ADAPTIVE ON-DEVICE DEEP LEARNING SYSTEMS

By

Biyi Fang

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Electrical Engineering — Doctor of Philosophy

2019

ABSTRACT

ADAPTIVE ON-DEVICE DEEP LEARNING SYSTEMS

By

Biyi Fang

Mobile systems such as smartphones, drones, and augmented-reality headsets are revolutionizing our lives. On-device deep learning is regarded as the key enabling technology for realizing their full potential. This is because communication with cloud adds additional latency or cost, or the applications must operate even with intermittent internet connectivity. The key to achieving the full promise of these mobile vision systems is effectively analyzing the streaming video frames. However, processing streaming video frames taken in mobile settings is challenging in two folds. First, the processing usually involves multiple computer vision tasks. This multi-tenant characteristic requires mobile vision systems to concurrently run multiple applications that target different vision tasks. Second, the context in mobile settings can be frequently changed. This requires mobile vision systems to be able to switch applications to execute new vision tasks encountered in the new context.

In this article, we fill this critical gap by proposing NestDNN, a framework that enables resource-aware multi-tenant on-device deep learning for continuous mobile vision. NestDNN enables each deep learning model to offer flexible resource-accuracy trade-offs. At runtime, it dynamically selects the optimal resource-accuracy trade-off for each deep learning model to fit the model's resource demand to the system's available runtime resources. In doing so, NestDNN efficiently utilizes the limited resources in mobile vision systems to jointly maximize the performance of all the concurrently running applications.

Although NestDNN is able to efficiently utilize the resource by being resource-aware, it essentially treats the content of each input image equally and hence does not realize the

full potential of such pipelines. To realize its full potential, we further propose FlexDNN, a novel content-adaptive framework that enables computation-efficient DNN-based on-device video stream analytics based on early exit mechanism. Compared to state-of-the-art early exit-based solutions, FlexDNN addresses their key limitations and pushes the state-of-the-art forward through its innovative fine-grained design and automatic approach for generating the optimal network architecture.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Mobile vision systems such as smartphones, drones, and augmented reality headsets are ubiquitous today. As AI chipsets emerge, these systems begin to support on-device live video stream processing, which is the enabler of a wide range of continuous mobile vision applications. This trend is fueled by the recent advancement in deep learning (i.e., Deep Neural Networks (DNNs)) [2] due to its success in achieving impressively high accuracies in many important vision tasks.

Continuous mobile vision applications require continuously processing the streaming video input, and returning the processing results with low latency. Unfortunately, DNNs are known to be memory and computationally intensive [3], and high computational demand translates into high processing latency and high energy consumption. Thus, considering mobile systems are constrained by limited resources in terms of computation, memory, and battery capacities, reducing resource demands of DNNs is crucial to realizing the full potential of continuous mobile vision applications.

The key to achieving the full promise of these mobile vision systems is effectively analyzing the streaming video frames. However, processing streaming video frames taken in mobile settings is challenging in two folds. First, the processing usually involves multiple computer vision tasks. This multi-tenant characteristic requires mobile vision systems to concurrently run multiple applications that target different vision tasks. Second, the context in mobile settings can be frequently changed. This requires mobile vision systems to be able to switch

applications to execute new vision tasks encountered in the new context.

In §3, we fill this critical gap by proposing NestDNN, a framework that enables resource-aware multi-tenant on-device deep learning for continuous mobile vision. It takes the dynamics of runtime resources in a mobile vision system into consideration, and dynamically selects the optimal resource-accuracy trade-off and resource allocation for each of the concurrently running deep learning models to jointly maximize their performance. We evaluate NestDNN using six mobile vision applications that target some of the most important vision tasks for mobile vision systems. Results have shown it outperforms the resource-agnostic counterpart significantly. From NestDNN, we discover yet another problem that it does not solve.

Although NestDNN is able to achieve resource-aware multi-tenant on-device deep learning, it essentially treats the content of each input image equally. In §4, to realize the full potential of DNN-based processing pipeline, we further propose FlexDNN, a novel content-adaptive framework that enables computation-efficient DNN-based on-device video stream analytics based on early exit mechanism. Compared to state-of-the-art early exit-based solutions, FlexDNN addresses their key limitations and pushes the state-of-the-art forward through its innovative fine-grained design and automatic approach for generating the optimal network architecture. We use FlexDNN to build three computation-efficient continuous mobile vision applications on top of MobileNets. Our evaluation results show that FlexDNN significantly outperforms both computation-efficient content-agnostic and state-of-the-art content-adaptive approaches in reducing computational consumption by a large margin.

Lastly, we conclude this report in §5.

Chapter 2

Related Work

Our related work contains four parts: 1) Mobile Sensing Systems; 2) Deep Neural Network Model Compression; 3) Continuous Mobile Vision; and 4) content-adaptive video stream analytics.

Mobile Sensing Systems. Our work is also broadly related to research in mobile sensing systems. Prior mobile sensing systems have explored a variety of sensing modalities that have enabled a wide range of innovative applications. Among them, accelerometer, microphone and physiological sensors are some of the mostly explored sensing modalities. For example, Mokaya *et al.* developed an accelerometer-based system to sense skeletal muscle vibrations for quantifying skeletal muscle fatigue in an exercise setting [4]. Nirjon *et al.* developed MusicalHeart [5] which integrated a microphone into an earphone to extract heartbeat information from audio signals. Nguyen *et al.* designed an in-ear sensing system in the form of earplugs that is able to capture EEG, EOG, and EMG signals for sleep monitoring [6]. Recently, researchers have started exploring using wireless radio signal as a contactless sensing mechanism. For example, Wang *et al.* developed WiFall [7] that used wireless radio signal to detect accidental falls. Fang *et al.* used radio as a single sensing modality for integrated activities of daily living and vital sign monitoring [8]. In this work, we explore infrared light as a new sensing modality in the context of ASL translation. It complements existing mobile sensing systems by providing a non-intrusive and high-resolution sensing scheme. We regard

this work as an excellent example to demonstrate the usefulness of infrared sensing for mobile systems. With the incoming era of virtual/augmented reality, we envision infrared sensing will be integrated into many future mobile systems such as smartphones and smart glasses.

Deep Neural Network Model Compression. Model compression for deep neural networks has attracted a lot of attentions in recent years due to the imperative demand on running deep learning models on mobile systems. One of the most prevalent methods for compressing deep neural networks is pruning. Han *et al.* [9] proposed a parameter pruning method that removes node connections with small weights. Although this method is effective at reducing model sizes, it does not effectively reduce computational costs. To overcome this problem, Li *et al.* [1] proposed a filter pruning method that has achieved up to 38% reduction in computational cost. Our work also focuses on compressing deep neural networks via filter pruning. Our proposed filter pruning approach outperforms the state-of-the-art. Moreover, unlike existing model compression methods which produce pruned models with fixed resource-accuracy trade-offs, our proposed multi-capacity model is able to provide dynamic resource-accuracy trade-offs. This is similar to the concept of dynamic neural networks in the deep learning literature [10–12].

Continuous Mobile Vision. The concept of continuous mobile vision was first advanced by Bahl *et al.* [13]. The last few years have witnessed many efforts towards realizing the vision of continuous mobile vision [14–17]. In particular, in [14], LiKamWa *et al.* proposed a framework named Starfish, which enables efficient running concurrent vision applications on mobile devices by sharing common computation and memory objects across applications. Our work is inspired by Starfish in terms of sharing. By sharing parameters among descendent models, our proposed multi-capacity model has a compact memory footprint and incurs

little model switching overhead. Our work is also inspired by [15]. In [15], Han *et al.* proposed a framework named MCDNN, which applies various model compression techniques to generate a catalog of model variants to provide different resource-accuracy trade-offs. However, in MCDNN, the generated model variants are independent of each other, and it relies on cloud connectivity to retrieve the desired model variant. In contrast, our work focuses on developing an on-device deep learning framework which does not rely on cloud connectivity. Moreover, MCDNN focuses on model sharing across concurrently running applications. In contrast, NestDNN treats each of the concurrently running applications independently, and focuses on model sharing across different model variants within each application.

Content-Adaptive Video Stream Analytics. FlexDNN is closely related to content-adaptive video stream analytics. In [18], the authors proposed a content-adaptive video stream analytics systems named Chameleon that dynamically changes the DNN models to adapt to the difficulty levels of the video frames. However, it requires the system to carry all the model variants with various capacities and thus does not fit resource-constrained mobile platforms. To address this limitation, BranchyNet [19] and MSDNet [20] use a single model with augmented early exits to realize the adaptation to the difficulty levels of video frames. Similar to BranchyNet and MSDNet, FlexDNN also uses a single model with augmented early exits for content adaptation. However, FlexDNN differs from these pioneer work in that it adopts a fine-grained approach to make early predictions at the granularity of filters, and it automatically generates the optimal early exit architecture and the optimal early exit insertion plan with the objective to maximize the benefit brought by early exits.

Chapter 3

Resource-Aware Multi-Tenant On-Device Deep Learning

3.1 Introduction of NestDNN

Mobile systems with onboard video cameras such as smartphones, drones, wearable cameras, and augmented-reality headsets are revolutionizing the way we live, work, and interact with the world. By processing the streaming video inputs, these mobile systems are able to retrieve visual information from the world and are promised to open up a wide range of new applications and services. For example, a drone that can detect vehicles, identify road signs, and track traffic flows will enable mobile traffic surveillance with aerial views that traditional traffic surveillance cameras positioned at fixed locations cannot provide [21]. A wearable camera that can recognize everyday objects, identify people, and understand the surrounding environments can be a life-changer for the blind and visually impaired individuals [22].

The key to achieving the full promise of these mobile vision systems is effectively analyzing the streaming video frames. However, processing streaming video frames taken in mobile settings is challenging in two folds. First, the processing usually involves *multiple* computer vision tasks. This multi-tenant characteristic requires mobile vision systems to *concurrently* run multiple applications that target different vision tasks [14]. Second, the *context* in mobile

settings can be frequently changed. This requires mobile vision systems to be able to switch applications to execute new vision tasks encountered in the new context [15].

In the past few years, deep learning (e.g., Deep Neural Networks (DNNs)) [2] has become the dominant approach in computer vision due to its capability of achieving impressively high accuracies on a variety of important vision tasks [23–25]. As deep learning chipsets emerge, there is a significant interest in leveraging the on-device computing resources to execute deep learning models on mobile systems without cloud support [26–28]. Compared to the cloud, mobile systems are constrained by limited resources. Unfortunately, deep learning is known to be resource-demanding [3]. To enable on-device deep learning, one of the common techniques used by application developers is compressing the deep learning model to reduce its resource demand at a modest loss in accuracy as trade-off [15,29]. Although this technique has gained considerable popularity and has been applied to developing state-of-the-art mobile deep learning systems [17,30–33], it has a *key drawback*: since application developers develop their applications independently, the resource-accuracy trade-off of the compressed model is predetermined based on a *static* resource budget at application development stage and is *fixed* after the application is deployed. However, the available resources in mobile vision systems at runtime are always *dynamic* because the concurrently running applications change depending on the context. As a consequence, when the resources available at runtime do not meet the resource demands of the compressed deep learning models, resource contention among concurrently running applications occurs, forcing the streaming video to be processed at a much lower frame rate. One the other hand, when extra resources at runtime become available, the compressed deep learning models cannot utilize the extra available resources to regain their sacrificed accuracies back.

In this work, we present NestDNN, a framework that takes the *dynamics of runtime*

resources into consideration to enable resource-aware multi-tenant on-device deep learning for mobile vision systems.

It replaces fixed resource-accuracy trade-offs with flexible resource-accuracy trade-offs, and dynamically selects the optimal resource-accuracy trade-off of the deep learning model at runtime to t the model's resource demand to the system's available runtime resources.

At its core, NestDNN employs a *pruning and recovery* scheme which transforms an off-the-shelf deep learning model into a single compact *multi-capacity model*. The multi-capacity model has two key features. First, the multi-capacity model is comprised of a set of sub-models. Each sub-model has a unique capacity to provide an optimized resource-accuracy trade-off. Second, unlike traditional model variants that are independent of each other, the sub-model with smaller capacity *shares* its model architecture and its model parameters with the sub-model with larger capacity, making itself *nested* inside the sub-model with larger capacity without taking extra memory space. By doing so, the multi-capacity model is able to provide various resource-accuracy trade-offs with a compact memory footprint.

The creation of multi-capacity model enables NestDNN to jointly maximize the performance of concurrent vision applications running on mobile vision systems. This possibility comes from two key *insights*. First, while a certain amount of runtime resources can be traded for an accuracy gain in some vision application, the same amount of runtime resources can be traded for a much larger accuracy gain in some other vision application. Second, some vision application does not require real-time response and thus can tolerate a relatively large inference latency (e.g., scene understanding). This presents an opportunity to reallocate some runtime resources from the latency-tolerant vision application to vision applications that need more runtime resources to meet the real-time requirement (e.g., road sign identification). NestDNN exploits these insights by incorporating the accuracy and in-

ference latency into a *cost function* for each vision application. Given the cost functions of all the concurrently running vision applications, NestDNN employs a *runtime scheduler* that selects the most suitable sub-model for each application and determines the optimal amount of runtime resources to allocate to each selected sub-model to jointly maximize the accuracy and minimize the inference latency of concurrent vision applications.

We have conducted a rich set of experiments to evaluate the performance of NestDNN. To examine the performance of the multi-capacity model, we evaluated it on six mobile vision applications that target some of the most important computer vision tasks for mobile vision systems. These applications are developed based on two widely used deep learning models – VGGNet and ResNet – and six commonly used datasets from computer vision community. To examine the performance of runtime scheduling, we implemented NestDNN and the six mobile vision applications on three smartphones. The detailed contributions and important results are summarized as follows:

- **Multi-Capacity Model.** We have designed a pruning and recovery scheme to transform an off-the-shelf deep learning model into a single compact multi-capacity model that is able to provide optimized resource-accuracy trade-offs. The pruning and recovery scheme consists of a *model pruning phase* and a *model recovery phase*. For model pruning, we have devised a state-of-the-art *filter pruning* approach named *Triplet Response Residual* (TRR) that significant reduces not only the size of a deep learning model but also its computational cost. For model recovery, we have devised an innovative *model freezing and filter growing* (i.e., *freeze-&-grow*) approach that generates the multi-capacity model in an iterative manner. Our experimental results show that the NestDNN multi-capacity model is able to provide optimized resource-accuracy trade-offs, and significantly reduce model memory footprint as well as model switching overhead.

- **Resource-Aware Scheduler.** We have designed a runtime scheduler that jointly maximizes the accuracy and minimizes the inference latency of concurrent vision applications running on mobile vision systems. The available runtime resources in mobile vision systems are dynamic and can be changed due to events such as starting new applications, closing existing applications, and application priority changes. The scheduler is able to select the most appropriate accuracy-resource trade-off for each of the concurrent vision applications and determine the amount of runtime resources to allocate to each application to optimize the scheduling objective. Our experimental results show that the NestDNN runtime scheduler outperforms the non-resource-aware counterpart on two state-of-the-art scheduling schemes, achieving as much as 4.2% increase on accuracy, 2.0× increase on frame rate and 1.7× reduction on energy consumption when running concurrent mobile vision applications.

To the best of our knowledge, NestDNN represents the first framework that enables resource-aware multi-tenant on-device deep learning for continuous mobile vision. It contributes novel techniques that address the unique challenges in continuous mobile vision. We believe that our work represents a significant step to turning the envisioned continuous mobile vision into reality [13, 14, 34].

3.2 Challenges and Our Solutions

The design of NestDNN presents a number of challenges. In this section, we describe these challenges followed by explaining how they can be effectively addressed by NestDNN.

Computational Intensity of Deep Learning Models. The foundation of NestDNN is the generation of a multi-capacity model from an off-the-shelf deep learning model. The

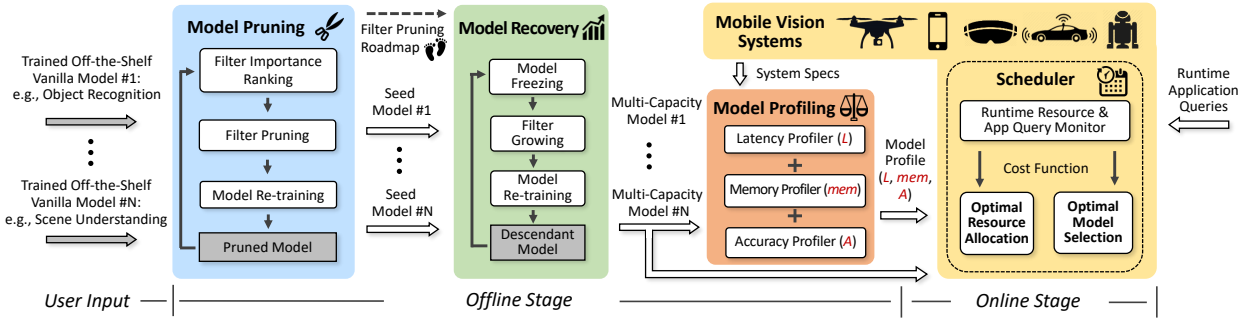

Figure 3.1: NestDNN architecture.

multi-capacity model needs to be compact and computationally lightweight such that it is able to efficiently run on resource-limited mobile systems. However, deep learning models are known to be memory and computational intensive [35, 36]. Running one deep learning model is sometimes challenging for mobile systems, let alone running multiple of them concurrently. To address this challenge, we propose a state-of-the-art deep learning model pruning approach named Triplet Response Residual (TRR). Most widely used pruning approaches focus on pruning model parameters [9, 37]. Although pruning parameters can significantly reduce model size, it does not necessarily reduce computational cost in terms of floating point operations (i.e., FLOP) [1, 38, 39], making it less useful for mobile systems which need to provide real-time services. In contrast, our approach focuses on pruning filters, which effectively reduces the size of a deep learning model and its computational cost.

