9-2019

Exploiting approximation, caching and specialization to accelerate vision sensing applications

Nguyen Loc HUYNH
Singapore Management University, nlhuynh.2014@phdis.smu.edu.sg

EXPLOITING APPROXIMATION, CACHING AND SPECIALIZATION TO ACCELERATE VISION SENSING APPLICATIONS

by

LOC NGUYEN HUYNH

SINGAPORE MANAGEMENT UNIVERSITY
2019

Exploiting Approximation, Caching and Specialization to Accelerate Vision Sensing Applications

by

Loc Nguyen Huynh

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Dissertation Committee:

Rajesh Krishna BALAN (Supervisor / Chair)
Associate Professor of Information Systems
Singapore Management University

Youngki LEE (Co-supervisor / Co-chair)
Assistant Professor
Seoul National University

Lingxiao JIANG
Associate Professor of Information Systems
Singapore Management University

Kotaro HARA
Assistant Professor of Information Systems
Singapore Management University

Matthai PHILIPOSE
Principal Researcher
Microsoft

Singapore Management University
2019

I hereby declare that this PhD dissertation is my original work
and it has been written by me in its entirety.
I have duly acknowledged all the sources of information
which have been used in this dissertation.

This PhD dissertation has also not been submitted for any degree
in any university previously.

Loc Nguyen Huynh
17 September 2019

Exploiting Approximation, Caching and Specialization to Accelerate Vision Sensing Applications

Loc Nguyen Huynh

Abstract

Over the past few years, deep learning has emerged as state-of-the-art solutions for many challenging computer vision tasks such as face recognition, object detection, etc. Despite of its outstanding performance, deep neural networks (DNNs) are computational intensive, which prevent them to be widely adopted on billions of mobile and embedded devices with scarce resources. To address that limitation, we focus on building systems and optimization algorithms to accelerate those models, making them more computational-efficient.

First, this thesis explores the computational capabilities of different existing processors (or co-processors) on modern mobile devices. It recognizes that by leveraging the mobile Graphics Processing Units (mGPUs), we can reduce the time consumed in the deep learning inference pipeline by an order of magnitude. We further investigated variety of optimizations that work on the mGPUs for more accelerations and built the DeepSense framework to demonstrate their uses.

Second, we also discovered that video streams often contain invariant regions (e.g., background, static objects) across multiple video frames. Processing those regions from frame to frame would waste a lot of computational power. We proposed a convolutional caching technique and built a DeepMon framework that quickly determines the static regions and intelligently skips the computations on those regions during the deep neural network processing pipeline.

The thesis also explores how to make deep learning models more computational-efficient by pruning unnecessary parameters. Many studies have shown that most of the computations occurred within convolutional layers, which are widely used

in convolutional neural networks (CNNs) for many computer vision tasks. We designed a novel D-Pruner algorithm that allows us to score the parameters based on how important they are to the final performance. Parameters with little impacts will be removed for smaller, faster and more computational-efficient models.

Finally, we investigated the feasibility of using multi-exit models (MXNs), which consist many neural networks with shared-layers, as an efficient implementation to accelerate many existing computer vision tasks. We show that applying techniques such as aggregating results cross exits, threshold-based early exiting with MXNs can significantly speed up the inference latency in indexed video querying and face recognition systems.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Vision Sensing Systems

The advances in deep learning research has revolutionized many important fields such as speech recognition, natural language processing, especially computer vision. Cameras, which are currently deployed on personal smartphones or in public and private spaces, have become ubiquitous and contributed an important role in the success of deep learning. By collecting a huge amount of imagery data from cameras, people can use it to train highly accurate deep learning models. Such models enable many new applications like Amazon Go [7] that allows users to experience "grab and go" services at retail stores without any lines and checkouts, or local assistant applications that give guidance advices for individuals who suffer from dementia. However, despite of its success, deep learning still relies on a massive amount of computational power and poses several challenges to many emerging vision sensing systems in terms of efficiency and scalability This thesis describes a set of optimizations to address those challenges in order to accelerate deep learning models that are widely used in many existing vision sensing systems.

Convolutional neural networks, a branch of deep learning, have successfully boosted the performance of many computer vision tasks (e.g., image recognition, object detection) and become the core of many vision sensing systems. For instance,

the error rate of image classification task on ImageNet dataset [22] has been rapidly reduced over 75% since 2012, from 18.2% (AlexNet [52]) to 4.49% (ResNet-152 [36]) in 2015. Such exceptional performance has brought a variety of challenging vision based applications to life such as image search, autonomous self-driving cars, etc.

The explosive growth of mobile and embedded computing has enabled many promising DNN-based vision sensing applications. Smartphones with built-in cameras allow developers to capture users' first person views, analyze the data using state-of-the-art CNN models to understand more about their users. However, outstanding performance from deep learning models comes at the cost of computational complexity. Original AlexNet requires 727 MFLOPS for a single inference while other models such as VGG-16 [76], ResNet-152 [36] require 16 and 11 GFLOPS respectively. Such requirement prevents deep learning models to be widely adopted on mobile and embedded devices with limited amount of computational power.

Despite the challenges, many compression techniques have been proposed to bring down the memory footprint and computational cost of state-of-the-art models by pruning unnecessary parameters [33, 49]. However, randomly removing model parameters results in irregular patterns in the network structures and requires researchers to build the customized hardware to speedup those compression algorithms in the most effective manners [32, 31]. Therefore, billions of existing mobile devices without specialized hardware would not be beneficial to those optimizations. In fact, modern mobile devices consist of many co-processors that computations can be offloaded on. We found that CNN models could be efficiently run on mobile GPUs via the support of OpenCL [80] and Vulkan [6]. Those frameworks allow us to build high performance deep learning framework that can be run directly on variety of mobile devices using existing processors and co-processors. Furthermore, by co-designing compression algorithms with mobile framework, we can make deep learning models to be smaller and faster on commodity mobile devices without any support from specialized hardware.

On the other hand, many works have explored various optimizations at application-level to improve overall performance [17, 45]. For example, one of the widely used techniques is to exploit the idea of reusable computation to avoid unnecessary works without compromising the final results. In video analytic, it is well-known that video stream often contains a lot of similarities between consecutive frames. Therefore, processing every frame would waste a lot of computational resources if the video contains a lot of duplicated data. As more vision sensing applications emerge, domain-specific optimizations would be beneficial to significantly boost up the performance of whole applications.

Finally, many existing works assume every object has the same difficulty [70, 36, 76]. In the case of Yolo object detector [70], detecting a cat is far more accurate than detecting a bottle, which implies that each object might have different difficulty level. Hence, time spending to detect easy objects should be less than time spending on hard objects. To achieve that goal, there should be changes in computational pipeline to intelligently switch between variant of models with different computational cost based on the context of input stream.

We believe the next generation of vision sensing systems would focus on:

- **Platform-specific frameworks:** highly optimized systems, which are built specifically for particular platforms (e.g., smartphones, smart-glasses), to maximize the overall performance,
- **Domain-specific optimizations:** ability to understand application's behaviors and exploit them to build efficient optimizations.
- **Compressed and specialized models:** the move towards using smaller and more computational-efficient models,
- **Efficient computational pipeline:** changes in network architectures and inference pipeline to speed up the computations.

This thesis is a step towards exploring above directions to improve the performance of vision sensing systems.

- We co-designed mobile framework and domain-specific optimization techniques for deep CNN models to enable continuous vision sensing applications on commodity mobile devices.

- We proposed pruning algorithm to intelligently remove redundant parameters within deep learning models to make them smaller and less computational intensive while preserving the regular patterns within network architecture to work on existing frameworks.

- We explored the use of multi-exit models, which consists of multiple models that share a network backbone, as an efficient implementation to accelerate many existing computer vision systems.

1.2 Motivation Scenarios

Elderly assistance: Alex, who is suffering from early stage of Alzheimer dementia, often forget the names of surrounding people around him. In order to help him communicate with others easily, one suggests Alex to use an application called WhoIsThis which proactively detects faces and reminds him names of surrounding people via a smart-glass device. However, Alex is aware that the application keeps sending captured videos to a cloud server to do detection and his privacy maybe severely violated. The development team explains to Alex that in order to provide useful reminders, they have to use cloud infrastructure to run state-of-the-art deep neural network model to recognize faces which is extremely computational intensive. The current model cannot be run locally on the smart-glass within acceptable delay. Alex denies to use the application unless the videos can be processed locally without leaving his device. However, thanks to the rapid advances in mobile processors, the team believes that they can leverage the latest generation of mobile

processor and its co-processors to achieve Alex's needs. In this scenario, the team has to build the system that should be able to

- run entirely on local devices without any supports from the cloud to preserve user's privacy,
- execute any pre-trained deep neural network models using available processors on the local device,
- provide the inference results within reasonable time to assure user's experience.

Video surveillance system: The government is building a traffic surveillance system that has video analytic feature that allows their agents to search for "interesting" events and objects. For example, the system provides search feature for a car accident event and returns all related objects such as cars, pedestrians that appear during the event. However, analyzing whole video stream using existing CNN models such as Yolov2 [71] requires a lot of computational resources (e.g., GPUs) and operational investment. For instance, searching for a single month-long video takes approximately 190 GPU-hour and costs over $380 in the Azure cloud [39]. Deploying such system at city scale would cost a huge amount of money on investment and maintenance. The consultant team observed that most of the traffic videos only contain a few objects comparing to the number of objects in commonly used datasets such as ImageNet [22]. Furthermore, they realize that the surveillance cameras are often set up statically and all the objects are captured from a same viewpoint. However, traditional CNN models are trained on general datasets such as ImageNet or Coco [61] that contain images from different viewpoints and angles. Observing the differences between traditional datasets and static traffic datasets, the consultant team believes that they can leverage those characteristics to build much smaller and faster models and propose new optimizations to achieve the same goals while consuming less computational resources. In the end, the team proposes a video analytic system that should be able to perform:

- construct a learning pipeline to find an computational-efficient model that is specialized to achieve best performance for a particular task,
- minimize the amount of time required to search for that optimal model,
- construct a efficient inference pipeline to minimize the waiting time of users.

1.3 Accelerate Vision Sensing Applications

1.3.1 Mobile deep learning framework for vision sensing

Deep learning has already made huge impact on many computer visions tasks [36, 70, 71, 27, 68]. Many deep learning frameworks (e.g., Tensorflow [8] and Caffe [47]) have been built and carefully optimized to maximize the performance on server-class processors and co-processors (e.g., GPUs). However, we found that those optimizations do not consider the differences between mobile and server co-processors. For example, server-class GPU has a separated high-bandwidth memory while mobile GPU shares the low-bandwidth system memory with the processors. Due to many differences in architectures, those frameworks perform poorly on mobile devices.

In chapter 2, we explore a suit of optimizations that can be used to 1) perform model inference in the fastest way using available mobile GPU, 2) reduce the precision during inference step while maintaining accuracy loss within acceptable level.

1.3.2 Exploiting similarity in video frames for smart caching

Continuous vision sensing applications often need to analyze video frames to gain insights. However, consecutive frames within a video often contain similar or static regions such as background or static objects. Spending resources to process those regions from frame to frame would not only waste a lot of computational power but also decrease efficiency of the systems. Furthermore, many caching systems, which

reuse outputs directly from previous frames [17], often fail to adapt to swift changes in the scene (e.g., sudden appearance of new object). Therefore, we should only skip computation on some regions of the video frame that do not contain changes, not a whole frame.

In chapter 3, we propose an algorithm that quickly determines similar regions between frames and explore the idea of caching intermediate results of those regions within CNN pipeline to reuse them for next frame. By ignoring unnecessary computations on several parts of input frame, we show that overall performance of the systems can be improved.

1.3.3 Exploiting model approximation and compression for fast inference

Approximation is a common technique that widely used to make deep learning models more computational efficient with little accuracy drop. In section 3.4.4, we explore the use of Tucker-2 decomposition to approximate various CNN models in order to reduce the computation cost.

However, approximation often requires huge effort in finding a good trade-off between latency speedup and accuracy drop. In chapter 4, we propose a compression technique *D-Pruner* that automatically learn which parameters are redundant during the training process. By attaching a simple block into the CNN models during training, the attached block learns the importance of each filters within CNN models and gives guidance to remove unnecessary filters while preserving the original accuracy.

1.3.4 Exploiting multi-exit models for efficient computational pipeline:

Most of widely used network architectures [70, 36, 76] use the same inference pipeline for every input. However, as suggested in [41], computational cost for easy

inputs should be less than for harder ones. Hence, it's more efficient to have adaptive computational pipeline whose computational cost can be scaled up depending on the application context.

In chapter 5, we exploit the idea of using multi-exit models as an efficient implementation of DNNs to accelerate existing computer vision tasks. By having multiple models that can share some layers with each other, we can exploit the shared-layers structure and design many optimization to speedup the inference pipeline in many existing vision sensing workloads.

1.4 Key Challenges

In this section, we quickly summarize the key challenges to address in this thesis.

Lack of supported frameworks on mobile devices: Desktop-class and server-class deep learning frameworks often leverage highly optimized linear algebra libraries for GPUs such as CuDNN [18], Viennacl [73], ClBlast [66]. However, due to the differences in architectures, those libraries are either not supported or not optimized to run commodity mobile devices. In section 2.3 and 2.5, we study the computational bottleneck of many CNN models and carefully design optimizations to parallel the executions on mobile GPUs using OpenCL framework.

Computational intensive models: Many state-of-the-art models such as VGG-16 [76], ResNet-152 [36] often require huge amount of computational capacity. For example, VGG-16 and ResNet-152 require 16 and 11 GFlops for a single inference. Naively running those models using un-optimized frameworks on mobile devices would take up to multiple seconds. Hence, to address the problem, we adopt approximation technique (section 3.4.4) and propose compression algorithm (4.3) to remove redundant parameters and make those models more computational-efficient.

Lack of efficient frame similarity measurement algorithm : It is well-known that video stream often contains similarity between consecutive frames. However, measuring the similar regions between two frames quickly is a challenging task,

especially on mobile devices. In section 3.4.3, we observe that color distribution between static regions we propose a fast and efficient algorithm that divide each frame into several blocks and measures differences between two blocks across two consecutive frames using histogram-based technique. We incorporate proposed algorithm with our novel convolutional caching and show the improvement in session 3.5.9.

Un-optimized models for special tasks: Some applications only interest in a small set of objects. For example, surveillance application using traffic cameras only need to recognize vehicles and pedestrians rather than detect boats or airplanes. However, most of existing models, which are trained on general datasets [22, 61, 26] that contain variety of different objects, are not optimized for those specific applications. Instead of using existing models, we could build specialized models that are targeted for those applications to bring out the best performance. Those specialized models can be much smaller in terms of size and computational cost comparing to general models but still provide similar or more accurate results.

Inefficient computational pipeline: [41] has suggested that easy objects should consume less computation resources than hard objects. However, existing models use the same inference pipeline for every input and waste a lot of resources on detecting easy inputs. Therefore, a change in network architecture should be made to allow early exiting for easy inputs. By attaching multiple early classification layers along the network backbone and treat them as multi-exit models, we can stop the execution at anytime whenever the model provides confident result. However, deciding what degree of confidence to stop execution is a challenging task. In chapter 5, we intend to use agreements between multiple classification layers as a voting mechanism to flexibly stop the execution.

1.5 Thesis Statement

This thesis shows that it is possible to significantly reduce the memory usage and latency of deep learning pipelines, with minimal loss of accuracy, by utilizing novel system optimizations such as 1) a smart caching algorithm that reuses feature data between multiple video frames, 2) a pruning technique that removes redundant filters of existing models and, 3) an efficient implementation of shared computations across many models by exploiting multi-exit models.

The thesis is established through the following steps:

- First, it recognizes the inefficiency of existing server-class deep learning frameworks on mobile devices, identifies the performance bottleneck of various deep learning models and proposes an optimized mobile framework that can run many existing deep learning models directly on mobile GPUs.

- Second, it proposes a novel convolutional caching technique that exploits the similarity between consecutive frames in video stream and the internal processing structure of convolutional layers to reuse the intermediate results without re-executing them for every frame. It also presents a fast histogram-based algorithm to quickly detect similar regions between two consecutive frames.

- Third, it addresses the computational complexity problem of existing models by automatically learning the importance of each filter within CNN models. It then proposes a novel compression algorithm, called *D-Pruner*, to iteratively remove unnecessary filters during training steps while maintaining original accuracy.

- Finally, it studies the adoption of multi-exit models (MXNs), which consists of many early classification/detection layers to enable early stopping the com-

putation, to accelerate many existing computer vision workloads. In particular, it explores the degree of agreements across many models within MXNs and proposes a aggregation mechanism to speedup the inference pipeline. Moreover, it also proposes using Focal Loss to enhance the accuracy of early exiting decision to improve the performance of MXNs.

Chapter 2

DeepSense: A GPU-based deep convolutional neural network framework on commodity mobile devices

Recently, a branch of machine learning algorithms called deep learning gained huge attention to boost up accuracy of a variety of sensing applications. However, execution of deep learning algorithm such as convolutional neural network on mobile processor is non-trivial due to intensive computational requirements. In this paper, we present our early design of *DeepSense* - a mobile GPU-based deep convolutional neural network (CNN) framework. For its design, we first explored the differences between server-class and mobile-class GPUs, and studied effectiveness of various optimization strategies such as *branch divergence elimination* and *memory vectorization*. Our results show that *DeepSense* is able to execute a variety of CNN models for image recognition, object detection and face recognition in soft real time with no or marginal accuracy tradeoffs. Experiments also show that our framework is scalable across multiple devices with different GPU architectures (e.g. Adreno and Mali).

2.1 Introduction

A variety of smart glasses are continuously emerging, opening up new opportunities for *continuous vision sensing applications*. For example, *WhoIsThis* application reminds user of names of nearby people in a large conference by recognizing faces from first-person-view video streams. The conventional processing pipeline in these applications is to continuously capture videos or images, extract a set of distinguishing features, and run inference algorithms. Nowadays, various deep learning algorithms such as deep neural network (DNN) or convolutional neural network (CNN) are getting huge attention, as they are known to achieve higher inference accuracy for various vision-based applications [52, 76, 68].

Deep learning algorithms, however, incur heavy computational overhead and power consumption when executing on wearable or mobile devices. A conventional approach to overcome these challenges is offloading computation onto powerful clouds. However, this approach has a few fundamental limitations. First, it has potential threats to expose private data of users. Captured first-person-view images often contain sensitive information such as where they are located, who they are with, which activities they are doing. This may prevent users from offloading data to the clouds, invalidating the use of cloud resources. Second, continuously sending video streams to clouds consumes huge bandwidth which is a big concern when users are connected via cellular networks. Moreover, offloading is no longer effective in scenarios where network connectivity is poor or unavailable.

In this paper, we propose and explore an alternative approach, a *DeepSense* framework, to execute deep learning algorithms on mobile devices without cloud offloading. By leveraging mobile graphical processing unit (GPU) recently integrated into smartphones, we aim to support developers for 1) adopting a wide range of existing DNN, CNN models trained to run on server-class machines with minimal programming effort, 2) achieving real-time or soft real-time latency for continuous sensing and intervention, 3) minimizing energy consumption on the computing mo-

bile devices. Our *DeepSense* framework is built up on OpenCL [80], which is now officially supported by a number of mobile GPUs such as Adreno and Mali.

As a first step towards this direction, our work is focused on supporting CNN/DNN that is widely adopted by various vision sensing applications. We first investigated several existing CNN models (such as AlexNet [52], Vgg-F [15], Vgg-M [15], Vgg-verydeep-16 [76], Vgg-Face [68], etc.), and found out that over 90% of computation occurred within convolutional layers, increasing the processing latency significantly. To reduce the latency, *DeepSense* offloads the convolutional layers to mobile GPUs considering unique characteristics of mobile GPUs as well as the data representation within the CNN structure. Moreover, it adopts various optimization strategies such as *branch divergence elimination* and *memory vectorization* to further reduce latency. Finally, *DeepSense* provides developers the ability to trade off accuracy and latency with the use of half floating points in computation.

We conducted extensive experiments on 3 commodity smartphones (Samsung Galaxy S5, Note 4 and S7) with three 3 CNN models (Vgg-F, Vgg-M and Vgg-16). Our results show that *DeepSense* can achieve soft real-time latency (less than 1.5 second) for various CNN models. With the use of half floating points, *DeepSense* can further reduce latency; for instance, running Vgg-F takes 403ms, 259ms and 155.2ms with only 4.62% accuracy drop on Samsung Galaxy S5, Note 4 and S7 respectively. We believe that more carefully devised optimization techniques and adoption of more powerful GPUs on smartphones would make it feasible to execute large-scale models on mobile devices in real time.

The contribution of our paper can be summarized as follows:

- We proposed *DeepSense* , an OpenCL-based framework to run various deep learning inference algorithms on mobile GPUs; it now supports various CNN models with low latency and power consumption. Note that OpenCL has highly advantageous in that it supports a wide range of commodity mobile GPUs (e.g., Adreno and Mali) comparing to CUDA-based devices (e.g.,

Nvidia Jetson [54]).

- We explored a variety of design choices and optimization techniques to efficiently execute CNN on mobile devices (such as memory vectorization, data representation, usage of half floating points).

- We conducted experiments using various models (AlexNet, Vgg-F, Vgg-M, Vgg-16, Vgg-Face, etc.) on variety of mobile GPU (Adreno 330, 410 and Mali T880). Our preliminary results show that we are able to execute Vgg-F in real-time (803ms on S5, 480ms on Note 4 and 361ms on S7) without any accuracy drop. In addition, with the calculation of half floating point enabled, the execution time of Vgg-F on S5 is reduced to 450ms by sacrificing only 4.62% accuracy.

2.2 Background

We begin with a brief introduction of the two underlying techniques of our system: OpenCL and CNN.

2.2.1 OpenCL

OpenCL [80] is a framework to support parallel programming across heterogeneous platforms such as central processing units (CPUs), graphical processing units (GPUs) or even digital signal processors (DSPs). Recently, OpenCL has been widely supported on both popular smartphone processors (e.g., Snapdragon and Exynos) and popular mobile GPUs(e.g., Adreno and Mali).

In order to use OpenCL for parallel programming, developers first need to divide their problem into a number of small identical sub-problems, then implement each sub-problem as OpenCL kernel code. The OpenCL run-time will spawn multiple parallel processing units (i.e., *work-items*), each runs independent compiled

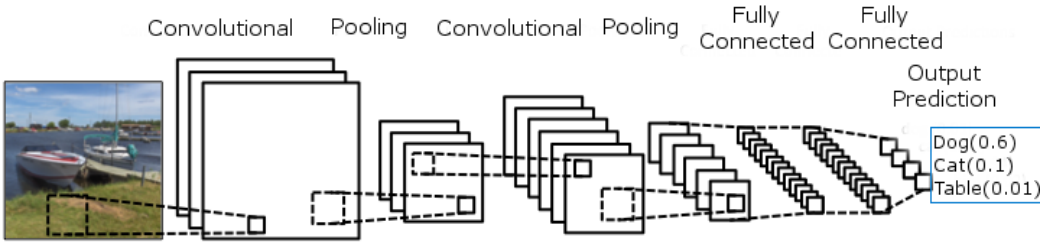


Figure 2.1: Convolutional Neural Network [54]

kernel program and is scheduled to be executed on multi-core CPU, GPU or both depending on the charateristics of application requirements.

Its flexible parallel programming model and applicability on a wide range of mobile processors serve the goal and functionality of *DeepSense*, and thus we adopt OpenCL as our underlying programming and execution framework.

2.2.2 Convolutional Neural Network

Convolutional neural network (CNN) is a type of feed-forward neural network that is widely adopted for image and video recognition [52, 68].

Figure 2.1 shows an example of CNN architecture which consists three fundamental layers: *convolutional*, *pooling* and *fully connected*. To briefly explain, each convolutional layer applies multiple filters to convert lower-level features from the previous layer into higher-level features. A pooling layer is used to capture invariants that do not change even when an image output by a convolutional layer is translated, rotated or scaled. Finally, a fully connected layer aggregates extracted high-level features for further classification task.

As shown in figure 2.2, a CNN layer consists of two main processing steps: *Input Padding* and *Main Computation*. The input padding step is required to match the output of previous layer as an input of current layer. For example, borders of input images can be zero-padded to match the input size of the current layer. Once padding is done, each layer conducts the core computational operations; for convolutional layers, dot products are the key operations. For pooling and fully-

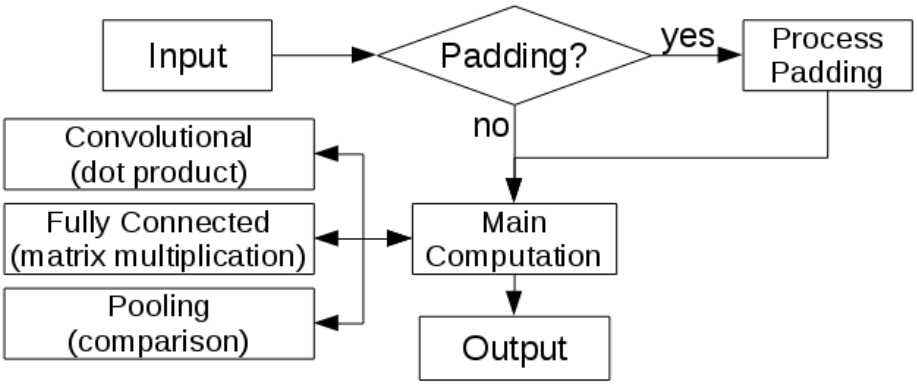


Figure 2.2: Processing Flow of Single Layer

connected layers, comparison and matrix multiplications are the core operations, respectively.

2.3 CNN Performance Breakdown

In this section, we breakdown the performance of CNN in order to identify its bottleneck for optimization. To study the performance of CNN, we use five existing models including AlexNet, VGG models (Vgg-f, Vgg-m, Vgg-verydeep-16, Vgg-Face). Table 2.1 shows the important properties (such as application, accuracy, number of parameters and architecture) of the models. Vgg-Face is trained to recognize human faces (out of 2,622 candidates) within an image while the other models are trained to classify images into one of 1,000 categories. It is noticeable that the accuracy is affected by two factors: (1) the number of convolutional layers, and (2) the size of model (which implies the size of filters and the stride to apply the filter on the input).

To understand the bottleneck, we measure the running time of different CNN layers on Samsung Galaxy Note 4. We implemented a CPU version of a CNN executor in C/C++ using Android NDK. For best CPU performance, we compiled the program with *armeabi-v7a ABI(Application Binary Interface)* to enable external floating point processing unit.

Table 2.2 shows the excution time per types of layers. Most importantly, com-

	App	Size (M)	Top-1 Acc.	Top-5 Acc.	Arch.
AlexNet	IR	60.8	58.2%	80.8%	5c,3p,3fc
Vgg-f	IR	60.8	58.6%	80.9%	5c,3p,3fc
Vgg-m	IR	102.9	63.1%	84.5%	5c,3p,3fc
Vgg-16	IR	138.4	71.7%	90.5%	13c,5p,3fc
Vgg-face	FR	145	98.95%	-	13c,5p,3fc

Application(IR: image recognition, FR: face recognition), Size: number of parameters
Architecture(c: convolutional layers, p: pooling layers, fc: fully connected layers)

Table 2.1: CNN Models

	Conv.(ms)	FC.(ms)	Pooling(ms)	Total(ms)
Vgg-F	8072	1079	26	9177
Vgg-M	19521	2122	156	21800
Vgg-16	213371	2408	882	216662

Table 2.2: CNN Latency Breakdown

putation bottleneck is occurred within convolutional layers for all three inspected models. For instances, over 87% of the processing time in Vgg-F is occupied by the convolutional layer followed by 11% and 0.2% for fully-connected and pooling layers, respectively. For a large model such as Vgg-16, over 98% of computation time is taken in convolutional layers. We also figured out that the total number of addition-multiplication operations within convolutional layers is much higher than operations within fully connected layers (e.g. Vgg-16 requires 15346M addition-multiplication operations for convolutional layers comparing to only 123M for fully connected layers).

2.4 System Overview

Figure 2.3 shows the overall architecture of *DeepSense* which consists of four main components including *model converter, model loader, inference scheduler, executor*.

Model converter: Each of DNN frameworks adopts different representation of its models. This module translates existing pre-trained models from multiple repre-

sentations into our predefined format. At present, *DeepSense* supports 3 formats of DNN trained by Caffe, MatConvNet and Yolo for different types of applications.

