University of Dublin

TRINITY COLLEGE

Wireless Augmented Reality

Bringing deep learning to mobile augmented reality systems through computational offloading to the cloud

Author:
Darragh McCaffrey

Supervisor:
Prof. Douglas Leith

A dissertation presented to the University of Dublin, Trinity College in fulfilment of the requirements for the degree of
Master in Computer Science
at the
School of Computer Science & Statistics

Submitted to the University of Dublin, Trinity College, May 2018

DECLARATION

I, Darragh McCaffrey declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

________________________________ ______________

Signed Date

Summary

There is currently a huge amount of interest in bringing smart augmented reality applications to mobile devices. Mobile devices however, have limited resources and this means they cannot run certain computationally expensive computer vision techniques in a sufficiently fast manner. Taking the use case of object detection, the current state of the art object detection models utilise deep learning techniques to make accurate predictions. Deep learning incurs high processing and memory loads which results in long response time and a large amount of battery drain in mobile devices with limited resources. One solution to this problem is to offload these deep learning computations to the cloud. Offloading to a more powerful backend server provides the mobile application with increased processing power and reduces battery consumption but is subject to network impairments.

In this dissertation, the use of a reduced deep-learning model run locally on a mobile device is compared to offloading the computation to a much larger, more accurate deep learning model for object detection on the cloud. The impact of varying networking conditions on the performance of computational offloading to the cloud are also investigated and any impairments due to poor networking conditions or due to limited resources on the mobile device are masked through local processing.

This is completed through the development of a mobile cloud computing framework which supports mobile object detection through deep learning. The framework was developed as an Android application while the backend server is modelled as a RESTful API. The mobile application chooses between running deep learning computations locally with a reduced deep learning model and offloading these computations to the RESTful API where detection calculations are performed and results returned. Impairments due to the limited resources of the mobile device or due to poor networking conditions are masked through the use of tracking and an offloading decision model which decides whether it is more favourable to perform the detection locally or offload to the cloud.

The evaluation of this framework shows a large increase in frame rate and accuracy and a decrease in energy consumption when offloading. It was also shown that it is vitally important for the application to be contextually aware when making decisions on whether to offload or not. It was found that there are trade-offs between energy consumption, accuracy and speed when making this decision. Parameters such as bandwidth, network latency, image size and deep learning model size were shown to have a large impact on this decision of whether it is more favourable to perform detections locally or offload to the cloud.

Abstract

There is currently a huge amount of interest in bringing smart augmented reality applications to mobile devices. Mobile devices however, have limited resources and this means they cannot run certain computationally expensive computer vision techniques in a sufficiently fast manner. Taking the use case of object detection, the current state of the art object detection models utilise deep learning techniques to make accurate predictions. Deep learning incurs high processing and memory loads which results in long response time and a large amount of battery drain in mobile devices with limited resources. One solution to this problem is to offload these deep learning computations to the cloud. Offloading to a more powerful backend server provides the mobile application with increased processing power and reduces battery consumption but is subject to network impairments.

In this dissertation, the use of a reduced deep-learning model run locally on a mobile device is compared to offloading the computation to a much larger, more accurate deep learning model for object detection on the cloud. The impact of varying networking conditions on the performance of computational offloading to the cloud are also investigated and impairments are masked through local processing. This is completed through the development of a mobile cloud computing framework.

The evaluation of this framework shows a large increase in frame rate and accuracy and decrease in energy consumption when offloading. It was also shown that it is vitally important for the application to be contextually aware when making decisions on whether to offload or not. It was found that there are trade-offs between energy consumption, accuracy and speed when making this decision. Parameters such as bandwidth, network latency, image size and deep learning model size were shown to have a large impact on this decision of whether it is more favourable to perform detections locally or offload to the cloud.

Acknowledgements

I would like to extend my gratitude to my supervisor Douglas Leith for all of the guidance and help he has provided to me throughout the course of this dissertation.

I would like to thank my family for all the support they gave throughout my time in college and during the time I spent working on this dissertation.

Finally, I would like to thank my friends and classmates for all of the help they have given me throughout my years spent in college.

Table of Contents

Summary ii

Abstract iv

Acknowledgements v

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Motivation 1

1.2 Research Objectives 2

1.3 Dissertation Overview 3

2 Background 4

2.1 Augmented Reality 4

2.2 Mobile Cloud Computing 6

2.2.1 Architecture 7

2.2.2 Benefits 8

2.2.3 Issues 9

2.2.4 Offloading Decision Making 10

2.3 Deep Learning & Object Detection 11

2.3.1 Object Detection 11

2.3.2 Convolutional Neural Networks 12

3 State of The Art 16

3.1 Object Detection 16

3.1.1 YOLO 16

3.2 Tracking 19

3.2.1 FAST 19

3.2.2 Feature Based Optical Flow 19

3.3 Mobile Cloud Computing 20

3.3.1 Glimpse 20

3.3.2 A Computational Offloading Framework for Object Detection in Mobile Devices..........21
3.3.3 Delivering Deep Learning to Mobile Devices via Offloading..........22

4 Design.......... 23

4.1 Problem Setup.......... 23

4.2 Mobile Application.......... 24

4.2.1 Tracking Decision..........24
4.2.2 Offloading Decision..........27
4.2.3 Object Detection..........28
4.2.4 Tracking..........29
4.2.5 Offloading..........30

4.3 Backend Server.......... 32

5 Implementation.......... 33

5.1 Android Application.......... 33

5.1.1 Camera & Settings..........33
5.1.2 Tracking..........34
5.1.3 Tracking Decision..........35
5.1.4 Offloading Decision..........36
5.1.5 Object Detection..........37
5.1.5.1 Tensorflow..........37
5.1.5.2 Implementation..........38
5.1.6 Offloading..........39
5.1.7 Results..........40

5.2 Backend Server.......... 41

6 Evaluation.......... 43

6.1 Evaluation Metrics.......... 43

6.1.1 Latency..........43
6.1.2 Energy Consumption..........44
6.1.3 Accuracy..........44

6.2 Experimental Setup.......... 45

6.3 Results.......... 46

6.3.1 Local Processing..........46

6.3.2 Offloading 48
6.3.3 Proposed Framework 52
6.3.3.1 Tracking 52
6.3.3.2 Tracking Decision 53
6.3.3.3 Offloading Decision 54
6.3.4 Discussion 54
7 Conclusions 56
7.1 Summary 56
7.2 Future Work 57
References 59

List of Figures

Figure 1 - Reality-Virtuality Continuum [7]....4
Figure 2 - Design of the first head-mounted display device [8]. 5
Figure 3 - Flowchart of a simple AR system [13]. 5
Figure 4 - Mobile AR game, AR Defender which uses marker images for tracking [13]....6
Figure 5 - Mobile cloud computing architecture. 8
Figure 6 - Parameters affecting offloading decision [25]. 10
Figure 7 – A simple three layered neural network, comprised of an input layer, a hidden layer and an output layer [31]. 13
Figure 8 - Architecture of a convolutional neural network....14
Figure 9 - Representation of max pooling calculation....15
Figure 10 - The YOLO model (different colours in class probability map refer to resulting class predicted by the CNN) [34]....17
Figure 11 - The YOLO detection network [34]. 17
Figure 12 - Detection of the optical flow in 3 temporally-consecutive images, showing the movement of an example shape." http://tcr.amegroups.com/article/view/3200/html 20
Figure 13 - Architecture of proposed framework. 23
Figure 14 - Chart plotting the effect of frame resolution on latency. 26
Figure 15 - Cropping scene based on amount of change in parts of the frame....27
Figure 16 - Stale frame due to latency of object detection. 29
Figure 17 - Processing flow of proposed application. 31
Figure 18 - Overview of RESTful API. 32
Figure 19 - Main menu of android application. 34
Figure 20 - Settings menu of android application....34
Figure 21 - Tracked feature points being drawn on screen in debug mode of android application as small multi-coloured rectangles....35
Figure 22 - Pseudocode of object detection flow in mobile application....36
Figure 23 - Input tensor format [50]. 38
Figure 24 - Diagram of object detection flow using Tensorflow [50]. 39
Figure 25 - Screenshot from mobile application displaying detection results....40

Figure 26 - Example of object detection endpoint response. 42
Figure 27 - The change in latency caused by changes in the frame resolution in local processing. 46
Figure 28 - The effect of frame resolution on battery on battery consumption in local processing. 47
Figure 29 - The latency of the object detection calculation due to changes in the frame resolution. Offloading is set at 5Mbps and optimal conditions refers to an unrestricted connection with a private network. 48
Figure 30 - The inference time of object detection using the more powerful version of YOLO on the backend server. 49
Figure 31 - The effect that changes in frame resolution have on the amount of battery consumed by the application after 30 minutes. 49
Figure 32 - The effect of various bandwidth speeds on the latency of object detection in the mobile application. 50
Figure 33 - The effect of network latency on the frame rate of the application. 51
Figure 34 - The accuracy of object detection run locally on the mobile device with varying amounts of delay. The accuracy of using just the object detection model is plotted against using both the object detection model and tracking. 52

List of Tables

Table 1 - A comparison of detection frameworks on Pascal VOC 2007+2012. The various versions of YOLO are based on the input image size, the actual model is the same for each [35]..18
Table 2 - Description of RESTful API endpoints. 41
Table 3 - The latency of various features in the proposed framework. 53

1 Introduction

1.1 Motivation

There is currently a huge amount of interest in bringing smart augmented reality (AR) applications to mobile devices. AR is very much in its infancy in terms of being a widely adopted technology, however it provides many great benefits and possibilities that could make improvements to every-day life. There is an abundance of research into augmented reality but it has only garnered mainstream attention in recent years due to a number of popular mobile applications such as Pokémon Go [1] and the Ikea furniture app [2] which allows users to place furniture in their home before buying.

One of the main reasons as to why augmented reality has not yet become a wide spread technology on every mobile device is that mobile devices have limited resources. This means they cannot run certain computationally expensive computer vision techniques in a sufficiently fast way. In this research, the focus is on the use case of bringing object detection through deep learning to mobile augmented reality applications. There are many uses for deep learning in augmented reality applications, a simple one being tagging people or objects in a scene and overlaying information about that person or object on the screen. Facial recognition can be utilised for authentication systems [3] using deep learning techniques. Object detection can also be used to assist visually impaired people in navigation [4].

Any lag in a mobile application can significantly affect user experience in a negative way and for use cases such as guiding the visually impaired it is vital that they receive real time results. Deep learning incurs high processing and memory loads which result in long response time and a large amount of battery drain in mobile devices with limited resources. One solution to this problem is to offload these deep learning computations to the cloud. There has been much research done into the benefits of offloading computationally expensive processes from mobile devices to the cloud [5]. Mobile cloud computing provides the mobile device with unlimited data storage and processing power and removes the obstacles that implanting deep learning

locally on the mobile device provide. However, there are disadvantages to computational offloading to the cloud such as latency due to network impairments.

In this dissertation, the use of a reduced deep-learning model run locally on a mobile device is compared to offloading the computation to a much larger, more accurate deep learning model for object detection on the cloud. The impact of varying networking conditions on the performance of computational offloading to the cloud is also investigated and impairments are masked through local processing. This is completed through the development of a mobile cloud computing framework to support mobile object detection through deep learning.

1.2 Research Objectives

The main research objective for this dissertation is to develop a framework for investigating deep learning through computational offloading to the cloud. A mobile application is to be developed in which object detection is implemented, and results overlaid on the screen, through both the use of local processing and through offloading input to a more powerful backend server. This application must provide the user with the ability to switch between these two use cases for the purposes of testing. Both of these methods are to be experimented with and performance metrics gathered such as accuracy, latency and energy consumption.

Another objective is to investigate the various state of the art object detection systems and to examine the accuracy/speed trade-off in order to choose the appropriate system for creating a real-time application. As speed is vitally important in this real-time application, the fastest, while still maintaining decent accuracy is to be chosen.

Research into computational offloading and also preliminary testing of the impairments caused by varying networking conditions is to be completed in order to create a simple decision-making model, which decides whether to offload to the cloud or not based on gathered metrics such as the energy consumed by the application and the current network bandwidth and latency. Methods that reduce the effects of latency in the system due to the constrained resources in the

mobile device or due to network impairments when offloading are also to be investigated and implemented.

Should these objectives be achieved, an extended objective is to use the information obtained from the results gathered from experimenting with this framework to create an intelligent offloading decision model. This decision model should take into account energy consumption, accuracy and speed to make a decision on whether it is more favourable to complete the deep learning computations locally or to offload them to the cloud.

1.3 Dissertation Overview

The structure of this dissertation is as follows, section 2 provides background information on the topics discussed in this dissertation. Section 3 provides an overview of the state of the art in object detection, tracking and also related works in mobile cloud computing. Section 3 gives a detailed description of the design of the proposed framework and explains the reasoning behind these design decisions. Section 4 contains a detailed description of how this design was implemented and explains the reasoning behind technologies used. Section 5 presents the testing that was completed and gives an interpretation of results. In section 6 conclusions are made and future works are recommended.

2 Background

This chapter gives a brief overview of the background of the topics covered in this dissertation, providing any essential information needed to understand the research decisions made. These topics include: augmented reality, mobile cloud computing, deep learning and object detection.

2.1 Augmented Reality

Augmented reality (AR) is described in [6] as being the "real-time mixing of computer generated content with live-video display". AR was developed from research into virtual reality and involves an interaction with a virtual world while also having a degree of interdependence with the real world. While virtual reality fully immerses the user into a virtual/computer generated world, augmented reality combines the real world with a virtual world by overlaying virtual inforamtion on top of the existing natural environment. This allows users an improved natural world where virtual information can assist them in their everyday life. AR is defined in [7] as being a subset of mixed reality in that it exists somewhere between a fully virtual environment and the real environment, this concept is demonstrated in Figure 1.

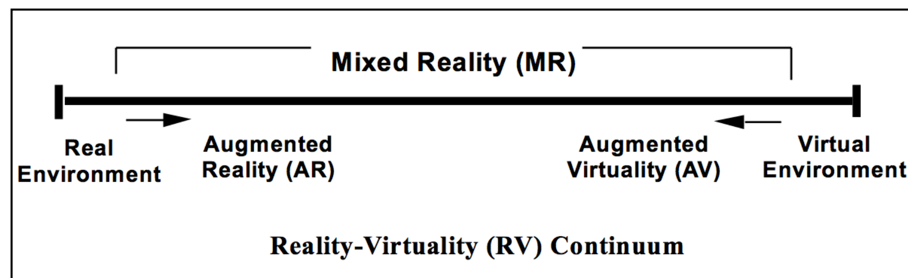

Figure 1 - Reality-Virtuality Continuum [7].

The concept of augmented reality originated in 1968 when Ivan Sutherland invented the first head-mounted display system [8]. This system used computer generated graphics to show users simple wireframe drawings. The design of this system is shown in Figure 2. Following this invention, it took 45 years before augmented reality through a head-mounted display was

brought to public consciousness through the release of Google Glass in 2013 [9]. More recently Microsoft released the Hololens [10], which is a much more powerful device than Google Glass.

