaInfo Technologies (China), Inc.

Ye Ouyang
AsiaInfo Technologies (China), Inc.

Ya-Qin Zhang
Institute for AI Industry Research (AIR), Tsinghua University

Yunxin Liu
Institute for AI Industry Research (AIR), Tsinghua University

ABSTRACT

Deep learning models are increasingly deployed to edge devices for real-time applications. To ensure stable service quality across diverse edge environments, it is highly desirable to generate tailored model architectures for different conditions. However, conventional pre-deployment model generation approaches are not satisfactory due to the difficulty of handling the diversity of edge environments and the demand for edge information. In this paper, we propose to adapt the model architecture after deployment in the target environment, where the model quality can be precisely measured and private edge data can be retained. To achieve efficient and effective edge model generation, we introduce a pretraining-assisted on-cloud model elastification method and an edge-friendly on-device architecture search method. Model elastification generates a high-quality search space of model architectures with the guidance of a developer-specified oracle model. Each subnet in the space is a valid model with different environment affinity, and each device efficiently finds and maintains the most suitable subnet based on a series of edge-tailored optimizations. Extensive experiments on various edge devices demonstrate that our approach is able to achieve significantly better accuracy-latency tradeoffs (e.g. 46.74% higher on average accuracy with a 60% latency budget) than strong baselines with minimal overhead (13 GPU hours in the cloud and 2 minutes on the edge server).

ACM Reference Format:
Hao Wen, Yuanchun Li, Zunshuai Zhang, Shiqi Jiang, Xiaozhou Ye, Ye Ouyang, Ya-Qin Zhang, and Yunxin Liu. 2023. AdaptiveNet: Post-deployment Neural Architecture Adaptation for Diverse Edge Environments. In *The 29th Annual International Conference On Mobile Computing And Networking (MobiCom '23), 2-6 Oct 2023, Madrid, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/xxxxxx.xxxxxx

1 INTRODUCTION

Deep learning has enabled and enhanced many intelligent applications at the edge, such as driving assistance [20, 65], face authentication [6, 7], video surveillance [58, 64], speech recognition [39, 52], etc. Due to latency and privacy considerations, it is an increasingly common practice to deploy the models to edge devices [11, 56], so that the models can be invoked directly without transmitting data to the server.

The diversity of execution environments is a unique characteristic of edge devices as compared with the cloud. For example, a video app may deploy a super-resolution model on millions of smartphones, ranging from low-end devices to high-end ones, and their computational capacity may differ by up to 20 times. Generating a model for each type of device to guarantee the user experience is very time-consuming; an object detection model may run on different kinds of driving assistance systems, and the computational power may range from 20 to 1000 TOPS. To guarantee safety, the model inference usually has a strict latency budget. Even on the same type of devices, the model execution environments may also vary across instances and change over time due to different hardware states and concurrent processes. To provide a good

Yuanchun Li (liyuanchun@air.tsinghua.edu.cn) is the corresponding author. Yuanchun Li and Yunxin Liu are also affiliated with Shanghai AI Laboratory.

MobiCom '23, 2-6 Oct 2023, Madrid, Spain

ACM ISBN 978-1-4503-9185-6/22/06.
https://doi.org/10.1145/xxxxxx.xxxxxx

and uniform user experience, developers are usually required to generate tailored models for diverse edge environments.

There are many techniques proposed to automatically generate tailored models according to the target environments. Most of them are cloud-based approaches, in which the models are determined on the cloud side before distributing to edge devices. We call them **"pre-deployment approaches"** in this paper. Neural Architecture Search (NAS) [3, 5, 35, 36, 41, 69] is the most popular technique of this type due to its superior flexibility to change network architectures. It typically requires collecting information (about computational resources, runtime conditions, data distribution, etc.) from the target environments to guide the model architecture search and training processes in the cloud.

Despite the effectiveness to find optimal model architecture based on the target environment, NAS approaches are less practical in many edge/mobile scenarios where the model execution environments may be very diverse and dynamic. Searching and maintaining the optimal model architecture in the cloud for each edge would be very compute- and labor-intensive. Thus, *a more economic and ideal solution is to let the model self-adapt to the target environment after deployment,* which we call **"post-deployment approach"** to distinguish with the conventional methods, as illustrated in Figure 1. Doing so brings several other benefits - the quality of model architectures can be more precisely measured in the target environment, and user privacy can be better protected because there is no need to collect edge information.

The idea of adapting the model to the target device has been explored in both the mobile computing community [10, 14] and the machine learning community [37, 53]. The mobile community is mainly focused on model scaling, *i.e*. adjusting the model complexity to fulfill certain latency requirements, while ML research mainly aims to deal with different data distributions or hard/easy samples. Model scaling approaches share a similar goal as ours, but prior work only shrinks the model size through pruning or quantization instead of changing the model architecture, which limits the opportunity to achieve optimal accuracy-latency tradeoffs.

To this end, we introduce AdaptiveNet, an end-to-end system to generate models for diverse edge environments through post-deployment on-device neural architecture adaptation. We focus on two related challenges in AdaptiveNet. First, generating the search space of model architectures is non-trivial since the space must contain enough high-quality candidates that are suitable for different edge conditions. Second, directly searching the optimal model architecture at the edge may be time-consuming due to the limited on-device resources.

AdaptiveNet addresses the above challenges by training a supernet once and letting the edge devices choose the satisfactory subnet on their own. The method can be divided into two stages, the on-cloud elastification and on-device search.

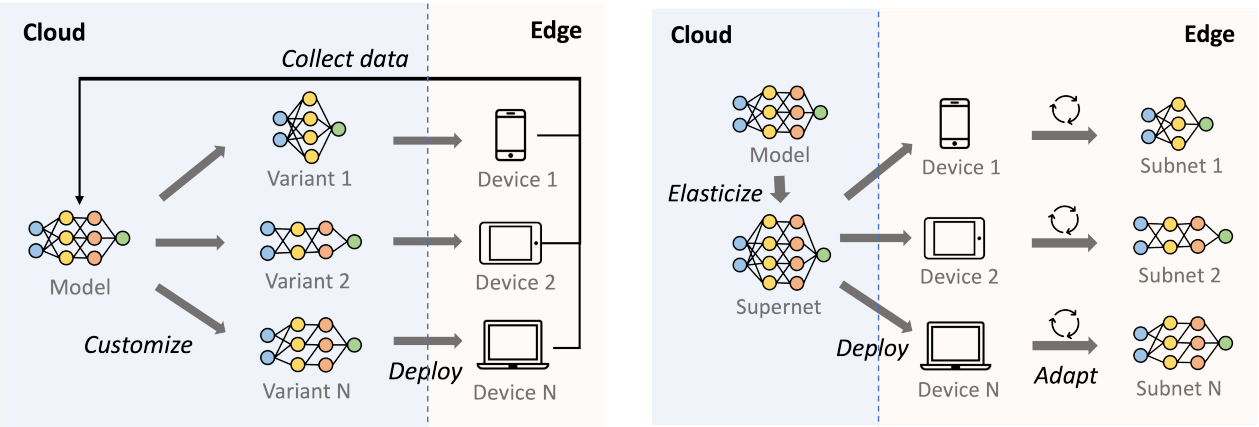
In the first stage, we design an on-cloud model elastification method to generate a high-quality search space for edge devices. Specifically, the elastification takes an arbitrary pretrained model as the input and converts it to a multi-path supernet by adding branches into the pretrained model, ensuring each path in the supernet is a valid and useful model. We introduce block-wise knowledge distillation to train the newly added branches, which consequently improves the quality of the subnets. Our supernet offers millions of model variations with different structures, and the edge side only needs to find the best structure iteratively without additional training.

In the second stage, to improve the efficiency of architecture search and update at the edge, we systematically optimize the search process according to edge characteristics. We first build a performance model on the device by profiling each block in the supernet, which guides the candidate selection during the search, therefore reducing the number of iterations needed to find the optimal model architecture. Then we introduce a reuse-based model evaluation method, which caches intermediate features across model candidates to reduce the time required to evaluate the models in each iteration.

To evaluate our approach, we conduct experiments with various popular tasks (image classification, object detection, and semantic segmentation) and models (ResNet [19], MobileNetV2 [42], EfficientNetV2 [46], etc.) on three edge devices including Jetson Nano, Android smartphone (Xiaomi 12), and edge server (NVIDIA 2080 Ti GPU). We compare AdaptiveNet with several strong baselines including LegoDNN [14], FlexDNN [9], SkipNet [53], and Slimmable Neural Networks [61]. The results have shown that our method can achieve significantly better accuracy-latency tradeoffs than state-of-the-art baselines. For example, our method can generate models that have 46.74% higher accuracy than those produced by other methods with a latency budget of 60%. Meanwhile, the overhead of our method is minimal which only takes 13 GPU hours for elastification in the cloud and 2 minutes for adaptation on the edge server.