Large Amount of Model Variants. One key feature of NestDNN is the provision of flexible trade-offs between accuracy and resource use. To achieve this resource-aware flexibility, one naive approach is to have all the possible model variants with various resource-accuracy trade-offs installed in the mobile system. However, since these model variants are *independent* of each other, this approach is not scalable and becomes infeasible when the mobile system concurrently runs multiple deep learning models, with each of which having multiple model variants. To address this challenge, we propose an innovative freeze-&-grow

approach that uses the filter pruning roadmap generated during the model pruning phase as the guideline to produce a single multi-capacity model with all model variants within itself. More importantly, unlike traditional model variants that are independent of each other, these model variants share model architecture and parameters with each other, making the multi-capacity model having a compact memory footprint.

Lack of Resource-Aware Scheduler. The available runtime resources in mobile vision systems are dynamic and are changed due to events such as starting new applications, closing existing applications, and application priority changes. As such, using a model with a fixed resource-accuracy trade-off for a vision application will make the application underperformed when extra resources become available. To address this problem, we propose a runtime scheduler that is resource-aware. Once runtime resources become available, the scheduler determines which resource-accuracy trade-off to be adopted for each application and how many runtime resources should be allocated for each application. As such, runtime resources are best utilized to maximize the performance of all the vision applications concurrently running on a mobile vision system.

3.3 NestDNN Overview

Figure 3.1 illustrates the architecture of NestDNN, which is split into an *offline stage* and an *online stage*.

The offline stage consists of three phases: model pruning (§4.1), model recovery (§4.2), and model profiling.

In the model pruning phase, by following the filter importance ranking provided by our TRR approach, filters in a given trained off-the-shelf deep learning model (i.e., *vanilla model*)

is iteratively pruned. During each iteration, less important filters are pruned, and the pruned model is retrained to compensate the accuracy degradation (if there is any) caused by filter pruning. The iteration ends when the pruned model could not meet the minimum accuracy requirement set by the user. This smallest pruned model is called *seed model*. As a result, a *filter pruning roadmap* is created where each footprint on the roadmap is a pruned model with its filter pruning record. This filter pruning roadmap including the seed model is passed to the model recovery phase.

In the model recovery phase, by following the filter pruning roadmap and the freeze-&-grow approach, the *multi-capacity model* of the vanilla model is iteratively generated. Model recovery uses the seed model as the starting point. During each iteration, the architecture and parameters of the model is first frozen. By following the filter pruning roadmap in the *reverse order* and adding the pruned filters back, a *descendant model* with a larger capacity is generated. Accuracy is regained by retraining the descendant model. By repeating the iteration, a new descendant model is grown upon the previous descendant model. As a result, the final descendant model has the capacities of all the previous descendant models and is thus named multi-capacity model.

In the model profiling phase, given the systems specs of a mobile vision system, a model profile is generated for the multi-capacity model including the accuracy, memory footprint, and processing latency of each of its capacities.

Finally, in the online stage, the NestDNN scheduler (§4.3) continuously monitors events including changes in available runtime resources, starting new applications, closing existing applications, and changes in application priority, and is triggered once such event is occurred. Once triggered, the scheduler examines the model profiles of all running vision applications, selects the appropriate descendant model for each application, and determines the amount

Terminology	Explanation
Vanilla Model	Off-the-shelf deep learning model (e.g., ResNet) trained on a given dataset (e.g., ImageNet).
Pruned Model	Intermediate result obtained during model pruning.
Seed Model	Smallest pruned model generated during model pruning that meets the minimum accuracy requirement set by the user. It is also the starting point of model recovery.
Descendant Model	A sub-model grown upon the Seed Model during model recovery. It has a unique resource-accuracy capacity.
Multi-Capacity Model	The lastly generated Descendant Model that has the capacities of all the previously generated descendant models but nested in a single model.

Table 3.1: Defined terminologies and their explanations.

of runtime resources to allocate to each selected descendant model to jointly maximize the accuracy and minimize the inference latency of those applications.

For clarification purpose, Table 3.1 summarizes the terminologies defined in this work and their brief explanations. In the next section, we describe NestDNN in detail.

3.4 Design of NestDNN

3.4.1 Filter based Model Pruning

3.4.1.1 Background on CNN Architecture

Before delving deep into filter pruning, it is important to understand the architecture of a convolutional neural network (CNN). In general, a CNN consists of four types of layers: convolutional layers, activation layers, pooling layers, and fully-connected layers. Due to the computational intensity of convolution operations, convolutional layers are the most computational intensive layers among the four types of layers. Specifically, each convolutional

layer is composed of a set of *3D filters*, which plays the role of "feature extractors". By convolving an image with these 3D filters, it generates a set of features organized in the form of *feature maps*, which are further sent to the following convolutional layers for further feature extraction.

3.4.1.2 Benefits of Filter Pruning

Figure 4.6 illustrates the details of filter pruning. Let $\Theta_{j-1} \in \mathbb{R}^{w_{j-1} \times h_{j-1} \times m_{j-1}}$ denote the input feature maps of the jth convolutional layer $conv_j$ of a CNN, where w_{j-1} and h_{j-1} are the width and height of each of the input feature maps; and m_{j-1} is the total number of the input feature maps. The convolutional layer $conv_j$ consists of m_j 3D filters with size $k \times k \times m_{j-1}$ ($k \times k$ is the 2D kernel). It applies these filters onto the input feature maps Θ_{j-1} to generate the output feature maps $\Theta_j \in \mathbb{R}^{w_j \times h_j \times m_j}$, where one 3D filter generates one output feature map. This process involves a total of $m_j k^2 m_{j-1} w_j h_j$ floating point operations (i.e., FLOP).

Since one 3D filter generates one output feature map, pruning one 3D filter in $conv_j$ (marked in green in $conv_j$) results in removing one output feature map in Θ_j (marked in green in Θ_j), which leads to $k^2 m_{j-1}$ parameter and $k^2 m_{j-1} w_j h_j$ FLOP reduction. Subsequently, m_{j+1} 2D kernels applied onto that removed output feature map in the convolutional layer $conv_{j+1}$ (marked in green in $conv_{j+1}$) are also removed. This leads to an additional $k^2 m_{j+1}$ parameter and $k^2 m_{j+1} w_{j+1} h_{j+1}$ FLOP reduction. Therefore, by pruning filters, both model size (i.e., model parameters) and computational cost (i.e., FLOP) are reduced [1].

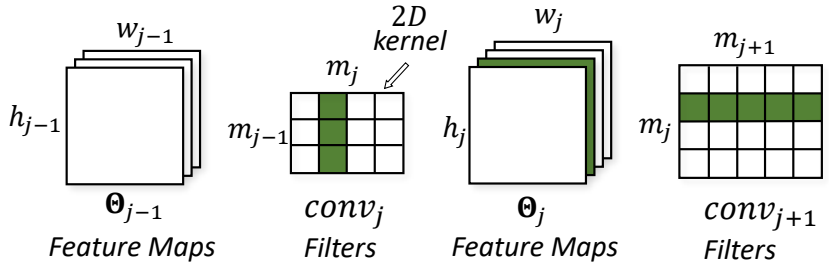

Figure 3.2: Illustration of filter pruning [1]. By pruning filters, both model size and computational cost are reduced.

3.4.1.3 Filter Importance Ranking

The key to filter pruning is identifying less important filters. By pruning those filters, the size and computational cost of a CNN model can be effectively reduced.

To this end, we propose a filter importance ranking approach named *Triplet Response Residual* (TRR) to measure the importance of filters and rank filters based on their relative importance. Our TRR approach is inspired by one *key intuition*: since a filter plays the role of "feature extractor", *a filter is important if it is able to extract feature maps that are useful to differentiate images belonging to different classes*. In other words, a filter is important if the feature maps it extracts from images belonging to the same class are more similar than the ones extracted from images belonging to different classes.

Let $\{anc, pos, neg\}$ denote a triplet that consists of an anchor image (anc), a positive image (pos), and a negative image (neg) where the anchor image and the positive image are from the same class, while the negative image is from a different class. By following the key intuition, TRR of filter i is defined as:

$$TRR_i = \sum(\|\mathcal{F}_i(anc) - \mathcal{F}_i(neg)\|_2^2 - \|\mathcal{F}_i(anc) - \mathcal{F}_i(pos)\|_2^2) \quad (3.1)$$

where $\mathcal{F}(\cdot)$ denotes the generated feature map. Essentially, TRR calculates the $\mathcal{L}2$ distances

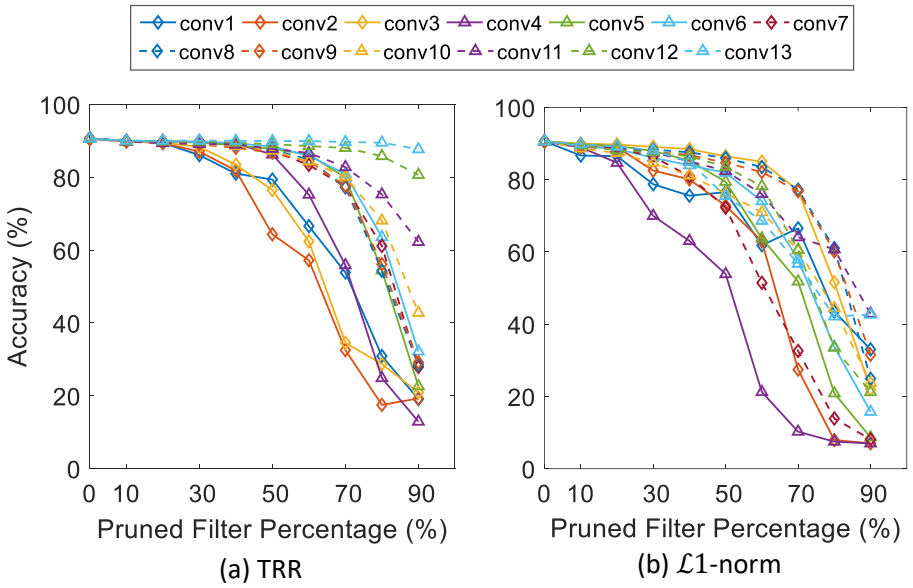

Figure 3.3: Filter importance profiling performance of (a) TRR and (b) $\mathcal{L}1$-norm on VGG-16 trained on CIFAR-10.

of feature maps between (*anc*, *neg*) and between (*anc*, *pos*), and measures the residual between the two distances. By summing up the residuals of all the triplets from the training dataset, the value of TRR of a particular filter reflects its capability of differentiating images belonging to different classes, acting as a measure of importance of the filter within the CNN model.

3.4.1.4 Performance of Filter Importance Ranking

Figure 3.3(a) illustrates the filter importance profiling performance of our TRR approach on VGG-16 [36] trained on the CIFAR-10 dataset [40]. The vanilla VGG-16 model contains 13 convolutional layers. Each of the 13 curves in the figure depicts the top-1 test accuracies when filters of one particular convolutional layer are pruned while the other convolutional layers remain unmodified. Each marker on the curve corresponds to the top-1 test accuracy when a particular percentage of filters is pruned. As an example, the topmost curve (blue dotted line with blue triangle markers) shows the accuracies are 89.75%, 89.72% and 87.40%

when 0% (i.e., vanilla model), 50% and 90% of the filters in the 13th convolutional layer $conv_{13}$ are pruned, respectively.

We have two key observations from the filter importance profiling result. First, we observe that our TRR approach is able to effectively identify redundant filters within each convolutional layer. In particular, the accuracy remains the same when 59.96% of the filters in $conv_{13}$ are pruned. This indicates that these pruned filters, identified by TRR, are redundant. By pruning these redundant filters, the vanilla VGG-16 model can be effectively compressed without any accuracy degradation. Second, we observe that our TRR approach is able to effectively identify convolutional layers that are more sensitive to filter pruning. This is reflected by the differences in accuracy drops when the same percentage of filters are pruned at different convolutional layers. This sensitivity difference across convolutional layers has been taken into account in the iterative filter pruning process.

To demonstrate the superiority of our TRR approach, we have compared it with the state-of-the-art filter pruning approach. The state-of-the-art filter pruning approach uses $\mathcal{L}1$-norm of a filter to measure its importance [1]. Figure 3.3(b) illustrates the filter importance profiling performance of $\mathcal{L}1$-norm on the same vanilla VGG-16 model trained on the CIFAR-10 dataset. By comparing Figure 3.3(a) to Figure 3.3(b), we observe that TRR achieves better accuracy than $\mathcal{L}1$-norm at almost every pruned filter percentage across all 13 curves. As a concrete example, TRR achieves an accuracy of 89.72% and 87.40% when 50% and 90% of the filters at $conv_{13}$ are pruned respectively, while $\mathcal{L}1$-norm only achieves an accuracy of 75.45% and 42.65% correspondingly. This result indicates that the filters pruned by TRR have much less impact on accuracy than the ones pruned by $\mathcal{L}1$-norm, demonstrating that TRR outperforms $\mathcal{L}1$-norm at identifying less important filters.

3.4.1.5 Filter Pruning Roadmap

By following the filter importance ranking provided by TRR, we iteratively prune the filters in a CNN model. During each iteration, less important filters across convolutional layers are pruned, and the pruned model is retrained to compensate the accuracy degradation (if there is any) caused by filter pruning. The iteration ends when the pruned model could not meet the minimum accuracy goal set by the user. As a result, a *filter pruning roadmap* is created where each footprint on the roadmap is a pruned model with its filter pruning record. The smallest pruned model on the roadmap is called *seed model*. This filter pruning roadmap is used to guide the model recovery process described below.

3.4.2 Freeze-&-Grow based Model Recovery

3.4.2.1 Motivation and Key Idea

The filter pruning process generates a series of pruned models, each of which acting as a model variant of the vanilla model with a unique resource-accuracy trade-off. However, due to the retraining step within each pruning iteration, these pruned models have different model parameters, and thus are independent from each other. Therefore, although these pruned models provide different resource-accuracy trade-offs, keeping all of them locally in resource-limited mobile systems is practically infeasible.

To address this problem, we propose to generate a single *multi-capacity model* that acts equivalently as the series of pruned models to provide various resource-accuracy trade-offs but has a model size that is much smaller than the accumulated model size of all the pruned models. This is achieved by an innovative *model freezing and filter growing* (i.e., *freeze-&-grow*) approach.

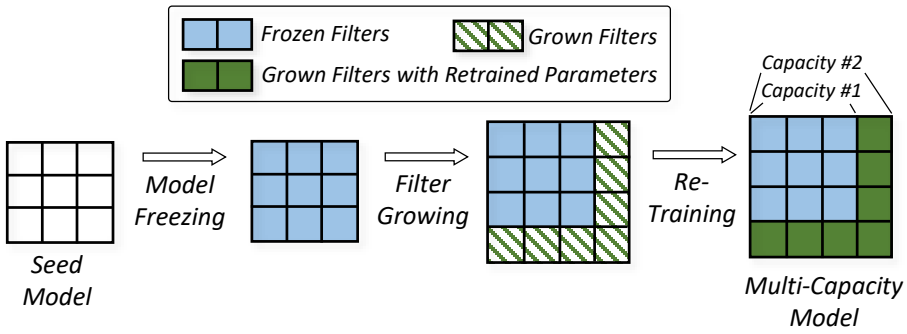

Figure 3.4: Illustration of model freezing and filter growing.

In the remainder of this section, we describe the details of the freeze-&-grow approach and how the multi-capacity model is iteratively generated.

3.4.2.2 Model Freezing and Filter Growing

The generation of the multi-capacity model starts from the seed model derived from the filter pruning process. By following the filter pruning roadmap and the freeze-&-grow approach, the multi-capacity model is iteratively created.

Figure 3.4 illustrates the details of model freezing and filter growing during the first iteration. For illustration purpose, only one convolutional layer is depicted. As shown, given the seed model, we first apply *model freezing* to freeze the parameters of all its filters (marked as blue squares). Next, since each footprint on the roadmap has its filter pruning record, we follow the filter pruning roadmap in the *reverse order* and apply *filter growing* to add the pruned filters back (marked as green stripe squares). With the added filters, the capacity of this *descendant model* is increased. Lastly, we retrain this descendant model to regain accuracy. It is important to note that during retraining, since the seed model is frozen, its parameters are not changed; only the parameters of the added filters are changed (marked as green solid squares to indicate the parameters are changed). As such, we have generated a single model that not only has the capacity of the seed model but also has the capacity of

the descendant model. Moreover, the seed model *shares* all its model parameters with the descendant model, making itself *nested* inside the descendant model without taking extra memory space.