Model loader: Application triggers this module to load converted models into memory. It allocates appropriate host(CPU) and device(GPU) memory for individual layer's data structure to store both CNN/DNN's meta-data and weights. Our current implementation of *DeepSense* stores model's meta-data in host memory while all weights of convolutional and fully connected layers are stored in device memory. Other configurations such as enabling half floating point optimization is also processed by this module.

Inference Scheduler: Inference requests are submitted into this module's queue to be scheduled for execution. Since GPU is known to be good at executing SIMD (Single Instruction Multiple Data) task, submitting multiple requests to mobile GPU might interfere each other tasks and increase the latency. In order to prevent interference, this scheduler is built to guarantee that only a single request is executed at a time.

Executor: Execution of inference request is taken place in this module. *Executor* takes allocated model's memory from *model loader*, input data from inference request and compute the output of CNN/DNN. During execution pipeline, only parts of operations such as padding, intermediate memory allocation are executed by CPU while the other heavy computation parts (e.g. convolutional, pooling and fully connected operations) are done by mobile GPU.

2.5 Design Considerations

In this section, first, we investigate behaviours of *branch divergence* and *memory coalescing* on mobile GPU. Second, based on our observations, we propose a memory layout to represent input and parameters in effective manner to achieve low latency on mobile GPU. Finally, we study the impact of half floating point approximation on both the latency and accuracy for different CNN models.

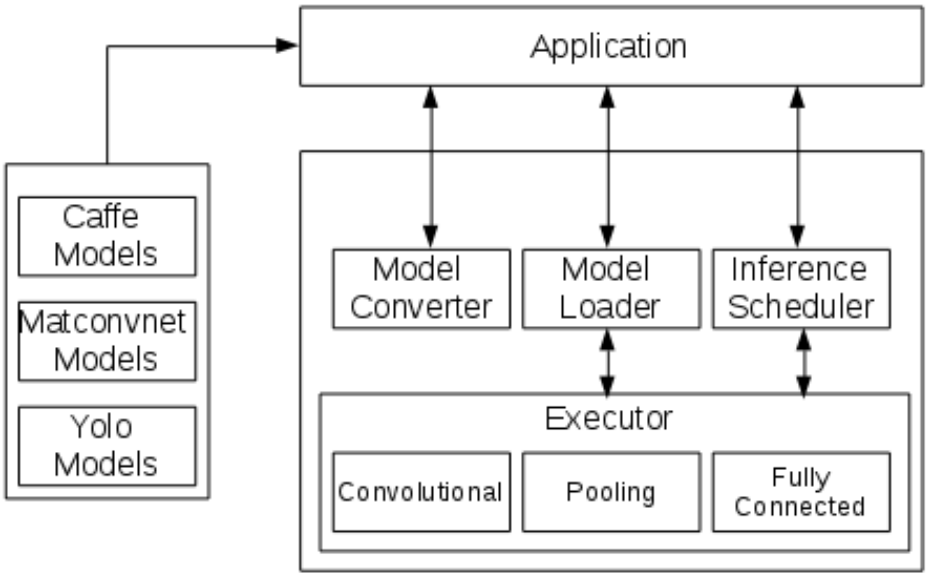


Figure 2.3: *DeepSense* System Overview

We perform evaluations on three different devices including Samsung Galaxy S5, Note 4 and S7 to make our design choices. These devices are powered by two different mobile GPU architectures, Adreno and Mali. Our version of Galaxy S7 integrates Mali T880 GPU while S5 and Note 4 are powered by Adreno 330 and 420 respectively. All platforms support at least OpenCL 1.1 embedded profile.

2.5.1 Branch Divergence

One important issue to improve the latency of CNN execution on GPU is handling *padding* operation efficiently. This operation takes the input and pads data into in order to get desired size. Most of existing CNN models requires padding operations in many layers. Conventional CPU approach to solve this problem is to ignore padded input values when processing. However, this approach imposes condition branches which are inefficient to run on GPU (branch divergence problem). Since *DeepSense* is proposed to executing existing models, this problem should be studied carefully.

Within GPU program, branch divergence is a common problem which causes the GPU to process both conditional blocks of code. This problem increases the

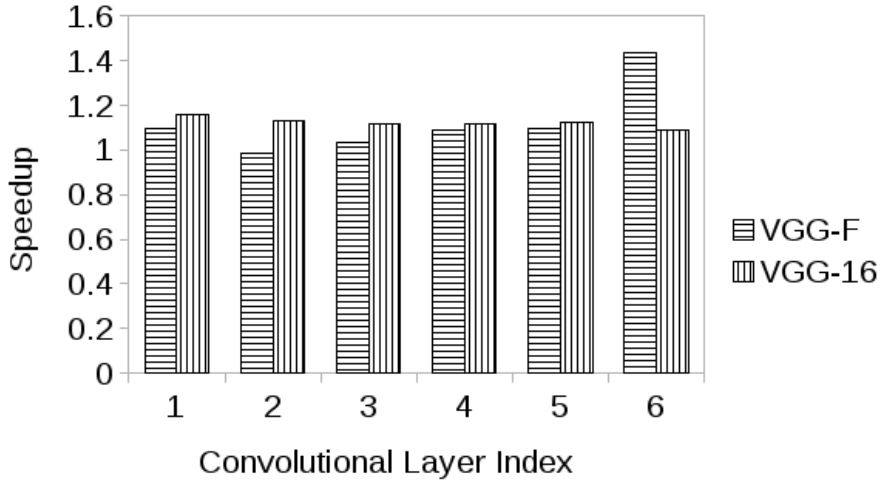


Figure 2.4: Explicit and Implicit padding

execution time of every work item running openCL kernel. However, behaviour of branch divergence when executing CNN on mobile GPU is still not fully evaluated. To address this problem, we consider two types of padding including implicit and explicit padding when executing CNN. The former one processes padding (e.g. ignore padded input using conditional branch) within GPU kernel code and possibly leads to *branch divergence*. On the other hand, the latter approach tries to avoid *branch divergence* by allocating new memory block and migrating corresponding input data into new location before executing GPU kernel. However, overhead occurred by multiple memory copying operations may significantly overwhelm the running time of GPU code.

We carefully evaluate both approaches with two different models (Vgg-16 and Vgg-f) on Samsung Galaxy Note 4 with Adreno 420 GPU. For easy comparison, we use speedup which is defined as a latency fraction between using implicit and explicit approaches.

$$speedup = \frac{runtime_{implicit}}{runtime_{explicit}}$$

Figure 2.4 shows speedup over the first six layers of two models. In most of cases, running explicit padding is faster than executing implicit padding within GPU kernel. We observe that the sixth layer of model Vgg-F has high speedup due to two

Figure 2.5: Memory Coalescing vs Memory Vectorization

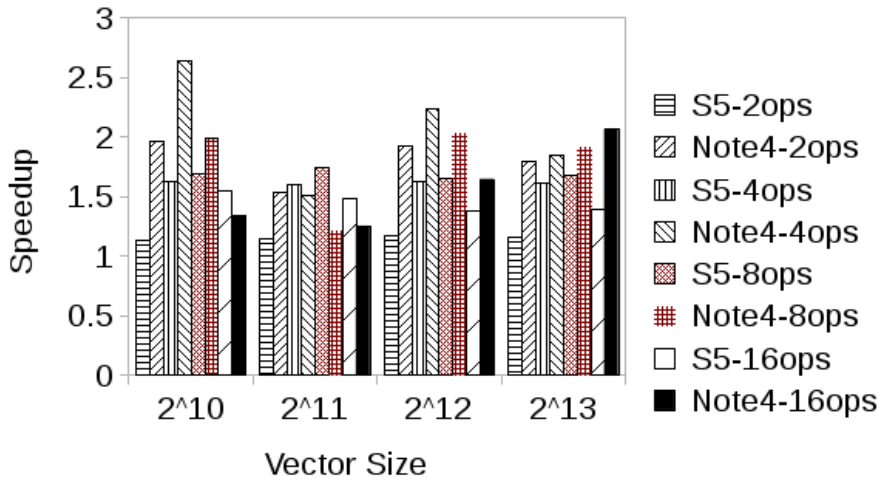
reasons. First, this is one of the bottom layers in VGG-F and the input size of that layer is small so there is little overhead of padding operations. Second, the amount of addition and multiplication operations that needs to be processed is large so the processing latency is overwhelming the padding overhead. Finally, as Vgg-16 consists of more layers than Vgg-F, we also notice the similar characteristic as the processing reaches later layers.

2.5.2 Memory Coalescing vs Memory Vectorization

In this section, we show that correctly reading data into GPU's work items can significantly reduce latency.

One approach is to make used of a well-known technique in GPGPU community called *memory coalescing*. Memory coalescing makes multiple work items to access memory within single transaction. For instances, 4 input values of memory need to be loaded together to fit into memory bank within a single transaction and are distributed across 4 work items to do computation in parallel as shown in figure 2.5a.

We compare memory coalescing with another approach which we call *memory vectorization*. Instead of relying memory architecture to reduce latency as mentioned above, we optimistically read a contiguous memory block into single work item and process it locally. Figure 2.5b shows the example that each of 4 work items read and process a block of 4 input values. As multiple work items access to memory concurrently using OpenCL supported functions, memory bandwidth of the system is utilized.



Each work item computes a fraction of output values (2, 4, 8, 16 and 32 values)

Figure 2.6: Memory Coalescing and Memory Vectorization.

We use vector addition program to evaluate two proposed techniques. To compute each value within output vector, it requires only a single addition operation but accesses to three memory locations (two input and one output locations). This application is best fit for us to measure the memory throughput and latency of two above approaches. Similar to previous evaluation, we define speedup as a latency fraction between using memory coalescing and memory vectorization for comparison.

$$speedup = \frac{runtime_{coalescing}}{runtime_{vectorization}}$$

Figure 2.6 shows the speedup between two techniques. First, we observe that memory vectorization outperforms memory coalescing in all cases. Second, using block size of 4 values results in speedup around 1.7 on S5 and 2.0 on Note 4.

As the result, we organize our data in a way to be loaded as a block of contiguous data into each work item using memory vectorization.

2.5.3 Memory Representation

Representation of data in memory also affect latency of executing CNN.

In OpenCL kernel, parameters are represented as 1D array or 3D image of data. The input and parameters of convolutional layer is 3D and 4D array respectively which have to be reshaped into 1D array or 3D image. However, maximum size of 3D image is also limited by OpenCL framework and the running GPU hardware. To address arbitrary size of parameters and input, all data is reshaped into 1D array. The question is how to represent it in order to achieve best performance.

Suppose we have a convolutional layer with these characteristics:

- Input: size of *[h* x *w]* and *c* channels
- Weight parameters: *n* filters, each filter has size of *[d* x *d]* and *c* channels
- Output: size of *[h'* x *w']* and *n* channels

To compute single output value, CNN does a dot product between a single filter and portion of input data with identical size to filter. This operation requires to read filter's parameters and portion of input into work item. In the end, each work item will trigger a lot of memory reading operations. Reducing number of memory reading operations per work item may result in improving latency. Fortunately, OpenCL provides vload*n*/vstore*n* to allow reading/writing a block of contiguous memory up to 16 float values at a time. Reducing total number of memory reading operations is now corresponding to maximizing the size of contiguous memory block. We try to organize data in CNN in the way that we can maximize the size of single block that each work item has to read into its memory space.

The filter size and input which is used in dot product operation can be represented in 2 ways: *[c* x *d* x *d]* and *[d* x *d* x *c]*. However, since the input to this operation is only a portion of layer's input, its memory is not contiguous. In this case, the former approach can access a block size of maximum *d* contiguous values while the latter approach can access to a block size of maximum *c* contiguous values.

Table 2.3 shows the total number of accesses to contiguous memory block

	Mem Repre.	# blks to access	Max blk size
1	[c x d x d]	c*d blocks of d	11
1	[d x d x c]	d*d blocks of c	512

Table 2.3: Memory Representation and Maximum Number of Memory Accesses per Work Item

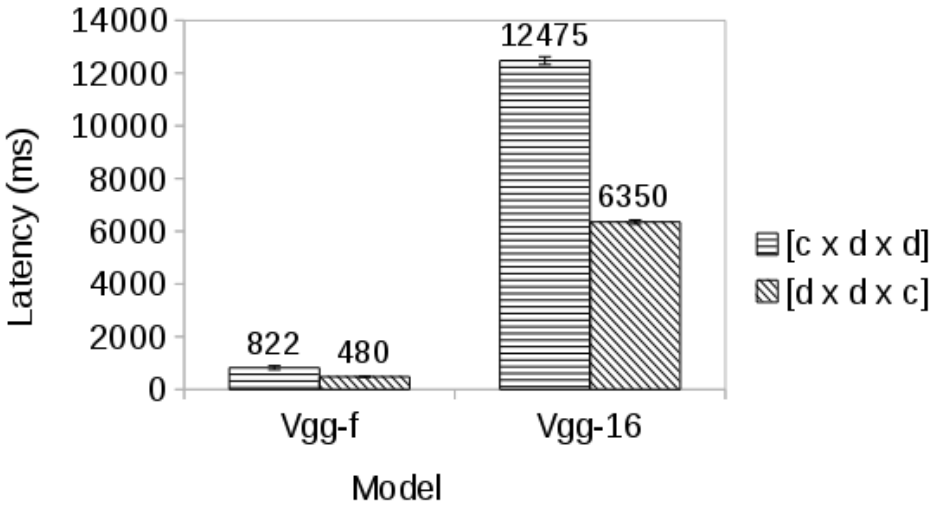


Figure 2.7: Latency of Memory Representations

for each representation. We investigate several models including AlexNet, Vgg-verydeep-16, Vgg-f, Vgg-M and observe that the maximum size of d is much smaller than the maximum size of c. As a result, using *[d* x *d* x *c]* as representation of filter, we can maximize the size of contiguous memory block as well as reduce the number of memory reading operations that needed to be called.

Figure 2.7 shows the latency comparison between using two representation approaches for different CNN models on Samsung Galaxy Note 4. Important observation is that using *[d* x *d* x *c]* approach is more efficient than other approach. For instances, Vgg-16 reduces latency 1.96 times when migrating from using *[c* x *d* x *d]* to *[d* x *d* x *c]* representation.

Finally, input and filters are represented as *[h* x *w* x *c]* and *[n* x *h* x *w* x *c]* corresponding to our design choice.

2.5.4 Half Floating Point

Another optimization technique to improve executing latency on GPU is to minimize transferring data between host memory and GPU. In this section, we propose an algorithm to convert parts of CNN into half floating point to reduce memory bandwidth usage for data transferring in order to improve executing time.

During CNN inference using GPU, a large amount of memory bandwidth and latency is consumed for reading/storing floating point values from/to main memory. To address this problem, we convert parts of CNN's parameter into floating point 16 bits instead of 32 bits. Cutting half of data to load into single work item reduces latency significantly. However, this approach may suffer accuracy drop. To address this problem, we develop a greedy algorithm to choose the most suitable layers to convert parameters into half floating point as follow:

Algorithm 1 Half floating point approximation algorithm

Data: (1)desired accuracy loss *L*, (2)Network *N*, (3)Network accuracy *Acc* , (4)list of convolutional layers *T*, (5) validation set *V*

Result: list of convolutional layers *T'*

```
loss ← 0
T' ← {}
while loss ≤ L and size(T') ≤ size(T) do
    tmp_list ← {}
    for ∀ l in T do
        if l ∈ T' then
            l' ← convert l into FP16
            N' ← N with l ← l'
            tmp_acc_loss ← Acc - N'(V)
            tmp_list ← tmp_list ∪ {l, tmp_acc_loss}
        end
    end
    l, tmp_acc_loss ← argmin(tmp_list[tmp_acc_loss])
    if tmp_acc_loss < L then
        l ← convert l into FP16
        T' ← T' ∪ l
        L ← L - tmp_acc_loss
    end
end
```

We set desired accuracy drop at 5% and run proposed algorithm on three image

Model	CPU-FP32(ms)	GPU-FP32(ms)	GPU-FP16(ms)
Vgg-F	9177	480	259
Vgg-M	21800	1166	558
Vgg-16	216662	6315	2922

Table 2.4: Full and Half Floating Point Latency on Note 4

Model	Top-1 Acc. Drop	Top-5 Acc Drop
Vgg-F	5.82%	4.62%
Vgg-M	3.96%	3%
Vgg-16	2.62%	1.66%

Table 2.5: Half Floating Point Accuracy Drop

recognition models (Vgg-f, Vgg-m and Vgg-16). We use the first 5000 images from ILSVRC2012 validation set [52] to measure accuracy drop since it is also validation set used to evaluate original models.

First, it is surprising that we can convert all convolutional layers into using half floating point for less than 5% of top-5 accuracy drop. Table 2.5 also points out that low accurate model suffers accuracy drop more than high accurate models even though the number of layers and parameters to be converted into half floating point are less than other models.

Second, inference time reduces significantly in our experiments on Samsung Galaxy Note 4 as shown in table 2.4. Converting to half floating point, latency reduces 1.85, 2.08, 2,16 times when executing Vgg-F, Vgg-M, Vgg-16 respectively. That means within convolutional layers, memory bandwidth is highly utilized and needed to be taken into consideration for further improvement.

2.5.5 Performance Overview

We combine several proposed techniques to design *DeepSense* framework. As shown in table 2.4, *DeepSense* significantly reduces inference time up to 74 times comparing to conventional CPU implementation. For small and medium models such as Vgg-F and Vgg-M, *DeepSense* executes one inference within 600ms. For

Model	FP-32(mJ)	FP-16(mJ)
Vgg-F	1135	665
Vgg-M	2584	1487
Vgg-16	14491	8767

Table 2.6: Consumed Energy on Galaxy Note 4

large model such as Vgg-16, *DeepSense* is still able to provide reasonable latency within 3 seconds. Furthermore, energy consumption for single inference request is also shown in table 2.6. From our calculation, continuously executing *DeepSense* for vision sensing with Vgg-F model can last up to 2.5 hours on commodity devices with only modest battery capacity at 2000mAh.

Chapter 3

DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications

The rapid emergence of head-mounted devices such as the Microsoft Holo-lens enables a wide variety of continuous vision applications. Such applications often adopt deep-learning algorithms such as CNN and RNN to extract rich contextual information from the first-person-view video streams. Despite the high accuracy, use of deep learning algorithms in mobile devices raises critical challenges, i.e., high processing latency and power consumption. In this paper, we propose DeepMon, a mobile deep learning inference system to run a variety of deep learning inferences purely on a mobile device in a fast and energy-efficient manner. For this, we designed a suite of optimization techniques to efficiently offload convolutional layers to mobile GPUs and accelerate the processing; note that the convolutional layers are the common performance bottleneck of many deep learning models. Our experimental results show that DeepMon can classify an image over the VGG-VeryDeep-16 deep learning model in 644ms on Samsung Galaxy S7, taking an important step towards continuous vision without imposing any privacy concerns nor networking cost.

3.1 Introduction

The popularity of head-mounted augmented reality (AR) devices such as the Microsoft Hololens [3] and the Google Glass [2] has given rise to a new class of continuous mobile vision applications. These range from identifying road signs in real time to provide directions [17], to identifying people in the environment to give guidance to individuals suffering from dementia [13]. In all these use cases, the commonality is the need to perform computer vision algorithms in real time on a continuous video stream provided by the AR devices.

The current state-of-the-art approach to continuous video processing is to use a deep neural network (DNN) approach where the video streams are processed by a large and well-trained convolutional neural network (CNN) or recurrent neural network (RNN). However, these networks require large amounts of CPU and memory resources to run efficiently. It has thus proved challenging to achieve adequate performance when executing large deep learning networks on commodity mobile devices. For example, a commonly used object recognition model, VGG-Verydeep-16 [76], has 13 convolutional layers and three fully connected layers and takes $\approx$100 seconds to process a single image using CPU on a Samsung Galaxy S7 smartphone.

One way to overcome this limitation is to use cloud resources to run the required DNNs [35]. However, this introduces significant privacy concerns (as the video feed is now available on the cloud server) in addition to possible latency, and energy concerns depending on where the cloud is located and what network interface (LTE etc.) is used.

In this paper, we present a system, called *DeepMon*that uses the graphics processing unit (GPU) on mobile devices to execute the large DNNs required for continuous video processing. *DeepMon* can achieve continuous video processing (at about 1-2 frames per second) of full HD (1080p) video frames using just the memory, CPU, and GPU resources of a commodity smartphone. This speedup allows *DeepMon* to be used, with a larger processing pipe-line where *DeepMon* can ex-

tract features from video frames that can then be processed by cloud resources to produce a complete knowledge. This greatly reduces the privacy impact of using a cloud (as only features and not actual video frames are sent to the cloud) as well as the latency and energy concerns (the feature set is much smaller than the full video image). However, in this paper, we focus solely on the optimisations and techniques to reduce the local processing time from multiple seconds to ≈600ms per frame and leave the integration with a complete cloud-enabled solution to future work.

Before building *DeepMon*, we analysed various deep learning models (e.g., VGG-Verydeep-16 [76] and YOLO [70]) to identify their performance bottlenecks. We noticed that they commonly adopt a large number of convolutional layers (to extract and refine features) along with a small number of fully connected layers (to make inferences). Our measurement showed that the convolutional layer processing takes a significant portion of the entire processing – e.g. 88.7% for VGG-Verydeep-16 and 85% for YOLO (see Section 3.2).

We thus focused on techniques to reduce the processing latency of convolutional layers. One clear solution was to offload the DNN convolutional layer computation to the mobile GPU as these layers have highly parallel and repetitive processing structures. However, prior offloading techniques were developed for server-class GPUs and required re-design/optimization for mobile GPUs with much smaller number of processing cores and memory bandwidth; for instance, NVidia GTX 980 GPU for desktops have 2,048 GPU cores and 224GB/s memory read/write bandwidth while Mali T880 GPU on Samsung Galaxy S7 has 12 GPU cores and 25.6GB/s memory bandwidth.

We developed a suite of optimizations for processing convolutional layers on mobile GPUs. First, we designed a smart caching mechanism specially designed for convolutional layers. The key idea is to exploit the similarity between consecutive frames in first-person-view videos. Our mechanism is unique in that it utilizes the internal processing structure of convolutional layers to reuse the intermediate results of the previous frame to calculate the current frame, instead of just simply

reusing its final output. Second, we decompose the matrices used in the convolutional layers to accelerate the multiplications between high-dimensional matrices, which are the bottleneck when running convolutional layers on GPUs. Also, we applied a number of system-level optimizations (described in Section 3.4 to accelerate the matrix calculation in mobile GPUs).

We implemented *DeepMon* using OpenCL [80] and Vulkan [6] and tested it on various mobile GPUs (Adreno 420, Adreno 430, and Mali T 880) with multiple large DNN models. For developers to adopt various DNN models in *DeepMon*, we also developed a tool that automatically converts pre-trained legacy models and loads them to *DeepMon* with its various optimization strategies applied.

Our results show that *DeepMon* significantly accelerates the processing of large DNNs. For example, the latency of VGG-VeryDeep-16 model-based inference reduces ≈5 times compared to the naive GPU-based implementation with just a marginal reduction in inference accuracy (≈5%). This enables low-latency image classifications (i.e., 3 frames per 2 seconds). Note: VGG-Verydeep-16 is the model many applications such as face recognition (Deep Face from Oxford [68]) and object detection (YOLO [70] and Fast R-CNN [27]) rely on. In addition, we conducted experiments on other models for object detection (such as YOLO) on commodity smartphones (Samsung Galaxy S7, Note 4, etc.). Our results showed that our proposed techniques could achieve a latency of 644ms for VGG-Verydeep-16 and 1,006ms for YOLO on Samsung Galaxy S7.

The contributions of our paper can be summarized as follows:

- To the best of our knowledge, *DeepMon* is the first system to allow large DNNs to run on commodity mobile devices at a low latency. Prior work, such as DeepX [54] and MCDNN [35], has focused on smaller DNNs, cloud computation, and non-commodity more powerful mobile devices such as the Tegra K1.

- We devised a suite of optimization techniques to reduce the processing latency

of the convolutional layers of DNNs. Our smart caching mechanism leverages similarities of consecutive images to cache internally processed data within the deep convolutional neural network. Also, we adopted and improved state-of-the-art matrix multiplication techniques such as model decomposition [49] and unfolding [16] to accelerate multiplication operations (the bottleneck operation in convolutional layers) on mobile GPUs.

- We shared lessons about implementing *DeepMon* on OpenCL and Vulkan and scaling it to support various mobile GPUs. Prior work has focused primarily on CUDA [5] which, to the best of our knowledge, is not supported by commodity smartphones. *DeepMon's* OpenCL implementation can be deployed on a variety of Android-based devices with Snapdragon and Exynos chipsets while its Vulkan implementation (the first such implementation we could find) can be deployed on recent iPhone models such as the iPhone 7. Finally, developers can easily load pre-trained legacy DNN/CNN/RNN models on various mobile GPUs by using *DeepMon's* model converting tool.

- We conducted extensive experiments showing that *DeepMon* can execute very deep models such as VGG-Verydeep-16 on video streams in near real-time, reducing the processing latency to execute one frame from 3 seconds down to 644 ms.

3.2 Deep Learning Pipelines

Vision applications use many deep learning pipelines. We explored the most popularly used models, such as AlexNet, VGG-F, VGG-VeryDeep-16, YOLO, Fast R-CNN (Region-based CNN), to characterize their computational requirements and performance – summary provided in Table 3.1. In this paper, we primarily focus on models (VGG-VeryDeep-16 and YOLO in particular) that adopt more than 15 processing layers to achieve higher accuracy.

	App	Size (M)	Top-1 Acc. (%)	Top-5 Acc. (%)	Arch.
Deep Models					
VGG-VeryDeep-16	IR	138.4	71.7	90.5	13c,5p,3fc
VGG-Face	FR	145	98.95	-	13c,5p,3fc
YOLO	IR	275	63.4	-	24c,4p,2fc
Shallow Models					
AlexNet	IR	60.8	58.2	80.8	5c,3p,3fc
VGG-F	IR	60.8	58.6	80.9	5c,3p,3fc
VGG-M	IR	102.9	63.1	84.5	5c,3p,3fc
LRCN (CNN+LSTM)	AR	62.5	68.2	-	5c,3p,2lrn, 3fc

Application (IR: image recognition, FR: face recognition,
AR: activity recognition),
Size: number of parameters,
Architecture (c: convolutional layers, p: pooling layers,
fc: fully connected layers, lrn: local response normalization)

Table 3.1: Comparison of DNN Models

3.2.1 Background on Various Models

VGG-VeryDeep-16 and VGG-Face. Figure 3.1 shows the detailed processing structure of VGG-VeryDeep-16. The architecture is composed of 13 convolutional layers, 5 pooling layers, and 3 fully-connected layers. Convolutional layers are in charge of extracting various features from an image and refining them while fully connected layers make inferences from extracted features. Pooling layers convert the data from the previous layer to feed to the next input layer. The softmax layer is the final layer to aggregate and normalize the scores generated by the last fully connected layer and outputs the final classification result.

VGG-VeryDeep-16 [76] is used to classify images into one of 1,000 different image types with a confidence probability; it outputs top-N image types with the probability per type. VGG-Face [68], is based on VGG-VeryDeep-16, and performs face recognition. We only use VGG-VeryDeep-16 in our evaluation as VGG-Face has the same structural and algorithmic properties.

YOLO [70] recognizes and locates objects in an image. YOLO can be trained

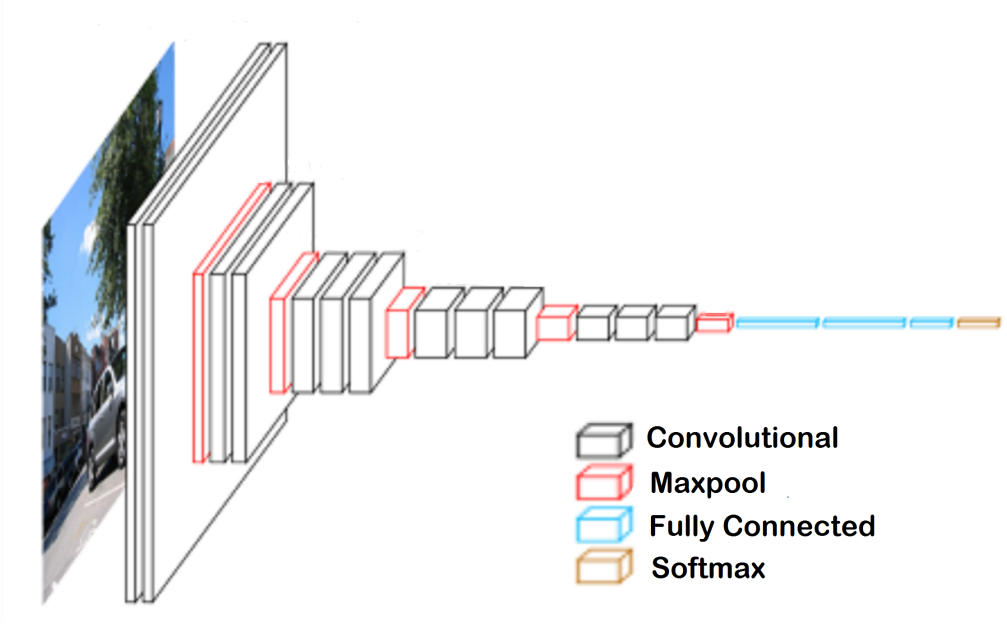


Figure 3.1: Macroarchitecture of VGG-VeryDeep-16 [1]

with different datasets. For example, YOLO trained with the VOC dataset [61] identifies 20 objects and tracks their locations while YOLO trained with the Pascal VOC dataset [26] can identify and localize 80 different objects. The architecture of YOLO is composed of 24 convolutional layers and two fully connected layers, resulting in higher computational requirements compared to VGG-VeryDeep-16 or VGG-Face.