Our work makes the following technical contributions:

(1) We propose and develop the concept of on-device post-deployment neural architecture adaptation, and implement it with an end-to-end system.
(2) We introduce a pretraining-assisted model elastification method that can effectively and flexibly generate a model search space, as well as edge-tailored strategies to search the optimal model from the space and maintain it at runtime.
(3) Our method achieves significantly better accuracy-latency tradeoffs than SOTA baselines according to experiments on various edge devices and common tasks. The tool and models will be open-sourced for edge AI developers to use.



(a) Pre-deployment on-cloud model generation (conventional). **(b) Post-deployment on-device model adaptation (ours).**

Figure 1: Comparison of pre-deployment and post-deployment model generation approaches.

2 BACKGROUND AND MOTIVATION

2.1 Current Practice and Related Work for Edge Model Generation

Deploying deep neural networks (DNNs) at the edge is increasingly popular due to latency requirements and privacy concerns. Since DNN models are mostly computationally heavy, deploying them to the edge usually has to consider two characteristics of edge devices. First, edge devices are mostly resource-constrained. As a result, there are already a lot of efforts on improving the performance of DNN models on edge devices, including optimizing the DNN inference framework on heterogeneous edge devices [13, 25, 51], designing lightweight model backbones [21, 22, 42, 45, 46, 66] and compressing the models to be deployed [15, 43, 50, 63, 67].

Besides the resource limitation, another major challenge of edge environments is the diversity - model developers usually need to deploy a certain model to thousands even millions of devices that are different from each other. The deployed models are usually expected to meet a certain budget of latency while achieving higher accuracy, or achieve certain expected accuracy while minimizing the latency. Thus, customizing the model for different target devices becomes a necessity.

The current practices to handle edge environment diversity are mostly cloud-based pre-deployment approaches, *i.e.* the central server generates models for different edge devices before distributing them for deployment. Since manually designing models for diverse edge environments is cumbersome, the common practice is to use automated model generation techniques. NAS [12, 29, 44, 55, 59] is the most representative and widely-used model generation method, which searches for the optimal network architecture in a well-designed search space. Most NAS methods require training the architectures during searching [36, 41, 44, 69], which is very time-consuming (10,000+ GPU hours) when generating models for a large number of devices. One-shot NAS [3, 5, 23, 34] is proposed to greatly reduce the training cost by allowing the candidate networks to share a common over-parameterized supernet. Among them, several approaches also mention the concept of directly searching the architecture for target data and devices [5, 34]. However, they require to collect much information from the edge devices to build accuracy and latency predictors, which are used to guide the search process in the cloud.

There are also several approaches proposed to scale models at the edge to provide a wider range of resource-accuracy trade-offs. Most of them apply structured pruning (or similar techniques) to generate various descendent models [10, 14, 30, 37, 54, 60, 61], which can adjust the size of each network module without changing the architecture. However, they have limited abilities to generate optimal models for diverse edge environments due to the restricted model space. Dynamic Neural Networks [16] are a type of DNN that support flexible inference based on the difficulty of input. When the input is easy, Dynamic Neural Networks can reduce the computation by skipping a set of blocks [53, 57] or exiting from the middle layers [1, 9, 27, 28]. However, this kind of work only considers dynamically adjusting the depth of DNN models, and they are not completely suitable for situations where latency budgets are strict.

2.2 Limitations of Current Practice

We conduct several motivational studies to understand the limitations of the conventional model generation method.

We argue that the cloud-based pre-deployment model generation approaches underestimate the diversity of edge environments. We identify three types of diversity:

(1) **Inter-device diversity.** Edge devices are equipped with various types and grades of processors for DL inference,

Table 1: Average latency (ms) of two DNN models on four mobile phones.

Device	MobileNetV2	ResNet50
XIAOMI 12 (Snapdragon 8 Gen 1)	14.43	90.67
HUAWEI nova 4 (HiSilicon KIRIN 970)	53.05	372.22
Google Pixel 2 (Snapdragon 835 64)	46.45	283.74
Google Pixel 6 Pro (Google Tensor)	31.29	144.21

Table 2: Average latency (ms) of two DNN models under different conditions on NVIDIA 2080 Ti. The background processes use the same setting of the process in the "normal" condition, *i.e.* continuous DNN inference with batch size 32 and CUDA version 11.3. In the "CUDA version changed" condition, we change the CUDA version to 10.1. In the "different batch size" condition, the batch size is set to 64.

Condition	MobileNetV2	ResNet50
Normal	13.35	33.79
1 background process	14.37	50.36
3 background processes	24.07	115.09
CUDA version changed	13.69	35.89
Different batch size	12.23	31.71

such as CPU, GPU, and AI accelerators. Even for devices with the same type of hardware, their conditions can be different. We measure the inference latency of two popular DNN models on four different mobile devices. As shown in Table 1, the inference latency of a model varies a lot on different devices.

(2) **Intra-device diversity.** Even on the same device, the inference latency of a model may also be affected by various factors, including background processes, software versions, hardware aging, ambient temperature, etc. Table 2 shows the non-neglectable impact of varying conditions on inference latency.

(3) **Data distribution diversity.** NAS approaches need to search for the optimal architecture over a given dataset. However, edge devices are usually used in different locations and by different users, dealing with different data distributions [26]. For example, some smart cameras are deployed in outdoor scenarios while some are indoor, and the common classes of objects in the scene may be different across devices.

Such complex and ubiquitous diversity poses several difficulties for cloud-based model generation. **First, tailoring models for diverse edge environments is a heavy task.** To generate optimal model architecture for each edge environment, the current practice requires repeating the search process for all types of environments and maintaining them in the cloud. The required manual and computational effort are

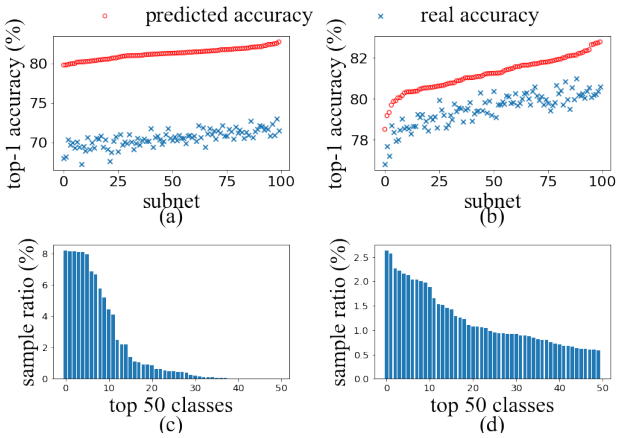


Figure 2: Performance of cloud-trained accuracy predictor on distribution-shifted edge data. The edge data is simulated with Dirichlet distributions with (a) $\alpha = 0.005$ and (b) $\alpha = 0.1$. The sample ratios of top-50 classes are shown in (c) and (d).

determined by the granularity of edge environment diversity to consider, which might be burdensome if the developers want to achieve optimal latency-accuracy tradeoffs on more devices. Meanwhile, handling the dynamicity of the edge environment is even more difficult since it requires frequent communications with each edge device and rapid reactive model updating in the cloud.

Second, modeling the edge environment may also be difficult. A necessary step in the cloud-based model generation is to estimate the performance of the candidate model, such that the model architecture can be optimized according to the target hardware and data. For example, existing NAS methods are usually based on accuracy and latency predictors [4]. Building the predictors requires collecting intensive edge information, which is not easy, especially for the accuracy predictor that depends on the potentially private edge data. The compromise solution is to use a unified accuracy predictor for different edge devices [3]. However, the unified accuracy predictor may not perform well for edge devices with data distribution shifts. As shown in Figure 2, the accuracy values and rankings of candidate models predicted by the once-for-all accuracy predictor [3] are both different from the ground truth, which indicates that the predictors can be unreliable on edge distributions, leading to sub-optimal model generation.

2.3 Post-deployment Neural Architecture Generation: Goal and Challenges

The limitations of existing edge model generation methods motivate us to think, can we directly search for the optimal neural architecture on the edge device after deployment?

Doing so brings several key advantages. Unlike traditional on-cloud NAS which has to estimate the performance of subnets, edge devices can directly evaluate the performance of a given model architecture natively, which is more precise. Besides, searching on the device is a plug-and-play process and does not need to collect edge information to the cloud, bringing the benefits of protecting user privacy and reducing the computation overhead of the cloud.

On the other hand, finding the optimal model architecture directly at the edge is challenging. **First, generating the model search space for edge devices is difficult**. The search space should be flexibly and easily customizable to support diverse edge applications and different ranges of target devices. Meanwhile, since the training abilities of edge devices are usually weak, the search space should contain high-quality candidate models that can be used in different edge environments with minimal (or even no) further tuning. **Second, the model search process can be time-consuming at the edge.** Existing architecture search methods require either training a lot of candidate models or repeatedly evaluating the performance of the candidates. Both are very heavy for the edge devices because of the limited computing resources and tight deadline of model initialization. Dynamically updating the model according to environment changes is even more time-sensitive.

3 ADAPTIVENET OVERVIEW

To solve the aforementioned challenges and realize the vision of post-deployment model generation, we introduce AdaptiveNet, an on-device neural architecture adaptation approach for diverse edge environments. To the best of our knowledge, AdaptiveNet is the first end-to-end system to enable on-device architecture adaptation.

