FlexDNN: Input-Adaptive On-Device Deep Learning for Efficient Mobile Vision

Biyi Fang†, Xiao Zeng†, Faen Zhang*, Hui Xu*, Mi Zhang†
†Michigan State University, *AInnovation

***Abstract*—Mobile vision systems powered by the recent advancement in Deep Neural Networks (DNNs) are enabling a wide range of on-device video analytics applications. Considering mobile systems are constrained with limited resources, reducing resource demands of DNNs is crucial to realizing the full potential of these applications. In this paper, we present FlexDNN, an input-adaptive DNN-based framework for efficient on-device video analytics. To achieve this, FlexDNN takes the intrinsic dynamics of mobile videos into consideration, and dynamically adapts its model complexity to the difficulty levels of input video frames to achieve computation efficiency. FlexDNN addresses the key drawbacks of existing systems and pushes the state-of-the-art forward. We use FlexDNN to build three representative on-device video analytics applications, and evaluate its performance on both mobile CPU and GPU platforms. Our results show that FlexDNN significantly outperforms status quo approaches in accuracy, average CPU/GPU processing time per frame, frame drop rate, and energy consumption.**

***Keywords*-Mobile Deep Learning Systems, On-Device AI, Dynamic Deep Neural Networks, Mobile Vision**

I. Introduction

A. Motivation

Mobile vision systems such as mobile phones, drones, and augmented reality (AR) headsets are ubiquitous today. Driven by recent breakthrough in Deep Neural Networks (DNNs) [19] and the emergence of AI chipsets, state-of-the-art mobile vision systems start to use DNN-based processing pipelines [10], [14], [35], [36] and shift from *cloud-assisted* processing (e.g., MCDNN [10]) to *on-device* video analytics.

On-device video analytics requires processing streaming video frames at high throughput and returning the processing results with low latency. Unfortunately, DNNs are known to be computation-expensive [31], and high computation cost directly translates to high processing latency and energy consumption. Since mobile systems are constrained by limited compute resources and battery capacities, reducing the computation cost of DNN-based pipelines is crucial to any application built on top of on-device video analytics.

To reduce computation cost, most existing work pursues model compression techniques [8], [9], [24]. However, model compression yields an *one-size-fits-all* network that requires the same set of features to be extracted for all video frames *agnostic* to the content in each frame.

In fact, computation consumed by a DNN-based processing pipeline is heavily dependent on the *content* of input video frames [15]. For video frames with contents that are *easy* to recognize, a small low-capacity DNN model is sufficient while a large high-capacity model is overkill; on the other hand, for video frames with contents that are *hard* to recognize, it is necessary to employ large high-capacity 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 [34].

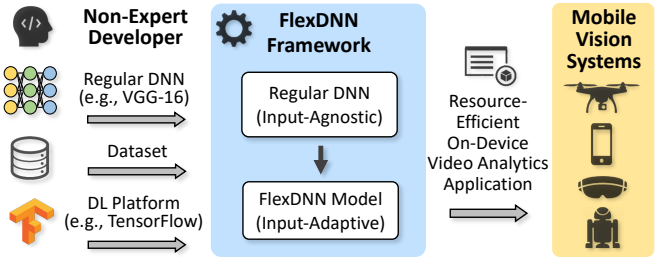


Fig. 1. A high-level view of FlexDNN framework.

B. State-of-the-Arts & Their Limitations

Based on this observation, *input-adaptive* video analytics systems such as Chameleon [15] have recently emerged. Leveraging the easy/hard dynamics of video contents, these systems effectively reduce the computation cost of DNN-based processing pipelines by dynamically changing the DNN models to adapt to the difficulty levels of the video frames. Unfortunately, as we demonstrate in §II-B, this *bag-of-model* approach is a misfit 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 to have *multiple* independent model variants with various capacities to adapt to different difficulty levels of video frames. To address this limitation, BranchyNet [33] introduces the idea of constructing a *single* model with early exit branches inserted at the outputs of convolutional layers of a regular DNN model for input adaptation. Figure 2 provides a conceptual illustration of the early exit mechanism. For an easy frame, it exits at the early exit inserted at an earlier location

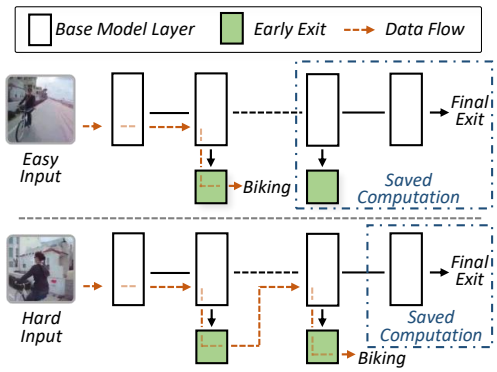


Fig. 2. Conceptual illustration of the early exit mechanism.

since the extracted features are good enough to correctly recognize its content. For a hard frame, it proceeds deeper until the extracted features are good enough. With such early exit mechanism, easy frames do not need to go through all the layers and their computation cost is thus reduced.

While the early exit mechanism is promising, the pioneer work has two key drawbacks:

- First, 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 the early exit mechanism. Unfortunately, the early exit architecture of prior work is designed based on heuristics without focusing on the trade-off between early exit rates and the incurred overheads. Without accounting for such trade-off, the incurred overheads could considerably diminish the benefit brought by early exits.
- Second, the *number* and *locations* of the inserted early exits in prior work are also determined based on heuristics. While effective in comparison against models without early exits, considering the exponential combinations of number and locations of early exits, even for developers with expertise, without considerable efforts on trial and error, it would be extremely challenging to derive an early exit insertion plan that can fully leverage the 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.

C. Proposed Approach

In this paper, we present an input-adaptive DNN-based on-device video analytics framework named FlexDNN that effectively addresses the aforementioned drawbacks. Similar to BranchyNet, instead of carrying a bag of model variants, FlexDNN leverages the early exit mechanism to construct a single model on top of a regular DNN model (i.e., base model). As such, it replaces the fixed capacity of the regular DNN model with flexible capacities. Since only one model is involved, FlexDNN has a compact memory footprint and incurs no model selection and model switching overhead. At runtime, FlexDNN is input-adaptive: it dynamically adapts its model capacity to matching the difficulty levels of the input video frames at the granularity of each frame in real-time.

	Mode	Single Model	Early Exit Architecture	Early Exit Insertion Plan
Chameleon [15]	Server	No	-	-
MCDNN [10]	Cloud-Assisted	No	-	-
BranchyNet [33]	On-Device	Yes	Heuristics	Heuristics
FlexDNN	**On-Device**	**Yes**	**Optimized**	**Optimized**

TABLE I
COMPARISON BETWEEN FLEXDNN AND EXISTING INPUT-ADAPTIVE VIDEO ANALYTICS FRAMEWORKS.

