Realtime Image Processing for Resource Constrained Devices

by

Christopher Streiffer

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Landon Cox, Supervisor

_____
Jeffrey Chase

_____
Bruce Maggs

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Computer Science
in the Graduate School of Duke University

2018

Abstract

Realtime Image Processing for Resource Constrained Devices

by

Christopher Streiffer

Department of Computer Science
Duke University

Date: _____
Approved:

_____
Landon Cox, Supervisor

_____
Jeffrey Chase

_____
Bruce Maggs

An abstract of a thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in the Department of
Computer Science in the Graduate School of
Duke University

2018

Copyright © 2018 by Christopher Streiffer
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

With the proliferation of embedded sensors within smartphones and Internet-of-Things devices, applications have programmatic access to more data processing than ever before. At the same time, advances in computer vision and deep learning have fostered methodology for performing complex, yet powerful operations on spatial and temporal data. Capitalizing on this union, applications are capable of providing advanced functionality to their users through features such as augmented reality and image classification. However, the devices responsible for running these libraries often lack the sufficient hardware to replicate the parallelization and straight-line speed of high-end servers. For image processing applications, this means that realtime performance is difficult without compromising functionality.

To detail this emerging paradigm, this work examines two image processing applications which offer advanced functionality. The first, DarNet, utilizes the TensorFlow library to perform distracted driving classification based on image data using a Convolutional Neural Network (CNN). The second, PrivateEye, uses the OpenCV library to provide a camera based access-control privacy framework for Android users. While this advanced processing allows for enhanced functionality, the computationally expensive operations impose limitations on the realtime performance of these applications due to the lack of sufficient hardware.

This work posits that realtime image processing applications running on resource constrained devices require the use of edge servers. To this extent, this work presents

ePrivateEye, an extension to PrivateEye which provides code offloading to an edge server. The results of this work shows that offloading video-frame analysis to the edge at a metro-scale distance allows ePrivateEye to analyze more frames than PrivateEye's local processing over the same period, and achieve realtime performance of 30 fps with perfect precision and negligible impact on energy efficiency.

To my parents, Cheryl and Howard, for their unwavering support throughout the many different facets of life.

Contents

Abstract **iv**

List of Tables **x**

List of Figures **xi**

List of Abbreviations and Symbols **xiv**

Acknowledgements **xv**

1 Introduction **1**

1.1 From the End to the Edge 5

1.2 Contributions 6

1.3 Roadmap 7

2 Related Work **8**

2.1 Mobile Offloading 8

2.2 Network Optimization 9

2.3 Deep Learning Optimization 9

3 Background and Motivation **11**

3.1 Deep Learning Limitations 11

3.2 Computer Vision Limitations 14

3.3 The Cloud vs Edge 15

4 Detecting Distracted Driving **17**

4.1 Background and Motivation 17

4.2 Acknowledgements 21
4.3 Related Work 21
4.4 System Design 24
4.4.1 Collection Agents 24
4.4.2 Centralized Controller 24
4.4.3 Analytics Engine 26
4.5 System Implementation 26
4.5.1 Data Streaming Framework 26
4.5.2 Analytics Engine 27
4.5.3 Privacy Preserving Analytics 32
4.6 DarNet Evaluation 34
4.6.1 Evaluation Setup 35
4.6.2 Evaluation Results 36
4.6.3 Denoising-CNN Results 38
4.7 Feature Extraction Runtime Comparison 39
5 From the End to the Edge **42**
5.1 Background and Motivation 42
5.2 Acknowledgements 43
5.3 PrivateEye Overview 44
5.4 Design Principles 46
5.5 Implementation 48
5.5.1 ePrivateEye Design 49
5.5.2 Server Configurations 49
5.6 ePrivateEye Evaluation 53
5.6.1 Experiment Setup 54

5.6.2 Frames per Second 56
5.6.3 Precision and Recall 59
5.6.4 Resource Usage 62
5.6.5 Scalability 64
5.7 Case study: Metro 66
5.7.1 Impact of Packet Loss 66
5.7.2 Impact of Network Delay 68
5.8 ePrivateEye in the Cloud 69
6 Discussion **71**
6.1 Impact of Offloading 71
6.2 Managing Network Conditions 71
6.3 Deep Learning 72
7 Conclusion **73**
7.1 Contributions 73
7.2 Future Directions 74
Bibliography **75**

List of Tables

4.1 Driver behavior classes listed by corresponding model types and amount of data collected. 35

4.2 Ensemble model Top-1 classification results for the collected dataset. 36

4.3 CNN and dCNN Top-1 classification percentages for the alternative distracted driving dataset. 38

List of Figures

4.1 Diagram depicting the classification system where the probability distributions from models are combined using a bayesian network to produce probabilities for each class and the overall classification. 20

4.2 Diagram depicting the collection agents which run embedded within IoT devices, the controller which collects and prepares the data, and the analytics engine which performs the ensemble classification. . . . 23

4.3 The privacy implementation for the system where the paths between the mobile device and server correspond to the different levels of downsampling – 100x100, 50x50, 25x25. 32

4.4 From right to left: the undistorted image; the image distorted to the size of 100x100 pixels; the image distorted to the size of 50x50 pixels; the image distorted to the size of 25x25 pixels. 33

4.5 The training configuration for the dCNN models. 34

4.6 Confusion matrices for the collected datasets for architectures: (a) CNN+RNN (*DarNet*) (b) CNN+SVM (c) CNN (frame data only). . 35

4.7 Configuration specifications and runtime results for each model, with the following abbreviations: CPU-A – 1.3 GHz Intel Core M, CPU-B – 2.65 GHz Intel Xeon E5-2640 v3 Haswell, GPU – NVIDIA Tesla K80s. 40

5.1 A region on a whiteboard marked as public using PrivateEye's marker (a), and An app's view while capturing the same region using PrivateEye's software enabled smartphone (b). 44

5.2 Series of computer vision algorithms applied on image data under PrivateEye. 44

5.3 ePrivateEye architecture where the client component runs as part of camera service, a trusted module of Android OS, and the client and server maintain a persistent TCP connection for as long as the app session runs. 48

5.4 Metro configuration where the AP resides within the American Tobacco Campus (1), connects to an Arista 7050 SDN switch in TelCom (2), which finally connects to a Cisco 4500X SDN located within Fitzpatrick East datacenter that contains a NVIDIA Tesla P100 GPU (3), totalling a line distance of 2.85 miles with a corresponding ping time of 0.5ms. 51

5.5 SDN configuration where the top path represents the typical production path through the network, while the bottom path represents the SDN path which traverses two Cisco 4500X and one Arista OpenFlow switches, and the choice of path is selected through a controller not pictured in this diagram. 52

5.6 FPS results for the *still* camera movement. 56

5.7 FPS results for the *spin* camera movement. 57

5.8 FPS results for the *scan* camera movement. 57

5.9 Precision and recall for *still* camera motion. 60

5.10 Precision and recall for *spin* camera motion. 60

5.11 Precision and recall for *scan* camera motion. 61

5.12 Memory usage for analyzing a video for 30 seconds under different configurations for ePrivateEye server. 63

5.13 CPU load for analyzing a video for 30 seconds under different configurations for ePrivateEye server. 63

5.14 Power consumption for analyzing a video for 30 seconds under different configurations for ePrivateEye server. 64

5.15 Median rate of frame delivery under different configuration of ePrivateEye server. 65

5.16 Impact of packet loss on the rate of frame delivery. 67

5.17 Impact of network delay on the rate of frame delivery. 68

5.18 The secure channel setup to provide connectivity between the ePrivateEye client on Duke's trusted network and ePrivateEye server on an instance of AWS server. 69

5.19 FPS achieved by configuring ePrivateEye server on an instance of AWS server. 70

List of Abbreviations and Symbols

Abbreviations

AOSP	Android Open Source Platform
AP	Access Point
BN	Bayesian Network
dCNN	Denoising Convolutional Neural Network
CNN	Convolutional Neural Network
FPS	Frames per second
IoT	Internet of Things
IMU	Inertial Measurement Unit
ML	Machine Learning
NHTSA	National Highway Trafficc Safety Administration
RNN	Recurrent Neural Network
SDN	Software Defined Networking
SVM	Support Vector Machine

Acknowledgements

I am incredibly thankful to have attended Duke University for the duration of both my undergraduate and graduate experiences. During that time, Landon Cox has provided a tremendous amount of guidance and support, and it has been an honor to have him as my advisor. This thesis is a culmination of my work over the past two years, and there are a number of people I would like to express my gratitude towards.

First and foremost I would like to thank my committee members Landon Cox, Jeff Chase, and Bruce Maggs. They provide an incredible amount of expertise and knowledge, and together, form a strong and well-balanced committee. Through working with and taking classes taught by them, I have gained a wealth of knowledge in the many different facets of systems and networking. Working as Jeff's Teaching Assistant at the both the undergraduate and graduate level has helped me learn how to present complex material in a concise manner while being supportive and informative to the students in the class.

I am tremendously grateful to have worked with Theo Benson across multiple research projects. When I approached Theo as an undergraduate with the idea of an independent study, he enthusiastically took me under his guidance. Theo has been a great mentor and friend, and is incredible to work with. His relentless approach towards pursuing complex problems is a trait I hope to exemplify for the rest of my life.

I would like to thank my mentor, Ramya Raghavendra, and manager, Mudhakar Srivatsa, at IBM Research. Working in the Cognitive IoT department allowed me to improve my skills as both a researcher and critical thinker. Ramya provided a tremendous amount of guidance and advice, and together, we tackled exciting and intricate research problems. It has been a pleasure working with you! With the additional support of my peers, Savvas Petridis and Shawn Pachgade, I will always cherish the summers I spent at IBM.

I would like to thank my advisor Landon Cox. When I initially approached Landon to express my interest in research, without hesitation, he found a complex problem for me to work on, and provided a great introduction the problems I would encounter as a graduate student. Since then, Landon has pushed me to be a better researcher and has provided invaluable guidance on how to solve problems I encounter not only in my work, but also in life. The lessons and skills I have developed under his guidance will serve as invaluable problem solving tools for any future problem I encounter.

I would like to thank my friends and family. It has been great sharing an office and working with Ali Razeen – one day we will complete our billion dollar idea, Switchboard. I would like to thank my sister and grandmother for always being there for me. Finally, to my parents, your unconditional support throughout my tenure at Duke has been an instrumental part of my success. The values you instilled within me at an early age have helped me to overcome and solve the multifaceted problems I have encountered within my work. Thank you for always being there for me and for always encouraging me to pursue my dreams.

1

Introduction

With an increase in the number of Internet-of-Things (IoT) devices, there exists an abundance of data for monitoring and observing our everyday lives. From smartphones (Android, IoS), to personal assistants (Alexa, Google Home), to fitness trackers (FitBit, Apple Watch), these devices are becoming increasingly ubiquitous as a result of an increase in consumer demand. Among these devices, cameras have emerged as a commonly supported hardware feature. With most smartphones, computers, and other low-powered electronics coming equipped with a high resolution camera, applications which collect, process, and stream video have become increasingly popular.

At the same time, the ability to process, make logical inferences about, and transform data has improved through advances in computer vision (CV), machine learning (ML), and deep learning (DL). Developers have not only gained programmatic access to interface with the embedded sensors, but also have powerful data processing libraries (Tensorflow, OpenCV) and APIs (IBM Watson, Google ML) at their disposal. Through this combination of emerging paradigms, applications have evolved to offer advanced functionality to users through features such as predictive

analytics, augmented reality, and data inference.

Within this space, image processing has become increasingly important for mobile applications, through use cases such as video-streaming, mobile-gaming, and image classification. While not all image-based applications require realtime speeds, video-streaming services typically require a high rate of frame delivery. However, designing a system that combines video-streaming and image processing faces the following challenges:

1. **Large Data Size**

   Frames received by applications from high resolution embedded cameras can be on the order of several megabytes. For instance, the Android camera delivers frames at a resolution of 1280x960 pixels, resulting in a total size of 1.85MB. Processing these full resolution frames without access to parallelization presents a significant bottleneck.

2. **High Computational Cost**

   Computer vision and deep learning algorithms can take on the order of several hundred milliseconds to execute. For instance, within the OpenCV library, applying a Gaussian blur to a 3 Megapixel image running on a Galaxy Note 10.1 takes approximately 135.85 milliseconds to execute [14]. On the deep learning side, using a large Convolutional Neural Network (CNN) to perform image classification running on an HTC One M9 takes 709.00 milliseconds to execute, even when optimized for efficient performance [45]. Without optimization, image classifications can take upwards of several seconds to execute [45, 58, 43].

Because of these two constraints, incorporating image processing and video-streaming for applications running on resource constrained devices is a highly difficult task. Modern system-on-a-chip (SoC) designs have undergone rapid improvements

in CPU and GPU processing capabilities, but, as a result of power and space constraints, these platforms cannot duplicate the parallelism, straight-line speed, or large memories of high-end servers. For example, a high-end GPU may consume 500 Watts of power, whereas a high-end SoC GPU will consume fewer than 10 Watts. Similarly, high-end smartphones rarely offer more than 4GB of RAM, whereas mid-range servers offer 128GB of RAM. This resource gap significantly limits the execution quality of realtime image processing applications.

Within this work, we examine two example applications which use different forms of *state-of-the-art* image processing methodology to perform specialized functionality within applications designed for execution on smartphones. These two applications can be described as follows:

1. **DarNet**

   DarNet consists of two primary components: a data collection system and an analytics engine. The system takes advantage of advances in machine learning (ML) to classify driving behavior based on input sensor data. In the system implementation, data is collected from an inward facing camera and Inertial Measurement Unit (IMU) data from a smartphone. The analytics component employs a Convolutional Neural Network (CNN) to classify the image data and a Recurrent Neural Network (RNN) to classify the IMU sequence data. The results of our work show how the combination of the CNN and RNN allows for significant performance improvements over existing work, but also details the difficulty of achieving high frame rates because of the computational cost of the CNN [68].

2. **PrivateEye**

   PrivateEye presents a privacy based access-control framework for protecting visual secrets. The framework allows for users to define a generic two-dimensional

geometric shape that is easy to draw and recognize by specialized computer vision software. Because of this intuitive design, the system is capable of providing privacy with a high degree of accuracy, but requires significant compromises in functionality in order to achieve a frame rate of 20 fps [60].

While both applications are highly successful at carrying out their intended functionality, both must make significant sacrifices in order to achieve realtime performance. Because of the significant inference time of the CNN, DarNet can only perform classifications at a rate of 4 fps, and therefore, ignores the majority of received frames. PrivateEye is capable of delivering a frame rate of 20 fps, but must skip every 6th frame in order to maintain a low computational footprint. In this setting, an ideal image processing application should be able to provide the following service guarantees:

1. **High Precision and Accuracy**

   Image classification and object recognition tasks should be able to execute with a high accuracy and precision. Ideally, this means running full sized models and as complex of operations as possible without incurring a large delay within the program execution.

2. **Low Operational Cost**

   Obtaining the results of the image processing algorithm should not impose any constraints on the resource constrained device. When performing the processing, the application should not consume a significant amount of CPU cycles and should maintain a small memory footprint.

3. **High Frame Rate**

   When performing image processing on each frame, the frame rate of the application should be as close as possible to the theoretical frame rate of the

hardware. Building on the previous guarantees, this means that the image processing algorithms should not be a bottleneck in the program.

To meet these guarantees, this work posits moving the complex image processing operations from the end-devices to the edge. Edge computing offers resource constrained devices low-latency access to high-performance computing infrastructure, which fits perfectly with the needs of delivering a high frame rate, while maintaining a high level of accuracy and precision from the image processing model.

1.1 From the End to the Edge

Edge computing is an approach to compute infrastructure that places small data centers geographically close to clients. In recent years, both industry and academia have undertaken initiatives related to edge computing, including cloudlets [63], fog computing [9], and micro data centers [6]. The goal of edge computing is to give resource constrained clients access to greater compute, memory, and storage resources than are available locally, and to provide that access over a higher-quality connection than is possible over the wide area. Prior research has sought to close the performance and energy gaps between resource-poor devices and server-side infrastructure through remote execution and code offload [7, 13, 15, 16, 32, 47, 56], and image processing applications provide natural use cases for offloading computation to edge servers.

Edge computing promises to address one of the biggest problems with code offload for realtime image processing: network latency. Realtime image processing applications often require results within tens of milliseconds, and many campus, municipal, and regional networks offer both the capacity (e.g., fiber-optic networks) and software-defined networking (SDN) orchestration to establish low-latency and high-bandwidth paths between sensors and edge clusters.

The goal of this work is to show that offloading computationally expensive image

processing algorithms to the edge allows for video-streaming applications running on resource constrained devices to maintain realtime performance. The PrivateEye [60] access-control framework for visual information is an example of a system that could benefit from edge computing. PrivateEye defines a generic two-dimensional special shape that is easy for users to draw, e.g., on a piece of paper, on a whiteboard, or within a projected presentation, and easy for software on a recording device to identify in realtime. The original PrivateEye implementation performed all marker-recognition locally on a mobile device and could process frames at a rate of 20 fps. Within this work, we show that using edge computing allows for the service to perform at a rate of 30 fps, achieving realtime performance.

1.2 Contributions

To demonstrate the performance gains from offloading to the edge, we present an edge-enabled version of PrivateEye called *ePrivateEye*. ePrivateEye differs from the original PrivateEye in one significant facet: it offloads marker detection to an edge server. We explore different network and server configurations for running ePrivateEye and provide benchmark evaluations for each. Specifically, we measure frames per second (fps), battery consumption, and precision/recall under ePrivateEye. The three configurations consist of marker detection running locally on the mobile device, running on a wireless access point, and running on a GPU server connected directly to the access point. We also perform experiments with our campus SDN to control network behavior in order to understand how "well provisioned" the path between a recording device and an edge server must be to support ePrivateEye. We envision the system running within an enterprise setting where SDN could be utilized to provide secure, fast-path access to the edge server running ePrivateEye. We make the following contributions:

- We evaluate ePrivateEye under several network and server configurations, including offloading work to an access point, to an in-building server, and to an off-campus server.

- We leverage our campus SDN to evaluate ePrivateEye under metro-scale geographic distances between a device and server.