The main idea of AdaptiveNet is to generate high-quality model search spaces based on developer-specified pretrained networks through modular expansion and distillation, and efficiently search for the optimal architecture on the target device guided by performance modeling. Figure 3 illustrates the architecture of AdaptiveNet, which includes an on-cloud model elastification and an on-device subnet search.

Our model elastification is efficient by leveraging the guidance of a developer-specified pretrained model. It mainly consists of a granularity-aware graph expansion step and a distillation-based training step. Given an arbitrary pretrained network, we first discover the repeating basic blocks and determine the replaceable paths in the computational graph. Then we add optional branches to the model to extend it into a supernet. The added branches include layers that can replace multiple original layers, or structured-pruned layers that reduce the computational cost of individual layers. Each path from the input to the output in the graph is a valid subnet, which consists of both original and newly added modules.

The supernet obtained by graph expansion contains a large variety of architectures with different levels of computational complexity. We then further improve the quality of each candidate architecture in the supernet through training, so that on-device training can be avoided to save computation cost. Since our supernet is generated from a pretrained model, we use branch-wise distillation to efficiently train the newly-added branches to mimic the original branches. The distillation is followed by a whole-graph fine-tuning to further improve the overall accuracy of the subnets. With all these techniques, the supernet would contain high-quality subnets that can fit in different edge environments, and it is deployed to the edge devices for further adaptation.

The on-device subnet search stage aims to find the most appropriate subnet (that can achieve the highest accuracy within the latency budget) on resource-limited edge devices. We first build a latency model by profiling the blocks in the supernet to precisely estimate the latency of subnets in the native environment. Based on the latency model, we design a search strategy that initializes a set of promising candidate models and iteratively mutates the candidates around the latency budget. The search efficiency is further improved by reusing the common intermediate features during candidate model evaluation. The optimal model is also adaptively updated by the runtime monitor to handle environment dynamicity.

4 ELASTIFICATION ON CLOUD

The input of the model elastification stage is a developer-specified pretrained model, similar to the common scenarios in edge AI deployment. The pretrained model is determined as the best-performing model that can fulfill (or slightly exceed) the latency budget on the highest-end target device.

The goal of elastification is to convert the given pretrained model into a *supernet*, by expanding alternative basic blocks, making connections between them, and letting each path in the supernet (namely *subnet*) behave correctly. In this way, the supernet is granted with the *elasticity*: each subnet has the discriminative performance characteristics in terms of inference latency and accuracy. The supernet is then deployed onto the edge, where the particular edge device can search for the most suitable subnet according to its own hardware capacities and data distribution.

There are two main problems to solve in elastification. The first is how to generate the supernet architecture. Although prior work [3] has discussed hand-crafted supernets for certain models, it is still an open question to automatically generate supernets based on an arbitrary pretrained model, especially considering the diversity of DNNs. Another problem is how

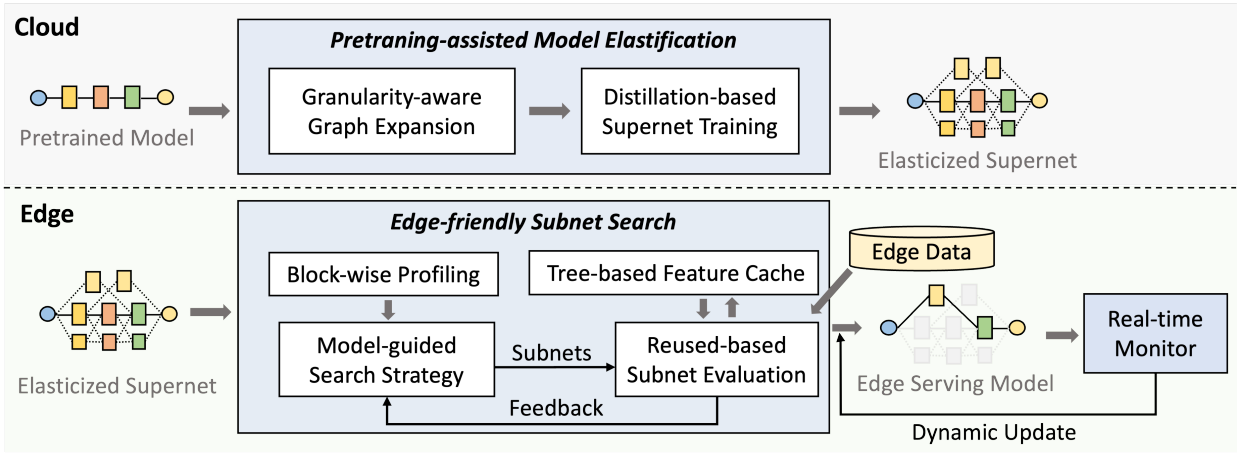


Figure 3: The architecture overview of AdaptiveNet.

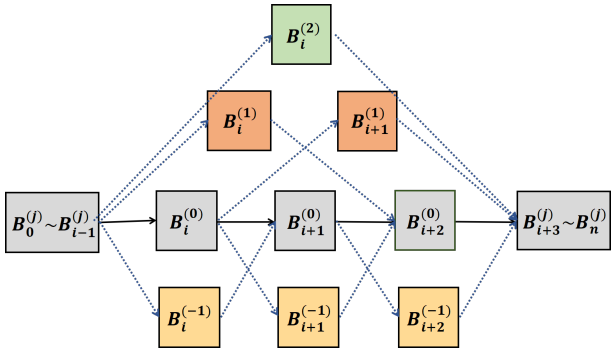


Figure 4: Supernet architecture in AdaptiveNet.

to train the subnets in the supernet to improve their quality. A supernet typically contains millions of subnets, thus training them separately is time-consuming. We propose two techniques to address these problems accordingly.

4.1 Granularity-aware Graph Expansion

Let $\mathcal{N}$ denote the pretrained model we want to elasticize. The first step is to analyze the computational graph of $\mathcal{N}$ to determine how it can be expanded. We call the smallest unit in the graph that be replaced as a *basic block*, and the block partitioning is determined by the following principles. First, the size of blocks determines the subnet search space size and the granularity of how the latency can be controlled. Thus, we limit the block parameter size to no more than $\gamma \cdot P_0$ where γ is a parameter to control the granularity and P_0 is the parameter size of the original model N. Second, the blocks should not span fusion layers. For example, Conv and ReLU can be fused in most inference frameworks [24]. Third, each basic block should be single-input and single-output in the original model graph. Following these principles, we can represent the supernet as a set of connected basic blocks $\mathcal{N} = graph\{\mathcal{B}^{(0)}, C\}$, where $\mathcal{B}^{(0)}$ is the set of blocks and C denotes the connections between them.

Next, we generate the supernet graph $\mathcal{S}$ by expanding the graph of the pretrained model $\mathcal{N}$ through adding alternative blocks and connections. Particularly, we consider two expanding strategies including merging and shrinking, as shown in Figure 4. First, we add merged blocks $B_i^{(j)}$ that can replace multiple basic blocks ($j > 0$ represents the number of reduced blocks in the replacement). Suppose $\{B_i^{(0)}, B_{i+1}^{(0)}, ..., B_{i+j}^{(0)}\}$ are the basic blocks in $\mathcal{N}$ that can be replaced by $B_i^{(j)}$, The input shape of $B_i^{(j)}$ is the same as $B_i^{(0)}$, and output shape is the same as $B_{i+j}^{(0)}$. The parameter size of $B_i^{(j)}$ is determined by the largest among the replaced blocks. Second, like traditional model scaling approaches [10, 14], we also add different levels of shrunk blocks $\mathcal{B}_i^{(-1)}, \mathcal{B}_i^{(-2)}, ...$ for each basic block $\mathcal{B}_i^{(0)} \in \mathcal{B}^{(0)}$ by reducing its size with structured pruning and network slimming techniques [30, 60, 61]. The granularity of merged blocks and pruned blocks can be balanced to control the size of subnet search space.

Compared to the existing model scaling methods [10, 14, 61], our supernet has higher elasticity because it allows the subnets to have different architectures, rather than just different sizes of the same architecture. This is also the reason why NAS outperforms other model generation techniques on the server side [3, 45].

4.2 Distillation-based Supernet Training

Next, we need to train the generated supernet to improve the quality of its subnets, so that the subnet can be directly used at the edge without further training. We achieve efficient and effective training by fully utilizing the supernet. The whole training process includes a branch distillation phase and a whole-model tuning phase.

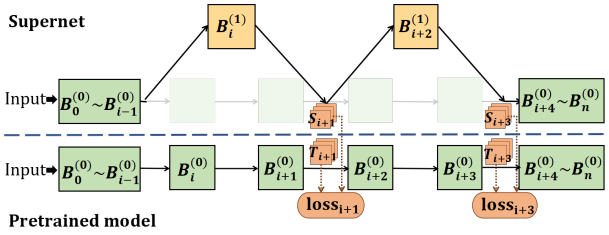


Figure 5: Illustration of the branch-wise distillation phase.