By repeating the iteration, a new descendant model is grown upon the previous one. As such, the final descendant model has the capacities of all the previous descendant models and is thus named *multi-capacity model*.

3.4.2.3 Superiority of Multi-Capacity Model

The generated multi-capacity model has the following three key advantages.

One Compact Model with Multiple Capabilities. The generated multi-capacity model is able to provide multiple capacities nested in a single model. This eliminates the need of installing potentially a large number of independent model variants with different capacities. Moreover, by sharing parameters among descendant models, the multi-capacity model is able to save a large amount of memory space to significantly reduce its memory footprint.

Optimized Resource-Accuracy Trade-offs. Each capacity provided by the multi-capacity model has a unique optimized resource-accuracy trade-off. Our TRR approach is able to provide state-of-the-art performance at identifying and pruning less important filters. As a result, the multi-capacity model delivers state-of-the-art inference accuracy under a given resource budget.

Efficient Model Switching. Because of parameter sharing, the multi-capacity model is able to switch models with little overhead. Switching independent deep learning models causes significant overhead. This is because it requires to page in and page out the *entire* deep learning models. Multi-capacity model alleviates this problem in an elegant manner by only requiring to page in and page out a very small portion of deep learning models.

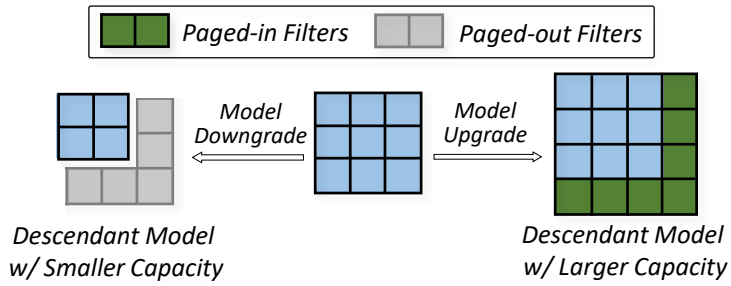

Figure 3.5: Illustration of model switching (model upgrade vs. model downgrade) of multi-capacity model.

Figure 3.5 illustrates the details of model switching of multi-capacity model. For illustration purpose, only one convolutional layer is depicted. As shown, since each descendant model is grown upon its previous descendant models, when the multi-capacity model is switching to a descendant model with larger capability (i.e., *model upgrade*), it incurs *zero page-out overhead*, and only needs to page in the extra filters included in the descendant model with larger capability (marked as green squares). When the multi-capacity model is switching to a descendant model with smaller capability (i.e., *model downgrade*), it incurs *zero page-in overhead*, and only needs to page out the filters that the descendant model with smaller capability does not have (marked as gray squares). As a result, the multi-capacity model significantly reduces the overhead of model page in and page out, making model switching extremely efficient.

3.4.3 Resource-Aware Scheduler

3.4.3.1 Motivation and Key Idea

The creation of the multi-capacity model enables NestDNN to jointly maximize the performance of vision applications that are concurrently running on a mobile vision system. This possibility comes from two *key insights*. First, while a certain amount of runtime resources can be traded for an accuracy gain in some application, the same amount of runtime resources

may be traded for a *larger* accuracy gain in some other application. Second, for applications that do not need real-time response and thus can tolerate a relatively large processing latency, we can *reallocate* some runtime resources from those latency-tolerant applications to other applications that need more runtime resources to meet their real-time goals. NestDNN exploits these two key insights by encoding the inference accuracy and processing latency into a *cost function* for each vision application, which serves as the foundation for resource-aware scheduling.

3.4.3.2 Cost Function

Let V denote the set of vision applications that are concurrently running on a mobile vision system, and let $A_{min}(v)$ and $L_{max}(v)$ denote the minimum inference accuracy and the maximum processing latency goals set by the user for application $v \in V$. Additionally, let M_v denote the multi-capacity model generated for application v, and let m_v denote a descendant model $m_v \in M_v$. The cost function of the descendant model m_v for application v is defined as follows:

$$\begin{aligned} C(m_v, u_v, v) = \max(0, A_{min}(v) - A(m_v)) + \\ \alpha \cdot \max(0, \frac{L(m_v)}{u_v} - L_{max}(v)) \end{aligned} \tag{3.2}$$

where $A(m_v)$ is the inference accuracy of m_v, $u_v \in (0, 1]$ is the computing resource percentage allocated to v, and $L(m_v)$ is the processing latency of m_v when 100% computing resources are allocated to v. The values of $A(m_v)$ and $L(m_v)$ are obtained via profiling at the multi-capacity model development stage.

Essentially, the first term in the cost function represents the penalty for selecting a descendant model m_v that has an inference accuracy lower than the minimum accuracy

goal. The second term in the cost function represents the penalty for selecting a descendant model m_v that has a processing latency higher than the maximum processing latency goal. $\alpha \in [0, 1]$ is a knob set by the user to determine the latency-accuracy trade-off preference. A large α weights more on the penalty for latency while a small α favors higher accuracy.

3.4.3.3 Scheduling Schemes

Given the cost function of each descendant model of each concurrently running application, the resource-aware scheduler incorporates two widely used scheduling schemes to jointly maximize the performance of concurrent vision applications for two different optimization objectives.

MinTotalCost. The MinTotalCost (i.e., minimize the total cost) scheduling scheme aims to minimize the total cost of all concurrent applications. This optimization problem can be formulated as follows:

$$\min_{u_v, m_v \in M_v} \sum_{v \in V} C(m_v, u_v, v) \tag{3.3}$$

$$s.t. \quad \sum_{v \in V} S(m_v) \leq S_{max}, \quad \sum_{v \in V} u_v \leq 1$$

where $S(m_v)$ denotes the runtime memory footprint of the descendant model m_v. The total memory footprint of all the concurrent applications cannot exceed the maximum memory space of the mobile vision system denoted as S_{max}.

Under the MinTotalCost scheduling scheme, the resource-aware scheduler favors applications with lower costs and thus is optimized to allocate more runtime resources to them.

MinMaxCost. The MinMaxCost (i.e., minimize the maximum cost) scheduling scheme

aims to minimize the cost of the application that has the highest cost. This optimization problem can be formulated as follows:

$$\min_{u_v, m_v \in M_v} k \tag{3.4}$$

$$s.t. \quad \forall v : C(m_v, u_v, v) \leq k,$$

$$\sum_{v \in V} S(m_v) \leq S_{max}, \quad \sum_{v \in V} u_v \leq 1$$

where the cost of any of the concurrently running applications must be smaller than k where k is minimized.

Under the MinMaxCost scheduling scheme, the resource-aware scheduler is optimized to fairly allocate runtime resources to all the concurrent applications to balance their performance.

3.4.3.4 Cached Greedy Heuristic Approximation

Solving the nonlinear optimization problems involved in MinTotalCost and MinMaxCost scheduling schemes is computationally hard. To enable real-time online scheduling in mobile systems, we utilize a greedy heuristic inspired by [29] to obtain approximate solutions.

Specifically, we define a minimum indivisible runtime resource unit Δu (e.g., 1% of the total computing resources in a mobile vision system) and start allocating the computing resources from scratch. For MinTotalCost, we allocate Δu to the descendent model m_v of application v such that $C(m_v, \Delta u, v)$ has the smallest cost increase among other concurrent applications. For MinMaxCost, we select application v with the highest cost $C(m_v, u_v, v)$,

Type	Dataset	DNN Model	Mobile Vision Application
Generic Category	CIFAR-10	VGG-16	VC
	ImageNet-50	ResNet-50	RI-50
	ImageNet-100	ResNet-50	RI-100
Class Specific	GTSRB	VGG-16	VS
	Adience-Gender	VGG-16	VG
	Places-32	ResNet-50	RP

Table 3.2: Summary of datasets, DNN models, and mobile vision applications used in this work.

and allocate Δu to v and choose the optimal descendent model $m_v = \arg\min_{m_v} C(m_v, u_v, v)$ for v. For both MinTotalCost and MinMaxCost, the runtime resources are iteratively allocated until exhausted.

The runtime of executing the greedy heuristic can be further shortened via the caching technique. This is particularly attractive to mobile systems with very limited resources. Specifically, when allocating the computing resources, instead of starting from scratch, we start from the point where a certain amount of computing resources has already been allocated. For example, we can cache the unfinished running scheme where 70% of the computing resources have been allocated during optimization. In the next optimization iteration, we directly start from the unfinished running scheme and allocate the remaining 30% computing resources, thus saving 70% of the optimization time. To prevent from falling into a local minimum over time, a complete execution of the greedy heuristic is performed periodically to enforce the cached solution to be close to the optimal one.

3.5 Evaluation

3.5.1 Datasets, DNNs and Applications

3.5.1.1 Datasets

To evaluate the generalization capability of NestDNN on different vision tasks, we select two types of tasks that are among the most important tasks for mobile vision systems.

Generic-Category Object Recognition. This type of vision tasks aims to recognize the generic category of an object (e.g., a road sign, a person, or an indoor place). Without loss of generality, we select 3 commonly used computer vision datasets, each containing a small, a medium, and a large number of object categories respectively, representing an easy, a moderate, and a difficult vision task correspondingly.

- **CIFAR-10** [40]. This dataset contains 50K training images and 10K testing images belonging to 10 generic categories of objects.
- **ImageNet-50** [41]. This dataset is a subset of the ILSVRC ImageNet. It contains 63K training images and 2K testing images belonging to top 50 most popular object categories based on the popularity ranking provided by the official ImageNet website.
- **ImageNet-100** [41]. Similar to ImageNet-50, this dataset is a subset of the ILSVRC ImageNet. It contains 121K training images and 5K testing images belonging to top 100 most popular object categories based on the popularity ranking provided by the official ImageNet website.

Class-Specific Object Recognition. This type of vision tasks aims to recognize the specific class of an object within a generic category (e.g., a stop sign, a female person, or a kitchen). Without loss of generality, we select 3 object categories: 1) road signs, 2) people,

and 3) places, which are commonly seen in mobile settings.

- **GTSRB** [42]. This dataset contains over 50K images belonging to 43 classes of road signs such as speed limit signs and stop sign.
- **Adience-Gender** [43]. This dataset contains over 14K images of human faces of two genders.
- **Places-32** [44]. This dataset is a subset of the Places365-Standard dataset. Places365-Standard contains 1.8 million images of 365 scene classes belonging to 16 higher-level categories. We select two representative scene classes (e.g., parking lot and kitchen) from each of the 16 higher-level categories and obtain a 32-class dataset that includes over 158K images.

3.5.1.2 DNN Models

To evaluate the generalization capability of NestDNN on different DNN models, we select two representative DNN models: 1) VGG-16 and 2) ResNet-50. VGG-16 [36] is considered as one of the most straightforward DNN models to implement, and thus gains considerable popularity in both academia and industry. ResNet-50 [35] is considered as one of the top-performing DNN models in computer vision due to its superior recognition accuracy.

3.5.1.3 Mobile Vision Applications

Without loss of generality, we randomly assign CIFAR-10, GTSRB and Adience-Gender to VGG-16; and assign ImageNet-50, ImageNet-100 and Places-32 to ResNet-50 to create six mobile vision applications labeled as VC (i.e., VGG-16 trained on the CIFAR-10 dataset), RI-50, RI-100, VS, VG, and RP, respectively. We train and test all the vanilla DNN models and all the descendant models generated by NestDNN by strictly following the protocol

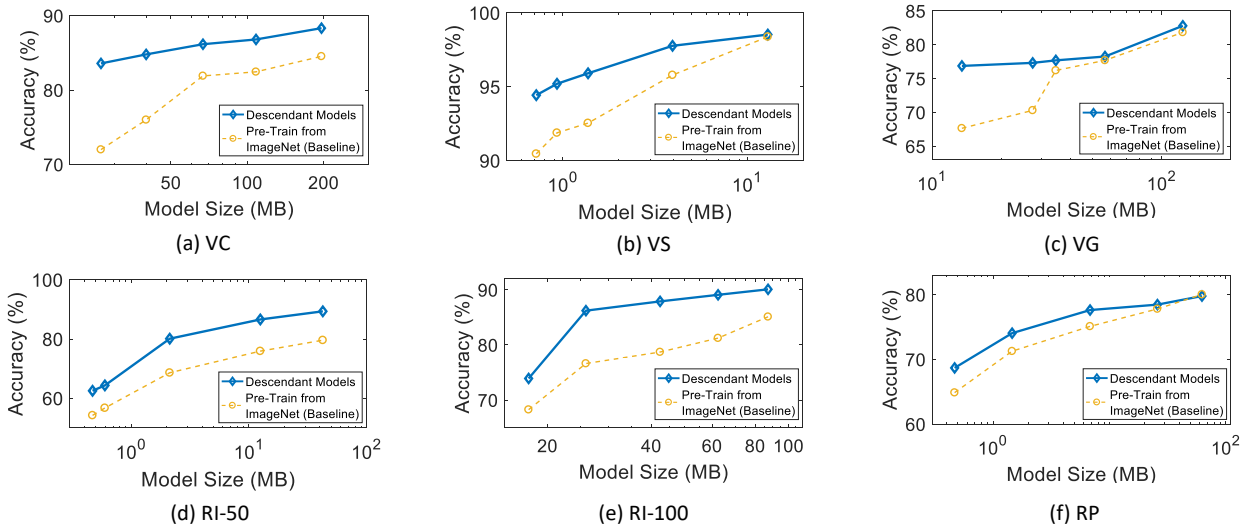

Figure 3.6: Top-1 test accuracy vs. model size comparison between descendent models and baseline models.

provided by each of the six datasets described above.

The datasets, DNN models, and mobile vision applications are summarized in Table 3.2.

3.5.2 Performance of Multi-Capacity Model

In this section, we evaluate the performance of the generated multi-capacity model with the goal to demonstrate its superiority listed in §4.2.3.

3.5.2.1 Experimental Setup

Selection of Descendant Models. Without loss of generality, for each mobile vision application, we generate a multi-capacity model that contains five descendant models. These descendant models are designed to have diverse resource-accuracy trade-offs. We select these descendant models with the purpose to demonstrate that our multi-capacity model enables applications to run even when available resources are very limited. It should be noted that a multi-capacity model is neither limited to one particular set of resource-accuracy trade-offs nor limited to one particular number of descendant models. NestDNN provides the flexibility

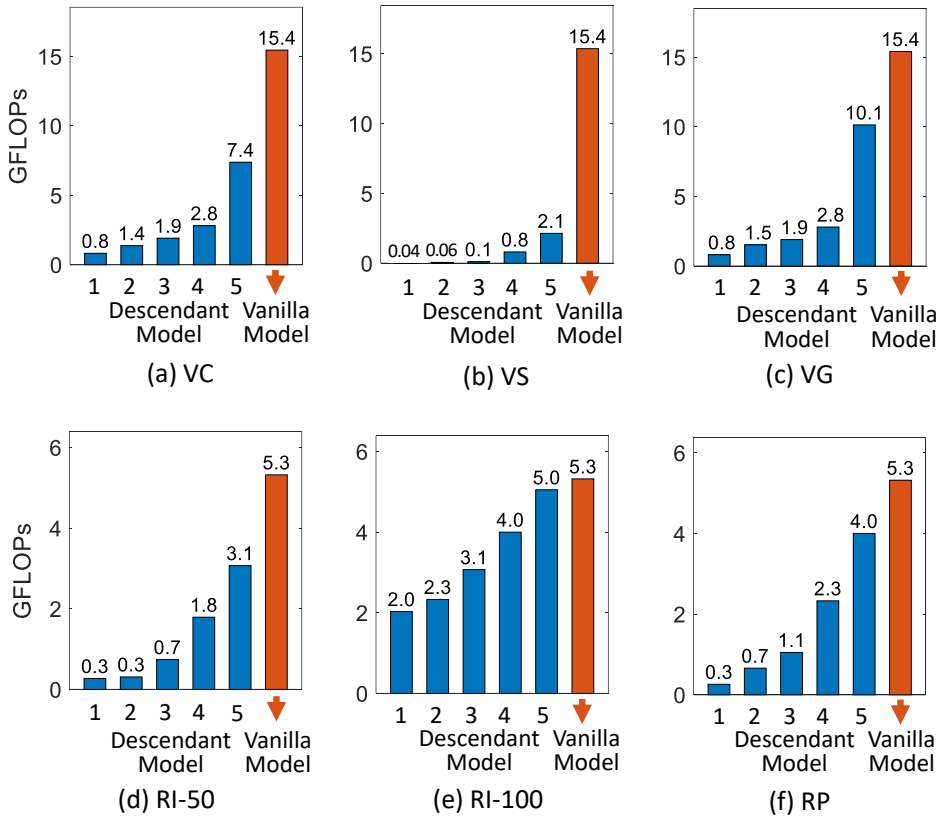

Figure 3.7: Computational cost comparison between descendant models and vanilla models.

to design a multi-capacity model based on users' preferences.