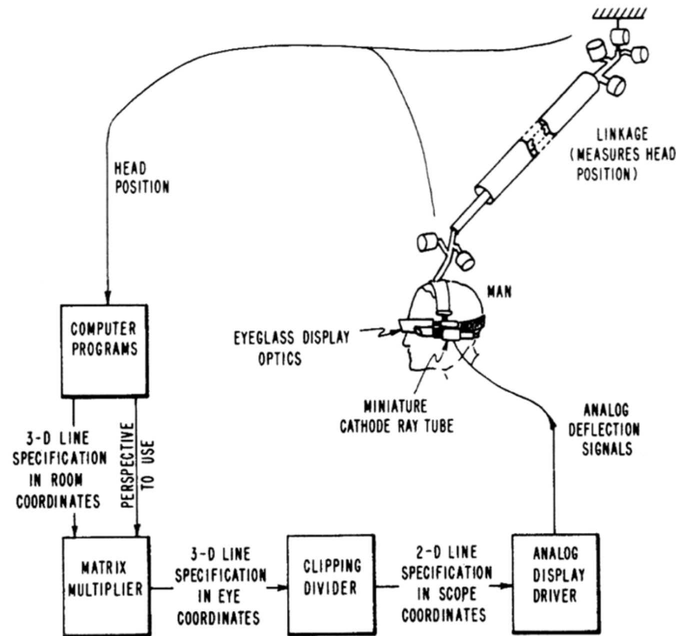

Figure 2 - Design of the first head-mounted display device [8].

The term augmented reality was first coined in 1990 in a study by two researchers for Boeing [11]. Augmented reality was only considered in a research setting until the release of ARToolkit in 1999 [12] which is an open source SDK for developing AR applications. The release of ARToolkit along with the increased processing power and better camera quality in mobile devices led to a huge amount of AR applications to be released to the public in previous years.

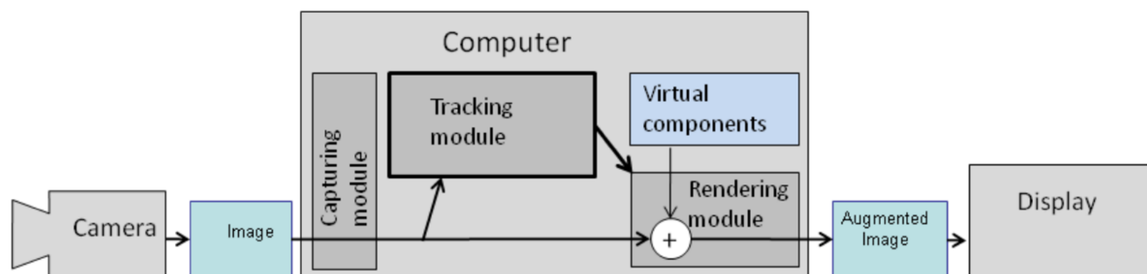

Figure 3 - Flowchart of a simple AR system [13].

A simple augmented reality system as demonstrated in [13], Figure 3, consists of a camera, a CPU and a display. This simple augmented reality system augments computer generated objects on top of a marker image. The system takes an image of the environment and detects a marker image using a tracking module which calculates the location and orientation of the camera based on this image. A rendering module uses the location and orientation of the camera in relation to the marker image to then render a virtual object on top of this image and display the results on the screen. This is the most common type of augmented reality application and is widely used in entertainment and gaming, such as the game AR Defender, shown in Figure 4. Tracking a marker image is not that computationally expensive and can be run on most smartphones. However, this use case is constrained in what it can achieve and adding deep learning to this tracking system opens a wide range of possibilities for new innovative augmented reality applications. Using object detection, the AR system knows both the location and the type of an object in the scene and can render virtual objects or information based on this.

Figure 4 - Mobile AR game, AR Defender which uses marker images for tracking [13].

2.2 Mobile Cloud Computing

Cloud computing has grown increasingly popular in recent years and infrastructure as a service (IaaS) such as Amazon EC2 [14] , platform as a service (PaaS) such as Heroku [15] and software as a service (SaaS) such as Salesforce [16] have become standard in enterprise computing. Cloud computing provides computing, storage and applications over the internet. The cloud refers to infrastructures in datacentres that support the business's needs, generally in

the form of data storage and remotely hosted applications. These infrastructures greatly benefit companies in that they reduce costs by eliminating the need for physical hardware, which allows companies to outsource data and computations on demand [17].

Applications on mobile devices are becoming increasingly complex and they have a higher demand on resources. Advances in smartphone hardware and battery life has not kept up with the advances and the higher demands of applications over the years. Mobile devices are constrained by low processing power, limited memory and limited battery [18]. Therefore many applications and processes such as deep learning are still unsuitable for mobile devices.

Computation offloading is a technique utilized in mobile applications in which expensive computations are transferred from the mobile device to a more powerful cloud or server. Computational offloading to the cloud can reduce energy consumption, increase speed and allow a mobile device to run certain applications that are unable to run due to insufficient resources. Mobile cloud computing is defined in [19] as “an integration of cloud computing technology with mobile devices to make the mobile devices resource-full in terms of computational power, memory, storage, energy, and context awareness”.

2.2.1 Architecture

The general architecture of mobile cloud computing is described in [5] and is shown in Figure 5. Mobile devices are connected to cloud services either through a mobile network or via Wi-Fi using access points. When connected through a mobile network, the devices connect to the cloud through a base station or a satellite link. When connecting via an access point, the mobile device connects through Wi-Fi which is connected to the internet service provider which provides internet connectivity and access to cloud services to the device. Wi-Fi connections provide low latency and consume less energy when compared to a connection through a 3G mobile network connection [20]. Therefore when utilizing mobile cloud computing a Wi-Fi connection is preferred [19].

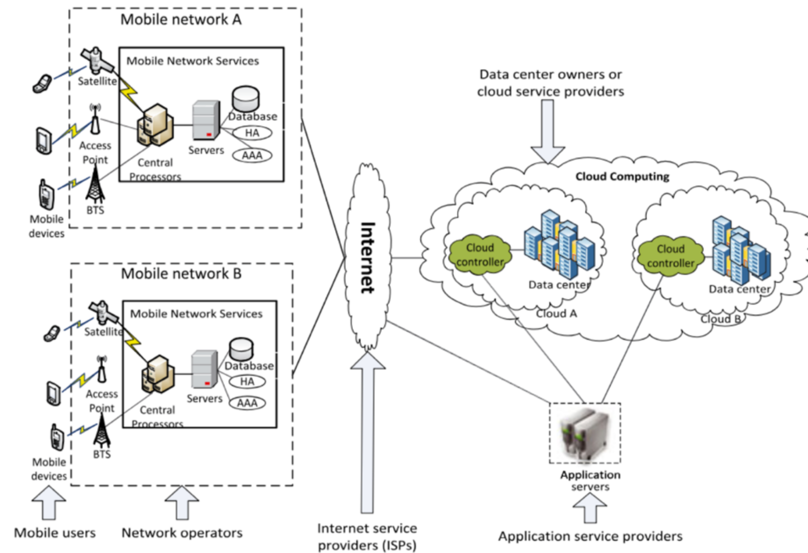

Figure 5 - Mobile cloud computing architecture.

2.2.2 Benefits

The many advantages of adding mobile cloud computing to mobile applications are listed in [5]:

- The first advantage is the extension of battery lifetime. This is a huge concern for mobile devices. Using computation offloading, the large computations which take a long execution time result in a large amount of power consumption. [21] produced research which evaluates offloading techniques through several experiments. They demonstrated that computation offloading can save energy significantly and showed a 45% decrease in energy consumption when offloading large matrix calculations.
- Another benefit of mobile cloud computing is the improvement of data storage capacity. Storage capacity is another constraint for mobile devices. Mobile cloud computing allows users to store large amounts of data in the cloud through wireless networks. Cloud data storage provides cost savings, recovery for lost files and accessibility i.e. files can be accessed from anywhere with an internet connection.
- Mobile cloud computing gives mobile devices access to unlimited processing power. Computationally expensive processes may add a huge amount of latency to an

application while also draining battery power. [20] showed a large increase in speed when they used offloaded calculations for making optimal moves in a chess game.

2.2.3 Issues

There are many challenges that we are faced with when implementing mobile cloud computing applications. Below three issues are listed that can affect a MCC system:

- Network impairments can cause many issues with mobile cloud computing systems. Service availability is a large issue on mobile devices and the mobile user may not be able to connect to cloud services due to traffic congestion, network failures or due to a lack of signal. A low bandwidth connection is much more likely in mobile networks as the radio resource is scarce compared to wired networks [5]. This can severely slow the speed of requests resulting in added latency to the mobile application. [22] shows the effects of bandwidth on speed when offloading deep learning computations to the cloud. In the case of dynamic network environments, it is important for the mobile application to keep track of changes in service availability and bandwidth.
- Another issue is the cost of offloading, offloading may not always be the most effective way of saving energy. Sometimes the actual cost of offloading can exceed the cost of the computation being offloaded. [23] provide a solution in which a program is partitioned based on an estimation of energy consumption. This partitioning is based on the trade-off between the offloading and computation costs. The cost of offloading decision-making models also needs to be considered as this can add an additional strain on resources.
- Security is a large concern in any form of cloud computing. [24] provides an extensive review of the challenges caused by security issues in mobile cloud computing. These security concerns include secure communication, data integrity, privacy, data availability, data access control, mobile application security and the distribution of data over a distributed infrastructure.

2.2.4 Offloading Decision Making

A mobile cloud application goes through a number of steps before offloading to the cloud. In general the network connectivity is first checked, noting available resources in the process. Then whether offloading is favourable or not is decided upon depending on the users objective which could include goals such as saving energy or increasing speed. If offloading is the favourable option then the computation is offloaded to the cloud and the application waits for results. If unfavourable, the application will perform the computation locally [19].

In a dynamic environment, context awareness is vitally important when implementing a mobile cloud computing architecture. This refers to the mobile devices understanding of the parameters that affects the decision in computation offloading. If parameters such as the network speed or the available resources on the device change, it is vitally important that the device can automatically adapt when making a decision. The parameters which affect the offloading decision are shown in Figure 6.

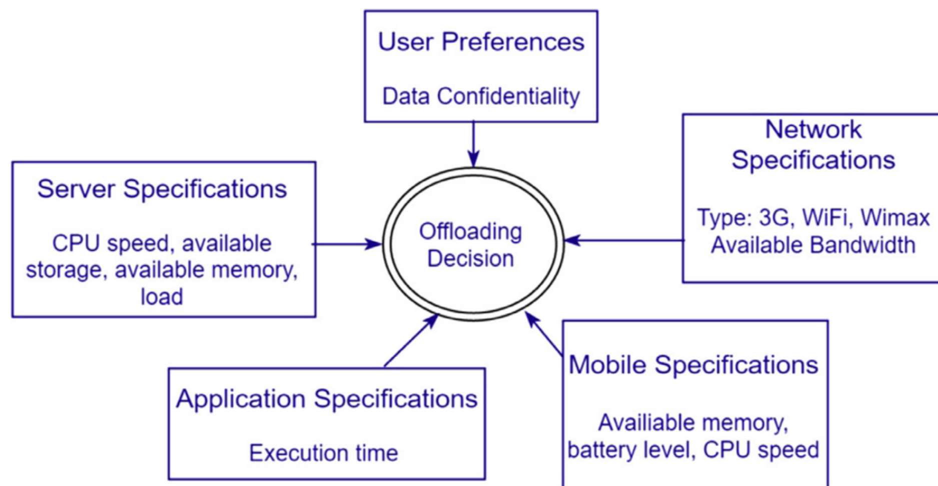

Figure 6 - Parameters affecting offloading decision [25].

2.3 Deep Learning & Object Detection

2.3.1 Object Detection

Object detection is a computer vision technique in which not only is every object in an image classified but localization is also used to locate the object and return a value for a bounding box around it. Deep learning has greatly advanced the performance of visual recognition systems in recent years. The popularity of deep learning has increased dramatically recently according to [26], due to a number of reasons such as: “the dramatically increased chip processing abilities (e.g. GPU units), the significantly lowered cost of computing hardware, and the considerable advances in the machine learning algorithms.” Deep learning in the form of convolutional neural networks have proven to be extremely successful in previous years in the various visual recognition challenge competitions such as Imagenet [27] and Pascal VOC [28].

In general, object detection models are evaluated using the mean average precision (mAP). Precision is a measure of how relevant the returned results are and is calculated as the percentage of true positives in the results.

$$precision = \frac{true\ positves}{true\ postives + false\ positves} \quad (1)$$

Recall is a measure of the truly relevant results that the system retrieves and is calculated as the percentage of correct results retrieved.

$$recall = \frac{true\ positives}{true\ positives + false\ negatives} \quad (2)$$

Average precision is a common metric that summarizes a precision recall curve which plots precision as a function of recall. Average precision is explained in [29] as being equal to taking the area under the curve and is defined as the precision averaged across all values of recall between 0 and 1.

$$\int_0^1 p(r)dr \qquad (3)$$

This value can be approximated by summing over the precisions at every possible threshold value which defines what the system will accept to be an object, multiplied by the change in recall.

$$\sum_{k=1}^{N} P(k)\Delta r(k) \qquad (4)$$

N is the total number of images, $P(k)$ is the precision at a cut-off of k-images and $\Delta r(k)$ is the change in recall that happened between cut-off k-1 and cut-off k.

2.3.2 Convolutional Neural Networks

Artificial neural networks, as shown in Figure 7, are described in [30] as consisting of a number of connected processors called neurons which each produce a sequence of real-valued activations. Activations are functions that define the output of a neuron given an input or set of inputs. Input neurons are activated by perceiving the environment and generally take input in the form of a multidimensional vector. A simple neural network contains three layers: an input layer, a hidden layer and an output layer, each of which transforms the aggregate activation of the network. Neurons in the hidden layers are activated through a weighted connection with other active neurons. Learning involves deriving weights which make a neural network perform a certain action, such as detecting an object. Deep learning also involves deriving weights to make the neural network perform a certain action however it involves computations across many hidden layers as opposed to just one.

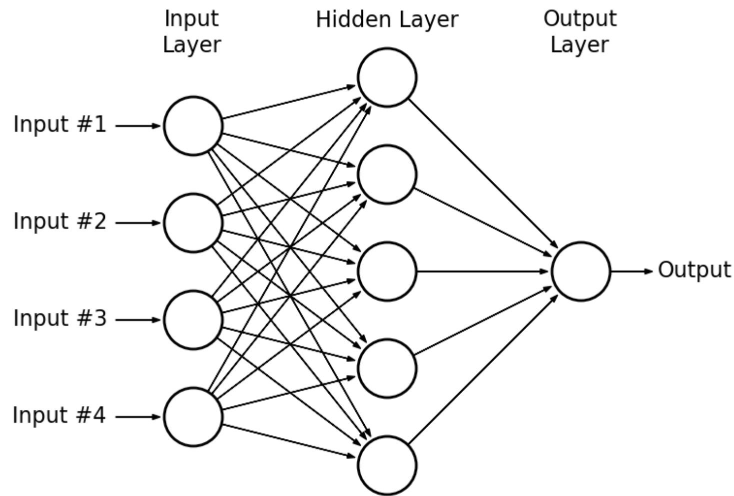

Figure 7 – A simple three layered neural network, comprised of an input layer, a hidden layer and an output layer [31].

Convolutional neural networks (CNN) are similar to standard neural networks (NN), in that they have layers made up of neurons with learnable weight and biases and all other functionality for neural networks still apply. The main difference between a CNN and a NN is that CNNs are tailored specifically for pattern recognition within images. This allows CNNs to make more suitable networks for tasks involving the processing of an image by encoding image-specific features into the architecture. These image specific features allow for a large reduction in the parameters required to set up the model [32].