- We show that offloading marker-detection to an edge server at a metro-scale distance allows ePrivateEye to analyze 50% more frames than PrivateEye's local processing over the same period, with perfect precision and negligible impact.

1.3 Roadmap

This work is organized as follows: Chapter 2 looks at the existing work in this field, focusing on current methodologies for offloading code to high-performance servers and optimizations for running image processing algorithms on resource constrained devices. Chapter 3 provides an in-depth examination of the limitations and constraints imposed by state-of-the-art image processing libraries (TensorFlow, OpenCV) when run on resource constrained devices. Chapter 4 looks at the application DarNet, and presents evaluation results showing the efficacy at which the system can classify distracted driving using deep learning, but also, the realtime limitations of such a system. Chapter 5 presents our work with ePrivateEye, detailing and showing empirical results for a framework to offload computationally expensive algorithms to an edge server. Chapters 6 and 7 provide a discussion of the result, a summary of our contributions, and future directions for this work.

2

Related Work

This section looks into three different areas of related work – mobile offloading, network optimization, and deep learning optimizations for mobile devices.

2.1 Mobile Offloading

MAUI [15] and LEO [28] make runtime scheduling decisions about how to split processing between on-device and remote execution. Both systems solve an Integer Linear Program (ILP) that encapsulates the context in which the system is running, to determine how to best distribute processing. Rather than performing a runtime decision, ePrivateEye offloads the most computationally expensive portion of the image processing pipeline. This fixed offloading approach avoids the complexity of synchronizing client and the server. Further, based on the quality of network, ePrivateEye can be quickly reconfigured to switch between on-device and remote processing.

Kahawai [16], a platform for high-quality gaming, performs collaborative rendering between server and client. Within this work, both the client and server execute the application, and if the server finishes before the client, the client will stop pro-

cessing and incorporate the completed results from the server. ePrivateEye avoids the redundancy of performing the remote and local processing concurrently due to the high overhead imposed by the local processing.

CloneCloud [13] transforms mobile applications into a distributed system by selecting a single thread running on the device to be executed in a clone running remotely. Because the most computationally expensive component of ePrivateEye is known ahead of time, this section is offloaded entirely rather than having to make a runtime decision.

2.2 Network Optimization

Systems relying on remote processing are dependent on network conditions. IC-Cloud [65] predicts network conditions using signal strength and historical information to make lightweight offloading decisions. Whereas IC-Cloud makes internal decisions to combat poor network quality, we show that ePrivateEye maintains a strong performance across varying network conditions.

SDNs have become ubiquitous in several enterprise and campus networks, and provide more control over the data flow. ECOS [27] utilizes software defined networking to control mobile application offloading with a focus on ensuring privacy, fair resource allocation, and enforcing security constraints. ECOS is implemented as an application running within a NOX controller, and communicates with mobile clients seeking to perform application offloading with certain privacy and performance requirements. While ECOS primarily focuses on security, we utilize SDN within our network to minimize network latency.

2.3 Deep Learning Optimization

There has been a significant amount of work put into optimizing image processing functionality for execution on mobile devices. CNNDroid provides a GPU accelerated

library for executing fully trained CNNs on mobile devices [45].

Services like DeepX [44], DXTK [42], and Q-CNN [76] provide software acceleration to improve the execution times of a variety of deep learning methodologies. Google recently released their MobileNets [30] platform designed to provide smaller, more efficient deep learning models varying degrees of accuracy loss based on the needs of the devices. Overall, each of these methodologies provides the tradeoff of accuracy for performance. While this is practical for certain applications, this work takes the stance of offloading to the edge to maintain a high-level of performance without sacrificing accuracy.

3

Background and Motivation

This section presents the current difficulties faced by running deep learning and computer vision libraries on resource constrained devices. Because of the large computational cost and associated memory overhead, optimizations typically have the tradeoff of improved speed at the cost of diminished accuracy. For deep learning, the focus stems around the popular TensorFlow library, while computer vision focuses on an analysis of the OpenCV library.

3.1 Deep Learning Limitations

Neural networks have been proven to be a powerful tool for classification tasks. In particular, Convolutional Neural Networks (CNNs) have made tremendous strides in the field of computer vision at the task of image classification. Deep CNN architectures such as VGGNet [66], GoogleNet [71], and Inception-V3 [72] have consistently excelled at categorizing images into succinct classifications. The architectures that compose these models consist of a progression of convolutional layers, fully connected layers, pooling functions, and non-linear activation functions, and can contain millions of parameters requiring billions of calculations to compute an inference. For

instance, the Inception-V3 architecture contains roughly 24 million parameters requiring 5 billion multiply and add operations per classification[72]. A common trend within this field is that the deeper the model, the better the overall performance, but the higher computational cost.

The convolutional and fully connected layers within CNNs are primarily responsible for the memory and compute intensive aspects of these models. Each layer has its own computational features and use cases. Fully connected layers are characterized by a large number of parameters, and represent a linear combination of the neurons from the preceding layer to create a new set of output neurons [43]. Convolutional layers have a smaller memory footprint than fully connected layers, but a significantly higher computational cost and are useful for recognizing patterns within images [43]. The typical architecture of a CNN consists of convolutional layers at the beginning of the model to perform feature extraction and dimensionality reduction on the input image, followed by fully connected layers at the end to translate the extracted features into distinct classifications.

Two models which exemplify this design are VGGNet [66] and Inception-V3 [72]. These models are both composed of a significant number of layers and contain 144 million and 24 million parameters, respectively. Through their advanced design and size, both models have performed extremely well on the Imagenet dataset classification challenge. This dataset consists of 150,000 images spanning 1,000 different classes, and is considered a respectable benchmark for evaluating image classification models. VGGNet and Inception-V3 produce Top-1 error rates of 23.7% and 17.2%, respectively, which places them as top performing models for this task [72, 66].

Because of Inception-V3's exemplary performance, its architecture is widely used as the base model for other image classification tasks. For instance, in our work, DarNet, we use this model to perform the task of classifying distracted driving [68]. However, integrating these large models into applications designed to run on resource

constrained devices requires overcoming the following challenges:

1. **Memory Footprint**

   Because of the large number of parameters, deep models have a significantly high memory footprint. As mentioned previously, the Inception-V3 model has roughly 24 million parameters which translates to a file size of 250 MB [72]. For resource constrained devices, this consumes a significant portion of the available RAM, and makes the application itself extremely bulky [43].

2. **Computational Speed**

   Tangential to the large memory footprint, deep models have a high computational cost when performing image classification. Convolutional and fully connected layers require a large number of matrix operations to compute the output of the layer. While GPUs are highly optimized to perform these operations in parallel, most resource constrained devices lack the hardware support to efficiently execute these layers. Because these operations must be executed by the CPU, classifications can take a significant amount of time to compute.

3. **Binary Size**

   The binary size of most deep learning libraries can be significantly large. For instance, the TensorFlow binary size for mobile is 12 MB [31]. Because this binary must be included with application in the app store, downloading such a bulky application can be unappealing.

Because of these constraints, running deep CNN models on resource constrained devices can be a highly difficult task. In the TensorFlow documentation, the recommendations for optimizing for mobile consist of using smaller models, reducing the floating point precision of parameters, and using a limited TensorFlow library to

keep the binary size small [31]. Each of these recommendations presents the tradeoff of improved processing times at the cost of a reduction in accuracy.

Google recently released its MobileNets platform which is a set of CNN models designed specifically for running on mobile devices. Developers can select between different model sizes and corresponding classification accuracy based on the needs of the application. For instance, the top performing model, with a Top-1 classification accuracy of 70.7%, contains 4.24 million parameters and requires 569 million Multiplication and Addition operations (MACs) [30]. For comparison, the smallest model has a Top-1 classification accuracy of 41.3% and 0.47 million parameters requiring 14 million MACs [30]. As can be directly observed, the tradeoff of using the smaller model is a 29.4% reduction in model accuracy.

3.2 Computer Vision Limitations

Computer vision techniques provide powerful algorithmic methods for detecting and recognizing faces, performing text extraction on images, tracking moving objects, and performing image editing. Because of the broad range of uses, these algorithms are often employed within camera based applications to provide improved functionality. With a supporting community of nearly 47,000 members, OpenCV is one of the most popular computer vision libraries and is utilized by companies such as Google, Microsoft, and IBM [53]. While the platform is designed for realtime performance for desktop based applications, running these algorithms on a mobile device can be challenging.

Typically, applications will implement a pipeline of computer vision algorithms with operations such as image resizing, grayscale conversion, and object detection [60, 35]. Because of the complexity and number of operations, running a computer vision operation on frames received from a camera is a computationally expensive task.

A 2015 study conducted by Cobârzan et al. compared the performance of different

OpenCV functions operating on 3 Megapixel images for Android and IoS devices [14]. The study found that functions such as Gaussian blur, face detection, and canny edge detection took 592.62, 2471.53, and 145.00 milliseconds when running on a Galaxy Note 10.1 [14]. The results for the iPhone 5S were significantly worse with execution times of 1106.79, 6338.24, and 179.60 milliseconds. The 2014 work by Hauswald et al. studied the use of offloading when executing these types of computer vision pipelines. Their findings suggested that a hybrid offloading to a server using high-performance computing (CPU, GPU) produced the best results [35].

3.3 The Cloud vs Edge

While the cloud provides a logical solution for assisting applications with the execution of complex functionality, it is not ideal for video-streaming applications. These types of applications require low-latency between the end-device and server, large amounts of bandwidth, and close geographical proximity. When running the server-side application, each of these requirements is difficult to guarantee, and can be explained as follows:

1. **High Network Latency**

   When accessing the cloud, users and application developers have no control over the path between the end-device and server. Because of services such as network virtualization and middleboxes, network connections can be subjected to large amounts of network latency [18, 64]. For instance, within a large data center, because applications typically run within their own virtual machine (VM), network traffic can encounter delays due to large amounts of congestion based on VM placement. Because of the uncertainty of network latency, video-streaming applications can be subjected to lag and jitter which adversely affect the user experience.

2. **Poor Spectral Efficiency**

   While wireless LANs have high amounts of available bandwidth, it is possible to encounter paths across the open Internet which have significant bandwidth limitations. While this is typically not an issue when loading web pages, video-streaming applications require access to large amounts of bandwidth across each hop between end-device and server. Experiencing a bandwidth drop from 400 Mbps to 2 Mbps can be the difference between 80 milliseconds and 16 seconds when loading a 4 MB JPEG image [64].

3. **Geographical Barriers**

   When running applications in the cloud, users can find themselves located significant distances from the server where the processing takes place. For instance, an application could be deployed in US-East AWS availability zone. When accessing this application from the west coast, the path must traverse a large geographic distance before being processed by the high-end server.

Edge computing offers an alternative to centralized computing, in that resource constrained devices are able to connect to powerful, geographically-proximal end-points. Scenarios like home and enterprise networks can be tuned to better serve clients. Such an edge-computing based solution can significantly overcome the issues faced by PrivateEye. Specifically, the compute intensive operations can be performed remotely enabling continuous detection mode execution. Also, a good connectivity between the edge server and the client can ensure a high rate of frame delivery.

As opposed to making tradeoff decisions in order to run complex image processing algorithms on the end-device, the edge should be utilized to achieve the combination of both high-quality and realtime performance.

4

Detecting Distracted Driving

4.1 Background and Motivation

Distracted driving, wherein a driver is engaged in another activity that takes their attention away from operating the vehicle, is identified to be a significant cause of fatal motor vehicle crashes and injuries. According to a 2014 study conducted by the National Highway Traffic Safety Administration (NHTSA), 3,179 drivers were killed and 431,000 drivers were injured in accidents involving distracted driving[51]. The most typical distracted driving behaviors include but are not limited to cell phone use, talking to passengers, grooming, and adjusting the radio and navigation systems [2]. These behaviors require both cognitive and visual attention and increase the chance of an accident by nearly a factor of 3, according to a 2013 report conducted by the NHTSA [23].

Given the severity of the problem, detecting distracted driving has received significant attention from the research community, industry and government agencies [73, 22, 21]. Commercial solutions that collect onboard data e.g. fuel consumption and emergency alerts, and incentivize good driving are currently being

offered [4, 55]. There have been research efforts to detect cellphone usage based on acoustic or cellphone sensing [79, 74, 8], alcohol impairment using supervised and semi-supervised techniques [46, 48], and drowsy driving by tracking driver eye gaze [49]. Our system advances prior work by creating a mobile middleware system comprising a data collection framework which in turn relies on an inbuilt ensemble of deep learning techniques to enable us to detect a richer set of distracted driver classes and to better disambiguate between these classes by combining various sensing modalities.

We take a combined *Internet of Things* (IoT) and *deep learning* based approach to counter the problem of distracted driving. This is made possible by the significant proliferation of IoT devices e.g. embedded mobile sensors, that are capable of collecting a variety of information ranging from vehicle speed, acceleration and braking, to driver's heart rate and dashboard images. Each of these inputs provides richer contextual information that allows machine learning models to better disambiguate human behavior and enable fine-grained analysis. For instance, the image of a driver sending a text message can be cross-validated by checking the acceleration of the mobile device from the embedded accelerometer. Further, applications such as Waze [75] turn off usage in driving mode regardless of who is using the phone, whereas image analysis can discern whether it is actually the driver or passenger using the device. Such instances motivate us to explore multimodal analysis to improve classification accuracy without deploying expensive sensors. Further, with the advancements in Big Data and improvements in computational power, analyzing IoT data using deep learning allows for more powerful and accurate methods for performing this analysis.

Detecting distraction is critical for retroactively analyzing accident scenes, providing variable insurance rates, and providing real-time alerts to drivers and fleet managers. However, building such a system requires overcoming practical challenges to realize a working solution. First, *how* do we extract data being generated by mul-

tiple sources and at varying time scales such that they can be meaningfully analyzed? Second, *where* should the processing be performed - at the edge, remote server, or split between the two. To make this decision, the system must have a sense of processing capability, network bandwidth and latency. Finally, the system must provide certain *privacy* guarantees to the user when collecting IoT data, without sacrificing classification accuracy.

To this end, we present *DarNet*, a multi-modal data collection and analysis system designed to detect and classify distracted driving behavior. The system is composed of a data collection framework which collects streaming input from multiple sensors and an analytics engine responsible for combining the different modalities into a cogent output. The data collection framework can be further decomposed into *data collection agents* which run on each IoT device and a *centralized controller* designed to communicate with the agents. The *analytics engine* utilizes advances in deep learning to process the data received from the controller.

DarNet operates on two different data modalities - IMU data collected from the driver's mobile device and image data received from a dashboard camera mounted to face the driver. The system is currently configured to operate in a remote configuration where all data is shipped to a remote server for classification. In the remote configuration, the user has the option to down-sample the image data which serves as a basis for privacy, while also saving network bandwidth and diminishing latency.

The analytics engine builds on two advanced deep learning techniques for spatial and temporal data analysis, specifically Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). The system employs a CNN for processing the image data received from the camera and a RNN for processing the sensor data (IMU) received from the mobile device. An ensemble-based learning approach is employed to strategically combine the output of the two models to produce the overall classification. Our system is designed to make classifications at each time-

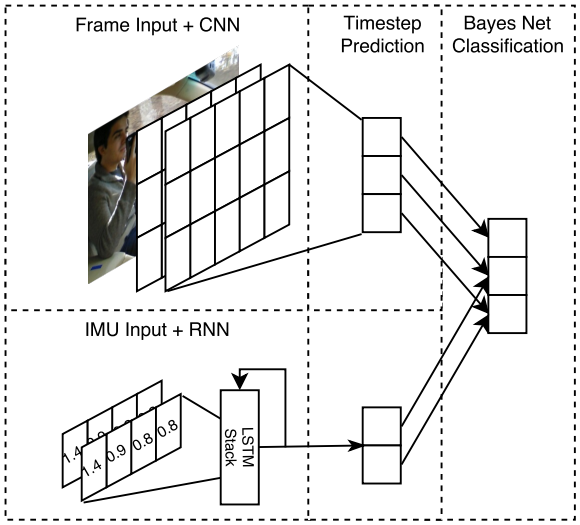


FIGURE 4.1: Diagram depicting the classification system where the probability distributions from models are combined using a bayesian network to produce probabilities for each class and the overall classification.

step from the data, making it amenable to near real-time detection.

Placing a camera inside a car imposes major privacy concerns that might deter individuals from using the application, despite a surge in their popularity [1]. To mitigate these concerns, we explore the feasibility and tradeoffs involved with minimizing the risk of potentially compromising data leaving the vehicle. We implement a methodology that allows for a distortion filter to be applied to the video data before processing. This design allows for a user to specify the level of privacy associated with their data, with the tradeoff of altering the classifications.

We make the following contributions in this paper:

- We build a data collection framework to collect and align data across multiple IoT devices and modalities. This is open sourced and can be useful for quickly collecting, aggregating and labeling data for data processing tasks.

- We demonstrate how to use deep learning across different IoT modalities - CNN for video analysis, and RNN for IMU time-series analysis. Further, we explored

[1] http://www.ceoutlook.com/2016/08/10/the-strange-emerging-dvr-market/

several techniques to *combine* these modalities and to the best of our knowledge, we present a novel Bayesian Network combiner approach to strengthen driver distraction detection.

- Because we are working with sensitive data, we take the first steps towards developing an unsupervised learning methodology for adding different levels of privacy to the system.

We believe this is an initial step towards developing middleware solutions to processing IoT sensor data in a scalable, useful and private manner. By making the software and learning models available to the general research community, we hope to facilitate future work in this field of important and emerging technology.

4.2 Acknowledgements

The contents of this chapter are based on the publication "DarNet: A Deep Learning Solution for Distracted Driving Detection" [68] which appeared in Middleware 2017. This was a collaborative work with Ramya Raghavendra, Theophilus Benson, and Mudhakar Srivatsa.

4.3 Related Work

Detecting distracted driving is receiving significant amount of attention from the research community and the industry. There is a wealth of data that has been has been made available to the research community to study this problem [73, 22, 21].

There have been efforts that recognize driver distraction using data extracted from wearables, cellphone and onboard diagnostics to obtain sensing information such as accelerometer, gyroscope, etc. to model phone usage while driving. Additionally, using this data to collect unsafe lane changes [79, 74, 8, 48] is an active

area of research. A detailed review that discusses various driver condition monitoring techniques can be found in recent survey papers [17, 52].