To address the drawbacks of BranchyNet, FlexDNN adopts an architecture search based scheme that is able to find the optimal architecture for each early exit branch that balances the trade-off between early exit rate and its computational overhead. Moreover, FlexDNN is able to derive an early exit insertion plan which identifies the optimal number and locations of early exits to be inserted to maximize the benefit brought by input adaptation. As such, FlexDNN allows developers with limited deep learning expertise to build efficient DNN-based on-device video analytics applications with minimum effort. As illustrated in Figure 1, given a dataset that can be trained, validated, and tested on, FlexDNN automatically transforms a regular DNN model (e.g., VGG-16) that is input-agnostic into an input-adaptive DNN model for efficient on-device video analytics.

Table I provides a comparison between FlexDNN and existing input-adaptive video analytics frameworks. FlexDNN differs from Chameleon and MCDNN as it is an on-device solution that does not require any assist from the cloud. Compared to BranchyNet, the combination of the optimized early exit architecture and the early exit insertion plan makes FlexDNN a superior on-device solution.

We implemented FlexDNN in TensorFlow [4]. To evaluate its performance, we use it to build three representative on-device video analytics applications: Activity Recognition, Scene Understanding, and Traffic Surveillance and deploy these applications on both mobile CPU and mobile GPU platforms. Our results show that:

- The generated FlexDNN model has a compact memory footprint and is able to achieve high early exit rate without loss of accuracy. With the computation-efficient early exit design, FlexDNN maximally preserves the high computation reduction benefit brought by the adaptation.
- At runtime, FlexDNN significantly outperforms both input-agnostic approach and BranchyNet: Compared to the input-agnostic approach, FlexDNN achieves as much as 7% accuracy gain with similar amount of resources, or achieves the same accuracy with as little as 23.4% of the computational resources (i.e., 4.3× speedup in processing time) and up to 4.2× energy consumption reduction. Compared to BranchyNet, FlexDNN reduces up to 49.8% computational cost and up to 1.9× energy consumption.

II. Background and Motivation

We begin with illustrating the intrinsic dynamics of videos taken in real-world mobile settings and demonstrate that considerable resource demand can be reduced by leveraging this dynamic characteristic (§II-A). We then show that the bag-of-model approach incurs significant overhead that considerably dwarfs the benefit brought by the adaptation (§II-B), which motivates the design of FlexDNN.

A. 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 3 shows four frames of a video clip of biking captured using a mobile camera in the human activity video dataset UCF-101 [30]. 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).

Fig. 3. 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.

The intrinsic dynamics of video contents creates an opportunity to reduce computation cost by matching the capacity of the DNN model to the difficulty level of each video frame. To quantify how much computation cost can be reduced, we first profile the minimum computation cost 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 VGG-16 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 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 3. In addition, the difference between "areas" under the two curves reflects the

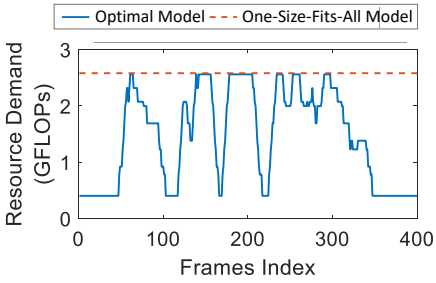


Fig. 4. Blue solid curve: minimum computation cost to correctly recognize the content in each frame (optimal model). Red dotted curve: computation cost of the one-size-fits-all model. (unit: GFLOPs)

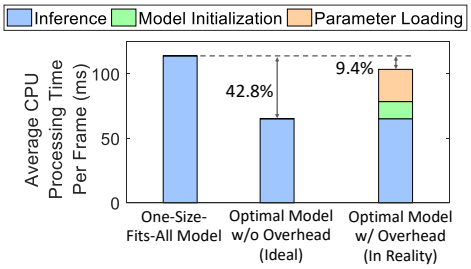


Fig. 5. Benefit brought by the adaptation vs. model switching overhead of the bag-of-model approach.

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.

B. Why Bag-of-Model Approach is a Misfit

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 [15] use multiple model variants interchangeably to achieve content adaptation. While effective as a solution for resourceful systems, this multi-model-variant approach is a *mismatch* to resource-constrained mobile platforms for the following two reasons.

First, the bag-of-model approach requires all the model variants with various capacities to be installed in the mobile system. This, unfortunately, is not a scalable solution and could lead to large memory footprint. For the example used in Figure 4, the total memory footprint of 10 model variants is 513 MB, and the memory footprint would only increase if the number of model variants increases.

Second, the bag-of-model approach 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, model switching could occur very frequently (106 times in 400 frames) because the contents of videos captured by mobile cameras can change drastically in a short period of time. To quantify model

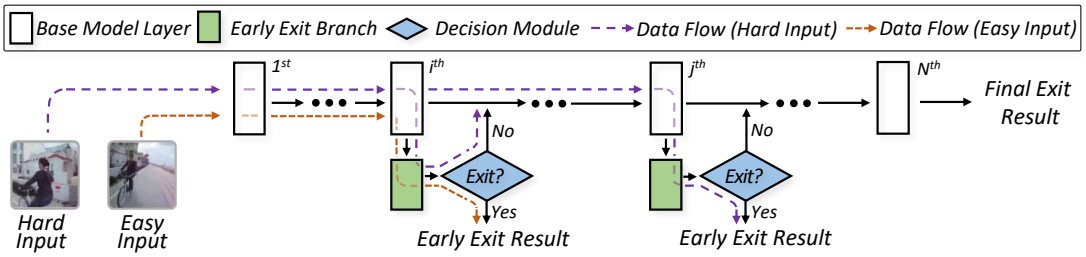


Fig. 6. A closer look at the FlexDNN architecture.

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 the optimal model and one-size-fits-all model.

Figure 5 shows our profiling result. Specifically, the average inference time per frame of the one-size-fits-all model and the optimal model is 114.0 ms and 65.2 ms, respectively. The difference between them measures the benefit brought by the adaptation: by using the optimal model instead of the one-size-fits-all model on each frame, we are able to reduce 42.8% of the computation. Unfortunately, the average processing time of model initialization and parameter loading is 13.3 ms and 25.0 ms, which increases 21.8% and 11.6% of the resource demand, respectively. As a result, the model switching overhead drops the *actual* computation reduction to only 9.4%, which significantly cuts the benefit brought by the adaptation.