Other Models. There are other smaller-sized but popular models used for image classification, such as VGG-F, VGG-M [15] and AlexNet [52]. Their architecture incorporates a much smaller number of layers; for example, they use just 5 convolutional layers to extract features and 4 fully connected layers for inference. These models are much smaller than VGG-VeryDeep-16 or YOLO with correspondingly lower accuracies given the same train and test data. We omit these shallow models from the rest of the paper as (i) they have already been studied by prior work [54, 55], and (ii) higher accuracy object and face recognition would be more usable for end user applications.

At the other end, some extremely deep models achieve even higher accuracies. For instance, ResNet-152 [36] has 152 layers and achieves 3.62% higher accuracy

compared to VGG-VeryDeep-16. However, we noticed that the accuracy improvements of such models are marginal compared to the models that have 15 to 25 layers while incurring much higher computational costs. We do not expect those extremely deep models can be run on mobile devices in near real-time and thus exclude them from this work.

There are other models such as Faster-RCNN [27] for object detection and *Long-term Recurrent Convolutional Networks* (LRCN [25]) for activity recognition – LRCN is the combination of CNN and *Long Short Term Memory* (LSTM [37]). These models have some common characteristics with VGG-VeryDeep-16 or AlexNet and also modify the structures to achieve better performance and accuracy. Even though they are applied in different scenarios, we noticed that they have lots of commonality with VGG-based models and our workload characterization and optimization techniques apply well to these models.

Effect of the model depth on accuracy and latency. In general, the deeper the model becomes, the higher accuracy it achieves for the same classification task. This increase in accuracy has been validated by recent results [84] (although there are a few special cases where a shallow network is equally accurate). For instance, AlexNet with 5 convolutional layers achieves 80.8% top-5 accuracy to recognize an image while VGG-Verydeep-16 with 13 convolutional layers achieves 90.5% top-5 accuracy. Also, ResNet-152 with 152 layers shows 94.3% top-5 accuracy. On the other hand, deeper models impose much higher computational or memory requirements; For example, the number of operations required to execute VGG-VeryDeep-16 is 21 times more than that of AlexNet while ResNet-152 requires 4 times more memory space than VGG-VeryDeep-16.

3.2.2 Workload Characterization

We noticed important common characteristics in the workloads of deep deep-learning models that drove the optimisations in *DeepMon*. First, each deep model

	Conv. (ms)	FC. (ms)	Pooling (ms)	Total (ms)
VGG-VeryDeep-16	2,647	294	40	2,984
YOLO	3,345	536.1	44.9	3,935
(CNN+LSTM)	5,488.8	161.7	2,158.8	8,301

Table 3.2: Latency Breakdown

has a large number of computational layers – with the accuracy of the model increasing as more layers (convolutional layers in particular) were added. Second, the majority of the layers are convolutional layers. Convolutional layers play a critical role to extract useful features from images and then refine them; in particular, they apply various types of *filters* over the small blocks of an image to abstract out visual features such as edges and shapes. Table 3.1 confirms that, in deep models, the most processing layers are convolutional layers, with a small number of fully connected layers and pooling layers.

Hence, it is likely that most of the processing time is spent in convolutional layers. To check if this was the case, we measured the running time of different deep learning models on a Samsung Galaxy S7 and broke down the processing latency per layer type. To do this, we implemented a GPU-based deep learning execution framework (without any optimization techniques applied).

Table 3.2 shows the execution time broken down per layer type (i.e., convolutional, fully-connected, and pooling). It indicates that the convolutional layers dominate the processing time. For VGG-VeryDeep-16, over 88.7 % of the processing time is occupied by the convolutional layers followed by 9.8% and 1.3% for fully-connected layers and pooling layers, respectively. For the YOLO model, over 85% of computation time is spent in convolutional layers. The reasons for these time breakdowns are (i) there are many more convolutional layers than other layers in deep models, and (ii) the total number of addition and multiplication operations within convolutional layers is much higher compared to fully connected layers and pooling layers (e.g. VGG-VeryDeep-16 requires 15,346M addition and multiplica-

tion operations for convolutional layers while only 123M operations are necessary for fully connected layers). These results suggest that optimizing the processing time of convolutional layers would lead to huge improvements in overall model processing latencies.

3.3 Design Considerations

We developed *DeepMon* with the following design goals:

No cloud offloading: Our primary design goal, for this paper, was to use local phone resources only without any cloud offloading to process deep DNNs as this area has compelling use cases without any viable solutions. There are also scenarios, such as processing of sensitive video feeds or video processing in places with poor or expensive networking connectivity, where offloading is either unwanted (due to privacy concerns) or impossible (due to networking issues). We do plan to extend our solution to support cyber foraging (e.g. MAUI [20] and Chroma [9, 10]), where local and cloud resources are used in a dynamic fashion.

Near real-time latency: Our intended application scenarios require near-real time processing of image streams to give on-the-fly feedback to the users. However, we do not aim to provide strict real-time support (e.g., ¡ 50ms with strict inter-frame timings) as we do not believe this is possible with current commodity smartphones and deep DNNs. Instead, we aim to push the research boundary to provide 1-2 frames per second processing capability (the current state-of-the-art is 1 frame every 3-4 seconds).

Minimal accuracy loss: While achieving near-real-time processing latencies is good, it cannot be done at the cost of accuracy – otherwise improving latency becomes trivially easy. We thus require *DeepMon* to be only about 5% less accurate than running the same model on a desktop PC.

Efficient power use. Minimizing the energy use of *DeepMon* is essential as we aim at running complex deep learning pipelines on mobile devices. In this paper,

Phone	GPU	APIs	# GPU Cores (#ALUs)	Memory Size (GB)	Memory Bandwid. (GB/s)
Samsung S7	Mali T880	OpenCL/ Vulkan	12	4	25.6
Samsung Note 4	Adreno 420	OpenCL	4 (128)	3	12.8
Sony Z5	Adreno 430	OpenCL	4 (192)	3	12.8

Table 3.3: Specs for Commodity Mobile GPUs

we focused on reducing the power consumption of executing deep learning pipeline on a mobile device and rely optimising the power consumption of the video camera (to capture and store continuous video feeds) to prior work [57].

Support a wide range of mobile GPUs and programming APIs: There has been prior work [35, 54] that used external mobile development boards, such as the Tegra K1, to test their solutions. We designed *DeepMon* to work well on commodity smartphones and tested it across a range of mobile phones and programming APIs (the full list of test devices is shown in Table 3.3). In particular, *DeepMon* supports both the OpenCL [80] and Vulkan [6] programming APIs.

3.4 Implementation

In this section, we first show the overall architecture of *DeepMon*, and then describe, in detail, the various techniques we adopted to optimize the execution of deep learning pipelines.

3.4.1 Architecture Overview

DeepMon is built on top of $DeepSense$ framework to leverage the low-level optimizations of $DeepSense$. Figure 3.2 shows the overall architecture of *DeepMon*, and Table 3.4 summarizes our techniques. *DeepMon* works through two different

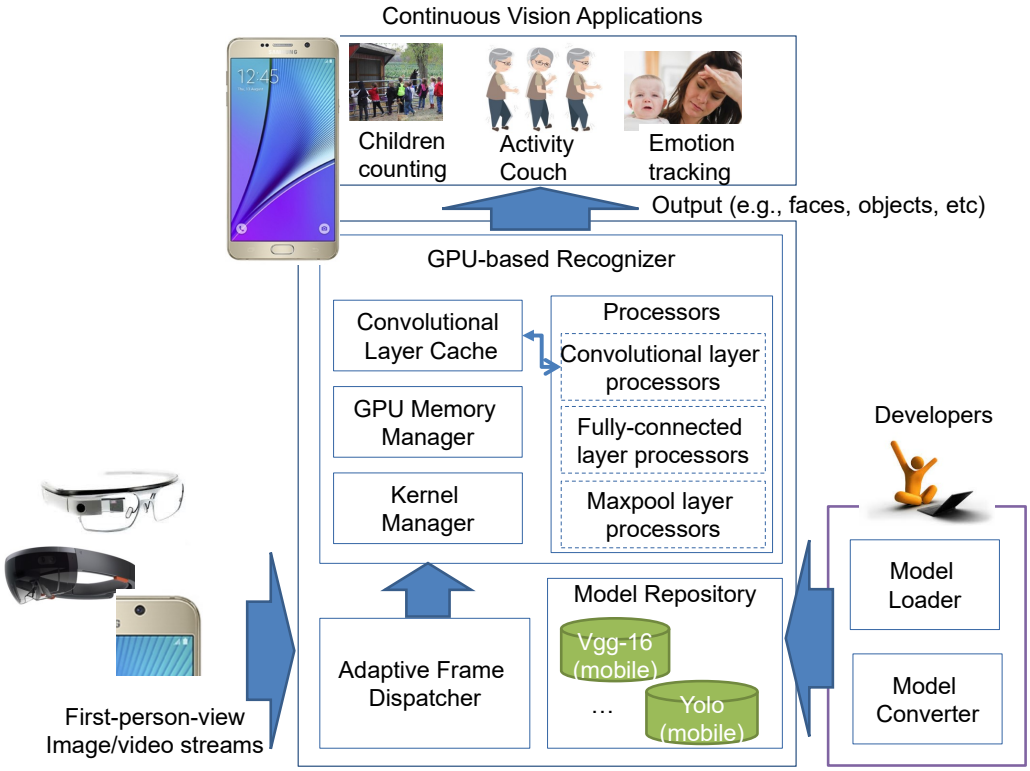


Figure 3.2: *DeepMon* System Architecture

Techniques	Description	Evaluation
Model Conversion/Loading	Section 3.4.2	-
Convolutional Layer Caching	Section 3.4.3	Section 3.5.2–3.5.5& Section 3.5.8–3.5.9
Layer Decomposition	Section 3.4.4	Section 3.5.2–3.5.5
Convolution Optimizations	Section 3.4.5	Section 3.5.2–3.5.5
Scaling to various GPUs/APIs	Section 3.4.6&3.4.2	Section 3.5.6&3.5.7

Table 3.4: Summary of *DeepMon's* techniques

phases: (1) the model conversion phase to convert existing models to run on mobile GPUs, and (2) the inference phase to process image streams using the converted model to recognize useful information.

Model conversion and loading. To use *DeepMon*, developers first need to convert existing deep learning models (built for desktop GPUs) to fit on mobile GPUs. For this, we provide *model converter* and *model loader* tools – the current *DeepMon* prototype can convert a variety of existing models including the ones described in Table 3.1. The model converter adapts the configurations and parameters of an

existing model and generates a new model that can run efficiently on mobile GPUs (See Section 3.4.4). The model loader then loads the generated model on *DeepMon*– it allocates adequate memory spaces to lay out input data for efficient convolution computation and structures the processors for all the layers composing the model (See Section 3.4.2)

DeepMon currently supports the models from three different deep learning frameworks, namely *Caffe* [47], *Matconvnet* [86] and *YO-LO* [70].

Real-time Inference. During the inference phase, *DeepMon* takes a stream of first-person-view images as its input. The *frame dispatcher* selects important frames to recognize and feeds them to the GPU-based recognizer. Then, the *GPU-based recognizer* executes the deep learning pipeline and outputs its inference results to the applications of interest. During the execution, it applies a suite of optimization techniques, such as convolutional layer caching and matrix multiplication optimizations, to boost the recognition speed (explained in detail in Sections 3.4.3 and 3.4.5).

DeepMon supports both OpenCL and Vulkan and was tested on phones with Adreno and Mali GPUs. We present our evaluation results for various GPUs and Vulcan in Section 3.5.6 and 3.5.7.

3.4.2 Loading Models into Mobile GPUs

Figure 3.3 shows the detailed flow of the model conversion and loading process. First, the model convertor decides how to layout the input data into the memory space. The challenge here occurs mainly because the memory space is linear while the input data are multi-dimensional matrices. The wrong unfolding of the multi-dimensional data into a linear space would result in huge fragmentation of the data, which will slow down the convolution processing significantly. Intuitively, the model converter lays out the data such that matrix multiplications can be done by reading consecutive memory blocks and reusing them as much as possible once they are in memory. This is particularly important for devices with low memory

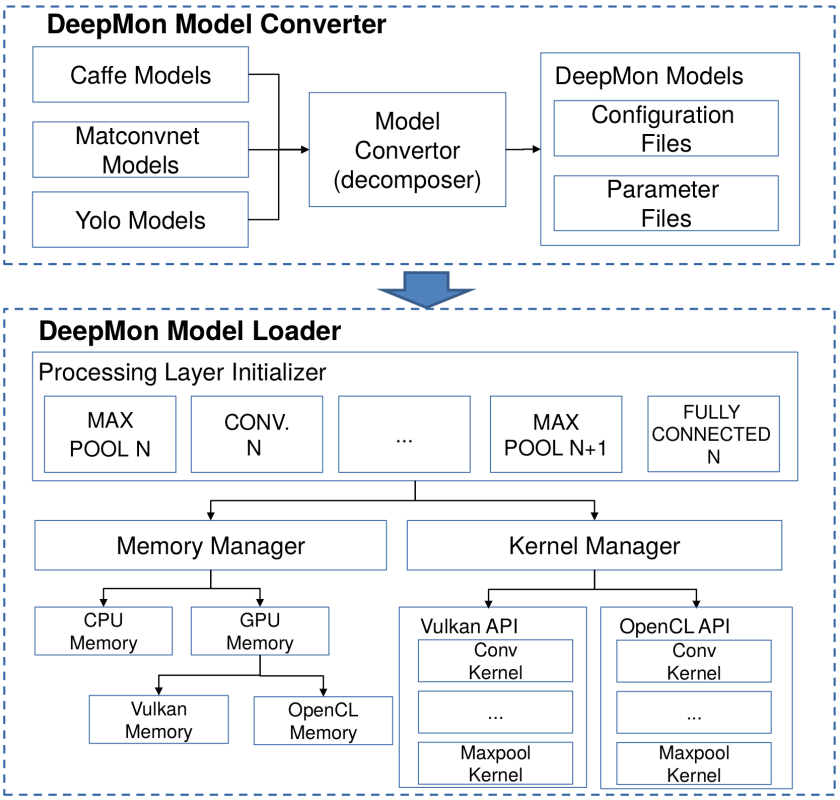


Figure 3.3: The Flow of Model Conversation and Loading

bandwidth (e.g. Samsung Galaxy Note 4 and Sony Xperia Z5).

Once the data layout is decided, the *model loader* initializes all the necessary additional layers (e.g. convolutional, pooling, fully-connected, etc.) within the *DeepMon's* recognizer. During initialization, *DeepMon* performs two important tasks: (a) memory allocation and (b) kernel code compilation.

First, upon layer initialization, *DeepMon* needs to allocate memory spaces to store the metadata (e.g. size of filters, input size, output size, etc.) or parameter values. *DeepMon* stores all the metadata in the host memory (or CPU memory) for easy data access and stores all the parameters in the device memory (or GPU memory). The GPU memory space is allocated based on the API used (OpenCL or Vulkan). The memory space for the actual input and output data is also allocated in the GPU memory for efficient computation. This space can be mapped to the host memory when necessary (e.g. to return final output to application).

Second, a specific *kernel code*, containing the code block to be parallelized by

(a) An image at time t_0 (Left)
(b) An image at time $t_0 + 500ms$ (Middle)
(c) The same image blocks marked as black (Right)

Figure 3.4: Example First-Person-View Images

the layer, needs to be built and loaded. Building these kernel code is handled differently for OpenCL and Vulkan. For OpenCL, a kernel is written in the OpenCL C-like language. It does not require pre-built binary code for any specific device – Instead, it supports compilation capabilities on the target device itself, making it easy to be ported to other OpenCL-enabled devices. Vulkan, on the other hand, uses SPIR-V (Standard Portable Intermediate Representation), an intermediate language for graphics and parallel computation. In Vulkan, SPIR-V code can be loaded onto various Vulkan-enabled devices without building binary code. *DeepMon* prepares two separate convolutional implementations in advance and compiles the kernel code on demand, based on the chosen API, when a layer is initialized and loads the kernel into memory.

3.4.3 Convolutional Layer Caching

As shown earlier (Section 3.2), the convolutional layers are the main performance bottlenecks. To accelerate the computation of these layers, we designed a caching mechanism optimised for convolutional layers. Our key observation is that first-person-view images tend not to change much over a short time duration. For example, Figure 3.4 shows three first-person-view images; the left and middle images were taken at time t_0 and $t_0 + 500ms$ while the rightmost image, taken at time

$t_0 + 500ms$ shows the same image blocks (marked as black).

In particular, the background of images across multiple continuous image frames often remains still while foreground objects tend to move. Such commonality in images incurs heavy repetition in the execution of convolutional layers as applying the full pipeline on one image at a time applies the same convolution computations on many different "repeated" frames and sub-frames.

Our caching mechanism reduces this repetitive computation significantly. A plausible caching approach would be to reuse the final result from the previous frame when the difference between frames is under a certain threshold (Chen et al. [17] proposed a similar idea). However, this approach would not work in many cases as foreground objects (that take a small portion of the entire image but are important to recognize) tend to change noticeably while the background images do not. This makes the previously cached results either stale (on a cache hit) or incurs lots of cache misses. To overcome this, we *cache the partial results of convolutional layers* – i.e., we reuse the convolution outputs for the unchanged blocks of an image while recalculating convolutions for the changed blocks.

3.4.3.1 Caching Mechanism

The overall flow of our caching mechanism is as follows. First, we divide the image into a grid (e.g. an 8x8 grid) where each grid block contains a fixed number of pixels. During the execution, we compare corresponding blocks, $b_{(t-1)}$ and b_t of two consecutive images to determine if the outputs of the previous convolutional layer, $b_{(t-1)}$, are reusable (i.e., it is a cache hit). Upon a cache hit, *DeepMon* skips the convolution computation on the pixels within the entire block. *DeepMon* caches the convolution outputs for the first N convolutional layers only (where N is determined empirically for every model) since the computation for the later convolutional layers are often quite small, and the caching overhead is higher than the benefit. Cached values expire after a certain duration – for example, we set the default expiration times, determined empirically, to 650ms for VGG-VeryDeep-16 and 1000ms for

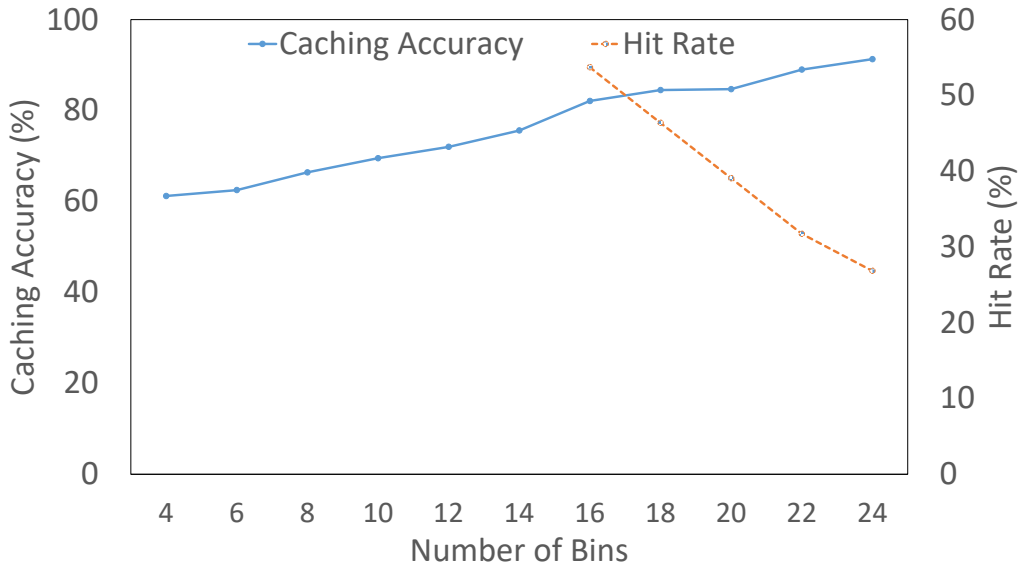


Figure 3.5: Effect of the Number of Bins on Caching

YOLO.

However, the key challenge with this caching scheme is that it is a non-trivial task to determine if the two image blocks are similar or not. Indeed, if the image comparison is too heavyweight, the caching overhead will quickly exceed its benefit. There are a few image comparison algorithms with high comparison accuracy, for example SIFT-based [64] and Hog-based [21] algorithms. However, their computational cost is high and not suitable for our cache design (See Section 3.5.9 for the relevant results.).

To solve this problem, we adopted a light-weight algorithm based on colour histograms. For the two image blocks to compare, we compute the histogram of the colour distribution and compute a *chi-square distance metric*. If the distance is less than a pre-defined threshold, the cell is marked as ”reusable”.

For efficient caching, it is important to choose the right number of bins (to calculate the histogram) and the distance threshold. We carefully chose the right parameters through empirical studies using AlexNet. First, we investigated the effect of the number of bins by fixing the distance parameter to 0.005. Figure 3.5 shows that as the number of bins increases, the accuracy increases while the number of

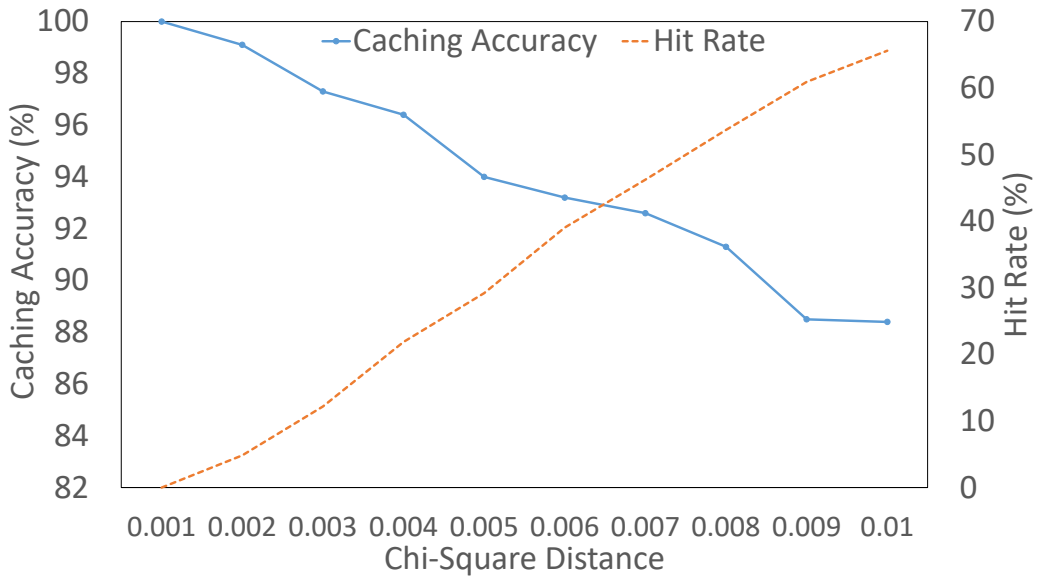
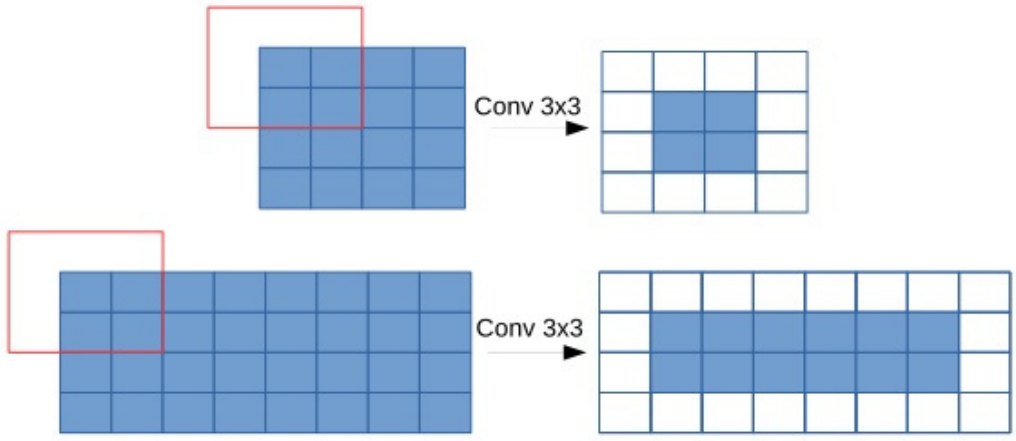


Figure 3.6: Effect of the Distance Values on Caching

"cacheable" blocks decreases; in the figure, the caching accuracy indicates how closely *DeepMon* outputs the final classification results with respect comparing to the original model. We also explored the effect of distance threshold – the number of bins was set to 16. Figure 3.6 shows the trade-off between accuracy and cache hit rates for various distance threshold values. We use the cross-over points of the accuracy and hit rate to decide a plausible number of bins and distance threshold value.

To make caching work efficiently along with our GPU-based recognizer, we carefully re-implemented our GPU-specific kernels. Intuitively, we first initialize all the memory spaces (that need to contain the output of a convolutional layer) with the cached results. Only for those blocks with cache misses, *DeepMon* maps the new outputs into the corresponding memory spaces. This makes updating uncached results easy.

When reusing cached results, we had to be careful about the edges of an image block. Figure 3.7 shows two examples of caching applied on a block size of 4x4 in a convolutional layer with a filter size of 3x3. Figure 3.7(a) shows an example where a convolutional filter is applied to the edge of the cached block. In this case,



(a) Example with a 4x4 Block and a 3x3 Filter (Top)
(b) Example with two 4x4 Consecutive Blocks and a 3x3 Filter (Bottom)

Figure 3.7: Caching on the Edge of an Image Block

the output value becomes non-cachable as the 3x3 block being calculated may refer to non-cachable data (data outside of cached block). For that reason, *DeepMon* will not reuse the results for the edges of the block. However, when two or more consecutive blocks can be cached, as shown in Figure 3.7(b), *DeepMon* reuses the cached results for the edges that are shared by the cacheable consecutive blocks. Importantly, for the models we are considering, the block size is quite large for the first few convolutional layers (e.g. 28x28 pixels for the first layer for VGG-VeryDeep-16), making this caching technique effective for all those layers.

3.4.4 Convolutional Layer Decomposition

We further optimize convolutional layers by decomposing the convolutional parameters. Convolutional layers are well-known to have redundant parameters [46], making them computationally inefficient on resource-constrained devices. Prior research have provided a few different methods (such as the tucker decomposition [50] and CP decomposition [56]) to decompose a convolutional layer into three smaller convolutional layers so that the total computation of the decomposed layers is less than that of the original layer.

DeepMon adopts a variance of the Tucker decomposition named *Tucker-2* [50] over other alternatives since it is a better match to *DeepMon's* caching algorithm. The weights of a convolutional layer are often represented as a tensor *T* of size *[N x C x D x D]* in which N and C are the numbers of input and output channels, respectively, while D is the size of the filters. Tucker-2 decomposes T into three smaller tensors *T1, T2, T2* with the sizes of *[C' x C x 1 x 1]*, *[N' x C' x D x D]*, *[N x N' x 1 x 1]* respectively, where the number of new input and output channels (i.e. N' and C') are reduced compared to those in the original tensor (i.e. N and C). Intuitively, the decomposition reduces the number of dot product operations from *(N x C x D x D)* to *(C' x C) + (N' x C' x D x D) + (N x N')*, enabling *DeepMon* to further reduce the latency.