By integrating powerful sensors into top-end cars (e.g., cameras, radar, and ultrasonic sensors), manufacturers are bringing similar forms of driver behavior monitoring to the consumer market [5, 20, 12]. While safety technology is getting cheaper, few safety features are available in entry-level vehicles. In contrast, smartphone solutions such as *DarNet* can be used with little additional expense in all cars today. Commercial solutions such as Automatic [4] and LDWS [50] provide tracking and emergency services using OBD and GPS data but require special hardware. Progressive Insurance offers a service, Snapshot [55] that uses OBD-II data from a mounted device to detect driving behavior. To the best of our knowledge, there is no commercial solution that combines video and sensor data to analyze driving behavior.

Recognizing distracted driving behavior (i.e., driver drowsiness, lane departure, and following distance) using fixed vehicle-mounted devices is an active area of research. Computer Vision techniques and learning-based methods [46, 80, 78] are used to extract features corresponding to eye-tracking, driver postural stability, cellphone usage etc. out of driving images or video.

Our work builds on prior work by combining the use of the powerful Convolutional Neural Networks (CNN) [41, 71, 72] on image data with the use of a Recurrent Neural Network (RNN) for temporal data. These networks have been shown to be effective for analyzing sequential information specially for NLP tasks, and are recently gaining popularity in other applications [33, 36, 54, 19]. To the best of our knowledge, our work is the first to use an ensemble neural network to combine video and sensor data to produce accurate classifications for distracted driving. We believe applying deep neural network ensembles on temporal and spatial data to distracted driver detection problem opens up several exciting possibilities, enabling us to easily learn non-linear relationships between sensors and driver behavior in a scalable fashion

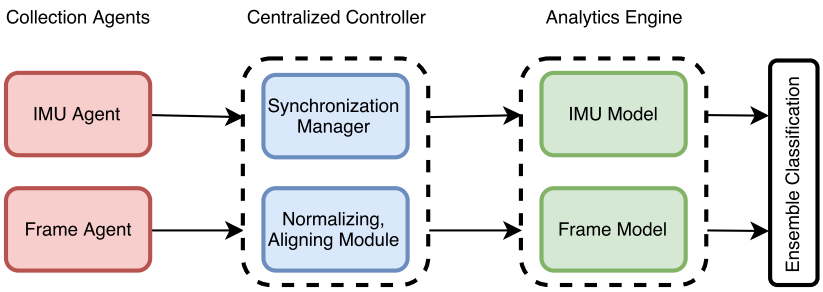


FIGURE 4.2: Diagram depicting the collection agents which run embedded within IoT devices, the controller which collects and prepares the data, and the analytics engine which performs the ensemble classification.

(e.g., centripetal acceleration to detect phone usage).

Next, *we build a unified neural network framework to analyze multimodal sensing data* obtained from dashboard videos and IMU sensors, and without the need for adding expensive sensors (such as hi-res cameras). Combining modalities helps us easily identify contexts that a single modality does not provide e.g. *is the driver or passenger using the phone?* . Multimodal deep learning approach has received recent attention [57] for being able to identify nonintuitive features largely from cross-sensor correlations which can result in a more accurate estimation, with prior work taking this approach to detect human activity recognition through sensor data from wearable devices.

Recent work by Kim et.al. looks at the issue of sharing of private dashcam videos under anonymity [40]. Privacy of driving data, even before it leaves the car, is a critical issue going forward and one that has not received much attention. We take a step towards anonymizing *dashcam video data* while still ensuring classification can be performed. We explore heterogeneous learning architectures wherein we train a CNN to reproduce the feature activations learned by the deep model. We expand upon this approach by training a deep CNN to reconstruct features from down sampled images using a similar error minimization methodology. The problem of multimodal data anonymity will be the subject of our future work.

4.4 System Design

DarNet aims to tackle the task of detecting distracted driving by providing a well connected framework for performing data collection and analysis. Integral to the design of *DarNet* is the proliferation of available IoT sensor devices. We therefore design our system to allow for new devices to be easily incorporated into the existing framework. The more data available for processing and training, the better the overall performance of the system.

4.4.1 Collection Agents

The *collection agent* is designed as an abstraction to allow for easy-integration into a broad range of IoT devices. The responsibilities of the agent include periodically polling the device's sensor, maintaining an internal clock for timestamping the data, and transmitting the data to the centralized controller at a specified frequency. The periodicity at which the agent polls the sensor should be determined based on the operating frequency of the sensor itself, while the transmission frequency should be determined based on the latency and bandwidth between the agent and the controller. The implementation of each agent is specific to the system and sensors in which it is embedded.

4.4.2 Centralized Controller

The *centralized controller* has several responsibilities which include: aggregating, smoothing, and aligning data received from the agents, maintaining clock synchronization between the agents, and deciding how and where the data should be processed by the analytics engine.

Data Normalization: The controller receives periodic updates from each agent at different frequencies based on the agent's configuration, network latency, and the order in which the operating system interleaves the received network packets. As

such, the agent relies on the timestamp associated with each tuple to determine the ordering of the collected data. Because the timestamps for data received from different agents will not align exactly, the controller uses interpolation to fill in the gaps, and to aggregate the data at consistent intervals. Additionally, the controller performs a smoothing operation on the data by maintaining a sliding moving average. Because the system runs on commodity sensor hardware, we expect the data measurements to fall within a bounded range of error, therefore the smoothing average normalizes the data against any aberrations or irregularities.

Clock Synchronization: We have implemented a protocol between the agents and controller to maintain clock synchronization. In this protocol, the controller maintains a *system time* and pushes this time to each agent. Whenever an agent receives an updated system time, the agent will update its own clock to reflect that of the controller's, plus an additional constant to account for network latency. This protocol is set to run periodically in order to account for internal clock drift of each agent.

Processing Decision: In determining where the data should be processed, the controller can choose between a local and remote configuration. A remote server would have a greater amount of processing power, and hence is a computationally better option over the local configuration that runs on the device. However, under poor network conditions, the controller has the option of processing all data locally, albeit slower. When remote configuration is selected, the user has the option of specifying the *degree of privacy* at which the image data is transmitted to the analytics engine. Privacy is added to the system by down-sampling the image data to varying sizes. The advantage of this technique is that it not only obfuscates the user's identifying information, but also improves bandwidth by transmitting less data.

4.4.3 Analytics Engine

The *analytics engine* is responsible for processing the data streams and producing a classification. The engine is designed to be entirely modular – the system maintains a 1-to-1 relationship between device data-streams and machine learning models. Rather than training a single model to process the data, we train individual models and combine the results at a later stage. This design provides two benefits: 1. New devices can be incorporated into the network without requiring the existing models to be retrained. 2. Each machine learning model can be specified based on the type of data streamed from the device i.e. spatial models for image data, temporal models for IMU data. The engine combines the results of each model to produce the overall classification.

4.5 System Implementation

We have designed and deployed an application making use of this system which collects data from an inward facing camera and mobile device. *DarNet* collects and processes the data from these devices to perform driver behavior classification.

4.5.1 Data Streaming Framework

We have developed two Android modules — i) the *collection agent* which interfaces with IMU data sensors and the camera hardware, and ii) the *centralized controller* which aggregates data from the agents, maintains clock synchronization, and formats the data for processing. We have made the source code for these modules available to the public [2].

Data Collection Agent: In our setup, we use a Nexus 7 tablet running Android 6.0.1 (API v23) to emulate the dashcam, and a Nexus S device running Android

[2] https://github.com/cstreiffer/sensordataclient
https://github.com/cstreiffer/driverdatarecorder

4.1.2 (API v16) acting as the driver's cellphone. Each *collection agent* operates by registering *listeners* to receive updates from the device's embedded sensors. Each listener receives updates every 25 ms through Android's *sensor manager*.

The agent running on the Tablet is responsible for interfacing with the embedded camera, while the agent running on the mobile device interfaces with the *accelerometer, gyroscope, gravity, and rotation sensors*. Each agent is designed to communicate with the controller over Bluetooth or 802.11 point to point wireless.

Centralized Controller: The controller runs on the Nexus 7 tablet. Upon startup, the controller will open a two-way communication channel with each collection agent registered by the system. Since we are collecting data with multiple modalities, synchronizing the timestamps across the devices is important for further analysis. We implement a *timestamp manager* in our application with master-slave architecture for clock distribution with the controller acting as the master and distributing its UTC timestamp to the agents over their communication channel. The agent sets its own clock to the master's UTC, plus the empirically measured network delay. Because the system clock is highly susceptible to drift, this synchronization process is repeated every 5 seconds.

We have implemented *DarNet* to perform all data processing on a remote server, in both the normal and privacy-preserving settings. When the controller receives an update from an agent, it will perform the aforementioned aligning and normalization techniques, and store the data in a time-series database e.g. statsd. The data is transferred and processed in an offline manner.

4.5.2 Analytics Engine

Within our system, we make use of the advances in deep learning to perform our distracted driver analysis. We utilize two different types of neural networks – a Convolutional Neural Network (CNN) for performing classification on a per-frame

basis, and a Recurrent Neural Network (RNN) for classifying the time-series data. CNNs have proven to perform exceptionally well at image classification tasks, while RNNs excel at analyzing time-series data by discovering non-linear dependencies which are not visible to other learning methodologies.

Frame-Sequence Architecture: Because our approach for detecting distracted driving requires analyzing image frames, we choose to employ a convolutional neural network to facilitate the classification by performing feature extraction at each timestep. The CNN is first trained to perform the task of image classification on the distracted driver dataset, allowing for the convolutional filters to learn the corresponding activations for the given classes. Once fully trained, the output of the final convolutional layer is used as the output feature vector for the corresponding image. This approach exploits the ability of the CNN to learn spatial features in order to produce a lower dimensionality vector representation of the original image.

We use the Inception-V3 [72] architecture for performing frame classification. We made the decision to use this model because of its exceptional performance on the ILSVRC 2012 classification task [72]. The design of the model exploits the "hebbian principle" [3] which proposes that spatially close neurons should activate together, and is a refinement over GoogleNet [71], focusing on maximizing computation efficiency while allowing for additional expansion of layers. Due to the large amount of time required for training deep networks, we take a fine-tuning approach by initializing our model using the weights of a pre-trained model. We use the model made available through the Tensorflow source code which has been trained to a 78.7% Top-1 classification percentage on the ILSVRC 2012 [29] dataset as our initialization point. We modify the final fully connected layer of this network, such that the number of outputs corresponds to the number of driving classes.

The network is trained to minimize the softmax-cross-entropy error between the output logits and corresponding labels. We employ label-smoothing regularization

(LSR) to deter over-fitting and produce a more robust model [72]. This LSR methodology replaces the label distribution $q(k) = \delta_{k,y}$ where $\delta_{k,y} = 1$ for the target class with the following equation:

$$q'(k) = (1 - \epsilon)\delta_{k,y} + \epsilon u(k)$$

where $u(k) = 1/K$ is the normal distribution over the K different classifications. This results in the following loss equation:

$$H(q, p) = -\sum_{k=1}^{K} \log p(k) q'(k)$$

$$H(q', p) = (1 - \epsilon)H(q, p) + \epsilon H(u, p)$$

where ϵ is a smoothing parameter $\in [0, 1]$.

We train the model using stochastic gradient descent with added momentum to avoid over-estimating the gradient. We train the model using an initial learning rate of 10^{-3}, decayed every 2 epochs at an exponential rate of 0.25. We use the TensorFlow [1] implementation of the RMSProp algorithm with a momentum of 0.9, decay of 0.9 and ϵ of 1.0. We maintain a running average of the weights of the parameters and use these values to run the evaluation on our test dataset.

We train the model using a batch size of 64 across 10,000 steps. We parallelize the training across 4 NVIDIA Tesla K80 GPUs by placing an instance of the model on each GPU. Each batch is divided between the models, and the resulting gradients are averaged together before updating the weights.

IMU-Sequence Architecture: Because we perform the distracted driving classification over a sequence of inputs, we choose to use a recurrent neural network to analyze the IMU-sequence data and features extracted by the CNN from the frame-sequence data. These networks have been proven to excel at modeling and determining nonlinear aspects of timeseries data, making them perfect for our

classification task. The network captures temporal relationships by maintaining an internal state h_t that is a function of all prior perceived inputs. The network computes the output vector y_t for a given timestep t using a non-linear function that combines the hidden state h_t with the current input x_t. Over a given input sequence $\mathbf{x}$=$(x_1, x_2, ..., x_t)$, the network will compute the hidden state and output using the following equations:

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_t)$$

$$y_t = W_{hy}h_t + b_y$$

where f is an element-wise activation function, h_{t-1} is the previous state of the network, h_t is the current state, and W represents the weight matrices for the internal neurons.

We choose to utilize the Long Short Term Memory cell [37] within our network because of its unique ability to capture long-term dependencies between input features while diminishing the vanishing gradient problem inherent to these networks. The computation for the hidden state h_t within this architectures follows:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)$$

$$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o)$$

$$h_t = o_t \tanh(c_t)$$

where σ is the sigmoidal activation function, and i_t, f_t, c_t, and o_t represent the input, forget, cell, and output gates. The computations made by each gate allows for the network to determine which features are important for the current and future outputs.

We use a *deep bidirectional architecture* [34, 81] to implement our network. The *deep* aspect of the architecture refers to stacking LSTM cells by connecting the output h_t of one cell to the input of the next cell in the stack. The *bidirectional* aspect of the architecture refers to implementing both a forward and backward propagating network. Because the standard design of these networks only captures dependencies from prior inputs, we utilize the backward propagating network to capture dependencies from future inputs [34]. We combine the output of the two networks to produce the output vector y_t using the following equation:

$$y_t = W_{fh}h_{ft} + W_{bh}h_{bt} + b_y$$

where h_{ft} is the output from the forward propagating network and h_{bt} is the output from the backward propagating network [34].

We made the decision to use this type of network over support vector machines because of the RNNs ability to capture long-term dependencies between input features while diminishing the vanishing gradient problem inherent to these networks. The architecture for the RNN consists of 2 bidirectional LSTM cells containing 64 hidden units. Because we use a sampling frequency of 4Hz and a time window of 5 seconds, the network is trained and evaluated on a sliding window of 20 data points.

Ensemble Learning: Because the RNN and CNN output probability distributions for a different set of classes, we implement a Bayesian Network (BN) [25] to combine the outputs into a single inference. Each class is assigned its own BN consisting of two parent nodes and a child node. We compute the conditional probability tables (CPTs) for each class based on the number of true-positive observations from the training data presented to the system.

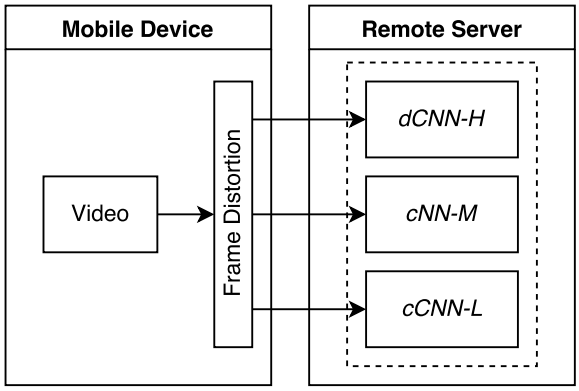


FIGURE 4.3: The privacy implementation for the system where the paths between the mobile device and server correspond to the different levels of downsampling – 100x100, 50x50, 25x25.

4.5.3 Privacy Preserving Analytics

In Figure 4.3, we present the privacy design for processing the down-sampled video data. This novel architecture introduces two unique components: the distortion module which down-samples the video data, and a set of deep learning models for performing feature extraction. We design this portion of the system to be minimally invasive — each model is trained using unsupervised techniques which do not require any knowledge of the original dataset.

System Implementation: In our design, the distortion module down samples the video according to user-specified preference and tags the video with the down-sampling rate. At the remote server, the analytics engine picks the appropriate classifier for performing feature extraction on the distorted video. In *DarNet*, we provide three levels of distortion – low, medium, and high, thus explaining the different paths to the remote server.

The image data is distorted using nearest neighbor down sampling to the sizes of 100x100 (dCNN-L), 50x50 (dCNN-M), and 25x25 (dCNN-H) pixels. Being able to reduce the image from 300x300 pixels these sizes represents approximately a 9x, 25x, and 144x decrease in amount of data required for transmission. This is ideal for

FIGURE 4.4: From right to left: the undistorted image; the image distorted to the size of 100x100 pixels; the image distorted to the size of 50x50 pixels; the image distorted to the size of 25x25 pixels.

situations where a user desires a high amount of privacy, or where network conditions are poor and lower resolution frames must be transmitted to the remote server. Examples of the down-sampled frames can be observed in Figure 4.4.

Training Methodology: The motivation behind the training methodology stems from the success exhibited by de-noising auto-encoders [77, 39]. In our work, we train an additional CNN to reproduce the classifications results of the original model. We reuse the Inception-V3 architecture and initialize the weights using the CNN trained on the driving dataset. Because the goal of the training is for the dCNN model to mimic the behavior of the CNN model, we believe that this initialization methodology provides a good starting point for the weights. This technique allows for the training to be completely unsupervised, and allows for new data to be incorporated into the system.

The methodology is described as follows: 1. The image is passed through the original CNN and the output of the final layer is recorded. Note: In this design, the original image does not have to leave the device to perform this action. 2. The image is down-sampled and sent to the remote server with the appropriately marked tag. 3. The server aggregates the data, the distortion level, and the prediction results observed from passing the image through the original CNN. 4. Finally, the dCNN model is trained by passing the down-sampled image through the model,

and comparing the observed output against the recorded output. The loss-function computes the L2 euclidean distance between these two vectors, and the model is trained using stochastic gradient descent as the optimization technique. This has the effect of training the dCNN model to perform a de-noising operation on the image to produce accurate predictions.