Branch-wise distillation. In this phase, we first freeze the weights of B_i^0 so that the accuracy of the original pretrained model is preserved. Then we adopt feature-based knowledge distillation [49] to let the added blocks imitate their corresponding original blocks. As illustrated in Figure 5, in each iteration, we randomly sample a subnet from the supernet and use $\mathcal{N}$ as the teacher model to train the new branches in the subnet. Specifically, let S_i denote the output feature map of a newly added block $B_i^{(j)}$ and T_i denotes the output feature map of the last block that $B_i^{(j)}$ replaces, we use the L2 distance as the distillation loss. The loss function is

$$\mathcal{L}_{distillation} = \frac{1}{M}\sum_{i=1}^{M} \|T_i - S_i\|_2^2, \quad (1)$$

where M denote the number of new blocks in the sampled subnet. With enough iterations applied, all new blocks in the supernet will be trained multiple times to improve their individual quality. Since we only train the new blocks and use the feature maps of the pretrained model as strong supervision, the distillation process is efficient and easy to converge.

Further tuning. We further train the supernet using labelled data to improve the end-to-end quality of the subnets. In each step of tuning, we randomly sample a subnet $B_i^{(j)}$, forward a batch of samples, compute the Cross-Entropy [38] loss between the subnet outputs and the labels, and update the parameters of the added blocks in $B_i^{(j)}$ via gradient descent. The performance of the supernet is measured by sampling a new set of random subnets and testing each of them on validation data. We use the *latency-range accuracy* as the training progress indicator, which records the average accuracy achieved by subnets in each latency range. This phase starts from the distilled supernet, and thus the learning rates are relatively small and the convergence is fast.

Notes on design rationale. Each phase in our design is indispensable to ensure training efficiency and effectiveness. Using distillation only will lead to suboptimal final accuracy, and direct training will significantly slow down the convergence. Merging the two phases together is also not desirable since it will make the loss design and training more difficult. The experimental comparison can be found in Section 7.5. We also note that our method does not modify the parameters of the pretrained model, so it guarantees that the latency-accuracy tradeoffs will be better than or at least equal to the pretrained model.

5 ADAPTATION ON EDGE

The supernet generated by model elastification is uniformly deployed to different edge devices, but it is not directly usable since each edge device has different characteristics and requirements. Thus, we further introduce the on-device adaptation stage to obtain the optimal architecture adaptively in the target environment by searching the subnet space. Such a search process is similar to traditional on-cloud NAS but has a higher requirement for efficiency.

According to our analysis, using a normal search method as in NAS can cost more than 10 hours on edge devices. Most of the searching time is spent on evaluating the subnets. This is because we have to perform model inference hundreds of times to get the accuracy of candidate models in each search iteration and use the accuracy results to guide the next search iteration. To reduce the searching overhead, we introduce a latency model-guided search strategy and a reuse-based model evaluation method.

5.1 Model-guided Search Strategy

We first optimize the search strategy to find the optimal model architecture (*i.e.* the architecture that can achieve the highest accuracy within the latency budget) with fewer iterations. The core idea of the optimization is to prune the search with the guidance of a natively-built latency model.

Formally, suppose T_{budget} denotes the latency budget in the target environment and D_{edge} is the edge dataset. Our goal is to find a subnet $\mathcal{N}' = \{B_{i_1}^{(j_1)}, B_{i_2}^{(j_2)}, ..., B_{i_n}^{(j_n)}\}$ from the supernet $\mathcal{S}$ whose latency $T(\mathcal{N}') \leq T_{budget}$ and accuracy $Acc(\mathcal{N}', D_{edge})$ is optimal. During the search, the accuracy of the candidate model is directly measured on the edge dataset D_{edge}, and the latency is computed with a latency model.

The latency model is a table $\mathcal{T} = \{T_i^{(j)}\}$ where $T_i^{(j)}$ is the latency of basic block $B_i^{(j)}$ in the supernet. The block latency is precisely measured on the device through profiling after deployment. Note that this process is quick (within seconds) because the number of basic blocks is small (much smaller than the number of subnets). Our supernet generation strategy (Section 4.1) ensures that the basic blocks will not be fused, thus the latency of a chosen subnet is the sum of all its blocks, *i.e.* $T(\mathcal{N}') = \sum_{k=0}^{n} T_{i_k}^{(j_k)}$. Note that we use the latency model to compute the latency rather than directly measure it because end-to-end latency measurement under the actual model operating condition is time-consuming.

The subnet search process contains two main steps, including candidate initialization and candidate mutation, where

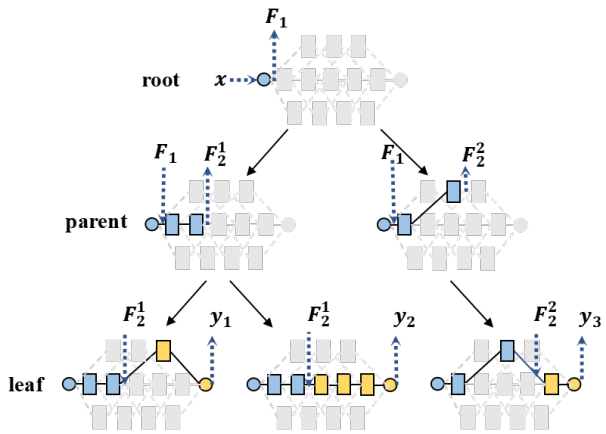


Figure 6: Illustration of subnets tree. Prefix layers shown in blue, subsequent in yellow. x, F_i^j, y_i represent input, shared features, output respectively.

the initialization step produces a set of seed subnets and each mutation step changes the subnets iteratively to better fit the target environment. Both the initialization and mutation are customized with the latency model in our approach. Our experiments show that the optimal subnets are often near the latency budget. Therefore, the initialization and mutation are designed to keep the search of candidates near the budget.

Specifically, we design two supporting functions $NearbyInit$ and $NearbyMutate$. $NearbyInit$ generates the initial candidate subnets by randomly sampling a group of subnets whose latencies lie in the range of $[T_{budget} - \Delta T, T_{budget} + \Delta T]$. The models with latency higher than T_{budget} are unlikely to be useful at the current moment, but they may be used later to handle dynamic environment change (see Section 5.3). Next, when we change the candidate subnets in each iteration, we first randomly mutate a subnet by replacing a branch in it. If the latency of the subnet after mutation is out of $[T_{budget} - \Delta T, T_{budget} + \Delta T]$, we iterate the rest branches in the supernet and find the best alternative branches that can reduce the latency change.

Both the initialization and mutation strategies are graph operations that do not involve heavy computation, but they can significantly reduce the evaluation overhead by improving search efficiency. Meanwhile, the model-guided initialization and mutation can be integrated into most standard search algorithms including evolutionary search [41] and simulated annealing [34]. Algorithm 1 shows the subnet search strategy based on evolutionary search.

5.2 Reuse-based Model Evaluation

While the model-guided search strategy reduces the required iterations, the model evaluation overhead in each iteration is still high. In each iteration, we usually need to evaluate hundreds of candidate subnets with the edge data to find the most accurate ones. The candidate subnets usually share common prefix substructures, so we have the opportunity to save time by reusing common intermediate features across subnets. For example, let $\mathcal{N}_1' = G_{prefix} \cup G_1$ and $\mathcal{N}_2' = G_{prefix} \cup G_2$ denote two different subnets and $\mathcal{N}_1'$ is evaluated before $\mathcal{N}_2'$, during the evaluation of $\mathcal{N}_1'$, the output feature of $G_{prefix}(D_{edge})$ can be saved and reused when evaluating S_2, which saves the inference cost of computing $G_{prefix}(D)$.

However, saving all the common features during model evaluation is infeasible because it will take too much memory. Thus, we can only save part of the common features and improve the reuse ratio as much as possible. In order to achieve this, we introduce a tree-based feature cache to schedule the evaluation. The leaf nodes and non-leaf nodes in the tree represent subnets and common prefix substructures respectively. The leaf nodes (subnets) sharing the same parent node have

Algorithm 1 Edge-friendly optimal subnet search

Input: Elasticized supernet S, Local data D, Latency budget T.
Output: Optimal subnet M.

function MAIN:
  $candidates \leftarrow NearbyInit(S, T)$; // Section 5.1
  $LatencyTable \leftarrow$ Block-wise latency table of S;
  $root \leftarrow BuildTree(candidates, LatencyTable)$;
  $DFS-Evaluate(D, root)$;
  for $i = 0; i < search\_times; i$+= 1 **do**;
    $candidates \leftarrow NearbyMutate(candidates)$; // Section 5.1
    $DFS{-}Evaluate(D, BuildTree(candidates, LatencyTable))$;
    Record best candidate;
  end for
return Optimal subnet M;
end function

function DFS-EVALUATE($D, root$):
  if $root.child == 0$ **then**
    return Evaluate $root$;
  else
    for each $child$ of $root$ **do**
      $PrefixFeature \leftarrow child.prefix.forward(D)$;
      Save $PrefixFeature$ in cache;
      $DFS-Evaluate(PrefixFeature, child)$;
      Release $PrefixFeature$;
    end for
  end if
end function

function BUILDTREE($candidates, LatencyTable$):
  Get $prefixes$ shared by $candidates$;
  for $prefix \in prefixes$ **do**
    Get $prefix.latency$ from $LatencyTable$;
    $prefix.importance \leftarrow prefix.latency \times PrefixingRate$;
  end for
  Delete less important prefixes if one subnet has several prefixes;
  Return the tree of subnets based on $subnet.prefix$;
end function

the same prefix substructure represented by the parent node. And two non-leaf nodes with the same parent node have the same smaller prefix substructure.

