Imperial College London

INDIVIDUAL PROJECT REPORT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

MEng Individual Project
Mobile Medical Image Analysis

Author:
Isaac Hutt

Supervisor:
Dr Bernhard Kainz
Second Marker:
Professor Daniel Rueckert

June 15, 2020

Submitted in partial fulfillment of the requirements for the MEng Computing Degree of Imperial College London

Abstract

Over the last decade we have watched the rise of the cheap and portable smartphone as the world's computer, simultaneously seeing breakneck advances in computer vision capabilities. Despite these trends, current proliferation of machine imaging techniques into the mobile market largely consists of camera filters. This project takes recent medical image analysis models and enables their use on an Android device.

The result of the project is a multipurpose Android app that can hook itself over other apps, such as those used by mobile ultrasound scanners. This app can be used to perform tasks such as classification on the output of other apps, while providing a modular design to easily allow future medical or non medical models to be added.

An extensive qualitative and quantitative evaluation is carried out, commenting on the project's usefulness and usability. Potential future work is then discussed with wide ranging implications in both medical and non medical areas.

Acknowledgements

I would like to thank my supervisor Dr Bernhard Kainz for his support, as well as his help tailoring the initial project idea to my skill-set. Special thanks to my second marker Professor Daniel Rueckert for his feedback on my ideas. Also to Jeremy Tan and Sam Budd for supporting my work and helping explain the details behind some of the papers which I relied on.

Finally I would like to thank my family for their support throughout my education, especially with the unfortunate global health situation over the last few months.

Contents

1 Introduction **7**
1.1 Context and Motivation 7
1.2 Related Work 8
1.3 Objectives and Contributions 9

2 Background **11**
2.1 Android Platform 11
2.1.1 Services 11
2.1.2 Screen Capture 12
2.1.3 Screen Overlay 12
2.2 Convolutional Neural Networks 13
2.2.1 Convolutional Layers 13
2.2.2 Pooling Layers 14
2.2.3 Fully Connected Layers 14
2.2.4 VGG16 14
2.2.5 ImageNet and MobileNets 15
2.2.6 Theano and Lasagne 15
2.2.7 Data Augmentation 15
2.2.8 Artistic Style Transfer 16
2.3 TensorFlow 16
2.3.1 TensorFlow 1.x 16
2.3.2 TF 2.x and Keras 17
2.4 TFLite 17
2.4.1 Android Interpreter 17
2.5 Network Translation 18
2.5.1 ONNX 18
2.5.2 Manual Translation 18
2.6 Network Optimization 19
2.6.1 Pruning 19
2.6.2 Quantization 19
2.7 Ultrasound Analysis 20
2.7.1 Ultrasound 20
2.7.2 SonoNet 20
2.7.3 Detection 21
2.7.4 Localisation 22
2.7.5 Results 22
2.8 Ultrasound Biometrics Collection 22
2.8.1 Segmentation 23
2.8.2 Calculation 23
2.8.3 Results 23

3 Mobile Optimization of Models **25**
3.1 Methodology and Tools 25
3.2 SonoNet Conversion 25
3.2.1 Recreating the Model 26
3.2.2 Data 26
3.2.3 Training 27
3.2.4 Pruning 28
3.2.5 TFLite Conversion 28
3.2.6 Localisation 30

3.3 Biometrics Conversion . . . 31
3.3.1 Data . . . 31
3.3.2 Training . . . 32
3.3.3 TFLite Conversion . . . 32
3.3.4 Calculation . . . 33

4 The Analysis App 34
4.1 Methodology and Tools . . . 34
4.2 Android Foundations . . . 34
4.2.1 Screen Capture . . . 34
4.2.2 Screen Overlay . . . 35
4.2.3 Processing Pipeline . . . 36
4.2.4 View-finding . . . 36
4.2.5 Overlaying on the Input . . . 36
4.3 Model Implementation . . . 39
4.3.1 Image Classification . . . 39
4.3.2 Object Detection . . . 39
4.3.3 SonoNet . . . 40
4.3.4 Biometrics . . . 40
4.3.5 GPU Delegate . . . 41
4.3.6 Style Transfer . . . 42
4.4 The Final App . . . 42

5 Evaluation 43
5.1 Model Optimization . . . 43
5.1.1 SonoNet TF 2.X . . . 43
5.1.2 SonoNet TFLite . . . 47
5.1.3 Biometrics TF 2.x . . . 47
5.1.4 Biometrics TFLite . . . 48
5.1.5 Pruning . . . 48
5.2 The Android Wrapper . . . 50
5.2.1 Software Compatibility . . . 50
5.2.2 Device Selection . . . 50
5.2.3 Pipeline Performance . . . 51
5.2.4 On Screen Overlay . . . 51
5.3 The Final App . . . 52
5.3.1 Device Specific Performance . . . 53
5.3.2 APK Structure and Size . . . 54
5.3.3 User Feedback . . . 56
5.3.4 SonoNet Specific Feedback . . . 58

6 Conclusion 60
6.1 Summary . . . 60
6.2 Future Work . . . 60

References 62

Appendices 68

A User Guide 68

B App Screenshots 70

C Additional Tables 75

D Model Listings 76

List of Figures

2.1 Activity and Service Lifecycles [31][25] . . . 12
2.2 Facebook Messenger Chat Head on top of the YouTube App . . . 13
2.3 Twilight Blue Light Filter . . . 13
2.4 The VGG16 Architecture [34] . . . 14
2.5 ImageNet Model Performance Evaluation [36] . . . 15
2.6 A Style Transfer Model Architecture [12] . . . 16
2.7 The SonoNet Architecture [3] . . . 21
2.8 SonoNet Localisation [3] . . . 21
2.9 SonoNet Evaluation [3] . . . 22
2.10 Biometrics FCN Architecture [57] . . . 23
2.11 Red Ground Truth and Yellow Prediction Ellipsis for Biometrics [57] . . . 23
2.12 Biometrics Results [57] . . . 24

3.1 SonoNet Loss and Accuracy Trained on Incorrect Data . . . 27
3.2 SonoNet Loss and Accuracy Trained on Correct Data . . . 27
3.3 SonoNet Pruning Accuracy, Sparsity and and Threshold . . . 29
3.4 SonoNet 32 Quantization Results . . . 31
3.5 Segmentation Loss and Accuracy . . . 32
3.6 SonoNet 32 Quantization Results . . . 32

4.1 Average Pixel Component Information Loss by Overlay Alpha . . . 38
4.2 Real Time Overlay FPS Limit . . . 39

5.1 Old and New SonoNet 32 Stats . . . 45
5.2 Old and New SonoNet 32 Confusion Matrices . . . 46
5.3 SonoNet 32 Desktop Inference FPS . . . 47
5.4 Biometrics Comparison. Value (Standard Deviation) . . . 48
5.5 HC Bland Altman Plots (Original and TF 2.x) . . . 49
5.6 BPD Bland Altman Plots (Original and TF 2.x) . . . 49
5.7 TF 2.X Pruning Effects . . . 49
5.8 Android Evaluation Devices . . . 51
5.9 Android Frame Rates . . . 52
5.10 On Screen Overlay Flame Graph . . . 53
5.11 TFLite FPS by Thread Count . . . 54
5.12 SonoNet TFLite Android Frame Rates . . . 55
5.13 Segmentation TFLite Android Frame Rates . . . 55
5.14 Segmentation TFLite Android Frame Rates (No Overlay) . . . 55
5.15 APK Components . . . 56
5.16 Mobile ultrasound scanners combined with intelligent software have the potential to improve patient experience 59
5.17 The ability to capture additional images or video beyond standard planes is important to me. . . . 59
5.18 An app like this could be helpful in a training setting. . . . 59

B.1 MEng Project Play Store Listing . . . 70
B.2 Model Selection in Full and Generic Flavours . . . 71
B.3 System Permission Dialogs . . . 71
B.4 Object Detection . . . 72
B.5 Viewfinding over an Image of an ultrasound scan . . . 72
B.6 Brain Segmentation and Metric Calculation . . . 73
B.7 View-finding over A Real Ultrasound App . . . 73
B.8 Style Transfer Template and Examples . . . 74

C.1 SonoNet Integer Quantization Results . . . 75

D.1 A Simple Sequential Implementation of SonoNet 32 in Keras . . . 76
D.2 A Non Sequential Keras Model for Segmentation in Head Biometrics . . . 77

1. Introduction

1.1 Context and Motivation

As smartphones have become more ubiquitous in modern life, the hardware and software underpinning them have been advancing at record pace. Technologies such as Google Lens, which have the capability to classify objects in images from the camera, have been around in some form or another for a number of years. Amazon's X-Ray feature can detect which actors are present on screen during a movie or TV Show. Most of these tools did, until recently rely on a cloud server to perform the actual processing, with the mobile device simply sending the images and receiving the classification back. Today modern devices have the capability to do this processing locally and offline.

Apps such as Snapchat and Instagram are already making use of this modern technology with "filters" that allow manipulation of the camera feed in realtime beyond what would have been possible just a few years ago. While the tech behind this is undoubtedly cool, it is currently limited to a direct feed from the camera. What if we could enable augmentation directly over any app currently on the screen of a phone? What if we could use this tech to do something slightly more impactful than taking a fun selfie?