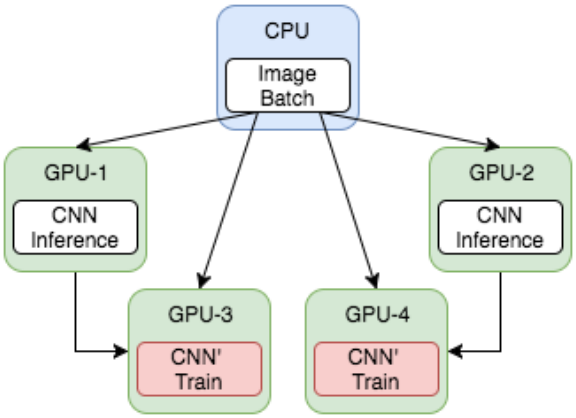


FIGURE 4.5: The training configuration for the dCNN models.

We implement a distributed training approach for the dCNNs using 4 Nvidia K80 GPUs. A diagram of this design can be observed in Figure 4.5. In this distributed setting, we place instances of the base CNN on 2 GPUs to perform inference, and place instances of the dCNN on the remaining 2 GPUs to perform the backpropagation and gradient descent. The CPU is responsible for transmitting the original frames x_i to the base CNN models, and the distorted frames x'_i to the dCNN models. This configuration can be observed in Figure 4.5. The dCNN models are trained using an initial learning rate of 10^{-4} decayed every 5.0 epochs at an exponential rate of 0.16.

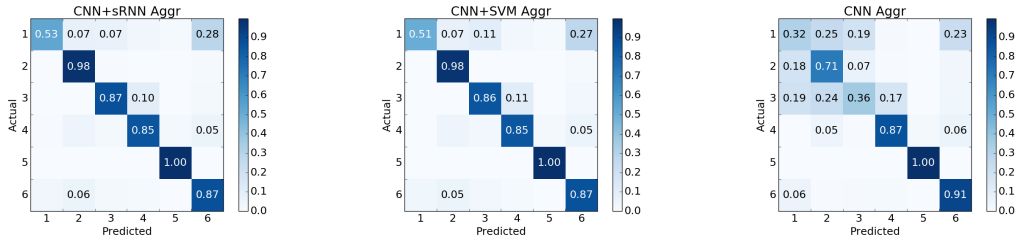
4.6 DarNet Evaluation

We evaluate our framework by running *DarNet* on a dataset collected using the data collection component. In our evaluation, we look to answer the question *how can*

different modalities be incorporated together to strengthen classifications? We present results showing how an ensemble approach produces stronger classifications than a single modality, and how deep learning can strengthen these results. Additionally, we present results showing the efficacy of *DarNet* to operate on the down-sampled frames in our privacy configurations.

Class	**Description**	**Data Types**	**Frame Count**
1	Normal Driving	Image, IMU	5,286
2	Talking	Image, IMU	10,352
3	Texting	Image, IMU	9,422
4	Eating/Drinking	Image, —	9,463
5	Hair and Makeup	Image, —	4,848
6	Reaching	Image, —	17,709

Table 4.1: Driver behavior classes listed by corresponding model types and amount of data collected.



(a) (b) (c)

FIGURE 4.6: Confusion matrices for the collected datasets for architectures: (a) CNN+RNN (*DarNet*) (b) CNN+SVM (c) CNN (frame data only).

4.6.1 Evaluation Setup

Data Collection: To perform our analysis, we collect datasets from 5 drivers performing the driving tasks listed in Table 4.1. Each driver was given the same route to drive, used the same vehicle, and drove under varying degrees of lighting. Each driver was instructed (by the passenger, in real time) to perform a scripted

set of "distractions" for a duration of 15 seconds and the entire script was repeated 10 times for each driver. Each video was verified at a later point in time for the accuracy of the given driving test. We divide the collected dataset into an 80/20 partition for running the training and evaluation on the ML models.

The complete list of distraction classes we consider and the corresponding number of collected data points are outlined in Table 4.1. The data type column refers to the types of data we were able to collect for the given task. For instance, for the eating/drinking category, while we were able to collect image data, the mobile device was placed in the "Normal Driving" position, and thus, we were not able to collect any IMU sequence data specific to the distraction task. The classes are reflective of the distraction classes recognized by Statefarm insurance company [3].

In this paper, we take the first step towards demonstrating the strength of our classifier by positioning the client mobile device in one of five varying orientations during data collection. The Texting orientation corresponds to the driver holding the phone between waist and eye level in either the left or right hand, while the Talking orientation corresponds to the driver holding the phone to either ear. The final orientation corresponds to all remaining classes and consists of the device being positioned horizontally in the drivers front-right pocket.

Model	Hit@1
CNN+RNN	87.02%
CNN+SVM	86.23%
CNN	73.88%

Table 4.2: Ensemble model Top-1 classification results for the collected dataset.

4.6.2 Evaluation Results

We evaluate the performance of *DarNet* against two additional architectures. In the first architecture, we only consider the CNN portion of the system operating on the

[3] www.kaggle.com/c/statefarmdistracteddriverdetection#evaluation

frame data. In the second architecture, we combine the CNN frame architecture with a support vector machine (SVM) trained to classify the IMU sequence data. The Top-1 classification results for the three architectures can be found in Table 4.2, while the resulting confusion matrices can be observed in Figure 4.6.

The results show that the top performing model is the selected architecture for *DarNet* which consists of the CNN for frame classification and RNN for IMU sequence classification. While each method produces strong classifications results, Figure 4.6 shows that all three models output a high number of false positives when predicting normal driving. From a safety perspective, classifying normal driving as distracted driving is acceptable in that the overall safety of the driver is not diminished by the incorrect classification. However, from a usability perspective, a high false positive rate for distracted driving would diminish the user experience. Because of the small size of the dataset, we believe that these false positives can be reduced by training across data collected from a larger participant study.

Out of the three architectures considered, both ensemble approaches (CNN+RNN, CNN+SVM) significantly outperform the single modality approach (CNN) with Top-1 classification percentages of 87.02% and 86.23% compared to 73.88%. These results show that adding the IMU sequence data to the frame data has the effect of improving the accuracy of the system by ¿16.7%.

For the per-class results, the biggest improvement when using multiple modalities is the system's ability to discern between texting, talking, and normal driving. For instance, the CNN model produces a classification accuracy of 36.0% for texting, whereas the CNN+RNN produces an accuracy of 87.0%. Similar improvements can be observed for normal driving and talking. Overall, the ensemble approach is able to eliminate the majority of the noise between these three groups as observed in Figure 4.6c. Among the classes which did not have corresponding IMU sequence data, there was an observable drop in accuracy for eating/drinking and reaching, of

2% and 4% respectively. Whereas the CNN model misclassifies reaching as normal driving, both ensemble models produce a higher talking misclassification rate for this category of ~5%. An explanation for this result is that the movement that occurs when reaching for an object adds enough noise to the IMU data to produce a talking classification. The complete confusion matrix for each class and model can be observed in Figure 4.6.

Between the ensemble models, using the RNN architecture over the SVM corresponds to a ~1% increase in performance. When considering the amount of time an individual spends in their vehicle during the day, a 1% increase in the amount of accurately classified driving behaviors could be the difference in deterring an accident. When running the RNN and SVM models on the IMU sequence dataset alone, the RNN produces a classification accuracy of 97.44% compared to 95.37% for the SVM. Because the RNN architecture better captures non-linear dependencies between the data, we believe that the CNN+RNN architecture is best suited for this system.

4.6.3 Denoising-CNN Results

The dCNNs were trained and evaluated on a previously collected distracted driver dataset. The dataset consists of 18 classes, and was collected from a total of 10 drivers. All video was recorded using a GoPro Hero 3 camera at a rate of 30 fps.

Model	Hit@1
CNN	78.87%
dCNN-L	80.00%
dCNN-M	77.78%
dCNN-H	63.13%

Table 4.3: CNN and dCNN Top-1 classification percentages for the alternative distracted driving dataset.

As can be viewed in Table 4.3, the most surprising result of the evaluation was the dCNN-L outperforming the other networks by achieving a Top-1 classification per-

centage of 80.00%. In comparison, the baseline CNN achieved a Top-1 classification percentage of 78.87%.

While the dCNN-L performed the best, the most promising results from the evaluation were the performance of the dCNN-M and dCNN-L models. As can be viewed in Table 4.3, the dCNN-M achieved a Top-1 classification percentage within <1% of the baseline CNN, while the dCNN-H was able to produce the moderately-accurate Top-1 classification results of 63.13%. As can be viewed in Figure 4.4, the distortion levels for dCNN-M and dCNN-H render the image almost unidentifiable. Future work is still required to determine how effective these distortion techniques are for preventing adversarial networks from performing classification tasks e.g. facial recognition.

Why does the dCNN-L outperform the baseline CNN? The biggest anomaly in our results was the improved performance of the dCNN-L over the baseline CNN. A potential explanation for this phenomenon is that the baseline CNN displays effects of over-fitting accrued during training. The loss of information from the image data mitigates these effects, to a degree. As can be observed by the results of dCNN-H, too much loss of information results in significantly diminished performance, leading us to suspect that there is a local distortion maxima that results in the optimal performance. Although we are fascinated by these results, we leave further investigation for future work.

4.7 Feature Extraction Runtime Comparison

Because we have developed an application for collecting, and aligning frame-by-frame data with IMU-data, the infrastructure exists for implementing real-time detection. The bottleneck for the detection time depends on the inference time required by the deep networks. Specifically, the feature extraction performed by the CNN incurs a high overhead [43]. Because it is infeasible to currently run these deep networks on

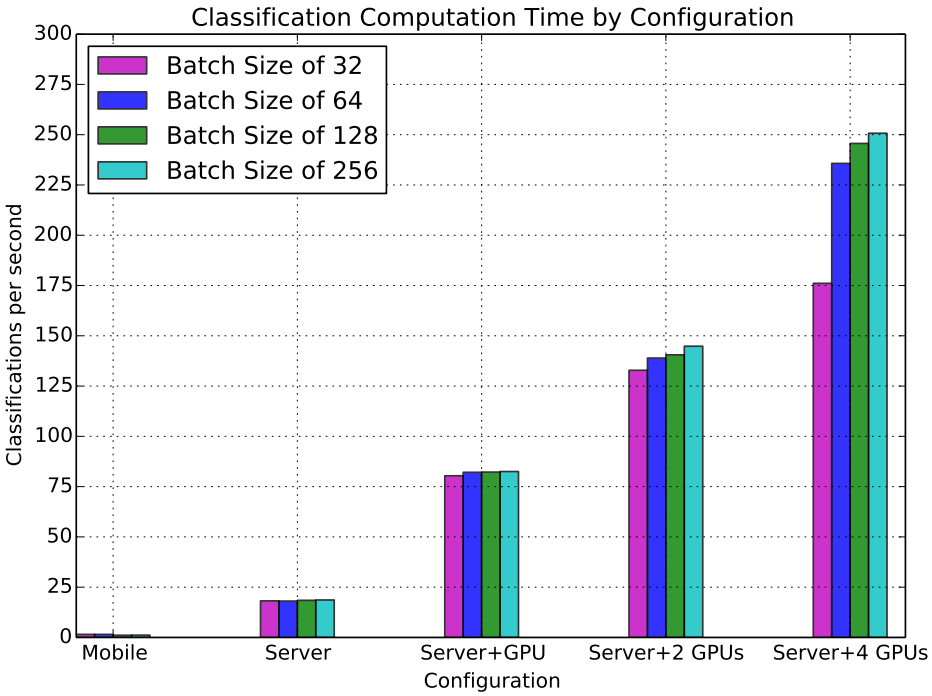


FIGURE 4.7: Configuration specifications and runtime results for each model, with the following abbreviations: CPU-A – 1.3 GHz Intel Core M, CPU-B – 2.65 GHz Intel Xeon E5-2640 v3 Haswell, GPU – NVIDIA Tesla K80s.

mobile devices at the desired rate of 30 fps, we must rely on an offloading based solution. Optimizing these deep networks to run locally on the mobile device or on an edge device is currently an active area of research [28].

We evaluate and compare the runtime of the CNN across multiple device configurations to show how performing classifications in a high-performance cloud infrastructure significantly improves processing speed. The results of the evaluation can be observed in Figure 4.7. Across a batch-size of 32 frames, the commodity CPU runs at a rate of <1.5 fps, falling well below our desired rate of 4 fps. Switching to a high-performance CPU, the performance improves to >18 fps. Adding a GPU further improves performance by running at >80 fps. Increasing the number of GPUs and batch size further improves the performance. Across a batch size of 512 frames split between 4 GPUs, we were able to achieve a performance of >261 fps. The results

indicate that high-performance edge computing provides processing rates well above the fps required by DarNet.

5

From the End to the Edge

5.1 Background and Motivation

With the ever-more evident need for sensitive data protection in businesses and the provision of privacy guarantees to application users, PrivateEye has shown to be a viable solution in protecting data in real time. PrivateEye uses advanced computer vision methods to detect and block regions within a frame that contain sensitive information. Integrated within the Android camera subsystem, PrivateEye functions as middleware between the camera driver and applications accessing the camera service. PrivateEye is able to deliver frames at a median rate of 20 FPS [60], providing reasonable quality of service for users while simultaneously blocking sensitive regions within the video frames.

A deeper look inside PrivateEye's system design reveals that PrivateEye runs in two modes: (1) Detection, and (2) Tracking. In detection mode, PrivateEye performs analysis on incoming frames to determine which regions have been marked public by the user. During tracking mode, PrivateEye finds the location of the last detected public region in incoming frames. While detection mode is computationally intensive

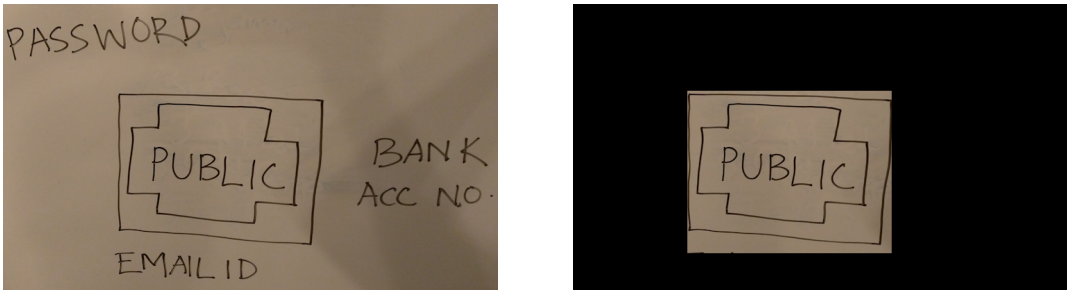
(slow), tracking is conversely light on computation (fast). As a result, PrivateEye runs in detection mode periodically (on 10% of the frames) and employs tracking mode in the intervening intervals.

While this method of operation helps PrivateEye to achieve a high frame delivery rate, the use of tracking inherently results in occasional inaccuracies in determining the location of the public region and causes a small drop in recall. Furthermore, since computer vision algorithms are imperfect, the use of tracking can result prolonged exposure of a region that was wrongly judged as public during the last detection mode.

Edge computing [26] offers an alternative to centralized computing, in that resource constrained devices are able to connect to powerful, geographically-proximal endpoints. Scenarios like home and enterprise networks can be tuned to better serve clients. Such an edge-computing based solution can significantly overcome the issues faced by PrivateEye. Specifically, the compute intensive operations can be performed remotely enabling continuous detection mode execution. Also, a good connectivity between the edge server and the client can ensure higher rate of frame delivery. This forms the motivation behind exploring the feasibility of PrivateEye as an edge computing based system, ePrivateEye.

5.2 Acknowledgements

The contents of this chapter are based on the publication "ePrivateEye: To the Edge and Beyond!" [69] which appeared in the Symposium on Edge Computing, 2017. This was a collaborative work with Animesh Srivastava, Victor Orlikowski, Yesenia Velasco, Vincentius Martin, Nisarg Raval, Ashwin Machanavajjhala, and Landon P. Cox.



(a) (b)

FIGURE 5.1: A region on a whiteboard marked as public using PrivateEye's marker (a), and An app's view while capturing the same region using PrivateEye's software enabled smartphone (b).

5.3 PrivateEye Overview

Before we dive into the details of ePrivateEye's system design, we provide some details of PrivateEye's inner workings. PrivateEye is a system whose aim is to provide more control over the information being accessed by a third-party app to the user. PrivateEye focuses on two-dimensional regions and takes a privacy marker approach [59] wherein a user marks a two-dimensional area as a public region using a special marker as shown in Figure 5.1a. Subsequently, when a camera-enabled application captures the same two-dimensional region when using a PrivateEye enabled smart device, PrivateEye intercepts the image data obtained by the camera, detects the marked region, blocks the remainder of the region, and delivers the data to the application (Figure 5.1b).

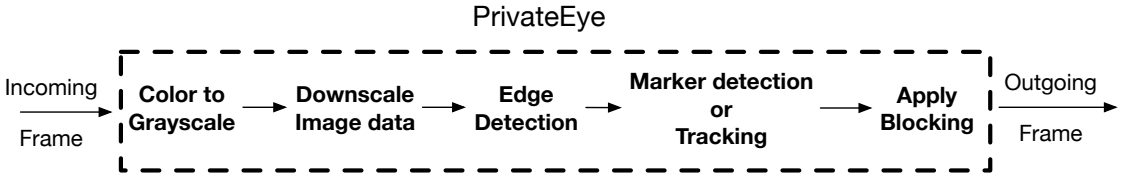


FIGURE 5.2: Series of computer vision algorithms applied on image data under PrivateEye.

Inside PrivateEye's logic for discerning public and private regions, each individual

camera frame undergoes significant processing. First, the colorspace of the frame is converted from color to grayscale; most computer vision algorithms operate in grayscale in order to achieve faster results. Next, the grayscale image is scaled down to a size of 800×480 for further processing. PrivateEye detects edges in the frame using the well-known Canny [11] algorithm. After edge detection, the system detects contours in the edge-based image using Suzuki's algorithm [70]. The algorithm returns the contours in a tree data structure which preserves the contour hierarchy. PrivateEye runs an approximation to find a parent contour that can be approximated as a rectangle and a child contour that has 12 corners with area at least 50% of the parent rectangle contour. This is the region marked by the user. Finally, the system finds the bounding box for the marked region and blocks everything else in the image.

As mentioned earlier, since the marker detection algorithm is slow, PrivateEye runs the detection algorithm periodically. For other frames, the system employs tracking on the last observed marked region in the current frame using the Lucas-Kanade optical-flow-in-pyramids technique [10]. Figure 5.2 depicts the series of transformations a frame experiences under PrivateEye.