After building the feature cache tree, we evaluate all the subnets in the depth-first order. When we traverse to node N, we cache the output feature of that node in memory. Then, when evaluating the descendant leaf nodes (subnets) of N, we can reuse the cached feature. After evaluating all descendant leaf nodes (subnets) of N, we can release the feature from memory since it won't be reused by later subnets. As a result, the number of saved features is no more than the depth of the tree, and we can adjust the depth of the tree to control the size of the feature cache.

Another problem of model evaluation is that testing the models one by one may lead to too frequent data I/O operations. Thus, we adopt *batch-wise model group evaluation*, *i.e*. loading a batch of data and evaluating all candidate subnets using the batch. The performance of the subnets is the average of them on all batches.

5.3 Dynamic Model Update

The optimal subnet found by search is used in the target environment for serving. However, the subnet may become suboptimal at runtime upon environment change.

AdaptiveNet deals with environment change by dynamically paging in and paging out alternative blocks. In order to provide subnets of different latency-accuracy trade-offs, we maintain a subnet pool during searching (Section 5.1) and save the $[arch, latency, accuracy]$ tuples of all the searched subnets, where $arch$ denotes the encoded architecture of the subnet. After searching, we save the subnet architectures that achieved the highest accuracy at different levels of latency (within the latency range $[T_{budget} - \Delta T, T_{budget} + \Delta T]$). For each of these subnet architectures, we save the relative latency as compared with the current optimal subnet.

At runtime, a latency monitor runs to detect the latency change of the running model. When the inference latency exceeds the budget, the latency monitor reports the latency scaling ratio $r = \frac{actual\ latency}{estimated\ latency}$, and searches in the subnet pool to find the subnet that achieves the highest accuracy within the scaled latency budget T_{budget}/r. Similarly, when the actual latency is smaller than the largest relative latency in the subnet pool, the monitor also replaces the running model with a better one. If the environment change is too significant and no subnet in the pool can fulfill the latency budget, we restart the search process to obtain the new optimal subnet and subnet pool.

6 IMPLEMENTATION

We implement our method using Python and Java. The on-cloud elastification part and on-device searching part are developed with PyTorch and PyTorch Mobile [40].

Handing two-stage models. Some deep learning applications such as object detection and semantic segmentation often require two-stage training, *e.g*. pretraining the backbone on ImageNet [8] and fine-tuning on the smaller task dataset. When a DNN model needs to be trained on two datasets, AdaptiveNet uses a two-stage elastification strategy. Let $\mathcal{N}$ denote the DNN model well-trained on two datasets $\{D_1, D_2\}$ in order. We first elasticize the backbone of $\mathcal{N}$ and train the newly added branches on D_1 based on feature-based distillation (Section 4.2.1) method. After distillation, we connect the elasticized backbone to the head of $\mathcal{N}$ to make it a supernet, and further train it on D_2 (Section 4.2.2).

Devices with limited memory. The supernet generated by our method is about 2×-5× larger than the pretrained model, which may not fit in the memory of some low-end devices. We use block-wise loading and inference to reduce the memory overhead. Specifically, only the blocks required by the current subnet are loaded into the memory during searching, and others are retained in the disk. Therefore, AdaptiveNet requires no more memory in the on-edge stage than that required by the optimal subnet.

7 EVALUATION

We conduct experiments to answer the following research questions: (1) Is AdaptiveNet able to generate models with better latency-accuracy tradeoffs? (§7.2, §7.3) (2) Can AdaptiveNet utilize the edge data distribution? (§7.4) (3) What's the efficiency of AdaptiveNet in both on-cloud and on-edge stages? (§7.5, §7.6)

7.1 Experimental Setup

Edge environments. We use three edge devices including an Android Smartphone (Xiaomi 12) with Snapdragon 8 Gen 1 CPU and 8 GB memory, a Jetson Nano with 4 GB memory, and an edge server with NVIDIA 3090 Ti with 24 GB GPU memory. The batch sizes are set to 1, 1, and 32 on the three devices to simulate real workloads. We use different latency budgets to simulate intra-device hardware diversity. The data distribution diversity is not considered in most experiments to fairly compare with the baselines. In Section 7.4, we use Dirichlet distribution to generate edge data, the same setting used in most Federated Learning research [17, 31, 68].

Baselines. LegoDNN [14] and NestDNN [10] are the most relevant baselines of our work. LegoDNN [14] is a pruning-based, block-grained technique for model scaling. NestDNN [10] generates multi-capacity DNN models using filter pruning and recovering methods. Since the source code of NestDNN

is unavailable and it underperforms LegoDNN [14], we conduct a detailed comparison with LegoDNN. We also include three methods that can be used for on-device model generation, including Slimmable Networks [60, 61], FlexDNN [9] and SkipNet [53]. Slimmable Networks [60, 61] design models whose widths can be flexibly changed without retraining, FlexDNN [9] is an input-adaptive method that supports early exits, and SkipNet [53] is a representative dynamic neural network that can dynamically switch different routes for different inputs. We adapt SkipNet [53] by letting it search for an optimal fixed route on the target device as the generated model. And we adapt FlexDNN [9] by specifying an early exit layer for each latency budget. We also compare with the EfficientNetV2 series [46], which are examples of state-of-the-art models generated by on-cloud NAS.

Tasks, Models, and Datasets. We evaluate the performance of AdaptiveNet on three common vision tasks.

- **Image classification** aims to recognize the category of an image. We select three popular classification models, MobileNetV2 [42], ResNet50 [19], and ResNet101 [19] to represent small, middle, and large models. The dataset used in this task is ImageNet2012 [8].
- **Object detection** aims to detect objects in an image, predicting the object bounding boxes and categories. We choose EfficientDet [47] with ResNet50 [19] backbone as the detection model, which is one of the top-performing detection models, and COCO2017 [32] as the dataset. The performance of detection models is measured by mean average precision over Intersection over Union threshold 0.5 (mAP@0.5).
- **Semantic segmentation** aims to predict the class label of each pixel in an image. We choose FPN [33] model with ResNet50 [19] encoder pretrained on ImageNet2012 [8]. The dataset is CamVid [2], a road scene understanding dataset. The performance is measured by Mean Intersection over Union (mIoU).

7.2 General Model Scaling Performance

We first evaluate the quality of models generated by our method and the baselines. Specifically, we elasticize MobileNetV2, ResNet50, and ResNet101 which represent small, medium, and large models respectively. For small models, we elasticize them into supernets containing five types of replaceable blocks $\{B_i^{(1)}, B_i^{(2)}, B_i^{(0)}, B_i^{(-1)}, B_i^{(-2)}\}$. The pruning rate of $B_i^{(-1)}, B_i^{(-2)}$ are 0.5 and 0.25 respectively. For medium and large models such as ResNet50 and ResNet101, we elasticize them into supernets that only contain original and merging optional blocks $\{B_i^{(1)}, B_i^{(2)}, B_i^{(0)}\}$. After elasticizing, the supernets contain $2.58 \times 10^8, 1.06 \times 10^5, 2.57 \times 10^{17}$ subnets, respectively. To make a fair comparison, we divide the validation set into two subsets, one smaller subset (3000 images) to search for the optimal subnet under 10 latency budgets, and the rest to evaluate the optimal subnet.

The result is displayed in Figure 7, AdaptiveNet achieves higher accuracy than baseline approaches at almost every latency budget, and increases accuracy by 10.44% and 28.03% on average compared to LegoDNN with 90% and 70% latency budget respectively. This is because our elastification creates better search space of subnets and the two-stage training technique allows subnets to learn from both the original pretrained model and the labels. Thus, AdaptiveNet can outperform LegoDNN which only trains the descendent blocks to mimic the original blocks.

Besides, we observe that AdaptiveNet outperforms the baseline models more at a lower latency budget. At the 60% and 80% latency budget, AdaptiveNet achieves 42.53% and 29.16% higher accuracy on average respectively. This is because our approach includes merging two or more blocks into one replacement block compared to pruning-based model scaling techniques. Such block merging can save more latency with a smaller loss of accuracy than high-ratio pruning.

We also notice that the gap between AdaptiveNet and Slimmable Networks is small on smartphones and 3090 Ti. The main reason is that slimmed networks can better utilize the computational resources on such devices. However, because the slimmable models are based on custom backbones, they cannot support SOTA pretrained models and are not flexible for normal developers to use.