The number of pixels in an image is determined by three parameters: height, width and number of colour channels. In regular images, there are 3 colour channels: red, green and blue. If a standard neural network is dealing an image of size 300x300 that means if a neuron in the first hidden layer is fully connected to each neuron in the input layer that means the neuron in the hidden layer has 300x300x3 = 270,000 weights. There would almost always be multiple neurons in the hidden layers and this would lead to the neural network becoming overly complex and would introduce overfitting.

There are two main differences between a convolutional neural network and a standard neural network. The first being that the neurons are modelled as a 3-dimensional array in a CNN with

the dimensions being, height, width and depth. Depth refers to the number of colour channels, so for coloured images this value would generally be 3. This reduces the number of neurons in the input layer by a huge amount. The second main difference is that not every neuron in a hidden layer is connected to every neuron in the layer before and only connect to a small region of the previous layer. This leads to a reduction in computational complexity and reduces number of parameters.

As demonstrated in Figure 8, there are three main types of layers in convolutional neural networks: Convolutional Layer, Pooling Layer, Fully-Connected Layer.

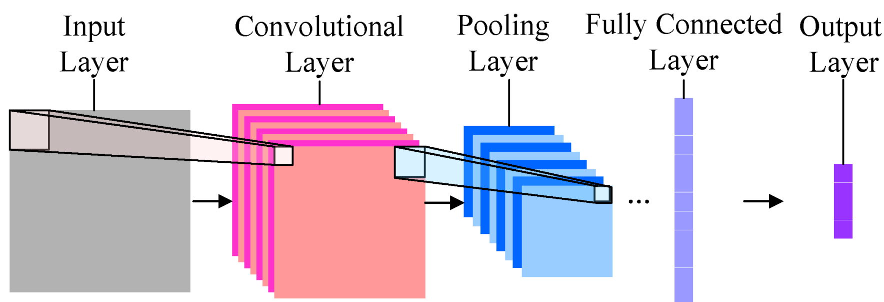

Figure 8 - Architecture of a convolutional neural network.

The convolutional layer is responsible for extracting features from an image, a feature is detected by a hidden neuron and is a pattern in the input that causes that neuron to activate. This could be anything such as an edge, a corner or a shape. When a feature is detected by a CNN it is reported in a feature map which is outputted from the layer. These feature maps are then used in the following convolutional layers to then extract more complex features eventually leading to the detection of an object. The convolutional layer works by determining an output of neurons which are connected to local regions of the input by applying a series of dot product calculations between weights in that layer and the input. Weights in the convolutional layers are often referred to as the kernels/filters. These weights are trained to learn certain features in an image.

Pooling layers are commonly placed between two convolutional layers. Its function is to progressively reduce the number of parameters and computation in the network, and thus control overfitting. The pooling layer reduces the complexity of the model by down sampling

along the spatial dimensionality of the given input [32]. Max-pooling is the method most used in CNNs. Max-pooling is achieved by applying a max filter to subregions of the input. For each of the regions that the filter is applied to the max value is added to a new output matrix. An example representation of this calculation is provided in Figure 9.

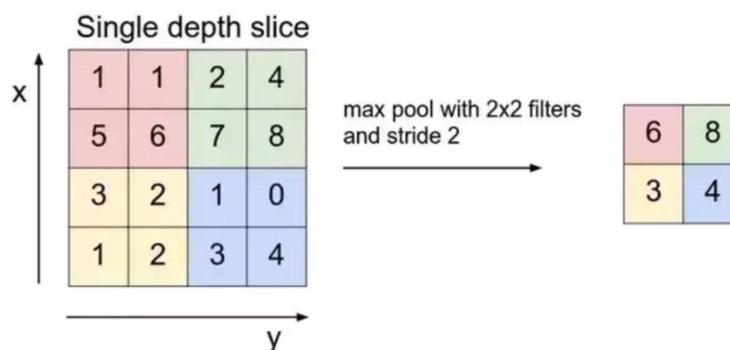

Figure 9 - Representation of max pooling calculation.

The fully connected layer, as the name suggest contains neurons that connect to every neuron in the previous layer. This layer produces discrete probabilities for each class, for example in object detection these classes could be things like a person, a dog, etc. The max probability will then be chosen as the resulting output.

Concolutional Neural Networks are extremely powerful but very resource heavy. For example take a filtering a large image of size 227x227 with 64 kernals, this will result in three activation vectors of size 227x227x64, which roughly calculates to 10 million activations or a huge 70mb of memory per usage [32]. Only recently, with the increased processing power of computers, has it been possible to achieve real-time object detection with a high accuracy using convolutional neural networks. Processing power in mobile devices however is still constrained and real-time detection is not yet a possibility.

3 State of The Art

This chapter describes the current state of the art in object detection and tracking and provides an overview of some related works in the field of mobile object detection through computation offloading. These related works are a subsample of the research that was completed, taking the research that was most relevant to what was investigated in this dissertation.

3.1 Object Detection

3.1.1 YOLO

In the past object detection systems such as R-CNN [33], were performed in a 2-step process. First, region proposal methods are used and what could be tens of thousands of proposals for bounding boxes were made around what the system believed could potentially be objects. The second stage of the process involved an image classifier classifying the image inside these bounding box proposals to assess whether the object was of a particular type. This process is extremely accurate however it is not a very efficient solution. These networks are slow and hard to optimize because each individual component must be trained separately.

You Only Look Once (YOLO) [34] is an object detection network that achieves real-time detection by reframing object detection as a single regression problem and uses only one neural network for both bounding box proposals and object recognition. YOLO works by dividing the input image into an S x S grid. Each of these grid cells acts as a classifier and is responsible for generating bounding boxes and confidence scores for potential objects. A grid cell is responsible for detecting an object if the centre of that object lies in that grid cell. This process is demonstrated in Figure 10.

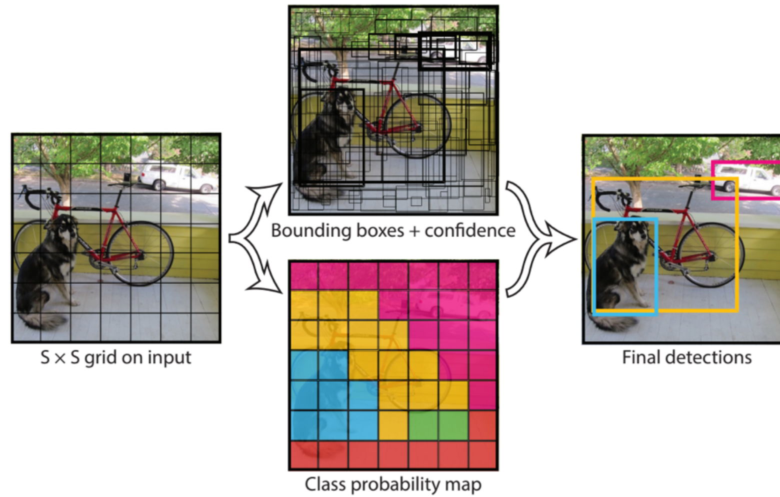

Figure 10 - The YOLO model (different colours in class probability map refer to resulting class predicted by the CNN) [34].

While it is common for CNNs to be structured like in Section 2.3.2, It is also possible to have multiple convolutional layers stacked in a row, this allows for more complex features of the input vector to be selected. It is also beneficial to split large convolutional layers into smaller sized layers as this reduces the amount of computational complexity within each layer [32]. YOLO like many other of the state of the art object detection systems employ these strategies in its own custom network structure, which is displayed in Figure 11.

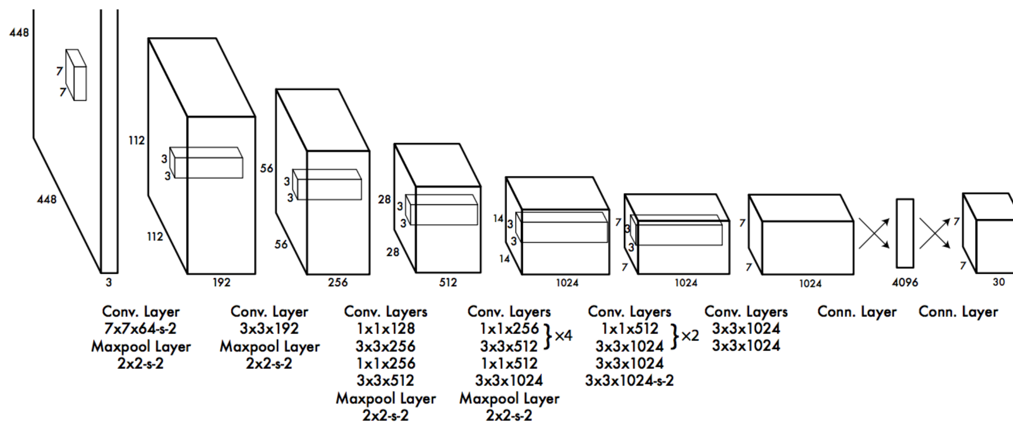

Figure 11 - The YOLO detection network [34].

YOLO achieves a mean average precision of 63.4 at a speed of 45 frames per second (FPS) when tested on the Pascal VOC detection challenge dataset [28]. In comparison, the state of the art model at the time Faster-RCNN VGG16 had a much higher mAP of 73.2 however that only ran at 7 frames per second. YOLO also provides a faster version, Tiny YOLO, that runs at 207 frames per second with a mAP of 57.1. This faster version uses a neural network with fewer convolutional layers (9 instead of 24). This makes it viable to run this version on a mobile device. The use of fewer convolutional layers makes the model much faster but also much less accurate.

In 2017 a new version of YOLO was released [35]. This version pooled a number of past works along with the authors own novel ideas to improve YOLO's performance. YOLO v2 boasts a state of the art performance of 78.6 mAP on the Pascal VOC detection dataset while running at 40 frames per second. A lower resolution version of YOLO v2 runs at 67 fps with a mAP 76.8. A comparison of different object detection systems provided in [35] is shown in Table 1.

Detection Frameworks	Train	mAP	FPS
Fast R-CNN [5]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[15]	2007+2012	73.2	7
Faster R-CNN ResNet[6]	2007+2012	76.4	5
YOLO [14]	2007+2012	63.4	45
SSD300 [11]	2007+2012	74.3	46
SSD500 [11]	2007+2012	76.8	19
YOLOv2 288 × 288	2007+2012	69.0	91
YOLOv2 352 × 352	2007+2012	73.7	81
YOLOv2 416 × 416	2007+2012	76.8	67
YOLOv2 480 × 480	2007+2012	77.8	59
YOLOv2 544 × 544	2007+2012	**78.6**	40

Table 1 - A comparison of detection frameworks on Pascal VOC 2007+2012. The various versions of YOLO are based on the input image size, the actual model is the same for each [35].

3.2 Tracking

The need for detecting corners, image features or interest points in tracking objects from frame to frame is described in [36]. The use of these features makes it much easier to track objects due to a reduction of the amount of points to be tracked and also because these features are complex and provide values which allow them to be tracked.

3.2.1 FAST

FAST (Features from accelerated segment test) [37] is a corner detector that is both simple and very fast. FAST works by taking a circle of points around the pixel being processed. This pixel is determined to be a corner if there is an adjoining path of at least 9 pixels around the pixel with a brightness value either brighter or darker than the brightness of the current pixel being examined. Non-maxima suppression is used to ensure not more than one response is returned for each corner. Non-maxima suppression is a process whereby, if neighbouring pixels are both detected as corners, the pixel which has the biggest difference in pixel brightness with its surrounding pixels is chosen as being the most likely to be a corner and the neighbouring pixels are supressed. According to [36] this detector is 5-10 times faster than Harris [38] and 50 times faster than the much more complex feature detector, SIFT [39].

3.2.2 Feature Based Optical Flow

Optical flow is a technique used when tracking objects, whereby, an apparent motion direction and magnitude is calculated for every point in an image. An example of optical flow is demonstrated in Figure 12, in which an object moves and the arrows represent the optical flow calculations which characterize the movement of the object. Optical flow can also be calculated for only feature points within the image. This involves much less computations and is much faster than calculating for every pixel in the image. Following the calculation of bounding boxes in object detection it is possible to then calculate feature points within these bounding boxes and then calculate the optical flow for each of these feature points in each frames following this detection.

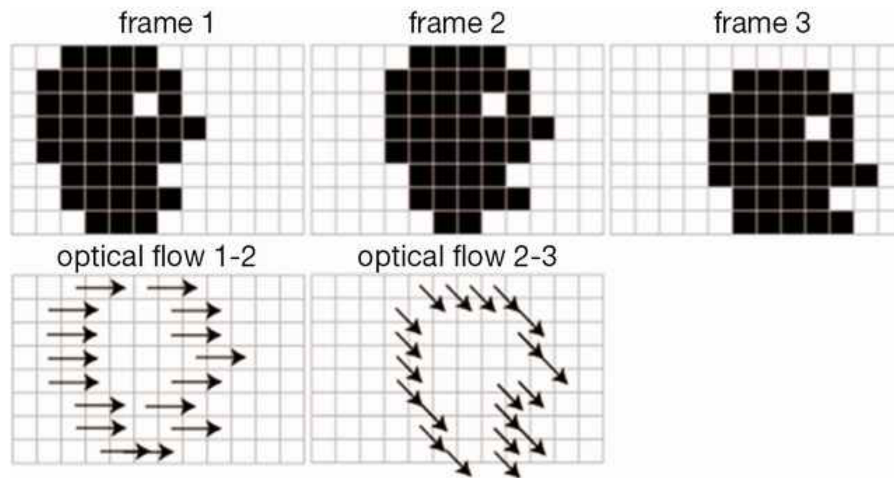

Figure 12 - Detection of the optical flow in 3 temporally-consecutive images, showing the movement of an example shape." http://tcr.amegroups.com/article/view/3200/html

Lucas-Kanade [40] proposed an algorithm for computing optical flow which is faster than other traditional techniques. The Lucas-Kanade algorithm aims to associate a movement vector (u,v) to every feature in the image by comparing two consecutive images. It uses the intensity gradient of an image to detect the motion between two intensity images. Optical flow is estimated by minimizing the sum of squared error between the two images. A simple description of the technique employed by this algorithm is given in [41].

3.3 Mobile Cloud Computing

3.3.1 Glimpse

Glimpse [42] is a continuous real-time object recognition system for mobile devices. Glimpse tackles the problems caused by latency from offloading detection to a server. When offloading to a server, it can take a large amount of time for the results to be returned. During this time, the detection will be old and the bounding box provided will be in the wrong location. Glimpse employs local tracking and an active cache in order to tackle this problem. An active cache stores all frames during the time it takes for the detection results to return from the server. A subsampling of these cached frames is chosen with which to track features from frame to frame.

When the detection results return from the server the bounding boxes catch up to the current frame using the tracking of the active cache and subsequent frames are then tracked in real-time. Glimpse also reduces bandwidth consumption by not sending every frame to the server. It chooses when to offload to the server in three ways, it first checks the tracking performance and when it is about to fail it offloads. Second, it detects any changes in the scene and offloads if there is a significant amount of change. Finally, if the scene is changing a significant amount, it avoids offloading too many frames by keeping track of the number of frames being offloaded currently. Glimpse achieved impressive results and the research completed in this dissertation compliments this work by using many of these techniques in the final framework solution. Glimpse however does not use deep learning in its object detection and can only detect faces and road signs. We thought it would be interesting and greatly beneficial to take these techniques and test them using the current state of the art object detection systems.