Tucker-2 decomposition is more appropriate to be used with our caching technique due to its unique characteristic – two of the decomposed layers have the filters with the size of *[1 x 1]*. *[1 x 1]* filters do not reduce the input size to the subsequent layers, keeping the cacheable block size across layers; note that if the block sizes get reduced, the overhead to compute cache hit/miss will increase, compromising the benefit of caching. On the other hand, other decomposition methods use filters larger than *[1 x 1]*, reducing the size of *cachable* blocks and making the caching less effective. Moreover, the *[1 x 1]* filter does not require separate handling of the edges of cacheable blocks (as shown in the Figure 3.7). This enables us to develop a more efficient GPU-kernel to reduce the latency further.

The non-trivial problem, here, is to choose the right *N' and C'*. In practice, manual trial and error is still a common yet inefficient approach that requires a lot of effort. Instead, we devised a *double binary search algorithm* to reduce the amount of effort needed. The key idea behind the algorithm is to find *N' and C'* that maximizes the variance when we reconstruct the original tensor (e.g. similar to principle component analysis). We define the desired variance that we need to sufficiently reconstruct the tensor and then use binary search to find the parameters that best produce the required variance. Finally, we fine-tune the model to recover

from the possible loss in its accuracy.

3.4.5 Optimizing Convolutional Operation

The execution of a deep learning pipeline heavily relies on matrix multiplication. However, linear algebra libraries for OpenCL (such as ClBlast and ViennaCL used in Caffe) are tuned for desktop GPUs and do not perform efficiently on smartphones.

To accelerate convolutional operation, existing frameworks use a technique called *unfolding* that converts inputs into a large matrix and then uses matrix multiplication on the unfolded input and filters to compute the result [16]. The unfolding technique requires a large amount of memory and bandwidth when executing convolutional layers. Unfortunately, the memory bandwidth on mobile GPU is quite small compared to server GPU. This makes the unfolding technique unsuitable for *DeepMon*.

Deeper observations showed that convolutional operations performed without unfolding tend to consume less bandwidth for memory access. However, it also stores the data, in memory, in a non-contiguous fashion, making it inefficient when running on memory-constrained mobile GPUs. Our second observation is that carefully laying out the convolutional weights in the format of *[N x D x D x C]* and its input in *[H x W x C]* makes it more GPU friendly as we can read multiple items at the same time using OpenCL functionality. We also note that Caffe and YOLO use the format of *[N x C x D x D]* for the weights of convolutional layers.

Figure 3.8 shows the speedup between our implementation and conventional unfolding approach. We benchmark two approaches using convolutional layers extracted from VGG-VeryDeep-16. We drop convolutional layer 7, 10, 12 and 13 from our benchmark since they have similar parameters to the other layers. We extract unfolding kernel from Caffe and use ClBlast library (one of three linear algebra used in Caffe) to do convolutional operations. Results show that on lower bandwidth devices (Note 4 and Z5), our approach almost provides the better latency

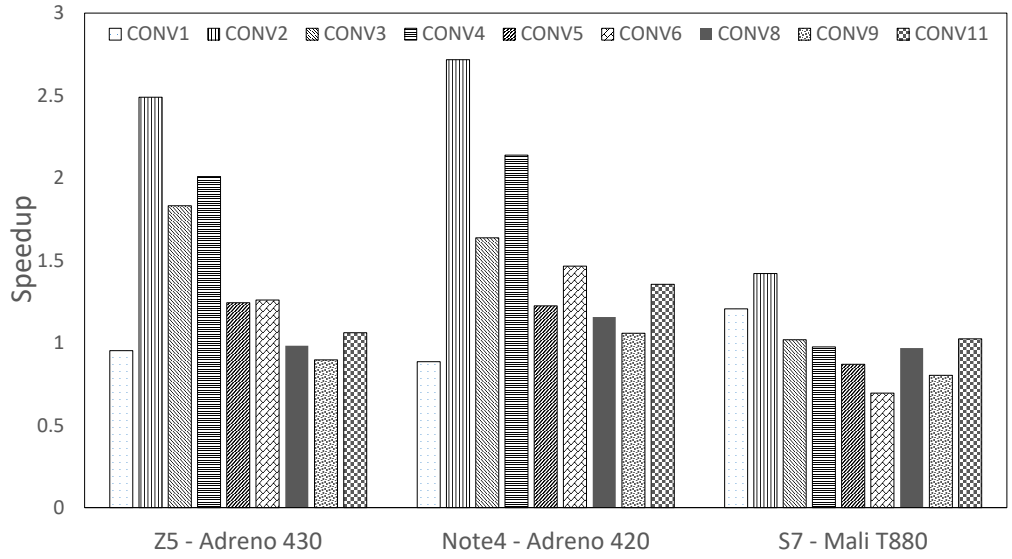


Figure 3.8: Speedup Comparison with ClBlast

(speedup > 1). However, on S7, since the device has integrated LPDDR4 which has doubled bandwidth comparing to two other devices, conventional approach starts to benefit at some layers. We additionally validated the latency of convolutional layers with another commonly-used library, ViennaCL (on Caffe). We found out that ViennaCL performs slower than ClBlast on Samsung Galaxy S7 – mainly due to its lack of support to optimize various parameters.

We further reduce the processing latency of the convolutional operations by using half floating point precision in OpenCL. Since the memory bandwidth is limited on the mobile devices (compared to desktop machines), it is highly useful to reduce the size of memory reads and writes by half by dropping the last half digits of the data. Our results, shown in Section 3.5, indicate that this optimisation is effective at reducing latency without any significant impact on the accuracy.

3.4.6 Scaling to Various Mobile GPUs

We implemented a number of techniques to allow *DeepMon* to support various types of mobile GPUs. The most important consideration was to adapt to the different memory architectures of different mobile GPUs and the ways in which they read-

/write data from the main memory.

Mobile GPUs support unified memory access that allows GPUs to directly access the main memory and use it as its own memory. However, the main memory is shared among the many components of a mobile device and its data read/write bandwidth is limited. This limited bandwidth could slow down the processing of *DeepMon* as DNN execution usually requires the GPU to read a large amount of data from the main memory.

One possible solution is to use *local memory* on the GPU chipset itself. The local memory is a small memory (for instance 8KB on Adreno 330 and 32KB on Adreno 430) which is used as a cache to accelerate memory access during computation (data is first loaded into local memory and is reused during computation). However, the size and architecture of the local memory vary across different GPUs. For example, different Adreno boards have different sizes of local memory while Mali GPUs have no local memory. Such differences are the key challenge in making *DeepMon* support different mobile GPUs.

We address this issue by building kernel codes that can exploit different amounts of local memory (including a kernel code for no memory) and dynamically uses the appropriate code at runtime. In particular, when executing convolution layers, if the memory requirement for a single filter fits into the small local memory, we adaptively use kernel code that supports that amount of local memory. Otherwise, we use the non-local-memory version.

We also build the kernel code in a way that the filters within a convolutional layer are shared to evaluate all input values. Accordingly, for the first layer of VGG-VeryDeep-16, we can fit all 64 filters with the size of *[3 x 3 x 3]* into the 8KB local memory of the Xperia Z5. For the deeper layers that require more than available local memory, *DeepMon* loads a subset of filters into the local memory and compute partial outputs at a time. We also find out that the half floating point approximation reduces the size of filters by half, allowing *DeepMon* to load more filters into the local memory. Table 3.5 shows the processing time for the four first

	conv_1 (ms)	conv_2 (ms)	conv_3 (ms)	conv_4 (ms)
Host memory	78.66	667.10	340.59	757.12
GPU local memory	63.98	526.9	262.57	584.80

Table 3.5: Benefit of using GPU Local Memory

convolutional layers while executing VGG-VeryDeep-16 on the Sony Xperia Z5 phone. It indicates that the use of local memory accelerates the processing time by 23-30%.

3.5 Experiments

3.5.1 Experimental Setup

We extensively measured the performance of *DeepMon* with a variety of deep learning models and mobile GPUs.

Workloads. We used a variety of deep learning models as shown in Table 3.1. We mainly report the results for two deep models, VGG-VeryDeep-16 and YOLO, and report the results for other models only when they are significant. We used the VGG-VeryDeep-16 model trained with the ILSVRC2012 train dataset [22] and YOLO trained with the Pascal VOC 2007 train dataset [26].

Metrics and datasets. We used processing latency, accuracy, and power consumption as our key evaluation metrics. For the latency, we measured the duration to process an image, i.e., $t_1 - t_2$ where t_1 is the time that *DeepMon* outputs the inference result and t_2 is the time that *DeepMon* receives the input image. For the latency evaluation, we used two test datasets: (i) the *UCF101* dataset [78] comprising 13,421 short videos (less than a minute long) created for activity recognition and (ii) LENA dataset [77] consists of 200 first-person-view videos captured from Google Gla-sses, and report the average latency across all processed frames along with the 95% confidence interval. We used the UCF101 dataset by default while we report the performance for LENA dataset in Section 3.5.8 and Section 3.5.9.

For accuracy, we measured the percentage of accuracy drop compared to the original models. We focused on the drop as our goal is not to improve the accuracy but to keep it close to that of the original models while accelerating inference speed. Note: unlike prior work [35], we did not reduce the total number of possible output categories (e.g., the number of objects that can be recognized by the model). For the accuracy evaluation, we used the ILSVRC2012 [22] validation dataset for VGG-VeryDeep-16 and the Pascal VOC 2007 test dataset [26] for YOLO, and calculated the average accuracy over each test dataset. For YOLO, we used the mean average precision (mAP), which is a standard metric to evaluate the YOLO's accuracy regarding both object recognition and localization [26].

Finally, we measured the power consumption using the Monsoon power monitor [4]. We reported the average energy consumption of the smartphone while processing an image in uAh by measuring the baseline energy consumption before running the processing logic and deducting the baseline from the measured value. For energy evaluation, we used the UCF101 dataset (the same dataset used in the latency evaluation), and report the average energy consumption across all processed frames along with the 95% confidence interval.

Alternatives. We compared the performance of *DeepMon* with other plausible smartphone-based alternatives such as *basic-CPU* and *basic-GPU*, and a few cloud-based alternatives. *basic-CPU* only uses the mobile CPUs to compute the full deep learning pipelines while *basic-GPU* utilizes the mobile GPUs for all processing layers without optimization. For the cloud-based approaches, the mobile device sends images to a cloud server, the server processes the images and return the results back to the mobile device (details of the cloud-based alternatives are explained in Section 3.5.4). Also, to look into the benefit of *DeepMon*, we applied different combinations of the optimization techniques presented in Section 3.4 such as convolutional layer caching (denoted as CA in the figures), layer decomposition (DC), and half floating-point calculation (HF).

Devices and APIs. We used a Samsung Galaxy S7 (with Mali T880 GPU), a

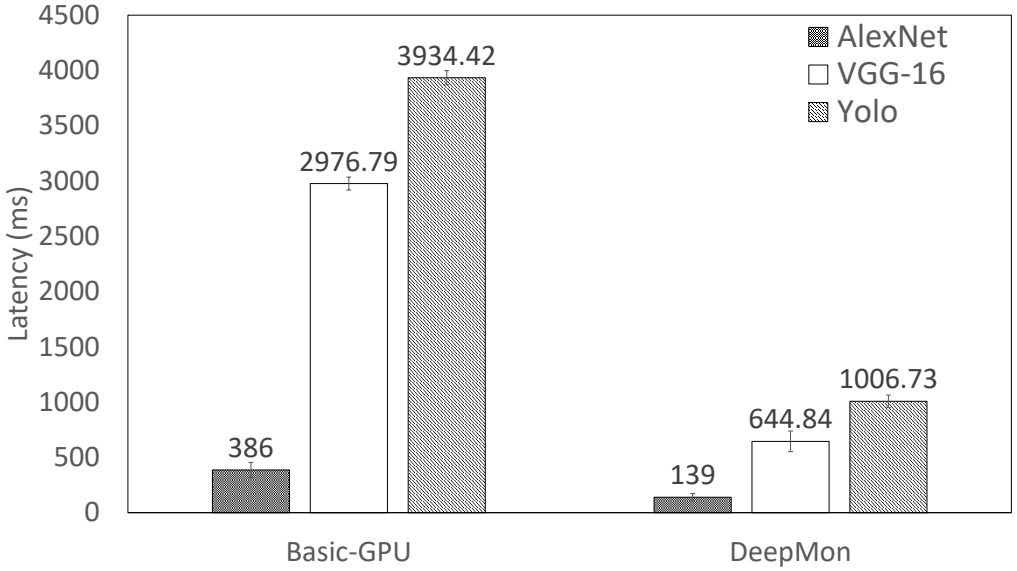


Figure 3.9: Overall Processing Latency

Samsung Galaxy Note 4 (with Adreno 420), and a Sony Xperia Z5 (with Adreno 430) as our experiment devices. Unless mentioned, we used the S7 as the default device. Also, we used the OpenCL implementation of *DeepMon* by default while we measured the performance of the Vulkan 1.0 implementation in Section 3.5.7.

3.5.2 Processing Latency

We first study the overall processing latency of *DeepMon* in comparison with naive approaches. Figure 3.9 shows the results, on an S7, for the three models: AlexNet (trained with the ILSVRC2012 train dataset), VGG-VeryDeep-16 and YOLO.

The figure shows that *DeepMon* accelerates the processing of deep learning models by 3-5 times compared to *basic-GPU*. *DeepMon* processes VGG-VeryDeep-16, a model with 13 convolutional layers and 3 fully-connected layers, at the latency of 644ms, enabling near real-time processing of continuous image streams. YOLO takes about 1 second as it includes more number of convolutional layers to track their locations of the objects.

For smaller models such as AlexNet (or equivalents such as VGG-F or VGG-M with 5 convolutional layers and 3 fully-connected layers), *DeepMon* can process

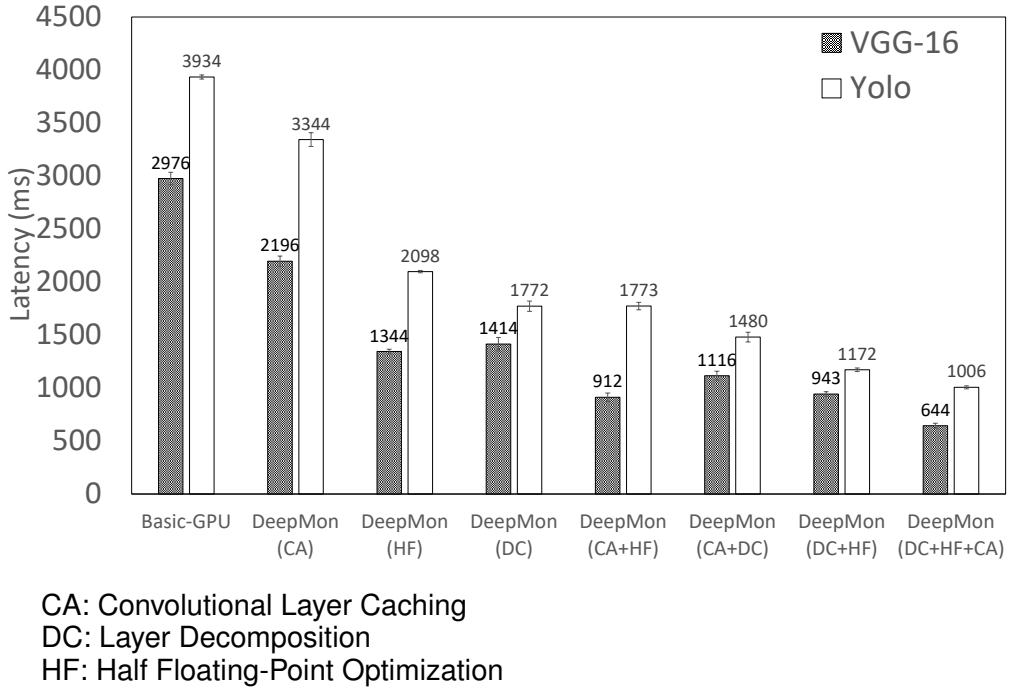


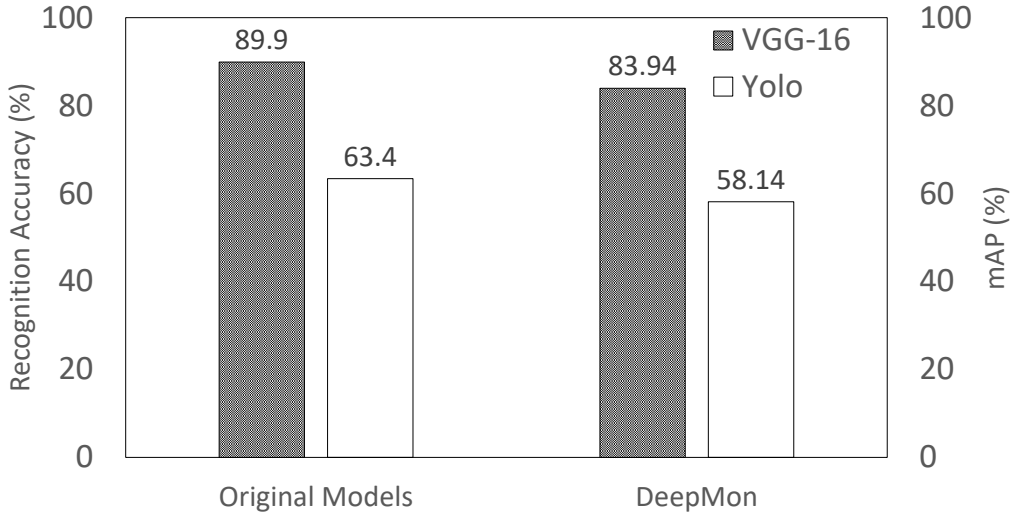
Figure 3.10: *DeepMon* Latency Breakdown

an image with just 139 ms of latency. Note: The processing time of *basic-CPU* is slower by one or two orders of magnitude depending on the model. It takes 6345ms for *basic-CPU* to process an image using AlexNet, which is 45.6 times slower than *DeepMon*.

Digging deeper, we analysed which *DeepMon* techniques contribute to the processing benefits. Figure 3.10 shows the latency breakdown for VGG-VeryDeep-16 and YOLO while cumulatively applying the various optimization techniques. The results show that all techniques significantly contribute to the latency reduction for VGG-VeryDeep-16. For YOLO, the benefit of the caching was smaller than VGG-VeryDeep-16 as the layer decomposition technique highly optimizes the first few convolutional layers, making the reuse of the cached results less beneficial.

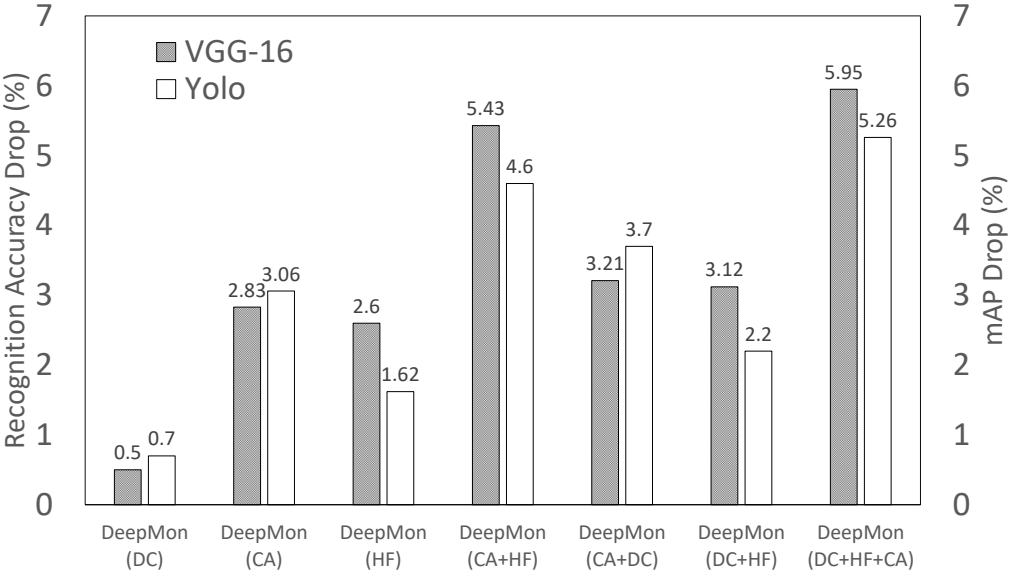
3.5.3 Recognition Accuracy

Next, we investigate how much accuracy *DeepMon* compromises in return for the latency benefits. Figure 3.11 shows the classification accuracy of the original VGG-



We reported the classification accuracy for VGG-VeryDeep-16 and the mean average precision (mAP) for YOLO.

Figure 3.11: Recognition Accuracy



CA: Convolutional Layer Caching
DC: Layer Decomposition
HF: Half Floating-Point Optimization

Figure 3.12: Breakdown of *DeepMon* Accuracy Drop

VeryDeep-16 and the mAP of YOLO as well as the converted models optimized to run on *DeepMon*. The figure shows that *DeepMon* drops about 5-6% of accuracy while accelerating the latency 4-5 times. We designed our techniques to keep the properties of the original architecture, thus minimizing the impact on the recognition output. Note that Fast-YOLO [70], a lightweight version of YOLO shows the lower mAP of 52.7%, which is 5.44% lower than that of *DeepMon*, while the latency benefit of Fast-YOLO was similar to *DeepMon* (i.e., ≈4.5 times when experimented on Samsung S7).

We further analysed which of *DeepMon's* components contributed to the accuracy drop. Figure 3.12 shows the results by applying the three different techniques that affect the accuracy. The accuracy drop by layer decomposition is marginal, indicating that our binary-search-based decomposition selects suitable decomposition parameters. Also, the convolutional layer caching reduces accuracy by about ≈3%, showing that the use of cached results marginally affects the accuracy for video streams.

3.5.4 Comparison with Other Approaches

We now compare the processing latency of *DeepMon* with DeepX, the state-of-the-art mobile deep learning inference engine. Figure 3.13 shows the latency and accuracy drop of DeepX and *DeepMon*; we ran AlexNet using the SnapDragon 801 processor. DeepX consumes 500ms to process an image with an accuracy drop of 5%. *DeepMon's* latency was 269ms, twice as fast as DeepX, when all techniques are applied while its accuracy drop was 1% higher at 6%. *DeepMon* can be adjusted to only use the layer decomposition method which achieves 333ms latency (≈33% faster than DeepX), but with only a 1.6% accuracy drop.

We also compared the latency of *DeepMon* with the cloud-based alternatives. Figure 3.14 shows the results. We used three different cloud variants: *edge-strong*, *remote-strong*, and *remote-weak*. For, *edge-strong*, the mobile phone and the server

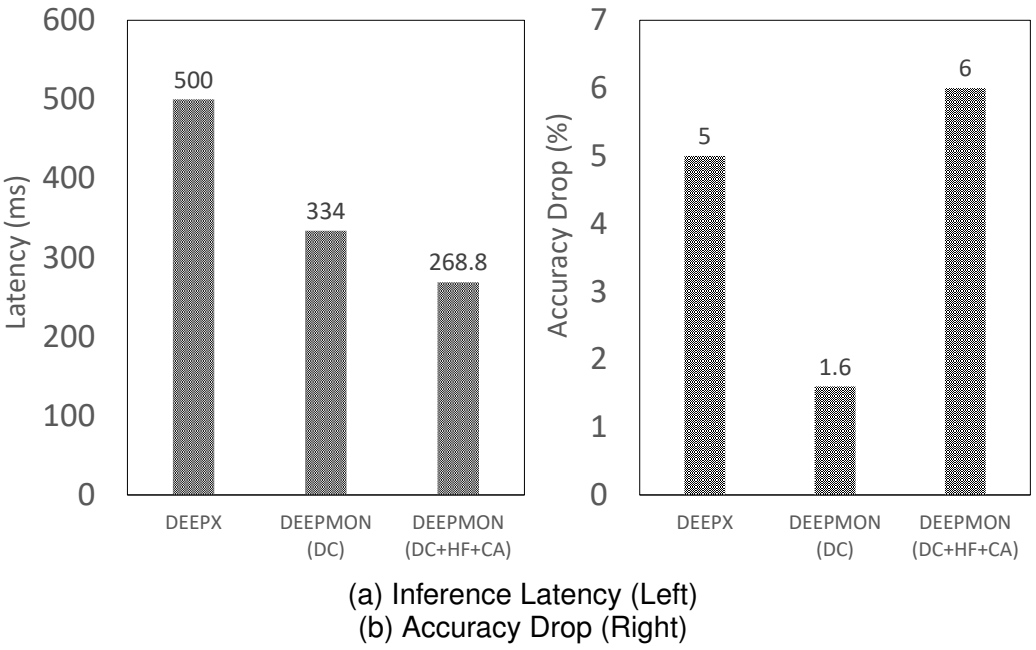


Figure 3.13: Comparison to DeepX on Samsung Galaxy S5

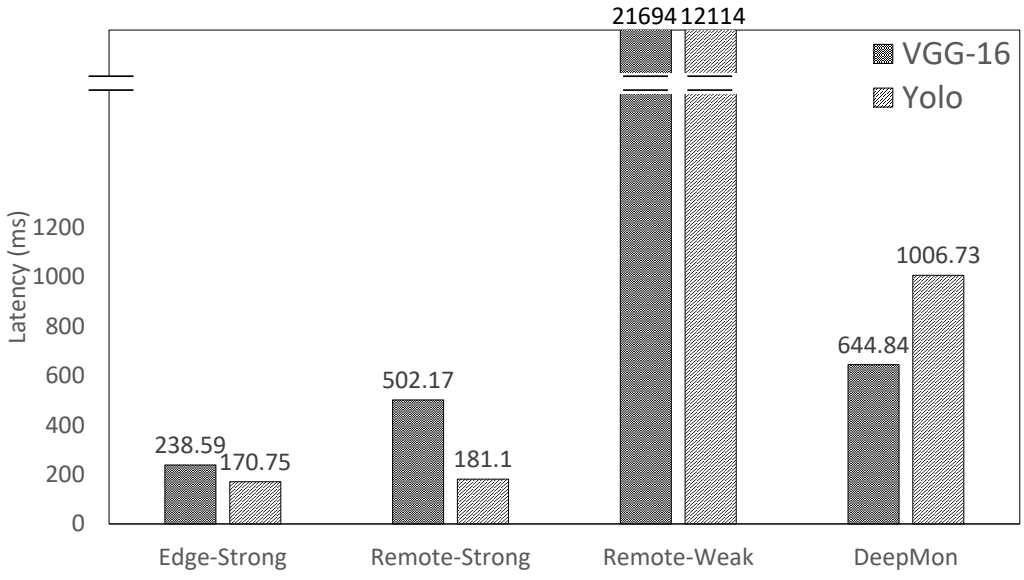


Figure 3.14: Comparison with the Cloud-based Approach

was connected through the local Wi-Fi network while the server is equipped with a NVidia GTX 980 GPU (2,048 GPU cores, 8GB memory size and 224GB/s memory bandwidth). For *remote-strong* and *remote-weak*, we used Amazon EC2 servers (in particular g2.2xlarge and t2.medium instances respectively) located in the EC2 Asia Pacific (Singapore) datacenter. *remote-strong* was equipped with a K520 GPU (with 8 cores and 15 GB of memory) with while *remote-weak* had no GPU. We used the Caffe [47] and YOLO [70] frameworks to run the models on the cloud.

edge-strong is 2.7 times faster than *DeepMon* while *remote-strong* is only 28% faster than *DeepMon*for VGG-VeryDeep-16. The latencies of *remote-weak* were 33.6 and 12 times slower than *DeepMon* , respectively, due to its CPU-based execution of deep learning models. This suggests that we can leverage cloud services for home- or office-based applications where the user can offload the data safely to the edge servers with low networking latency and fewer privacy concerns. On the other hand, we need to be careful about using the remote clouds even when the users are willing to send the data. The cost for *remote-strong* (using g2.2xlarge server instance) is 1 USD per hour, imposing huge service cost for continuous vision applications. We can use less powerful instances, although doing so might not improve the latency as indicated by the numbers for *remote-weak*.

3.5.5 Power Consumption

We now investigate the power consumption of *DeepMon* in comparison with *basic-GPU*, *remote-strong*, and *remote-weak*. Figure 3.15 shows the overall power consumption for each approach along with the breakdown. All *DeepMon* measurements were done on Samsung Note 4 as it has a detachable battery that could be tapped with the Monsoon power meter.