Baseline. To make a fair comparison, we use the same architecture of descendant models for baseline models such that their model sizes and computational costs are identical. In addition, we pre-trained baseline models on ImageNet dataset and then fine-tuned them on each of the six datasets. Pre-training is an effective way to boost accuracy, and thus is adopted as a routine in machine learning community [45, 46]. We also trained baseline models without pre-training, and observed that baseline models with pre-training consistently outperform those without pre-training. Therefore, we only report accuracies of baseline models with pre-training.

3.5.2.2 Optimized Resource-Accuracy Trade-offs

Figure 3.6 illustrates the comparison between descendent models and baseline models across six mobile vision applications. For each application, we show the top-1 test accuracies of both descendant models and baseline models as a function of model size. For better illustration purpose, the horizontal axis is plotted using the logarithmic scale.

We have two key observations from the result. First, we observe that descendant models consistently achieve higher accuracies than baseline models at every model size across all the six applications. On average, descendant models achieve 4.98% higher accuracy than baseline models. This indicates that our descendant model at each capacity is able to deliver state-of-the-art inference accuracy under a given memory budget. Second, we observe that smaller descendant models outperform baseline models more than larger descendant models. On average, the two smallest descendant models achieve 6.68% higher accuracy while the two largest descendant models achieve 3.72% higher accuracy compared to their corresponding baseline models. This is because our TRR approach is able to preserve important filters while pruning less important ones. Despite having a small capacity, a small descendant model benefits from these important filters while the corresponding baseline model does not.

Figure 3.7 shows the computational costs of five descendant models and the corresponding vanilla models of the six applications in GFLOPs (i.e., GigaFLOPs). As shown, all descendant models have less GFLOPs than the corresponding vanilla models. This result indicates that our filter pruning approach is able to effectively reduce the computational costs across six applications, demonstrating the generalization of our filter pruning approach on different deep learning models trained on different datasets.

Application	Multi-Capacity Model Size (MB)	Accumulated Model Size (MB)	Reduced Memory Footprint (MB)
VC	196.0	437.5	241.5
VS	12.9	19.8	6.9
VG	123.8	256.0	132.2
RI-50	42.4	58.1	15.7
RI-100	87.1	243.5	156.4
RP	62.4	97.1	34.7
All Included	524.6	1112.0	587.4

Table 3.3: Benefit of multi-capacity model on memory footprint reduction.

3.5.2.3 Reduction on Memory Footprint

Another key feature of multi-capacity model is sharing parameters among its descendant models. To quantify the benefit of parameter sharing on reducing memory footprint, we compare the model size of multi-capacity model with the accumulated model size of the five descendant models as if they were independent. This mimics traditional model variants that are used in existing mobile deep learning systems.

Table 3.3 lists the comparison results across the six mobile vision applications. Obviously, the model size of the multi-capacity model is smaller than the corresponding accumulated model size for each application. Moreover, deep learning model with larger model size benefits more from parameter sharing. For example, VC has the largest model size across the six applications. With parameter sharing, it achieves a reduced memory footprint of 241.5 MB. Finally, if we consider running all the six applications concurrently, the multi-capacity model achieves a reduced memory footprint of 587.4 MB, demonstrating the enormous benefit of multi-capacity model on memory footprint reduction.

3.5.2.4 Reduction on Model Switching Overhead

Another benefit of parameter sharing is reducing the overhead of model switching when the set of concurrent applications changes. To quantify this benefit, we consider all the possible model switching cases among all the five descendant models of each multi-capacity model, and calculate the average page-in and page-out overhead for model upgrade and model downgrade, respectively. We compare it with the case where descendant models are treated as if they were independent, which again mimics traditional model variants.

Table 3.4 lists the comparison results of all six mobile vision applications for model upgrade in terms of memory usage. As expected, the average page-in and page-out memory usage of independent models during model switching is larger than multi-capacity models for every application. This is because during model switching, independent models need to page in and page out the *entire* models while multi-capacity models only need to page in a very small portion of the models. It should be noted that the page-out overhead of multi-capacity model during model upgrade is zero. This is because the descendant model with smaller capability is part of the descendant model with larger capability, and thus it does not need to be paged out.

Table 3.5 lists the comparison results of all six applications for model downgrade. Similar results are observed. The only difference is that during model downgrade, the page-in overhead of multi-capacity model is zero.

Besides memory usage, we also quantify the benefit on reducing the overhead of model switching in terms of energy consumption. Specifically, we measured energy consumed by randomly switching models for 250, 500, 750, and 1,000 times using descendant models and independent models, respectively.

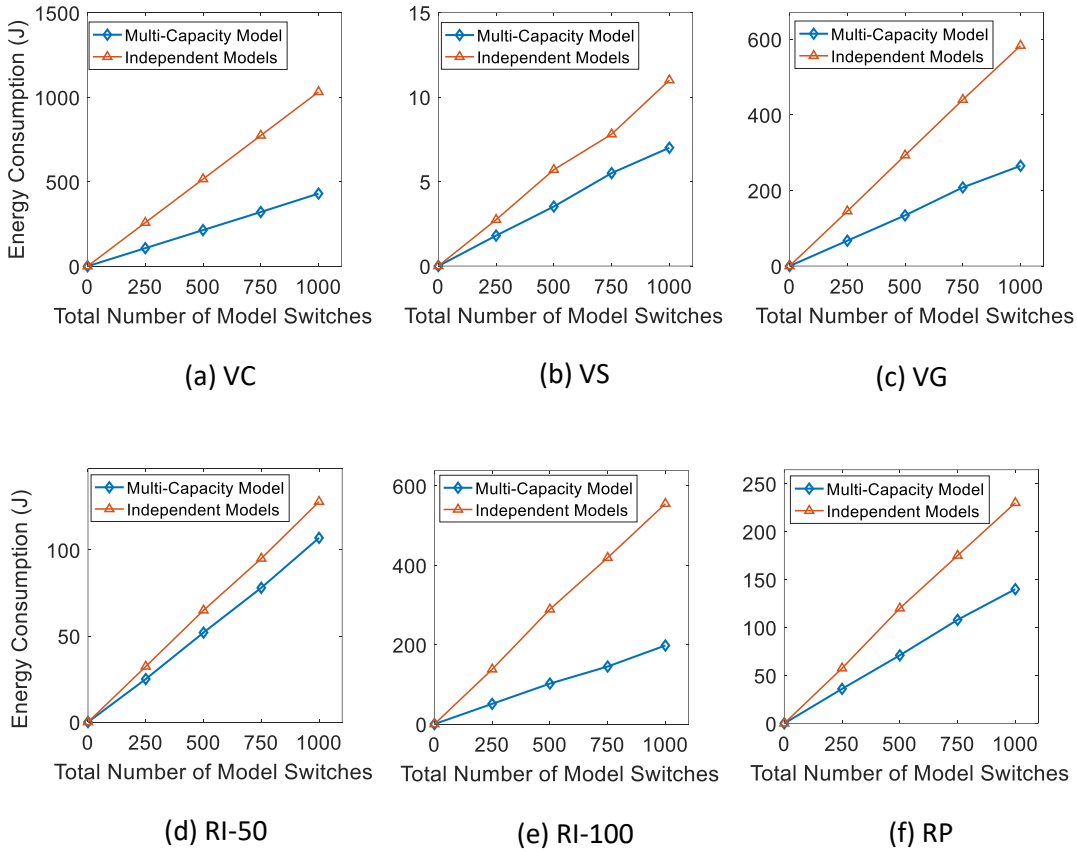

Figure 3.8: Model switching energy consumption comparison between multi-capacity models and independent models. The energy consumption is measured on a Samsung Galaxy S8 smartphone.

Figure 3.8 shows the comparison results across all six mobile vision applications. As expected, energy consumed by switching multi-capacity model is lower than switching independent models for every application. This benefit becomes more prominent when the model size is large. For example, the size of the largest descendant model of VC and VS is 196.0 MB and 12.9 MB, respectively. The corresponding energy consumption reduction for every 1,000 model switches is 602.1 J and 4.0 J, respectively.

Taken together, the generated multi-capacity model is able to significantly reduce the overhead of model switching in terms of both memory usage and energy consumption. The benefit becomes more prominent when model switching frequency increases. This is particularly important for memory and battery constrained mobile systems.

Application	Multi-Capacity Model Upgrade Overhead (MB)		Independent Models Upgrade Overhead (MB)	
	Page-In	Page-Out	Page-In	Page-Out
VC	81.4	0	128.2	46.8
VS	1.3	0	1.7	0.3
VG	50.0	0	76.2	26.2
RI-50	19.2	0	21.2	2.0
RI-100	38.3	0	67.9	29.5
RP	26.4	0	34.9	4.6

Table 3.4: Benefit of multi-capacity model on model switching (model upgrade) in terms of memory usage.

Application	Multi-Capacity Model Downgrade Overhead (MB)		Independent Models Downgrade Overhead (MB)	
	Page-In	Page-Out	Page-In	Page-Out
VC	0	81.4	46.8	128.2
VS	0	1.3	0.3	1.7
VG	0	50.0	26.2	76.2
RI-50	0	19.2	2.0	21.2
RI-100	0	38.3	29.5	67.9
RP	0	26.4	4.6	34.9

Table 3.5: Benefit of multi-capacity model on model switching (model downgrade) in terms of memory usage.

3.5.3 Performance of Resource-Aware Scheduler

3.5.3.1 Experimental Setup

Deployment Platforms. We implemented NestDNN and the six mobile vision applications on three smartphones: Samsung Galaxy S8, Samsung Galaxy S7, and LG Nexus 5, all running Android OS 7.0. We used the Monsoon power monitor [47] to measure the power consumption. We have achieved consistent experimental results across all three smartphones. We only report the best results obtained from Galaxy S8.

Baseline. We use the model located at the "knee" of every yellow curve in Figure 3.6 as our baseline for each mobile vision application. This is the model that offers the *best* resource-accuracy trade-off among all the model variants.

Benchmark Design. We have designed a benchmark that simulates runtime application queries in different scenarios. Specifically, our benchmark creates a new application or kills a running application with certain probabilities at every second. The number of concurrently running applications is from 2 to 6. At the same time, the maximum available memory to run concurrent applications is set to 400 MB. Each simulation generated by our benchmark lasts for 60 seconds. We repeat the simulation 100 times and report the average runtime performance.

Figure 3.9 illustrates the profile of all accumulated simulations generated by our benchmark. Figure 3.9(a) shows the time distribution of different numbers of concurrent applications. As shown, the percentage of time when two, three, four, five and six applications running concurrently is 9.8%, 12.7%, 18.6%, 24.5% and 34.3%, respectively. It shows that our benchmark has covered all the numbers of concurrent applications. Figure 3.9(b) shows the running time distribution of each individual application. As shown, the running time of each application is evenly distributed, indicating our benchmark ensures a fair time share among all applications.

Evaluation Metrics. We use the following two metrics to evaluate the scheduling performance.

- **Accuracy Gain**. Since the absolute Top-1 accuracies achieved by the six vision applications are not in the same range, we use accuracy gain over the baseline within each application as a more meaningful metric.

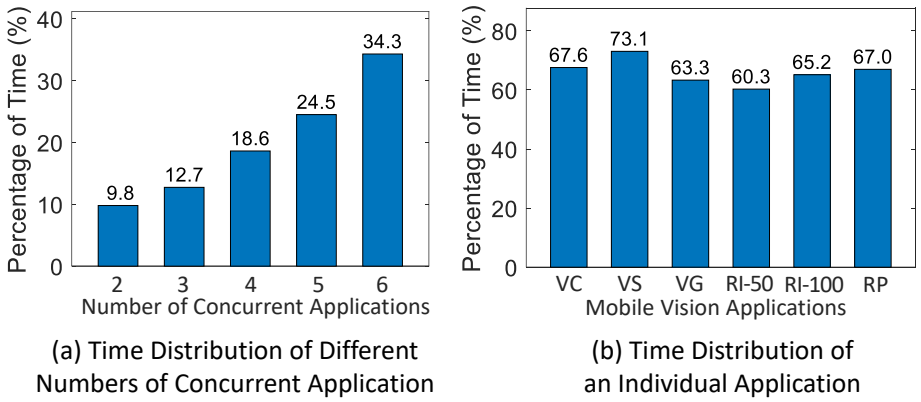

Figure 3.9: Profile of all accumulated simulations generated by our designed benchmark.

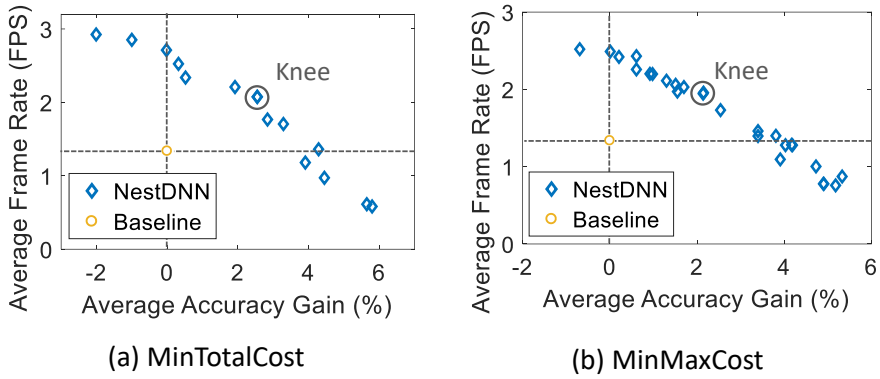

Figure 3.10: Runtime performance comparison between baseline and NestDNN under scheduling scheme: (a) MinTotalCost; (b) MinMaxCost.

- **Frame Rate**. We use frame rate as the metric to measure the real-time performance of a vision application. It is a standard metric for mobile vision systems. Frame rate is inversely proportional to inference latency. The lower the inference latency is, the higher the frame rate is.

3.5.3.2 Improvement on Accuracy and Frame Rate

Figure 3.10(a) shows the comparison between the baseline and NestDNN under the MinTotalCost scheduling scheme. Each blue diamond marker represents the runtime performance obtained by scheduling with a particular α in the cost function defined in §4.3.2. The yellow circle represents the runtime performance of the baseline.

We have two key observations from our comparison result. First, by adjusting the value of α, NestDNN is able to provide a variety of trade-offs between accuracy and frame rate. In contrast, the baseline is fixed and offers no trade-off between accuracy and frame rate. Second, the vertical dotted line and the horizontal dotted line altogether partition the figure into four quadrants. The upper right quadrant represents a region that has both higher accuracy gain and higher frame rate compared to the baseline. As shown, there are many blue diamond markers locating at this upper right quadrant. At each of those blue diamond markers, NestDNN is able to achieve both higher accuracy gain and higher frame rate compared to the baseline. In particular, we select 3 of those blue diamond markers to demonstrate the runtime performance improvement achieved by NestDNN. Specifically, when NestDNN has the same accuracy gain as the baseline, NestDNN achieves 2.0× frame rate in the unit of frame per second (FPS) compared to the baseline. When NestDNN has the same frame rate as the baseline, NestDNN achieves 4.1% accuracy gain compared to the baseline. Finally, we select the "knee" of the blue diamond curve, which offers the best accuracy-frame rate trade-off among all the α. At the "knee", NestDNN achieves 1.5× frame rate and 2.6% accuracy gain compared to the baseline.

Figure 3.10(b) shows the comparison between the baseline and NestDNN under the MinMaxCost scheduling scheme. When NestDNN has the same accuracy gain as the baseline, NestDNN achieves 1.9× frame rate compared to the baseline. When NestDNN has the same frame rate as the baseline, NestDNN achieves 4.2% accuracy gain compared to the baseline. At the "knee", NestDNN achieves 1.5× frame rate and 2.1% accuracy gain compared to the baseline.

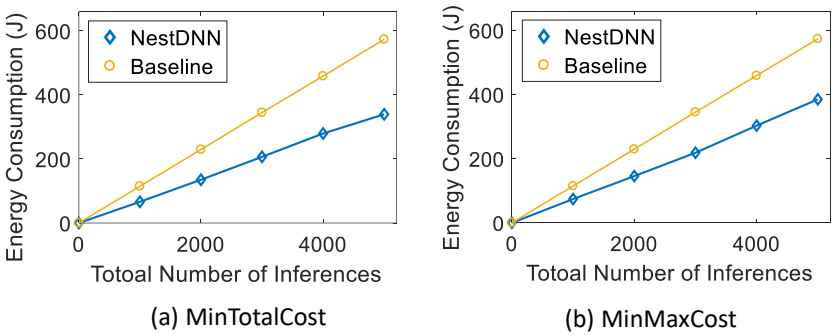

Figure 3.11: Energy consumption comparison between baseline and NestDNN under scheduling scheme: (a) MinTotalCost; (b) MinMaxCost.