Further, AdaptiveNet can be used with multiple pretrained models to achieve more wide-range and fine-grained trade-offs. Figure 9(a) shows the performance of models generated from two oracle EfficientNetV2 models, where AdaptiveNet provides over 20 meaningful latency-accuracy trade-offs between the oracle models. Thus, developers can use AdaptiveNet as an effective supplement to manually-created or cloud-generated models to offer more choices for the edge with little overhead (dozens of hours).

7.3 Performance on Other Tasks

We also test AdaptiveNet on object detection and semantic segmentation to evaluate its generalizability and performance on complex two-stage tasks (pretrained on ImageNet2012 [8], fine-tuned on COCO2017 [32] and CamVid [2]). Our object detection model, EfficientDet [47], consists of a backbone, neck, and head, among which the backbone takes up most of the inference latency (more than 90% according to our measurements), thus we only elasticize the backbones. For the same reason, we only elasticize the encoder of FPN [33]. Since the official code of our baseline LegoDNN [14] on object detection and semantic segmentation cannot run properly, we implement the training process of LegoDNN [14] on both tasks. To make fair comparisons, AdaptiveNet and LegoDNN

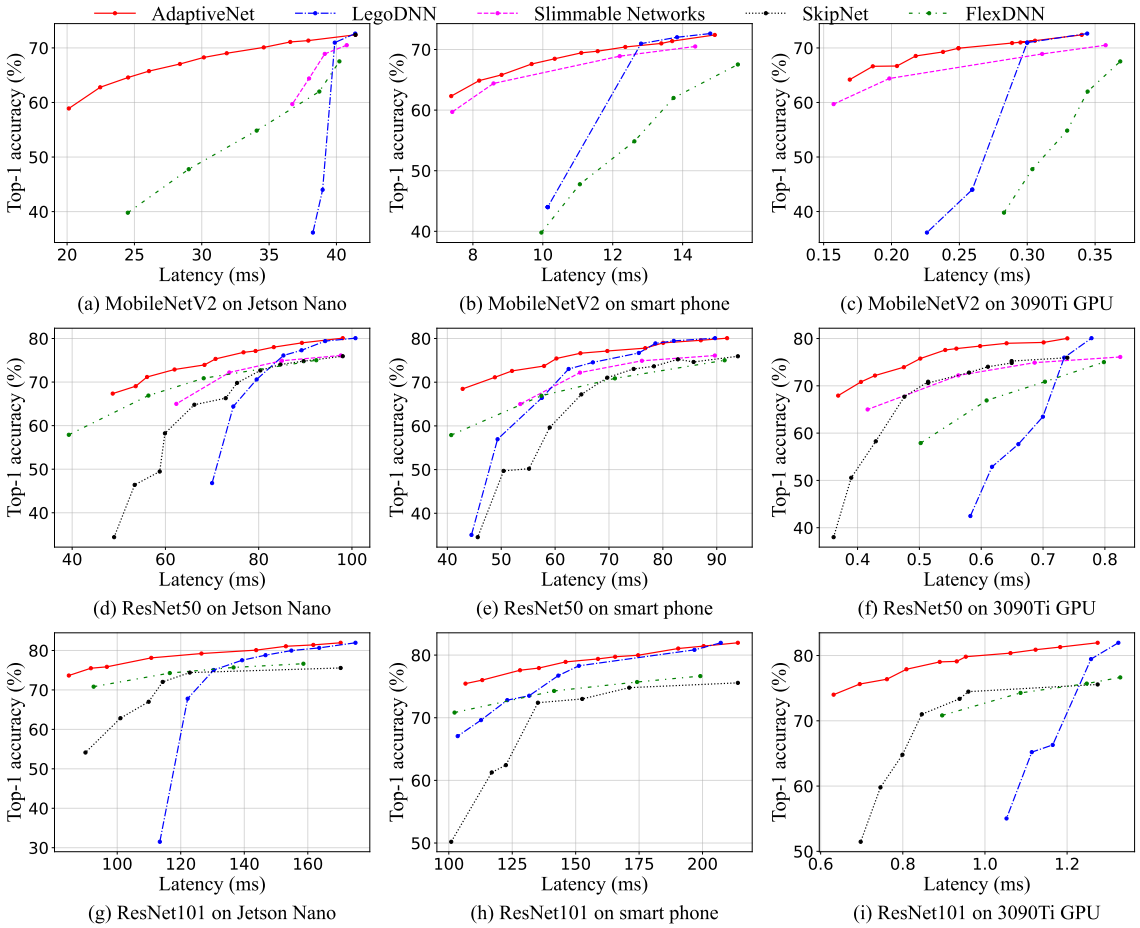


Figure 7: The latency-accuracy tradeoffs of models generated by different techniques on the target devices.

[14] start from the same pretrained model and train for the same GPU hours. After training, we randomly sample the same number (500) of subnets for both tasks and evaluate them on the test set.

The results are shown in Figure 8. Similar to the classification tasks, AdaptiveNet achieves reasonable scaling performance and outperforms the baseline. Some of the FPN subnets can even achieve better tradeoffs than the original pretrained model, which is because the original model is over-fitted. Our subnets generated by merging some original blocks together can reduce the parameter size of the original model, which reduces over-fitting and improves accuracy.

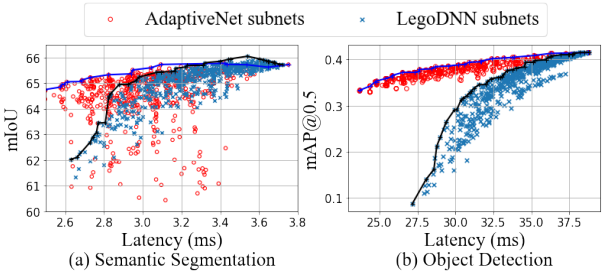
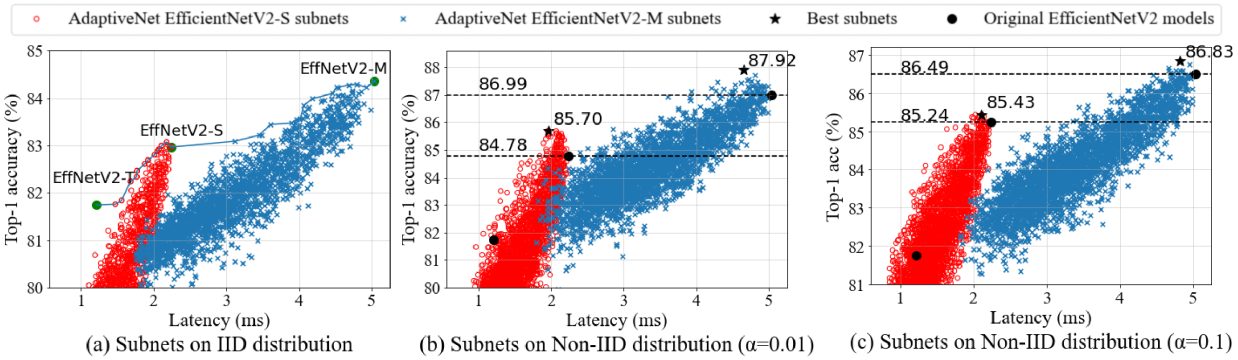


Figure 8: Quality of models generated for detection and segmentation tasks.

7.4 Impact of Edge Data Distribution Shift

Since AdaptiveNet generates the model directly on the target device, it can utilize edge data as compared with on-cloud NAS. We examine the quality of models generated by AdaptiveNet on different edge datasets simulated with Dirichlet distributions. The results are shown in Figure 9.

We notice that AdaptiveNet can outperform the EfficientNetV2 models that are generated by extensive on-cloud NAS [46] on unbalanced edge data distributions. Some of the subnets may improve the top-1 accuracy by 1.07% while saving 7.71% latency at the same time, which is a hard-won improvement since the original model has achieved excellent



(a) Subnets on IID distribution (b) Subnets on Non-IID distribution (α=0.01) (c) Subnets on Non-IID distribution (α=0.1)

Figure 9: Quality of models generated by AdaptiveNet on different edge data distribution in comparison with cloud-generated oracle models (Latency measured on NVIDIA 3090 Ti GPU).

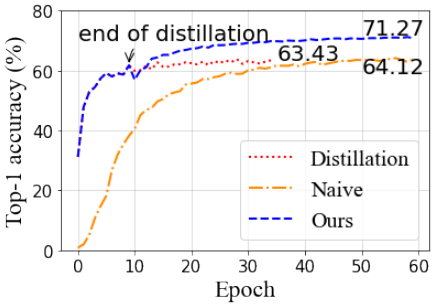


Figure 10: Training efficiency of on-cloud elastification.

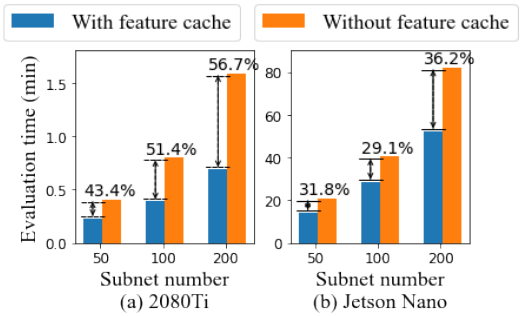


Figure 11: Speed of evaluating a group of subnets.

performance (with 86.99% top-1 accuracy and 5.03ms latency). We also observe that the advantage of AdaptiveNet increases when the data distribution is more unbalanced.