Despite fetal developmental issues arising during pregnancy being one of the main risks of perinatal mortality, the detection rate of abnormalities is still low and varies dramatically by region [35]. 2D ultrasound provides a safe and quick means of screening. Fortunately, as well as improvements in smartphones, other technologies have also been evolving. Recently released mobile compatible ultrasound scanners have the capability to improve front-line access to ultrasound across the world.

Ultrasound is not without its issues. High amounts of noise and frequent image artefacts, combined with poor positioning can make differentiating features a challenge. Guiding an ultrasound reader to capture specific fetal features and correctly interpreting them is a job that can require years of experience. The work done on a network named SonoNet [3], a convolutional neural network (CNN), has attempted to mitigate these challenges by detecting a set number of standard fetal views in a live video feed, helping sonographers with their scans.

This is just one of a variety of cutting edge medical imaging networks that target a specific area of medicine. Applying machine learning to medical imaging is a booming speciality and researchers around the globe are working on tackling everything from cancer screening to cardiovascular abnormalities [33].

Currently, SonoNet and most of these other game changing models, require a high powered desktop graphics card to evaluate their input stream (ultrasound data for

SonoNet). Transmitting the video feed to the computer in a reasonable time-frame is often not feasible. In the last few years, cheap Android devices have become far more prevalent in developing nations. These areas are also more likely to suffer from a poor network connection and therefore have the potential to benefit most from using SonoNet, offline and on a mobile device.

This project will focus on bringing CNNs like SonoNet to Android. It will create an application to allow CNNs that perform detection and localisation operations to operate in real-time on the screen output of currently open Android applications. The whole process will happen offline and entirely on-device. This allows analysis to take place almost anywhere and without a network connection. For ultrasound images, a scan could be captured in a remote location and the app would automatically save key images from the scan. These could be uploaded at a later time once an internet connection was available, for review by an expert.

This project will be implemented by running a background service on Android and making use of screen recording and overlay functionality provided by modern Android versions. TFLite will be used for image processing. Optimizations that can be made to networks will be researched, with the purpose of getting them as close to real time performance as is feasible given reasonable accuracy constraints.

A successful project will in the general case provide an app and examples of converted CNNs that can be run over Android apps. In the specific case, it will provide a useful tool that, when used in conjunction with an Android ultrasound probe, will aid the capturing of key information from the fetal scan.

1.2 Related Work

When looking for related work on this project, attempts to find examples of apps that performed realtime processing of a screen recording or any other work done in this area were fruitless. Technology has only very recently advanced to the point where it has been feasible to do so, both from a software and hardware perspective. Despite this, there has been a considerable amount of work on running neural networks on mobile devices that has been outlined below.

There have been a variety of papers focused on optimising CNNs for mobile devices. However, as TFLite is a relatively new framework the majority of papers deal with conversion before this Google backed approach was available. One such proposed method was to convert a CNN in the Caffe [4] framework to C. This could then be run natively on iOS/Android using a tool such as the Android NDK (Native Development Kit) and using the JNI (Java Native Interface) to coordinate with a UI written in Java [59]. This approach does not allow the network to run on a GPU, leading to lackluster performance.

In two other papers, researchers propose (non TFLite) CNN frameworks that can

use Android GPUs [47] [38]. Again, these papers were published before TFLite was announced in 2017. Little work has been done in this area since then as Google controls the direction of the Android ecosystem. This allows them to promote hardware and software features which make their own approach considerably more efficient.

The paper "AI Benchmark: Running Deep Neural Networks on Android Smartphones" [41] evaluates various mobile frameworks and the devices on which they run. It was published before TFLite had matured to the extent it has now and therefore it advises to use TF Mobile (a precursor to TFLite). The devices tested are also a few years older than might be found today, but overall it provides some information regarding the variations in speed between different Android devices running CNNs, which would be useful when planning this project's evaluation.

While we were unable to find any papers dealing specifically with the optimization of a CNN using TFLite, there has been some very recent work that has used TFLite in a scientific context for image analysis [58] [10], some of which was in a medical context. However, TFLite is still maturing, and in a few years there will be many more resources in this area.

With regards to the specific area of mobile ultrasound analysis, there has been one publication which aimed to do a similar task to SonoNet on the Kidneys [60]. The paper describes an attempt to detect and localise abnormalities. There was no effort to apply this in real-time or on videos and took 2.3 seconds to process an image with resolution of 678 x 542. This stunted performance can be explained by the use of a considerably older device and framework.

The background section will cover the two papers whose models this project optimizes in more detail. They will also be mentioned here due to their significance. The optimization section of the project is based on the methods and models that "SonoNet: Real-Time Detection and Localisation of Fetal Standard Scan Planes in Freehand Ultrasound" [3] (referred to as SonoNet) and "Human-level Performance On Automatic Head Biometrics In Fetal Ultrasound Using Fully Convolutional Neural Networks" [57] (referred to as Biometrics) set out.

1.3 Objectives and Contributions

The original milestone objectives laid out for this project and set in early discussions were to:

1. Build out a simple Android pipeline to allow future models to run over other apps.
2. Get an off-the-shelf CNN such as image classification working with the pipeline.
3. Research and experiment with a process to convert a simple model to run on Android.

4. Successfully convert the SonoNet model to run over other apps on Android.

These objectives were met and exceeded fairly quickly, as this report will show. After almost nine months of work, I am very pleased with what has been accomplished. The key contributions, to be laid out in the body of this report are as follows:

- An Android app that allows models to run on-top of the current screen and output a classification.
- A mechanism for overlaying data on-top of the screen while losing minimal information, allowing realtime automated annotation.
- A description and evaluation of recent techniques for optimally porting models to Android.
- An implementation of two medical models, SonoNet and Biometrics on Android.

These achievements combine to form an Android app that has great potential, both in medical and other areas of realtime image analysis.

2. Background

Mobile image analysis requires both insight into mobile specific paradigms combined with knowledge of modern machine inference techniques and how they translate to a mobile setting. In addition to presenting information on these topics, we will briefly cover the context of the ultrasound use case.

2.1 Android Platform

The Android operating system is the leading mobile operating system by installs, with almost 90% of the smartphones running Android [39]. As well as smartphones, Android runs on devices ranging from tablets to cars and is prevalent across the globe.

Since the explosive growth of Android as an operating system, the ecosystem of Android devices has also grown. Flagship devices now feature multiple powerful cameras and it is possible for additional peripherals such as infrared and ultrasound sensors to be plugged in and used with a downloaded app [53].

Android as an operating system is still maturing and one of the more recent changes has been a tightening of security with regard to how much system access apps have. Apps have always been sand-boxed but they have had access to system API calls to perform actions beyond their scope. Recently, Google have sought to crack down on a lot of these permissions, removing them all together or requiring explicit user approval, in the form of a system popup [14].

2.1.1 Services

One of the key features of Android is the way that it structures its applications. Applications spawn activities which are the main windows that users interact with. These activities are paused when the user switches between apps and they might even be kicked out of memory. Activities are not allowed to run in the background [23].

To allow applications to do background work, a Service [25] must be used. This is a process that runs in the background, but it can still be killed by the OS. In order to reduce the odds of being killed, an app can place a permanent notification in the notification bar and become a Foreground Service. Some examples of Activity and Service lifecycles are included in Figure 2.1.

Services (and Activities) are initiated with an Intent object. Intent objects are messaging objects which describe which Activity/Service is started when passed as a parameter to startActivity()/startService() and their variants, as well as carrying additional serialized data [17].

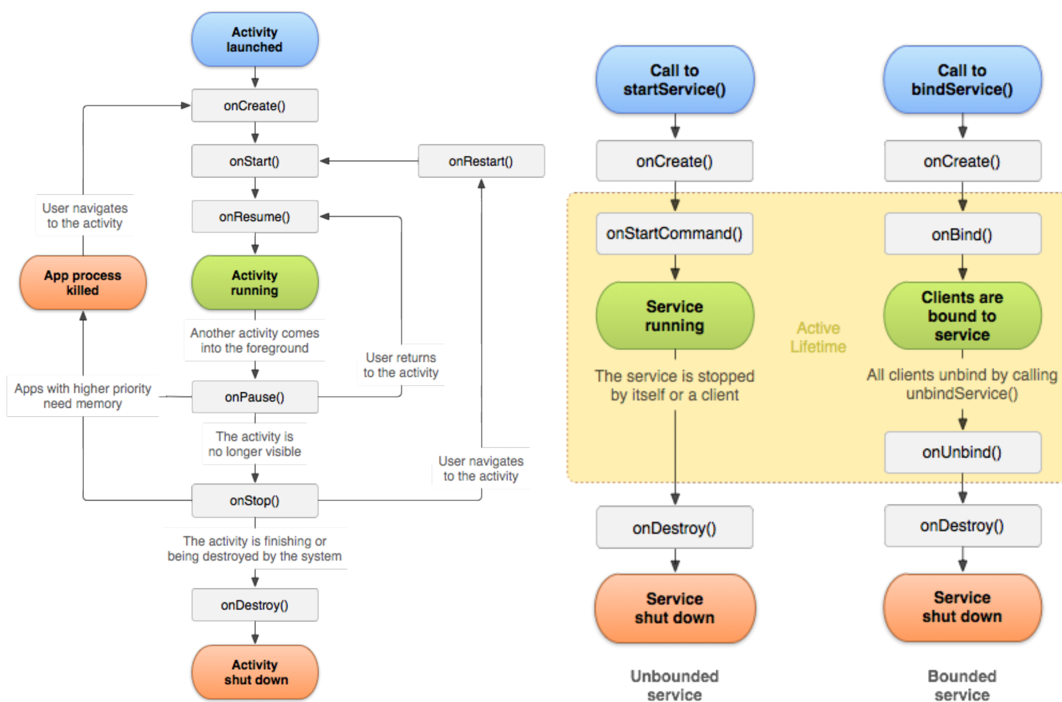


Figure 2.1: Activity and Service Lifecycles [31][25]

2.1.2 Screen Capture

Screen capture on Android must be handled by calling a string of system APIs. This ensures that the user has explicitly authorized the screen recording. The key system API is the *MEDIA_PROJECTION_SERVICE*, which can be used to get a callback for a screenshot. This screenshot can then be converted to a bitmap before being processed in the callback.