The figure shows that *DeepMon* is lower than the power consumption of basic-GPU by more than 5 times for both VGG-VeryDeep-16 and YOLO. This savings is mostly from the reduced processing time. *remote-strong* consumes 3 times lesser

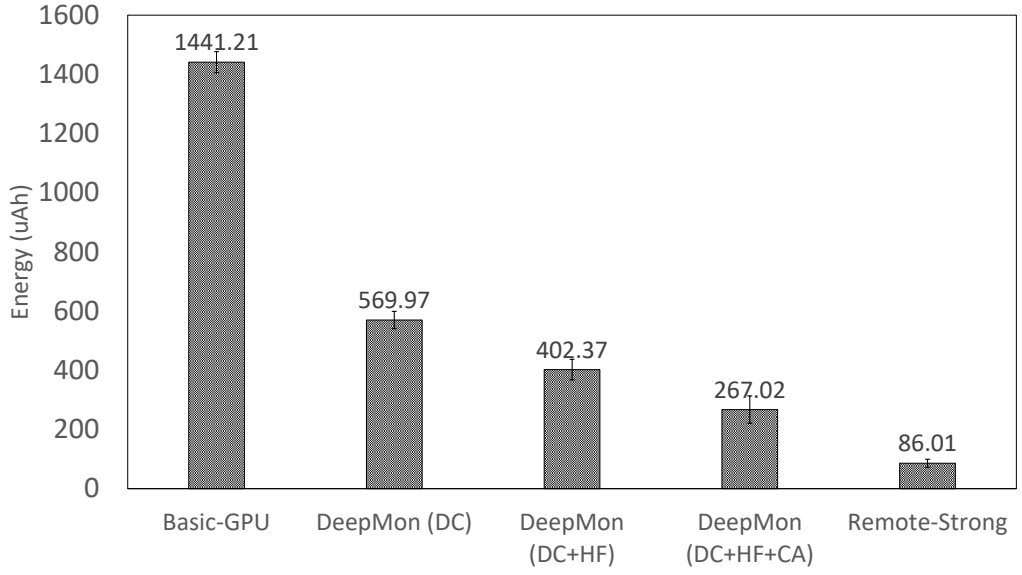


Figure 3.15: Overall Power Consumption

power as the mobile device consumes power only to send the image to the cloud and then goes into power saving mode until it receives the result. However, as stated earlier, you need a large expensive server instance to see small latency benefits compared to *DeepMon*.

3.5.6 Latency on Other Mobile GPUs

We next studied the processing latency of *DeepMon* across different GPUs. We used a Samsung Galaxy Note 4 (with Adreno 420) and a Sony Xperia Z5 (Adreno 430). Figure 3.16 shows the results. While the latency reduction pattern by all our optimization strategies remains similar, the absolute processing latency increases by 2.4 times for the Note 4 and 2.34 times for the Z5, compared to the Samsung Galaxy S7 (with Mali T 880). Even though the direct comparison between Mali and Adreno is non-trivial, Mali's faster performance is likely to result from having more GPU cores and higher memory bandwidth compared to Adreno 420 and 430. We also noticed that the original VGG-VeryDeep-16 model cannot be run on Z5 due to the limitations of the heap memory size – although it can run after the decomposition technique reduces the model size by half.

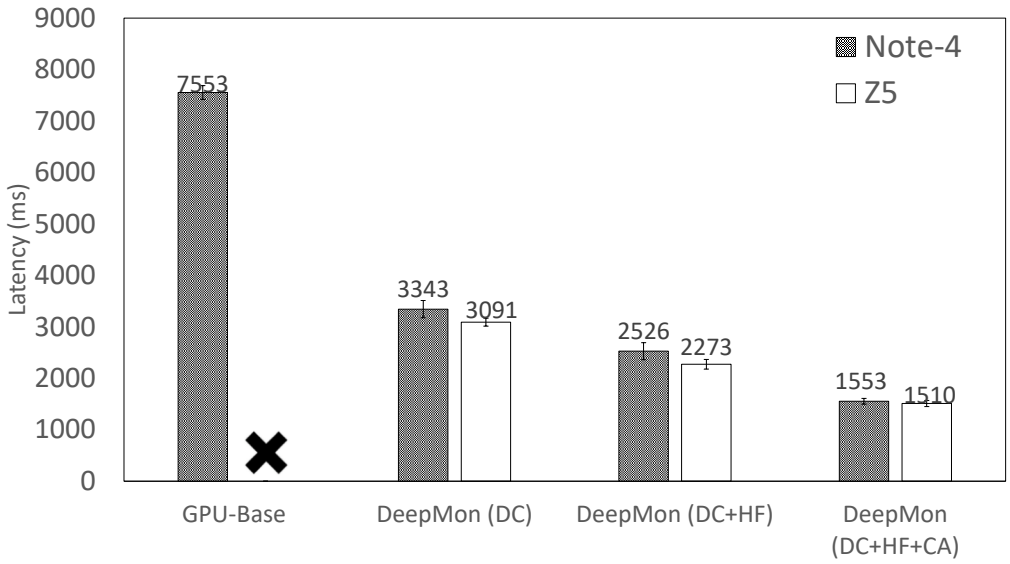


Figure 3.16: Processing Latency for Different GPUs

3.5.7 Latency of Vulkan

We also explored the performance of the Vulkan implementation of *DeepMon*. We used the Samsung Galaxy S7 that supports both Vulkan and OpenCL. Figure 3.17 shows the processing time per convolutional layer for VGG-VeryDeep-16. Even though there are small differences in processing time per layer (compared to OpenCL), all our techniques are equally effective on Vulkan as well, resulting in similar overall processing times.

3.5.8 Performance on First-Person-View Videos

We further evaluated the latency and accuracy of *DeepMon* over the first-person-view dataset, LENA, which could be the typical workload for *DeepMon*. For accuracy, we reported the percentage of frames that the base model and *DeepMon* outputs the different classification result – we define this as the output difference ratio. For VGG-VeryDeep-16, we consider that the output is different when the top-1 classification results of the base model and *DeepMon* are different. For YOLO, we consider that the output is different when the positions of the detected object (indicated as rectangles on the image) overlap less than 50% (i.e., *Intersection-Over-*

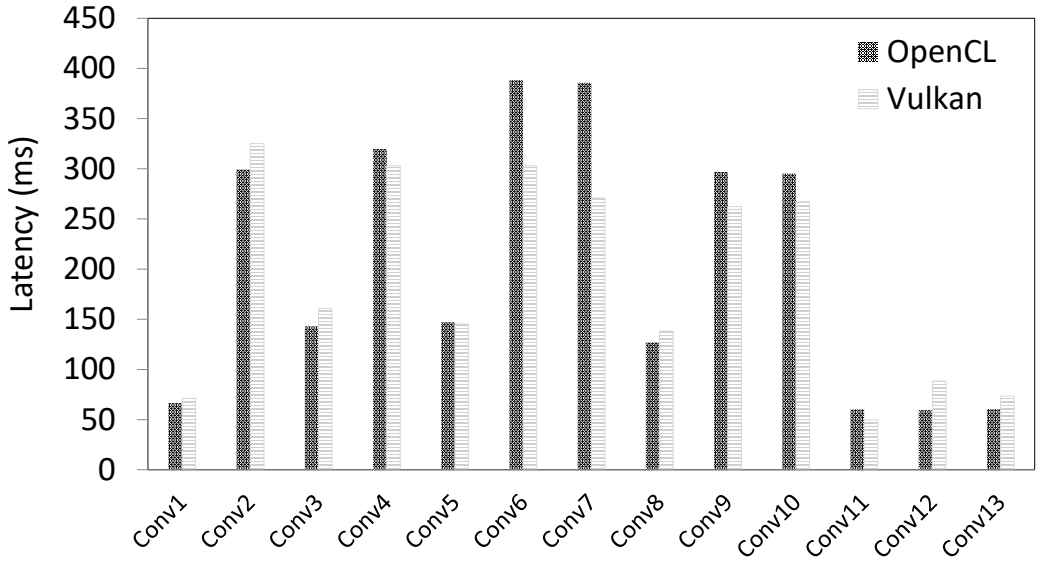


Figure 3.17: Performance of Vulkan

Union (IoU) ¡ 50%).

Figure 3.18 shows the latency of *DeepMon* on the entire LENA dataset. *DeepMon* shows ≈4 times of overall latency reduction, which is comparable to the benefit over the UCF101 dataset. In particular, our caching technique reduced ≈22% and ≈13% of the total execution times of VGG-Verydeep-16 and YOLO, respectively. The reduction rate was slightly decreased compared to that of the UCF101 dataset since the first-person-view videos tend to have more frequent changes in the recorded scenes due to continuous head movement. However, the results show that our caching technique is still effective for the first-person-view videos.

Figure 3.19 shows the output difference ratio. *DeepMon* produces different outputs for 25.89% and 12.28% of the total frames compared to the base VGG-VeryDeep-16 and YOLO models, respectively. We empirically looked into such differently-classified frames and found out that most of those frames are not correctly classified or do not have a matching class in the base model, margin-ally affecting the actual accuracy.

Interestingly, the output difference ratio of VGG-Verydeep-16 is much higher than that of YOLO. This is because VGG-Verydeep-16 always outputs one of the

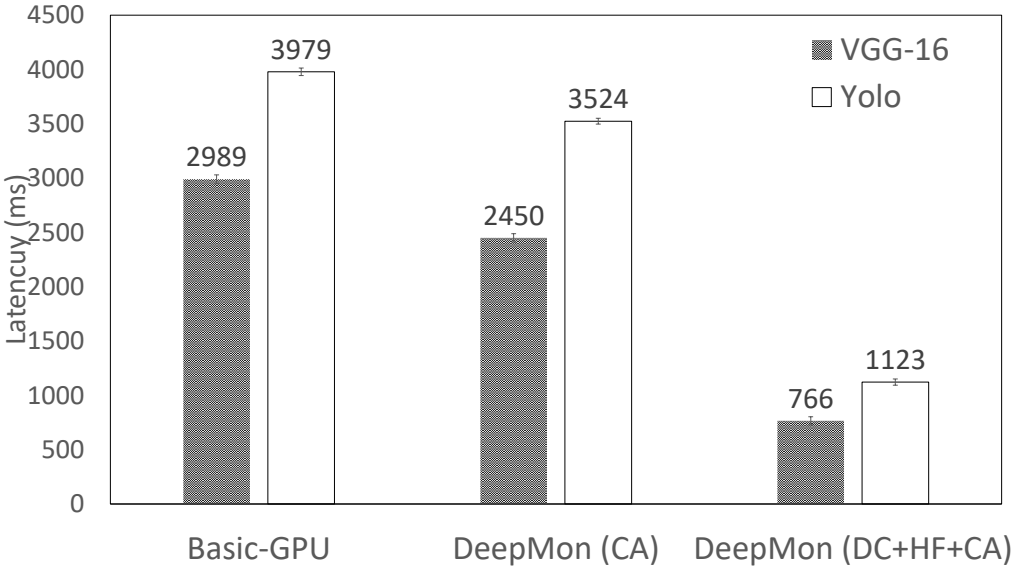
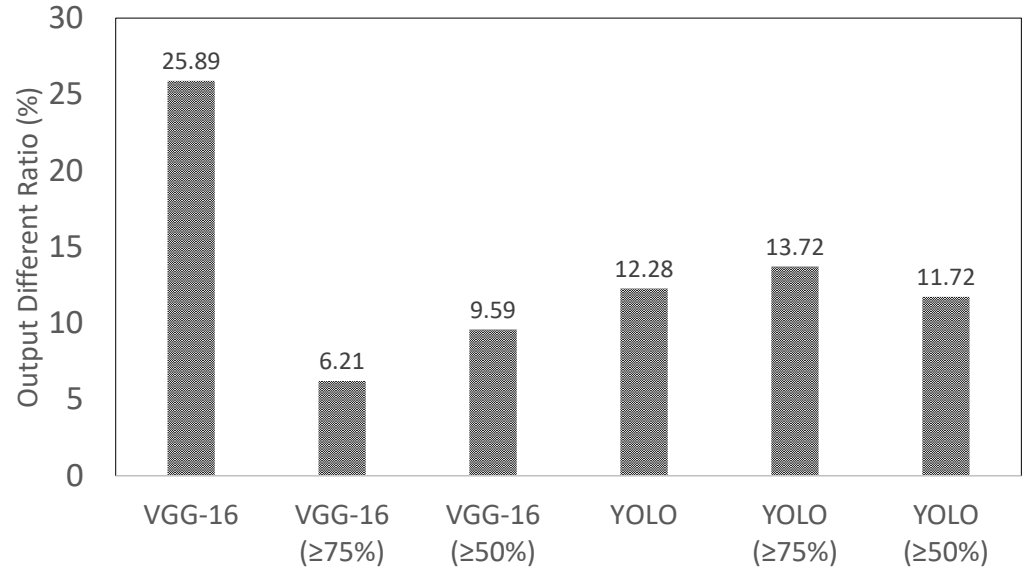


Figure 3.18: Latency on the LENA Dataset



VGG-16($\geq$75%) indicates the accuracy evaluated only on the videos with the average confident score above 75%. Similar explanation applies to VGG-16($\geq$50%), YOLO($\geq$75%), and YOLO($\geq$50%).

Figure 3.19: Accuracy on the LENA Dataset

	SIFT-based	Histogram-based
Overhead (ms)	2,580	4.77
Overall latency change (ms)	2,064 (increased)	-534 (decreased)
Output difference ratio (%)	3.875	6.21
Cache hit rate (%)	31.4	35.52

Table 3.6: Caching Performance Analysis

1,000 pre-trained classes even though the target frame is unlikely to be one of the 1,000 classes; for the consecutive frames with low classification confidences, their top-1 classified objects vary sensitively from one frame to another (although the frames include the same object), making our caching results different from the newly calculated ones. We further calculated the output difference ratio only over the videos that have the average classification confidence higher than 75% and 50%, and the output difference ratio was reduced to 6.21 and 9.59, respectively. For YOLO, the output difference ratio did not vary much since the model eliminated "others" when its classification confidence was below a certain threshold.

3.5.9 Convolutional Layer Caching Performance

We further studied how our caching technique performed over the LENA dataset. We used Vgg-VeryDeep-16 for this study. Table 3.6 shows the results on the videos with the average confidence score over 75%. *DeepMon* (with its histogram-based caching) shows the average latency reduction of 538 ms. The benefit comes from 35.52% of cache hits, significantly reducing unnecessary recalculation of convolution operations. We noticed that the latency reduction was ≈20% less than that of the UCF101 dataset. As expected, the cache hit rate over LENA, the first-person-view dataset, was lower compared the cache hit rate over the UCF101 dataset. This is mainly because head-mounted cameras tend to move more than third-person-view cameras, resulting in bigger differences between the two consecutive images.

We also compared our proposed histogram-based caching algorithm against an alternative using SIFT features [64]. Although SIFT-based algorithm provides the

	Base	Base+HF	DC	DC+HF
VGG-Verydeep-16(MB)	578	289	517	258.5
YOLO(MB)	1,116	558	1,002	501

"Base" indicates the original model.

Table 3.7: Memory Footprint

lower output difference ratio (3.875%) than the ratio of the histogram-based algorithm (6.21%), extracting SIFT features from multiple blocks of an image is highly time-consuming; it took over 2.5 seconds to calculate SIFT features for an image (across all convolutional layers). Due to high overhead to calculate the SIFT features, it cannot be used to compare image blocks for caching. On the other hand, our histogram-based approach can compare blocks of an image within 5 ms, making it much more suitable to be adopted for our caching algorithm.

3.5.10 Memory Footprint

Table 3.7 shows the memory footprint for VGG-Verydeep-16 and YOLO. The memory usage is well within the available memory spaces of commodity mobile devices, showing that *DeepMon* manages its memory usage efficiently. Also, the decomposition and half-floating point approximation reduce the memory usage of *DeepMon*; they reduce the memory usage from 578MB and 1116MB down to 258.5MB and 501MB for VGG-Verydeep-16 and YOLO, respectively. For the models that require large memory spaces, other optimization techniques such as *Singular Value Decomposition* (SVD) [54] can be applied to further reduce the memory usage.

DeepMon mainly uses the memory to load the model and stores input and output of a layer. *DeepMon* stores the entire model within system memory for efficient inference since it is time-consuming to load the model on-demand from the external memory. On the other hand, *DeepMon* only stores input and output of the currently executing layer – it discards all output data from previous layers once they become of no use to keep memory usage as low as possible. Accordingly, memory usage of

DeepMon is capped at the size of the model and the largest input and output size of a single layer.

Chapter 4

D-pruner: Filter-based pruning method for deep convolutional neural network

The emergence of augmented reality devices such as Google Glass and Microsoft Hololens has opened up a new class of vision sensing applications. Those applications often require the ability to continuously capture and analyze contextual information from video streams. They often adopt various deep learning algorithms such as convolutional neural networks (CNN) to achieve high recognition accuracy while facing severe challenges to run computationally intensive deep learning algorithms on resource-constrained mobile devices. In this paper, we propose and explore a new class of compression technique called *D-Pruner* to efficiently prune redundant parameters within a CNN model to run the model efficiently on mobile devices. *D-Pruner* removes redundancy by embedding a small additional network. This network evaluates the importance of filters and removes them during the fine-tuning phase to efficiently reduce the size of the model while maintaining the accuracy of the original model. We evaluated *D-Pruner* on various datasets such as CIFAR-10 and CIFAR-100 and showed that *D-Pruner* could reduce a significant amount of parameters up to 4.4 times on many existing models while maintaining accuracy drop

less than 1%.

4.1 Introduction

The appearance of augmented reality devices such as Google Glass and Microsoft Hololens has been opening up various new vision sensing applications. The core function of these applications is to continuously capture contexts of users and surroundings from streaming video data and enable situational interactions with users. For example, a virtual assistant system for dementia patients identifies objects and people near to the patient and provide the patient with the intelligent guidance in real-time [13]. Recently, deep learning algorithms such as a convolutional neural networks (CNN) have been actively adopted for various computer vision tasks such as image recognition, object detection, and identification tasks to achieve higher recognition accuracy [36, 76, 81].

The key challenge to enable continuous vision applications is to run the state-of-the-art CNN models efficiently on resource-constrained mobile devices. Recent CNN models such as VGG-16 [76], ResNet [36], and Inception [81] often require a huge amount of computational resources regarding CPU/GPU cycles or memory usage, making their execution slow on mobile devices. For instance, VGG-16 and ResNet-152 require 15.3 GLOPS and 11.6 GLOPS to recognize a single image, which often takes at least hundreds of milliseconds on the commodity smartphones [44, 45, 54]. To address this problem, cloud offloading is often considered. However, the offloading approach has critical privacy concerns as it may expose a massive volume of private images and videos of users to the cloud.

Previous works [12, 24, 33, 50, 87] have shown that CNNs usually have a lot of redundancy in terms of filters and parameters. The problem is further aggravated since developers often leverage *transfer learning* [67] to fine-tune the state-of-the-art models on new datasets to increase recognition accuracy. For example, the first 13 convolutional layers in VGG-16 can be used to provide robust features for a

variety of new tasks such as classifying different types of fruits or animals which are not available in the ImageNet dataset [22]. Developers can attach a few additional layers on top of the existing 13 layers to fine-tune the network on new datasets. In many cases, if we don't process it carefully, transfer learning makes the model unnecessarily large and redundant to run on mobile devices.

Compression of the neural networks has been actively studied for efficient execution of deep neural networks. Some works [12, 24] focus on approximating each layer separately via factorization techniques and fine-tune the whole network to restore accuracy. However, without global knowledge about relationships between lower and upper filters, independent pruning of filters might lead to significant loss in recognition accuracy.

In this paper, we propose a general technique called *D-Pruner* to reduce the memory footprint and computational cost of many existing and transferred CNN models. *D-Pruner* automatically figures out redundant filters in convolutional layers and removes them to make the model smaller in terms of memory and computational requirements. Its key idea is to embed a small network called *masking layer* into every convolution layer to score how effectively each filter contributes to the outcome. *Masking layers* removes only low scored filters and fine-tune the network to keep the accuracy while pruning out the unnecessary filters. By learning the extended network end-to-end, *D-Pruner* can figure out the relationship between filters and make a better pruning decision.

We conducted several experiments on two different datasets (CIFAR-10 and CIFAR-100 [51]) to evaluate *D-Pruner*. Our results show that *D-Pruner* can compress existing models to be 4.4× and 2.76× smaller in terms on memory footprint, 4.57× and 2.9× better in term of computational cost on CIFAR-10 and CIFAR-100 respectively. In our latency tests, pruned models on CIFAR-10 and CIFAR-100 achieve the speedup of 1.85× and 1.61× on Samsung Galaxy S7 device. Furthermore, *D-Pruner* achieves 8% smaller in size with accuracy of 90.48% comparing to pruned VGGNet with accuracy of 90.5% as proposed in *DeepIoT* [87] on CIFAR-

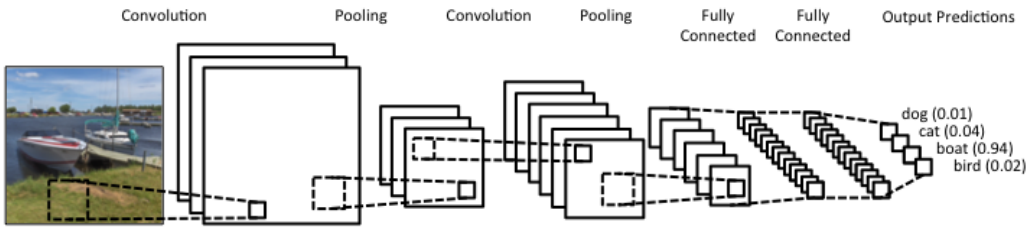


Figure 4.1: Convolutional Neural Network Architecture

10. We believe that mobile developers would be beneficial from *D-Pruner* to build small and efficient CNN models for many vision sensing tasks.

The contribution of our paper can be summarized as follows:

- We propose *D-Pruner* , a simple but effective compression technique to remove redundancy within existing and transferred CNN models. *D-Pruner* introduces a novel concept of the masking block to figure out redundant filters which have low impacts on final accuracy.

- We leverage the knowledge from the training set to effectively remove only a subset of redundant filters to maintain accuracy at the highest level.

- We conducted intensive experiments using two different datasets on two network architectures to demonstrate the usefulness of *D-Pruner*. Our results on CIFAR-10 and CIFAR-100 [51] show that *D-Pruner* can compress existing models to be 4.4× and 2.76× smaller in terms on memory footprint, 4.57× and 2.9× better in term of computational cost on CIFAR-10 and CIFAR-100 respectively. In our latency evaluation, pruned models on CIFAR-10 and CIFAR-100 achieve the speedup of 1.85× and 1.61× on Samsung Galaxy S7 device.

4.2 Convolutional Neural Network

Since AlexNet architecture was proposed in 2012 [22], there have been many significant changes in the first network architecture (Figure 4.1) to improve the ca-

pabilities of CNN on many computer vision tasks. One interesting change is the replacement of fully connected layers or dense layers by [1×1] convolutional layer and global average pooling in many state-of-the-art models such as ResNet [36], Inception network [81]. As dense layers consume the most parameters in CNN [44], this change significantly reduces the size (or memory footprint) of state-of-the-art models. However, as modern networks still rely heavily on convolutional layers to extract meaningful visual features, high computational cost is still an open problem [44].

There are two widely used methods to reduce computational cost in CNN. The first method is to use factorization techniques such as SVD (singular-value decomposition) to approximate the weights matrices during inference step to reduce the total processing operations. However, this approach tends to have high accuracy loss on very deep networks [12, 54]. The second method is to prune the redundant filters to achieve simpler but more efficient CNNs. As the computational cost is proportional to the number of filters, pruning unnecessary filters will result in improving both training and inference time. Many works have shown potential results using this approach [50, 87].

D-Pruner follows the latter approach by recognizing redundancy automatically during fine-tuning process. *D-Pruner* is designed as a general technique to compress any modern CNN models to be smaller and less resource-consuming to work efficiently on both servers and mobile devices.

4.3 D-Pruner Algorithm

In this section, we first introduce briefly how the technique works. Secondly, we provide details about our novel masking block to determine removable filters. Finally, we show how the training process takes place to prune unnecessary parameters based on the knowledge from masking blocks.

The algorithm works in multiple pruning iterations. In each iteration, we first

expand all convolutional layers with extra layers called masking blocks to score how much each filter impacts on final accuracy. Each masking blocks will output a set of candidate filters to be removed for each particular image input. In order to prune only filters that have little impacts on the outcome, we leverage the all training images to collect the probability to be removed of each filter. We only remove those with high probability of being removed (e.g. over 95% on training set). We then fine-tune the new network to recover original accuracy and achieve a smaller model. Finally, we repeat the pruning process again until it converges (e.g. accuracy drop is above certain threshold.).

4.3.1 Masking Block

The goal of masking block is to determine removable filters during the pruning process. For example, fine-tuning ImageNet models such as VGG-16 or ResNet to detect multiple types of fruits might contain a lot of redundant filters to recognize animals, which can be removed to make the model smaller and simpler. By attaching masking block to convolutional layer, it will inspect the output of every filter and score how effectively they affect the final outcome. Hence, unnecessary filters may be removed if they have no or little impact on the final accuracy. Furthermore, masking block incurs very small computational overhead and should be easily fine-tuned.

Our masking block is inspired by SE block in Squeeze-and-Excitation network [40] which is used to measure the importance of each filter within a single convolutional layer. We leverage SE block and add masking function in order to filter out top-K unimportant filters.

Masking block as shown in Figure 4.2 consists of an average pooling followed by 2 dense layers and a softmax layer to compute the score of each particular filter. Afterwards, the masking layer takes the scores, a maximum number of filters K to be removed and outputs the binary masks which zero out top-K lowest scores. At the

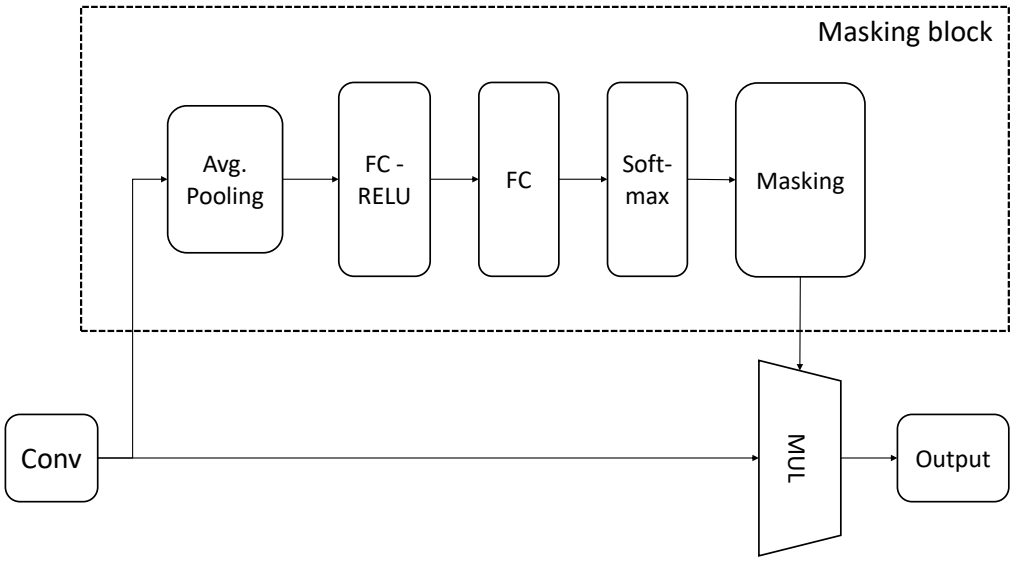


Figure 4.2: Masking Block

end, we multiply the masks and previous output of convolutional layer to remove all unnecessary outputs corresponding to removed filters. Hence, only remaining output will contribute to the final outcome during fine-tuning process.

4.3.2 Pruning Method

The pruning process consists four main stages as described in Algorithm 2.

- Firstly, we attach masking blocks to original network as shown in Figure 4.2 and fine-tune the network on training set (line 4). We fix the original network and train only the masking blocks for first few epochs and then fine-tune the whole network for few more epochs afterwards. (line 5)

- Secondly, we predict which filters should be removed within the final network architecture. As the masking block outputs a set of removable filters for every single image input, one filter can be removed for a particular input but can be preserved for another. We collect the removal distribution of each filter on the all training images and only remove the filters that have removable probability higher than predefined threshold (e.g. 95%) (line 6-14). For instance, the first convolutional layer of VGG-16 has 64 filters. If we use K=10 filters, during

the training phase, masking block will automatically zero-out all the output of 10 filters with lowest scores on each training image. Only 54 remaining filters will contribute to the final output. However, masking block does not always produce the same 10 filters for every training image. In order to make correct pruning decision, we use all training images to collect the probability to be removed of all 64 filters and remove only those have higher than a threshold T.