The entire PrivateEye module is integrated into the trusted portion of Android itself; PrivateEye is contained within the camera service, which resides between the camera sensor and the application framework. Every frame captured from the camera sensor is received by the camera service, before being sent to the associated application. PrivateEye monitors the frames received by camera service and applies all required image transformations before sending the frame to the application.

ePrivateEye is an extension of PrivateEye and uses the same techniques employed in PrivateEye. To get more details about the PrivateEye, we would encourage the readers to read the original work [60].

5.4 Design Principles

In this section we present the design principles behind ePrivateEye. Our primary goal was to achieve near real-time performance for camera apps. With this as our motivation, we designed the server and client component of ePrivateEye to minimize network latency, while also reducing traffic size.

Reduce Client-side Computation: Since the client runs on a resource constrained device, we identified all the compute-intensive functionality and offloaded it. When a frame is received by the ePrivateEye client running inside the camera service, it is processed in the following fashion: (1) A copy of the frame data is saved, and a numeric identifier associated to it, (2) A grayscale color conversion is performed, and (3) The grayscale image data is downsized to a resolution of 400×240. The remainder of the image transformations shown in Figure 5.2 are part of the server component of ePrivateEye. After subsequent processing by the server that results in the marked region information being available for the frame, the client blocks the region outside the marked region and returns the modified frame data to the camera service.

Reduce Network Traffic: Again, since the ePrivateEye client runs on a resource constrained device, it is often the case that the software on such devices is designed to save power by reducing the amount of data transmitted, where possible. In our case, sending a frame to the ePrivateEye server at its original resolution (e.g. 1280×960) would cost significant time and energy, thereby negating some of the gains we would hope to achieve through offloading processing. We instead chose to reduce the amount of traffic sent by the client to the server by performing the sequence of relatively compute-inexpensive image transformations described above before sending the frame to the server. Our original frame data was encoded in NV12 format frames. Since edge detection performs satisfactorily on grayscale images, each

frame is downsampled to this format, resulting in a size savings of 33%. We then further downsample the data by converting it to a resolution of 400×240, as the computer vision algorithms we use provide acceptable performance at that resolution. Through these transformations, we reduce the size of the data transferred by the client from $1280 \times 960 \times 1.5$ to 400×240.

On the server side, we reduce the amount of traffic sent to the client by returning the coordinates of the detected regions only, rather than the masked frames.

By making the design choice to perform image blocking transformations on the client, we were able to reduce the total amount of traffic significantly. As mentioned earlier, these operations have a small computational overhead; the reduction in transferred data size makes the slightly increased computational cost of retaining these operations on the client a worthwhile trade.

Minimize Latency: PrivateEye is implemented to interact in a non-blocking fashion with the camera service. In this design, the camera service submits a frame to PrivateEye for processing, before making a request for the most recently processed frame. If the frame is available, PrivateEye will return the masked frame to the service. Otherwise, no frame is returned and the camera service will wait for the next frame to be received from the camera sensor.

To minimize latency between the client and server, we made several modifications to the TCP sockets maintained by each. On the client side, we disable Nagle's algorithm on each socket. Nagle's algorithm purposefully delays frames in order to reduce the number of TCP packets sent over the network, and thereby adversely affects latency sensitive applications. Disabling the algorithm forces packets to be sent over the socket as soon as they are received, thus reducing latency. On the server side, we look to reduce the number of `read` syscalls made by the server by modifying the receive buffer. We only allow for the server to make a `read` on a socket once a certain number of bytes have arrived. We set the number of bytes to be equivalent

to the total size of the expected frame.

To minimize latency when transmitting across the metropolitan network, we utilize Software Defined Networking (SDN) to control the path along which the data is transmitted. This allows for a "fast path" to be established between the access point (AP) and server. This implementation will be discussed in more detail in the following section.

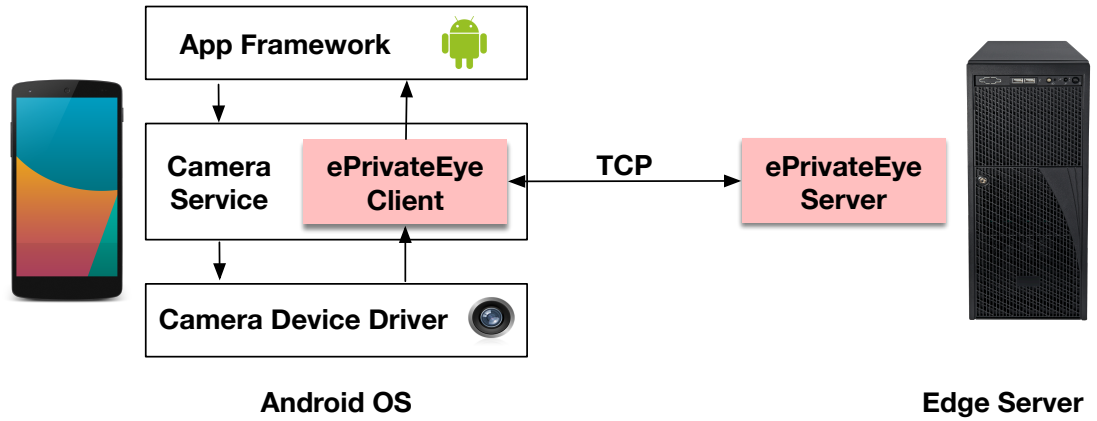


FIGURE 5.3: ePrivateEye architecture where the client component runs as part of camera service, a trusted module of Android OS, and the client and server maintain a persistent TCP connection for as long as the app session runs.

5.5 Implementation

In this section we provide the implementation details of ePrivateEye. We also discuss the various network configuration used for the evaluation purposes. We implemented ePrivateEye client by modifying the Android Open Source Project (AOSP) version of Android 6.0.1. Our prototype currently runs on a Nexus 5 smartphone. We implemented ePrivateEye server as a standalone desktop application running on Linux, using the OpenCV libraries to perform various computer vision tasks. In our implementation, the version of OpenCV used is 2.4.13.

5.5.1 ePrivateEye Design

ePrivateEye Client: The client module has three primary responsibilities. The first is to receive frames from the *camera service*, and send them to the server for processing. The client maintains a configuration file which contains the server's IP and port. When ePrivateEye is first instantiated on the device, it opens 3 persistent TCP connections with the server. As soon as the client receives a frame, it places the frame on a queue, *InQueue*, for processing. At this point, a thread with a dedicated TCP connection retrieves the frame from *InQueue* and sends the frame to the server for processing. The second responsibility is to apply the requisite mask to the frame once a response has been received from the server, and to enqueue it onto another queue that the client maintains, *OutQueue*. Upon being enqueued in *OutQueue*, the frame for which the mask information has been received is dequeued from *InQueue*. The client's final responsibility is to communicate to the *camera service* that the processed frame is now ready to be sent to user-level apps.

ePrivateEye Server: The server module runs an event-driven, desktop version of the original *PrivateEye*. The primary responsibility of the server is to perform the computationally expensive marker detection algorithm. The server receives the frame from the client and immediately begins looking for any marked areas. If the server detects a marked region, it sends the corresponding bounding box coordinates which enclose the marked region to the client. Otherwise, the server returns a null response for the frame.

5.5.2 Server Configurations

Both the client and server modules are designed to be completely portable. This allows for us to construct varying network configurations around the client and server for evaluation purposes. Within our work, we consider three different network configurations running ePrivateEye, and compare the performance against the original

PrivateEye running within a local configuration on the mobile device. Because these configurations are applicable to an enterprise setting, we assume that all transmitted data never leaves the confines of the network, and thus remains private from the public internet.

Edge: The *edge* configuration consists of ePrivateEye running on an Access Point (AP). We configured a laptop with a 1.3 GHz Intel Core M CPU and 8 GB of RAM as the AP. The mobile device connects to the AP in order to establish a bridge to the internet. In doing so, the mobile device is able to directly access ePrivateEye. In this configuration, the client and server are separated by a single hop, and communicate using the 802.11ac wireless protocol, with a maximum possible transmission bandwidth of 433 Mb/s.

Building: The *building* configuration consists of ePrivateEye running on a server connected to the same wireless router as the mobile device. The server has the same specs as those described in the *edge* configuration. In this configuration, the mobile device and server are within close proximity of one-another and communicate over the same network using 802.11n, with a maximum bandwidth of 300 Mb/s.

Metro: The *metro* configuration consists of ePrivateEye server and the client being separated across a metropolitan area. The server is equipped with 6 Intel Xeon Processor 2.10 GHz v4 CPUs with 128 GB of RAM. As can be observed in Figure 5.4, the total distance between the client and server is more than 2 miles. Because of this large distance, we utilize Duke's SDN infrastructure to maintain control over the path through which the communication must occur. In this configuration, the mobile device connects directly to the AP, and the AP connects to the server across a path primarily composed of fiber. Because the transmission time across fiber is approximately two-thirds of the speed of light [67], this line of communication offers the fastest possible path between the AP and server. The maximum bandwidth between the AP and server is 1 Gb/s and has an approximate ping time of 0.593 ms.

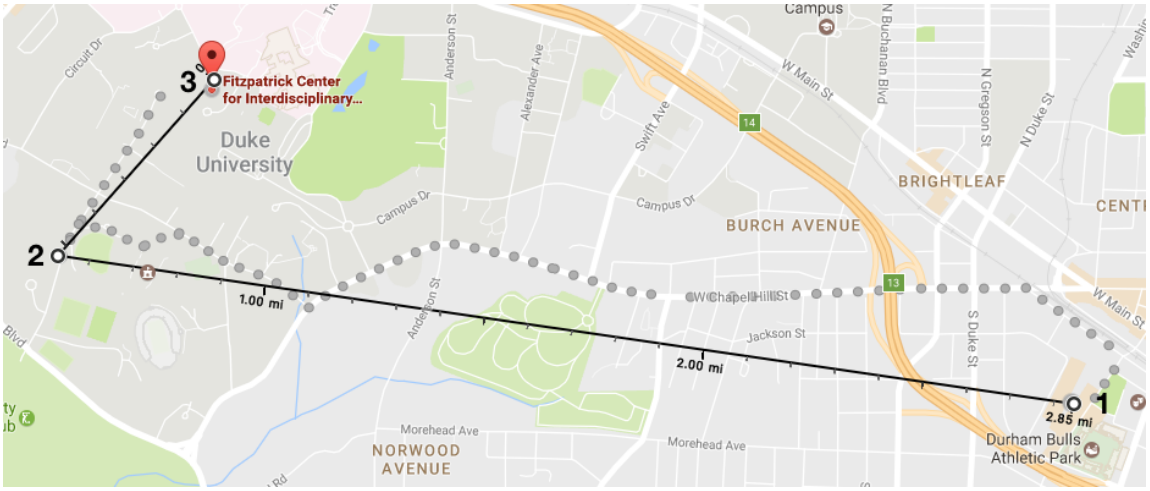


FIGURE 5.4: Metro configuration where the AP resides within the American Tobacco Campus (1), connects to an Arista 7050 SDN switch in TelCom (2), which finally connects to a Cisco 4500X SDN located within Fitzpatrick East datacenter that contains a NVIDIA Tesla P100 GPU (3), totalling a line distance of 2.85 miles with a corresponding ping time of 0.5ms.

In order to ensure maximum bandwidth and minimal latency, we re-use the previously described AP configuration to create a bridge between the mobile device and server. In this configuration, the AP uses a Belkin USB-C Ethernet adapter to interface with the SDN infrastructure. The mobile device connects to the AP using the 802.11ac wireless protocol. The AP serves as a bridge between the mobile device and remote server. As a result of the control of the forwarding elements afforded by the SDN, the AP and server appear to be separated by only a single hop. The maximum possible bandwidth between the server and client is 433 Mb/s. As a result of the near-instantaneous transmission between the AP and server, however, this configuration allows for similar performance as if ePrivateEye was running on the AP.

SDN Fast Path: We leverage Duke's SDN infrastructure to construct a *fast-path* between the AP and server, eliminating any unpredictable sources of latency that might have been incurred along the normal production path. In a production environment, SDN would provide the ideal setting for deploying ePrivateEye because of

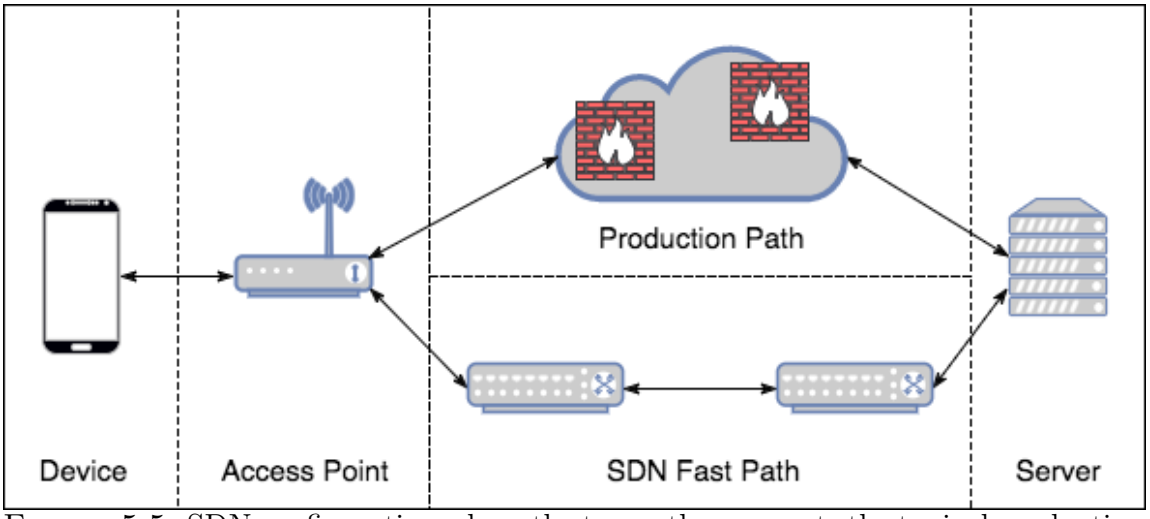


FIGURE 5.5: SDN configuration where the top path represents the typical production path through the network, while the bottom path represents the SDN path which traverses two Cisco 4500X and one Arista OpenFlow switches, and the choice of path is selected through a controller not pictured in this diagram.

its ability to create on-the-fly firewall policies, and because it can provide a trusted, fast-path connection between the client and server. Utilizing Duke's SDN allowed us to construct a path that is primarily fiber and consists of the following hops: (1) The Nexus 5 connects to the AP using 802.11ac which creates a bridge to the SDN infrastructure across a Belkin USB-C Ethernet adapter. (2) The Ethernet adapter connects to a Cisco 4500X OpenFlow switch. (3) The first switch connects to a second OpenFlow switch (an Arista 7050) via a DWDM. (4) The second switch connects to a third OpenFlow switch (another Cisco 4500X) located in one of Duke's primary datacenters. (5) The third OpenFlow switch is connected via fiber to a Tesla P100 GPU-enabled server running the ePrivateEye server software. Please refer to Figure 5.4 for an approximation of this route. The connection between the AP and first OpenFlow switch is a standard copper Ethernet cable, but all remaining links in the topology are using fiber optic cable.

As can be observed in Figure 5.4, the line distance between the AP and server is approximately 2.85 miles, while the ping time between the two is approximately

0.5ms. As a result of the fiber links and the programming of the forwarding elements, the *fast path* very nearly gives the impression that the AP is connected directly to the server. Since the mobile device connects directly to the AP, however, this configuration, in reality, contains a total of 5 hops between the device and the server.

Figure 5.5 shows both the typical network configuration, and the SDN configuration for the paths between the AP and server. When the packets traverse the production path, they must be processed by firewalls and other network elements that can add non-deterministic latency to the communication stream. Using Duke's SDN, these sources of unexpected latency can be bypassed through the programmatic control of the forwarding path between the AP and the server. This ensured that our data took a path that resulted in minimal variance for our measurements.

5.6 ePrivateEye Evaluation

ePrivateEye uses the same computer vision techniques that are used in the existing system, PrivateEye. ePrivateEye is designed to provide better performance and realtime app experience to a user. Therefore, we focus our evaluation on the performance aspect of the system under various settings. We, specifically, want to answer the following:

1. Can ePrivateEye achieve real-time frame delivery to an app?
2. Can ePrivateEye continue to provide reliable recognition of marked regions?
3. What is the cost of running ePrivateEye on a resource constrained smartdevice?
4. How does increase in number of marked regions in the camera view affect ePrivateEye's performance?
5. What are the network characteristics required to support smooth functioning of ePrivateEye?

To answer the first two questions, we prepared a video benchmark with typical camera settings and camera movements. We used these videos to compute frames per second (fps) delivery, precision, and recall with which ePrivateEye reveals public regions. Next, we measured the power consumption, CPU usage, and memory consumption to show the impact of running ePrivateEye on a smartdevice's resources. Finally, we evaluated the impact of multiple marked regions on the scalability of our system.

To understand the network support required for a good performance, we investigated the *metro* server configuration where the server and client are separated by a distance of 2.8 miles. Specifically, we measured the impact of network delays and packet losses on the frames per second delivery. Further, we investigated the case of cloud infrastructure where ePrivateEye server is run on Amazon Web Server (AWS).

5.6.1 Experiment Setup

Video Benchmark: To carefully study the accuracy and impact of ePrivateEye for different camera settings and motions, we developed a simple video benchmark. We recorded several videos, each 20 seconds in length, at a resolution of 1280×960 at 30 fps using a Nexus 5 smartphone. The average bitrate for each video was computed to be 9321.33 kb/s.

We focused on three different two-dimensional regions: whiteboard, presentation, and laptop screen. We marked a public region on each two-dimensional region and recorded the video under the following three camera motions: (1) *Still*: the camera was steady and remained focused on the marked region for the entire video, (2) *Spin*: the camera rotated while remaining focused on the marked region, and (3) *Scan*: the marked region first moved out of the camera view and then came back in. The three camera motion *still*, *scan* and *spin* simulate the scenario of still photography, video recording, accidental capture of public regions, and changing orientation of the

recording device. We recorded videos for all the three two-dimensional regions under the three different camera settings. As a result of space constraints, we only show results obtained for public regions marked on whiteboard.