There are a variety of parameters that can be used to customize the process. Everything from the resolution and pixel format of the captured image to the target frame capture rate can be specified. Screen capture should also be possible on iOS [2] but it is considerably less documented.

2.1.3 Screen Overlay

Android allows applications to draw over the current activity. The most common use of this is Facebook Messenger Chat Heads (Figure 2.2). These allow users to talk to their friends without leaving the app that they are in. This is an example of an interactive overlay, as the user can interact with it as if it were an Android Activity. Another common overlay is Twilight, which is a simple overlay designed to remove a screens blue light at night by overlaying a translucent red mask on top (Figure 2.3).

In order for a service to draw on top of the current activity it must use specific API calls. Android has been getting more and more protective of these over the years. This is because there is the potential of using these to trick a user into accidentally

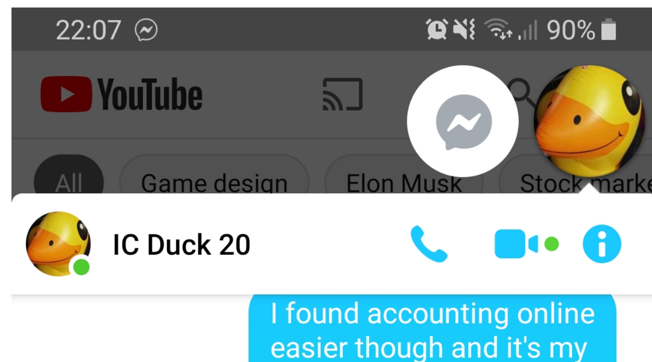


Figure 2.2: Facebook Messenger Chat Head on top of the YouTube App

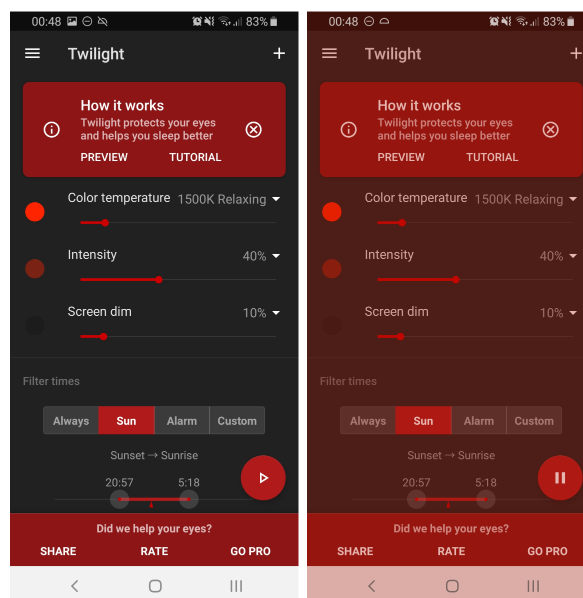


Figure 2.3: Twilight Blue Light Filter

divulging a banking password while they think that they are typing into their bank's app. Despite this, there is still a surprising amount of freedom available to developers once the user has granted permission. On iOS this would simply not be possible without jailbreaking the device due to the increased sand-boxing of iOS apps [40].

2.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of deep neural network used to process images. CNNs use multiple different types of neuron layers to classify images [45].

2.2.1 Convolutional Layers

Convolutional layers perform a convolution on images. They have a few parameters, the main parameter being the kernel size. This specifies the area that the convolu-

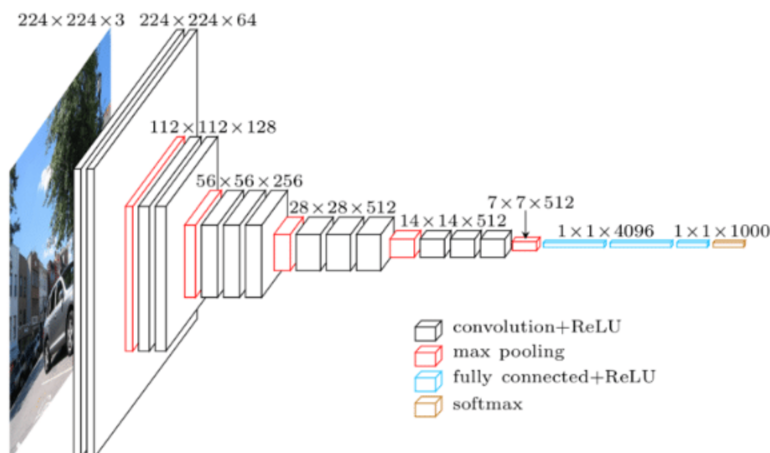


Figure 2.4: The VGG16 Architecture [34]

tion operates over and allows for features to be learnt and then built up. Another important parameter is the number of input channels such as red, green and blue.

An example of a classical convolution might be a Gaussian kernel (used to blur images), which sets a pixel's value to the sum of n layers of surrounding pixels weighted by their distance and plugged into a Gaussian function. This kernel can be represented as a matrix of multiplying factors. When these are centered over the pixel in question and summed, the desired output will be obtained. In machine learning these multiplying factors are parameters to be learned.

2.2.2 Pooling Layers

Pooling layers are used to reduce the dimensionality of the data by grouping multiple neurons into a single neuron. This is useful to reduce the number of parameters that will eventually need to be learnt in the network, while ensuring that specific features are learnt. One technique to perform pooling is called Max Pooling. This gives the maximum value of a group of neurons as the output.

2.2.3 Fully Connected Layers

A fully connected layer connects each output neuron with every input neuron. An example of this might be a final classification layer, determining the likelihood of an image being of a particular class by summing the previous neurons weighted according to learned parameters.

2.2.4 VGG16

A popular example of a CNN is VGG16 (Figure 2.4). VGG16 derives its name from VGG, which was the paper's authors' team name in the 2014 ImageNet challenge [54] and 16 which is the sum of the 13 convolutional layers and the 3 fully connected layers in the network [56]. As can be seen in the figure, the network com-

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Figure 2.5: ImageNet Model Performance Evaluation [36]

prises groups of convolutional layers with 3x3 kernels and an increasing number of kernels inter-spaced between a total of 5 max-pooling layers.

After these convolutional layers, there are 3 fully connected layers. The last of which has 1000 channels corresponding to the 1000 classes in the ImageNet challenge. The final layer is a softmax layer, which outputs the final class probabilities.

2.2.5 ImageNet and MobileNets

ImageNet is a collection that groups images that represent the words in a structured hierarchy collated by concept [8]. Google's Mobile approach to solving ImageNet using "MobileNets" makes use of "Depthwise Separable Convolutions". These are 8-9 times more computationally efficient than standard convolutions, while only resulting in a small loss of accuracy (Figure 2.5) [36]. This performance speedup is impressive, but such radical changes to model architecture were out of the scope of this project, which focused more on direct porting and making use of other less radical approaches to model optimization.

2.2.6 Theano and Lasagne

Theano is a Python library primarily for evaluating mathematical expressions, efficiently making use of GPU hardware if available [7]. Theano has not seen a release since the end of 2017 due to the rise of popular competing frameworks such as TensorFlow. Lasagne is a machine learning library designed to train neural networks using Theano [46]. It supports CNNs via layers that allow specification of parameters such as kernel size. Support for Lasagne has petered out in line with the depreciation of Theano. This can make it difficult to run older models due to Python dependency updates.

2.2.7 Data Augmentation

Often one of the limiting factors on the accuracy of a machine learning model is the amount of data available to train with. If the pool of training data is too small, then the model is likely to fit to features that discriminate adequately in the training data, but inadequately during validation. In order to reduce the likelihood of the model over-fitting, additional data can be generated from existing data via various image operations such as rotation, flipping or blurring. By adding images with (for example) different rotations, the model is less likely to fit on features that might only

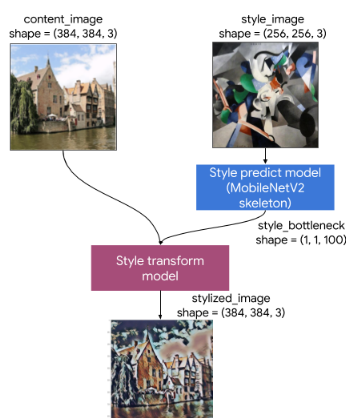


Figure 2.6: A Style Transfer Model Architecture [12]

be vertical in training data. This allows the model to also distinguish those features at varied angles [18].