- Finally, we build a new network by removing masking blocks and removed filers from previous step (line 15). We transfer the learned parameters to the new network and fine-tune it for few epochs to recover original accuracy (line 16-17).

- We update the new model if validation accuracy is within affordable range (line 18-21) and repeat the pruning process until we satisfy with the result or final accuracy drops below a certain threshold (line 3).

4.4 Experiments

4.4.1 Experiment Setup

Datasets. We evaluated *D-Pruner* by compressing existing models on two datasets: CIFAR-10 and CIFAR-100 [51]. Each dataset consists of 60.000 32x32 color images (50.000 images for training and 10.000 images for validation). CIFAR-10 and CIFAR-100 contains images in 10 and 100 classes respectively.

Models. We trained the ALL-CNN-C model from [79] which achieves accuracy of 90.19% on CIFAR-10 and 61.71% on CIFAR-100. In order to show the robustness of *D-Pruner* on variety of architectures, we also trained NIN (network in network) model from [59] which achieves accuracy of 89.39% on CIFAR-10 for further evaluations. Unlike other models such as VGGNet [87] that use dense

Algorithm 2 Pruning Algorithm

Data: Network O, Dataset D, const K, theshold T, epochs N
Result: Network P

$acc \leftarrow acc(O)$
$P \leftarrow$ Network O
while $acc \geq expected\_accuracy$ **do**
    P' $\leftarrow$ Network P + {masking blocks}
    Finetune P' on D for N epochs
    $R \leftarrow \{\}$
    for $\forall$ *masking block l in P'* **do**
        for $\forall$ *filter f in l* **do**
            $\Pr_f \leftarrow$ P(*mask*(f) == 1 — D)
            if $Pr_f \leq T$ **then**
                $R \leftarrow R \cup \{\text{f}\}$
            end
        end
    end
    P" $\leftarrow$ Network P - R
    Transfer learned paraemters from P' to P"
    Finetune P" on D for N epochs
    if *acc(P")* $\geq$ *acc* **then**
        $acc \leftarrow acc(P'')$
        $P \leftarrow P''$
    end
end

	CIFAR-10					
	M1	M1(*)	Impr.	NIN	NIN(*)	Impr.
Acc.(%)	90.19	89.34	-0.85	89.39	88.83	-0.44
# Params.	1.3M	310K	4.4×	966K	348K	2.77×
# Ops	281M	61M	4.57×	222M	132M	1.68×
Lat (ms)	211(±8)	113(±14)	1.85×	185(±25)	131(±11)	1.41×

	CIFAR-100		
	M1	M1(*)	Impr.
Acc.(%)	61.71	61.08	-0.63
# Params.	1.3M	501K	2.76×
# Ops	282M	97M	2.9×
Lat (ms)	208(±11)	129(±14)	1.61×

M1 : ALL-CNN-C (*): pruned model
Impr.: Improvement

Table 4.1: Overall Performance of *D-Pruner*

	CIFAR-10				
	M1	M1(*)	M1(**)	VGGNET	VGGNET-DEEPIOT
Acc.(%)	90.19	**90.48**	89.34	90.5	**90.5**
# Params.	1.3M	**664K**	310K	29.7M	**724K**

M1: ALL-CNN-C (*): Pruned model at 4th iteration (**): Final pruned model

Table 4.2: Comparison with *DeepIoT*

layers for classification, both networks in our evaluations use only convolutional layers which results in fewer number of parameters while achieving similar accuracy. ALL-CNN-C and NIN uses approximately about 281M and 222M Mul-Add operations respectively. Network architectures of ALL-CNN-C and NIN models on CIFAR-10 are shown in Table 4.3.

Training process. We used Keras [19] in *D-Pruner's* implementation. For every pruning step, we tried to remove $K = 20\%$ of the filters and fine-tuned the network for $N = 35$ epochs (10% of number of epochs we used to train original network). We used Nesterov Gradient Descent [65] for fine-tuning with learning rate, momentum and decay set to 0.01, 0.9 and 0.000001 respectively. We also used threshold T of 0.95 to determine which filter will be removed. We repeated the pruning process for several iterations until there was no filter to be removed or the expected accuracy

	Filter Shape	
Type / Stride - Activation	**ALL-CNN-C**	**NIN**
Conv1 / s1 - ReLU	3×3×3×96	5×5×3×192
Conv2 / s1 - ReLU	3×3×96×96	1×1×192×160
Conv3 / s2 - ReLU	3×3×96×96	1×1×160×96
Conv4 / s1 - ReLU	3×3×96×192	5×5×96×192
Conv5 / s1 - ReLU	3×3×192×192	1×1×192×192
Conv6 / s2 - ReLU	3×3×192×192	1×1×192×192
Conv7 / s1 - ReLU	3×3×192×192	3×3×192×192
Conv8 / s1 - ReLU	1×1×192×192	1×1×192×192
Conv9 / s1 - ReLU	1×1×192×10	1×1×192×10
Global Average Pool / s1		
Softmax		

Table 4.3: Network Architectures

loss was larger than 1%.

Metrics. We use accuracy, number of parameters, amount of mul-add operations and processing latency as our key performance metrics. For latency evaluation, we evaluated pruned models using *DeepMon* framework [45] and report the average latency on Samsung Galaxy S7 (with Exynos 8890 processor and Mali-T880 GPU).

4.4.2 Overall Results

Overall, *D-Pruner* successfully compresses investigated models to be much smaller and less computational consuming. Table 4.1 shows the performance of pruned versions of ALL-CNN-C and NIN models on both CIFAR-10 and CIFAR-100.

On CIFAR-10, *D-Pruner* easily compress both ALL-CNN-C and NIN models to be 4.4× and 2.77× smaller in memory footprint (in terms of number of parameters). It also reduces 4.57× and 1.68× computational cost (in terms of the amount of require Mul-Add operations) in ALL-CNN-C and NIN models respectively. We notice that performance of *D-Pruner* on ALL-CNN-C model is significantly higher than on NIN network due to several reasons. First, original NIN model has 1.34× less number of parameters comparing to ALL-CNN-C model which makes reduction in memory footprint seem to be lower. Second, NIN network leverages [1×1]

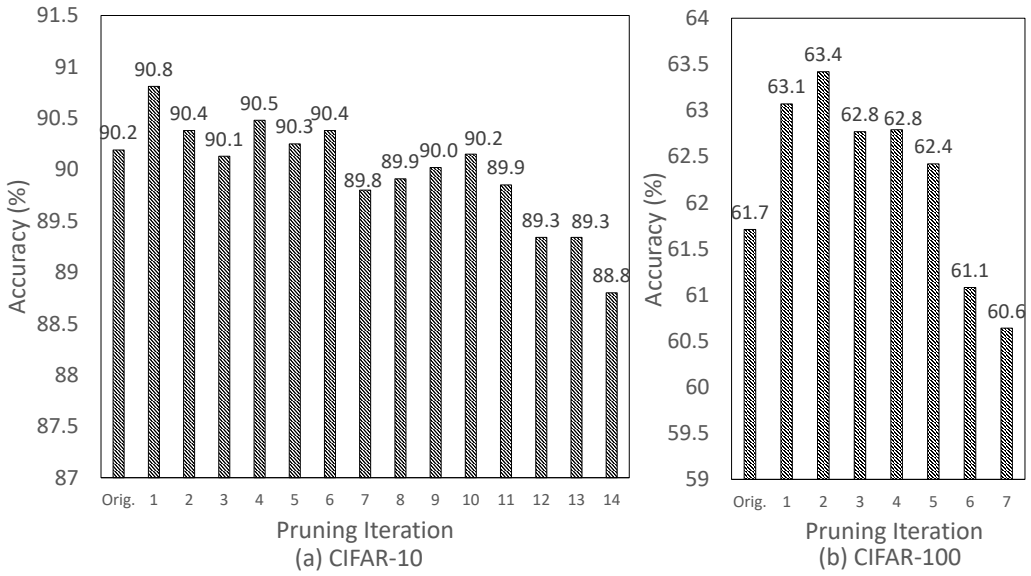


Figure 4.3: Accuracy

convolutional filter which results in significantly reduction in computational cost, which explains why computation cost is reduced only 1.68× while memory footprint is reduced 2.77×. In latency evaluations, pruned models from ALL-CNN-C and NIN networks improves inference time up to 1.85× and 1.41× respectively.

Similarly, pruned version of ALL-CNN-C achieves 2.76×, 2.9× and 1.61× reduction in memory footprint, computational cost and inference time on CIFAR-100.

4.4.3 Performance Breakdown

Next, we investigate how *D-Pruner* affects the models during each pruning iteration in terms of accuracy, amount of parameters, number of Mul-Add operations and the amount of filters in each convolutional layer using results from pruning ALL-CNN-C model. In general, giving the expected accuracy drop, *D-Pruner* gradually compresses the model by pruning unnecessary filters over various iterations and makes it smaller in terms of memory footprint and computational cost while trying its best to maintain the highest accuracy.

Impacts on Accuracy We now investigate the impact of *D-Pruner* on the final accuracy. Figure 4.3 shows the accuracy of pruned models during multiple pruning

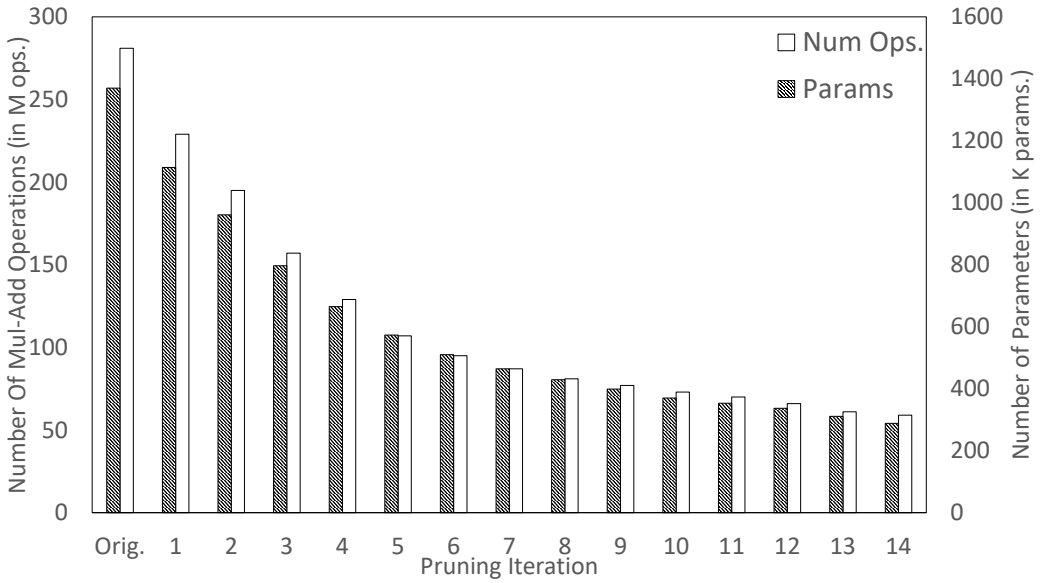


Figure 4.4: Parameters and Operations Reduction on CIFAR-10

iterations on both CIFAR-10 and CIFAR-100. We achieve accuracy of 89.34% and 61.08% comparing to 90.19% and 61.71% from original models after 13 and 6 pruning iterations on CIFAR-10 and CIFAR-100 respectively.

Firstly, we notice that it takes us 14 and 7 iterations to make the accuracy loss above 1% threshold on CIFAR-10 and CIFAR-100. This implies that the original models tend to have a lot of redundancy and *D-Pruner* can effectively prune them without significant loss in the final accuracy.

Secondly, we figure out that accuracy increases for the first few iterations which indicates that the some redundancy negatively affects the accuracy. Hence, *D-Pruner* can be used to slightly improve accuracy by eliminating most negatively redundant parameters.

Finally, we also want to note that the pruning process converges faster on CIFAR-100 than CIFAR-10. As we use same architecture on both tasks, it is understandable that classifying 100 classes requires more network capacity in terms of filters and parameters than classifying 10 classes.

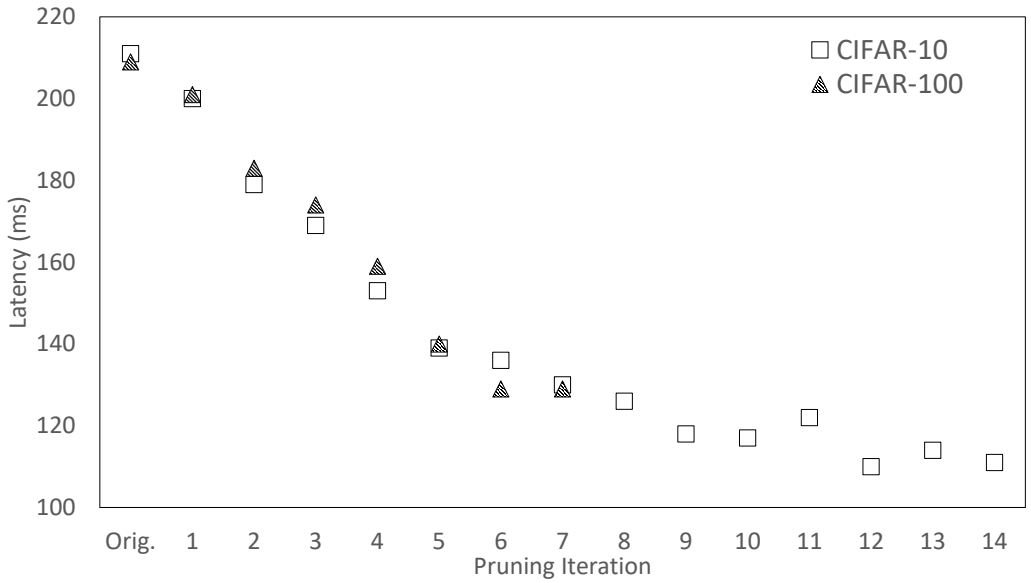


Figure 4.5: Latency

4.4.3.1 Impacts on Parameters and Operations

Next, we investigate on how many parameters and number of operations *D-Pruner* can prune during each iteration. Figure 4.4 shows that both the number of parameters and operations gradually decrease during pruning process. At 13th iteration, we achieve 4.4× and 4.57× reduction in number of parameters and Mul-Add operations on CIFAR-10. As *D-Pruner's* optimization is to reduce the number of filters during each pruning iteration, both parameters and number of operations would always decrease during the pruning process.

Similarly, we also see the same trend on CIFAR-100 dataset which results in 2.76× and 2.9× improvement on model's parameters and number of Mul-Add operations.

4.4.3.2 Impacts on Latency

We also explore the performance of pruned models on existing mobile deep learning frameworks. Figure 4.5 shows the latency per pruning iteration on both datasets using DeepMon framework. We achieve the speedup of 1.85× and 1.61× on CIFAR-10 and CIFAR-100 respectively.

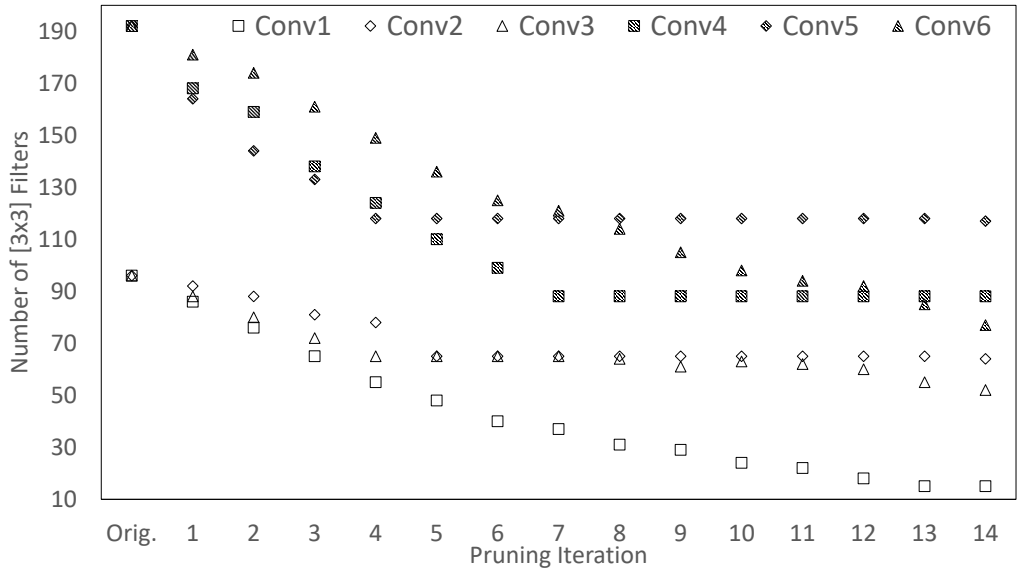


Figure 4.6: Number of Filters per Iteration on CIFAR-10

However, we notice that the latency slowly decreases after 9th iteration. One reason is that number of operations per layer become too small to make DeepMon utilize GPU resources efficiently. However, batching multiple images as input could improve the average inference time.

4.4.3.3 Impacts on Number of Filters

Finally, we investigate the reduction of filters during the pruning process on CIFAR-10. We plot the amount of filters within two first blocks in ANN-CNN-C model which consist the first 6 convolutional layers as shown in Figure 4.6. Each block consists of 3 convolutional layers which have 96 and 192 filters respectively. Both blocks end with spatial dimension reduction using convolutional layer with stride is set to 2 instead of using Max-Pooling.

Interestingly, two blocks share the same trend in filters reduction. The first and last convolutional layers within the block stop reducing after a certain threshold while the middle layer keeps reducing during pruning process. This shows some insights for us the build better network architecture where last layer inside a block should have few filters comparing to previous layers.

4.4.3.4 Comparisons with DeepIoT

We compare our pruned models with compressed VGGNet from *DeepIoT* [87] on CIFAR-10 dataset as shown in Table 4.2. At 4th iteration, *D-Pruner* provides a model with 8% less parameters than *DeepIoT's* model while achieving comparable accuracy (90.48% vs 90.5%), even though we start with less accurate model. If we are willing to sacrifice 1.16% (comparing to *DeepIoT*), we will achieve 2.33× smaller model.

We also notice that *DeepIoT* leverages recurrent neural network (RNN) to prune the parameters. However, RNN is prone to gradient vanishing problem and may not work well in very deep neural network such as ResNet or Inception network. Instead, *D-Pruner* 's masking blocks can be easily integrated into CNN and can be trained at ease.

Chapter 5

Exploiting Cost-Quality Trade-off with Multi-Exit Networks

In recent years, cameras have become ubiquitous with billions of them deployed on personal smartphones, in public and private spaces such as traffic intersections, organizations, etc. As deep learning has shown huge success in yielding state-of-the-art performance in many computer vision tasks, video analytics systems have adopted deep learning models to improve overall performance. However, pre-trained state-of-the-art models often use a fixed computational pipeline for every inference without any considerations whether the input is easy or not. In this work, we did an intensive study of how multi-exit models (*MXNs*) can be used to accelerate variety of machine learning workloads and which techniques can be applied to improve their efficacy.

We evaluated multi-exits models and their optimization techniques on two real applications including indexed video querying and object re-identification in video-based recognition. Our results show that *MXNs* reduce the latency of current existing systems up to 4.4× in video query system and 1.29× in video-based face recognition system with minimal loss in accuracy.

5.1 Introduction

The popularity of cameras has enabled many vision sensing applications such as video query system that allow users to seek for interesting moments. Those applications often have to run a pipeline of computer vision algorithms on multi-hour-length videos to provide the information to users. Hence, minimizing user's waiting time is crucial.

The current state-of-the-art approach for video processing is to apply pre-trained deep neural network models on video frames. Advances in deep learning algorithm, especially deep convolutional neural networks (CNNs), have significantly boosted the accuracy of many computer vision tasks such as object detection, image classification. For example, EfficientNet [82] has reduced the error rate of image classification task on Imagenet dataset up to 2.67$\times$ comparing to the famous Alexnet model [52].

Despite having high accuracy, state-of-the-art models such as EfficientNet [82], ResNet152 [36] and Yolov2 [71] require huge amount of computational capability, making it inefficient to use on lengthy videos. Hsieh at el. has shown that continuously running Yolov2 object detector on a month-long video costs over 380$ on Azure cloud using a high-end GPU [39]. Many works have adopted a cascade of multi-models approach to speedup inference time by using a computational-efficient but less accurate model along with a computational-intensive but highly accurate model. Recently, video query system NoScope [48] leverages a low-cost binary classification model, which is trained to recognize only queried class using a small segment of the video as training dataset at run-time, to classify the remaining video frames. NoScope only triggers the big but accurate model on video frames that the low-cost classifier does not provide confident results. Other works [35, 75] exploit the skewed class distributions over time, track the changes in the dominant classes and train a lightweight specialized model at run-time to recognize only a set of dominant classes in order to speedup the inferences. Whenever a specialized

PASCAL VOC2012 Dataset					
Model	MFlops	Easy Objects		Hard Objects	
		Object 1	Object 2	Object 3	Object 4
Model 1	567	94.15	93.47	50.95	62.30
Model 2	437	85.71	93.17	34.32	60.95
Model 3	333	86.68	86.35	16.84	37.61

Table 5.1: Accuracy of classifying easy/hard objects

model recognizes classes that are not in the skewed set via a special "other" class, the system will trigger a general model to get the correct result. By not triggering high-cost models, previous systems can save unnecessary computation and reduce the incurred latency. However, there are two drawbacks with this approach. First, if low-cost models are not confident about their results, the system has to trigger bigger and more expensive models. In this case, all computations spent on former models is wasted and the efficiency of cascaded approach is significantly reduced if heavy models are triggered frequently. For example, if we use three classification models as a cascade to recognize 20 objects in VOC dataset [26] as shown in table 5.1, there are chances that model 2 and 3 will fail to provide correct results on class 3 and we have to fallback on using model 1 for the final outcome. In this case, approximately 770 Mflops spent on model 2 and 3 are wasted and will be treated as an overhead of the system. Second, training a cascade of models is time consuming as we need to train multiple models separately. For instance, training ResNet-50 on a single NVIDIA M40 GPU takes up to 14 days [90]. As the number of models and the complexity of each model increase, the cost of training a cascade would be significantly high.

In this work, we adopt the idea of multi-exit models (*MXNs*) by attaching classification or regression layers along a existing network backbone (e.g., ResNet, Mobilenet) to generate multiple cheap and expensive models [83, 41]. This approach allows computations, which have been done at early classification/regression layers, to be reused to compute latter layers with a small amount of overhead. Moreover, *MXNs* help us build multiple cheap and expensive models with different complex-

ity and accuracy at a single training cost. At early stages, models in *MXNs* only rely on low-level features, which require little computations to compute, for classification/detection tasks. Hence, early models often perform well on easy objects but fail to detect complex instances. As the depth and complexity increases, latter models in *MXNs* become more accurate in detecting complex objects. Table 5.1 shows the accuracy of three models with different computational cost on four easy and hard objects within VOC dataset [26]. There is an accuracy gap between recognizing easy and hard objects across all three models. Second, as the capacity of the model decreases, accuracy on hard objects drops significantly while accuracy on easy objects only decreases slightly. These observations support the idea of using cheap models to recognize easy objects while using heavy models only on hard cases. Similar to a cascade of models, processing *MXNs* is done by executing the models in a sequence, from cheapest model to most expensive one. During a *MXNs'* processing pipeline, if we are satisfied with the results, we can stop the execution immediately at any model without wasting computation on more expensive models. However, the major difference between *MXNs* and a cascade of models is that intermediate features, which are computed from early models, can be reused to compute latter models due to parameters sharing feature.

In this work, we study two key questions when using multi-exit models. 1) When MXNs are applied to problems beyond anytime inference, and become drop-in replacements for DNNs? How do they perform in terms of training, inference, indexed inference, similarity matching, etc? 2) If they are inadequate for these purposes, can they be improved by other optimizations?

We conduct experiments using *MXNs* on many general tasks such as image recognition and face recognition to understand the performance of *MXNs*. Then, we propose two key techniques to optimize for applications using *MXNs*: 1) using focal loss [60] to improve the accuracy of early exiting decision, 2) aggregating results across multi-models and early exiting based on confidence score to improve performance of indexed video querying and object re-identification in video-based

recognition systems.

(1) *Improving accuracy of confidence-score-based systems.* Conventional approach to check how confident a model is to its result is to look at the confidence score. For example, Yolov2 [71] uses confidence score to filter out uncertain predictions that may lead to incorrect results. However, given a particular threshold, it is hard to determine if the result with confidence score that is higher than predefined threshold is correct. In the case of using *MXNs*, we need our models to be highly accurate at deciding whether to exit a computation or to let the latter and more accurate models handle current input.

To improve above mentioned accuracy, we adopt Focal Loss technique [60] to make a model more conservative when outputting a result. Comparing with the traditional softmax cross entropy loss, *MXNs* trained with Focal Loss gives better early exiting decisions given a arbitrary threshold.

(2) *Improving performance of video query system via aggregation of predictions across* MXNs' *models.* State-of-the-art video query system Focus [39] separates the processing pipeline into two steps, ingestion and query step. At ingestion step, the system leverages a cheap model, which is specialized to recognize majority of objects that appear in the target video, to index incoming video frames. At query step, Focus retrieves the frames associated to user's query label and use high computational cost but accurate ground-truth CNN model to classify them. By separating into two steps, Focus can balance the latency cost between ingestion and query time. Furthermore, in order to guarantee high recall at ingestion time, Focus leverages top-K predictions from the cheap model to index video frames instead of using only the prediction with highest probability. However, if we use high "Top-K" value at ingestion step, we need to process and store more indices into a database and eventually affect the performance at query time as we need to trigger ground-truth CNN model on a large number of video frames.

Instead of relying on high "Top-K" results of a single model as in Focus, we replace that single model with *MXNs* and aggregate predictions of multiple models

within *MXNs*. We use lower "Top-K" at each model and remove duplicated predictions within aggregated results. As multiple models are likely to agree on a same correct answer, it will significantly reduce number of indices we need to ingest into database. In case of disagreement, it will automatically fallback to use higher "Top-K" value. For example, if we have two models and "Top-K" is set to one, agreement between two models results in only one index while disagreement will result in two indices. This approach considerably reduces the latency at both ingestion and query time as we can reduce the number of video frames that we need to process, store and trigger big ground-truth CNN model on.

(3) *Improving performance of video-based face recognition systems via early facial features matching from* MXNs . Existing works [35, 75] use a cascade of multiple models to speedup the inference latency of video-based face recognition task. In this work, we treat that problem as an object re-identification task in video-based system and propose an approach by adopting *MXNs* to accelerate the inference process. We use *MXNs* of face verification models, trained on general face dataset such as Casia Webface [88], with different levels of accuracy to match incoming input with existing faces within our database. If an early model is not confident about its face matching output, the system triggers the next model in *MXNs* with only minimal amount of overhead. If a new face is not existed in the database, our system will trigger an oracle model (e.g., a separated model or the last model within *MXNs* that is trained as a face classifier) to get the label, store it and all facial features extracted from our *MXNs* into the database for further face verification requests. By leveraging *MXNs*, not only do we allow most of computations to exit at early models on easy samples to reduce the inference latency but also minimize the overhead of switching between models.

We evaluate *MXNs* by applying the idea to improve performance of existing video sensing systems such as video query system Focus [39] and propose a new approach to improve video-based face recognition sytem. By using optimizations based on *MXNs*, our approach accelerates up to 4.4× and 1.29× improvements

over two existing applications including video query systems and face recognition in videos accordingly. In summary, we make the following contributions:

- We adopt multi-exit models as an efficient implementation to enable flexible computational pipeline for many computer vision workflows such as image classification, face recognition.

- We show that by using *MXNs*, we can train and serve shared-models faster than training and serving catalog implementations based on many distinct models.

- We propose using 1) threshold-based approach with *MXNs* to do early exiting efficiently, 2) aggregation of results from multiple models within *MXNs* to achieve accurate predictions, 3) Focal Loss to improve the accuracy of early-exiting decisions,

- We evaluate the idea of *MXNs* on two existing applications and show that adopting *MXNs* and *MXNs'* optimizations helps accelerate existing systems. We achieve up to 4.4× and 1.29× in latency reduction in video query and video-based face recognition systems.