To test ePrivateEye, we modified the camera service to read the pre-recorded video files and load them into the memory. When a camera frame is delivered from the camera sensor to the camera service, the camera service replaces the frame data with the data coming from the pre-recorded videos and passes them to the apps. For the app, the frames appear to arrive directly from the camera sensor. Using this setup, we measure precision, recall, and performance of ePrivateEye under different scenarios.

ePrivateEye Setup: Our goal is to show the performance of the edge-computing based system, ePrivateEye, under each of several different scenarios and, at the same time, present a comparison with the local-computing based system, PrivateEye. Therefore, we configured ePrivateEye server in different ways to simulate those scenarios: (1) On-device: where the server runs on the smartphone; this simulates the previous system PrivateEye, (2) Edge: where the server runs on an access point to which the smartphone (ePrivateEye client) is connected directly, (3) Building: where the ePrivateEye server is reachable via a router, and (4) Metro: where the server and client are separated physically by a large distance and connected through different networks. The details of the various server configurations are previously described in Section 5.5.2.

There is one subtle difference between PrivateEye and *on-device* configuration of ePrivateEye. As previously mentioned, PrivateEye achieves higher fps by running primarily in tracking mode, with periodic transitions to detection mode. We must note, however, that in our *on-device* configuration ePrivateEye performs detection on every frame; we did so in the interest of having the code remain invariant while performing comparisons using differing server configurations.

5.6.2 Frames per Second

To compute fps we set the *still*, *spin*, and *scan* videos to loop, replay them for 30 seconds, and record the times at which each frame is finished processing. We then process the logs to compute the rate of frame delivery for each configuration.

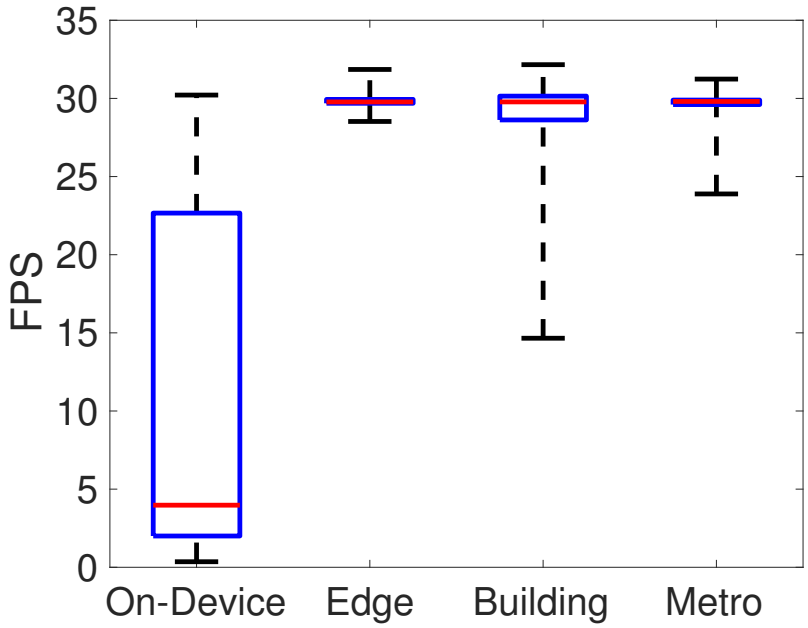


FIGURE 5.6: FPS results for the *still* camera movement.

As can be observed across Figures 5.6, 5.7, 5.8, ePrivateEye significantly outperforms PrivateEye, achieving a median frame rate of $\approx$ 30 fps across each remote configuration. Because the processing time for a single image on the server was $\approx$ 6ms, ePrivateEye can handle frame rates exceeding 30 fps. In comparison, the *on-device* configuration achieves a median frame rate of $\approx$ 5 fps, equivalent to an 83.33% reduction in performance. Although PrivateEye has proven to have a frame rate of $\approx$ 20 fps, the detection of marked regions in consecutive frames causes the performance to deteriorate. ePrivateEye offers a significant improvement in quality of service over that provided by PrivateEye, by running in real-time with no noticeable lag or delay.

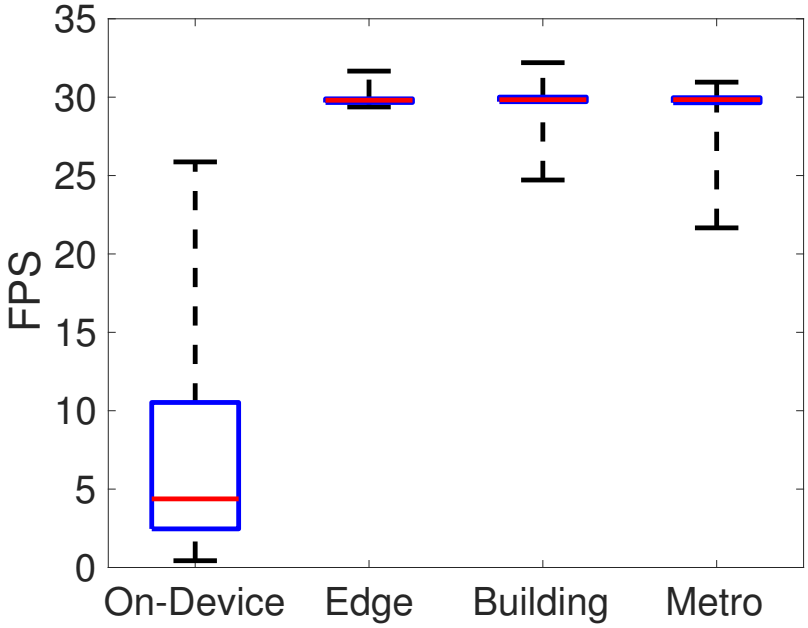


FIGURE 5.7: FPS results for the *spin* camera movement.

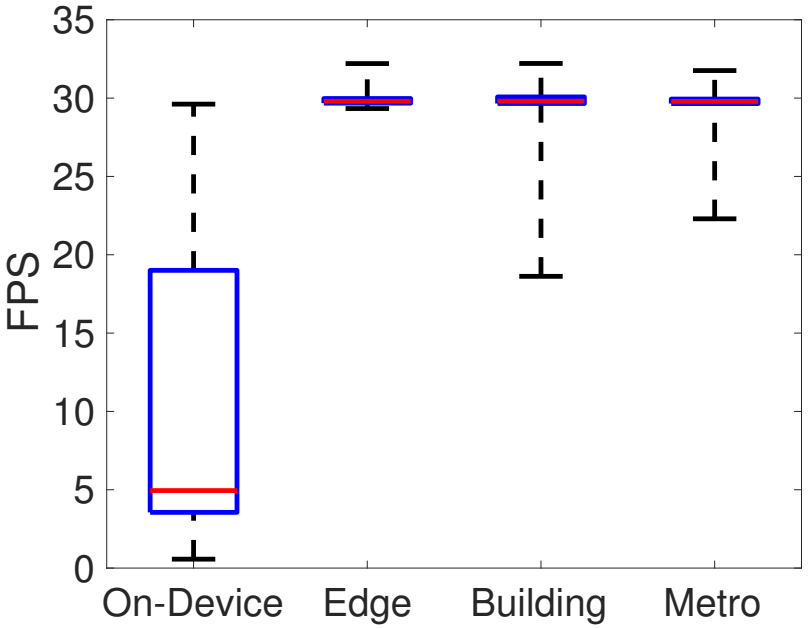


FIGURE 5.8: FPS results for the *scan* camera movement.

While the median fps for all the camera motion settings are around 5 fps, the *on-device* configuration delivers some frames at $\approx$ 18 fps for *still* and *scan* camera motion settings. In case of *still* the marked region is in focus, which eases the task

of detection for the software. In case of *scan*, since the view is constantly changing, the frame data can become blurry, and the software does not always have as many distinct features in the image to process. This results in occasional fast processing of the frame and faster delivery of the frames. In the *spin* case, the object stays in the focus throughout the video, but the orientation changes constantly. This causes the detection to run on every frame with many features to process, and, as a result, never reaches a high rate of frame delivery. This observation is in agreement with the frames per second evaluation of PrivateEye [60].

Between the remote configurations, the *edge* configuration produces the most consistent results with the lowest amount of standard deviation. As this configuration establishes the shortest path between the client and server, these results are expected. The *metro* configuration produces the second most consistent results, as can be observed in Figures 5.7 and 5.8. Although the standard deviation for the *building* configuration is higher than the other two configurations, it still achieves a median frame rate of ≈ 30 fps.

The reason that the *metro* configuration has a more consistent performance than the *building* is because of two reasons. First, the SDN establishes a *fast path* between the AP and server which has minimal overhead due to the selection of the path. Second, within the *building* configuration, the mobile device and server both utilize the 802.11n wireless, whereas within the *metro* configuration, the mobile device communicates using the 802.11ac protocol, which has a higher maximum bandwidth. Further, because the mobile device and server are both connected to the router over the wireless connection, there is a higher probability for packet loss and latency to occur. This shows the importance of using SDN – even though the client and server are separated by a few feet in the *building* configuration, the *metro* configuration, where the client and server are separated by 2.85 miles, produces superior performance results.

Overall, these results show that realtime performance can be achieved across each remote configuration of ePrivateEye, with the *edge* offering the least deviation from the optimal performance of 30 fps.

5.6.3 Precision and Recall

To compute precision and recall, we divide the prerecorded videos into pixel cells of size 5×5. We manually mark all the cells inside the marked region as *public* and cells outside the marked region as *private*. Our system should reveal *public* cells and block *private* cells. We consider a pixel cell from a frame to be blocked if none of it is visible in the corresponding frame processed by ePrivateEye; otherwise we consider the cell to be revealed by our system. This is a strict enforcement as we assume that a partially visible cell can have enough information to reveal the entire cell.

Formally, for a given frame, let C_t and C_r be the number of cells marked public and number of cells revealed by the system, respectively. Then we define precision and recall for that frame as follows:

$$precision = \frac{|C_t \cap C_r|}{|C_r|} \qquad\qquad recall = \frac{|C_t \cap C_r|}{|C_t|}$$

While precision represents how secure is our system, recall represents how usable the camera frame is to an app.

Figures 5.9, 5.10, 5.11 show that ePrivateEye achieves a 100% precision for all camera motion settings. This indicates that ePrivateEye does not let any sensitive information through to any app. Further, the evaluation reveals that precision and recall results remain constant across each network configuration running ePrivateEye. This is an expected outcome as change in network settings has nothing to do with the detection of marked regions as long as the frame data doesn't get modified while in-transit. This indicates that running ePrivateEye in any remote configuration will

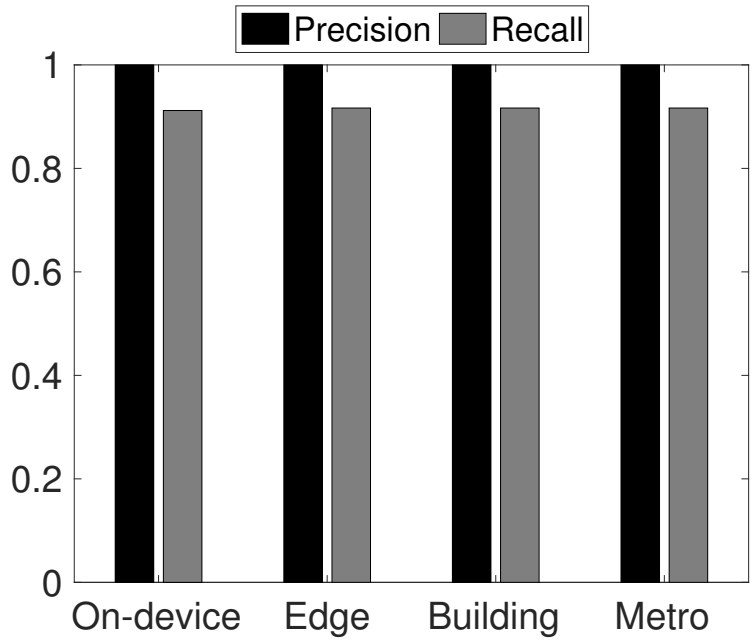


FIGURE 5.9: Precision and recall for *still* camera motion.

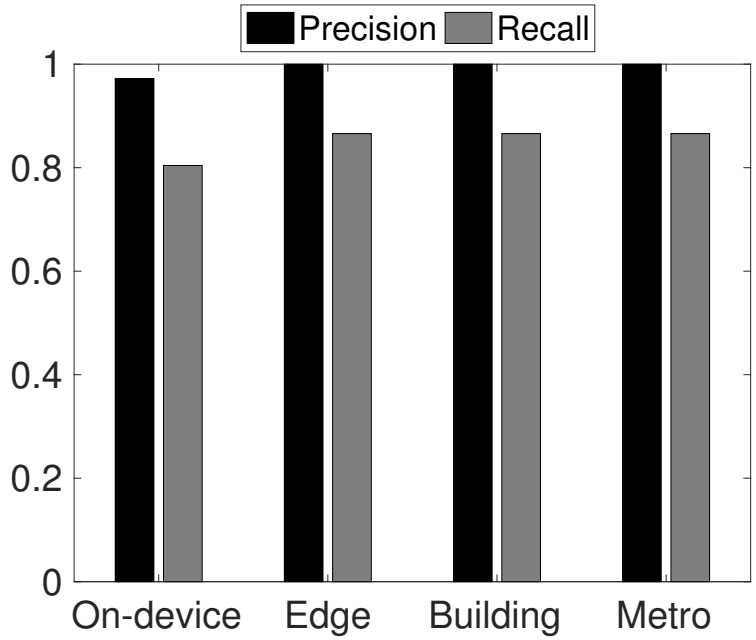


FIGURE 5.10: Precision and recall for *spin* camera motion.

produce strong precision results.

On-device configuration provides a median precision of 0.96 in case of *spin* camera setting, while achieving a 100% precision for *still* and *scan* settings. In *still* setting,

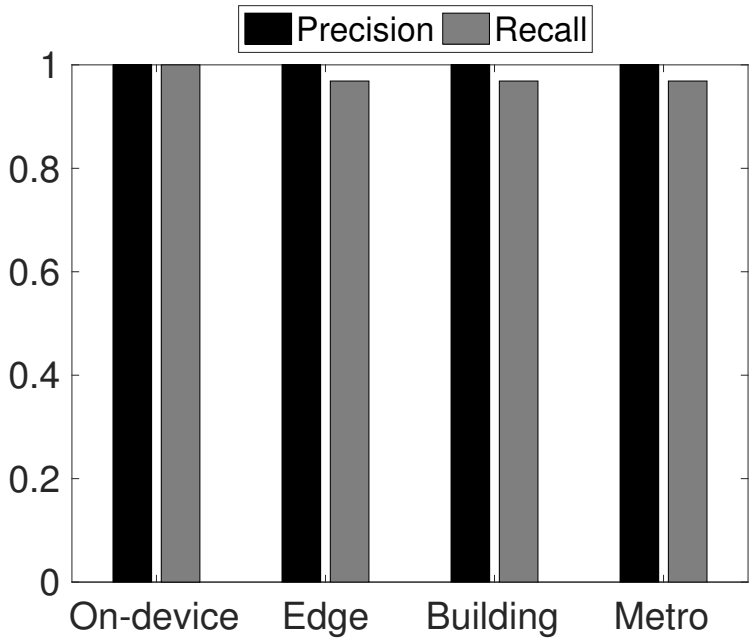


FIGURE 5.11: Precision and recall for *scan* camera motion.

the camera is focused on the marked region which helps in picking up the features to perform detection effectively on the device.

The precision for *on-device* configuration under *spin* camera setting drops slightly. Under this camera motion, although, the marked region stays in focus of the camera, its orientation changes constantly which makes the detection of key features slightly difficult. However, for the same camera setting, all the remote configurations achieve higher precision. We suspect that this difference in precision values across configurations comes from the OpenCV libraries used by the ePrivateEye. The OpenCV libraries undergo different optimizations for them to work on resource constrained ARM-based platform. As the remote configurations have significantly stronger processing power, they are able to take full advantage of the functionality that OpenCV has to offer.

The cost of higher precision for ePrivateEye comes with the tradeoff of decreased recall performance. In comparison to PrivateEye, ePrivateEye has slightly weaker recall results. Because recall is indicative of usability, the remote configurations of

ePrivateEye underperforms in this category. However, the performance does not suffer too much as ePrivateEye still achieves a minimum recall of 82%. Overall, although the recall performance decreases when using ePrivateEye, the perfect precision coupled with the high rate of frame delivery as observed in the previous section, makes ePrivateEye the superior implementation in terms of both security and QoS.

5.6.4 Resource Usage

We wanted to investigate the impact of running ePrivateEye on a mobile device's resource usage. To do this, we designed an experiment where we recorded a 30 second video containing one marked public region. We set the camera setting to be *spin*. We ran this prerecorded video 5 times through ePrivateEye and measured CPU usage (in percentage), memory usage (in MB) and power consumption (in mW). For this experiment, we fixed the smartphone's brightness level to 50% and rebooted the device between trials. During all the trials, we kept the wireless radio active.

We took measurements every 100 ms using Trepn [38], a performance monitoring tool developed by Qualcomm. Trepn reports very accurate measurements for the devices that come with Qualcom's snapdragon processor and Nexus 5 is one of them. We used this app and recorded the CPU usage, memory consumption and power consumption.

In case of memory consumption, offloading compute intensive processing has no impact. The ePrivateEye client maintains a pre-allocated memory to hold a constant number of frames that needs to be processed by the server component. Due to this, we do not see any significant change in memory consumption (Figure 5.12).