2.2.8 Artistic Style Transfer

Artistic Style Transfer involves generating an image (called a pastiche) based on two input images. One image representing the desired style and the other the desired content [12]. In the method proposed by Google (Figure 2.6), the content image is passed into a model along with the output of a second model. This second model is a dedicated style prediction model that takes the image representing the outputs desired style as input. A two model approach speeds up the inference time as a large portion of processing only needs to be done once and not per image.

2.3 TensorFlow

TensorFlow is a machine learning system that operates at large scale and in heterogeneous environments. TensorFlow uses data-flow graphs to represent its computations, shared states and state change operations. When run on a single device these graphs can be mapped across multiple computation devices such as CPUs, GPUs and other specifically designed machine inference hardware know as TPUs (Tensor Processing Units) [1].

Often a higher level API will be used with TensorFlow to increase ease of use such as Keras [44] allowing the user to write simple lines of code to perform common operations.

2.3.1 TensorFlow 1.x

In TensorFlow 1.x a user first must build a computational graph then create a session to execute it. This two step process can be rather jarring and forms a divide between a model and its input/output. Additional many useful functions in TF 1.x were in

external python packages. For model data flow TF 1.x requires the use of placeholders, dummy variables that the session will later use to interact with the model, but which are included in the computational graph [43].

2.3.2 TF 2.x and Keras

TensorFlow 2.x was a major improvement on the usability and performance of TensorFlow 1.x. TF 2.x incorporates a high level Keras API that allows models to be easily built and saved in just a few lines, while retaining the options that a more custom approach provides. This is supported by Eager Execution mode, which removes the need for a user created session. TF 2.x also brings a lot of external packages into the main package and greatly simplifies the data pipeline allowing more natural IO. Finally (and most importantly for this project) TF 2.x supports the SavedModel and Keras model file formats, allowing one line TF 2.x model conversion to TFLite [15].

2.4 TFLite

TFLite (TensorFlow Lite) is an "open source deep learning framework for on-device inference". TFLite consists of a set of tools to run TensorFlow models on mobile, embedded and IoT devices. In this project it allows compact, low latency, offline, machine learning inference on Android [27].

There are two main components to TFLite. The interpreter and the converter. These convert TensorFlow models to their TFLite form and then allow interpretation of the new optimized models. The advantages of offline inference with a tool such as TFLite are multi-fold, mainly centering around improved latency, privacy and accessibility in areas with limited network connectivity.

TFLite also features, cross platform compatibility, multiple API languages, the possibility of hardware acceleration on supported devices and a small and portable model format called FlatBuffer. A typical TFLite workflow would involve converting a TensorFlow model to a TFLite model, deploying it and then attempting to optimize its on device performance.

There are however a few catches. Firstly TFLite only supports a subset of TensorFlow operators which have already been optimized. Some additional operators can be added to the TFLite binary, but there might be some issues with the more experimental among them. TFLite also does not currently support on-device training, but that is unlikely to be an issue for this project.

2.4.1 Android Interpreter

The TFLite Android interpreter is designed to be lean and fast. It uses static graph ordering and its own memory allocator to reduce latency across the board. The TFLite Android interpreter supports both Java/Kotlin calls as well as C++ via the

JNI. Input and Output is handled by Native ByteBuffers.

One of TFLite Android's key advantages is its support for hardware acceleration of models. On Android there are three main possible execution delegates. First the CPU, on which multiple threads can be used to make use of modern multi-core architectures. Then there is the GPU and finally the NNAPI (Neural Networks API) which allocates tasks to specialized device hardware such as the GPU, DSP (Digital Signal Processor) or the NPU (Neural Processing Unit) if such devices are available. Out of these three delegates the GPU normally performs best if supported by the model being run, as it is able to run operations with a greater degree of parallelism than the CPU and the NNAPI only really adds overhead on devices that do not have specialised DSP or NPU hardware.

While TFLite is at the cutting edge of mobile inference it is far from complete. Many features that would greatly improve performance or compatibility are not yet implemented. features such as Sparse Model Execution and GPU execution of integer quantized models are still on the road-map [28].

2.5 Network Translation

Network translation is the process of translating from one Neural Network format to another, either manually or programmatically, possibly via a number of intermediary networks. There are a variety of frameworks out there [48] but in order to end up with a standard TFLite model for use with Android we will first need to convert the given model to be of that form.

2.5.1 ONNX

ONNX (Open Neural Network eXchange) is an open format designed to represent machine learning models. It defines common operators and a common file format [52]. ONNX was developed by Microsoft and Facebook to "create [an] open ecosystem for AI model interoperability" [49].

When converting from one framework to another ONNX can be used as a bridge between ML libraries and their underlying computational graphs. Unfortunately ONNX does not support Theano/Lasagne [61] as these frameworks are now considered to be legacy. This means ONNX can not be used for this project despite potentially being highly useful.

2.5.2 Manual Translation

Manual translation is the process of converting a model from one framework to another by hand. Each layer in the old model must be created in the new model. Once this has been done a user can either retrain the model using the original data (easier but slow to train), or attempt to translate the old weights (can be tricky

especially with older frameworks). While manual translation can be slow it also allows for more modern framework features to be used rather than performing a direct translation; this could result in improved performance.

2.6 Network Optimization

In order to increase the frame rate at which neural networks run it may be necessary to perform optimisations on them with the aim of reducing the network size without noticeably effecting accuracy. There are a number of optimisations that can be made with wide ranging effects. As TFLite is a light weight framework simply translating a network to it has the potential to reduce raw network size and improve performance.

2.6.1 Pruning

Pruning is a process to reduce the size of a network. It can also be used to reduce over-fitting in a network. One simple approach to prune is to remove all connections with weights below that of a specific threshold and then retrain on the new sparse network. One such approach was able to reduce the parameters in AlexNet and VGG16 by 9x and 13x respectively [32].

A second pruning approach is one which attempts to include the size of the model in the optimization function for the CNNs weights [50]. The approached outlined takes a trained network and then attempts to work out the importance of neurons during training using the squared change in loss when removing a particular neuron to drop the unimportant ones. This approach produced similar results on the networks that it was applied to.

2.6.2 Quantization

Another standard optimisations is quantization. Quantization is the process of reducing the size of floating point values in the network (ideally down to 8 bit integers) and is one of the few optimisations that is specifically mentioned on the TFLite site. Quantization can be performed in TFLite relatively easily [22] and there are multiple degrees of quantization that can be performed.

The most basic level of quantization is to limit the model weights (originally float32s) to float16 types. This shrinks the network size in half and still allows GPU acceleration of the model. If GPU acceleration is not required than the network parameters can be quantized to integers. This should make the network about 25% smaller than the original and also give a 2-3x performance speedup.

Finally if a representative data-set is available then both the parameters and activations can be quantized to integers. This results in potentially an even larger speedup, with the bonus that if a model can be converted to exclusively utilize integers then it can also be run on hardware devices such as Edge TPUs very efficiently.

As quantization by TFLite is conventionally done post training there is the potential for some accuracy loss. Users must be aware of implications and potentially re-evaluate the new quantized model. If the accuracy loss is deemed to be too high then there is the potential to instead use "Quantization Aware Training".

Quantization Aware Training emulates the quantization that would be there during inference and can (for most simple models) be performed by simply passing the original model into a TensorFlow library function before compiling and training. It does however result in less flexibility in the generated model as it will never be able to regain the precision that would have been in a non quantized, trained model. Quantization Aware Training is well built out in TensorFlow so it is even possible to define custom quantization functions [24], however this would require substantial knowledge of the framework beyond that of a standard Keras model.

2.7 Ultrasound Analysis

Both the SonoNet and Biometrics model are designed to operate on fetal ultrasound images. This section provides some context to their respective use cases and explains how these models work in detail.

2.7.1 Ultrasound

Ultrasound scans (also called sonograms) are procedures that use high-frequency sound waves to capture images of inside of the body. In an ultrasound scan a probe emits sound waves (inaudible to humans) and detects their return as they bounce off the object being imaged [51]. During pregnancy, an ultrasound scan is normally carried out between weeks 18 and 22 of gestation. Images are taken of a variety of standard scan planes; these planes are then used to calculate biometric measurements such as head circumference. The measurements can be used to date the fetus and evaluate the potential for abnormalities [9].

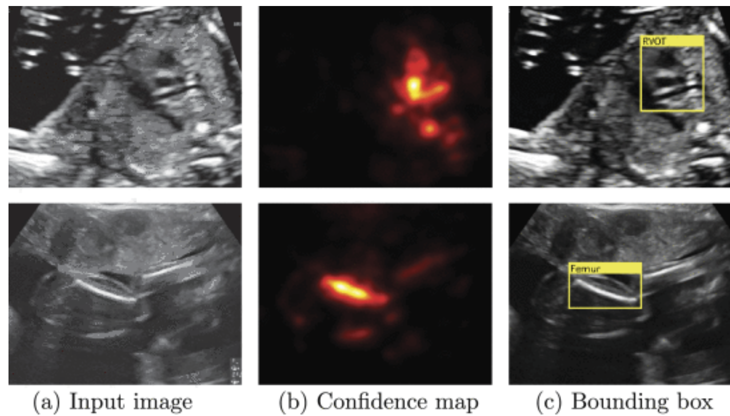
Recently mobile ultrasound scanners such as Philips Lumify [53] have allowed for app based ultrasound scans that allow patients to be met at the point-of-care facilitating faster diagnosis. In the future, healthcare is likely to shift further in this direction as technology in general becomes more mobile.