3.3.2 A Computational Offloading Framework for Object Detection in Mobile Devices

[43] propose a computational offloading framework for object detection in mobile devices. This framework consists of three parts: a monitoring engine, an offloading engine and a communication engine. The monitoring engine monitors the overall performance of the application through collecting a number of metrics such as: processing time, memory usage, battery consumption and the required data to complete a detection. This framework then uses these gathered metrics in the offloading engine to make a decision on whether to offload or not. The decision model proposed in [23] is used to make this decision. This decision model has a focus on saving energy and calculates the minimum bandwidth required to make offloading to the cloud the correct decision. The communication engine then offloads images to the server for detection. This framework shows a response time speedup that could reach five times for small images and 1.5 times for larger images. Energy saving ranges between 50% to 80%. While these are good results, this framework does not attempt to mask any of the impairments caused by a large amount of latency.

3.3.3 Delivering Deep Learning to Mobile Devices via Offloading

[22] conducted a preliminary research into the viability of a framework that uses current networking conditions and back end server loads to intelligently determine an offloading strategy. Similar to my research they created a mobile application which runs Tiny YOLO while using YOLO v2. In a cloud server. Changes in the system parameters of video resolution, deep learning model size and offloading decision where tested against the object detection accuracy, battery consumption and latency. The impact of varying network conditions was also tested and conclusions were made. These tests provided invaluable knowledge before planning research and it was possible to use the results of this investigation in the design of the offloading framework.

4 Design

This chapter describes the design of the proposed framework, which is shown in Figure 13. The system is described in two parts. The first part describes a mobile application which combines tracking, object detection and a decision-making system to achieve optimal speed, accuracy and energy usage. The second part describes a cloud platform which provides a fast and accurate object detection service and also provides context on networking conditions. The overall design and reasons for these design decisions are also discussed.

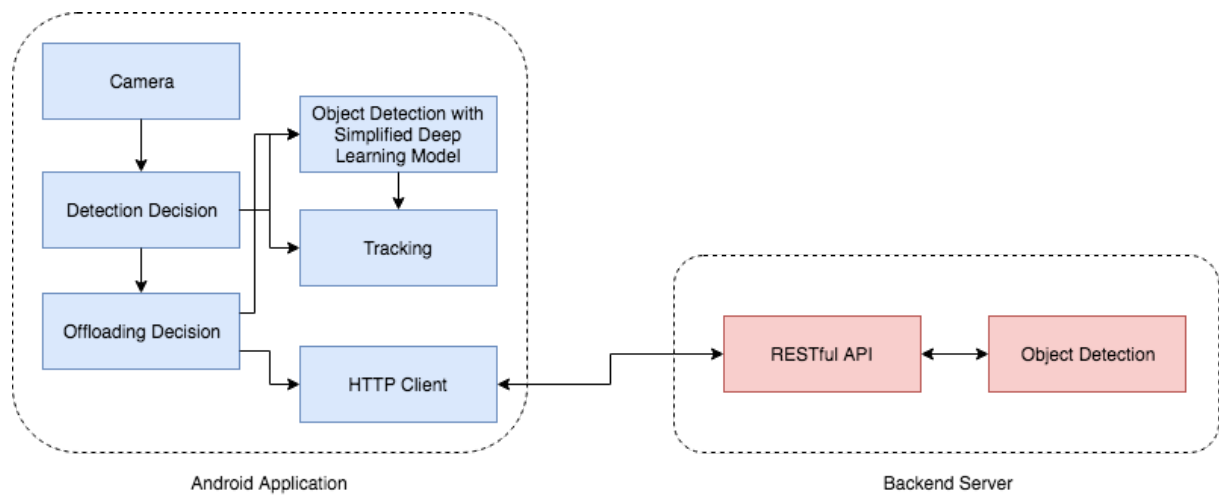

Figure 13 - Architecture of proposed framework.

4.1 Problem Setup

The real-time requirement of this framework imposes a hard deadline for generating an overlay of the object detection results for each frame. The handset has a number of options for generating the overlay for a given frame:

- Send all or part of the frame to a deep learning server in the cloud which then returns bounding boxes and tags. Communication is over a wireless link which may suffer from variable delay (due to changes in contention/bandwidth allocation, retries due to packet loss, queueing and so on).

- Process all or part of the frame using a simplified deep learning algorithm executed on the mobile handset. This avoids the need to communicate over the wireless link but comes at the cost of (i) a lower accuracy overlay (due to the use of a reduced deep learning model with fewer layers) and (ii) increased drain on the mobile handset battery.
- For objects already tagged in a previous frame the mobile handset can use tracking to update the bounding box in the next frame. This relies on their being relatively small changes in the image of the object between frames, and so the quality of the estimated bounding box tends to degrade over time (e.g. due to changes in the size and orientation of an object making it hard to track). This avoids the need to communicate with the cloud and also incurs minimal battery drain.

Since a frame may contain several objects, a combination of these three analysis approaches may be employed e.g. the handset may choose to use tracking to update the overlay for objects recently tagged using the cloud, use the cloud to generate an overlay for stale objects and for new parts of the frame (due to camera motion the field of view can change from frame to frame) and fall back to local processing if the cloud is too slow to reply (due to fluctuations in the wireless link delay). The task for the handset is to select which combination to use (and the associated partitioning of the frame between these analytic tools so as to meet frame deadlines while maximising the accuracy of the overlay and respecting battery constraints.

4.2 Mobile Application

4.2.1 Tracking Decision

The framework is designed so that there are two decisions to be made before passing an image into a CNN for object detection or before offloading it to the server. The first decision is whether tracking detected objects from previous frames can be relied upon. Techniques implemented in Glimpse [42] as described in Section 3.3.1, were implemented in the proposed solution. This decision is based on two things:

1. The amount of change in the scene between two consecutive frames.
2. Tracking failure due to a change in the size, angle or appearance of the objects.

The amount of change in the scene is calculated using a frame differencing function. This lightweight function calculates any movement or change between two consecutive frames by first converting the frames to greyscale and then calculating the absolute difference, $d_{i,j}(x, y)$, between pixel values of the current frame, $f_k(i, j)$, and the previous frame, $b(i, k)$.

$$d_{i,j}(x, y) = |f_k(i, j) - b(i, k)| \qquad (5)$$

The overall difference is then calculated with the formula below. This overall difference is divided by the max amount of change that could potentially be in the scene (width x height), to calculate a percentage change in the scene. Computing the frame difference is linear to the size of the image and takes only a few milliseconds on the mobile device. If the calculated percentage is above a certain threshold, the object detection is either completed locally or offloaded, if not, only tracking is executed on previously detected objects.

$$d_{i,j} = \sum_{x,y} d_{i,j}(x, y), d_{i,j} \geq 0 \qquad (6)$$

Based on experiments performed in [22] and recreated in this dissertation, where the effects of the changes in image resolution on the latency in the system were tested, a solution was formed based on only offloading the parts of the frame that have a significant amount of change. As you can see in Figure 14, the latency increases with the image size for both offloading and local processing when utilizing the YOLO model.

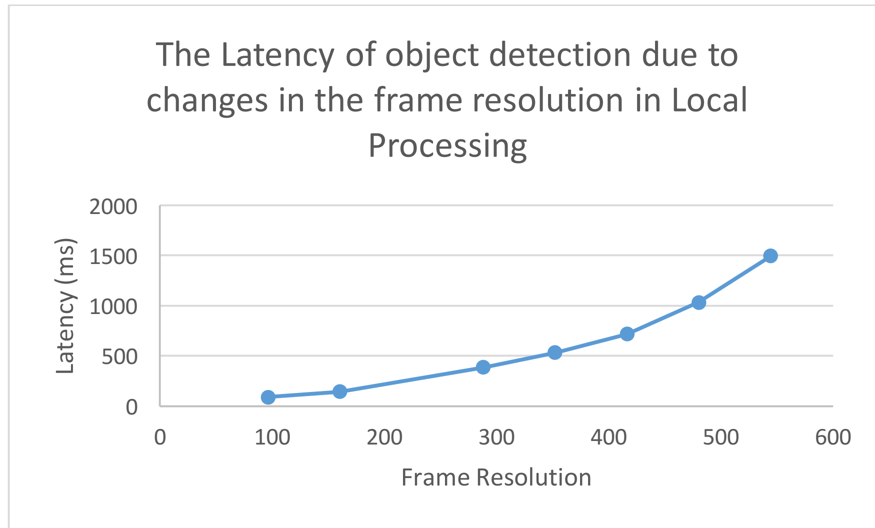

Figure 14 - Chart plotting the effect of frame resolution on latency.

This process, as demonstrated in, Figure 15, involves first splitting the image into 4 sections and calculating the frame difference, as described above, in each of these 4 sections. If the results for only one section exceeds a specified threshold then that section is cropped and is either offloaded or passed into the local object detection model. When the results are returned for this section of the image, the pixel location of the returned bounding boxes are then mapped back to the original pixel values of the full frame.

Figure 15 - Cropping scene based on amount of change in parts of the frame.

4.2.2 Offloading Decision

As explained in Section 2.2.4 it is important for the application to be contextually aware of all of the conditions that affect the offloading decision. Therefore, one of the main requirements when designing the framework was the collection of any data that could affect the cost of offloading in order to make intelligent decisions. Firstly, metrics with regards to the current networking conditions were gathered, these include the current network type, the link speed, the latency, the bandwidth and signal levels. Metrics with regards to the context of the system

include, the battery usage, time taken to perform local processing, the current CPU usage and the current memory usage.

A simple offloading decision model was created based on experimentation and the gathered metrics. This offloading decision model was created to test the framework with a view of creating a much more intelligent model in future work which utilises all of the metrics gathered. The offloading decision was implemented as a series of conditions based on the gathered metrics. The framework has 2 states, local and offloading. The current state of the application is decided based on this series of conditions which include:

- If network is unavailable, if there is low signal or if the network type is worse than 3G then the application is run in local processing state. Based on research anything below 3G such as 2G or EDGE meant that it was costlier to offload than to perform the processing locally.
- If bandwidth drops below a certain level the resolution is decreased. If the bandwidth drops further meaning its less expensive to run locally then state is changed to local.
- If a request returns an error or times out then the application falls back on local processing.

This offloading decision model can be easily swapped with another decision model in future work which can utilise the metrics gathered.

4.2.3 Object Detection

As mobile devices have constrained resources, deep learning models which are configured to use limited processing and memory are preferred. The application allows for the easy input of various different object detection models. These models come in the form of pre-trained neural networks and can be loaded in from memory. The object detection model that was chosen for the mobile device was Tiny YOLO. This model was chosen because, as described in Section 3.1.1, it uses a smaller number of convolutional layers resulting in it using less processing and memory than YOLO and results in it being much faster. However, this comes with a trade-off with accuracy.

The object detection is designed to run on a background thread. This allows a smooth output of the camera feed on screen. Whenever a detection is completed the results are either passed into the tracker or drawn on screen.

4.2.4 Tracking

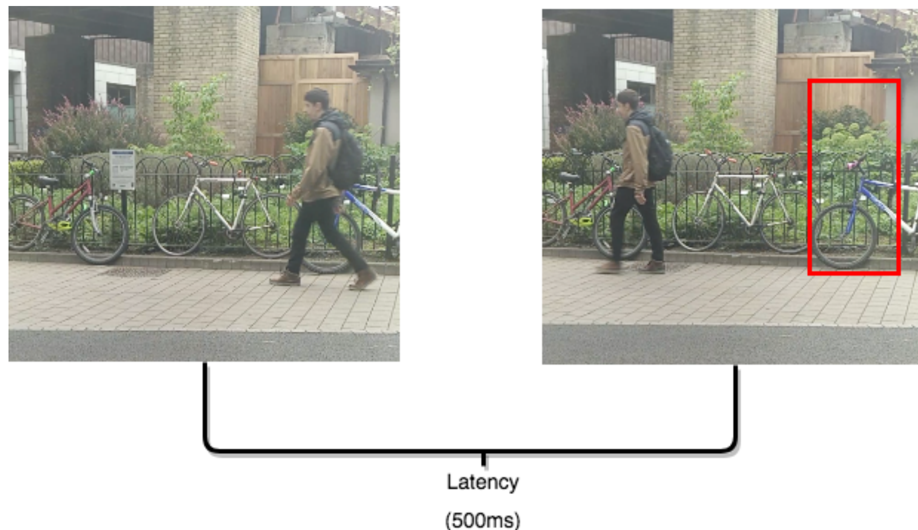

Figure 16 - Stale frame due to latency of object detection.

Tracking was implemented to combat the decrease in accuracy due to latency in the system. As we can see in Figure 16, when a frame is grabbed from the camera and is assessed by the object detection system, by the time the results are returned the person has moved and these results are now stale. The calculated bounding box is then placed in the wrong location leading to a decrease in accuracy.

As described in Section 3.2 object tracking is completed in two steps:

1. A suitable feature detection algorithm detects salient features in the image, that will then be tracked from frame to frame. For the purposes of this project, FAST feature detection was used as speed is vitally important.
2. The optical flow is then computed using the Lucas-Kanade algorithm, which is used as it is faster than other traditional techniques.

This tracking is run on every frame on a background thread so as not to add additional latency to the system. The tracker stores the optical flow keypoints of every frame at a timestamp so when a detection is completed the path of the features within the returned bounding box are followed and the tracker walks the position forward along the collected keypoint deltas. This results in the bounding boxes being placed in the correct location of the current frame.

4.2.5 Offloading

Both streaming the camera feed to the backend server using Real-time Transfer Protocol (RTP) over the User Datagram Protocol (UDP) and simply sending one frame at a time over Hypertext Transfer Protocol (HTTP) over the Transmission Control Protocol (TCP) were assessed when designing the offloading functionality.

RTP is a network protocol which focuses on delivering audio and video over IP networks [44]. RTP is used extensively in communication and entertainment systems. RTP utilises UDP instead of TCP as it relies on timeliness and the speed of streaming video/audio. RTP is insensitive to packet loss and does not require the reliability of TCP. UDP has less overhead for headers so one packet can carry more data, thus using the network bandwidth more efficiently. HTTP in general runs over a TCP connection. TCP is much more reliable than UDP as each packet sent from client to server is acknowledged and if there are lost packets the client will then retry sending these packets [45]. This means that in general there is no packet loss, but this acknowledgment leads to the connection being slowed down. In UDP streaming there is no acknowledgement and is therefore much faster. UDP does not care for packet loss, what matters most is on-time delivery of content.

[46] provide a comparison between TCP and RTP for video streaming. The two protocols were compared with regards to data integrity and transmission delay. It was found that TCP was more reliable when it comes to data integrity, as is the nature of the protocol, there is no loss when transferring packets and data integrity was 100%. RTP was between 97% and 99% however and this had little effect on playback quality. When testing the transmission delay there was a

large difference between the two protocols. TCP added between 500ms to 620ms in transmission delay whereas RTP was between 20ms and 70ms. Therefore, this shows that RTP can effectively reduce the transmission delay and improve efficiency.

Although it was found that for streaming media RTP is generally faster and more widely used, in the final solution the images were offloaded to the server using HTTP. This was due to a number of reasons, the first being HTTP is widely interoperable and has countless implementations that could be used both server and client side. Whereas, the implementation of RTP varies and is difficult to incorporate into an application. Having done research it was found that there is a lack of libraries for implementing RTP effectively and it was decided that a HTTP offloading strategy would be acceptable for the initial testing of the proposed framework. This allowed time to be spent on other more important aspects of the research. RTP is to be investigated further and to be incorporated in the design of the framework in future work as it should provide a speed up to the offloading process.