III. FlexDNN Design

A. Overview and Design Challenges

Overview. In this work, we present, FlexDNN, an input-adaptive DNN-based framework for efficient on-device video analytics. At the high level, similar to the bag-of-model approach, FlexDNN leverages the intrinsic dynamics of video contents, and is capable of dynamically changing the model complexity to adapt to the difficulty levels of the input video frames to reduce computational demand. However, FlexDNN differs on how the adaptation is realized: instead of carrying a bag of model variants with different capacities, FlexDNN leverages the layered architecture of DNNs, and constructs a single model on top of a regular DNN model with early exits inserted throughout the layered architecture.

A typical DNN consists of various types of layers, including convolutional layers, activation layers, pooling layers, and fully-connected layers. Among them, convolutional layers play the role of "feature extractors", which generate sets of features organized in the form of feature maps. These convolutional layers are sequentially connected such that the set of features generated from the early layer is the input of the next convolutional layer, which generates a new set of features at the next level.

The complexity of a DNN is dependent on its depth. A more complex DNN contains more convolutional layers. However, as we have shown in Figure 2, not all the inputs necessarily need to go through every convolutional layer. By inserting early exits at different convolutional layers of a given regular DNN model (we refer to it as the base model), FlexDNN provides a mechanism to let inputs exit as soon as the features become good enough to them. In doing so, this single-model architecture becomes input-adaptive, and is able to dynamically adapt its model complexity to the difficulty levels of input video frames at the granularity of each frame.

Design Challenges. Figure 6 provides a closer look at the FlexDNN architecture. As shown, FlexDNN is built on top of a base model with the addition of early exits inserted throughout the base model. For each early exit, it consists of two components – *early exit branch* and *decision module* – that are cascaded together. The *early exit branch* is essentially a small-size neural network. Like a regular DNN, it also contains convolutional, activation, pooling, and fully-connected layers, but with small sizes. It takes the intermediate features generated by the internal convolutional layers of the base model and transforms them into early predictions. The *decision module* takes the early prediction results generated by the early exit branch and makes decision on whether to exit the inference process and output the early prediction results or to continue the inference process and pass the generated feature maps to the next layer.

While such early exit mechanism is promising, it also introduces two key challenges:

- First, the inserted early exits themselves have overheads. Take Figure 6 as an example: for the easy frame, the overhead comes from the computation consumed at the early exit branch inserted at the i^{th} layer; for the hard frame, the overhead comes from the computation consumed at the two early exit branches inserted at both i^{th} and j^{th} layers. As such, it is necessary to minimize the computational overheads of the early exits. On the other hand, early exits with extremely lightweight architecture could exit much less frames (i.e., low early exit rate), which makes the early exit mechanism considerably less useful. Therefore, there exists a trade-off between early exit rate and computational overhead in the design space of the early exit architecture.
- Second, inserting early exits at all the convolutional layers of the base model may not be the optimal choice. This is

because for some inserted early exits, their overheads can be actually higher than the benefits they bring. Therefore, identifying the number and locations of the early exits to be inserted that can fully leverage the benefit brought by early exits represents another challenge.

Addressing these challenges is not trivial. Unfortunately, the early exit architecture and the early exit insertion of the prior work are designed based on heuristics. With a careless design, overheads of early exits can overshadow the benefits they bring, and early exit rates can be considerably low unless letting hard frames exit prematurely, which sacrifices the original accuracy of the base model.

In the following, we describe the techniques we develop in FlexDNN that effectively address those challenges followed by a summary of the key characteristics of the input-adaptive model generated by FlexDNN.

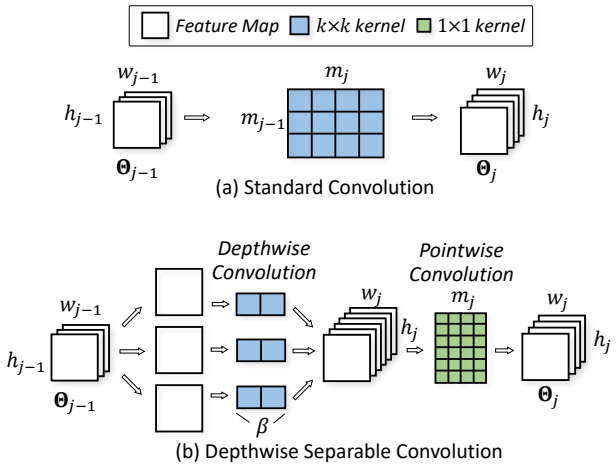


Fig. 7. Structural differences between standard convolutional layer and depthwise separable convolutional layer.

B. Design of Early Exit Branch

The design of early exit branch needs to take the trade-off between computational overhead and early exit rate into consideration. To achieve this, FlexDNN leverages computation-efficient operator as the building block, and employs an architecture search scheme to find the optimal architecture that optimizes the trade-off between early exit rate and computational overhead for each early exit branch.

Among all the types of layers that an early exit branch includes, convolutional layers are the most computation-intensive. To reduce the overhead, we propose to use depthwise separable convolution [6], a computation-efficient convolution operator to replace the standard convolution as the building block for the design of the early exit branch. Figure 7 illustrates the structural differences between standard convolutional layer and depthwise separable convolutional layer. Let $\Theta_{j-1} \in \mathbb{R}^{w_{j-1} \times h_{j-1} \times m_{j-1}}$ and $\Theta_j \in \mathbb{R}^{w_j \times h_j \times m_j}$ denote the input and output feature maps for both types of convolutional layers, respectively. For the standard convolutional layer (Figure 7(a)), it applies m_j 3D filters with size $k \times k \times m_j$ ($k \times k$ is the size of the 2D kernel) onto the input feature maps Θ_{j-1} to generate the output feature maps Θ_j. This process consumes a total of $k^2 w_j h_j m_{j-1} m_j$ floating point operations (FLOPs). In contrast, for the depthwise separable convolutional layer (Figure 7(b)), it adopts the idea of matrix decomposition, and reduces the computational cost by decomposing the standard convolution into two cheap consecutive specialized convolutions: 1) depthwise convolution, and 2) pointwise convolution. The depthwise convolution applies a $k \times k \times \beta$ filter on each of the m_{j-1} input feature maps, where β is the channel multiplier. The pointwise convolution then applies a $1 \times 1 \times m_j$ filter on each channel of the output of the depthwise convolution to generate Θ_j. Therefore, via the decomposition trick, the computational cost of depthwise separable convolutional layer is reduced to $(k^2 \beta w_j h_j m_{j-1} + \beta w_j h_j m_{j-1} m_j)$ FLOPs.