2.7.2 SonoNet

SonoNet is a CNN architecture (Figure 2.7) built to detect the 13 fetal standard planes required by the UK FASP guidelines for mid pregnancy-ultrasound [9]. The system can be used to analyse specific frames or to retrieve the standard planes from a longer video of a freehand ultrasound scan. Weak supervised learning has also been used to implement localisation. This improves scan accessibility for non-experts. Figure 2.8 shows an example of detection and localisation using SonoNet.

	Feature Extractors	Adaptation Layers	Classification Layer
SonoNet-64	2x[3x3x64/1], MP, 2x[3x3x128/1], MP, 3x[3x3x256/1], MP, 3x[3x3x512/1], MP, 3x[3x3x512/1]	1x[1x1x256/1], 1x[1x1x*K*]	
SonoNet-32	2x[3x3x32/1], MP, 2x[3x3x64/1], MP, 3x[3x3x128/1], MP, 3x[3x3x256/1], MP, 3x[3x3x256/1]	1x[1x1x128/1], 1x[1x1x*K*]	
SonoNet-16	2x[3x3x16/1], MP, 2x[3x3x32/1], MP, 3x[3x3x64/1], MP, 3x[3x3x128/1], MP, 3x[3x3x128/1]	1x[1x1x64/1], 1x[1x1x*K*]	
SmallNet	1x[7x7x32/2], MP, 1x[5x5x64/2], MP, 2x[3x3x128/1]	1x[1x1x64/1], 1x[1x1x*K*]	

Figure 2.7: The SonoNet Architecture [3]



(a) Input image (b) Confidence map (c) Bounding box

Figure 2.8: SonoNet Localisation [3]

The training data for SonoNet consisted of 2694 2D ultrasound exams of volunteers with gestational ages that were between 18 and 22 weeks. These exams were labeled by sonographers according to the UK FASP handbook. The use of a variety of scan styles means that the data set should be representative of the range of possible styles that could be used by sonographers. The labeled data set contains 27731 images of the ”standard” planes and 6856 of other views such as the bladder, cervix etc.. There was not an equal amount of images for each of the planes, ranging from 500-5000. This was due to some of the scans containing multiple images of the same plane while others missed certain planes.

There were also 2638 video recordings of fetal exams, the majority of which were associated with scan image sets. These videos were used as the sole method of evaluation (as they were a more realistic scenario) as well as to bridge a small domain gap between images and video. The data was pre-processed using a series of steps culminating in splitting it into training and test sets. 80% of the data was used for training and 20% was used for testing.

2.7.3 Detection

Various numbers of kernels were used to produce SonoNet, resulting in: SonoNet-16, 32 and 64. All have 13 convolution layers, 2 adaption layers and then a classification layer. SonoNet was trained using mini-batch gradient descent with 20% of the training data used for validation.

TABLE III
CLASSIFICATION SCORES FOR THE FOUR EXAMINED NETWORK ARCHITECTURES

Network	Precision	Recall	F1-score
SonoNet-64	**0.806**	0.860	**0.828**
SonoNet-32	0.772	0.843	0.798
SonoNet-16	0.619	**0.900**	0.720
SmallNet	0.354	0.864	0.461

TABLE IV
FRAME RATES IN FPS FOR THE DETECTION (FORWARD PASS), LOCALISATION (BACKWARD PASS) AND THE TWO COMBINED

Network	Detection	Localisation	Det. & Loc.
SonoNet-64	70.4	21.9	16.7
SonoNet-32	125.4	35.8	27.9
SonoNet-16	196.7	55.9	43.5
SmallNet	574.1	226.0	162.2

Figure 2.9: SonoNet Evaluation [3]

Frames were classified based on a softmax layer producing the confidence for each class for each frame.

2.7.4 Localisation

SonoNet uses weakly supervised localisation to learn how to identify features. In order to generate bounding boxes a confidence map is generated from a saliency map. A saliency map can be computed by determining how much each pixel in an input influences the predicted class. Class score weightings are used to improve the output of the saliency map.

A confidence map is generated by blurring the saliency map with a 5x5 Gaussian kernel. Once the confidence map reaches a threshold, a bounding box can be generated and outputted to the user.

2.7.5 Results

Figure 2.9 shows the classification scores and frame rates that the various SonoNet architectures achieved. From these tables we can see that there are clear diminishing returns for larger network size. Small increases in accuracy result in a hefty frame rate penalty.

2.8 Ultrasound Biometrics Collection

As well as plane classification, work has also been done by researchers (some at Imperial) on calculating key metrics such as measurements of fetal Head Circumference (HC) and Biparietal Diameter (BPD) [57]. These measurements can be automatically calculated given an appropriate input image. Measurements such as these

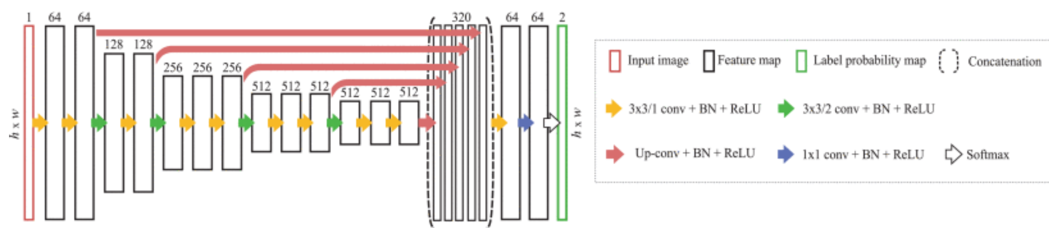


Figure 2.10: Biometrics FCN Architecture [57]

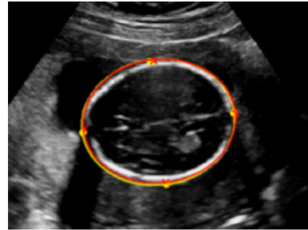


Figure 2.11: Red Ground Truth and Yellow Prediction Ellipsis for Biometrics [57]

(as well as measurements of the abdomen and femur) can be used to estimate fetal properties such as age and weight as well as allowing early diagnosis of abnormalities such as Down's syndrome.

2.8.1 Segmentation

In the Biometrics paper, the standard trans-ventricular (TV) brain view is used as input. A CNN that has been trained on almost 2000 images is then run on the input with the task of segmenting two classes, brain and background. The result is a 2 channel output with one channel being the likelihood of a pixel being part of the brain and the other the likelihood of not being part of the background (Figure 2.10).

2.8.2 Calculation

Once an image has been segmented an ellipsis is fitted [11] over the segmentation (Figure 2.11). From this ellipsis, it is possible to calculate HC and BPD measurements using simple equations. These metrics are captured in pixels so must be multiplied by the dimensions of a pixel in order to obtain the appropriate measurements in millimeters.

2.8.3 Results

In order to effectively evaluate their model, the researchers first compared their model's output with the dataset's measurements that had been annotated by the original sonographers. In addition to this, they also conducted an Intra- and Inter-observer study on 100 of the randomly sampled test images. This gave insight into the variability of human measurement when comparison with that of the model. The results can be seen in Figure 2.12. This table shows Mean Absolute Error (MAE), Mean Error (ME) and Dice coefficients. Two experts were consulted for the study,

Metric	*Intra-expert 1*	*Intra-expert 2*	*Mean Inter-expert*	*Mean Model-expert*	*All test data*
HC MAE (mm)	1.55 (1.30)	1.55 (1.14)	2.16 (1.16)	1.99 (0.87)	1.80 (1.49)
HC ME (mm)	0.18 (2.01)	-0.09 (1.92)	1.56 (1.70)	1.01 (1.62)	0.54 (2.28)
BPD MAE (mm)	0.40 (0.32)	0.60 (0.45)	0.59 (0.34)	0.61 (0.33)	0.68 (0.62)
BPD ME (mm)	-0.05 (0.51)	-0.06 (0.75)	0.01 (0.60)	0.29 (0.55)	0.13 (0.91)
Dice Coefficient	0.983 (0.006)	0.984 (0.005)	0.980 (0.005)	0.980 (0.005)	0.981 (0.007)

Figure 2.12: Biometrics Results [57]

with the Intra expert columns representing the difference between each of the experts' predictions and the original recorded measurements. The mean inter expert columns quantifies the difference between the two experts and the mean model expert column specifies the mean difference between the model and the experts' predictions. The table shows that the difference between the model and the experts is approximately what might be expected from the difference between two experts.

3. Mobile Optimization of Models

This chapter details the work done on porting models to the TFLite format for inference on supported mobile platforms. It also outlines steps taken to optimize these models for performance. TFLite supports interpreters on a variety of platforms, including both Android and iOS. The model conversion and optimization portion of this project is largely device agnostic. In order to show that this project has relevance beyond that of a single model, in this chapter both the SonoNet and Biometrics models are optimized.