The overall processing flow of the application is shown in Figure 17.

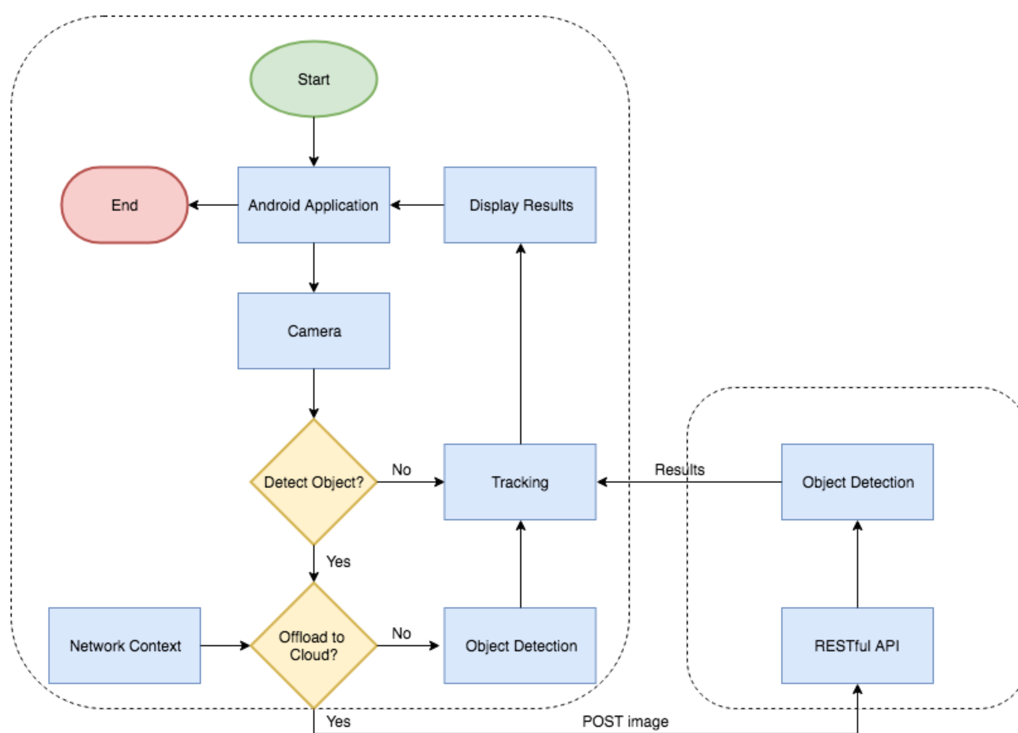

Figure 17 - Processing flow of proposed application.

4.3 Backend Server

The backend server handles a number of tasks including object detection, providing network context to the mobile application and also processing results received from the device when in test mode. This backend server is in the form of a multithreaded RESTful API and the endpoints are shown in Figure 18 below. The main use case for this server is the object detection endpoint. This endpoint listens for input in the form of a multi-part image and returns the results of object detection on this image. Each image is processed sequentially. This object detection is completed using pre-trained neural networks. The Pre-trained convolutional neural networks are only loaded once upon the server starting and are then utilised for object detection.

YOLO v2 as described in Section was utilized as the object detection system server side. As time is of the utmost importance in our application, YOLO was chosen for the task. Another reason why YOLO is a good candidate for the proposed framework is that it can dynamically change its network model based on the resolution of the inputted image and this allows for more control. The computation time of the network scales directly with the input resolution.

Further information on the specifics of this RESTful API and how it was implemented are given in Section 5.2.

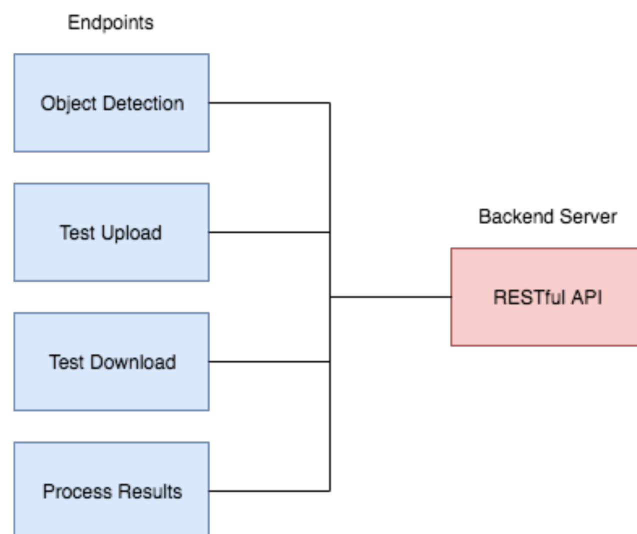

Figure 18 - Overview of RESTful API.

5 Implementation

This chapter describes the implementation of the mobile cloud computing framework. Section 5.1 describes the development of an Android application while section 5.2 describes the development of the backend server as a RESTful API.

5.1 Android Application

5.1.1 Camera & Settings

The proposed front-end solution was created as an Android application and written in a combination of Java and C++. An example android application [47], created by Tensorflow was utilised as a base for the development of the proposed framework. This example application provides object detection and tracking functionality.

On start-up, the app launches a menu with two options, the first being an option for settings and second for launching the application. This menu is shown in Figure 19. The settings menu, shown in Figure 20, uses the Android SharedPreference[1] interface to access and modify preference data. This creates a single instance of the SharedPreference class and these preferences can be accessed from anywhere in the application. Various parameters are set using this settings menu such as, the resolution of the camera, whether to enable tracking, whether to enable test mode and whether to enable offloading.

[1] https://developer.android.com/guide/topics/ui/settings

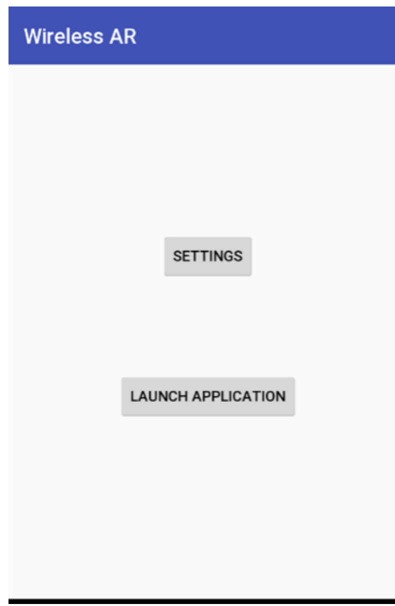

Figure 19 - Main menu of android application.

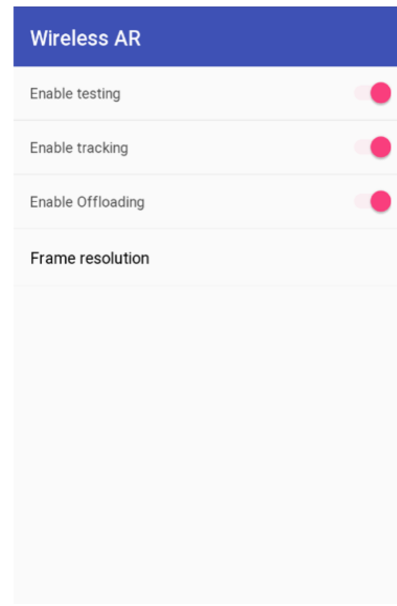

Figure 20 - Settings menu of android application.

When these preferences are set by the user and the user clicks the "launch application" button a new camera activity is launched which then starts a fragment. An activity in an android application represents a single screen, while a fragment represents a sub section or process within this activity. The fragment initializes the camera and then initializes the object detection interface using the predefined settings from preferences. This camera fragment then feeds a stream of images from the camera into the object detection interface.

5.1.2 Tracking

Each of these images are processed and then tracking is run on the current frame. Tracking is run on a background thread so it does not slow down the application. The feature points in that frame and the optical flow between the detected feature points of the current frame and the previous frame are calculated using C++. Android provides a Native Development Kit (NDK)[2] which are a set of tools that allow the use of C and C++ with Android. This C++ code is wrapped in a Java class which can initialize and call functions and classes from the C++ tracker code using the Java Native Interface (JNI)[3] framework. The resulting optical flow keypoints from

[2] https://developer.android.com/ndk/guides/
[3] https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

the tracker are stored in a list with the current timestamp attached. Following this, if the application is in debug mode, these feature points are drawn on screen as shown in Figure 21.

Figure 21 - Tracked feature points being drawn on screen in debug mode of android application as small multi-coloured rectangles.

5.1.3 Tracking Decision

The object detection and decision-making processes take place on a separate thread from the camera and the tracking. Once the current frame is received from the camera and processed, that frame and the previous frame are passed into the tracking decision interface. In this interface, firstly the change in the scene is calculated as described in Section 4.2.1. The amount of change in the scene was originally computed using Java however this was returning results of around 300ms which is not acceptable for the real-time requirements of the application. Upon rewriting this code in C++, the speed dropped down to around 8ms. If the overall percentage exceeds a threshold then the decision to perform object detection is made. This value, based on experimentation was set at a value of 10%. If only one section of the frame contains a significant amount of difference then that section is cropped and passed into the object detection interface.

Some basic pseudocode giving an overview of the object detection flow is given below in Figure 22. So as to keep this pseudocode simple and easy to understand the cropping of the frame is not included.

```
initializeThread(){

    Boolean sceneChange = trackingDecision.makeDecision(currentFrame, previousFrame);

    OffloadingMode offloadingMode = offloadingDecision.makeDecision(networkContext, systemContext);

    frameResolution = offloadingDecision.getOptimalResolution(networkContext, systemContext);
    detector.setInput(frameResolution);

    if(offloadingMode == OffloadingMode.HTTP && sceneChange == True){
        List<DetectionResults> results = offloadToServer(currentFrame);
    }
    else if(offloadingMode == OffloadingMode.LOCAL && sceneChange == True){
        List<DetectionResults> results = tensorflowObjectDetection(currentFrame);
    }

    tracker.trackResults(results, currentFrame, currentTimestamp);

    drawResults();

}
```

Figure 22 - Pseudocode of object detection flow in mobile application.

5.1.4 Offloading Decision

Before passing a cropped frame, or the entire frame into the object detection model, first a decision had to be made on whether to offload or not. This decision is based on a number of metrics gathered by the system. The system context including the local processing time, current battery level, current CPU usage and current memory usage are collected using Android developer tools. The current networking conditions such as the link speed, current network type

and signal strength are gathered through both the use of Android developer tools and through querying the backend server. Originally another thread was used to calculate the bandwidth and latency of the network in the background. This was completed by both uploading and downloading a file from the backend server and calculating the bandwidth based on the file size and length of time taken to upload/download the file. This however, proved to be a poor way of calculating bandwidth as it uses up the available bandwidth and may affect the offloading process. A much simpler technique was implemented in the final solution whereby, the time taken to perform the object detection server side is returned with the results. This time can then be subtracted from the time taken for the results to be returned in order to calculate the bandwidth after every offload. Should the networking conditions be poor however and the device chooses not to offload, the original solution for calculating bandwidth can be used until networking conditions improve and we begin offloading again.

This decision model is created in the form of an interface allowing, the easy substitution of different models into the code for testing different decision models in future work. As described in Section 4.2.2, currently a very basic decision model is implemented in which a series of conditions are checked such as the current network context to decide on whether to offload or not. The state of the system is controlled using an enum value, either "OFFLOADING" or "LOCAL". The results of this decision model simply change this enum value based on the conditions.

5.1.5 Object Detection

5.1.5.1 Tensorflow

Tensorflow [48] is an open source library with a focus on expressing and implementing machine learning algorithms. This library was used to implement the deep learning object detection models on the android application. [49] carried out an extensive study of the various popular deep learning toolkits. These frameworks were assessed in 5 categories: modelling capability, interfaces, model deployment, performance and architecture. It was found that Tensorflow was either ranked the best or the second best in each of these categories. Based on this review

Tensorflow was chosen for this project as well as other reasons such as the fact that Tensorflow is an extremely popular and widely used framework and there is an abundance of tutorials and documentation to assist me if there were any issues.

Tensorflow models machine learning algorithms as dataflow graphs. These graphs contain a set of nodes called operations and each of these nodes represent a unit of computation. These units of computation can range from being a simple addition operation to being a multivariate equation. The input to these graphs comes in the form of a tensor. A tensor is a multi-dimensional array of values. Each operation takes a tensor as input and outputs a tensor. The reason why it is called Tensorflow is that tensors flow from operation to operation based on the graph model.

5.1.5.2 Implementation

When the object detection is being performed locally, before being passed into the Tensorflow API the images have to be further processed. This processing involves, resizing the image and then creating an input tensor from the RGB values of that image. A bitmap image, which is the image data type used in Android, cannot be processed by Tensorflow as an input. This bitmap image must be converted into an input tensor. A tensor is a multi-dimensional array of values and is used by Tensorflow to pass data between its different operations. These values consist of the red, green and blue values for each pixel in an image. The format of a tensor is shown in Figure 23 below.

	0	1	...	223
0	101, 87, 230	96, 110, 207	... ...	101, 87, 230
1	101, 87, 230	101, 87, 230	... ...	101, 87, 230
...	... ...	... ...	... ...	101, 87, 230
223	101, 87, 230	101, 87, 230	101, 87, 230	101, 87, 230

Figure 23 - Input tensor format [50].

The pre-trained weights and a config file for the YOLO pre-trained neural network are downloaded from [51] and these weights and config file are then converted into a Tensorflow graph using the Darflow[4] tool. These graphs are then transferred onto the mobile device and read in from storage upon initializing the Tensorflow object detection API. Like the tracking implementation, the Tensorflow API core is written in C++ and utilized by a Java class through the JNI. The flow of object detection using Tensorflow is shown in Figure 24 below. The tensor which represents the current frame is simply passed into the Tensorflow API, this API will pass this tensor through the pre-trained neural network, in the form of a graph, and the results are returned as a tensor. The results are then extracted from this tensor and converted into a Java object for further use. These results include, the bounding box location of detected objects along with the predicted class for that object and a confidence score.

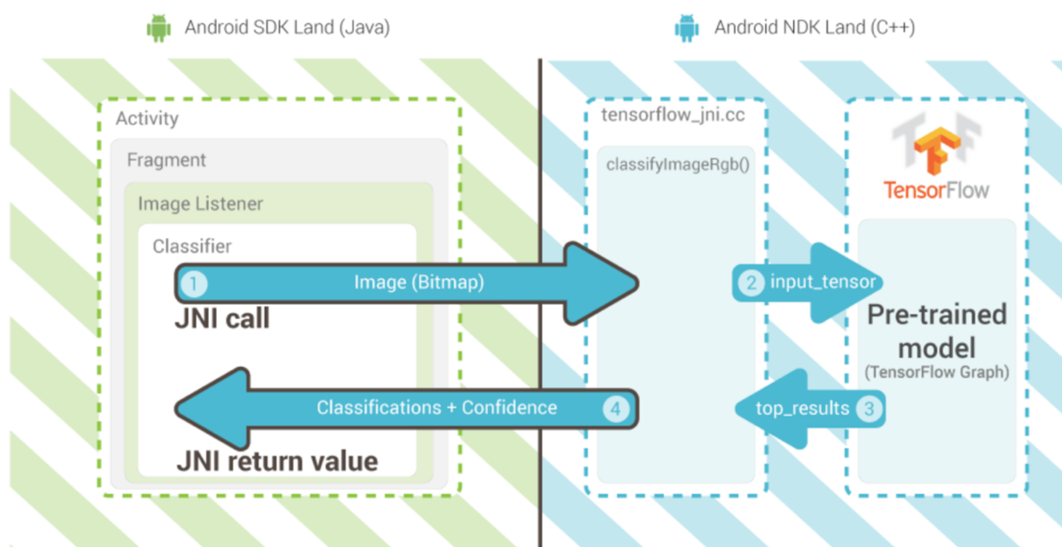

Figure 24 - Diagram of object detection flow using Tensorflow [50].