5.2 Multi-Exit Model Overview

Figure 5.1 illustrates an example of multi-exit models (*MXNs*) that consists of multiple exits instead of a single one (i.e., exit 3), which is normally seen in typical neural network models. Each model (or exit) within *MXNs* uses different sets of features from many shared layers for its classification/regression output. For instance, exit 1 and 2 share the first three convolutional layers to extract low-level features. If we have already computed the output of exit 1, the features extracted from the third convolutional layer would be instantly available to compute the forth layer without the need of recomputing the features again from the first layer.

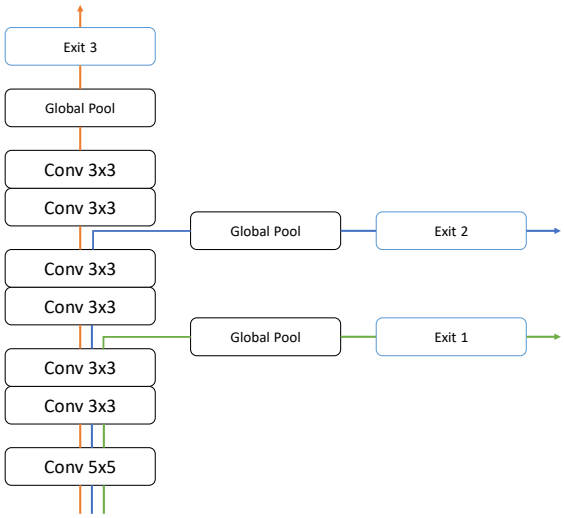


Figure 5.1: Example of *MXNs* architecture

Similar to previous works [83, 41] on anytime neural network, we use loss functions such as cross entropy loss function L_{f_i} for all the classifier/detector using features f_i extracted from layer i. We adopt weighted sum of loss across all classifiers/detectors on training dataset D as shown in BranchyNet [83]: $L = \frac{1}{|D|}\sum_{(x,y)\in D}\sum_{l=1}^{N} w_i * L_{f_i}$. Herein, w_i is the constant weight of each classifier i in *MXNs* , set by users before training, to trade-off between accuracy and computational cost across multiple models (or exits). For example, we can assign high w_i to early exits to focus more on those early classifiers during the training process.

5.2.1 Overall performance of multi-exit models on general tasks.

In this session, we evaluate the idea of *MXNs* on two general tasks including image classification and face recognition without any further optimizations. For all the experiments, we use same configuration to train *MXNs* and multiple single-exit models. We use SGD optimizer with momentum set at 0.9. The learning rate is set at 0.1 and will be reduced by 10 whenever the error plateaus. We also use early stopping to stop the training process instead of having fixed number of training iterations.

Exit Index	Accuracy (%)	
	Single-Exit Model	*MXNs*
1	47.84	45
2	57.82	55.76
3	61.16	59.52
4	66.25	64.51
5	66.44	65.68

Table 5.2: Accuracy of Image Recognition task

(a) Image Classification - Training Time ($\times$1000s)

Single-Exit Models					*MXNs*
Exit 1	Exit 2	Exit 3	Exit 4	Exit 5	
56.66	86.4	106.27	106.61	82.86	102.01

(b) Image Classification - Training Speedup

Training Time of 5 models (s)	Training Time of *MXNs* (s)	Speedup
521702	102016	5.11

Table 5.3: Training Time and Speedup of Image Recognition Task

First, we train *MXNs* for image classification task based on ResNet-18 architecture [36] on ImageNet dataset [22] to recognize 1000 objects. We choose 5 layers to add classification layers on top of them and train *MXNs*. Similarly, we train 5 separated models based on 5 chosen layers for comparison. Table 5.2 reports the validation accuracy of *MXNs* of 5 models and 5 separated single-exit models. Overall, naively training *MXNs* results in lower accuracy at each exit comparing to corresponding single-exit model because of the inferences between classifiers using low and high level features as explained in [42]. The accuracy loss ranges from 0.75% to 2.84%. However, it is worth pointing out that although these results hold for the hard 1000-class ImageNet dataset, as table 5.2 shows, for many day-to-day problems that are much easier than ImageNet, the gap may be even smaller (i.e., less than 0.75% or 2.84%).

Moreover, completely training *MXNs* of 5 models is 5.11x times faster than training 5 separated models as shown in table 5.3. This significantly reduces the time we need to search for efficient models that fit to our particular goals of many different applications.

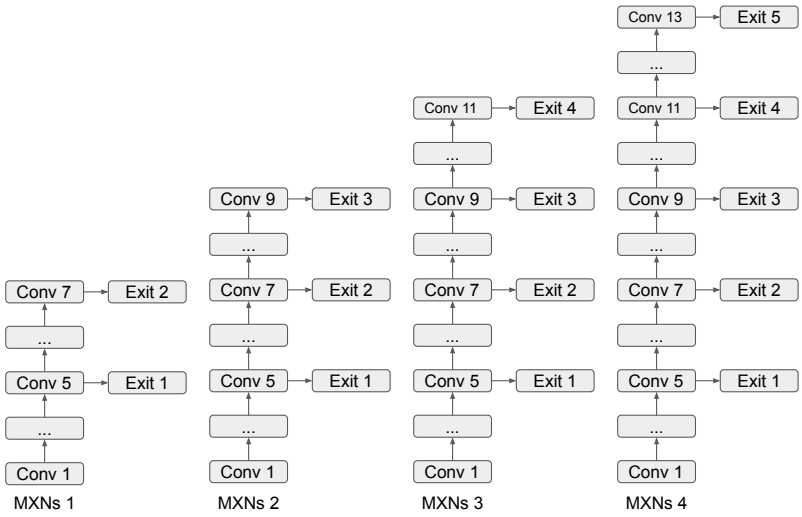


Figure 5.2: MobiletNet-based *MXNs* for Face Recognition Task

Second, we also train 5 single-exit models and *MXNs* of 5 models with different number of exits for face recognition task using mobilenet-based architecture on Casia-Webface dataset [88] to learn facial features and compute the accuracy of face recognition task on LFW dataset [43]. We specifically choose 5 layers from MobileNetv1 including Conv3, Conv5, Conv7, Conv9, Conv11 to train both single-exit models and multiple *MXNs*. For single-exit models, we attach a classification layer to one of 5 chosen convolutional layers and train a model with only one classifier. For *MXNs* , we train a set of 4 *MXNs*with 2 up to 5 classification layers from 5 chosen layers as shown in figure 5.2 in order to study the performance of different *MXNs*. We use the input size of 152x152 and the training configuration similar to previous one used in training image recognition task.

Table 5.4 shows that the accuracy of *MXNs* is within 1.5% different with single-exit models. Table 5.5(a) shows the training time in seconds of each model. In most cases, training a single-exit model is faster than training *MXNs*. However, as shown in table 5.5(b), when we aggregate the training time of multiple single-exit models and compare it with the training time of corresponding *MXNs*, training *MXNs* is always faster, from 1.61x up to 4.14x faster when the number of exits increases

Face Recognition - Accuracy (%)					
	Single-exit Model	*MXNs*			
Exit Layer		2 Exits	3 Exits	4 Exits	5 Exits
Conv 3	89.58	**89.66**	89.43	88.41	88.15
Conv 5	**96.16**	95.93	95.78	95.6	95.78
Conv 7	98.00	-	97.41	**98.20**	97.86
Conv 9	98.1	-	-	97.85	**98.13**
Conv 11	**98.36**	-	-	-	98.06

Table 5.4: Accuracy of Face Recognition Task

(a) Face Recognition - Training Time ($\times$1000s)								
Single-Exit Models					*MXNs*			
C3	C5	C7	C9	C11	2 Exits	3 Exits	4 Exits	5 Exits
58.77	58.90	121.03	57.13	58.28	72.92	73.98	71.43	87.88

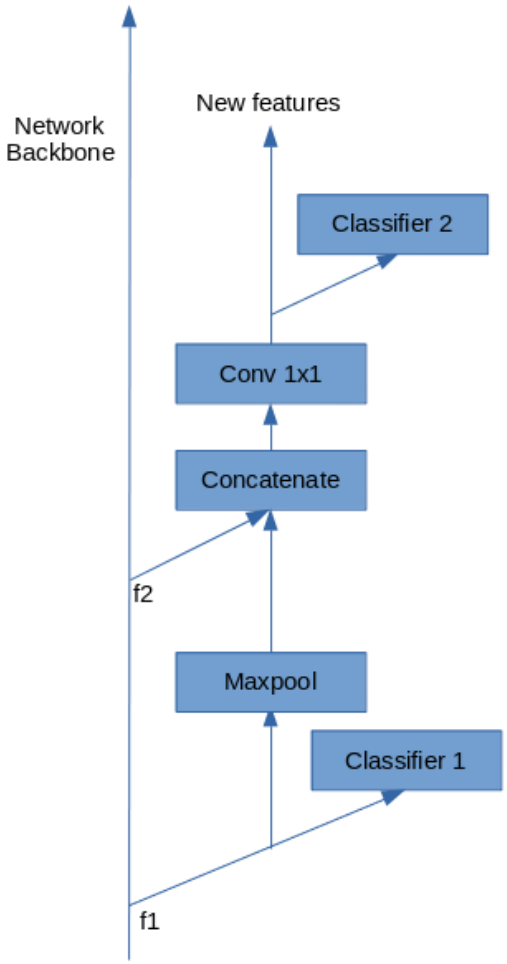
(b) Face Recognition - Training Speedup				
	2 Exits	3 Exits	4 Exits	5 Exits
Speedup	1.61	3.22	4.14	4.02

Table 5.5: Training Time and Speedup of Face Recognition Task

from 2 to 5 respectively.

5.2.2 Enhancing Accuracy of *MXNs* via Features Aggregation Between Exits

Previous works [41] have shown that there is interference between lower and upper classification/regression layers. In order to mitigate that problem, we adopt the technique in [41] which forwards the lower-level features to upper layers in a residual approach. This technique allows latter exits make concrete predictions based on both low-level and high-level features. However, instead of aggregating features in densenet-style [42] similar in [41], we directly aggregate low-level and high-level features using concatenation layers, following a 1×1 convolutional layer to compress those features into lower dimension to minimize additional computational cost. In case low-level and high-level features have different size, we apply max-pool layer on low-level features to reduce its size. Figure 5.3 shows the building



f1, f2: features extracted from backbone network

Figure 5.3: Aggregation Between Low-level and High-level features Block

block that we use to improve *MXNs* by aggregating features.

In order to demonstrate the effective of feature aggregation, we use three datasets including VOC Pascal [26], Coco [61] and ImageNet [22] to train *MXNs* to classify 20, 80 and 1000 objects respectively. We use ResNet18 [36] as a base network architecture to train those *MXNs* , each with 4 classification layers. Table 5.6 shows that aggregation between low-level and high-level features improves the overall accuracy of proposed *MXNs* on both VOC, Coco and ImageNet datasets, especially latter classifiers, which can re-use lower-level features to improve the results. For instance, this approach improves accuracy of every classifier in *MXNs*

(a) VOC Dataset - Accuracy (%)

Model	Classification layer			
	1	2	3	4
Base *MXNs*	60.03	71.0	73.66	77.68
Improved *MXNs*	62.43	74.50	75.97	79.16

(b) Coco Dataset - Accuracy (%)

Model	Classification layer			
	1	2	3	4
Base *MXNs*	47.59	57.24	58.87	65.41
Improved *MXNs*	47.58	60.96	63.33	68.86

(c) ImageNet Dataset - Accuracy (%)

Model	Classification layer				
	1	2	3	4	5
Base *MXNs*	45	55.76	59.52	64.51	65.68
Improved *MXNs*	45.69	58.88	62.88	67.64	67.93

Table 5.6: Effect of Feature Aggregation on *MXNs*

Exit Index	Accuracy (%)	
	Single-Exit Model	Multi-Exit Model
1	**47.84**	45.69
2	57.82	**58.88**
3	61.16	**62.88**
4	66.25	**67.64**
5	66.44	**67.93**

Table 5.7: Comparison between Enhanced Multi-Exit Models and Single-Exit Model

on VOC dataset from 1.48% up to 3.5%. On Coco dataset, it also increases up to 4.46% at latter layers while maintaining the similar accuracy at the first classifier to original *MXNs* . Even on complex dataset such as Imagenet, the proposed approach still outperforms original *MXNs*, ranging from 0.68% up to 3.36%. Furthermore, table 5.7 shows the accuracy comparison between our enhanced *MXNs* and single-exit models as shown in table 5.4. We notice that enhanced *MXNs* outperform single-exit models from 1.06% to 1.49% on 4 upper exits where features aggregation occurs. This indicates that using features aggregation not only helps improve the accuracy of *MXNs* but also closes the accuracy gap between single-exit and multi-exit models.

5.2.3 Improving Accuracy of Threshold-based approach using Focal Loss.

One of the challenges when using multi-exit models is to determine when to stop the execution. A conventional approach to make that decision is to use a confidence score threshold at each model [48]. If a confidence score is above the threshold, we will stop the execution. Otherwise, we will go to upper exit. As the overall accuracy of multi-exit models depends on the accuracy of early-exit decision, we want to make decisions that are highly accurate. If the early exit does not provide confident result, we will use upper and more accurate exit to maintain overall accuracy. However, it is challenging to determine if the result with a confidence score above a given threshold is correct or not. In this session, we introduce a method to improve the accuracy of such decisions by applying focal loss [60] instead of traditional cross entropy loss during the training process.

Focal loss is defined as: $FL(p_t) = -(1-p_t)^{\gamma} log(p_t)$ where p_t is the probability of being a correct class of input t and γ is a constant value to control the rate how much weight easy inputs are reduced. The modulating factor $(1-p_t)^{\gamma}$ controls how much weight we will add to a particular input during the training process. When the model misclassifies an input t, $p(t)$ will be small, the modulating factor will be near 1 and the loss will be unaffected. However, if the model classifies an input correctly with high probability $p(t)$, the modulating factor will be very low and the loss will be down-weighted. In this way, focal loss focuses more on hard cases during the training process and only gives high confidence score to easy instances.

We conduct experiments by training two small models based on ResNet18 on two datasets to show the benefits of using focal loss. We define precision as the percentage of instances that correctly exit and recall as the percentage of instances that exit. Table 5.8 shows the precision/recall of focal loss comparing to traditional cross entropy loss on 2 datasets (i.e., VOC, Coco) with two predefined thresholds. Given a arbitrary confidence score threshold, using focal loss always provides higher pre-

(a) VOC Dataset - Threshold Score 0.5

	Precision				Recall			
Model	C1	C2	C3	C4	C1	C2	C3	C4
ResNet18	79.77	84.02	83.07	84.05	50.86	70.26	79.25	87.33
ResNet18-FL	84.69	88.72	87.55	88.16	32.41	58.29	67.7	77.29

(b) VOC Dataset - Threshold Score 0.7

	Precision				Recall			
Model	C1	C2	C3	C4	C1	C2	C3	C4
ResNet18	89.41	92.13	90.46	90.46	26.12	48.8	59.08	71.95
ResNet18-FL	92.76	95.95	95.72	95.21	8.49	32.61	43.08	57.11

(c) Coco Dataset - Threshold Score 0.5

	Precision				Recall			
Model	C1	C2	C3	C4	C1	C2	C3	C4
ResNet18	75.98	80.54	79.79	82.16	37.39	52.3	56.54	67.31
ResNet18-FL	82.48	86.51	86.03	87.43	24.12	43.86	49.86	59.54

(d) Coco Dataset - Threshold Score 0.7

	Precision				Recall			
Model	C1	C2	C3	C4	C1	C2	C3	C4
ResNet18	86.94	90.46	89.78	90.62	20.59	34.07	38.58	50.26
ResNet18-FL	92.11	94.97	94.48	95.21	7.27	23.84	29.31	39.79

Table 5.8: Effect of Focal Loss on *MXNs*

cision than normal softmax cross entropy in the trade-off for lower recall. In the scenario of multi-exit models, it is beneficial to achieve high precision to make sure the model only exits when the result is correct. Otherwise, the more accurate models will handle that input.

5.2.4 Accelerating models serving using prefix batching

In this session, we investigate benefit of *MXNs* in terms of serving latency. We setup an experiment in which we use 50,000 validation images from ImageNet dataset [22] to measure the latency of getting result from all exits using both *MXNs* and a set of multiple single-exit models. We use ResNet18-based image classification models from table 5.2 and measure the total latency using Tensorflow framework [8] on a single NVIDIA Geforce GTX 1080Ti GPU.

Table 5.9 reports the best latency spent on each single-exit model and *MXNs*

Serving 50k images using Image Classification Models							
Latency (s)							
Single-Exit Model						*MXNs*	Speedup
M1	M2	M3	M4	M5	5-Models	5-Models	
48.68	54.18	59.67	65.84	68.02	296.4	76.1	3.89

M1, M2, M3, M4, M5: Model 1, Model 2, Model 3, Model 4 and Model 5
Speedup is computed between serving 5 single-exit models and *MXNs* .

Table 5.9: Serving Latency

using batch size of 32 images. Overall, serving user's requests using *MXNs* is 3.89x faster than executing a set of 5 separated models. As multiple models within *MXNs* can share some layers (or prefix) with each other, we only need to compute those layers for a batch of multiple images once per request. We call it a prefix batching technique. On the other hand, because multiple single-exit models don't share any parameters or layers, we will need to process every layer in all models for each single batch. Without the prefix sharing, it is significantly slower to process every single-exit models comparing to *MXNs* .

5.3 Evaluations on Real Applications

We evaluate *MXNs* approach on two real applications including indexed video querying and object re-identification in video-based recognition.

We run all the experiments on a machine with a Quad-core Intel i7 7700, a NVIDIA GeForce GTX 1080Ti GPU and 64 GB of Ram. We use Tensorflow as a framework to train and evaluate *MXNs* in our experiments.

5.3.1 Video Query System

In this session, we demonstrate the effect of multi-exit models on existing video query system Focus [39]. Focus is a video query system that allows users to seek for some particular objects within a single or multiple videos. The optimal solution to minimize the user's query time is to use highly accurate model to index all video

frames into a database. However, Hsieh at el. argues that if only a small fraction of frames are queried, most of the computations used for indexing the remaining frames would be wasted. He proposed Focus system, in which he separates the processing pipeline into two steps including ingestion step and query step, to balance time spent on indexing and querying. Figure 5.4 shows the overview of Focus system. Focus defines precision as the percentage of frames in the query's results that contains queried object and recall as the percentage of frames that actually contains queried object and the total frames in video that contains queried object. The goal of Focus is to achieve 99% of both precision and recall.

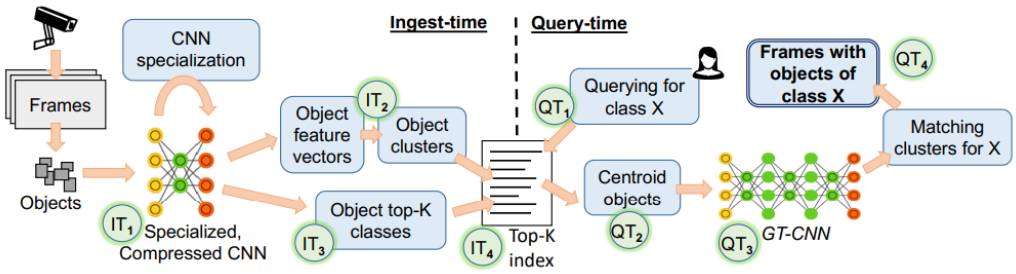


Figure 5.4: Focus Architecture [39]

At ingestion step, Focus uses very fast background subtraction technique to swiftly filter out frames that do not contain interesting objects. In order to further reduce ingestion latency, Focus leverages small specialized models, which are trained to classify majority of objects in a target video, to process interesting frames and index which objects appear in a database. However, lightweight models often have low accuracy both in terms of precision and recall. In order to compensate low accuracy of cheap models, Focus uses an empirical observation that correct label often falls into top-k confident results of a cheap model and indexes incoming frame using top-k results instead of relying only on the top confident result. This approach increases the chance that correct result will be indexed into the database and improves the final recall. Furthermore, to reduce the work at query time, Focus relies on a clustering algorithm to cluster similar objects into a single group using features extracted from cheap model and stores the centroid and its object members

into the database. Overall, for each frame, Focus needs to extract k labels from a cheap model and compares each label to N clusters' centroids in a database. Thus, the overhead at ingestion step is $O(Nk)$. If we can replace the cheap model with *MXNs* in such way that k can be reduced, we can improve the ingestion latency.

At query time, Focus uses a large but accurate model as a ground-truth CNN (GT-CNN) to re-classify indexed frames. First, Focus retrieves all the centroids whose labels are similar to queried object and runs GT-CNN on those centroids to filter out incorrect centroids from ingestion step. Second, Focus retrieves the all clusters' members associated with correct centroids and return them to users as final outputs. In general, query time depends on the number of centroids (or clusters) that the system needs to trigger GT-CNN on. However, as the number of clusters increase significantly over time, query time would be severely impacted. By adopting *MXNs* to filter out easy centroids, we can limit the number of times to trigger GT-CNN and further reduce the overall query latency.

5.3.1.1 Reduce Top-K by Aggregating Results Across *MXNs*

For each input x, Focus infers k possible classes $c_1, ..., c_k$. Each class input-class association (x, c_i) is inserted into a distinct cluster-set C_i. The cluster-set C_i for each class i, contains up to N=100 clusters. Thus, indexing overhead is $O(Nk)$. Focus typically uses a k ranging from 2 through 6. Reducing k thus provides a possible way to speed up the ingesting step.

In order to reduce the k value used in Focus, we replace Focus's specialized model with our *MXNs* of n models $M_1, ..., M_n$. At ingestion time, each model M_j infers top $k' \leq k$ classes $c^j_1, ..., c^j_{k'}$. Those results are then aggregated into a set $R = \{c^1_1, ..., c^1_{k'}, ..., c^n_{k'}\}$. As each model classifies the input independently, there is highly chance that one of those models infers correct label even if we use smaller k value. Furthermore, as models tend to agree on correct results, there would be many duplicates in R that can be removed to reduce the total objects that we need to index into a database. For example, many models can agree with each other on easy

	Focus Model			*MXNs*		
Video	Top-k	Exit	#indexing	Top-k	Exits	#indexing
Auburn	1	4	16724	1	0,1	16823
Auburn North Ross	2	4	23746	1	0,1	12003
Bellevue 150th Road	1	2	1365	1	0,1	1374
Bellevue Ne8th Road	1	2	4933	1	0	4933
Jackson Hole	2	4	53744	1	0,1,2,3	31157
Jackson Town	6	4	4410	3	0,1,2,3	3346
Lausanne	3	2	462045	2	0,1,2	314411
Sittard	2	2	45240	1	0,1,2,3	23712

#indexing: number of times to trigger clustering algorithm
Exit(Exits): index(indices) of exits that is(are) used as classifier
Network architecture: ResNet-18 with 5 chosen layers to be exits for both Focus model and *MXNs*

Table 5.10: Effect of Results Aggregation

samples while they can disagree with each other on hard samples. This approach leverages the agreements and disagreements between models in *MXNs* to lower the total number of indexing over entire video frames.

Table 5.10 shows the configurations of Focus and our approach on 8 traffic videos used in [39]. Overall, using *MXNs* can reduce the top-K value and number of images that need to be indexed to the database in 5 videos while maintaining up to 99% recall, similar to Focus system. For example, the original model in Focus system uses the full ResNet-18 and top-2 results on Auburn North Ross video to achieve 99% recall while *MXNs* of 2 models (at lower layers) can reduce from top-2 to top-1 by using results from both exits. Hence, *MXNs* can reduce the number of images that need to be indexed to the database by 1.97 times. Furthermore, *MXNs* also allow use to search for more efficient models while achieving similar recall on Auburn, Bellevue 150th Road and Bellevue Ne8th Road videos. By attaching early exits into existing models used in Focus, we find that aggregating results from those early exits also achieves 99% recall while having lower computational cost.

Table 5.11 shows the ingestion latency and speedup on 8 traffic videos. Overall, using *MXNs* is faster than using single-exit model in all 8 videos, ranging from 1.06x to 1.41x. Specifically, we can perform faster clustering algorithms on 6 videos (i.e.,

Video	Focus Latency(s)	*MXNs* Latency(s)	Speedup	Model-only Speedup
Auburn	32.3±0.38	22.89±0.18	1.41	1.71
Auburn North Ross	29.79±0.56	23.83±0.21	1.25	1.11
Bellevue 150th Road	3.82±0.19	3.43±0.2	1.11	0.99
Bellevue Ne8th Road	9.13±0.28	7.58±0.18	1.2	1.37
Jackson Hole	82.03±0.69	77.53±0.23	1.06	0.87
Jackson Town	6.93±0.19	5.53±0.21	1.25	0.91
Lausanne	321.2±0.54	271.78±0.53	1.18	0.9
Sittard	58.79±0.13	52.2±0.25	1.13	0.95

Table 5.11: Ingestion Latency

Auburn North Ross, Bellevue 150th Road, Jackson Hole, Jackson Town, Lausanne, Sittard), by lowering the top-K value to reduce the number of indexing times. For 2 remaining videos, *MXNs* helps us find smaller and more efficient models comparing to the models used in Focus to improve the latency.

We also noticed that running *MXNs* is slower than running a single bigger model in Bellevue 150th road, Jackson Hole, Jackson Town, Lausanne and Sittard videos as shown as Model-only speedup in table 5.11. The problem occurs due to the overhead of partial run operation in Tensorflow which allows us to reuse computed variables during *MXNs'* execution. However, there is a small overhead between each pause (e.g., getting current exit's result) and resume (e.g., execute next model) during *MXNs'* execution. Fortunately, this overhead only contributes little to the total latency as a lot of time was spent on clustering algorithm. Overall, we still achieve speedup, ranging from 1.06× to 1.41×, across 8 videos.

5.3.1.2 Reduce Query-Time by Co-Processing *MXNs* and Ground-Truth Model

At ingest time, Focus uses a ground-truth CNN model (GTCNN) to classify the centroids of clusters and return results to users. However, if the number of clusters is too high, we have to trigger GTCNN a lot and increase the query latency. Instead of running the GTCNN on all cluster's centroids, we use extra *MXNs*, which are trained with focal loss, to early accept results using strict confidence score thresholds. In

Video	Focus	*MXNs*		Speedup
	Latency (s)	Latency (s)	Precision (%)	
Auburn	21.61±0.12	5.67±0.23	99.01	3.81
Auburn North Ross	29.42±0.16	18.93±0.28	98.71	1.55
Bellevue 150th Road	2.66±0.05	0.6±0.16	99.26	4.43
Bellevue Ne8th Road	2.37±0.06	1.35±0.21	99.4	1.76
Jackson Hole	57.06±0.22	38.68±0.21	99.8	1.48
Jackson Town	18.03±0.14	20.78±0.21	98.9	0.87
Lausanne	3.16±0.1	0.96±0.14	99.01	3.3
Sittard	88.49±0.23	29.55±0.25	99.89	2.99

Table 5.12: Query Latency - Threshold explored on Coco Dataset

order to find out the thresholds, we use Coco dataset [61], which is used to train GTCNN or Yolov2, to search for the average thresholds that achieve 99% of true positive rate.

Table 5.12 shows the overall performance of applying *MXNs* into Focus system in terms of latency and precision. Our *MXNs*-based approach improves the query latency from 1.48x to 4.43x while preserving nearly 99% precision rate on 7 videos similar to Focus system. However, we don't improve the latency on Jackson Town video as it contains the most number of objects to recognize among 8 videos. In order to achieve 99% of true positive rate on multiple objects in Coco dataset, we end up using very high thresholds given a limited amount of computational capacity of ResNet-18. Hence, we often need to trigger the GTCNN to achieve such accuracy.

Table 5.13 shows the breakdown of number of times to trigger the GTCNN during *MXNs'* execution. We only need to trigger the GTCNN on less than 25% and 50% of cluster's centroids in 4 and 7 over 8 videos respectively. As we mentioned above, Jackson Town video has the most number of GTCNN triggers (89.2% miss rate).

To explore the potential of *MXNs*, we manually search for the best thresholds, measure the latency and speedup at query time based on new thresholds. As shown in table 5.14, most the videos can benefit from better thresholds to gain more speedup comparing to previous results from table 5.12. We believe *MXNs* can benefit from better techniques to choose the thresholds.