3.1 Methodology and Tools

After the initial planning and research phase, the first task to complete was setting up a productive development and evaluation environment that could serve a useful purpose throughout the project timeline. To this end, a GitLab Git repository was setup for source control and it was shared with the project's stakeholders. Furthermore, access was requested to codebases and data for a handful of projects relating to the models that were due to be converted. This was important because research papers often lack some key implementation details and data is not published. In this case, a lot of the ultrasound data was protected under NDA. Slack was also set up, where it was possible to connect with others who had worked on related topics.

After the foundations were in place and experimentation with the existing models had occurred, a Jupyter Notebook was created. This allowed rapid iteration on ideas and easier exploration of data. Using notebooks extensively on Industrial Placement, provided evidence for the positive impact they could make on productivity. A variety of compute resources were utilised during the project. Initially, one of my supervisor's machines was used but later we migrated to a GPU machine in labs. This was because it was under less load and would be less catastrophic if it crashed during the Covid-19 campus closure. */vol/bitbucket* was used to store data and the multitude of virtual environments that needed to be created to allow running of older models.

The main TensorFlow version used was originally a nightly TF build, but this was then upgraded to TF 2.2 when it was released some additional fixes were needed. TensorBoard was used extensively to track model performance, along with external packages on the Python side such as OpenCV, matplotlib and numpy.

3.2 SonoNet Conversion

Having setup my environment the project's first objective was to convert SonoNet to the TFLite format. SonoNet has a variety of configurations that trade accuracy for speed so, when selecting which configuration to use, a balance had to be met. Looking at the results presented in Figure 2.9 SonoNet 32 was the obvious choice

as it had decent performance while maintaining high accuracy. Both of the original SonoNet models used as a springboard for the project (Theano and TF 1.x based) also defaulted to SonoNet 32 and this cemented its selection.

3.2.1 Recreating the Model

Once the specifications of the network to convert had been decided upon, the next step was figuring out a set of steps to convert it to the TensorFlow 2.x format (a necessary step before conversion to TFLite).

Two SonoNet versions were available, one Lasagne/Theano and one TensorFlow 1.x. Right from the start, it became clear that ML libraries had been evolving at a breakneck pace and backward compatibility had clearly been an afterthought.

Various ways to translate the networks had been researched, but in the end it became clear that simply recreating the network in TensorFlow 2.x and retraining would be a much simpler and quicker endeavor than attempting to programmatically convert the model. Too much had been happening too quickly in the field for backwards compatibility in tooling to have been emphasized. The time taken to effectively write a tool to convert between two frameworks would have been enormous and required weeks of research into each frameworks specific design decisions. As this tool would in all likelihood only work for a small subset of the versions of each framework, it would not have been a useful allocation of time.

Fortunately, the SonoNet structure is fairly simple and it was easy to quickly build an equivalent network in TF 2.x (Figure D.1). At this point porting the weights from the original model was again researched. Both models had the same number of parameters and the same structure. However, there was an issue with the interface between Theano and TF. Specifically the weight ordering was not the same when linking a convolutional layer and a dense layer.

Converting the TF 1.x weights was postulated, but this would also have been challenging and as they were not the weights used for the original paper, it would not have provided a benefit in evaluation. In the end it was decided that the best course of action was to retrain the newly constructed model.

3.2.2 Data

In order to retrain the new model, data as close that that used in the original paper was needed. A hdf5 file was available that contained this data, but it was configured in a rather confusing manor. This led to a week of wasted training time as the model was effectively, accidentally being trained on primarily background images. While this did still give the opportunity to build the training and testing pipeline, it was not ideal to have spent valuable time training on the wrong data. The results of this incorrect training can be see in Figure 3.1.

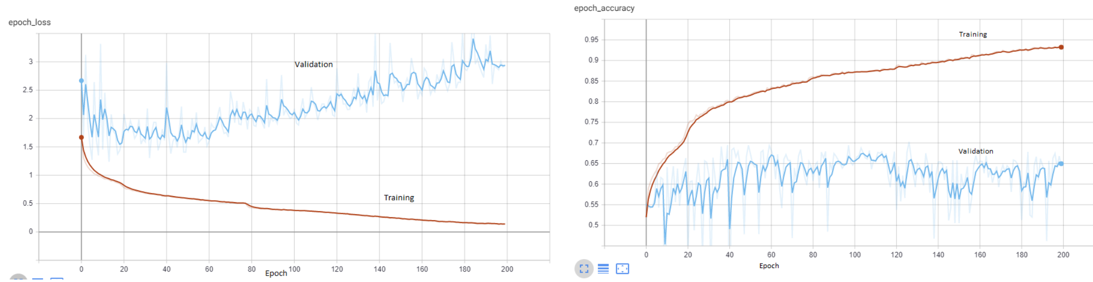


Figure 3.1: SonoNet Loss and Accuracy Trained on Incorrect Data

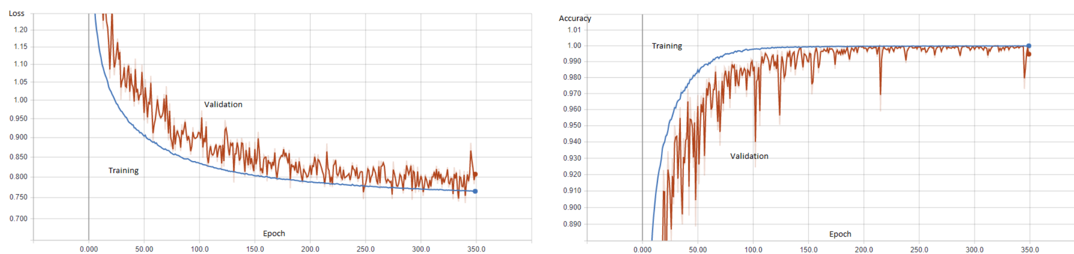


Figure 3.2: SonoNet Loss and Accuracy Trained on Correct Data

Once the data error was corrected, the data augmentation specified by the SonoNet paper was implemented. A Python library named imaug [42] was used to achieve this without having to write large amounts of custom code.

The original SonoNet paper used a large amount of background frames for their testing. However, during this project we were unable to clarify which sections of the dataset were used for this evaluation. As training an identical model was not the pure goal of this project, it was agreed that the discrepancy in the dataset while unfortunate would not have too large an impact. Each class had a fair amount of images and so should have been representative even if there was a significant mismatch between the two datasets. This discrepancy did result in a much lower support for the background class, but the discrimination between planes was still as expected.

3.2.3 Training

Minibatch gradient descent was used for training in the original paper. However, TF 2.x is structured around epochs and so, to replicate this approach a large epoch count was chosen, while still sampling mini-batches with a replacement strategy. This emulated the original paper as much as possible. Using the same minibatch size of the original paper and 576 batches per epoch (epochs are an arbitrary amount of minibatches): 200 epochs (About 24 hours of training) was the optimal train time.

Unfortunately, at this point a memory leak during tensor creation occurred. This was worked around by using an older TF build before migrating to a new release when it became available. At last a decent model was trained and TensorBoard was used to monitor the training. This is shown in Figure 3.2.

3.2.4 Pruning

When looking to increase a model's baseline performance, the most intuitive approach is to reduce the model's complexity and as such the number of calculations required during inference. Doing this without majorly refactoring the model is know as pruning.

TensorFlow helpfully has an entire sub-site dedicated to model optimization [29], but as with so many only resources utilised during this project, it was not up to date with the latest information. After piecing together clues to figure out its inner workings, Batch Normalization during pruned model creation was found to be the issue [13]. Fortunately, this was fixed in TF 2.2.0 release a few days after the issue arose and Google provided useful support on GitHub during this time. The pruning library works by automatically working out which layers it is capable of pruning. There are a number of parameters in the function which allow finer grained control of how exactly you want pruning to occur.

The TensorBoard output of a 14 epoch prune is shown in Figure 3.3. These graphs show accuracy starting strong, but then dropping as pruning kicks in and sparsity increases. Then after the initial drop, even as sparsity continued increasing (till flattening out at 50%) the model was able to regain its accuracy as it continued training. As the model sparsity increased the activation threshold for pruning also increases and this is shown by the third graph.

Despite having implemented pruning, no performance increase was observed. Scouring both the TensorFlow documentation and GitHub, a feature request calling for the inference engine to support performance improvements enabled by the sparsity introduced by pruning was found[26]. This issue and further comments on other issues like it suggest that full support for pruning will not be implemented until at least Q3 2020. As such we decided to cease our work on pruning (except for a short evaluation of the current state) and move on to other potential performance gains.

3.2.5 TFLite Conversion

Once a Keras model was built, it was easy to perform a direct conversation to TFLite. There were often memory issues with conversion due to a limited amount of GPU RAM available, but these could normally be solved with a Python kernel restart. On a few occasions it was necessary to instruct TensorFlow to use the CPU not the GPU as GPU ram was more limited.