5.1.6 Offloading

The offloading is completed using OkHttp[5], which is a HTTP and HTTP 2 client for android and Java applications. The bitmap image is compressed, converted into a byte array and then

---

[4] https://github.com/thtrieu/darkflow
[5] http://square.github.io/okhttp/

sent as a HTTP Post request to the server. The application then waits for the results to be returned. When the results arrive, in json form, they are converted into a Java object using the Jackson[6] json deserialiser.

5.1.7 Results

Upon the detections being calculated through either offloading or locally, the results which are below a user defined confidence level are then filtered out and they are passed into the tracker. As described in Section 4.2.4 the tracker, traces forward the movement of the feature points contained within the bounding box in the results and calculates where the bounding box should be in the current frame. This alleviates the effects of the latency of the object detection. Once the results have been processed by the tracker the bounding box, confidence level and class, as shown in Figure 25, are drawn on screen. This whole process is then repeated for the next frame.

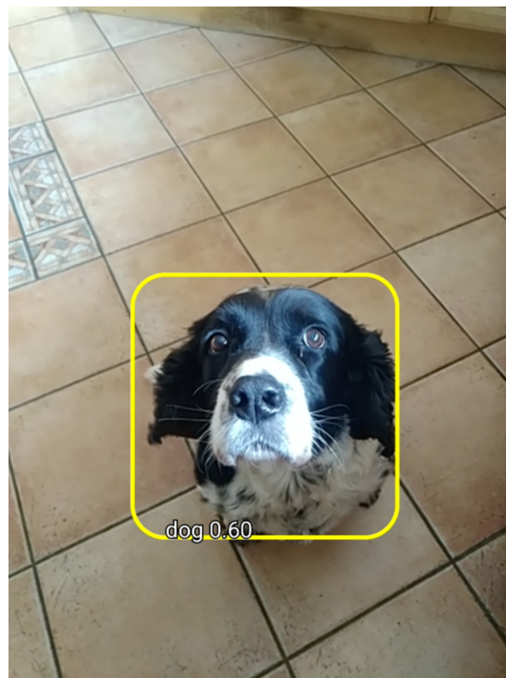

Figure 25 - Screenshot from mobile application displaying detection results.

[6] https://github.com/FasterXML/jackson-docs

5.2 Backend Server

Darknet[7] is an open source custom neural network framework written in C[8] and CUDA[9]. This framework provides an interface for running the various versions of YOLO. Flask was the framework chosen to implement the backend server. Flask is a micro-framework written in Python. Flask was chosen as it provides simplicity and it is lightweight. Flask keeps its core simple which allows freedom to choose additional libraries for things such as data storage. Each of the endpoints that were implemented and are described in Table 2.

Endpoint	Type	Parameters	Description
detect	POST		Performs object detection on received image and returns formatted results.
test_upload	POST		Used for testing download bandwidth by accepting the upload of a file and returning a response. The total time taken for the response to be returned is utilised by the mobile application to calculate upload bandwidth.
test_download	GET		Used for testing download bandwidth by returning an image to be downloaded by the mobile device. The mobile device can calculate bandwidth by timing this transaction.
results	POST		Takes results received from mobile application and writes them to json file for testing.

Table 2 - Description of RESTful API endpoints.

In order to run the YOLO detection model, the pre-trained weights and config file must be downloaded from [51]. In the python program, upon initialising the Flask server, the pre-trained neural network is initialised by Darknet using this config and weights file.

---

[7] https://github.com/pjreddie/darknet
[8] https://www.tutorialspoint.com/cprogramming/index.htm
[9] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

When the "detect" endpoint receives an image in the form of an array of bytes, this image must be converted into a numpy[10] array before being passed into the Darknet API for object detection. The results are returned with bounding box location, class of object and a confidence score. The bounding box location is returned in a different format than is needed on the android application so before sending back the results, the bounding box must be converted to match that of the front-end application. Darknet returns the bounding box with the location of the centre pixel of the bounding box and also the width and the height of the bounding box. The android application needs the location of the top left corner of the bounding box and the bottom right corner of the bounding box in order to be able to draw these bounding boxes on screen. Therefore, before sending the results back the bounding box location is converted to match that of the implementation on the android application. An example of a result returned from the API which demonstrates the formatting is shown in Figure 26 below.


```
[
  {
    "bottomright":{
      "x":216.10812,
      "y":294.69772
    },
    "confidence":0.8868352,
    "label":"bottle",
    "topleft":{
      "x":144.8456,
      "y":64.85053
    },
    "bottomRight":{
      "x":216.10812,
      "y":294.69772
    }
  }
]
```


Figure 26 - Example of object detection endpoint response.

[10] https://docs.scipy.org/doc/

6 Evaluation

This chapter describes the experiments that were performed on all three problem domains which have been implemented in the android application:

- Processing each frame using a simplified deep learning algorithm on the mobile handset.
- Sending every frame to the cloud.
- The proposed framework solutions to mask impairments.

These experiments were performed to test three performance metrics: energy consumption, latency and the accuracy of the detection. Each of these experiments were also be performed at various different networking conditions. This was achieved using the Linux tc traffic control tool[11] at the server side to induce impairments in the network. A comprehensive analysis is provided and conclusions are made based off of these results. The performance of the proposed framework solutions to masking impairments are discussed and also an analysis of the different factors that should be taken into consideration when building a more intelligent decision-making model in future work is also provided.

6.1 Evaluation Metrics

6.1.1 Latency

Latency in this system refers to the total time to do a detection on one frame of video and to draw this detection on screen. When run locally, this is the time it takes to execute the reduced deep learning model. When offloading to the server this is the total time it takes to retrieve a frame of the video stream, transfer it to the server, apply the deep learning model to the frame and return the resulting detection. The speed of each separate part of the application is also recorded and analysed.

[11] http://lartc.org/manpages/tc.txt

6.1.2 Energy Consumption

Energy consumption is gathered using Android developer tools. It is taken as a percentage decrease while running the application in its various form for a period of 30 minutes.

6.1.3 Accuracy

For the purposes of obtaining a measurement of accuracy for the object detection on the device the guidelines of The Pascal Visual Object Classes Challenge [28] are followed. In this challenge, the accuracy of object detection models are evaluated using the mean average precision which is briefly explained in Section 2.3.1. This Pascal VOC challenge determined that the precision-recall curve is the best way of characterising a classifier. Rather than comparing curves when evaluating classifiers, it is much more useful and simpler to compare a single value. [28] uses 11-point interpolated average precision and defines it as "the mean precision at a set of eleven equally spaced recall levels [0.0.0.1,…,1]".

$$Average\ Precision = \frac{1}{11} \sum_{r \in \{0,0.1,\dots,1\}} Pinterp(r) \qquad (7)$$

The precision at each recall level r is interpolated by taking the maximum precision measured for a method for which the corresponding recall exceeds r:

$$Pinterp(r) = \max_{\tilde{r}.\tilde{r} \geq r} p(\tilde{r}) \qquad (8)$$

where $p(\tilde{r})$ is the measured precision at recall $\tilde{r}$.

In order for a detection to be classified as being correct, the class has to match that of the ground truth and also the bounding boxes must overlap by over 50% using the formula

$$a_o = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})} \qquad (9)$$

where a_o is the area of overlap, B_p is the predicted bounding box and B_{gt} is the ground truth bounding box. Multiple detections of the same object are considered false detections.

6.2 Experimental Setup

In order to experiment with, and comprehensively test the application, new functionality had to be added to the mobile application to replace the camera feed with ground truth videos. This proved difficult however as reading frames from the video files could not be completed in real time and thus would affect the results. Instead, the application had to be taken apart and the multithreading removed and placed in a new activity. This new activity would read in a frame from the video file one at a time do the tracking and then a delay on the object detection was simulated by offsetting the detection by a number of frames from the current frame being processed. The results of these tests are sent to the flask server to be evaluated against the ground truth data. Upon receiving this data, the server appends the results to a text file in json form. Ground truth data was produced using the Darknet framework. A new python program was created in which a video file is opened using and a frame is isolated using OpenCV[12]. YOLO v2 object detection is then applied to each frame and the results are written to a text file in json format as shown in section 3.2.2. [52] is a simple utility tool that was used to evaluate the mean average precision between the results from my system and the ground truth images. This tool follows the guidelines and formulas as set out in [28]. This tool had to be adapted to read in text files and convert the json data to the correct format the tool needed. The mobile application was deployed on a mid-range mobile device, the Leeco LeMax2[13]. The backend server is equipped with a NVIDIA GTX 1080 GPU with 8GB of RAM.

---

12 https://opencv.org/
13 https://www.gsmarena.com/leeco_le_max_2-8051.php

6.3 Results

6.3.1 Local Processing

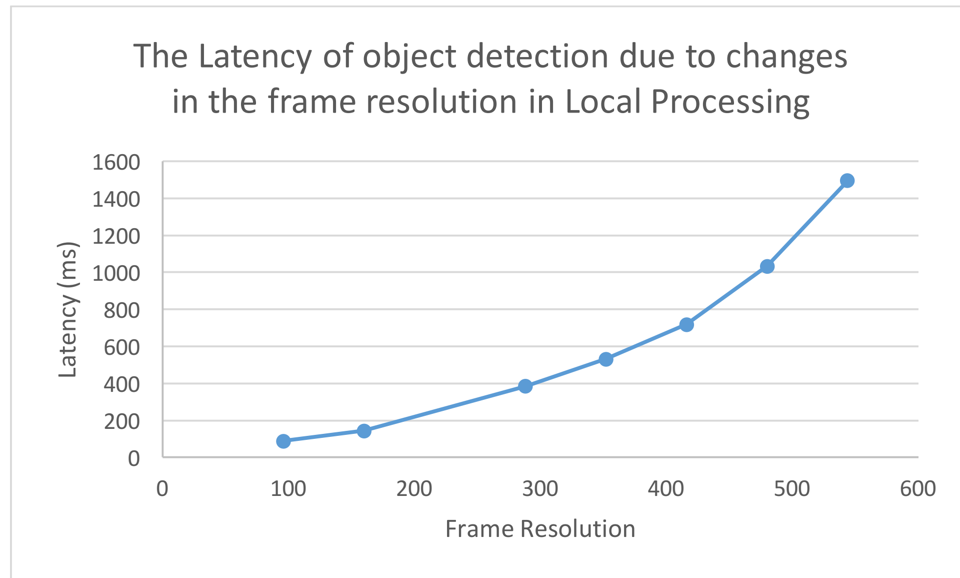

Figure 27 - The change in latency caused by changes in the frame resolution in local processing.

A number of evaluations were performed running the object detection solely on the mobile device using the simplified deep learning models. This was used as a baseline for further evaluation. Firstly, the latency is recorded at different image resolutions. As demonstrated in Figure 27, the average time when performing the object detection increases with frame resolution when using the Tiny YOLO model locally. This is because, as stated in Section 4.2.3, YOLO can change dynamically with image size and so speed is related to frame resolution. The larger the image the more processing that must be completed. The lowest possible latency when running Tiny YOLO locally is 88ms at a frame resolution of 96 x 96 which is a very small image size. This is equivalent to 11.36 frames per second which is far from the real-time requirement of 30 frames per second.

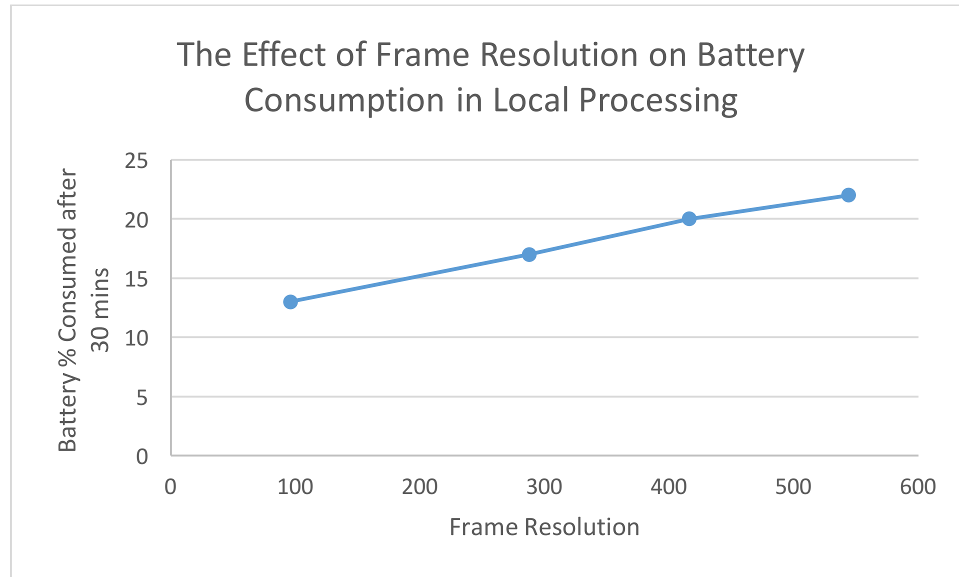

Figure 28 - The effect of frame resolution on battery on battery consumption in local processing.

Next the energy consumption is tested at varying frame resolutions, Figure 28. The smaller the resolution, the less battery consumed. At a standard image size of 416 x 416 the battery consumed after 30 minutes was 20%. This would mean the battery of the mobile device would be completely drained in the space of 2.5 hours. Energy consumption is of huge concern to users and having the battery drained in 2.5 hours would not be acceptable. As a comparison, we a video was played on the mobile device for the same space of time and this reduced the battery by 6%. These results show similarly to the latency calculations that energy consumption increases with frame resolution. Again, similar to latency this is due to the increased processing needed to detect objects in images of a larger size.

These tests on the Tiny YOLO model run on the mobile device show that the current frame resolution can be utilised when creating an offloading decision model. The predicted time taken for local processing must be evaluated with frame resolution in mind. When creating an algorithm to save energy, once again the frame resolution must be taken into account. There is however, a trade-off between energy consumption, latency and accuracy. According to [35], a decrease in image resolution can result in a significant reduction in accuracy. Choosing a smaller frame resolution may decrease the latency and the energy consumption but comes at a cost of accuracy.