Figure 5.13 shows that ePrivateEye server running on the device causes a higher median CPU load ($\approx$ 50%) than that of the case when the server runs remotely ($\approx$ 40%). Interestingly, even after offloading the computationally intensive part of processing, the difference between the CPU load is not very high. The reason is

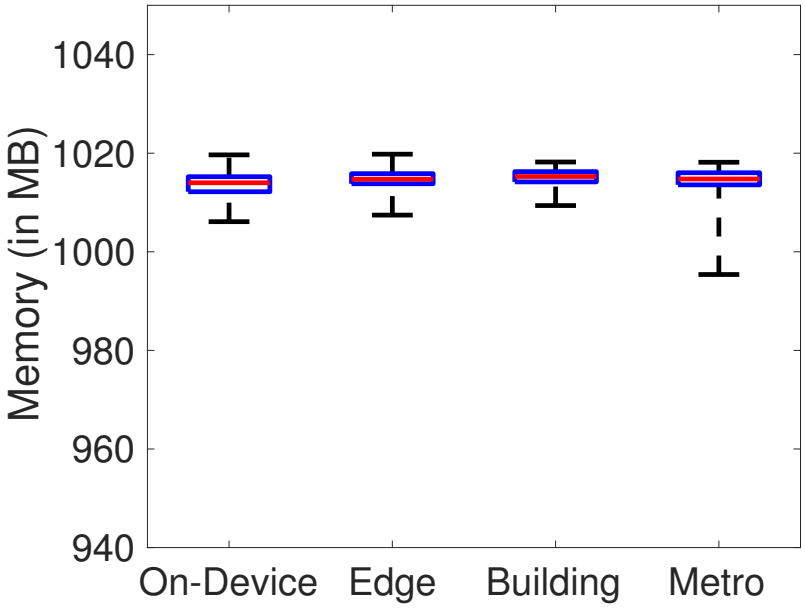


FIGURE 5.12: Memory usage for analyzing a video for 30 seconds under different configurations for ePrivateEye server.

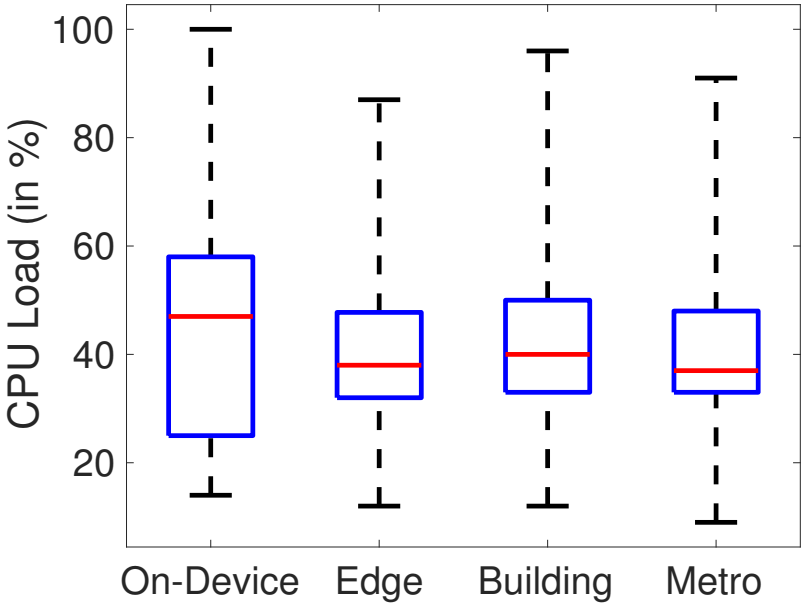


FIGURE 5.13: CPU load for analyzing a video for 30 seconds under different configurations for ePrivateEye server.

that in case of remote processing (*edge* , *building* and *metro*), the CPU stays active as it can process significantly more frames (real-time processing). As shown earlier,

ePrivateEye processes 500% more frames per second when it is offloaded.

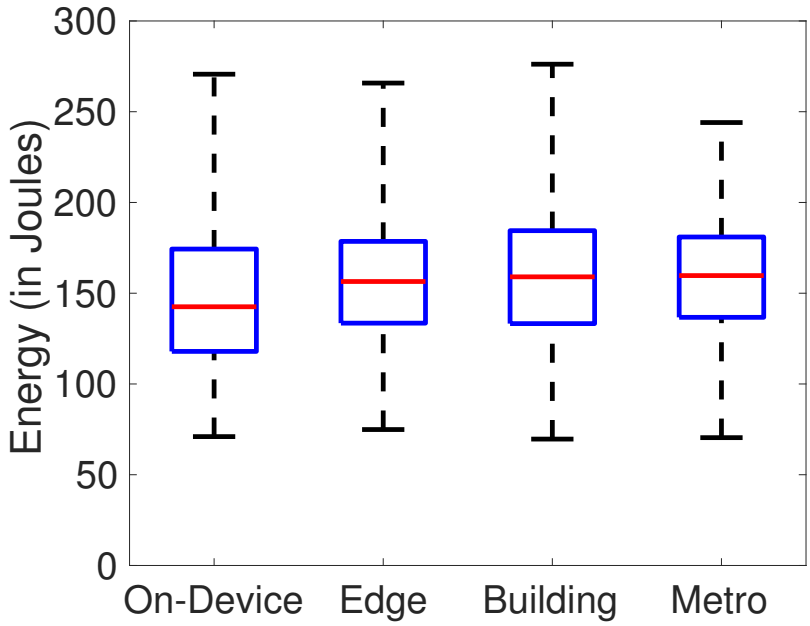


FIGURE 5.14: Power consumption for analyzing a video for 30 seconds under different configurations for ePrivateEye server.

Similarly, while the power consumption across all the configurations appear comparable (Figure 5.14), the real gains can only be understood when the results are looked with the number of frames processed per second. Figure 5.14 and Figure 5.7 show that on-device processing consumes $\approx$ 144 J while delivering frames at a median rate of 5. On the other hand, processing frames on a remote server, ePrivateEye consumes $\approx$ 156 J while delivering frames at a median rate of 30.

5.6.5 Scalability

To measure scalability, we expand our video benchmark to include three additional 20 second videos containing 1, 2, and 3 marked regions respectively under the camera setting *still*. These videos were recorded at the previously mentioned resolution and frame rate on a Nexus 5 smartphone. The purpose of adding more marked regions to the frame is to increase the number of computations required to run the detection

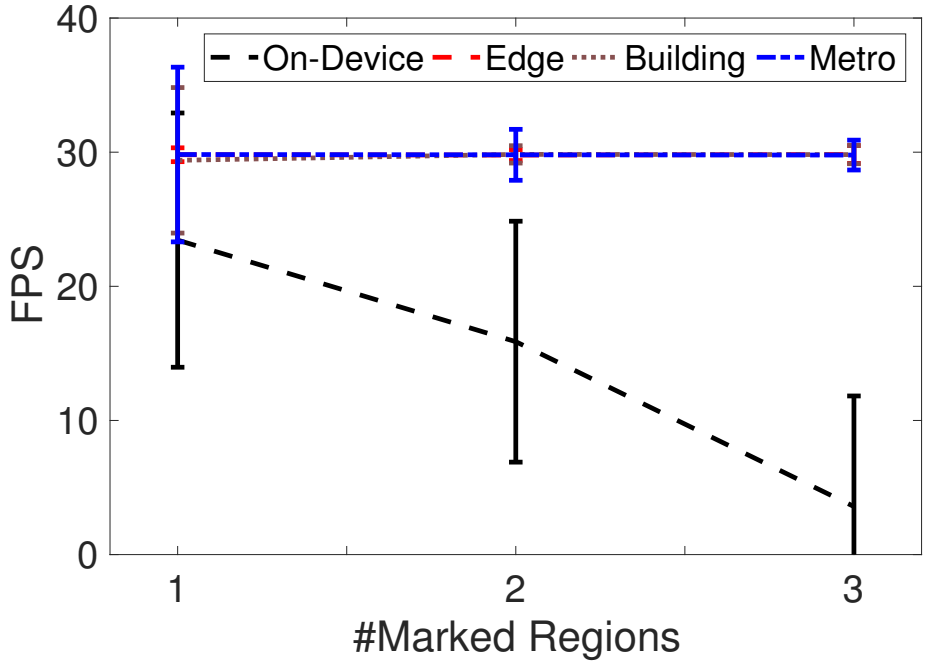


FIGURE 5.15: Median rate of frame delivery under different configuration of ePrivateEye server.

algorithm. As more regions are introduced, the computational complexity increases which places a larger load on the CPU on a per-frame basis. When measuring the performance, we replay these videos for their full duration and record the processing time for each frame.

The evaluation results for PrivateEye and ePrivateEye can be observed in Figure 5.15. As is expected, the fps performance for PrivateEye decreases as the number of marked regions increases. The median frame rate for the video with 1 marked region is 20 fps, while the median performance for three marked regions is 5 fps, corresponding to a 75% performance decrease. As the number of marked regions increases, PrivateEye struggles to scale and the usability suffers.

In comparison, the plots for ePrivateEye displayed in Figure 5.15 show that the median frame rate across each network configuration remains at 30 fps. This shows that performing the processing on the server does not add any additional latency as the number of marked regions increases. For the video with 3 marked regions, the

median performance for ePrivateEye offers a 500% improvement over the *on-device* configuration (30 fps v. 5 fps). These results show that ePrivateEye can continue to offer realtime processing rates as the complexity of the operation increases.

5.7 Case study: Metro

We wanted to investigate the impact of poor network conditions on the performance of ePrivateEye. Because the *fast path* configuration represents a near optimal connection between the AP and server, we wanted to measure the performance of ePrivateEye under less than ideal conditions. To create these conditions, we induce both latency and packet loss on the egress connection of the server. We then leverage the use of the SDN to eliminate all external sources of latency and packet loss, in order to ensure that the only source of the performance decay is from the ingress location. This allows us to construct a *slow path* and *lossy path* across the *metro* network configuration.

We introduce latency and packet loss to the network using `netem`, a Linux package designed to perform wide area network emulation [24]. We construct the *slow path* by adding latency to the network in increments of 10ms, iterating from 0 to 70ms. We construct the *lossy path* by adding random packet loss to the network in increments of 1%, iterating from 0 to 10%. We perform replay using the video containing 2 marked regions, and collect fps processing data for each path configuration.

5.7.1 Impact of Packet Loss

Because the client and server maintain several TCP connections, adding packet loss on the ingress link causes the client to start sending retransmission packets. As the loss increases from 1 to 10%, the amount of retransmission traffic increases proportionally. This has the effect of adding latency to the application since the server will not be able to process the frame until it arrives completely. Figure 5.16 shows

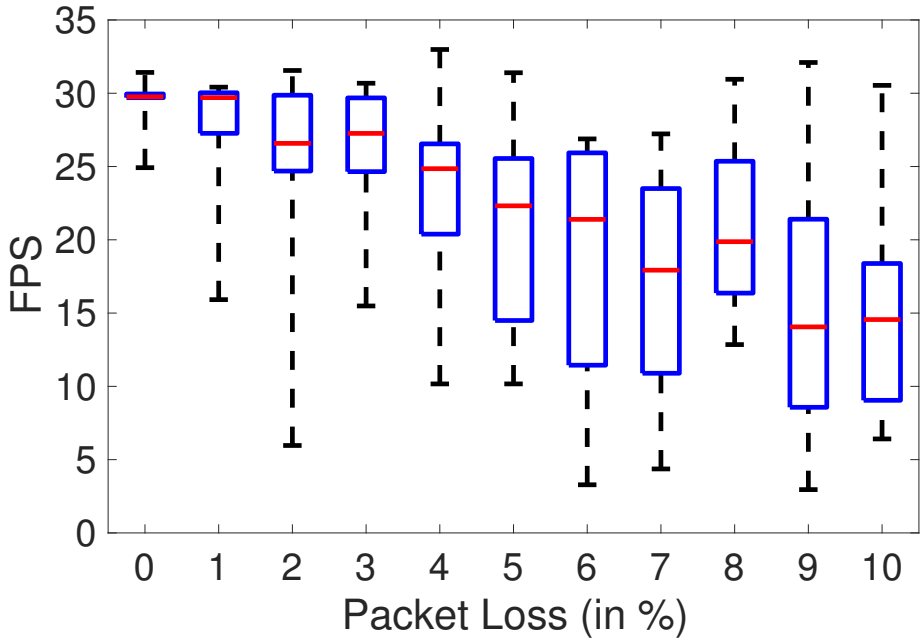


FIGURE 5.16: Impact of packet loss on the rate of frame delivery.

the performance results for ePrivateEye runninng across the *lossy path* in the *metro* network configuration. As can be observed, the impact of packet loss on the network can be observed almost instantaneously as the loss rate increases from 1 to 2%. This causes the median fps rate to drop from 30 fps to 26 fps. At 5% packet loss, the performance of ePrivateEye has dropped by 25% from 30 fps to 22.5 fps, and by 10% packet loss, the performance has dropped by 50% to $\approx$ 15 fps. The results indicate a nearly linear relationship between packet loss and number of frames delivered per second. Still, ePrivateEye is able to outperform the *on-device* configuration all the way up to 8% packet loss, and stays competitive through 10% packet loss.

In addition to the median fps decreasing, as the amount of packet loss increases, the variance between fps greatly increases. This has the visual effect of adding a large amount of judder to the experience of using the camera app. Since packet loss is not uniform, this creates periods of both low and high fps causing the camera app, at times, to appear laggy before returning to realtime.

5.7.2 Impact of Network Delay

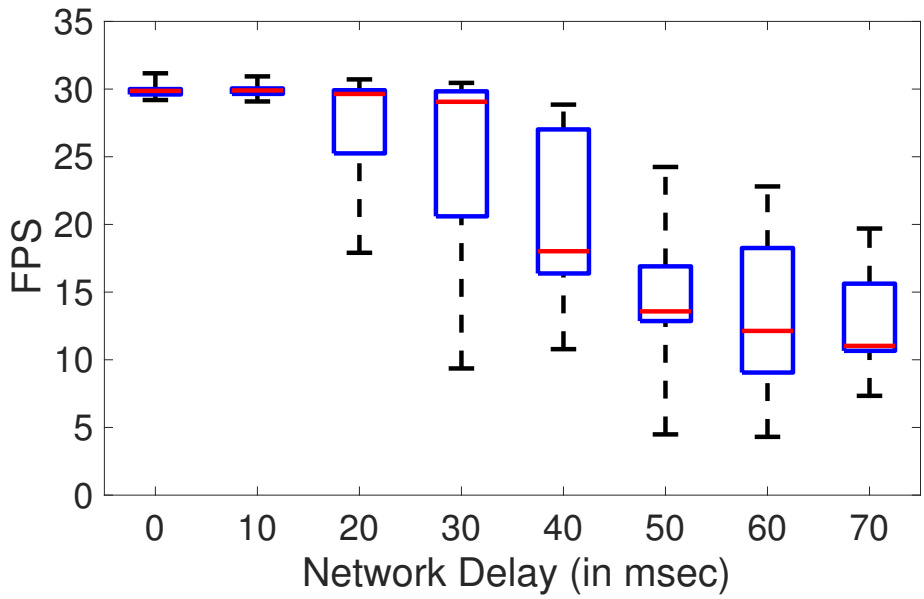


FIGURE 5.17: Impact of network delay on the rate of frame delivery.

Figure 5.17 shows the performance results for ePrivateEye runninng across the *slow path* in the *metro* network configuration. As can be observed, ePrivateEye maintains a median performance of 30 fps as the latency increases from 0 to 30 ms. When the latency increases from 30 to 40 ms of delay, the median performance drops by 42% to $\approx$ 17.5 fps. The reason for this drop is due to the processing rate of frames within the *camera service*. Because the *camera service* receives frames at a rate of 30 fps from the camera sensor, ePrivateEye has approximately 33 ms to perform the remote processing before the next frame is received. As long as the client and server are able to complete the data exchange within this time frame, the performance will not suffer. As soon as the latency eclipses this window, the performance of ePrivateEye becomes dictated by the latency of the network.

Although the performance starts to decay after this point, ePrivateEye is able to produce decent performance results. With 50 ms of delay, ePrivateEye still performs at a median rate of $\approx$ 15 fps, which is competitive with the *on-device* performance

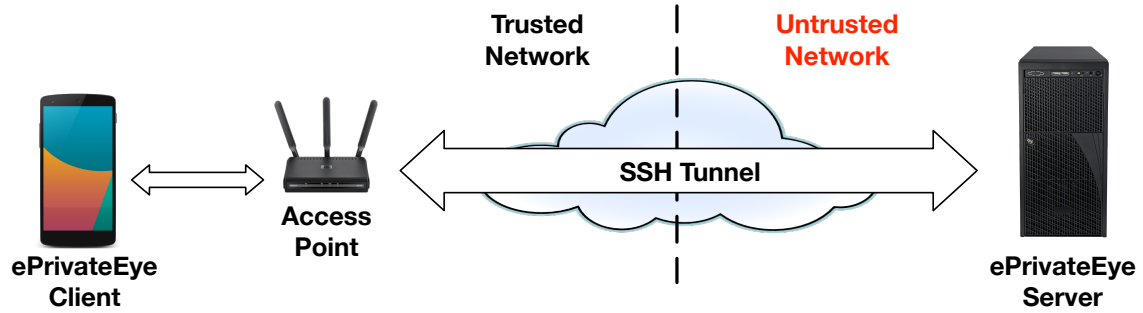


FIGURE 5.18: The secure channel setup to provide connectivity between the ePrivateEye client on Duke's trusted network and ePrivateEye server on an instance of AWS server.

as can be observed in Figure 5.15. This result show that even under networks with high latency ($\approx$ 30 ms), ePrivateEye is able to produce strong performance results.

5.8 ePrivateEye in the Cloud

PrivateEye is designed to protect sensitive information from being accidentally captured by the camera apps. ePrivateEye extends the original system to provide real-time performance without compromising the privacy guarantees. Therefore, it is of utmost importance that the edge computing resource used for running the ePrivateEye server is trusted. We envisioned a futuristic scenario, where a trusted cloud service provider aims to provide the benefits of ePrivateEye to the masses by running the server component on its secured cloud infrastructure. We are interested to see the rate of frame delivery in such a scenario.

For this, we used the AWS cloud infrastructure and configured the ePrivateEye server on an instance of AWS machine. Ideally, when the camera data leaves the smartdevice for processing on a cloud infrastructure, it should be encrypted. To emulate this scenario we setup Nexus 5 (client) to connect to an AP (within Duke University's network) and established a SSH Tunnel from the AP to the AWS server (Figure 5.18). We measured the rate at which the frames were delivered to the app

for a video with one marked region on a whiteboard under all the three camera motion setting.