With the depthwise separable convolution as the computation-efficient building block, FlexDNN employs an architecture search scheme to find the optimal architecture that balances the trade-off between early exit rate and computational overhead for each early exit branch. In general, 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 evolutionary algorithms (EA) and stacks cells together to form a network [22], [42], and 2) the *top-down* approach that prunes an over-parameterized network until the optimal network architecture is found [20], [39]. Although both approaches are able to find competitive network architectures, RL and EA are known to be computationally expensive. Therefore, FlexDNN employs the top-down approach due to its much more efficient architecture search process. Specifically, to identify the most efficient architecture of early exit branch, each early exit branch is initialized with three depthwise separable convolutional layers (each followed by one activation layer), two pooling layers, and one fully-connected layer. During the architecture search process, a depthwise separable convolutional layer is removed if no early exit rate drops at that particular early exit branch. This process terminates after the redundant depthwise separable convolutional layers at all the early exit branches are removed.

C. Design of Training Scheme

To train the FlexDNN model that is built on top of the base model with the inserted early exit branches, we need to combine the loss function of the base model with the loss functions of the inserted early exit branches.

In our design, we use cross entropy as the loss for both base model and early exit branches, and design a loss function that is the weighted sum of the loss of the base model and the loss of each individual early exit branch:

$$\mathcal{L} = \mathcal{L}_B + \sum_{i=1}^{N} \alpha_i \mathcal{L}_i \tag{1}$$

where $\mathcal{L}_B$ is the loss of the base model, $\mathcal{L}_i$ is the loss of the i^{th} early exit branch, N is the number of inserted early exit

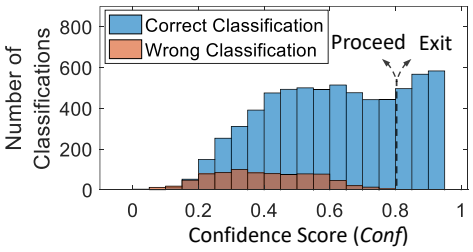


Fig. 8. Determination of the confidence score threshold.

branches, and α_i is the weight of the i^{th} early exit branch, reflecting its importance within the entire model. Empirically, we find that setting all α_i to 1 works well. Based on this loss function, we train the FlexDNN model using stochastic gradient descent.

D. Design of Decision Module

The inserted early exit branches introduce the possibilities of letting hard frames exit prematurely, which sacrifices the original accuracy of the base model. To preserve accuracy, FlexDNN incorporates an entropy-based confidence score, $Conf \in [0, 1]$, in the decision module and uses it to determine whether the input should be exited or be propagated to the following layers for further processing. Specifically, the confidence score is defined as:

$$Conf(\mathbf{y}) = 1 + \frac{1}{\log C} \sum_{c \in C} y_c \log y_c \tag{2}$$

where $\mathbf{y} = [y_1, y_2, ..., y_c, ..., y_C]$ is the softmax classification probability vector generated by the early exit branch, 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.

In essence, $Conf$ measures the confidence level of the early prediction result generated by the early exit branch. The higher the $Conf$ is, the higher the confidence level is. The decision module decides to early exit the input if the value of $Conf$ exceeds a pre-determined threshold.

Figure 8 illustrates how the threshold of the confidence score $Conf$ is determined. Specifically, it shows the confidence score distribution of $7,500$ early prediction results, which are generated by the early exit branch inserted at the first convolutional layer of the VGG-16 base model trained on the UCF-15 dataset (§IV-A). As shown, the confidence score distribution of the correct classification results (marked in blue) overlaps with the confidence score distribution of the incorrect classification results (marked in red) when the value of $Conf$ is in the range of 0.1 to 0.8. This overlapping range is where hard inputs may exit prematurely. In contrast, when the value of $Conf$ is in the range of 0.8 to 0.95, the distributions of the correct and incorrect classification results do *not* overlap, and only correct classification results reside in this range. This is where the early exit branch can make sufficiently certain classification to early exit the input *without sacrificing the original accuracy of the base model*. Therefore, to preserve accuracy, we use the confidence score value at the lower bound of the non-overlapping region as the threshold.

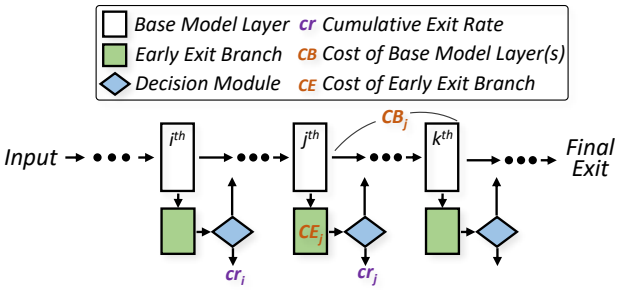


Fig. 9. Illustration of determination on whether to insert an early exit at the j^{th} convolutional layer.

E. Design of Early Exit Insertion Plan

In theory, early exit can be inserted at every convolutional layer of the base model. However, since early exits themselves have overheads, an early exit should *not* be inserted if its incurred overhead is higher than the benefit it brings.

To determine if an early exit should be inserted at the j^{th} convolutional layer of the base model, we define a metric, R_j, as the ratio between the benefit G_j that the early exit brings and the overhead C_j it incurs:

$$R_j = G_j / C_j \tag{3}$$

where C_j is the computation consumed by the 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 9 illustrates how G_j and C_j are calculated. Specifically, it shows three consecutive early exits inserted at the i^{th}, j^{th}, and k^{th} convolutional layer ($i < j < k$). Let N 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} convolutional layer per input video frame. The values of cr_i and cr_j are profiled and determined through cross validation.

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

$$C_j = N * (1 - cr_i) * CE_j \tag{4}$$

There are $N * (cr_j - cr_i)$ 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:

$$G_j = N * (cr_j - cr_i) * CB_j \tag{5}$$

Based on its definition in Eq. (3), if R_j is larger than 1, it indicates that the benefit of inserting an early exit at the j^{th} convolutional layer is larger than the overhead it incurs. Therefore, we start with the trained FlexDNN model with early exit inserted at every convolutional layer, and then remove early exits whose R values are less than or equal to 1 while maintaining those whose R values are larger than 1. In

doing so, we are able to identify the number and locations of the early exits that can fully leverage the benefit brought by the early exit mechanism.

F. Characteristics of FlexDNN Model

With our careful design, the input-adaptive model generated by FlexDNN exhibits a number of key characteristics which we summarize below.

Single Model with Flexible Complexities. With the inserted early exits, FlexDNN is able to provide flexible model complexities within a single model with compact memory footprint. This eliminates the necessity of installing potentially a large number of model variants with different complexities. Moreover, due to the single-model design, FlexDNN avoids model selection and model switching overhead at runtime.

Computation-Efficient Early Exit Branches. With the depth-wise separable convolution operator, the computational cost of the early exit branches incorporated in FlexDNN is significantly reduced. This effectively minimizes the overheads brought by the early exits.