6.3.2 Offloading

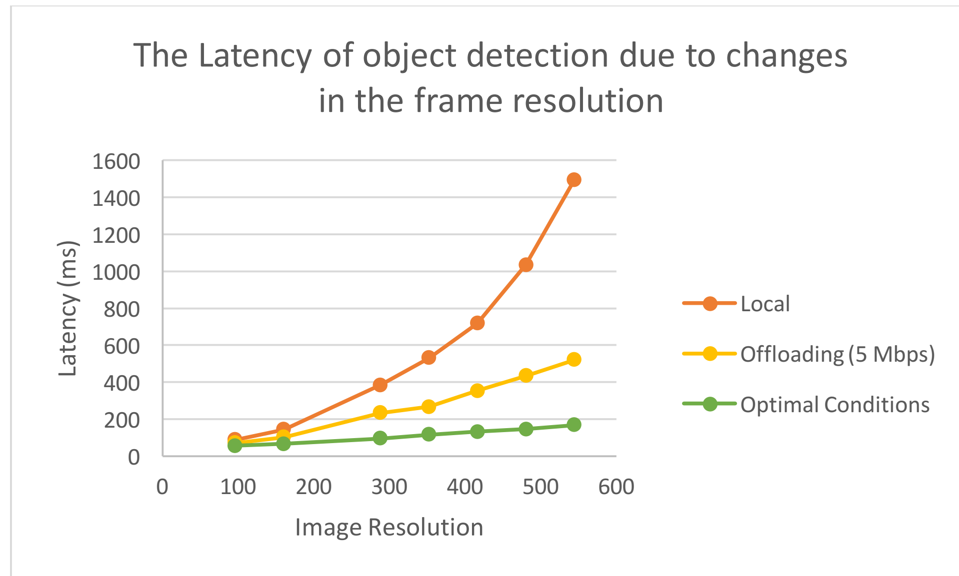

Figure 29 - The latency of the object detection calculation due to changes in the frame resolution. Offloading is set at 5Mbps and optimal conditions refers to an unrestricted connection with a private network.

The latency caused by a change in image size was recorded by setting the bandwidth at a fixed value of 5Mbps and then calculating the latency for each object detection calculation. As can be seen from Figure 29, the latency increases with an increase in image size. Both the time taken to transfer the image to the server and the time taken on the server to run the object detection on the image is increased. With an increased frame resolution, there is more data to be transferred and therefore the latency increases. As was discussed in the previous section the time taken for YOLO to perform object detection increases with image size and this is the case using the larger model server side as well as the small model in local processing. The time to perform the object detection server side is shown in Figure 30. Taking a standard image size of 416 x 416 the object detection only runs at 16.7fps server side. While the GPU that is used in testing this application is high-end and very powerful, according to [35] the researchers who created YOLO use a much more powerful GPU and achieve 67fps at the same resolution. This needs to be taken into account when assessing these results.

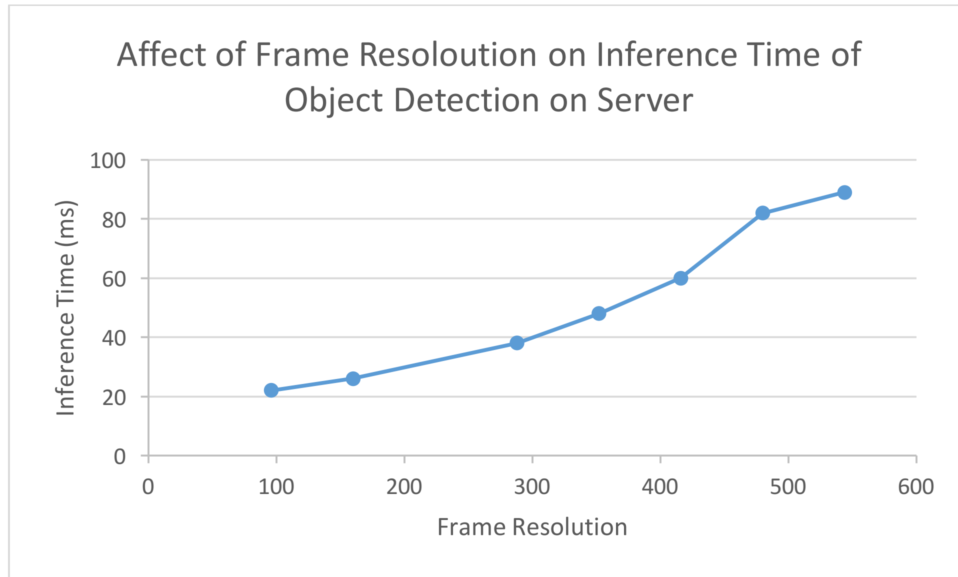

Figure 30 - The inference time of object detection using the more powerful version of YOLO on the backend server.

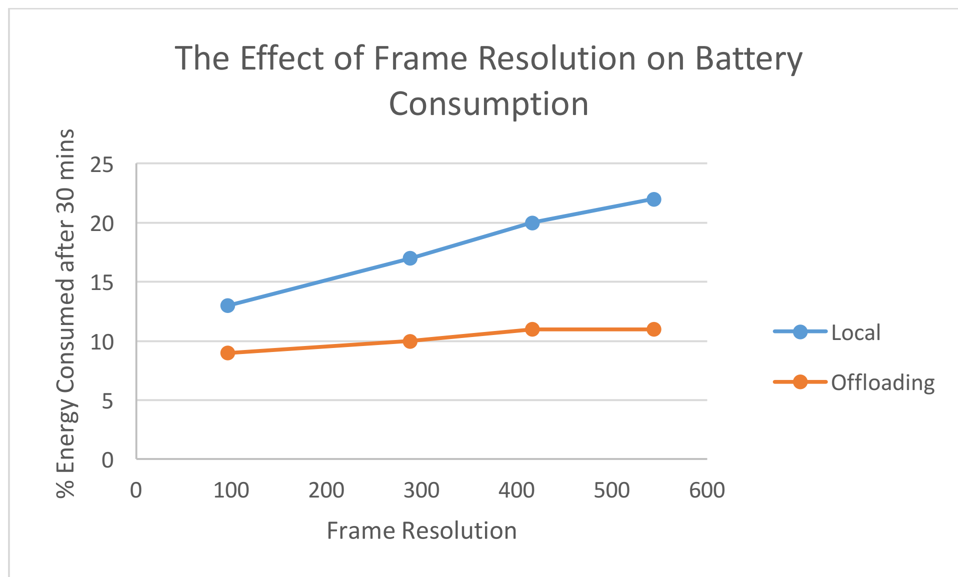

Figure 31 - The effect that changes in frame resolution have on the amount of battery consumed by the application after 30 minutes.

The effect of frame resolution on battery consumption when offloading was examined and plotted against the experiments on local processing in Figure 31. For this experiment, the bandwidth was set at 5Mbs for each calculation. As can be seen from this chart the battery consumption is still quite high, however it is much better than performing the deep learning

computations locally. Taking for example a standard image size of 416 x 416, offloading, results in a 45% decrease in battery consumption over the course of 30 minutes. However, according to [22], an increased bandwidth means an increase in battery consumption for the application. This is due to the fact that an increased bandwidth means more data is being transferred to the server and more energy is consumed due to the cost of transferring the data. Therefore, both the bandwidth and the frame resolution need to be taken into account when assessing energy consumption in an offloading decision model.

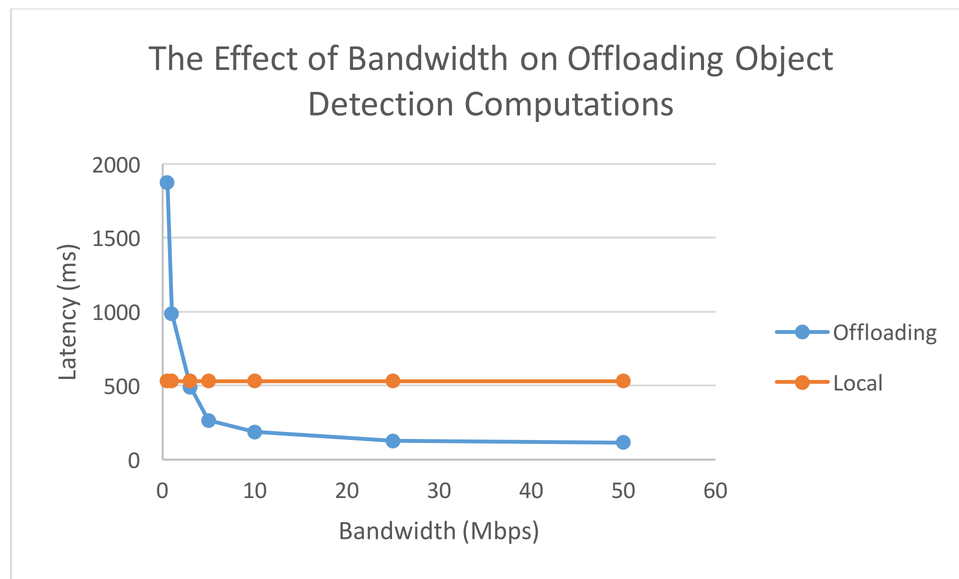

Figure 32 - The effect of various bandwidth speeds on the latency of object detection in the mobile application.

The latency caused by varying bandwidth was tested by setting the bandwidth to a fixed value using the Linux tc traffic control tool. A number of frames were offloaded to the server and the time taken for the object detection results to be returned was recorded and a running average is calculated. In Figure 32, we plot the affect that varying bandwidth has on an image of resolution 352 x 352. The results show that an increase in the bandwidth can dramatically decrease the latency caused by offloading. Taking for example a moderate bandwidth of 10Mbps can give a 35.4% decrease in latency. However, when networking conditions are poor and we only have a bandwidth of 0.5Mbps, this falls much lower than local processing and makes the application practically unusable as the responsiveness is decreased so dramatically.

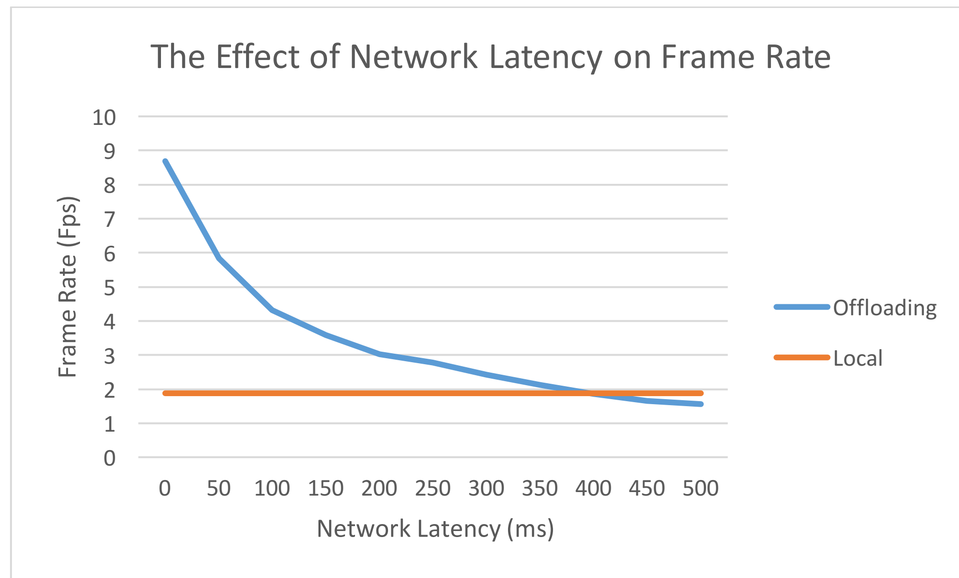

Figure 33 - The effect of network latency on the frame rate of the application.

The frame rate of the application was tested under varying network latencies which were added to the network through the use of the tc traffic control tool. The image size was set as 352 x 352 for this experiment. As you can see in Figure 33, as expected there is an inversely proportional relationship between latency and the frame rate of the application. This added network latency can be caused by a number of reasons such as traffic congestion and queuing due to server load.

Based on these results a simple offloading model can be formed using the latency and bandwidth. Both should be considered in the future when making a more intelligent model but as an example, based on the results in Figure 32, a condition could be set where the application switches to local processing when the bandwidth is below 3Mbps. Using the results shown in Figure 33, another condition could be set where the application switches to local processing when the network latency increases to 400ms. There is a trade-off however of these conditions with both accuracy and energy consumption. As discussed above and shown in Figure 31, switching to local processing causes a huge increase in energy consumption. There is also a large decrease in accuracy when switching to local processing. The accuracy of the Tiny YOLO model run locally vs the larger YOLO v2 model run at server side is given in [35]. As discussed previously the model run on the mobile device consists of 9 convolutional layers in its network

architecture while the larger model server side has 24 convolutional layers. This leads to a 34.5% increase in accuracy as YOLO v2 produces a mAP of 76.4 when tested on the Pascal VOC dataset [53] and Tiny YOLO produces a mAP of 57.1 when run on the same dataset.

Resource and time constraints did not allow for the server to be hosted on a public IP and be tested in real world conditions. This could be included in future work.

6.3.3 Proposed Framework

6.3.3.1 Tracking

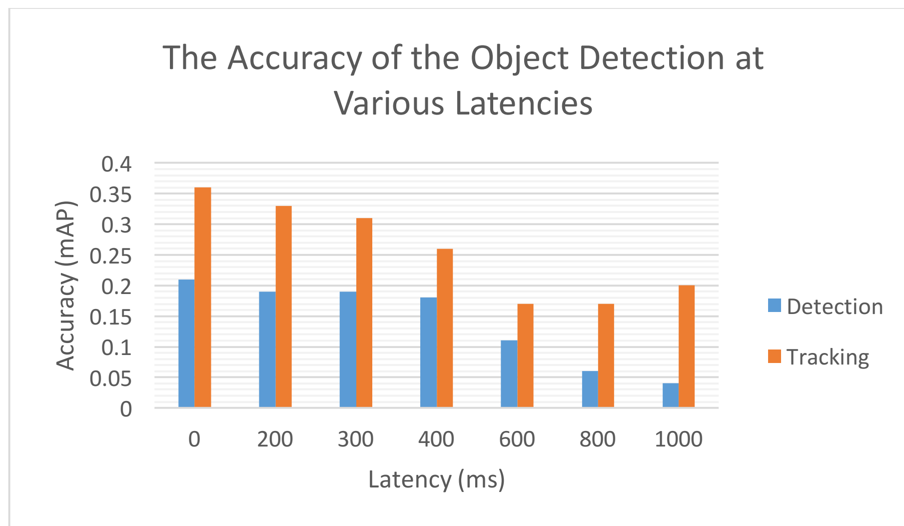

Figure 34 - The accuracy of object detection run locally on the mobile device with varying amounts of delay. The accuracy of using just the object detection model is plotted against using both the object detection model and tracking.

The time taken on average to compute the optical flow between frames as shown in Table 3, was 20ms. Upon updating the optical flow the currently tracked items are calculated in new positions based on this calculation and the number of tracked items can vary the time taken to perform this tracking functionality. This accuracy of tracking was recorded as the mean average precision and was tested against multiple ground truth videos containing a moderate amount of movement. Varying amounts of delay were simulated using the test environment described in

Section 6.2 and the resulting accuracy measurements are shown in Figure 34. On average, excluding the outlier of the accuracy at a 1 second delay which gives a 400% increase, the accuracy is increased by 81% when combining the object detection with the tracking algorithm. Even when there is no delay the accuracy is increased by 71%, this is due to the tracker keeping track of objects when, in new frames the detection model misses previously detected objects. The data shows an increase in accuracy at the 800ms mark, this is unexpected and may be due the variation in the ground truth data and limited test data. This is to be investigated further in future work.

Component	Latency
Tracking	~20ms
Tracking Decision	~8ms
Offloading Decision	<1ms

Table 3 - The latency of various features in the proposed framework.