3.5.3.3 Reduction on Energy Consumption

Finally, we evaluate the inference energy consumption during scheduling. In this experiment, we evaluate inference energy consumption of NestDNN at the "knee".

Figure 3.11(a) shows the comparison between the baseline and NestDNN under the MinTotalCost scheduling scheme. Across different numbers of inferences, NestDNN achieves an average 1.7× energy consumption reduction compared to the baseline. Similarly, Figure 3.11(b) shows the comparison between the baseline and NestDNN under the MinMaxCost scheduling scheme. NestDNN achieves an average 1.5× energy consumption reduction compared to the baseline.

3.6 Conclusion of NestDNN

In this chapter, we presented the design, implementation and evaluation of NestDNN, a framework that enables resource-aware multi-tenant on-device deep learning for mobile vision systems. It contributes novel techniques that address the unique challenges of mobile vision systems. We evaluated NestDNN using six mobile vision applications that target some of the most important vision tasks for mobile vision systems. Our results show that compared to the non-resource-aware counterpart, NestDNN achieves as much as 4.2% increase on

accuracy, 2.0× increase on frame rate and 1.7× reduction on energy consumption when running multiple mobile vision applications concurrently.

Chapter 4

Content-Adaptive On-Device Deep Learning

4.1 Introduction of FlexDNN

Mobile vision systems such as mobile phones, drones, and augmented reality headsets are ubiquitous today. Driven by recent breakthrough in Deep Neural Networks (DNNs) [2] and the emergence of AI chipsets, state-of-the-art mobile vision systems start to use DNN-based processing pipelines for on-device video stream analytics, which acts as the enabler of a wide range of continuous mobile vision applications.

On-device video stream analytics requires processing streaming video frames at high throughput and returning the processing results with low latency. Unfortunately, DNNs are known to be computationally expensive [3], and high computational consumption directly translates to high processing latency and high energy consumption. Given mobile systems are constrained by limited compute resources and battery capacities, reducing computational consumption of DNN-based pipelines is crucial to high-throughput, low-latency, and low-energy on-device video stream analytics.

To reduce computational consumption, most existing work pursues model compression techniques [9, 37, 48]. However, model compression yields an one-size-fits-all network that requires the same set of feature maps to be extracted for all video frames agnostic to the

content of each frame.

In fact, the computation consumed by a DNN-based processing pipeline is heavily dependent on the content of the video frames [20]. For video frames with contents that are easy to recognize, a small low-capacity DNN model is sufficient while a large high-capacity DNN model that consumes more computation is overkill; on the other hand, for video frames with contents that are hard to recognize, it is necessary to employ large high-capacity DNN models in the processing pipeline to ensure the contents to be correctly recognized. This is very similar to how human vision system works where a glimpse is sufficient to recognize simple scenes and objects in ordinary poses, whereas more attention and efforts are needed to understand complex scenes and objects that are complicated or partially occluded [49].

Based on this key insight, content-adaptive video stream analytics systems such as Chameleon [18] have recently emerged. Leveraging the dynamics of video contents, these systems effectively reduce the computational consumption of DNN-based processing pipelines by dynamically changing the DNN models in the pipeline to adapt to the difficulty levels of the video frames. However, this dynamic configuration approach is a mismatch to resource-constrained mobile systems. This is because it requires all the model variants with various capacities to be installed in the mobile system, which results in large memory footprint. More importantly, if a large number of model variants is incorporated and the content dynamics is substantial, the overhead of searching for the optimal model variant and switching models at runtime can be prohibitively expensive, which considerably dwarfs the benefit brought by adaptation.

The limitation of Chameleon is rooted in the constraint where it requires having multiple model variants with various capacities to adapt to various difficulty levels of video frames. To address this limitation, state-of-the-art such as BranchyNet [19] and MSDNet [20] introduces

the idea of constructing a single model by adding "early exits" at layers of a regular DNN model to make early prediction. With such early prediction mechanism, easy frames do not need to go through all the layers and their computational consumption is thus reduced. While the concept of early prediction is promising, these pioneer works are constrained by the following limitations:

- These pioneer works are coarse-grained approach where early exits are inserted at the outputs of convolutional layers of a DNN model. Each convolutional layer is composed of a large number of convolutional filters, and these filters dominate the computational consumption of the DNN model. However, not all the filters within each convolutional layer are needed to early exit easy frames. As a consequence, computation consumed by those unnecessary filters is wasted by the coarse-grained approach due to its constraint on making early predictions at the granularity of layers.
- The early exits themselves also consume computation. Computation consumed by frames that fail to exit at the early exits is wasted and becomes the overheads incurred by this approach. Unfortunately, the early exit architecture of prior works is designed based on heuristics without focusing on the trade-off between early exit rates and the incurred overheads. Without carefully accounting for the trade-off, the incurred overheads could diminish the benefit of early exits.
- Lastly, the number and locations of the inserted early exits in prior works are also determined based on heuristics. While effective in comparison to models without early exits, considering the exponential combinations of number and locations of early exits, even for developers with extensive domain expertise, without considerable efforts on trial and error, it would be extremely challenging to derive an early exit insertion plan that can

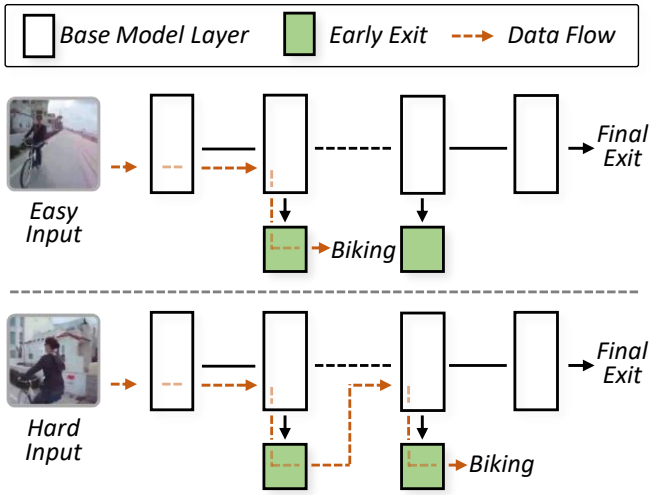

Figure 4.1: A conceptual overview of FlexDNN. FlexDNN is built on top of a base model with the augmentation of one or more early exits inserted throughout the base model. For an easy input, it exits at the early exit inserted at an earlier location since the extracted features are good enough to classify the content in the easy input. As such, the easy input avoids further computational cost incurred onward. For a hard input, it proceeds deeper until the extracted features are good enough to exit the hard input.

fully leverage the computational consumption reduction benefit brought by early exits. Moreover, since early exits incur overheads, the number and locations of the inserted early exits play a critical role in determining the amount of computation that can be saved, making the derivation of the early exit insertion plan even more challenging.

In this paper, we present FlexDNN, a content-adaptive framework for computation-efficient DNN-based mobile video stream analytics. Figure 4.1 provides a conceptual overview of FlexDNN. FlexDNN effectively addresses these limitations with three novel contributions:

- **Fine-Grained Early Prediction.** Instead of making early predictions at the granularity of layers, FlexDNN breaks the natural boundaries at layers and adopts a fine-grained approach to make early predictions at the granularity of filters. For this, we have designed a novel filter importance ranking scheme based on the concept of collective importance to identify redundant filters within each convolutional layer. With such fine-grained ap-

proach, FlexDNN is able to provide much more flexibility on making early predictions. Moreover, it cuts off unnecessary computation wasted by redundant filters within each layer that coarse-grained approaches could not achieve.

- **Optimal Architecture.** FlexDNN is able to generate the optimal architecture for each early exit and the optimal early exit insertion plan. For this, we have designed an architecture search-based scheme that is able to find the optimal architecture that balances the trade-off between early exit rate and computational overhead for each early exit. We have also formulated the determination of number of locations of early exits as an optimization problem to derive the optimal early exit insertion plan that maximizes the benefit brought by content adaptation. As such, FlexDNN allows developers with limited domain expertise to build DNN-based computation-efficient continuous mobile vision applications with minimum efforts.
- **Runtime Adaptation.** At runtime, mobile vision systems may experience workload dynamics under different contexts as well as system resource dynamics due to multi-tenancy. As its final contribution, the design of FlexDNN provides a natural mechanism to adapt to both workload and system resource dynamics at runtime.

We implement FlexDNN in TensorFlow [50] and use it to build three representative continuous mobile vision applications: Activity Recognition, Scene Understanding, and Traffic Surveillance based on VGGNet and MobileNets. We use real-world datasets with videos taken by mobile devices to profile these applications and evaluate their runtime performance on both mobile CPU and mobile GPU platforms.

While MobileNets (content-agnostic models) are known for their computation-efficient designs, our evaluation shows that FlexDNN significantly outperforms MobileNets due to

its content-adaptive capability: achieving up to 31.0% reduction in computation consumption and up to 29.6% reduction in energy consumption while having the same accuracy. We also compare FlexDNN to BranchyNet, a recent state-of-the-art content-adaptive model that is based on coarse-grained early exit design. Our results show that besides the benefit of automatically generating the optimal early exit architecture and the optimal early exit insertion plan, FlexDNN significantly outperforms BranchyNet due to its fine-grained design and architecture superiority, achieving up to 41.2% reduction in computation consumption and up to 38.4% reduction in energy consumption while having the same accuracy. Finally, our evaluation shows that under workload and system resource dynamics at runtime, FlexDNN is able to adapt to the dynamics and achieve much higher accuracies than BranchyNet.

4.2 Background & Motivation

4.2.1 Dynamics of Mobile Video Contents & Benefit of Leveraging the Dynamics

Due to mobility of cameras, videos taken in real-world mobile settings exhibit substantial content dynamics in terms of difficulty level across frames over time. To illustrate this, Figure 4.2 shows four frames of a video clip of biking captured using a mobile camera in the human activity video dataset UCF-101 [51]. Among them, since the entirety of both the biker and her bike is captured, frame (a) and (d) are relatively easier to recognize as biking activity. In contrast, frame (b) and (c) capture the biker with only part of the bike, and are thus relatively harder to recognize. In such case, a smaller model is sufficient for frame (a) and (d), but a more complex model is necessary for frame (b) and (c).

Figure 4.2: Illustration of four frames of a video clip of biking captured using a mobile camera in UCF-101 dataset: (a) and (d) are frames with contents that are easy to recognize; (b) and (c) are frames with contents that are hard to recognize.

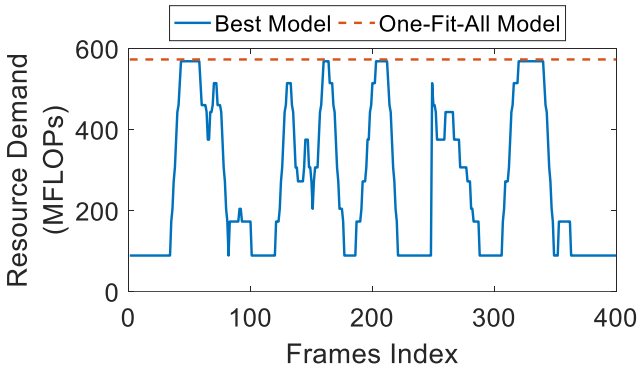

Figure 4.3: Blue solid curve: minimum computational consumption to correctly recognize the content in each frame (*optimal model*). Red dotted curve: computational consumption of the *one-size-fits-all model*.

The intrinsic dynamics of video contents create an opportunity to reduce computational consumption by matching the capacity of the DNN model to the difficulty level of each video frame. To quantify how much computational consumption can be reduced, we first profile the minimum computational consumption in terms of the number of floating point operations (FLOPs) that is needed to correctly recognize the content in each frame of an 400-frame video clip. Specifically, we derive ten model variants with different capacities from MobileNetV1 by varying its numbers of layers and filters. For each frame, we select the model variant with the lowest FLOPs that is able to correctly recognize the content in *that particular* frame (*optimal model*). We then compare it to the model variant with the lowest FLOPs that is able to correctly recognize the contents in *all* 400 frames (*one-size-fits-all model*) frame by frame.

Figure 4.3 shows our profiling result. As shown in the blue solid curve, the minimum computation consumed to correctly recognize the content in each frame changes frequently across frames. This observation strongly reflects the intrinsic dynamics of video contents illustrated in Figure 4.2. In addition, the difference between "areas" under the two curves reflects the *benefit* brought by the *optimal model*. The large difference indicates that considerable computational consumption can be reduced by matching the capacity of the model to the difficulty level of each video frame.

4.2.2 Drawbacks of Existing Solutions

The benefit of computation reduction motivates to dynamically change the model capacity to adapt to the contents of video frames. To achieve content adaptation, existing solutions such as Chameleon [18] use dynamic configuration where XXX. While effective as a solution for resourceful systems, dynamic configuration is a mismatch to resource-constrained mobile platforms for the following two reasons.

First, dynamic configuration requires all the model variants with various capacities to be installed in the mobile system. This is, unfortunately, not a scalable solution and could lead to large memory footprint. For the example used in Figure 4.3, a single MobileNetV1 model is 14 MB but the total memory footprint of ten model variants is 67 MB; the memory footprint would only increase if the number of model variants increases.

Second, dynamic configuration incurs large overheads on searching for the optimal model variant and model switching at runtime. Take model switching as an example. Model switching involves two steps: *model initialization* (i.e., allocating memory space for the model to switch to) and *parameter loading* (i.e., loading the model parameters into the allocated memory space). As shown in Figure 4.3, model switching could occur very frequently (110

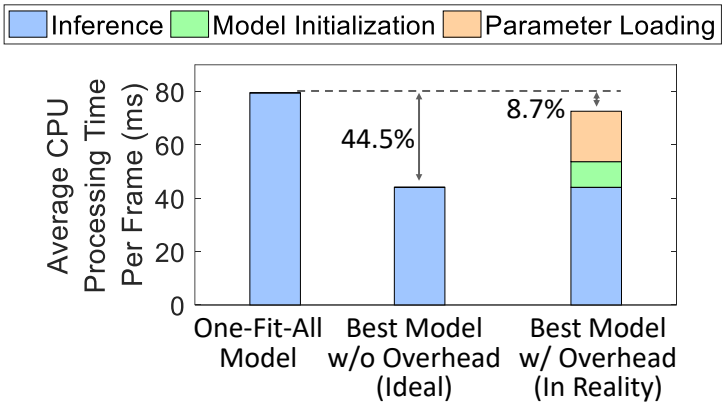

Figure 4.4: Benefit brought by the adaptation vs. model switching overhead of the dynamic configuration approach.

times within 400 frames) because the contents of videos captured by mobile cameras can change quite drastically in a short period of time. To quantify model switching overhead, we profile the average processing time of both model initialization and parameter loading of all the model switching occurred in the same 400-frame video clip on the Samsung Galaxy S8 smartphone CPU. To make the result more meaningful, we also profile the average inference time per frame of *optimal model* and *one-size-fits-all model*.

Figure 4.4 shows our result. The average inference time per frame of *one-size-fits-all model* and *optimal model* is 79.5 ms and 44.1 ms. The difference between them measures the benefit brought by the adaptation: by using *optimal model* instead of *one-size-fits-all model* on each frame, we are able to reduce 44.5% of the computation. However, the average processing time of model initialization and parameter loading is 18.9 ms and 9.6 ms, This model switching overhead drops the *actual* computation reduction to only 8.7%, which cuts the benefit brought by the adaptation significantly.

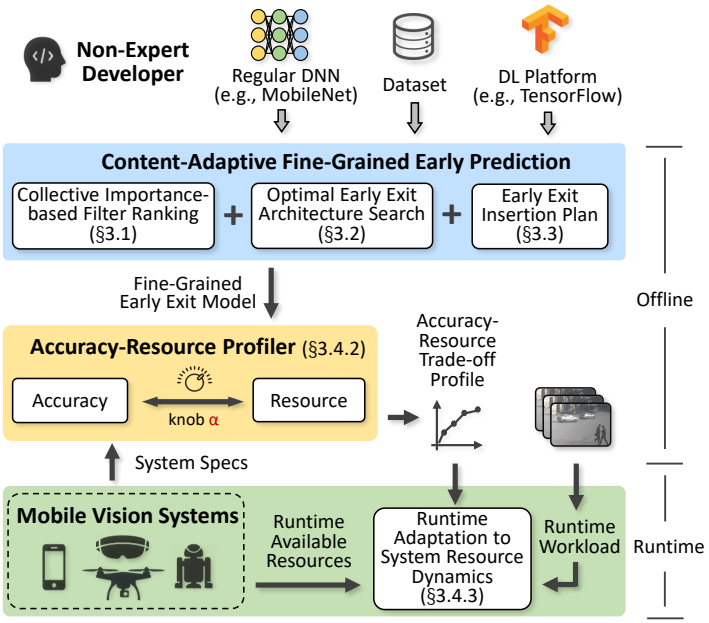

Figure 4.5: FlexDNN architecture.