High Early Exit Rate without Accuracy Loss. With the optimized early exit branch architecture inserted at the optimal locations throughout the base model as well as the design of entropy-based confidence score, FlexDNN is able to achieve high early exit rate while blocking hard inputs from exiting prematurely to preserve accuracy.

High Computational Consumption Reduction. Because of the computation-efficient early exit design and the high early exit rate, the generated FlexDNN model is able to preserve the high computation reduction benefit brought by the adaptation.

IV. Evaluation

A. Base Models and Applications

Base Models. We select Inception-V3 [32] and VGG-16 [29] as our base models to evaluate FlexDNN. We select Inception-V3 due to its superior accuracy-computation efficiency ratio compared to other popular DNN models such as MobileNets [12], [27] and ResNet [11]. To demonstrate the generability across DNN models and to investigate the benefit FlexDNN brings to resource-demanding base models, we select VGG-16 as another base model.

On-Device Video Analytics Applications. To demonstrate the generability of FlexDNN across applications, we use FlexDNN to build three representative on-device video analytics applications for three different mobile platforms:

Application#1: 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 build this application, we use UCF-101 human activity dataset [17], which 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). We split training and test videos by following the original paper [30]. Example video frames of UCF-15 are shown in Figure 10 (a).

Application#2: Scene Understanding for Mobile Augmented Reality. Scene understanding is one of the core capabilities of augmented reality. This application aims to recognize places from video streams captured by head-mounted cameras. Due to lack of publicly available datasets, we collected our own video clips in the wild with IRB approval. During data collection, participants were instructed to collect first-person view video footage from diverse places by wearing the ORDRO EP5 head-mounted camera [3]. 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. To avoid model overfitting, we use the same 8 places from Places-365 [41] as our training set, and our self-collected video frames as the test set. Illustration of data collection using the head-mounted camera and one example video frame of Place-8 are shown in Figure 10 (b).

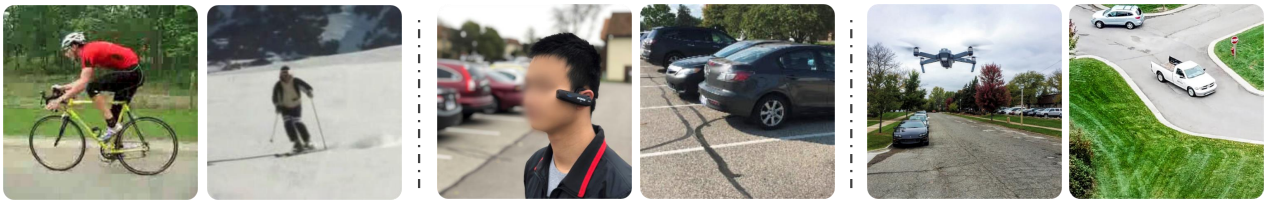
Application#3: Drone-based Traffic Surveillance. Due to its mobility, a traffic surveillance drone is able to track traffic conditions in a large area with a low cost that traditional fixed video camera-based traffic surveillance systems could not provide [18]. This application aims to detect vehicles from video streams captured by drone cameras. Due to lack of publicly available datasets, we use the commercial drone, DJI Mavic Pro [1], 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. We split the dataset into 15% and 85% for training and testing. Illustration of data collection using drone and one example video frame of VeDrone are shown in Figure 10 (c).

For each of the three applications, we use FlexDNN to generate a input-adaptive computation-efficient model from Inception-V3 and VGG-16 for our evaluation. The applications, base models, and models generated by FlexDNN are listed in Table II.

B. Model Performance

In this section, we focus on profiling the models generated by FlexDNN with the goal to quantify the characteristics summarized in §III-F.

High Early Exit Rate without Accuracy Loss. The model generated by FlexDNN is able to achieve high exit rates through its early exits without loss of accuracy. To quantify this characteristic, we profile each of the six models on the test set, and measure the cumulative exit rate at each early exit when the same accuracy is maintained as the base model.



(a) UCF-15 (b) Place-8 (c) VeDrone

Fig. 10. (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) VeDrone: illustration of data collection using a DJI Mavic Pro drone (left); example video frame of traffic surveillance in the residential area (right).

Application	Target Mobile Platform	Dataset	Total Length (min)	Number of Video Clips	Base Model	FlexDNN Model
Activity Recognition	Mobile Phone	UCF-15	240	1,863	Inception-V3, VGG-16	I-UCF, V-UCF
Scene Understanding	AR Headset	Place-8	46	165	Inception-V3, VGG-16	I-Place, V-Place
Traffic Surveillance	Drone	VeDrone	81	40	Inception-V3, VGG-16	I-VeDrone, V-VeDrone

TABLE II
SUMMARY OF THREE APPLICATIONS, TWO BASE MODELS, AND SIX GENERATED FLEXDNN MODELS.

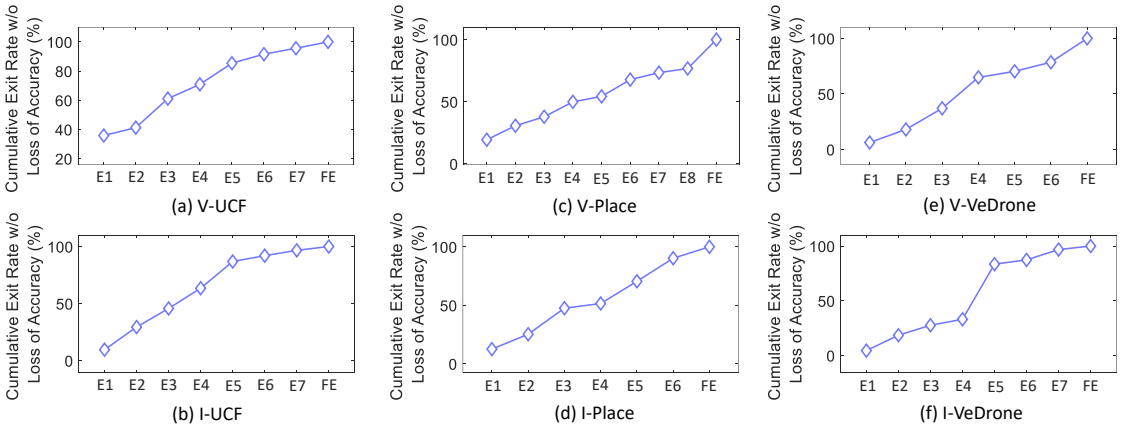


Fig. 11. The cumulative exit rate at each early exit without loss of accuracy. Early exits (marked as E1, E2, ...) are ordered based on their distances to the input layer (i.e., E1 is the earliest exit). FE denotes the regular exit of the base model.