6.3.3.2 Tracking Decision

The tracking decision adds around 8ms to the application every time it is called. It does however, reduce the latency between frames to around 30ms which is roughly the time taken to make this decision, draw results and perform tracking between the two frames. Both camera movement and new objects in the frame successfully cause the application to perform object detection. The frame differencing function picks up on very small amounts of movement between frames and this can be adjusted for by changing the threshold. It was observed that a threshold of around 10% change in the scene worked well. One solution to the fact that this function is very sensitive would be taking a running average of the difference of a number of preceding frames to give a more accurate depiction of the amount of change in the scene. This method will be further examined and implemented in future work. Another solution which could be explored in future work and could make this calculation more accurate is to utilise the mobile devices accelerometer and gyroscope sensors to calculate the movement of the camera.

The cropping of one section of the frame reduces the image size by quarter when it is passed into the object detection model and therefore, reduces the latency. Take for example an image of size 416 reducing the image size to 208 x 208 can decrease latency from 718ms to 194ms locally and therefore increase frame rate from 1.39fps to 5.05fps. This technique however only works when the camera is static as movement of the camera causes the frame differencing function to pick up movement in each of section of the frame.

6.3.3.3 Offloading Decision

The offloading decision works well with minimal effect on the overall speed requirements and succeeds in changing between local processing and offloading and also changing the image resolution based on the provided network context. As mentioned before in Section 4.2.2, this decision model only focuses on speed however and was simply used to test the mechanism for swapping between local and offloading and for changing frame resolution based on the network context.

6.3.4 Discussion

Based on these results it is clear to see the major improvements in speed, accuracy and energy consumption by offloading the deep learning calculations to the cloud. It is clear that the most important aspect of a mobile cloud computing solution is the offloading decision model. It is vitally important for the application to be contextually aware of networking conditions as they have a huge impact on results. The trade-off between speed, accuracy and energy consumption must be considered when forming an offloading decision model. These results also show some of the different parameters which need to be considered when creating a decision-making model. These parameters include, bandwidth, network type, image size and network latency.

Overall the proposed framework and solutions to masking these impairments worked well. The tracking works extremely well, masking the decrease in accuracy for objects. The tracking decision works very well and reduces the latency to around 30ms per frame when only tracking

is used. The frame differencing algorithm works well but could be made more accurate in future work by recording a running average of the difference between a number of preceding frames. In terms of latency the offloading decision model works well and caps the amount of latency that can be introduced based on the user's preferences by reducing the frame resolution and falling back on local processing when networking conditions are poor. A very basic model was created due to time constraints. The decision model can be improved considerably in future work by taking into account additional parameters collected by the framework and also the results as detailed above.

7 Conclusions

7.1 Summary

This dissertation evaluated the implementation of a framework for running object detection through deep learning on resource constrained mobile devices. This was completed through the implementation of an android application and a backend RESTful API. The mobile application can be used to choose between running deep learning computations locally with a reduced deep learning model or offloading these computations to the cloud. The parameters that affect the speed and accuracy of the detections and the energy consumption produced by the application were tested. In local processing, it was found that the size of the image can increase latency, increase energy consumption and lower accuracy. When offloading, the parameters that were tested were the effect of bandwidth, network latency and image size on the energy consumption and latency in the mobile device. It was shown that increased bandwidth and decreased network latency causes decreased latency between each detection and therefore increases the frame rate. However, there is a trade-off between latency and energy consumption as increased bandwidth and decreased latency causes more energy to be consumed. All of these parameters and trade-offs need to be considered when creating a model which decides upon whether to offload or perform the deep learning computations locally. A number of solutions were proposed to tackle the effects of network impairments and the resource constraints of local processing. These include tracking and an offloading decision model. The tracking worked extremely well and boasted an increase of 81% on accuracy and relying on this tracking when there are no new objects in the scene allow for real-time results. The offloading decision model works as intended. However, it is quite basic and will be improved in future work.

In terms of the objectives set at the beginning of this dissertation, the goal of creating a framework for investigating deep learning through computational offloading was achieved to a high level. This was then extended to tackle the goal of masking the affect that either networking conditions or the restricted resources of the mobile device have on both accuracy and latency. This goal again was satisfied with the use of tracking however there are some improvements to be made which are discussed in Section 6.3.3, and this objective can be further investigated in

future work. The framework facilitated the objective of extensively testing the effects of networking conditions on offloading. This objective was achieved to a reasonably high level. However, this testing could be extended in future work testing a much larger range of parameters which could affect networking conditions. The final objective that was set at the beginning of this dissertation was to form an intelligent offloading decision model based on these results. This objective was not completely achieved due to time constraints. A very simple decision model based solely on tackling latency was proposed. However, this model was simply used to test the framework and is very basic. The results detailed in this research however, will form the basis to allow this problem to be tackled in future work.

The hope is that frameworks such as the one proposed in this dissertation will enable real-time augmented reality systems that can utilize deep learning techniques to make them capable of much more intelligent applications. Resources on mobile devices are getting better, however they will at least for the foreseeable future be constrained due to weight, size and cost of resources such as memory, battery and processors [54]. This fact coupled with the fact that network connectivity is constantly improving and with the release of 5G mobile networks in the coming years [55] means that mobile cloud computing will become an even more viable solution to this problem.

7.2 Future Work

There are a number of improvements that could be made to this framework and further research that can be completed based on the results of this dissertation.

The testing interface could be improved upon in future work by converting the testing application which calculates the mean average precision of the results into a RESTful API. When the mobile application is in test mode the results could be sent to the API and both the accuracy could be calculated and the logs could be stored. This would make testing much easier and would improve the efficiency of the current system. Having to take the log files, extract the results needed, transfer them to a computer to then run the python program is quite time

consuming. Having results available on the device instantly following the testing would make the application much more user friendly when completing further research.

For the reasons described in Section 4.2.5, RTP could be utilized to stream the camera feed on the mobile device to the server. This was not implemented due to time and resource constraints. However, it could potentially improve the frame rate of the proposed framework. This does however come with added issues due to loss and a decrease in the image quality when offloading to the server. Some other potential solutions which could potentially speed up the framework should also be examined such as utilising the mobile device's GPU and the use of parallel processing between the mobile device and the server.

More extensive testing is to be completed in future work to evaluate the framework under a wider variety of networking scenarios. The evaluation completed during the course of this research was restricted to a private network focusing on the use case of only one device connected to the server. This can be further extended by simulating scenarios such as packet loss, server load and traffic congestion. The framework is also to be evaluated under real-world scenarios, testing different networks outside of the computer lab.

A new offloading decision model is to be created based on the research results. This decision model should take the users requirements when it comes to accuracy, energy consumption and latency and use them to make an optimal offloading strategy. This offloading decision model could potentially be created using machine learning techniques with the gathered metrics in the system to create a predictive model to achieve the optimal offloading strategy.

References

[1] K. Herrera, “What Is Pokemon Go And Why Is It Such A Big Deal?,” 2016. [Online]. Available: https://www.cinemablend.com/games/1533430/what-is-pokemon-go-and-why-is-it-such-a-big-deal. [Accessed 4 May 2018].

[2] A. Pardes, “Ikea's New App Flaunts What You'll Love Most About AR,” 9 August 2017. [Online]. Available: https://www.wired.com/story/ikea-place-ar-kit-augmented-reality/. [Accessed 4 May 2018].

[3] Y. Taigman, M. Yang, M. Ranzato and L. Wolf, “DeepFace: Closing the Gap to Human-Level Performance in Face Verification,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014.

[4] M. Irving, “Horus wearable helps the blind navigate, remember faces and read books,” 31 October 2016. [Online]. Available: https://newatlas.com/horus-wearable-blind-assistant/46173/. [Accessed 2018 May 7].

[5] H. T. Dinh, C. Lee, D. Niyato and P. Wang, “A survey of mobile cloud computing: architecture, applications, and approaches,” *Wireless Communications and Mobile Computing,* vol. 13, no. 18, pp. 1587-1611, December 2013.

[6] M. Mekni and A. Lemiux, “Augmented Reality: Applications, Challenges and Future Trends,” 2014.

[7] P. Milgram, H. Takemura, A. Utsumi and F. Kishino, “Augmented reality: A class of displays on the reality-virtuality continuum,” *Proceedings of SPIE - The International Society for Optical Engineering,* January 1994.

[8] I. E. Sutherland, “A Head-Mounted Three-Dimensional Display,” in *AFIPS Conference Proceedings* , 1968.

[9] D. Coldewey, “Google Glass to launch this year for under $1,500,” 22 February 2013. [Online]. Available: https://www.nbcnews.com/technology/gadgetbox/google-glass-launch-year-under-1-500-1C8503747. [Accessed 2018 May 1].

[10] J. Robert, “What is HoloLens? Microsoft’s holographic headset explained,” 30 March 2016. [Online]. Available: http://www.trustedreviews.com/opinion/hololens-release-date-news-and-price-2922378. [Accessed 2018 May 1].

[11] T. Caudell and D. Mizell, “Augmented reality: an application of heads-up display technology to manual manufacturing processes,” in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Kauai, HI, 1992.

[12] H. Kato and M. Billinghurst, “Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System,” in *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*, 1999.

[13] Siltanen, “Theory and applications of marker-based augmented reality,” 2012.

[14] Amazon, “Amazon EC2,” [Online]. Available: https://aws.amazon.com/ec2/. [Accessed 7 May 2018].

[15] Heroku, [Online]. Available: https://devcenter.heroku.com/. [Accessed 7 May 2018].

[16] Salesforce, [Online]. Available: https://www.salesforce.com/saas/. [Accessed 7 May 2018].

[17] A. Goyal and S. Dadizadeh, “A Survey on Cloud Computing,” 2009.

[18] M. Satyanarayanan, “Fundamental challenges in mobile computing,” in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, Philadelphia, 1996.

[19] A. u. R. Kahn, M. Othman and S. A. Madani, “A Survey of Mobile Cloud Computing Application Models,” *IEEE Communications Survey & Tutorials,* vol. 16, no. 1, pp. 393-143, 8 July 2013.

[20] E. Cuervo, A. Balasumramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, “MAUI: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, 2010.

[21] A. Rudenko, P. Reiher and G. J. Popek, “Saving portable computer battery power through remote process execution,” *ACM SIGMOBILE Mobile Computing and Communications Review,* vol. 2, no. 1, pp. 19-26, January 1998.

[22] X. Ran, H. Chen, H. Liu and J. Chen, “Delivering Deep Learning to Mobile Devices via Offloading,” in *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, 2017.

[23] K. Kumar and Y.-H. Lu, “Cloud computing for mobile users: Can offloading computation save energy?,” *IEEE Computer,* vol. 43, pp. 51-56, April 2010.

[24] L. Goel and V. Jain, “A Review on Security Issues and Challenges of Mobile Cloud Computing and Preventive Measures,” in *International Conference on Advances in Computer Engineering and Applications ICACEA*, 2014.

[25] K. Akherfi, M. Gerndt and H. Harroud, “Mobile cloud computing for computation offloading: Issues and challenges,” *Applied Computing and Informatics,* vol. 14, no. 1, pp. 1-16, January 2018.

[26] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu and M. S. Lew, “Deep learning for visual understanding: A review,” *Neurocomputing,* vol. 187, pp. 27-48, 26 April 2016.

[27] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Kholsa, M. Bernstein, A. C. Berg and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV),* vol. 115, no. 3, pp. 211-252, December 2015.

[28] M. Everingham, L. Van Gool, C. K. Williams, J. Winn and A. Zisserman, “The Pascal Visual Object Classes (VOC) Challenge,” *International Journal of Computer Vision,* vol. 88, no. 2, pp. 303-338, June 2010.

[29] S. McCann, “It’s a bird… it’s a plane… it… depends on your classifier’s threshold,” 1 September 2011. [Online]. Available: https://sanchom.wordpress.com/2011/09/01/precision-recall/. [Accessed 2 May 2018].

[30] J. Schmidhuber, “Deep Learning in Neural Networks: An Overview,” *Neural Networks,* vol. 61, 1 January 2015.

[31] AstroML, “Neural Network Diagram,” [Online]. Available: http://www.astroml.org/book_figures/appendix/fig_neural_network.html. [Accessed 14 April 2018].

[32] K. O'Shea and R. Nash, “An Introduction to Convolutional Neural Networks,” *arXiv,* 26 November 2015.

[33] R. Girshick, J. Donahue, T. Darrell and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* pp. 580-587, 2014.

[34] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[35] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[36] K. Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, Dublin: John Wiley & Sons Ltd., 2014.

[37] E. Rosten and T. Drummond, “Machine Learning for high-speed corner detection,” *ECCV'06 Proceedings of the 9th European conference on Computer Vision,* vol. Part I, pp. 430-443, 2006.

[38] C. Harris and M. Stephens, “A combined corner and edge detector,” *Conference: Alvey vision conference,* vol. Manchester, January 1998.

[39] D. G. Lowe, *D.G. International Journal of Computer Vision,* vol. 60, no. 2, pp. 91-100, 2004.

[40] B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” *Proceedings of the 7th international joint conference on Artificial intelligence (IJCAI),* vol. 2, pp. 674-679, 1981.

[41] R. Rojas, “Lucas-Kanade in a Nutshell,” 2009. [Online]. Available: http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/Lucas-Kanade2.pdf. [Accessed 1 May 2018].

[42] T. Chen, L. Ravindranath, S. Deng, P. Bahl and H. Balakrishnan, “Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices,” in *Conference: the 13th ACM Conference*, 2015.

[43] M. Abdelaty and A. Mokhtar, “A Computational Offloading Framework for Object Detection in Mobile Devices,” in *International Conference on Advances in Intelligent Systems and Computing*, 2017.

[44] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobsen, “RTP: A Transport Protocol for Real-Time Applications,” RFC Editor, 2003.

[45] S. Kumar and R. Sonam, “Survey on Transport Layer Protocols: TCP & UDP,” *International Journal of Computer Applications,* vol. 46, no. 7, May 2012.

[46] X. Ma and J. Gao, “The Comparison And Analysis Of The Streaming Media Transport Protocol In The Transmission System,” in *International Conference on Education Technology and Computer (ICETC2012)*, 2012.

[47] Tensorflow, [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android. [Accessed 8 May 2018].

[48] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Josefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” 9 November 2015. [Online]. Available: https://www.tensorflow.org/.

[49] K. Tran, “Evaluation of Deep Learning Toolkits,” 2016. [Online]. Available: https://github.com/zer0n/deepframeworks. [Accessed 10 April 2018].

[50] J. Alammar, “Supercharging Android Apps With TensorFlow (Google's Open Source Machine Learning Library),” 2016 January 2016. [Online]. Available: https://jalammar.github.io/Supercharging-android-apps-using-tensorflow/. [Accessed 8 May 2018].

[51] J. Redmon, “YOLO: Real-Time Object Detection,” [Online]. Available: https://pjreddie.com/darknet/yolov2/. [Accessed 8 May 2018].

[52] M. Garon, “mean_average_precision,” 2018. [Online]. Available: https://github.com/MathGaron/mean_average_precision. [Accessed 4 April 2018].

[53] M. Everingham and J. Winn, “The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Development Kit,” 18 May 2012. [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/devkit_doc.pdf.

[54] K. Kumari, “Challenging Issues and Limitations of Mobile Computing,” *COMPUSOFT, An international journal of advanced computer technology,* vol. 3, no. 2, February 2014.

[55] J. Kennedy, “Ireland to be first country in Europe to roll out 5G geographically,” 20 December 2016. [Online]. Available: https://www.siliconrepublic.com/comms/ireland-5g. [Accessed 5 May 2018].