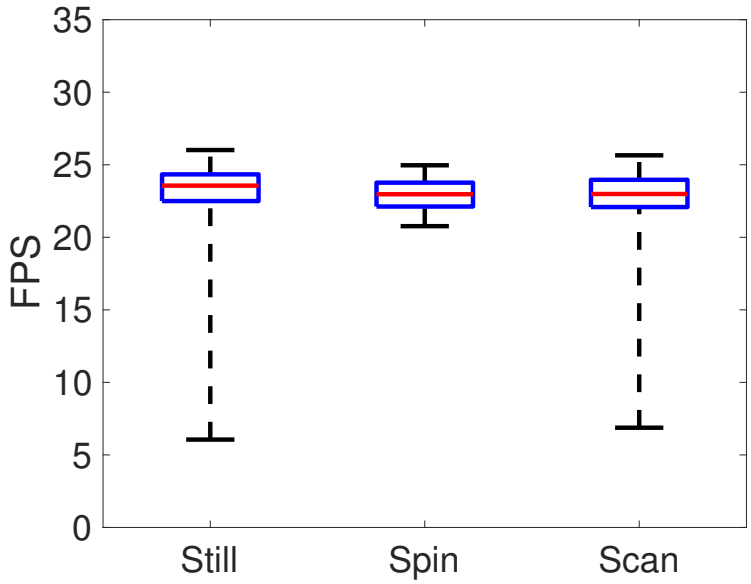


FIGURE 5.19: FPS achieved by configuring ePrivateEye server on an instance of AWS server.

Figure 5.19 shows that under the described setup the median rate at which frames are delivered is 23. This is due to the fact that SSH tunnel adds an additional computation burden of encryption and decryption. We noticed significant aberrations in processing delay during the *still* and *scan* recordings, showing how the additional network overhead can be unpredictable at times. Overall, this is a very encouraging result as it shows that ePrivateEye can give an acceptable performance under an additional layer of security.

6

Discussion

6.1 Impact of Offloading

The evaluation results of ePrivateEye show that by offloading to the edge, the performance of the original PrivateEye can be vastly improved. In this configuration, the system achieves a frame rate of 30 fps and a precision of 100%. At the same time, the CPU load decreases by 5% from running ePrivateEye *on-device*, while the memory consumption remains unchanged. The only performance limitations include a slight increase in power consumption of 8.3%. The primary improvement over PrivateEye can be observed by ePrivateEye's ability to handle scaling. ePrivateEye maintains a high level of performance while tracking multiple marked regions, whereas PrivateEye's performance degrades significantly when there's more than one region. With such a performance, ePrivateEye enables real-time camera apps to function smoothly while providing high privacy guarantees.

6.2 Managing Network Conditions

The network configuration *edge* is well suited for home network, while *building* and *metro* configurations can be adopted in an enterprise setting. By varying the net-

work configurations, we have explored the network conditions that might affect the performance of ePrivateEye. The results show that regardless of the offloading configuration, ePrivateEye will provide the performance guarantees to implement and scale the system. The lossy network evaluation shows ePrivateEye can sustain a packet loss up to 8% and a network delay up to 40 ms while delivering frames at the median rate of 20 per second. To ensure optimal performance under varying conditions, SDN offers a viable solution by providing greater control over path selection.

6.3 Deep Learning

While the *edge* configuration offers the best performance, the *metro* configuration offers the most promising results. The remote server within this configuration presents the possibility of running heavier computations that require GPUs to run e.g. deep learning. This environment is highly suitable for an edge-enabled deployment of DarNet. For processing image data, the size of the input on the GPU represented an insignificant load, and was easily handled. Advanced computer vision techniques coupled with deep learning can classify several classes of objects or shapes, but require this level of computing to operate at real-time speeds. Because the marker detection is completely run on the server, classifiers can be added or updated without changing anything on the client side.

7

Conclusion

7.1 Contributions

This work presents two systems – DarNet and ePrivateEye. DarNet presents a framework for combining multimodal data from IoT devices and processing the data on a resource constrained device. We designed an application using this framework to perform distracted driving classification on IMU data collected from a mobile device and video data collected from an in-vehicle camera. We showed how the application excels at the task of classifying driving behaviors, but is limited in performance due to the computational cost of performing the image classifications (CNN) on a mobile device.

To improve the performance of image processing applications on resource constrained devices, we implemented ePrivateEye, a framework for offloading computationally expensive operations to the edge. ePrivateEye is an edge-enabled version of PrivateEye which offloads an image processing pipeline to a high-performance server. ePrivateEye outperformed the existing system and achieves realtime performance (30 fps) while providing high security (precision 1) and acceptable app usability (recall

$\geqslant 0.86$) at a small cost of increased energy consumption (8%). In addition, we performed case studies to show that ePrivateEye can be adopted for large metropolitan areas as it can resist packet losses up to 4% and network delays up to 30 ms.

7.2 Future Directions

The next step of this work is to further test the usability and functionality of the edge for offloading applications. Because DarNet is designed to operate within a moving vehicle, offloading this system to the edge requires the additional use of "cloudlets" and VM migrations to maintain consistent performance within cities. Being able to utilize this system while navigating through metropolitan areas would be a significant engineering feat, drawing on multiple areas of existing research.

The next direction for ePrivateEye is to implement the image processing pipeline using deep learning. Two promising deep learning models, YOLO [61] and fR-CNN [62], allow for fast detection and tracking of multiple objects within a given frame. Porting the ePrivateEye computer vision pipeline to utilize these advancements in deep learning has the potential to allow for additional scaling, as well as incorporating additional patterns that can be recognized by the algorithms. Further, this system design would improve the ability for the image processing offloading framework to generalize across different computer vision algorithms.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., et al. (2015), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," Software available from tensorflow.org.

[2] Administration, N. H. T. S. (2014), "Distracted Driving," `http://www.distraction.gov/stats-research-laws/facts-and-statistics.html`.

[3] Arora, S., Bhaskara, A., Ge, R., and Ma, T. (2014), "Provable Bounds for Learning Some Deep Representations." in *ICML*, pp. 584–592.

[4] Automatic (2015), "Automatic," `https://www.automatic.com/`.

[5] Autoweek (2008), "Mercedes E-Class No Doze," `http://autoweek.com/article/car-news/no-doze-mercedes-e-class-alerts-drowsy-drivers`.

[6] Bahl, P. (2015), "The Emergence of Micro Datacenters (cloudlets) for Mobile Computing," .

[7] Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J. (2007), "Simplifying Cyber Foraging for Mobile Devices," in *Proc. of MobiSys*.

[8] Bo, C., Jian, X., Li, X.-Y., Mao, X., Wang, Y., and Li, F. (2013), "You'Re Driving and Texting: Detecting Drivers Using Personal Smart Phones by Leveraging Inertial Sensors," in *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking*.

[9] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012), "Fog Computing and Its Role in the Internet of Things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pp. 13–16, New York, NY, USA, ACM.

[10] Bouguet, J.-Y. (2001), "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm," *Intel Corporation*, 5, 4.

[11] Canny, J. (1986), "A Computational Approach to Edge Detection," *TMAPI*.

[12] Carrie Kim (2015), "Volvo," `https://www.cars.com/articles/volvo-car-seat-concept-could-cut-distracted-driving-by-parents-1420682901604`.

[13] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011), "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, pp. 301–314, ACM.

[14] Cobârzan, C., Hudelist, M. A., Schoeffmann, K., and Primus, M. J. (2015), "Mobile Image Analysis: Android vs. iOS," in *International Conference on Multimedia Modeling*, pp. 99–110, Springer.

[15] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010), "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62, ACM.

[16] Cuervo, E., Wolman, A., Cox, L. P., Lebeck, K., Razeen, A., Saroiu, S., and Musuvathi, M. (2015), "Kahawai: High-quality mobile gaming using gpu offload," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 121–135, ACM.

[17] Culp, J., Gindy, M. E., and Haque, A. (2008), "Driver Alertness Monitoring Techniques: A Literature Review," *International Journal of Heavy Vehicle Systems*, 15.

[18] David Strom (2012), "Layers of Latency: Cloud Complexity and Performance," `https://www.wired.com/insights/2012/09/layers-of-latency/`.

[19] Deng, L. and Platt, J. C. (2014), "Ensemble deep learning for speech recognition." in *INTERSPEECH*, pp. 1915–1919.

[20] Doug Newcomb (2012), "Bosch Drowsy Driver Alert System," `http://www.dougnewcomb.com/2012/05/07/bosch-debuts-low-cost-drowsy-driver-alert-system`.

[21] Federal Highway Administration (2008a), "Official US Government Website for Distracted Driving," `https://www.fhwa.dot.gov/goshrp2/About`.

[22] Federal Highway Administration (2008b), "SHRP2 Solutions," `https://www.fhwa.dot.gov/goshrp2/About`.

[23] Fitch, G. M., Soccolich, S. A., Guo, F., McClafferty, J., Fang, Y., et al. (2013), "The impact of hand-held and hands-free cell phone use on driving performance and safety-critical event risk," Tech. rep., National Highway Traffic Safety Administration.

[24] Foundation, T. L. (2017), "Netem," [Online; accessed 22-Apr-2017].

[25] Friedman, N., Geiger, D., and Goldszmidt, M. (1997), "Bayesian Network Classifiers," *Mach. Learn.*, 29, 131–163.

[26] Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., and Riviere, E. (2015), "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Review*, 45, 37–42.

[27] Gember, A., Dragga, C., and Akella, A. (2012), "ECOS: leveraging software-defined networks to support mobile application offloading," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, pp. 199–210, ACM.

[28] Georgiev, P., Lane, N. D., Rachuri, K. K., and Mascolo, C. (2016), "LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM*, pp. 320–333, ACM.

[29] Google (2015), "Inception-V3 Source Code," `https://github.com/tensorflow/models/tree/master/inception"`.

[30] Google (2017), "MobileNets: Open-Source Models for Efficient On-Device Vision," `https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html`.

[31] Google (2018), "Optimizing for mobile," `https://www.tensorflow.org/mobile/optimizing`.

[32] Gordon, M. S., Hong, D. K., Chen, P. M., Flinn, J., Mahlke, S., and Mao, Z. M. (2015), "Accelerating Mobile Applications Through Flip-Flop Replication," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pp. 137–150, New York, NY, USA, ACM.

[33] Graves, A., Mohamed, A., and Hinton, G. E. (2013a), "Speech Recognition with Deep Recurrent Neural Networks," *CoRR*, abs/1303.5778.

[34] Graves, A., Mohamed, A.-r., and Hinton, G. (2013b), "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, IEEE.

[35] Hauswald, J., Manville, T., Zheng, Q., Dreslinski, R., Chakrabarti, C., and Mudge, T. (2014), "A hybrid approach to offloading mobile image classification," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 8375–8379, IEEE.

[36] Hermans, M. and Schrauwen, B. (2013), "Training and analysing deep recurrent neural networks," in *Advances in neural information processing systems*, pp. 190–198.

[37] Hochreiter, S. and Schmidhuber, J. (1997), "Long Short-Term Memory," *Neural Comput.*, 9.

[38] Inc., Q. T. (2015), "Trepn Power Profiler," `https://developer.qualcomm.com/software/trepn-power-profiler`, [Online; accessed 7-Dec-2015].

[39] Jain, V. and Seung, S. (2009), "Natural image denoising with convolutional networks," in *Advances in Neural Information Processing Systems*, pp. 769–776.

[40] Kim, M., Lim, J., Yu, H., Kim, K., Kim, Y., and Lee, S.-B. (2017), "ViewMap: Sharing Private In-Vehicle Dashcam Videos," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA.

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

[42] Lane, N., Bhattacharya, S., Mathur, A., Forlivesi, C., and Kawsar, F. (2016a), "Dxtk: Enabling resource-efficient deep learning on mobile and embedded devices with the deepx toolkit," in *Proceedings of the 8th EAI International Conference on Mobile Computing, Applications and Services, ser. MobiCASE*, vol. 16, pp. 98–107.

[43] Lane, N. D., Bhattacharya, S., Georgiev, P., Forlivesi, C., and Kawsar, F. (2015), "An Early Resource Characterization of Deep Learning on Wearables, Smartphones and Internet-of-Things Devices," in *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App '15, pp. 7–12, New York, NY, USA, ACM.

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

[45] Latifi Oskouei, S. S., Golestani, H., Hashemi, M., and Ghiasi, S. (2016), "Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android," in *Proceedings of the 2016 ACM on Multimedia Conference*, pp. 1201–1205, ACM.

[46] Lee, J., Brown, T., Fiorentino, D., Fell, J., Traube, E., Nadler, E., et al. (2011), "Using Vehicle-Based Sensors of Driver Behavior to Detect Alcohol Impairment," in *22nd International Technical Conference on the Enhanced Safety of Vehicles (ESV)*.

[47] Lee, K., Chu, D., Cuervo, E., Kopf, J., Wolman, A., Degtyarev, Y., Grizan, S., and Flinn, J. (2015), "Outatime: Using Speculation to Enable Low-Latency

Continuous Interaction for Mobile Cloud Gaming," *GetMobile: Mobile Comp. and Comm.*, 19, 14–17.

[48] Liu, L., Karatas, C., Li, H., Tan, S., Gruteser, M., Yang, J., Chen, Y., and Martin, R. P. (2015a), "Toward Detection of Unsafe Driving with Wearables," in *Proceedings of the 2015 Workshop on Wearable Systems and Applications*.

[49] Liu, T., Yang, Y., Huang, G.-B., and Lin, Z. (2015b), "Detection of Drivers Distraction Using Semi-Supervised Extreme Learning Machine," in *Proceedings of ELM-2014 Volume 2*, pp. 379–387, Springer.

[50] Mazda (2012), "LDWS," `http://www.mazda.com/en/innovation/technology/safety/active_safety/ldws/"`.

[51] National Highway Traffic Safety Administration (2016), "Research Note: Distracted Driving 2014," `https://crashstats.nhtsa.dot.gov/api/public/viewpublication/812260`.

[52] of Transportation, U. D. (2012), "An Evaluation of Emerging Driver Fatigue Detection Measures and Technologies," Tech. rep., Technical Report, US Department of Transportation.

[53] OpenCV (2018), "About OpenCV," `https://opencv.org/about.html`.

[54] Pascanu, R., Gülçehre, Ç., Cho, K., and Bengio, Y. (2013), "How to Construct Deep Recurrent Neural Networks," *CoRR*, abs/1312.6026.

[55] Progressive (2014), "Progressive snapshot," `https://www.progressive.com/auto/snapshot/`.

[56] Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govindan, R. (2011), "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp. 43–56, ACM.

[57] Radu, V., Lane, N. D., Bhattacharya, S., Mascolo, C., Marina, M. K., and Kawsar, F. (2016), "Towards Multimodal Deep Learning for Activity Recognition on Mobile Devices," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, pp. 185–188, New York, NY, USA, ACM.

[58] Rallapalli, S., Qiu, H., Bency, A., Karthikeyan, S., Govindan, R., Manjunath, B., and Urgaonkar, R. (2016), "Are very deep neural networks feasible on mobile devices," *IEEE Trans. Circ. Syst. Video Technol.*

[59] Raval, N., Srivastava, A., Lebeck, K., Cox, L., and Machanavajjhala, A. (2014), "Markit: Privacy markers for protecting visual secrets," in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pp. 1289–1295, ACM.

[60] Raval, N., Srivastava, A., Razeen, A., Lebeck, K., Machanavajjhala, A., and Cox, L. P. (2016), "What you mark is what apps see," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 249–261, ACM.

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

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

[63] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. (2009a), "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, 8, 14–23.

[64] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. (2009b), "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, 8.

[65] Shi, C., Pandurangan, P., Ni, K., Yang, J., Ammar, M., Naik, M., and Zegura, E. (2013), "IC-Cloud: Computation offloading to an intermittently-connected cloud," Tech. rep., Georgia Institute of Technology.

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

[67] Singla, A., Chandrasekaran, B., Godfrey, P., and Maggs, B. (2014), "The internet at the speed of light," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, p. 1, ACM.

[68] Streiffer, C., Raghavendra, R., Benson, T., and Srivatsa, M. (2017a), "DarNet: A Deep Learning Solution for Distracted Driving Detection," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, ACM.

[69] Streiffer, C., Srivastava, A., Orlikowski, V., Velasco, Y., Martin, V., Raval, N., Machanavajjhala, A., and Cox, L. P. (2017b), "ePrivateeye: to the edge and beyond!" in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, p. 18, ACM.

[70] Suzuki, S. et al. (1985), "Topological structural analysis of digitized binary images by border following," *Computer vision, graphics, and image processing*, 30, 32–46.

[71] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015a), "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9.

[72] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015b), "Rethinking the inception architecture for computer vision," *arXiv preprint arXiv:1512.00567*.

[73] Virginia Tech (1996), "Virginia Tech Transportation Institute," `http://www.vtti.vt.edu/`.

[74] Wang, Y., Yang, J., Liu, H., Chen, Y., Gruteser, M., and Martin, R. P. (2013), "Sensing Vehicle Dynamics for Determining Driver Phone Use," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13.

[75] Waze (2016), "Waze," `http://www.waze.com`.

[76] Wu, J., Hu, Q., Leng, C., and Cheng, J. (2016), "Shoot to Know What: An Application of Deep Networks on Mobile Devices." in *AAAI*, pp. 4399–4400.

[77] Xie, J., Xu, L., and Chen, E. (2012), "Image denoising and inpainting with deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 341–349.

[78] Xuehan-Xiong and De la Torre, F. (2013), "Supervised Descent Method and its Application to Face Alignment," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[79] Yang, J., Sidhom, S., Chandrasekaran, G., Vu, T., Liu, H., Cecan, N., Chen, Y., Gruteser, M., and Martin, R. P. (2011), "Detecting Driver Phone Use Leveraging Car Speakers," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*.

[80] You, C.-W., Lane, N. D., Chen, F., Wang, R., Chen, Z., Bao, T. J., Montes-de Oca, M., Cheng, Y., Lin, M., Torresani, L., and Campbell, A. T. (2013), "CarSafe App: Alerting Drowsy and Distracted Drivers Using Dual Cameras on Smartphones," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*.

[81] Yue-Hei Ng, J., Hausknecht, M., Vijayanarasimhan, S., Vinyals, O., Monga, R., and Toderici, G. (2015), "Beyond short snippets: Deep networks for video classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4694–4702.