Figure 11 shows the cumulative exit rate at each early exit. 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 95.6%, 96.6%, 76.8%, 90.3%, 78.5%, and 96.8% of the input frames on V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. This result indicates that FlexDNN is effective at identifying efficient early exits while pruning less efficient ones.

Compact Memory Footprint. The model generated by FlexDNN has a compact memory footprint. To quantify this characteristic, 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.

As shown in Figure 12, FlexDNN is able to reduce the memory footprint for 7.8×, 3.2×, 7.9×, 2.8×, 7.8×, and 3.0× on V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. This result demonstrates the considerable benefit of FlexDNN on memory footprint reduction.

Computation-Efficient Early Exits. The early exit branches incorporated in the generated FlexDNN models are computation-efficient. To quantify this characteristic, we compare the accumulated computational cost of all the early exit of the FlexDNN model with the computational cost of its base model.

As shown in Figure 13, the accumulated computational cost of all the early exits of the FlexDNN model is only 1.4%, 1.3%, 1.4%, 0.8%, 1.4%, and 1.2% of its own base model for V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. This result demonstrates that even in the worst case scenario where an input frame goes through *all* the inserted early exits, these exits altogether incur marginal computational overhead compared to the base model.

High Computational Consumption Reduction. The computation saved by early exited input frames reflects benefit brought by the adaptation. The computational cost of the early exit branches reflects the overhead brought by our FlexDNN design. Because of the high early exit rate and the computation-efficient early exit design, the generated

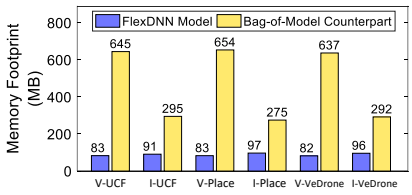


Fig. 12. Memory footprint comparison between the FlexDNN model and its bag-of-model counterpart.

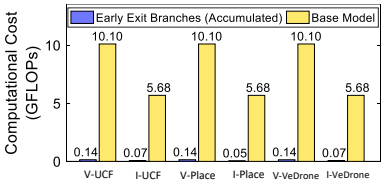


Fig. 13. Comparison between the accumulated computational cost of all the early exit branches and the computational cost of the base model.

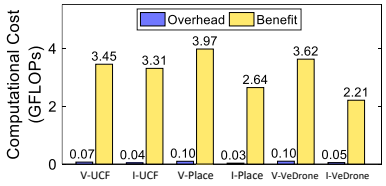


Fig. 14. Comparison between benefit (the average computation saving per input) and overhead (the average computational cost incurred by the early exit branches each input goes through).

FlexDNN model is able to preserve the high computation reduction benefit brought by the adaptation. To quantify this characteristic, we use the average computation saved from early exits per input frame to quantify the benefit, and use the average computation consumed by the early exits each input frame goes through but fails to exit to quantify the overhead.

Figure 14 compares the benefit and overhead of each model. As shown, the overhead is substantially lower than the benefit for each model. Specifically, the benefit-overhead-ratio is $51\times$, $78\times$, $41\times$, $84\times$, $37\times$, and $44\times$ for V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. The benefit-overhead-ratio achieved by FlexDNN is significantly higher than the one achieved by the bag-of-model approach illustrated in Figure 5. As we will show in §IV-C, the achieved high benefit-overhead-ratio is directly translated into various system performance improvement at runtime.

C. Runtime Performance

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

Deployment Platforms. To match the three applications to their target mobile platforms, we deploy V-UCF and I-UCF of the Activity Recognition application on a Samsung Galaxy S8 smartphone and run them on the smartphone CPU; we deploy V-Place and I-Place of the Scene Understanding application as well as V-VeDrone and I-VeDrone of the Traffic Surveillance application on a NVIDIA Jetson Xavier development board [2] and run them on the onboard GPU. We choose NVIDIA Xavier because it is the mobile GPU designed for DNN-based intelligent mobile systems such as AR headsets, drones, and robots.

Baselines. We compare FlexDNN with three baselines: 1) Input-Agnostic-Lossless: this is essentially the base model that the FlexDNN model is built upon. This baseline has the best inference accuracy that FlexDNN model aims to match to but is expensive in computational cost; 2) Input-Agnostic-Lossy: the computational cost of the base model can be reduced via model compression techniques but with a modest loss in accuracy as trade-off. Since it is practically difficult to generate a model variant of the base model that has exactly the same computational cost of the FlexDNN model, we generate a set of model variants with similar computational cost of the FlexDNN model as our second baseline. Note that we do not use the bag-of-model approach used in prior work [10], [15], [40] as baseline because of its drawbacks explained in §II-B; 3) BranchyNet [33]: To make fair comparisons, we use the same base models used in FlexDNN for BranchyNet, and train the BranchyNet to achieve the same inference accuracy as Input-Agnostic-Lossless.

Evaluation Metrics. We use four metrics to evaluate the runtime performance: 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) *frame drop rate*: computational cost translates into the number of frames that can be processed per time unit. A frame is dropped when its estimated processing completion time is over the maximum latency required by the video analytics application. We thus measure the frame drop rate and calculate the frame drop rate reduction percentage; 4) *energy consumption*: computational cost also translates into energy consumption. We thus measure the average energy consumption per frame and calculate the energy consumption reduction percentage.

Inference Accuracy vs. Computational Cost. Figure 15 compares FlexDNN with the baselines in the accuracy-computational cost space. We make three main observations.

First, compared to Input-Agnostic-Lossless, FlexDNN is able to achieve the same inference accuracy with a significant reduction on computational cost. Specifically, FlexDNN only uses 51.3%, 25.6%, 60.5%, 36.2%, 44.2%, and 23.4% computational resources on V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. This is equivalent to $2.0\times$, $3.9\times$, $1.7\times$, $2.8\times$, $2.3\times$, and $4.3\times$ speedup in CPU/GPU processing time.

Second, compared to Input-Agnostic-Lossy, FlexDNN is able to achieve a large gain in inference accuracy with similar computational cost. Specifically, FlexDNN achieves an average 7%, 3%, 6%, 3%, 3%, and 2% gain in inference accuracy on V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. It is important to note that in computer vision community, these accuracy gains are considered to be fairly significant, especially for I-UCF, V-VeDrone, and I-VeDrone which have high absolute accuracies.

FlexDNN Model	IFR (FPS)	MTL (ms)	Platform	Frame Drop Rate		
				FlexDNN	Input-Agnostic -Lossless	BranchyNet
V-UCF	2	2000	S8	10.0%	48.1%	37.2%
I-UCF	10	1000	S8	6.1%	75.1%	28.6%
V-Place	30	50	Xavier	7.4%	50.0%	32.1%
I-Place	30	50	Xavier	3.5%	50.0%	26.7%
V-VeDrone	30	50	Xavier	1.5%	50.0%	21.5%
I-VeDrone	30	50	Xavier	0.0%	50.0%	19.5%

TABLE III
FRAME DROP RATE COMPARISON BETWEEN FLEXDNN (INPUT-ADAPTIVE) AND INPUT-AGNOSTIC-LOSSLESS AND BRANCHYNET.