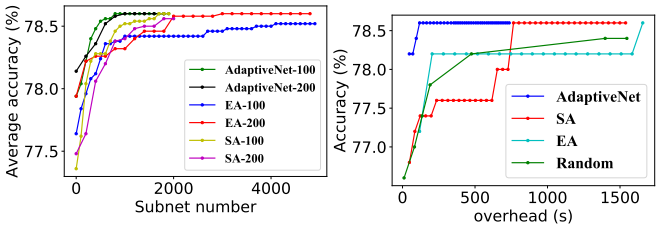
Given the fact that the edge data distributions are usually different from training ones [26], we believe the post-deployment model generation mechanism of AdaptiveNet is a more promising direction to seek in edge AI scenarios.

7.5 On-cloud Training Performance

We examine the efficiency and effectiveness of our on-cloud elastification stage. We compare our supernet training method with a supervised training baseline (*i.e.* our method without distillation) and a distillation-only baseline (*i.e.* our model without whole-model tuning) and plot the progressive top-1 accuracy on ImageNet in Figure 10. Although all of the training methods can converge after 50 epochs, our two-step training technique is 7.15% and 7.84% higher than using supervised training only and distillation only. Our supernet training also converges faster than the baselines with only 60 epochs (about 13 hours), which is also significantly faster than on-cloud NAS (>1200 GPU hours [3]).

7.6 On-device Adaptation Efficiency

In this section, we evaluate the performance of the on-device search in AdaptiveNet. Most of the subnet search methods



(a) Optimal accuracy achieved with different num of subnets. **(b) Optimal accuracy achieved with different search time.**

Figure 12: Comparison of search efficiency between different methods.

in conventional NAS are too heavy for edge devices (*e.g.* reinforcement learning [35, 69] and gradient-based methods [5, 36]). So we choose normal evolutionary search [41] and simulated annealing [34] as our baselines. To examine the effectiveness of our method, we conduct two ablation experiments and one end-to-end experiment. All the experiments use the same 500 images for searching.

Figure 11 shows the acceleration percentage of our reuse-based model evaluation method (Section 5.2). We compare the time spent to evaluate 50, 100, and 200 subnets respectively. Our method saves up to 56.7% and 36.2% of search overhead

Table 3: Size (MB) of AdaptiveNet-generated supernets and their corresponding pretrained models.

Model	Pretrained model	Supernet
MobileNetV2	14.20	32.88
ResNet50	102.48	381.66
ResNet101	178.71	503.82

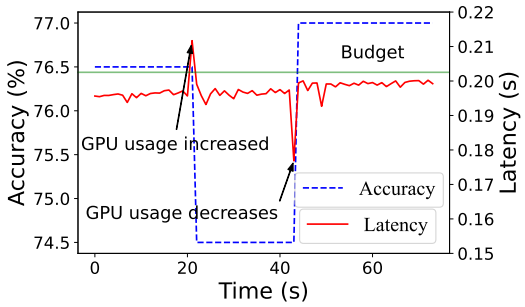


Figure 13: Demonstration of dynamic model update.

compared to normal evaluation pipeline, and consumes 100-500MB of memory. It is notable that the memory cost of our method is controllable by adjusting the depth of the subnet tree. If there is no space for feature maps, we can set the depth to 0, which will reduce the GPU memory overhead to zero with some sacrifice of search efficiency.

Figure 12a shows the benefits of our model-guided search strategy. To achieve the same average accuracy, AdaptiveNet, evolutionary algorithm, and simulated annealing need to try 800, 3100, and 1800 subnets respectively. Figure 12b shows the end-to-end search efficiency comparison between AdaptiveNet and baselines on NVIDIA 2080 Ti. We conduct three individual experiments with a population size of 50, 100, and 200, respectively, and show the best results for each strategy. AdaptiveNet, simulated annealing algorithm, and evolutionary algorithm to find optimal subnet in 117.6, 765.6, and 1656.9 seconds respectively, indicating that our method can improve search efficiency by more than 80%.

Network transmission overhead. Table 3 shows the size of AdaptiveNet and pretrained models. Although AdaptiveNet increases the size of models, we believe it can actually save network overhead. AdaptiveNet only needs to transmit the supernet to edge devices once, which is 1.32×-2.72× larger than the original pre-trained model. However, to achieve similar performance, conventional model deployment approaches have to collect device information and re-transmit the model when the edge environment changes, which is $n\times$ larger than the original model, where n is the time of changes.

Real-time Model Update Efficiency. We further test the performance of our dynamic model update module and present the result in Figure 13. We choose ResNet50 [19] as our pretrained model and the experiment is conducted on NVIDIA 2080 Ti. We adjust the GPU usage by running and killing MobileNetV2 inference processes. Our dynamic model update is fast and responsive. After the latency budget is exceeded or under-utilized, AdaptiveNet can find the optimal model and recover the latency within 1 second. Specifically, we obtain a pool of optimal subnets for a range of latency budgets during the on-device search. The runtime update module only needs to switch to the proper model, instead of searching from scratch. Thus, it should be easy for our method to catch up with the workload dynamics. Besides, if the real workload is very dynamic, we can control the update frequency to avoid overreaction.

8 DISCUSSION

An issue that may be a concern in the on-device model generation is the need for labeled edge data, which might be difficult if the data is not auto-labeled (like in many unsupervised tasks [18, 62]). Such a dataset can be generated by querying an oracle model with unlabelled edge data. Letting the cloud send public data to the edge is also an option, although the edge data characteristics will not be utilized in this way.

Although AdaptiveNet is mainly evaluated on vision tasks, it should be able to generalize to other tasks such as NLP. Transformer models [48] are also composed of repeated blocks such as encoders and decoders, so we should be able to elasticize them into supernets and choose the optimal subnet from them at edge devices.

We also want to discuss the relationship between AdaptiveNet and on-device training, which can be used to improve model quality after deployment. First, on-device training typically requires heavy computation and sufficient training data to be effective, which AdaptiveNet does not require. Besides, on devices with good training conditions, a better architecture found by AdaptiveNet can also be beneficial.

9 CONCLUSION

This paper proposes a novel approach for on-device, post-deployment, and environment-aware model architecture generation. The approach is implemented as an end-to-end system equipped with on-cloud model elastification and on-device model adaptation techniques. Experiments have demonstrated the remarkable model quality and model generation efficiency of our method. Developers can scale their AI applications to diverse and dynamic edge environments with our system by simply specifying a pretrained model.

ACKNOWLEDGEMENT

This work is supported by the National Key R&D Program of China (No.2022YFF0604501) and NSFC (No.62272261).

REFERENCES

[1] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. 2017. Adaptive Neural Networks for Efficient Inference. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 527–536. https://proceedings.mlr.press/v70/bolukbasi17a.html

[2] Gabriel J. Brostow, Jamie Shotton, Julien Fauqueur, and Roberto Cipolla. 2008. Segmentation and Recognition Using Structure from Motion Point Clouds. In *Computer Vision – ECCV 2008*, David Forsyth, Philip Torr, and Andrew Zisserman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–57.

[3] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once for All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations*. https://arxiv.org/pdf/1908.09791.pdf

[4] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *8th International Conference on Learning Representations, ICLR 2020*.

[5] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations*. https://arxiv.org/pdf/1812.00332.pdf

[6] Yimin Chen, Jingchao Sun, Xiaocong Jin, Tao Li, Rui Zhang, and Yanchao Zhang. 2017. Your face your heart: Secure mobile face authentication with photoplethysmograms. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057220

[7] Hsin-Rung Chou, Jia-Hong Lee, Yi-Ming Chan, and Chu-Song Chen. 2019. Data-specific Adaptive Threshold for Face Recognition and Authentication. In *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*. IEEE, 153–156.

[8] J. Deng, K. Li, M. Do, H. Su, and L. Fei-Fei. 2009. Construction and Analysis of a Large Scale Image Ontology. Vision Sciences Society.

[9] Biyi Fang, Xiao Zeng, Faen Zhang, Hui Xu, and Mi Zhang. 2020. FlexDNN: Input-Adaptive On-Device Deep Learning for Efficient Mobile Vision. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. 84–95. https://doi.org/10.1109/SEC50012.2020.00014

[10] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (2018).

[11] Peizhen Guo, Bo Hu, and Wenjun Hu. 2021. Mistify: Automating DNN Model Porting for On-Device Inference at the Edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 705–719. https://www.usenix.org/conference/nsdi21/presentation/guo

[12] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. 2020. Single Path One-Shot Neural Architecture Search with Uniform Sampling. In *Computer Vision – ECCV 2020*, Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm (Eds.). Springer International Publishing, Cham, 544–560.