Testing the converted TFLite file was possible with 4 lines of code by using the python TFLite interrupter. The results were identical results to the TensorFlow Model.

```
from tensorflow import lite
converter = lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
open("model.tflite", "wb").write(tflite_model)
```

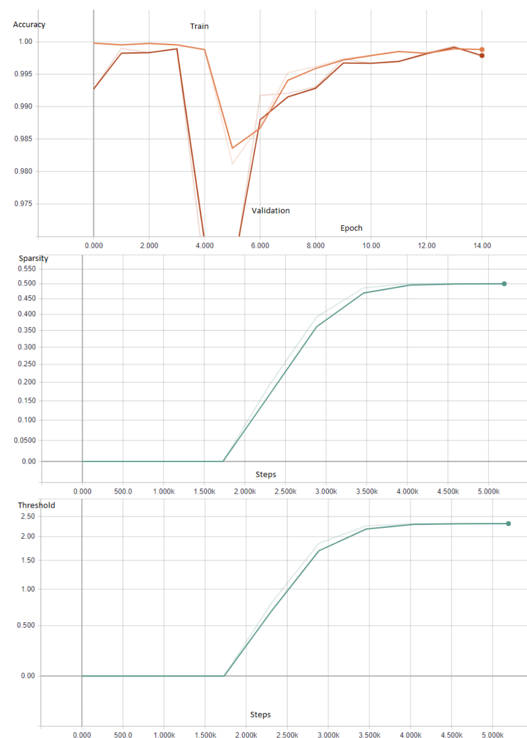


Figure 3.3: SonoNet Pruning Accuracy, Sparsity and and Threshold

There are multiple optimisation parameters that can be used to configure quantization. They are configured by setting properties of the converter object as follows:

```
# Float 16 Quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]

# Dynamic Range Quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Full Integer Quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_gen
```

In order to make measured decisions on the best parameters to optimize the SonoNet model's usability, it was necessary to evaluate all four possible settings. For speed and reproduction reasons the ideal evaluation environment would have been the TFLite Python interpreter running on a desktop machine. However, this threw up several major issues. The first being that the TFLite Python interpreter does not currently support the GPU. This is not mentioned in any documentation, but there are multiple comments online regarding the lack of mainstream support for the non mobile interpreter. Without GPU support, evaluation of runtime improvements would not be fair. Secondly, support for quantization is also limited on the interpreter. Dynamically quantized models ran 6x slower on the Python interpreter and integer quantized models ran an unbelievable 190x slower than a non quantized model. This is due to many operations in TFLite (specifically around quantization) only being optimized for ARM processors using "ARM Advanced SIMD (Single Instruction Multiple Data)", also known as NEON. While some operations have been ported to

x86 correctly, many require serial processing and slow memory access requests.

While unable to effectively use the python interpreter to evaluate speed, it was possible to use it to evaluate the accuracy impact of the quantization parameters. There was still an issue with full integer quantization though. A 160x penalty was simply too high and would have resulted in processing F1 statistics for the test data taking about a week of solid processing. This was simply too long to rely on a machine staying on for so an alternative approach needed to be found.

The full raw dataset was 12GB so it would have been impractical to transfer it to a mobile device. Finding and exporting a specific subset would have also been challenging mapping the mobile output to the original data would have posed a challenge. Instead the dynamic nature of Python Notebook's was used to setup a web server that interactively served the test image pixel values encoded into binary, while reading a GET header that denoted the previous images classification. Combining this with a quick Android loop and connecting a phone to the Imperial VPN and pointing it at the notebook, allowed the phone to classify the images using the integer quantized model in around 2 hours rather than the week that it would have otherwise taken.

Figure 3.4 shows the effects of the various quantization parameters of the model being run on the Android TFLite interpreter of a local Galaxy S8 on native inference time. There are a couple of interesting takeaways from this table. The first is that, while predictions are slightly different, accuracy is not noticeably reduced by quantization in the SonoNet model. A full breakdown and confusion matrix for full integer quantization can be seen in Figure C.1. Given this fact accuracy conscious quantization techniques such as quantization aware training were not pursued further.

Using the table shown in the quoted figure, it was clear that the best choice for quantization was float16 quantization as it still enabled GPU processing while halving model size and maintaining accuracy. In a production application it would make sense to include a full integer quantization model as well as float16 model as if GPU inference was not supported then the additional quantization provided by full integer would result in a significant performance boost while only adding 50% to model size. Both models combined are still only 75% of the file size of the not quantized model.

3.2.6 Localisation

In the SonoNet paper a specific method is discussed that uses saliency maps to localize features in an unsupervised manor. While the details are laid out in the paper, without a working example it proved to be infeasible to replicate in this project. Some preliminary work was done, looking at using feature maps based on pixel activations, but the results were not consistent in the tests that were carried out. Most activated regions often correlated more with negative space than with the fetal feature itself. An algorithm attempting to work out a bounding box might have been

Quantization	Model Size	CPU (ms)		GPU (ms)	Average F1
		1 Thread	8 Thread		
No Quantization	14523KB	839.0	348.4	91.3	0.93
Float16	7277KB	806.8	321.8	82.9	0.93
Dynamic Range	3650KB	650.1	309.4	N/A	0.93
Full Integer	3722KB	547.3	191.2	N/A	0.93

Figure 3.4: SonoNet 32 Quantization Results

possible, although it would have been a slow and plane specific process to implement. In conversations with others working in the area it became clear that more modern research was surpassing the methods originally outlined in the SonoNet paper and so it was not considered to be critical to copy this localisation technique. Instead efforts were focused on the Android side of localisation, building out enabling tools that would not just support localisation but also object detection and segmentation.

3.3 Biometrics Conversion

Once SonoNet had been converted and and work was underway on the Android side of the project it was suggested that the project might also benefit from converting ”Head Biometrics in Fetal Ultrasound” [57] to TFLite. This model was more complicated than the SonoNet model as it was not sequential (some layer outputs were consumed multiple times in the model). The model segments an image of the brain; then an ellipsis can be fitted to calculate measurements using classical methods. While we briefly looked at how a more modern end to end model might work the point of this project was to convert existing models, so we did not go in that direction.

Using a PyTorch implementation of the paper’s model to recreate a TF 2.x version was quickly completed. As the model is not sequential, instead of adding layers to a model object and then compiling the model, instead we initialize the layers which will be used and then define a forward pass function, which takes an input and passes it though. The final step is defining the model by passing it an input layer and the result of the forward pass function applied to that input (Figure D.2).

3.3.1 Data

A hdf5 file containing pixel intensity values was used for training, validation and testing on ultrasound images (all of dimension 384x320 pixels). Also available were ground truth segmentation masks of these images as well as the isotropic (same size vertically as horizontally) pixel sizes in millimeters and ground truth HC and BPD measurements in pixels. The paper implemented data augmentation in the form of flipping horizontally so this was also added to the training environment.

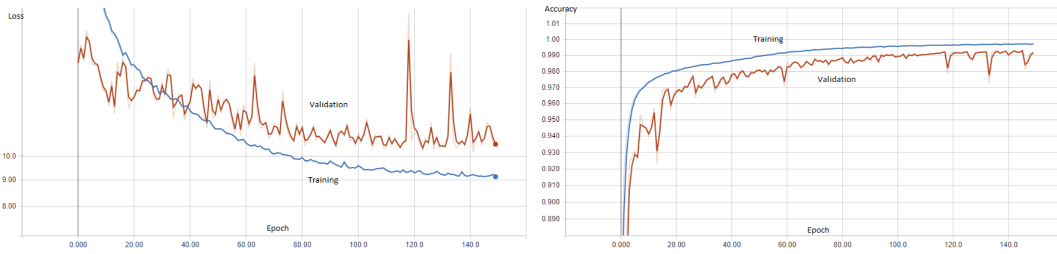


Figure 3.5: Segmentation Loss and Accuracy

Quantization	Model Size	CPU (ms)		GPU (ms)	Average Dice
		1 Thread	8 Thread		
No Quantization	30833KB	17703.5	9935.3	841.0	0.982
Float16	15434KB	17258.0	8420.3	833.1	0.982
Dynamic Range	7748KB	10018.6	3768.0	N/A	0.964
Full Integer	7825KB	7911.2	2534.5	N/A	0.959

Figure 3.6: SonoNet 32 Quantization Results

3.3.2 Training

The original paper used weights from training on ImageNet before training on the ultrasound data. However, this would have added additional complexity to the training process and it was deemed non essential to the project. Instead we directly trained on the data and found 100 epochs (5 and a half hours training) to be a good stopping point, shown in Figure 3.5.

The model was quickly tested on the SonoNet brain TV data to qualitatively inspect the results. The conclusion was that the model seemed to work but was not as accurate and there were much more prevalent segmentation artifacts. This result paved the way to potentially running both SonoNet and Biometrics if an image is classified as Brain-TV by SonoNet.

3.3.3 TFLite Conversion

As with SonoNet, once a working TensorFlow model had been built, converting to TFLite was trivial. Using a similar evaluation technique as described for SonoNet, with the results in Figure 3.6, the dice score remains relatively unchanged throughout the various quantization configurations.

As with SonoNet it is clear that the best course of action is to use float 16 quantization where a GPU is available and full integer quantization where it is not. Packing both into an APK file is still feasible so that was the decided final configuration.

3.3.4 Calculation

After the neural network has segmented the image of the brain, the final step is to calculate the HC and BPD measurements. This is done in the paper by fitting an ellipsis to the segmented image using the least squares method.