FlexDNN Model	IFR (FPS)	MTL (ms)	Platform	Energy Consumption (J/frame)		
				FlexDNN	input-Agnostic -Lossless	BranchyNet
V-UCF	2	2000	S8	2.36	4.81	3.19
I-UCF	10	1000	S8	0.51	1.89	0.89
V-Place	30	50	Xavier	0.40	0.63	0.49
I-Place	30	50	Xavier	0.27	0.68	0.41
V-VeDrone	30	50	Xavier	0.32	0.61	0.42
I-VeDrone	30	50	Xavier	0.17	0.71	0.32

TABLE IV
ENERGY CONSUMPTION COMPARISON BETWEEN FLEXDNN (INPUT-ADAPTIVE) AND INPUT-AGNOSTIC-LOSSLESS AND BRANCHYNET.

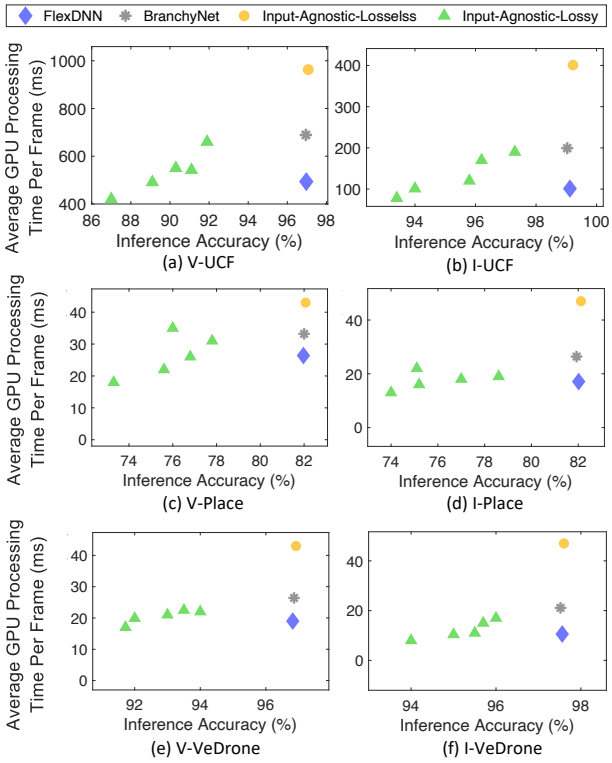


Fig. 15. Comparison between FlexDNN, BranchyNet, Input-Agnostic-Lossless and Input-Agnostic-Lossy in the inference accuracy-computational cost space.

Third, compared to BranchyNet that adopts heuristic early exit design, FlexDNN is able to achieve the same inference accuracy with much lower computational cost. Specifically, compared to BranchyNet, FlexDNN reduces 28.4%, 49.3%, 20.5%, 35.2%, 28.0%, 49.8% computational cost on V-UCF, I-UCF, V-Place, I-Place, V-VeDrone and I-VeDrone respectively, demonstrating its superiority over BranchyNet.

Reduction on Frame Drop Rate. Table III compares the frame drop rate of FlexDNN to Input-Agnostic-Lossless and BranchyNet. In our evaluation, we set up a reasonable video input frame rate (IFR) (in unit of frame per second (FPS)) and maximum tolerable latency (MTL) for the completion of processing a single frame (in unit of millisecond (ms)) for each of the three applications. Since the S8 smartphone CPU is much less powerful than the Xavier mobile GPU, the IFR is lower and the MTL is higher for V-UCF and I-UCF.

As shown, compared to Input-Agnostic-Lossless, FlexDNN is able to reduce the frame drop rate by 38.1%, 69.0%, 42.6%, 46.5%, 48.5%, and 50.0% on V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. Such reduction is achieved because FlexDNN continuously allocates the computation saved by the early exited easy frames to hard frames, whereas the input-agnostic counterpart wastes the unnecessary computation spent on the easy frames. Compared to BranchyNet, FlexDNN is able to reduce the frame drop rate by 27.2%, 22.5%, 24.7%, 23.2%, 20.0%, and 19.5%, respectively. Such reduction is achieved because FlexDNN has more efficient early exit branches inserted at optimized locations.

Reduction on Energy Consumption. Besides reducing frame drop rate, FlexDNN also consumes less energy compared to baselines. Table IV compares the average energy consumption across all the processed frames of FlexDNN to Input-Agnostic-Lossless and BranchyNet. As shown, compared to Input-Agnostic-Lossless, FlexDNN consumes $2.0\times$, $3.7\times$, $1.6\times$, $2.5\times$, $1.9\times$, and $4.2\times$ less energy on V-UCF, I-UCF, V-Place, I-Place, V-VeDrone, and I-VeDrone, respectively. Compared to BranchyNet, FlexDNN consumes $1.4\times$, $1.7\times$, $1.3\times$, $1.5\times$, $1.3\times$, and $1.9\times$ less energy, respectively.

Impact of Workload and Real-Time Constraint. At last, we evaluate the impact of input frame rate (IFR) and maximum tolerable latency (MTL) on the frame drop rate. IFR reflects

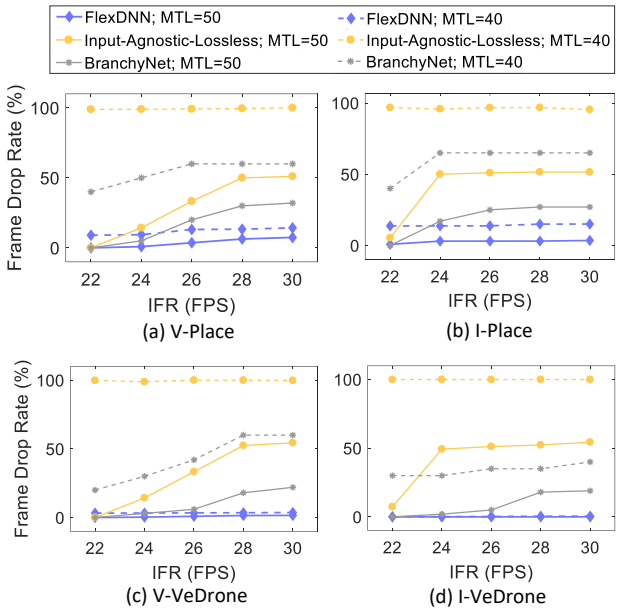


Fig. 16. Impact of workload (IFR) and real-time constraint (MTL) on frame drop rate.

the *workload* allocated to the mobile vision system, and MTL reflects the *real-time constraint* required by the video analytics application. We vary IFR from 22 FPS to 30 FPS to create a wide range of workloads, and we set MTL to 40 ms and 50 ms to create a tight and less tight real-time constraint, respectively. We did not evaluate on mobile CPU in this experiment because of its low IFR.