[13] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. 2019. MOSAIC: Heterogeneity-, Communication-, and Constraint-Aware Model Slicing and Execution for Accurate and Efficient Inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 165–177. https://doi.org/10.1109/PACT.2019.00021

[14] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y. Chen. 2021. LegoDNN: Block-Grained Scaling of Deep Neural Networks for Mobile Vision. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 406–419. https://doi.org/10.1145/3447993.3483249

[15] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *arXiv: Computer Vision and Pattern Recognition* (2016).

[16] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2022. Dynamic Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 11 (2022), 7436–7456. https://doi.org/10.1109/TPAMI.2021.3117837

[17] Chaoyang He, Songze Li, Jinhyun So, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. 2020. FedML: A Research Library and Benchmark for Federated Machine Learning. *Advances in Neural Information Processing Systems, Best Paper Award at Federate Learning Workshop* (2020).

[18] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum Contrast for Unsupervised Visual Representation Learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 9726–9735. https://doi.org/10.1109/CVPR42600.2020.00975

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[20] Yuze He, Li Ma, Zhehao Jiang, Yi Tang, and Guoliang Xing. 2021. VI-Eye: Semantic-Based 3D Point Cloud Registration for Infrastructure-Assisted Autonomous Driving. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 573–586. https://doi.org/10.1145/3447993.3483276

[21] Andrew Howard, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. 2019. Searching for MobileNetV3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 1314–1324. https://doi.org/10.1109/ICCV.2019.00140

[22] Andrew Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, M. Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv: Computer Vision and Pattern Recognition* (2017).

[23] Sian-Yao Huang and Wei-Ta Chu. 2021. Searching by Generating: Flexible and Efficient One-Shot NAS with Architecture Generator. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*.

[24] Daejin Jung, Wonkyung Jung, Byeongho Kim, Sunjung Lee, Wonjong Rhee, and Jung Ho Ahn. 2018. Restructuring Batch Normalization to Accelerate CNN Training. *CoRR* abs/1807.01702 (2018). arXiv:1807.01702 http://arxiv.org/abs/1807.01702

[25] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages. https://doi.org/10.1145/3302424.3303950

[26] Viraj Kulkarni, Milind Kulkarni, and Aniruddha Pant. 2020. Survey of Personalization Techniques for Federated Learning. In *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. 794–797. https://doi.org/10.1109/WorldS450073.

2020.9210355

[27] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D. Lane. 2021. Adaptive Inference through Early-Exit Networks: Design, Challenges and Directions. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning (EMDL'21)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3469116.3470012

[28] Stefanos Laskaridis, Stylianos I. Venieris, Hyeji Kim, and Nicholas D. Lane. 2020. HAPI: Hardware-Aware Progressive Inference. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[29] Changlin Li, Jiefeng Peng, Liuchun Yuan, Guangrun Wang, Xiaodan Liang, Liang Lin, and Xiaojun Chang. 2020. Block-Wisely Supervised Neural Architecture Search With Knowledge Distillation. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 1986–1995. https://doi.org/10.1109/CVPR42600.2020.00206

[30] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning Filters for Efficient ConvNets. *CoRR* abs/1608.08710 (2016). arXiv:1608.08710 http://arxiv.org/abs/1608.08710

[31] Qinbin Li, Bingsheng He, and Dawn Song. 2021. Model-Contrastive Federated Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[32] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. *CoRR* abs/1405.0312 (2014). arXiv:1405.0312 http://arxiv.org/abs/1405.0312

[33] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. 2017. Feature Pyramid Networks for Object Detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 936–944. https://doi.org/10.1109/CVPR.2017.106

[34] Chia-Hsiang Liu, Yu-Shin Han, Yuan-Yao Sung, Yi Lee, Hung-Yueh Chiang, and Kai-Chiang Wu. 2021. FOX-NAS: Fast, On-device and Explainable Neural Architecture Search. *CoRR* abs/2108.08189 (2021). arXiv:2108.08189 https://arxiv.org/abs/2108.08189

[35] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive Neural Architecture Search. In *Computer Vision – ECCV 2018*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer International Publishing, Cham, 19–35.

[36] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable Architecture Search. *CoRR* abs/1806.09055 (2018). arXiv:1806.09055 http://arxiv.org/abs/1806.09055

[37] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning Efficient Convolutional Networks through Network Slimming. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2755–2763. https://doi.org/10.1109/ICCV.2017.298

[38] Shie Mannor, Dori Peleg, and Reuven Rubinstein. 2005. The Cross Entropy Method for Classification *(ICML '05)*. Association for Computing Machinery, New York, NY, USA, 561–568. https://doi.org/10.1145/1102351.1102422

[39] Jinhwan Park, Yoonho Boo, Iksoo Choi, Sungho Shin, and Wonyong Sung. 2018. Fully Neural Network Based Speech Recognition on Mobile and Embedded Devices. In *NeurIPS*.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[41] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 587, 10 pages. https://doi.org/10.1609/aaai.v33i01.33014780

[42] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[43] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. 2019. And the Bit Goes Down: Revisiting the Quantization of Neural Networks. *CoRR* abs/1907.05686 (2019). arXiv:1907.05686 http://arxiv.org/abs/1907.05686

[44] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2815–2823. https://doi.org/10.1109/CVPR.2019.00293

[45] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 6105–6114. https://proceedings.mlr.press/v97/tan19a.html

[46] Mingxing Tan and Quoc Le. 2021. EfficientNetV2: Smaller Models and Faster Training. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, 10096–10106. https://proceedings.mlr.press/v139/tan21a.html

[47] Mingxing Tan, Ruoming Pang, and Quoc V. Le. 2019. EfficientDet: Scalable and Efficient Object Detection. *CoRR* abs/1911.09070 (2019). arXiv:1911.09070 http://arxiv.org/abs/1911.09070

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[49] Hui Wang, Hanbin Zhao, Xi Li, and Xu Tan. 2018. Progressive Blockwise Knowledge Distillation for Neural Network Acceleration. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*. AAAI Press, 2769–2775.

[50] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[51] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: Scalable and Efficient Deep-Learning Inference on Asymmetric Mobile CPUs. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/3447993.3448625

[52] Quan Wang, Ignacio Lopez Moreno, Mert Saglam, Kevin Wilson, Alan Chiao, Renjie Liu, Yanzhang He, Wei Li, Jason Pelecanos, Marily Nika, et al. 2020. VoiceFilter-Lite: Streaming targeted voice separation for on-device speech recognition. *arXiv preprint arXiv:2009.04323* (2020).

[53] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. 2018. SkipNet: Learning Dynamic Routing in Convolutional Networks. In *The European Conference on Computer Vision (ECCV)*.

[54] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 2082–2090.

[55] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10726–10734. https://doi.org/10.1109/CVPR.2019.01099

[56] Hao Wu, Xuejin Tian, Minghao Li, Yunxin Liu, Ganesh Ananthanarayanan, Fengyuan Xu, and Sheng Zhong. 2021. PECAM: Privacy-Enhanced Video Streaming and Analytics via Securely-Reversible Transformation. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/3447993.3448618

[57] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. 2018. BlockDrop: Dynamic Inference Paths in Residual Networks. In *CVPR*.

[58] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 129–144. https://doi.org/10.1145/3241539.3241563

[59] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In *The European Conference on Computer Vision (ECCV)*.

[60] Jiahui Yu and Thomas S. Huang. 2019. Universally Slimmable Networks and Improved Training Techniques. *CoRR* abs/1903.05134 (2019). arXiv:1903.05134 http://arxiv.org/abs/1903.05134

[61] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas S. Huang. 2018. Slimmable Neural Networks. *CoRR* abs/1812.08928 (2018). arXiv:1812.08928 http://arxiv.org/abs/1812.08928

[62] Seunghak Yu, Nilesh Kulkarni, Haejun Lee, and Jihie Kim. 2018. On-Device Neural Language Model Based Word Prediction. In *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations*. Association for Computational Linguistics, Santa Fe, New Mexico, 128–131. https://aclanthology.org/C18-2028

[63] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. In *European Conference on Computer Vision (ECCV)*.

[64] Huanhuan Zhang, Anfu Zhou, Yuhan Hu, Chaoyue Li, Guangping Wang, Xinyu Zhang, Huadong Ma, Leilei Wu, Aiyun Chen, and Changhui Wu. 2021. Loki: Improving Long Tail Performance of Learning-Based Real-Time Video Adaptation by Fusing Rule-Based Models. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/3447993.3483259

[65] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. 2021. EMP: Edge-Assisted Multi-Vehicle Perception. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 545–558. https://doi.org/10.1145/3447993.3483242

[66] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 6848–6856. https://doi.org/10.1109/CVPR.2018.00716

[67] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. 2020. Towards Unified INT8 Training for Convolutional Neural Network. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 1966–1976. https://doi.org/10.1109/CVPR42600.2020.00204

[68] Zhuangdi Zhu, Junyuan Hong, and Jiayu Zhou. 2021. Data-Free Knowledge Distillation for Heterogeneous Federated Learning. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, 12878–12889. https://proceedings.mlr.press/v139/zhu21b.html

[69] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *CoRR* abs/1611.01578 (2016). arXiv:1611.01578 http://arxiv.org/abs/1611.01578