Video	*MXNs*		
	# Clusters	# GTCNN	Miss Rate (%)
Auburn	2177	149	6.84
Auburn North Ross	2336	913	39.08
Bellevue 150th Road	60	12	20
Bellevue Ne8th Road	299	37	12.37
Jackson Hole	5268	1881	35.7
Jackson Town	1250	1115	89.2
Lausanne	72	32	44.44
Sittard	5717	1170	20.47

Clusters: number of clusters' centroids
GTCNN: number of times to trigger GTCNN
$Missrate = \frac{\#GTCNN*100}{\#Clusters}$

Table 5.13: Performance Breakdown at Query Time

Video	Focus Model	*MXNs*		Speedup
	Latency (s)	Latency (s)	Precision (%)	
Auburn	21.61±0.12	4.061±0.28	98.89	5.32
Auburn North Ross	29.42±0.16	13.47±0.36	98.63	2.18
Bellevue 150th Road	2.66±0.05	0.42±0.18	98.96	6.33
Bellevue Ne8th Road	2.37±0.06	0.737±0.26	99.15	3.21
Jackson Hole	57.06±0.22	18.87±0.44	99.2	3.02
Jackson Town	18.03±0.14	17.78±0.26	99	1.01
Lausanne	3.16±0.1	0.96±0.14	99.01	3.3
Sittard	88.49±0.23	17.48±0.34	99.52	5.06

Table 5.14: Query Latency - Best threshold

5.3.2 Face Recognition in Videos

In this session, we propose a novel approach for face recognition in videos by adopting *MXNs* for facial features extraction along with an accurate face classifier.

Shen at el. shows that only a set of dominant classes occur during a window of time within a video [75]. Hence, he uses a cascade of cheap model, which is trained to recognize a few dominant faces, and a heavy but accuracy model to detect variety of faces. However, as time goes by, if the set of dominant classes changes, the system has to train new cheap model to recognize a new set of dominant faces. This approach suffers from the overhead of switching and training models if there are frequent changes in dominant classes.

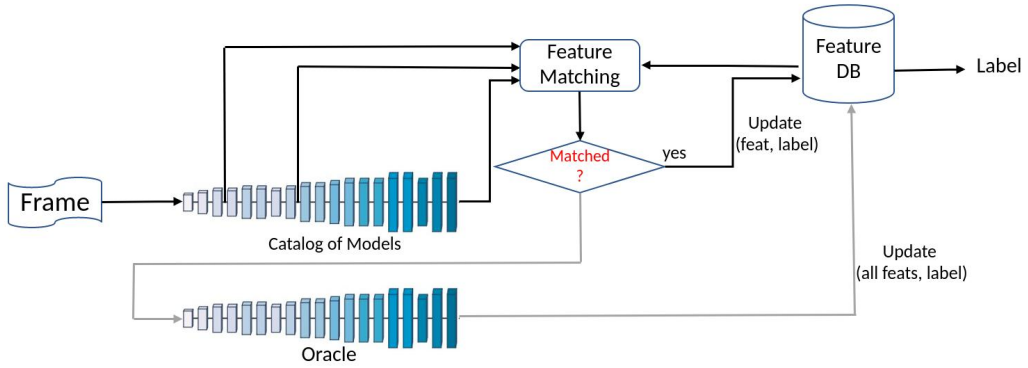


Figure 5.5: *MXNs*-based Face Recognition System Architecture

To solve the above-mentioned problem, we replace the cheap model with multi-exit face verification models. Instead of using cheap models to recognize faces directly, we use *MXNs* to extract facial features of recent faces and maintain them in a database. In order to match extracted facial features with a label, we use a large but accurate classifier to classify incoming face image into name and assign that name to extracted features and store them in database. When a new face image comes in, our system extracts the facial features from the incoming image using *MXNs* and match those features with recent facial features in the database using distance measurements such as mean square error. Facial features are commonly used to check face similarity [68, 43]. If it matches a face in the database, we retrieve the name from a database and return it to users. Otherwise, we will trigger a face classifier to recognize the face and store the name and its associated facial features into the database. Figure 5.5 shows the overview of our computational pipeline to recognize faces within video.

The processing pipeline for a single image is shown in Algorithm 3. First, we use each single model in *MXNs* to generate the facial features from input image and check with all existing faces in our database associated with current model as shown in line [3-14]. If the distance between incoming face and stored face is less than a pre-defined threshold, we will return the stored face as a result and jump to update phase (line [10-11]). Otherwise, if we cannot find any matches, we will trigger

Algorithm 3 *MXNs*-based Face Recognition Algorithm

Data:
Incoming face image *I*,
Oracle classifier O,
MXNs of n models *C*=$\{M_1, ..., M_n\}$,
thesholds *T*=$\{T_1, ..., T_n\}$,
Database D=$\{[M_1 : \{face_1 : f_{1,1}, ..., face_k : f_{1,k}\}, ..., M_n : \{face_1 : f_{n,1}, ..., face_k : f_{n,k}\}]\}$,
α
Result: label l

$F \leftarrow []$
$l \leftarrow unknown$
for M_i *in* C **do**
  $f \leftarrow M_i(I)$
  $F \leftarrow F \cup f$
  for $face_j$ *in* $D[M_i]$ **do**
    $f_j \leftarrow D[M_i][face_j]$
    $d \leftarrow distance(f, f_j)$
    if $d \leq T_i$ **then**
      $l \leftarrow face_j$
      $goto\ update$
    end
  end
end
$l \leftarrow O(I)$
update:
for i *from* $[1, ..., length(F)]$ **do**
  $D[M_i][l] \leftarrow (1-\alpha)*D[M_i][l]$ + α*$F[i]$
end
return l

MXNs		VGG-Face
Exit	Accuracy(%)	Accuracy(%)
0	88.57	97.28
1	93.47	
2	94.82	

Table 5.15: Face Recognition Accuracy on LFW Dataset

oracle classifier in line 15 to get the label. Finally, we update the facial features associated with the label for every executed models to compute the average facial features for each face over time as shown in line [17-19] and return the face label to user in line 20.

5.3.2.1 Experiment Setups

We use CasiaWebface [88] to train 3-exit mobilenetv1-based models [38] with 7 separable convolutional layers and residual connections between exits as shown in figure 5.3 for facial features extractions. To keep the computation cost low, we use the input size of 112x112 as proposed in [23]. For face recognition task, we use pre-trained VGG-Face [68] which is trained on VGG-Face-v1 dataset. Table 5.15 shows the accuracy of our *MXNs* and VGG-Face on LFW dataset [43], which is widely used to evaluate face recognition models.

In order to prevent *MXNs* providing incorrect results (i.e., stopping the execution when two faces don't match to each other), we collected 3 extra videos and use them to search for a distance threshold at each exit to achieve 99% of accuracy on all 3 videos.

We evaluate our proposed approach on 4 videos in [75] including Friends, The Departed, Good Will Hunting and Ocean's videos.

5.3.2.2 Explore The Facial Update Configurations

In this section, we explore the α value in algorithm 3 to keep the best facial features associated with a single face in order to achieve high matching accuracy.

The formula of facial features updating as follow: $F = (1 - \alpha) * F + \alpha *$

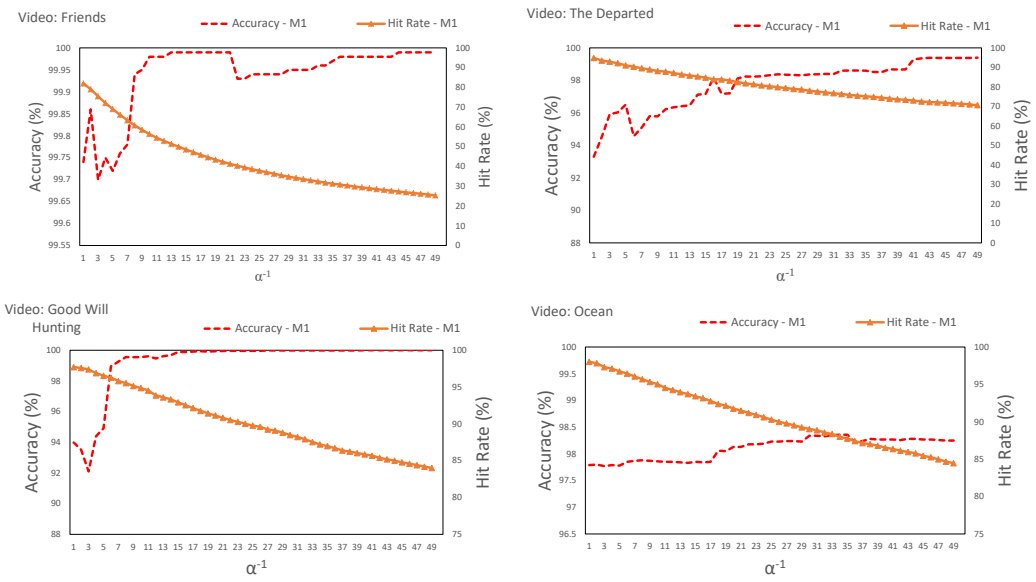


Figure 5.6: Alpha exploration

F'. Herein, F is the current features in our database, F' is the new facial features extracted from incoming image. In our case, α controls how much information about the facial features, associated with a particular label in our database, we need to keep and how much information we should get from the new features.

We fix the thresholds at each model in *MXNs* and use 4 videos in [75] to explore the α value. Figure 5.6 shows the accuracy and hit rate of the first model M1 in *MXNs*. The graphs show the relationships between $\frac{1}{\alpha}$, hit rate and accuracy. As the $\frac{1}{\alpha}$ increases, we will response less to new changes to stored facial features so the algorithm only accepts incoming faces it is already familiar with. Hence, the accuracy tend to improve when we increase $\frac{1}{\alpha}$. However, as the algorithm is getting more conservative, we suffer from hit rate decrements with low α. Furthermore, We can use 3 videos to explore for α and easily find the sweet spot, ranging from 8 to 18, to trade-off between hit rate and accuracy for the remaining video. For that reason, in all our experiments, we fix $\frac{1}{\alpha}$ to 15 to achieve high accuracy and evaluate the performance of proposed face recognition algorithm.

Video	VGG-Face		*MXNs*		Speedup
	Acc. (%)	Lat. (s)	Acc. (%)	Lat. (s)	
Friends	99.26	1120.67±1.52	99.76	93.76±0.81	11.95
The Departed	94.56	297.22±1.55	95.79	43.56±0.55	6.82
Good Will Hunting	97.32	333.1±1.32	98.95	25.57±0.56	13.03
Ocean	98.84	278.47±1.39	97.42	20.8±0.53	13.39

Acc.: Accuracy , Lat.: Latency

Table 5.16: Face Recognition Accuracy - VGG-Face as Oracle

Video	Hit Rate (%)				Accuracy (%)			
	M0	M1	M2	VGG Face	M0	M1	M2	VGG Face
Friends	87.99	3.35	3.28	5.38	99.99	100	99.86	95.75
The Departed	85.38	1.61	1.81	11.2	97.86	100	96.94	79.21
Good Will Hunting	93.37	1.26	1	4.37	99.67	100	93.44	84.64
Ocean	93.87	1.18	0.75	4.2	97.77	96.67	100	89.2

M0, M1, M2: 3 models in *MXNs*

Table 5.17: Hit Rate and Accuracy - VGG-Face as Oracle

5.3.2.3 Evaluation Results

Table 5.16 shows the performance of our approach comparing to using VGG-Face model. Overall, by not triggering the heavy VGG-Face model frequently, *MXNs* significantly improve the latency over VGG-Face, ranging from 6.82x-13.39x, while achieving comparable accuracy over existing systems using a VGG-Face model.

Next, we examine the effects of *MXNs* by counting the number of face images processed at each exit in terms of hit rate. Table 5.17 shows the hit rate and accuracy at each exit. More than 87% of the inputs exit at the first model with high accuracy. This implies that the proposed system can effectively learn and update the facial features associated to each face over time in a video and only forward the hard cases to more expensive models. Among all the hard instances that M0 forwards to upper models, 55.2%, 23.39%, 34.09% and 31.48% stops at M1 and M2 without triggering the VGG-Face in Friends, The Departed, Good Will Hunting and Ocean videos respectively. This indicates that *MXNs* successfully process a partial

Video	Oracle Acc. (%)	*MXNs* Acc. (%)	Max Speedup	Speedup	Miss Rate(%)
Friends	95.01	98.91	1.71	1.28	5.56
The Departed	93.34	96.55	1.72	1.23	11.35
Good Will Hunting	89.21	92.53	1.71	1.29	4.44
Ocean	90.91	98.86	1.7	1.29	4.91

Acc.: Accuracy

Table 5.18: Face Recognition Accuracy - Last model in *MXNs* as Oracle

of hard cases with only a small amount of extra computations given that VGG-Face is approximately 32x times more computational intensive than our models in terms of FLOPS.

To understand better the effectiveness of *MXNs* , we add a fourth exit into our existing *MXNs* and use it to replace VGG-Face as a oracle classifier. Table 5.18 shows the performance of 4-exit models comparing to only the fourth model as an oracle classifier. Interestingly, using *MXNs* improves the accuracy on all 4 videos, ranging from 3.2% to 7.95%. This confirms that using the average facial features over time to match incoming face helps to improve the accuracy of a single model. The max speedup shows the most benefit in latency that we can achieve if we only use the cheapest model comparing to using the most expensive model within *MXNs* . Overall, we achieve a speedup from 1.23 up to 1.29 given the maximum speedup we can achieve limited to 1.72.

5.4 Discussions

Here, we discuss some of the limitation of proposed *MXNs*.

- **Inefficient partial run implementation**: We rely on partial run of Tensorflow[8] to reuse the computations done at early exit for the next exits. However, the current implementation of partial run has some overhead that may reduce the performance of *MXNs* with too many exits. Choosing the

number of exits and where to put them should be carefully in considered to maximize the performance of the system.

- **Inefficient batching execution**: One of the techniques to maximize the GPU's utilization is to batch multiple images into a single request and use the GPU to process them at once. However, our experiment currently uses only a single image at a time. If multiple images are batched, some of them may exit at the first exit while the others might need to go further. This rises a problem that we need to read out the temporary outputs out of the GPU, remove the outputs associated with those exit early and resubmit new data into the GPU. That process causes a lot of overhead and hence, might out-weight the benefits of having *MXNs*. In order to support batching feature, new GPU operations should be introduced to allow easy modifications of the temporary data on the GPU itself.

Chapter 6

Literature Review

6.1 Deep Learning for Vision Sensing

Deep learning models Krizhevsky [52] at el. won ILSVRC (ImageNet Large Scale Visual Recognition Challenge) in 2012 by using a first deep convolutional neural network called AlexNet to push the error rate of image classification task down to 18.2%. After that, a lot of efforts have been done to improve the performance of deep learning models. For example, Simonyan at el. [76] introduced the potential of stacking multiple layers of 3x3 filters to construct VGG network and achieve significant boost in accuracy. He at el. [36] introduced the residual connections, which are used to learn the identification mapping to mitigate the gradient vanishing problem, to achieve outstanding performance in ResNet. Object detection task was also seen a huge boost in performance [27, 72, 62, 70, 71]. Girshick at el. proposed Fast-RCNN and Faster-RCNN frameworks [27, 72] that firstly detect regions of interests (ROIs - those that may contain objects) and secondly use image classification on those regions to classify the labels. Despite of achieving highly accurate results, those systems are slow because they have to run image classification task on huge number of regions of interests. Yolo object detection [70] was proposed to solve the latency issue by treating both ROIs detection and label classification as a single regression problem and learning it end-to-end. By simplifying the ROI detection,

Yolo allows use to detect multiple objects within a frame by just a single inference step. Liu at el. [62] pointed out that using features at different layers could help detect objects at multiple scales. Unlike Yolo, which only use features at the last layer to do detection, Liu proposed SSD framework that aggregates features across multiple layers to learn the regression models and significantly boost up the detection accuracy comparing to Yolo framework. There are also a tremendous works on various computer vision tasks such as activity recognition in video [25], image captioning [89] and many more.

Deep learning frameworks: Caffe [47], Theano [11] and Tensorflow [8] are the most common deep learning frameworks that are highly optimized to run on desktops and servers. Later on, Lane at el. have taken crucial first steps towards real-time execution of DNN and CNN on mobile devices [54, 55]. In [55], the authors showed that it is feasible to run entire DNN for audio sensing applications on low-power mobile DSPs. In addition, the DeepX framework enables the execution of DNN and CNN on mobile devices [54] by splitting computations across multiple co-processors. We believe that our work can complement DeepX in the following ways. First, DeepX is designed with a ML principal-driven approach where our works takes a system-driven optimization approach, giving the potential opprotunities to use both approaches together for further latency reduction. Second, DeepX is effective in reducing the latency of fully-connected layers while our framework focused on reducing latency of convolutional layers.

Vision Sensing Systems Ha et el. proposed the Gabriel framework [30] to support cognitive assistance applications using cloudlet to minimize occurred latency. Recently, Glimpse [17] leveraged the cloud to enable real-time object detection and tracking while MCDNN [35] executed deep learning algorithms across mobile devices and clouds. MCDNN [35] proposed efficient optimization techniques such as building multiple smaller DNN models to recognize frequently appearing objects, sharing visual features between applications and optimizing task offloading to the clouds. Gabriel [30] uses cloudlets to support cognitive assistance applica-

tions while LiKamWa et al. presented optimization techniques for image sensors to enable continuous mobile vision [57] and Starfish [58] to support concurrent execution of multiple vision applications. Kang [48] at el. proposed NoScope query video system that deploys specialized models, which mimic the behaviors of full model but only for a small set of potential classes, on particular video stream. Focus [39] improves the idea of NoScope by leveraging top-K results of specialized models at ingestion time to achieve high recall and fast indexing. After that, during query time, Focus uses state-of-the-art model to correct the mistakes made by specialized models. By doing so, Focus found a sweat spot to trade-off for ingestion latency and query latency.

6.2 Deep Learning Optimizations

Inference optimization: There has been a number of prior work to reduce training time of CNN and DNN [63]. However, little work has focused on optimizing inference time as most prior works used powerful servers and desktop machines for inferences. A few works aim at optimizing inference time. For instance, Vanhoucke et al. [85] develops a suite of low-level optimization techniques to reduce the inference latency (e.g., using fixed point arithmetic and SSSE3/SSE4 instructions on x86 machines). Also, approximation techniques are developed to reduce latency with trade-offs in accuracy [46]. However, these studies were focused on powerful desktop or server machines.

Model Quantization: HashNet [14] quantizes the network parameters by hashing weights into different groups before training. HashNet only needs to store shared weights and the hash to reduce the storage space. However, during the inference, shared weights need to be restored to original shape. Hence, neither inference time or memory usage is improved. [69] proposed an approach to quantize parameters into binary values and used bit-wise operations to speed up the inference for moderate accuracy loss.

Model approximation and pruning: Restructuring DNN models has been widely studied to reduce the size of the model and accelerate the inference speed [24, 29, 46, 50, 53, 74]. [24] explored the use of Singular Value Decomposition (SVD) to approximate the weight matrices within neural network to reduce both memory consumption and computational cost. However, it works well with fully connected layers but doesn't provide advantages on convolutional layers. Han at el. [34] pruned the unimportant connections within a model during training step. This method helps to remove near-zero weights and save a lot of storage space. However, inference time improvement is limited due to irregular network patterns and it requires dedicated hardware to achieve significant inference speedup. Recently, Bhattacharya and Lane proposed a framework to sparsify fully-connected layers and separate convolutional kernels, reducing the memory and computational costs of DNN/CNN significantly for wearables [53]. Kim et al. proposed a *Tucker-2 decomposition* technique [50]. It decomposes a tensor into three smaller ones, accelerating convolutional layer execution for mobile devices. Han et al. proposed a method to prune network connection based on magnitude of parameters during fine-tuning phase [33]. However, weights matrix becomes sparse after pruning process and makes it hard to leverage optimized library to execute inference step efficiently. Yao et al. proposed *DeepIot* system that leverages recurrent neural network (RNN) to learn the relationship between parameters across many layers and prune the redundancy automatically [87]. However, RNN is prone to gradient vanishing problem and may not work well if the input sequence is too large.

Chapter 7

Conclusions and Future Directions

In this chaper, I conclude this thesis by summarizing my contributions and outline some future directions.

7.1 Summary of Contributions

Mobile Deep-Learning Framework Optimizations: I explored a variety of design choices and optimization techniques to efficiently execute CNN models on mobile devices (such as memory vectorization, data representation, usage of half floating points) in order to build DeepSense and DeepMon frameworks. I proposed a smart caching mechanism leverages similarities of consecutive images to cache internally processed data within the deep convolutional neural network to reduce the latency of processing video data. I also show that we can leverage new programming API for graphics processing such as Vulkan, which is recently available on both Android and iOS platforms, to do CNN computation.

Deep Learning Models Compression: I proposed D-Pruner, a simple but effective compression technique to remove redundancy within existing CNN models. D-Pruner introduces a novel concept of the masking block to automatically figure out redundant filters which have low impacts on final accuracy during the training

process.

Multi-exit Models as Efficient Implementation to Accelerate Computer Vision Workloads: I adopted multi-exit models as efficient implementation to replace DNNs to improve the performance of many machine learning workflows. I propose using focal loss to improve the accuracy of early exiting decisions. By applying the idea of *MXNs* into many existing systems such as video query system Focus and face recognition system, it enables us to use many optimizations such as results aggregation across models, facial features matching, etc., which cannot be done before, to improve the overall performance.

7.1.1 Publications

The research work described in this thesis have led to publications in peer–reviewed conferences. Below is a list of selected publications.

WearSys '16	Huynh, Loc Nguyen, Rajesh Krishna Balan, and Youngki Lee. "Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices." Proceedings of the 2016 Workshop on Wearable Systems and Applications. ACM, 2016.
Mobisys '17	Huynh, Loc N., Youngki Lee, and Rajesh Krishna Balan. "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications." Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services. ACM, 2017.
EMDL '18	Huynh, Loc N., Youngki Lee, and Rajesh Krishna Balan. "D-pruner: Filter-based pruning method for deep convolutional neural network." Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning. ACM, 2018.

7.2 Future Directions

Efficient layers for generative models: Many generative models such as GAN [28] to generate artificial images relies on convolutional-transpose to upscale

lower input size into higher output size. However, the current implementation of convolutional transpose operations in many frameworks such as Tensorflow [8] is to upscale the input into output's size by padding zero and use convolutional operations on higher output size to yield the final output. This approach requires a lot of computational cost and will not run fast on resource-constraint devices such as embedded board. So I ask a question if there is a similar approach to separable convolution layers [38] to reduce the computational cost of convolutional transpose operation?

MXNs-based Optimizations at GPU operations: As I mentioned in 5.4, one of the technique to maximize the GPU's utilization is to batch multiple images into a single request and use the GPU to process them at once. In case of processing MXNs, if some instances exit early, we will have to read the data out of the GPU at every exit, modify and submit it back to the GPU. This process introduces a huge overhead to the system. What if we can introduce more operations at GPU-level to process it directly on the GPU without reading/writing GPU data back and forward? With those operations, we can enable supporting many more features for the catalog not just at framework-level but also at GPU-operational kernel-level such as batching, or dynamic re-sizing of a batch, etc.

Efficient Storage for Video Query Systems: Storing multiple month-long videos might consume a lot of data storage. However, there are many sessions within a single video that do not contain any objects/activities of interest. For example, many videos of traffic road at night only contain few objects for a short window of time. Hence, storing the video for a whole night would be inefficient. If we can use fast object detectors on those videos and only store video segments that contain objects/activities of interest, we would significantly reduce the amount of data that needs to be stored.

Bibliography

[1] A brief report of the heuritech deep learning meetup 5. https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/. Accessed: 2016-12-8.

[2] Google glass. https://developers.google.com/glass/. Accessed: 2016-12-8.

[3] Hololens. https://www.microsoft.com/microsoft-hololens/en-us. Accessed: 2016-12-8.

[4] Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/. Accessed: 2016-12-8.

[5] Nvidia cuda toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed: 2016-12-8.

[6] Vulkan. https://www.khronos.org/vulkan/. Accessed: 2016-12-8.

[7] 2019-05-10. https://www.amazon.com/b?ie=UTF8&node=16008589011.

[8] 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 *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[9] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 272–285, New York, NY, USA, 2007. ACM.

[10] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 273–286, New York, NY, USA, 2003. ACM.

[11] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

[12] S. Bhattacharya and N. D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM*

Conference on Embedded Network Sensor Systems CD-ROM, pages 176–189. ACM, 2016.

[13] J. Boger, J. Hoey, P. Poupart, C. Boutilier, G. Fernie, and A. Mihailidis. A planning system based on markov decision processes to guide people with dementia through activities of daily living. *IEEE Transactions on Information Technology in Biomedicine*, 10(2):323–333, 2006.

[14] Z. Cao, M. Long, J. Wang, and P. S. Yu. Hashnet: Deep learning to hash by continuation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5608–5617, 2017.

[15] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

[16] K. Chellapilla, S. Puri, and P. Simard. High Performance Convolutional Neural Networks for Document Processing. In G. Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), Oct. 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[17] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 155–168, New York, NY, USA, 2015. ACM.

[18] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[19] F. Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7:8, 2015.

[20] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[21] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Proceedings of 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1 of *CVPR '05*, pages 886–893. IEEE, 2005.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[23] J. Deng, J. Guo, N. Xue, and S. Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4690–4699, 2019.

[24] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[25] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '15, pages 2625–2634, 2015.

[26] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[27] R. Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[28] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[29] L. Grasedyck, D. Kressner, and C. Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[30] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.

[31] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[32] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[33] 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.

[34] 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.

[35] 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*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.

[36] 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.

[37] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[38] 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.

[39] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 269–286, 2018.

[40] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*, 2017.

[41] G. Huang, D. Chen, T. Li, F. Wu, L. Van Der Maaten, and K. Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. *arXiv preprint arXiv:1703.09844*, 2, 2017.

[42] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[43] G. B. Huang, M. Mattar, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database forstudying face recognition in unconstrained environments. In *Workshop on faces in'Real-Life'Images: detection, alignment, and recognition*, 2008.

[44] L. N. Huynh, R. K. Balan, and Y. Lee. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, pages 25–30. ACM, 2016.

[45] 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*, pages 82–95. ACM, 2017.

[46] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866, 2014.

[47] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[48] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[49] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015.

[50] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

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

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

[53] N. Lane and S. Bhattacharya. Sparsifying deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, pages 176–189. ACM, 2016.

[54] 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.

[55] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 283–294, New York, NY, USA, 2015. ACM.

[56] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[57] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl. Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 69–82, New York, NY, USA, 2013. ACM.

[58] 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*, MobiSys '15, pages 213–226, New York, NY, USA, 2015. ACM.

[59] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[60] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

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

[62] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[63] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[64] J.-M. Morel and G. Yu. Asift: A new framework for fully affine invariant image comparison. *SIAM Journal on Imaging Sciences*, 2(2):438–469, 2009.

[65] Y. Nesterov et al. Gradient methods for minimizing composite objective function, 2007.

[66] C. Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, page 5. ACM, 2018.

[67] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[68] O. M. Parkhi, A. Vedaldi, A. Zisserman, et al. Deep face recognition. In *bmvc*, volume 1, page 6, 2015.

[69] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[70] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[71] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[72] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[73] K. Rupp, F. Rudolf, and J. Weinbub. Viennacl-a high level linear algebra library for gpus and multi-core cpus. In *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.

[74] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Proceedings of 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '13, pages 6655–6659. IEEE, 2013.

[75] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3646–3654, 2017.

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

[77] S. Song, V. Chandrasekhar, N.-M. Cheung, S. Narayan, L. Li, and J.-H. Lim. Activity recognition in egocentric life-logging videos. In *Asian Conference on Computer Vision*, pages 445–458. Springer, 2014.

[78] 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.

[79] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[80] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[81] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[82] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

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

[84] G. Urban, K. J. Geras, S. E. Kahou, Ö. Aslan, S. Wang, R. Caruana, A. Mohamed, M. Philipose, and M. Richardson. Do deep convolutional nets really need to be deep (or even convolutional)? *CoRR*, abs/1603.05691, 2016.

[85] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, 2011.

[86] A. Vedaldi and K. Lenc. Matconvnet: Convolutional neural networks for matlab. In *Proceedings of the 23rd ACM International Conference on Multimedia*, pages 689–692. ACM, 2015.

[87] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher. Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems. ACM*, 2017.

[88] D. Yi, Z. Lei, S. Liao, and S. Z. Li. Learning face representation from scratch. *arXiv preprint arXiv:1411.7923*, 2014.

[89] Q. You, H. Jin, Z. Wang, C. Fang, and J. Luo. Image captioning with semantic attention. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4651–4659, 2016.

[90] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, page 1. ACM, 2018.