Figure 16 shows the results. We make two observations. First, when the real-time constraint is less tight (MTL = 50 ms, solid lines), FlexDNN, Input-Agnostic-Lossless and BranchyNet achieve similar frame drop rate when the workload is small. As the workload increases, the frame drop rate of FlexDNN remains low, while the frame drop rate of Input-Agnostic-Lossless and BranchyNet increases significantly. Second, when the real-time constraint is tight (MTL = 40 ms, dotted lines), Input-Agnostic-Lossless consistently fails to catch the deadline and thus drops almost all the frames. For BranchyNet, the frame drop rate is still quite high and it increases as the workload increases. In contrast, the frame drop rate of FlexDNN remains low even when the workload is heavy.

V. Related Work

Continuous Mobile Vision. Our work is closely related to continuous mobile vision [5]. Over the past few years, a great collection of works has been proposed to realize continuous mobile vision [7], [10], [14], [16], [21], [24]–[26], [28], [37]. One key theme of these works is to improve computational efficiency for video analytics applications, which is also the focus of this work. In particular, in [25], the authors proposed a framework named DeepEye that improves resource efficiency of concurrent DNNs by interleaving the execution of computation-heavy convolutional layers and loading of memory-heavy fully-connected layers. In [14], [37], the authors proposed to cache reusable intermediate results between consecutive video frames to improve computation efficiency. In [10], the authors proposed a framework named MCDNN, which uses model compression techniques to generate a catalog of model variants for trading off DNN classification accuracy for resource use. Similar to [10], the authors in [7] also use model compression techniques to reduce computation demand by trading off accuracy, but apply it to generate a single multi-capacity model instead of a catalog of model variants. In contrast, our work provides a complementary approach to achieve computation efficiency. By changing the model complexity dynamically to match the difficulty levels of video frames, our approach is able to reduce computation cost without trading off accuracy.

Our work is particularly inspired by [26], [28]. In [28], the authors proposed a highly efficient continuous mobile vision framework by leveraging the highly skewed class distributions in real-world videos. In [26], the authors proposed a continuous mobile vision system named Glimpse that supports "early discarding" uninteresting video frames. Our work is similar to them in the sense that we all exploit the characteristics in video inputs to reduce the computation cost of video analytics applications. Different from them, FlexDNN exploits the easy/hard dynamics in real-world videos and reduces the computation cost by dynamically adapting its model complexity to the difficulty levels of the input video frames.

Dynamic DNN. Our work is also related to the concept of dynamic DNN in the deep learning literature. Dynamic DNN is a type of DNN that is able to adjust the inference pathway based on inputs. It has attracted a lot of attentions in recent years due to the demand of improving computational efficiency of DNNs [13], [23], [33], [38]. In [23], the authors proposed a dynamic DNN named D^2NN that consists of multiple subnetworks and uses control modules to determine which subnetwork to execute at runtime. However, the concatenation of multiple subnetworks makes D^2NN extremely resource demanding, which is not suitable for mobile systems which have constrained resources. Our work is similar to [33], where the authors proposed BranchyNet, a dynamic DNN that adds early exit branches on top of a regular DNN for fast inference. However, the early exit architecture and the insertion plan of BranchyNet are designed based on heuristics with trial and error. In contrast, FlexDNN effectively addresses these drawbacks with automatically derived optimized early exit architecture and the insertion plan that fully realize the benefit brought by the early exit mechanism.

VI. Conclusion

In this paper, we presented the design, implementation and evaluation of FlexDNN, an input-adaptive deep learning framework for resource-efficient on-device video analytics. FlexDNN allows developers with limited deep learning expertise to build resource-efficient DNN-based video analytics applications that can efficiently run on mobile vision systems

with minimum effort. We evaluated FlexDNN using three representative video analytics applications. Our results show that FlexDNN significantly outperforms status quo input-agnostic approaches and BranchyNet. We believe our work represents a significant step towards turning the envisioned continuous mobile vision into reality.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by NSF Awards CNS-1617627, CNS-1814551, and PFI:BIC-1632051.

REFERENCES

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

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

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

[4] 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*, volume 16, pages 265–283, 2016.

[5] 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*, pages 53–60, 2012.

[6] F. Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, pages 1610–02357, 2017.

[7] B. Fang, X. Zeng, and M. Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *MobiCom*, 2018.

[8] 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.

[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*, pages 1135–1143, 2015.

[10] 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 *MobiSys*, 2016.

[11] 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*, pages 770–778, 2016.

[12] 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.

[13] 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.

[14] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *MobiSys*, 2017.

[15] 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*, pages 253–266. ACM, 2018.

[16] S. Jiang, Z. Ma, X. Zeng, C. Xu, M. Zhang, C. Zhang, and Y. Liu. Scylla: Qoe-aware continuous mobile vision with fpga-based dynamic deep neural network reconfiguration. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1369–1378. IEEE, 2020.

[17] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[18] 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.

[19] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

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

[21] 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*, pages 213–226. ACM, 2015.

[22] 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)*, pages 19–34, 2018.

[23] L. Liu and J. Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. *arXiv preprint arXiv:1701.00299*, 2017.

[24] 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.

[25] 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*, pages 68–81. ACM, 2017.

[26] 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*, pages 292–305. ACM, 2017.

[27] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE CVPR*, pages 4510–4520, 2018.

[28] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *CVPR*, 2017.

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

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

[31] 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*, 105(12):2295–2329, 2017.

[32] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[33] S. Teerapittayanon, B. McDanel, and H. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *ICPR*, pages 2464–2469. IEEE, 2016.

[34] 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*, 108(23):9661–9666, 2011.

[35] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 159–173. IEEE, 2018.

[36] J. Wang, Z. Feng, Z. Chen, S. A. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan. Edge-based live video analytics for drones. *IEEE Internet Computing*, 23(4):27–34, 2019.

[37] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144. ACM, 2018.

[38] T. Yang, S. Zhu, C. Chen, S. Yan, M. Zhang, and A. Willis. Mutualnet: Adaptive convnet via mutual learning from network width and resolution. In *European Conference on Computer Vision (ECCV)*, 2020.

[39] 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 ECCV*, pages 285–300, 2018.

[40] 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*, volume 9, page 1, 2017.

[41] 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.

[42] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *IEEE CVPR*, pages 8697–8710, 2018.