4.3 Design of FlexDNN

In this work, we present FlexDNN, a content-adaptive framework for computation-efficient DNN-based mobile video stream analytics, that effectively addresses the drawbacks of existing solutions. Figure 4.5 illustrates the high-level architecture of FlexDNN. As an overview, to address the limitation caused by coarse-grained design, FlexDNN adopts a fine-grained design to make early predictions at the granularity of filters. It incorporates a collective importance-based filter ranking scheme to rank the importance of filters within each convolutional layer (§4.3.1). Such filter ranking lays the foundation of FlexDNN and allows us to systematically generate the optimal architecture for each early exit through an architecture search scheme (§4.3.2) and derive the optimal early exit insertion plan through an optimization formulation (§4.3.3). Lastly, the flexible architecture of the content-adaptive computation-efficient model generated by FlexDNN provides a natural mechanism to offer trade-off between accuracy and resource demand (§4.3.4.2). Such mechanism allows

FlexDNN to adapt to both workload and system resource dynamics at runtime (§4.3.4.3).

4.3.1 Filter Ranking based on Collective Importance

4.3.1.1 Motivation

The foundation of FlexDNN is to rank filters based on their importance and identify less important filters within each layer. To rank the importance of filters of a particular layer, existing work adopts the approach which ranks each filter independently based on a predefined importance indicator. For instance, in [1], $\mathcal{L}2$-norm is used as the indicator based on the heuristic that important filters tend to have larger $\mathcal{L}2$-norm values. Unfortunately, this approach has a key drawback: it ignores the dependence between filters in each layer. As a result, information contained in the top-ranked filters can be highly overlapped, which is the root cause of redundancy.

4.3.1.2 Collective Importance-based Filter Ranking

To address the drawback of existing filter importance ranking approaches, we propose a filter importance ranking scheme that takes filter dependence into account to rank the collective importance instead of individual importance of filters within each layer.

Our collective importance-based filter ranking scheme is inspired by the sequential backward selection (SBS) approach in feature selection literature [52]. At a high level, our scheme starts with all filters, and iteratively removes the filter that least reduces the inference accuracy from the filter set.

Specifically, our scheme maintains two lists: ranked and unranked, with ranked initialized as being empty and unranked initiated with the full set of filters included in a convolutional

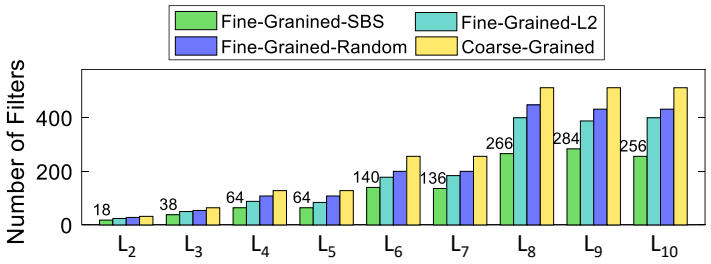

Figure 4.6: Comparison of the minimum number of filters needed to ahieve the same accuracy as using all filters within each layer between collective importance-based ranking scheme and the scheme based on independent ranking.

layer. For each filter f_i in unranked, we temporally drop it from unranked, add an early exit with the feature maps generated by the filters inside unranked as its input, and fine-tune the parameters of the early exit. Next, we obtain the validation accuracy of this early exit and store it in a table acc as a key-value pair with the key being f_i and the value being the validation accuracy. The dropped filter f_i is then added back to unranked. This procedure iterates until all the filters are gone through. Finally, the filter corresponds to the highest accuracy in the acc is identified as the least important filter among unranked. Hence, we permanently drop this filter from unranked, and insert it at the top of ranked. This process terminates untill the number of filters in unranked is less than 1/5 filters of this layer. This is because empirically the feature map extracted by less than 1/5 filters is not enough for an early exit to make early prediction with high confidence. We treat each of the remaining filters in unranked equally important, and add all of these remaining filters inside unranked to the top of ranked. Based on the ranked filters inside ranked, FlexDNN is able to find the minimum number of filters that an early exit needs to achieve the same accuracy as using all the filters within each layer. This is achieved by iteratively dropping the lowest ranked filters until the accuracy of early exit starts to drop.

4.3.1.3 Superiority of Collective Importance-based Filter Ranking

We compare our scheme to independent filter importance ranking scheme based on $\mathcal{L}2$-norm. To do this, we use UCF-15 as the video dataset and MobileNetV1 as our base model to rank the importance of filters of each layer. We find the minimum number of filters to achieve the same accuracy as using all filters within each layer for both our scheme (Fine-Grained-SBS) and the independent filter importance ranking scheme based on $\mathcal{L}2$-norm (Fine-Grained-L2).

Figure 4.6 shows the result. Due to the limitation of space, we do not show the results of L11 to L13 since they have similar results as L10. As shown, the minimum number of filters obtained by our collective importance-based filter ranking scheme is lower than $\mathcal{L}2$-norm across all the layers. In particular, our scheme is able to identify up to 28.1% more redundant filters compared to $\mathcal{L}2$-norm. This is because our scheme accounts for the dependence between filters within each layer and thus only selects filters that contain complementary information.

4.3.2 Searching for the Optimal Early Exit Architecture

4.3.2.1 Motivation

An early exit is essentially a self-contained neural network that can make early predictions. However, designing the architecture of early exits is not trivial: early exits consume computation. Computation consumed by frames that fail to exit is wasted and becomes the overheads. As such, it is necessary to minimize such overheads by using least number of layers and filters to build each early exit. However, early exits with such extremely lightweight architecture could exit much less frames, diminishing the benefit of early exits. Therefore, there exists a trade-off between early exit rates and computational overheads in the design

space of early exit architecture. Moreover, the locations of early exits to insert at also affect the early exit rates. This requires us to design the architecture for each early exit based on its inserted location, which makes the task of early exit architecture design even more challenging.

Existing work such as BranchyNet designs the architecture of early exits based on heuristics and considerable efforts on trial and error, but does not provide detailed design guidelines due to the complexity of this design task. As a result, their approach may not be generalized to other models. Moreover, it forces developers to have decent knowledge on DNN architecture and to spend considerable efforts on trial and error, prohibiting its wide adoption in practice.

4.3.2.2 Early Exit Architecture Search Scheme

To address this issue, we adopt a fundamentally different approach. Instead of manually designing the architecture based on heuristics and trials and errors, we propose a scheme based on architecture search to find the optimal architecture that optimizes the trade-off between early exit rates and computational overheads for each early exit.

In general, network architecture search techniques can be grouped into two categories: 1) the bottom-up approach that searches for an optimal cell structure based on reinforcement learning (RL) or genetic evolution and stacks cells together to form a network [53–55]; 2) the top-down approach that prunes an over-parameterized network until the optimal network architecture is found [1, 56]. Although both approaches work in our scenario, in this work, we select the top-down approach due to its more efficient search process.

At a high level, our early exit architecture search scheme starts with inserting an over-parameterized early exit at every possible location in a DNN model. It then maximizes the

early exit rate by training with emphasis on important filters followed by minimizing the computational overheads of early exits via iterative layer and filter pruning. In doing so, the best trade-off between early exit rate and computational overhead for each individual early exit is achieved.

We explain the details of each stage below.

- **Stage-1: Insert Over-Parameterized Early Exits**. For each layer, we insert M early exits where k^{th} early exit uses the first $\lceil kN/M \rceil$ most important filters ranked in ranked as input, where N denotes the total number of filters of this layer. We initialize each early exit with four layers with each layer having twice as many filters as its corresponding previous layer. We increase the number of filters by the factor of two so as to properly encode the increasingly richer representations as we go deeper. Similar setup can be found in popular DNNs such as MobileNets and InceptionV3.
- **Stage-2: Maximize Early Exit Rates by Training with Emphasis on Important Filters**. Given how we insert early exits in our previous step, the higher a filter is ranked in ranked, the more frequent the filter is used by early exits. To force these frequently used filters to learn more salient features, we train those filters with "emphasis" by assigning lower dropout rate compared to filters that are lower ranked. In doing so, this process essentially maximizes the exit rate of each early exit. We detail the definition of early exit rate in §4.3.2.3.
- **Stage-3: Minimize Overheads by Pruning Layers and Filters**. Although the exit rate of each early exit is maximized, the overhead of each over-parameterized early exit is still significant due to over-parameterization. FlexDNN minimizes this overhead by iteratively pruning layers and filters of each early exit until the exit rate starts to drop.

Specifically, for each early exit, we start with layer-wise pruning by iteratively pruning its layers until its exit rate drops. We then apply filter-wise pruning by iteratively pruning lower ranked filters within each remaining layer until its exit rate drops. As a result, the architecture of each inserted early exit achieves the optimized trade-off between the exit rate and computational overhead.

4.3.2.3 Early Exit Rate

Early exit rate of a given early exit reflects the probability of a frame to be exited to avoid further computation. A frame is exited if its confidence score is higher than a pre-defined threshold. Formally, our confidence score is defined as:

$$Conf(\mathbf{y}) = 1 + \frac{1}{\log C} \sum_{c \in C} y_c \log y_c \tag{4.1}$$

where $\mathbf{y} = [y_1, y_2, ..., y_c, ..., y_C]$ is the softmax classification probability vector generated by an early exit, C is the total number of classes, and $\sum_{c \in C} y_c \log y_c$ is the negative entropy of the softmax classification probability distribution over all the classes. The threshold for each early exit such that the frames are exited without loss of accuracy can be obtained using cross-validation [19]. We denote those thresholds as:

$$\mathbf{T_{lossless}} = (T_{EE_1}, ..., T_{EE_i}, ..., T_{EE_K}) \tag{4.2}$$

where EE_i denotes the i^{th} early exit of $\tilde{Net}$, K is total number of early exits inserted in $\tilde{Net}$.

4.3.3 Early Exit Insertion Plan

4.3.3.1 Motivation

Although the optimized trade-off between the exit rate and computational overhead of each early exit is achieved by our architecture search scheme explained above, the optimized trade-off for the entire network is not. This is because early exits have been inserted at every possible location throughout the network and hence accumulate immense overhead altogether.

To obtain the globally optimized trade-off, in contrast to BranchyNet and MSDNet which manually determine the early exit insertion locations by trial and error, FlexDNN adopts a systematic approach to derive an optimal early exit insertion plan. At a high level, we formulate the derivation of early exit insertion plan as an optimization problem and solve it using an efficient greedy heuristic that greedily prunes inefficient early exits in an iterative manner.

4.3.3.2 Problem Formulation

FlexDNN formulates the derivation of early exit insertion plan as the following optimization problem:

$$Net^* = \arg\min_{\tilde{Net}} Res(\tilde{Net}) \tag{4.3}$$

where Net^* is the model with optimal early exit insertion plan, $\tilde{Net}$ is the set of candidates with all the possible insertion combinations, $Res(\cdot)$ evaluates the computational consumption of a specific insertion plan.

Solving Eq.(4.3) by searching all the possible insertion plans is computationally expensive because there are 2^K combinations, where K is the total number of insertion locations. To

reduce the complexity of this process, FlexDNN utilizes a greedy heuristic to obtain an approximation solution based on the following key observation.

When a model is densely inserted with early exits, pruning any of the early exits leads to reduction of computational consumption. Among all early exits, pruning the early exit with smallest early exit rate and large computational overhead leads to largest reduction of computational consumption (i.e., inefficient early exit). Based on this observation, FlexDNN greedily prunes these these inefficient early exits in an iterative manner. FlexDNN terminates this iteration process when computational consumption of the model starts to increase. This is because at this stage, all the remaining early exits are contributing to the computational efficiency and hence are beneficial to the model. As such, the remaining early exits represent the optimal early exit insertion plan.

4.3.3.3 Efficiency of Early Exit

To identify inefficient early exits, we define a metric Q that quantifies the quality of the trade-off between early exit rate and computational overhead of a particular early exit. Specifically, for early exit j, we define its quality Q_j as the ratio between the gain G_j it brings and the cost C_j it incurs:

$$Q_j = G_j / C_j \tag{4.4}$$

where C_j is the computation consumed by the this early exit, and G_j is the computation avoided due to the existence of the early exit. Both C_j and G_j are measured by the number of floating point operations.

Figure 4.7 illustrates how G_j and C_j are calculated. In particular, the figure shows three consecutive early exits inserted at the i^{th}, j^{th}, and k^{th} early exit positions of base

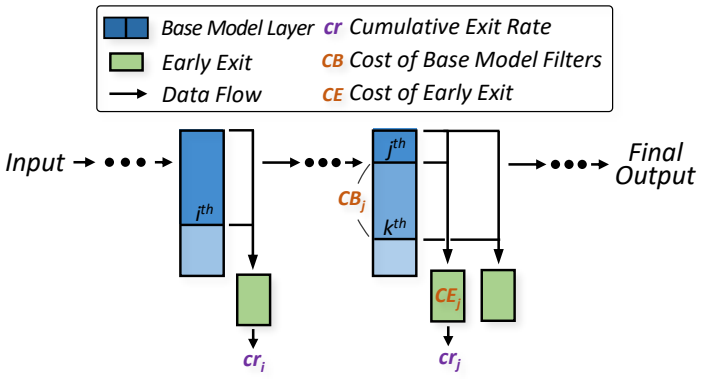

Figure 4.7: Illustration of derivation of quality of an early exit.

model ($i < j < k$). Let F denote the total number of input frames; cr_i and cr_j denote the cumulative exit rate of the i^{th} and j^{th} early exit, respectively ($0 \leq cr_i \leq cr_j \leq 1$); CE_j denote the computation consumed by the j^{th} early exit per input frame; and CB_j denote the computational cost of the base model between the j^{th} and k^{th} early exit per input video frame.

Since $F * cr_i$ input frames exit at the i^{th} early exit, there are $F * (1 - cr_i)$ input frames going through the j^{th} early exit. As a result, C_j is calculated as:

$$C_j = F * (1 - cr_i) * CE_j \tag{4.5}$$

There are $F * (cr_j - cr_i)$ input frames exiting at the j^{th} early exit. These input frames avoid further computational cost incurred between the j^{th} and k^{th} convolutional layer of the base model. Therefore, G_j is calculated as:

$$G_i = N * (cr_j - cr_i) * CB_j \tag{4.6}$$

4.3.4 Runtime Adaptation to Workload and System Resource Dynamics

4.3.4.1 Motivation

The moving speed of mobile vision systems varies depending on contexts. To ensure obtaining all the necessary information contained in video frames, videos captured by mobile vision systems moving in high speeds require high frame processing rates while videos captured by mobile vision systems moving in low speeds (or stationary) require low frame processing rates. Therefore, the workload of processing videos captured by mobile vision systems can vary in different contexts.

In addition, mobile vision systems usually run multiple application concurrently and hence their available runtime computation resources are dynamic due to the events such as starting new applications, closing existing applications, and application priority changes. In addition, the battery status of these systems changes and hence requires different energy budgets. Therefore, the system resources in mobile vision systems can also change dynamically.

To adapt to such workload and system resource dynamics, FlexDNN uses a knob to trade off accuracy and resource demand. Specifically, given the accuracy-resource profile generated offline, FlexDNN dynamically adjusts the knob to adapt to the current workload and system resource at runtime such that FlexDNN is able to achieve the highest accuracy.

4.3.4.2 Accuracy-Resource Profile

To obtain the accuracy-resource profile, we change the exit threshold of Net^* obtained in Eq.(4.3) by applying a knob $\alpha \in (0, 1]$ on $\mathbf{T_{lossless}}$ (defined in Eq.(4.2)). Formally, the exit

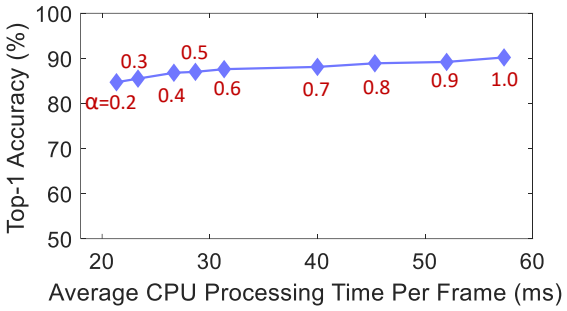

Figure 4.8: Accuracy-latency profile under different α values.

threshold $\mathbf{T}$ is given by:

$$\mathbf{T} = \alpha \mathbf{T_{lossless}} = (\alpha T_{EE_1}, ..., \alpha T_{EE_i}, ..., \alpha T_{EE_K}) \tag{4.7}$$

In essence, the lower α is, the less $\mathbf{T}$ are. With a lower α, a video frame is able to exit with a lower confidence. This creates a natural mechanism to trade off accuracy and resource consumption by adjusting the value of α.

Given $\mathbf{T}$, the accuracy-resource profile is then obtained by offline profiling the accuracy and the corresponding resource usage on a given mobile platform. In this work, FlexDNN focuses on two types of resources: computational resource and energy resource. Computational resource is evaluated as average CPU/GPU processing latency per frame and energy resource is evaluated as average energy consumption per frame. We obtain both accuracy-latency and accuracy-energy consumption profiles under 100% CPU/GPU utilization.

As an example, Figure 4.8 shows the accuracy-latency profile with α varying from 0.2 to 1.0. As shown, by adjusting α to different values, FlexDNN is able to trade off accuracy and frame processing latency. Specifically, FlexDNN reduces 62.8% frame processing latency by only sacrificing 4.9% Top-1 accuracy. We do not show the accuracy-energy consumption profile because it exhibits a similar profile.

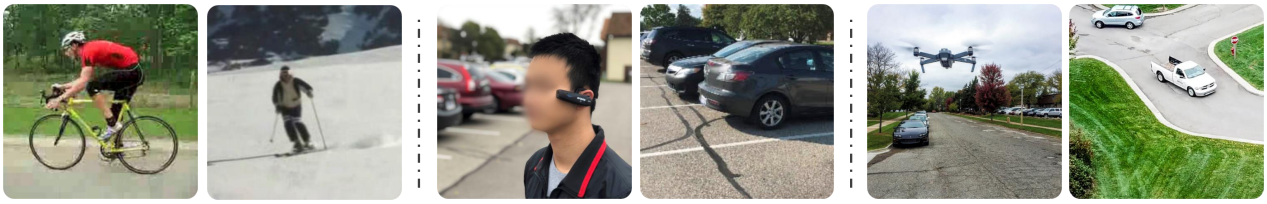

Figure 4.9: (a) UCF-15: example video frame of biking (left) and skiing (right). (b) Place-8: illustration of data collection using a ORDRO EP5 head-mounted camera (left); example video frame of parking lot (right). (c) TDrone: illustration of data collection using a DJI Mavic Pro drone (left); example video frame of traffic surveillance in the residential area (right).

4.3.4.3 Runtime Adaptation

The goal of runtime adaptation is to find the optimal α value such that FlexDNN sacrifices minimal accuracy while still being able to process all the frames of given by the workload within the system resource budget. To achieve this goal, FlexDNN continuously monitors the workload and system resources and maps them into the latency budget and energy budget. Based on the latency and energy budget, FlexDNN derives the optimal α value as follows. Specifically, FlexDNN takes in three inputs: 1) the frame rate of the current workload fr; 2) the ratio of the allocated compute resource at runtime to total compute resource of the mobile system r; and 3) the energy resource budget per frame Bud_e^{max}. Given those three inputs, FlexDNN first calculates the latency budget with 100% CPU/GPU utilization $Bud_t = 1/fr$. Next it adapts Bud_t to the allocated compute resource at runtime by dividing Bud_t with r: $Bud_t^{max} = Bud_t/r$. For energy budget, FlexDNN directly uses Bud_e^{max}. Finally, the optimal α value is obtained by finding the largest α from accuracy-latency and accuracy-energy consumption profiles such that FlexDNN runs within both latency budget Bud_t^{max} and energy budget Bud_e^{max}.

Application	Target Mobile Platform	Dataset	Number of Frames	Base Model	FlexDNN Model
Activity Recognition	Mobile Phone	UCF-15	1,080K	MobileNet, VGG-16	M-U, V-U
Scene Understanding	AR Headset	Place-8	123K	MobileNet, VGG-16	M-P, V-P
Traffic Surveillance	Drone	TDrone	146K	MobileNet, VGG-16	M-T, V-T

Table 4.1: Summary of three applications, two base models, and six generated FlexDNN models.

4.4 Implementation

We select MobileNets [30] and VGGNet [36] as our base DNN models to build our content-adaptive computation-efficient models. MobileNets are state-of-the-art computation-efficient DNN models designed for mobile platforms. To demonstrate the generability across DNN models as well as investigate the benefit FlexDNN brings to resource-demanding base models, we selectVGG-16 as a representative model. In our experiments, we insert early exits at the filters within the depthwise layers of the depthwise convolution blocks of MobileNet and the convolutional layers of VGG-16.

To demonstrate the generability of FlexDNN across applications, we use FlexDNN and three video datasets to build three representative continuous mobile vision applications for three mobile platforms:

Activity Recognition on Mobile Phones. Automatic labeling human activities in videos is becoming a very attractive feature for smartphones. This application aims to recognize activities performed by an individual from video streams captured by mobile phone cameras. To design this application, we use UCF-101 human activity dataset [51]. UCF-101 contains video clips of 101 human activity classes captured by either fixed or mobile cameras in the wild. We selected video clips of 15 activities (e.g., biking and skiing) captured by mobile cameras as our dataset (named UCF-15) to build and evaluate this application. UCF-15

consists of 1,080K video frames. We split training and test videos by following the original paper [51]. Example video frames of UCF-15 are shown in Figure 4.9 (a).

Scene Understanding for Augmented Reality. Sc-ene understanding is one of the core capabilities of augmented reality. This application aims to recognize places from video streams captured by head-mounted cameras. To design this application, we collected our own video clips in the wild with IRB approval due to lack of publicly available datasets. During data collection, participants were instructed to collect first-person view video footage from diverse places by wearing the ORDRO EP5 head-mounted camera [57]. Frames in all the video clips are manually labelled. From the labeled video clips, we selected 8 most common places that participants visited (e.g., parking lot, kitchen) as our dataset (named Place-8) to build and evaluate this application. Place-8 consists of 123K video frames. To avoid model overfitting, we use the same 8 places from Places-365 [44] as our training set, and our self-collected video frames as test set. Illustration of data collection using the head-mounted camera and one example video frame of Place-8 are shown in Figure 4.9 (b).

Drone-based Traffic Surveillance. Due to its mobility, a traffic surveillance drone is able to track traffic conditions in a large area with an extreme low cost that traditional fixed video camera-based traffic surveillance systems could not provide [58]. This application aims to detect vehicles from video streams captured by drone cameras. Due to lack of publicly available datasets, we use one of the most advanced commercial drones, DJI Mavic Pro [59], to collect our own traffic surveillance video clips in the wild with IRB approval. To ensure diversity, videos were recorded under various drone camera angles (25°to 90°), flying heights (2.5m to 51.2m), speeds (1m/s to 11.2m/s), weather conditions (cloudy, sunny), and road types (residential, urban, highway). Frames in all video clips are manually labelled. This

dataset (named TDrone) consists of 146K video frames. We split it into 15% and 85% for training and testing. Illustration of data collection using drone and one example video frame of TDrone are shown in Figure 4.9 (c).

We implement FlexDNN in TensorFlow. For each of the three applications, we use FlexDNN to generate a content-adaptive computation-efficient model from MobileNet and VGG respectively for our evaluation. We list the three applications, two base models, and six models generated by FlexDNN in Table 4.1.

4.5 Evaluation

4.5.1 Methodology

Baselines. We compare FlexDNN with two baselines: 1) Base model of FlexDNN: MobileNet and VGG-16 (i.e., content-agnostic). We compare FlexDNN model to its corresponding base model to demonstrate the benefit brought by content adaptation. 2) BranchyNet [19]: a recent state-of-the-art content-adaptive model based on the coarse-grained early exit design. To make fair comparisons, we use MobileNet and VGG-16 as the base models for BranchyNet.

Deployment Platforms. To match the three applications to their target mobile platforms, we deploy M-U and V-U of the Activity Recognition application on a Samsung Galaxy S8 smartphone and run them on the smartphone CPU; we deploy M-P and V-P of the Scene Understanding application as well as M-T and V-T of the Traffic Surveillance application on a NVIDIA Jetson Xavier development board [60] and run them on the onboard GPU. We choose NVIDIA Xavier because it is the state-of-the-art mobile GPU designed for next-

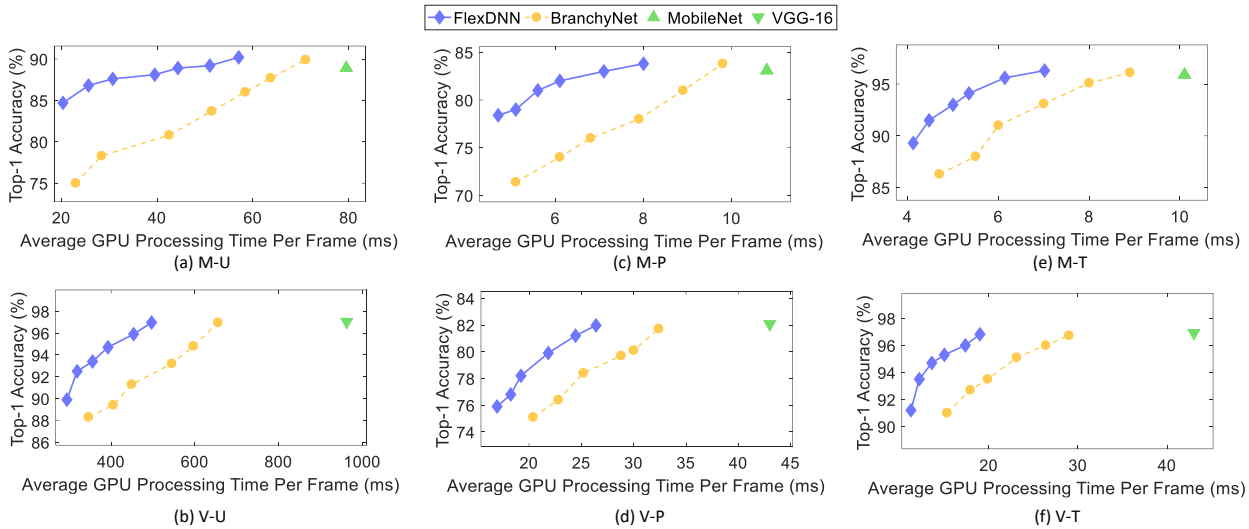

Figure 4.10: Comparison between FlexDNN and baseline approaches in the accuracy-frame processing time space.

generation DNN-based intelligent mobile systems such as AR headsets, drones, and robots.

Evaluation Metrics. We use three metrics to evaluate the performance of FlexDNN and the baselines: 1) *inference accuracy*: we use Top-1 accuracy of all the video frames in the test set as the metric of inference accuracy; 2) *computational cost*: we use average CPU/GPU processing time (with 100% CPU/GPU utilization) per frame as the metric of computational cost; 3) *energy consumption*: computational cost directly translates to energy consumption. We thus use the average energy consumption per frame as our third metric.

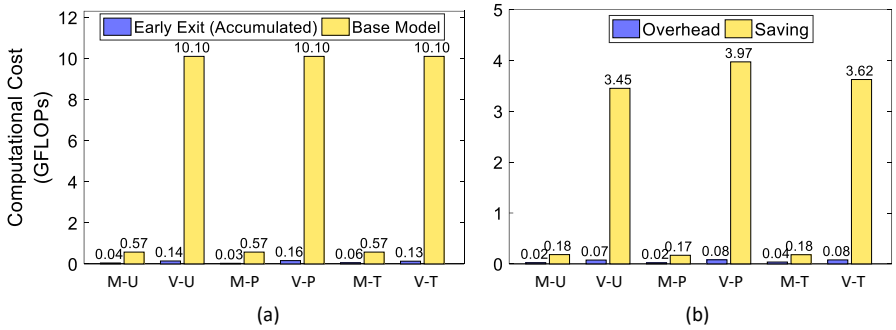

Figure 4.11: (a) Comparison between the accumulated computational cost of all the early exits and the computational cost of the base model. (b) Comparison between saving (the average computation saved from early exiting per frame) and overhead (the average computation consumed by the early exits each frame goes through but fails to exit).

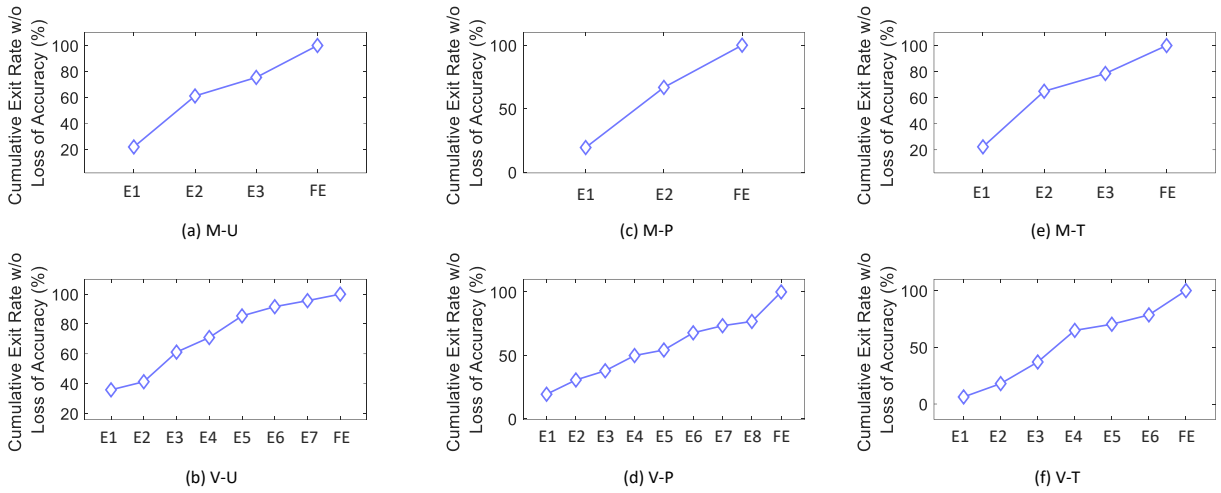

Figure 4.12: Cumulative exit rate at each "early exit" without loss of accuracy. "early exit" (marked as E1, E2, ...) are ordered based on their distances to the input layer (i.e., E1 is the earliest exit). FE denotes the regular exit of the base model.

4.5.2 Model Performance

In this section, we focus on profiling the models generated by FlexDNN to highlight their key characteristics.

4.5.2.1 High Early Exit Rate

The generated FlexDNN model is able to achieve high early exit rates through its early exits without loss of accuracy. To quantify this characteristic, we profile each of six models on the test set, and measure the cumulative exit rate at each early exit.

Figure 4.12 shows the locations of the inserted early exits (e.g., E1 represents that the early exit is closest to the input layer) and their cumulative exit rates. As shown, for each of the six models, the increasing cumulative exit rates imply the significance of each inserted early exit. Accumulatively, these early exits are able to exit 75.4%, 66.8%, 78.5%, 95.6%, 76.8%, and 78.5% of the frames on M-U, M-U, M-P, V-P, V-T, and V-T, respectively.

This result indicates that our technique is effective at identifying efficient early exits while

pruning less efficient ones.

We also observe that when MobileNet is the base model, FlexDNN prunes more early exits compared to VGG-16. This is because MobileNet is a very efficient base model in terms of accuracy-computational overhead ratio, and hence inserting too many early exits brings more cumulative computational overhead in proportion to its base model. In contrast, VGG-16 is a much less efficient base model, and therefore inserting early exits brings much less cumulative computational overhead in proportion to its base model. As such, FlexDNN only inserts up to three early exits when the base model is MobileNet, and inserts up to eight early exits when the base model is VGG-16.

4.5.2.2 Computation-Efficient Early Exits

The early exits incorporated in the generated FlexDNN models are computation-efficient. To quantify this characteristic, we compare the accumulated computational cost of all the early exits of the FlexDNN model with the computational cost of its base model. As shown in Figure 4.11(a), the accumulated computational cost of all the early exits of the FlexDNN model is only 7.8%, 1.4%, 5.0%, 1.6%, 9.7%, and 1.3% of its own base model for M-U, V-U, M-P, V-P, M-T, and V-T, respectively, indicating that even in the worst case scenario where a video frame goes through *all* the inserted early exits, these exits altogether incur *marginal* computational overhead compared to the base model.

4.5.2.3 High Computational Consumption Reduction

Because of high early exit rate (§5.2.1) and computation-efficient early exit design (§5.2.2), the generated FlexDNN model is able to achieve high computational consumption reduction. To quantify this characteristic, we use the average computation saved from early exiting

per frame to quantify the saving; and use the average computation consumed by the early exits each frame goes through but fails to exit to quantify the overhead. As shown in Figure 4.11(b), the overhead is substantially lower than the saving for each model: the saving-overhead-ratio is 8×, 46×, 8×, 49×, 5×, and 46× for M-U, V-U, M-P, V-P, M-T, and V-T, respectively. As we will show in the next subsection, the achieved high saving-overhead-ratio is directly translated into various system performance improvement at runtime.

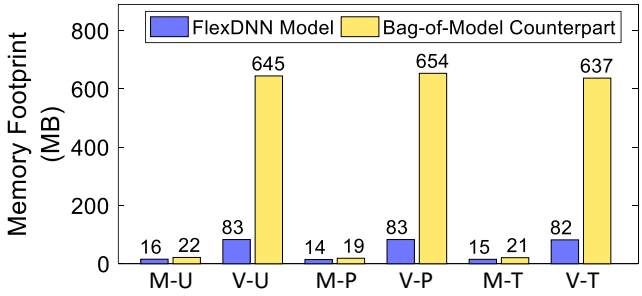

Figure 4.13: Memory footprint comparison between the FlexDNN model and its bag-of-model counterpart.

4.5.2.4 Compact Memory Footprint

The generated FlexDNN model has a compact memory footprint. To quantify this superiority, we compare its model size with its bag-of-model counterpart whose number of model variants equals the number of exits (early exits plus the regular exit of the base model) in the FlexDNN model.

Figure 4.13 illustrates the comparison result. As shown, FlexDNN is able to reduce the memory footprint for 1.4×, 7.8×, 1.3×, 7.8×, 1.4×, and 7.8× for M-U, V-U, M-P, V-P, M-T, and V-T, respectively. This result demonstrates the considerable benefit of FlexDNN on memory footprint reduction.

4.5.3 Runtime Performance

In this section, we focus on evaluating the runtime performance of the models generated by FlexDNN when deployed on mobile platforms and comparing them with baselines.

4.5.3.1 Top-1 Accuracy vs. Frame Processing Time

Figure 4.10 compares FlexDNN with the baselines in the accuracy-frame processing time space. For FlexDNN, each blue diamond marker on the blue curve represents the runtime performance obtained by using a particular α value defined in Eq.(4.7). We make two main observations.

First, FlexDNN outperforms both MobileNet and VGG-16, which represent the content-agnostic DNN models without modification of FlexDNN. Under the same accuracy achieved by MobileNet, FlexDNN reduces the computational consumption by 28.2%, 25.9%, 30.5%, 48.4%, 38.6%, and 55.8% for M-U, V-U, M-P, V-P, M-T, and V-T, respectively. This result demonstrates the superiority of FlexDNN due to its content-adaptive capability. Second, FlexDNN also outperforms BranchyNet across the accuracy-frame processing time space. For instance, when constrained at 58 ms average CPU processing time per frame for M-U, FlexDNN is able to achieve approximately 4% accuracy gain over BranchyNet. When constrained at 25 ms per frame for M-U, FlexDNN is able to achieve approximately 8% accuracy gain over BranchyNet. This result demonstrates the superiority of FlexDNN due to its fine-grained design and architecture superiority.

4.5.3.2 Reduction on Energy Consumption

Computation cost directly translates to energy consumption. Besides reducing frame processing latency, FlexDNN also consumes considerably less energy compared to baselines.

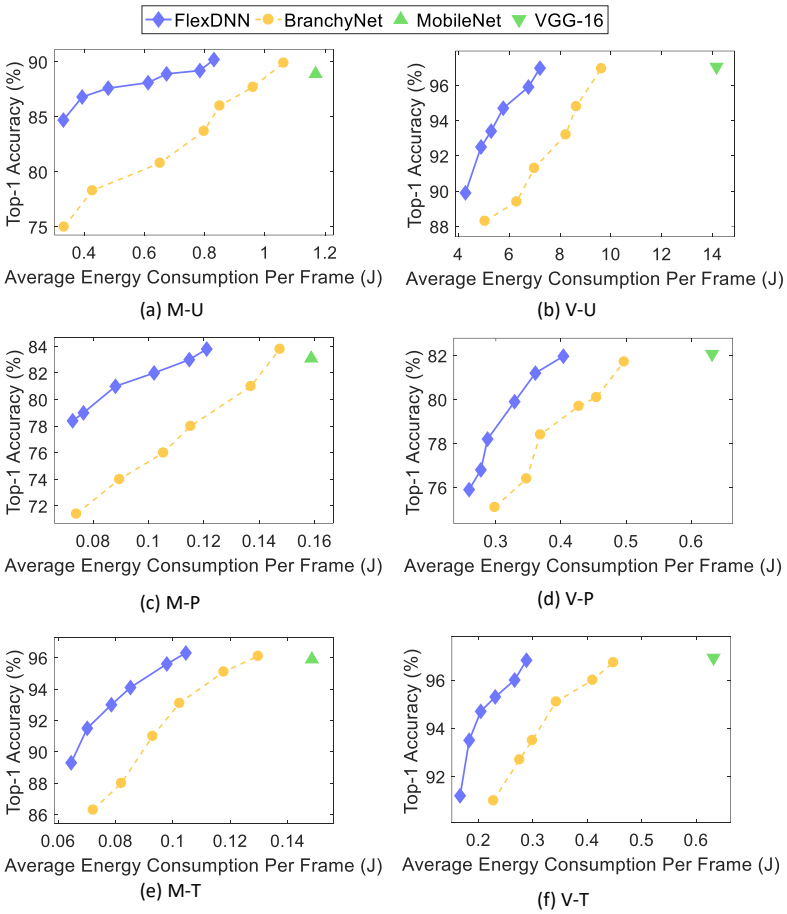

Figure 4.14: Comparison between FlexDNN and baseline approaches in the accuracy-energy consumption space.

As shown in Figure 4.14, under the same accuracy achieved by the corresponding base model, FlexDNN reduces the energy consumption by 31.8%, 23.8%, 23.0%, 50.4%, 38.6%, and 53.8% for M-U, V-U, M-P, V-P, M-T, and V-T, respectively. Across the entire accuracy-energy consumption space, compared to BranchyNet, FlexDNN also significantly reduces energy consumption. For instance, when the accuracy is 90% for M-U FlexDNN is able to reduce approximately 20% energy compared to BranchyNet. Similarly, when the accuracy is 85% for M-U, FlexDNN is able to reduce approximately 50% energy compared to BranchyNet.

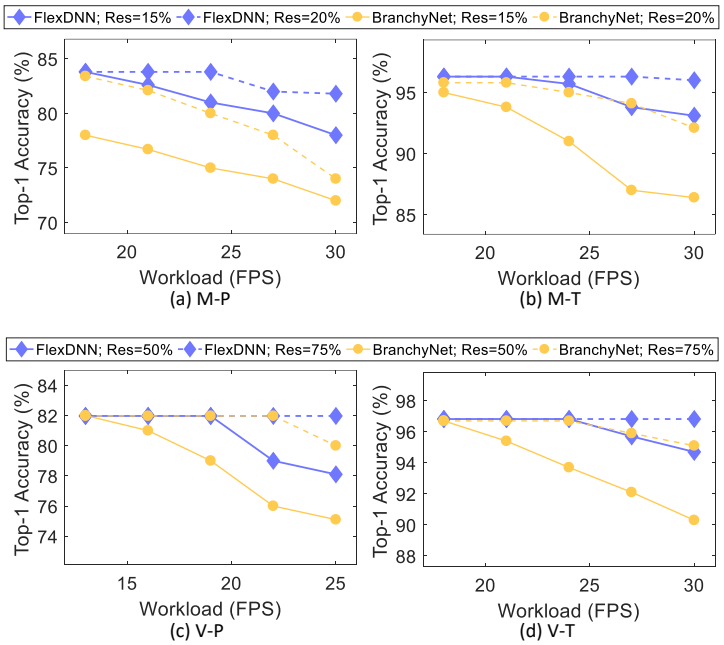

Figure 4.15: Performance of runtime adaptation to workload and system resource dynamics.

4.5.3.3 Performance of Runtime Adaptation

We compare the performance of runtime adaptation to workload and system resource dynamics between FlexDNN and BranchyNet. We did not evaluate on mobile CPU due to its low frame rate.

We use simulation to create both workload and system resource dynamics. Specifically, to simulate workload dynamics, for scenario when the base model is MobileNet, we increase the streaming video frame rate from 18 FPS to 30 FPS for M-P and M-T; and when the base model is VGG-16, we increase frame rate from 13 FPS to 25 FPS, and from 18 FPS to 30 FPS for V-P and V-T. To simulate system resource dynamics, we decrease GPU utilization from 20% to 15% for M-P and M-T; and decrease from 75% to 20% for V-P and V-T. We allocate smaller ratio of resource for scenario when the base model is MobileNet because it is less computational demanding model compared to VGG-16.

Figure 4.16 shows the results. When the resource constraint is less tight (Res = 20% for M-P and M-T, Figure 4.16(a)-(b); Res = 75% for V-P and V-T, Figure 4.16(b)-(c)) and

workload is low, FlexDNN and BranchyNet achieve comparable accuracies. However, as the workload increases, the accuracy of FlexDNN remains high while the accuracy of BranchyNet drops significantly. When the resource constraint is tight (Res = 15% for M-P and M-T, Res = 50% for V-P and V-T) and workload is the highest, FlexDNN outperforms BranchyNet by largest margins. In particular, FlexDNN outperforms BranchyNet by 6.0%, 6.5%, 3.0%, and 4.4% in accuracy for M-P, V-P, M-T, and V-T, respectively.

4.5.4 Performance on ImageNet

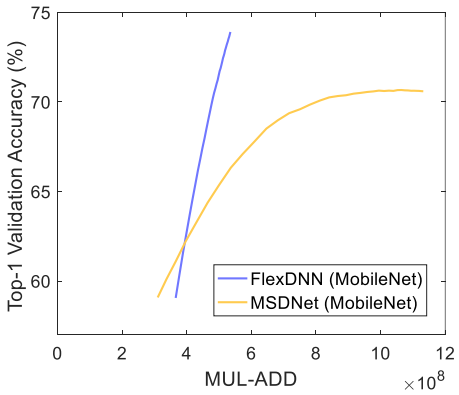

Figure 4.16: Top-1 accuracy comparison between MSDNet-mobile.

Finally, we compare the performance of FlexDNN and MSDNet. To make fair comparisons, both FlexDNN and MSDNet use MobileNetV2 for their base model. From the figure, we have four key observations. First, at the highest top-1 validation accuracy of MSDNet (MobileNet) (i.e., 70.6%), FlexDNN (MobileNet) consumes 51.3% less FLOPs compared to MSDNet (MobileNet). Second, when we sacrifice top-1 accuracy by 5% (i.e., 65.6%), FlexDNN (MobileNet) consumes 17.9% less FLOPs compared to MSDNet (MobileNet). Third, MSDNet (MobileNet) consumes less FLOPs than FlexDNN (MobileNet) when the top-1 validation accuracy is lower than 62.2%. However, considering that the accuracy is much lower than the highest accuracy, such models are less practically useful even if they have

less FLOPs. Lastly, FlexDNN is able to achieve higher top-1 accuracy (74.0%) compared to MSDNet (MobileNet) (70.6%).

In sum, FlexDNN is able to outperform MSDNet on ImageNet dataset.

4.6 Conclusion of FlexDNN

In this part, we presented the design, implementation and evaluation of FlexDNN, a content-adaptive framework for computation-efficient DNN-based mobile video stream analytics. FlexDNN addresses the limitations of existing solutions and pushes the state-of-the-art forward through its innovative fine-grained design and the automatic approach for generating the optimal architecture based on early exits for content adaptation. We used FlexDNN to built three continuous mobile vision applications on top of MobileNetV1, and used both mobile CPU and GPU platforms for runtime evaluation. Our evaluation results show that FlexDNN outperforms both computation-efficient content-agnostic and state-of-the-art content-adaptive approaches. We also show that FlexDNN outperforms state-of-the-art on ImageNet dataset.

Chapter 5

Conclusion

In this report, we cover two works: NestDNN and FlexDNN. NestDNN represents a framework that enables resource-aware multi-tenant on-device deep learning for mobile vision systems. The key idea of NestDNN is to generate a multi-capacity model that provides flexible resource-accuracy trade-offs, and to dynamically select the optimal one to jointly maximize their performance. To further enhance the computation-efficiency of DNN-based processing pipelines, we propose FlexDNN. FlexDNN is a novel content-adaptive framework that enables computation-efficient DNN-based on-device video stream analytics based on early exit mechanism. Compared to state-of-the-art early exit-based solutions, FlexDNN addresses their key limitations and pushes the state-of-the-art forward through its innovative fine-grained design and automatic approach for generating the optimal network architecture.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[3] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[4] F. Mokaya, R. Lucas, H. Y. Noh, and P. Zhang, "Burnout: a wearable system for unobtrusive skeletal muscle fatigue estimation," in *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*. IEEE, 2016, pp. 1–12.

[5] S. Nirjon, R. F. Dickerson, Q. Li, P. Asare, J. A. Stankovic, D. Hong, B. Zhang, X. Jiang, G. Shen, and F. Zhao, "Musicalheart: A hearty way of listening to music," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*. ACM, 2012, pp. 43–56.

[6] A. Nguyen, R. Alqurashi, Z. Raghebi, F. Banaei-kashani, A. C. Halbower, and T. Vu, "A lightweight and inexpensive in-ear sensing system for automatic whole-night sleep stage monitoring," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. ACM, 2016, pp. 230–244.

[7] Y. Wang, K. Wu, and L. M. Ni, "Wifall: Device-free fall detection by wireless networks," *IEEE Transactions on Mobile Computing*, vol. 16, no. 2, pp. 581–594, 2017.

[8] B. Fang, N. D. Lane, M. Zhang, A. Boran, and F. Kawsar, "Bodyscan: Enabling radio-based sensing on wearable devices for contactless activity and vital sign monitoring," in *The 14th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016, pp. 97–110.

[9] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[10] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *Advances In Neural Information Processing Systems*, 2016, pp. 1379–1387.

[11] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems*, 2017, pp. 2181–2191.

[12] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, vol. 1, no. 2, 2017, p. 3.

[13] P. Bahl, M. Philipose, and L. Zhong, "Vision: cloud-powered sight for all: showing the cloud what you see," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 53–60.

[14] R. LiKamWa and L. Zhong, "Starfish: Efficient concurrency support for computer vision applications," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 213–226.

[15] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. New York, NY, USA: ACM, 2016, pp. 123–136. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906396

[16] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 68–81.

[17] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17. New York, NY, USA: ACM, 2017, pp. 82–95. [Online]. Available: http://doi.acm.org/10.1145/3081333.3081360

[18] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 253–266.

[19] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *Pattern Recognition (ICPR), 2016 23rd International Conference on*. IEEE, 2016, pp. 2464–2469.

[20] 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," *arXiv preprint arXiv:1703.09844*, 2017.

[21] A. Puri, "A survey of unmanned aerial vehicles (uav) for traffic surveillance," *Department of computer science and engineering, University of South Florida*, pp. 1–29, 2005.

[22] "This Powerful Wearable Is a Life-Changer for the Blind," https://blogs.nvidia.com/blog/2016/10/27/wearable-device-for-blind-visually-impaired/.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[24] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1701–1708.

[25] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *NIPS*, 2014, pp. 487–495.

[26] "Google Clips," https://store.google.com/us/product/google_clips.

[27] "An On-device Deep Neural Network for Face Detection," https://machinelearning.apple.com/2017/11/16/face-detection.html.

[28] "Amazon DeepLens," https://aws.amazon.com/deeplens/.

[29] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance." in *NSDI*, vol. 9, 2017, p. 1.

[30] 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.

[31] X. Zeng, K. Cao, and M. Zhang, "Mobiledeeppill: A small-footprint mobile deep learning system for recognizing unconstrained pill images," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 56–67.

[32] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2016, pp. 1–12.

[33] B. Fang, J. Co, and M. Zhang, "DeepASL: Enabling Ubiquitous and Non-Intrusive Word and Sentence-Level Sign Language Translation," in *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Delft, The Netherlands, 2017.

[34] S. Naderiparizi, P. Zhang, M. Philipose, B. Priyantha, J. Liu, and D. Ganesan, "Glimpse: A programmable early-discard camera architecture for continuous mobile vision," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '17. New York, NY, USA: ACM, 2017, pp. 292–305. [Online]. Available: http://doi.acm.org/10.1145/3081333.3081347

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[37] 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.

[38] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *arXiv preprint arXiv:1707.06342*, 2017.

[39] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," *arXiv preprint arXiv:1611.06440*, 2016.

[40] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[41] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[42] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural networks*, vol. 32, pp. 323–332, 2012.

[43] G. Levi and T. Hassner, "Age and gender classification using convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2015, pp. 34–42.

[44] B. Zhou, A. Lapedriza, A. Khosla, A. Oliva, and A. Torralba, "Places: A 10 million image database for scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[45] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1717–1724.

[46] M. Huh, P. Agrawal, and A. A. Efros, "What makes imagenet good for transfer learning?" *arXiv preprint arXiv:1608.08614*, 2016.

[47] "Monsoon Power Monitor," https://www.msoon.com/LabEquipment/PowerMonitor/.

[48] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," 2018.

[49] D. B. Walther, B. Chai, E. Caddigan, D. M. Beck, and L. Fei-Fei, "Simple line drawings suffice for functional mri decoding of natural scene categories," *Proceedings of the National Academy of Sciences*, vol. 108, no. 23, pp. 9661–9666, 2011.

[50] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[51] K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," *arXiv preprint arXiv:1212.0402*, 2012.

[52] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[53] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[54] 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 *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[55] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[56] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300.

[57] "ORDRO EP5 Head-Mounted Camera," http://www.ordro.com.cn/product_detail/204, 2018.

[58] M. A. Khan, W. Ectors, T. Bellemans, D. Janssens, and G. Wets, "Uav-based traffic analysis: a universal guiding framework based on literature survey." ELSEVIER SCIENCE BV, 2017.

[59] "DJI Mavic Pro," https://www.dji.com/mavic, 2018.

[60] "NVIDIA Jetson AGX Xavier Developer Kit," https://developer.nvidia.com/embedded/buy/jetson-xavier-devkit, 2018.