Unfortunately the paper was fairly vague on the exact specifics of this approach, so using an external library seemed to be the best solution. The OpenCV Python library was first used to find the segmentation mask contours (the borders of contiguous regions). Then the largest contour was taken (assuming only minor segmentation artifacts this is always going to be the brain) and OpenCV was again used, to fit an ellipses to this contour. This approach ignored disjointed segmentation artifacts and appeared to work well given the data that was available. The code for ellipsis fitting is as follows:

```
z = model.predict(data).argmax(axis=-1)
image, contours = cv2.findContours(z.astype(np.uint8),cv2.RETR_TREE,cv2.
    CHAIN_APPROX_SIMPLE)
c = max(image, key=lambda item: len(item))
(xx, yy), (minor, major), angle = cv2.fitEllipse(c)
```

OpenCV returns the major and minor axis radii of the ellipsis in pixels (a and b), which can then be used along with the isotropic pixel size in millimeters (s_{xy}) to calculate BPD and HC in millimeters using the following formula:

$$BPD = bs_{xy}$$

$$HC = \pi(a+b)(1+\frac{3h}{10+\sqrt{4-3h}})s_{xy}$$

$$\text{where } h = \frac{(a-b)^2}{(a+b)^2}$$

Bringing together the segmentation model and the ellipsis fitting allows an image of the brain to be used to calculate the BPD and HC as detailed in the original paper.

4. The Analysis App

Parallel to the process of converting and optimizing models is the Android app, which allows these models to analyse the output of other applications. This app is the major novel contribution of this project. In order to achieve my objectives, an application that would be able to process the screen and output classifications needed to be built. Furthermore, it was necessary to research techniques to overlay bounding boxes and segmentation colouring without blocking the capture of the original underlying image. The final task was to build a process for extracting captured planes for viewing by an expert.

4.1 Methodology and Tools

Android Studio was chosen for this project as it is the standard IDE for Android. Kotlin was chosen to be the the main Android language of this project for a number of reasons, but mainly due to Google (the maker of TFLite and Android) pushing the language over Java in recent years [6]. In addition to this many of the more recent TFLite examples are written entirely in Kotlin. Despite having very little experience with Kotlin in the past, Kotlin was easy to pickup and a breeze to use, incorporating many of the best aspect of Java, while fixing some flaws.

APKs were built and tested locally on the whole. Only part of the evaluation was performed on cloud devices in order to evaluated broad performance. Due to this a full cloud build pipeline was not needed and deployment was done manually. Testing was considered but beyond a few very basic tests it was not widely performed as only a single developer was working on the App and the codebase was changing too rapidly to make writing tests a productive endeavor.

4.2 Android Foundations

This section deals with the general technical barriers that had to be overcome to enable models to preform as intended. Namely how to capture the screen, how to output results intuitively and how to run models as fast as possible on less powerful processors.

4.2.1 Screen Capture

The first challenge we face when analysing other running apps is obtaining a stream of bitmap images from the devices screen as fast as the system allows. Ideally one image for each updated frame. On Android this can be achieved with a MediaProjection API call, but (reasonably so) special permission are required for this. Every time we wish to record, we must first instruct the system to ask the user if recording is acceptable (with an option to auto allow) and wait for the decision (Figure B.3).

Once permission has been granted, capturing frames at a reasonable pace (greater than a frame or two a second) required creating a new thread. This thread loads and processes images from the image buffer exposed by the MediaProjection API. When creating this buffer, the caller is allowed to specify its size and often on lower end devices the app was liable to be killed for memory if this buffer size was too large. Conversely if it was too small, then it could be depleted, causing an app crash. A buffer size of 3 was the smallest that could reliably work without the app crashing. When using threads there was an issue when a reference to the image capture object was destroyed. The system had garbage collected the image capture object. As the object was gone it stopped notifying the image capture thread that there were new images. This stalled frame capture and processing. Fortunately, simply moving the image capture objects declaration outside of the function was enough to fix this issue.

When images are available other information (including the actual capture timestamp is also obtainable) and this would turn out to be useful. While not attempting an implementation research was done into saving the stream of images as a video or broadcasting it as a live feed. It should be possible to also attach a MediaRecorder object to the same recording surface that is used for image capture. This would allow the contents of the screen to be streamed over the network.

4.2.2 Screen Overlay

The second obstacle to overcome was the ability to output data intuitively and in a timely manor to the user without being in the foreground. One approach to this would have been to update a system notification with information. This would have necessitate the user swiping down from the status bar to see the app's output and would have no possibility for realtime annotation or augmentation. Instead we make use of an innovative Android feature, the screen overlay. Just like screen recording, a special permission is required for an app to be permitted to overlay over other apps. This has to be granted in the settings activity (Figure B.3). The app first makes a system call to check if we have the permission. If we do not, we can direct the user to the exact page in settings where they are able to grant it.

Once it has been granted we have the power to add an arbitrary number of windows to the screen. There are specific window flags which determine whether a window is interactive or not and the order in which windows are added determines which window receives input first. Newer windows have the potential to block older windows receiving this input. This is a very powerful tool which allows everything from opaque interactive UI elements to semi-transparent, passive overlays which blend with the screens content. The base overlay for this app is an opaque box centered at the bottom of the screen which contains buttons, areas for text output and a space for an image output. The majority of the screen is left free from overlay with the vast majority of the overlain app still interactable (Figure B.4).

4.2.3 Processing Pipeline

A basic processing pipeline was the key to link the input and the output portions of the app. This processing needed to happen in a Service so it could run in the background. One challenge to achieving this was the Android permission request structure. In Android, to request permission to view the screen the user must accept a system dialog that will make a callback to a specific Activity method with a specified result code. This callback contains an intent that can then be exchanged for an object allowing screen recording. The issue is this object can not be passed to the service as is not serializable, neither can the callback signal the Service directly as it is not an Activity. Fortunately Services can be started with Intents. This makes it possible to add the callback's Intent to an Intent created to start the Service. This intent can then be passed to the system to obtain the ability to record.

To demonstrate that the pipeline was working a simple ”red counter” model was implemented. This model inefficiently iterated though the pixels on screen and classified the image as ”Red” if over half the pixels had a large red component or ”Not Red” if not. This text was written to the screen overlay along with the FPS rate of classification. While this model was not complicated it formed the base for all future models and demonstrated that the projects aims were possible.

4.2.4 View-finding

While occasionally the user might desire analyse the whole screen, in most cases only a specific portion of the screen would be relevant to the analysis task at hand. In an ultrasound use case a large section of the screen might be UI elements. The actual analysis should only take place on the window dedicated for output.

How could we allow user flexibility in scale and location (aspect ratio is often fixed by the model so we keep it fixed as well) of the input view to be analysed? The approach taken was to add a translucent green box that could be dragged around and resized before being hidden. This box would be used to crop the image obtained from the system and this cropped image would be passed into the model. The user guide (Appendix A) explains in further detail how to manipulate the box.

4.2.5 Overlaying on the Input

The Android app makes use of an opaque intractable overlay for input and output. More complex analysis often involves either identifying and locating specific objects or segmenting regions of an image. While this is provided for in the form of a ”picture in picture” image in the control box, it is a very small mirror of the screen. It would be difficult to read the object classifications in an object detection model as they would be very small. A far better solution would be drawing output directly over the input at the real scale in the real location.

Unfortunately, this approach has an inherent issue. If we draw an overlay on the

screen then it will be captured later when input is being captured for analysis. This would be liable to mess up our processing and produce garbage output. There is a mechanism in Android for disallowing screen capture that is made use of in apps such as mobile banking. It allows a developer to mark a surface as ”SECURE”, blacking it out in screenshots and recordings. This is useless in this context though, as blocking out our overlay would just leave a black box over the input image. No apps currently on the market appear to advertise this capability and we found no solutions to this challenge online.

The first approach towards tackling this would only work if the input was a black and white image. In this case a 50% transparent red box could be drawn over the screen. Pixel intensities could be sampled using the theoretically unaffected blue or green components in order to process the original black and white image. Thinking in terms of information loss there would not be a large amount of information loss when a transparent overlay was drawn as the output pixels depended on both the original image and the overlaid image. We can see that it should be possible to reverse an image overlay.

While not explicitly mentioned in the documentation, by reading up on various stack overflow questions the formula which Android uses to calculate each pixel colour components was discovered. With p as the output pixel, o as the original pixel, n as the overlaid pixel and a as the overlaid alpha (between 0 and 255) then:

$$p = \frac{o(255 - a) + na}{255}$$

In order to work out the original pixel value given the other variables we can rearrange to get:

$$o = \frac{255p - 255n}{255 - a} \text{ or } \frac{p - n}{1 - \frac{a}{255}}$$

We can then work out how much information about the average original pixel is lost for varying levels of overlaid alpha using this information. Given an alpha a for the overlay, then $255 - a$ is the range into which the original pixel must be compressed. This would mean that for a 50% transparent overlay the 255 possible pixel colour component intensity values would need to be compressed down to only 128 values. This implies that 50% of the time a pixel would be rounded to ± 1 of its original value, leading to an average loss of 0.5 per pixel component.

If however an alpha of 253 was used then the original pixel would be compressed into only a range of 2 (either 0 or 255). This means that 253 of the possible inputs would be incorrect. We can form an equation to work out how far on average we would be from the correct pixel depending on the alpha of the overlay being used.

$$d = \left\lceil \frac{1}{2} \left(\frac{256}{256 - a} - 1 \right) \right\rceil \frac{a}{256} \text{ for } 0 <= a <= 255$$

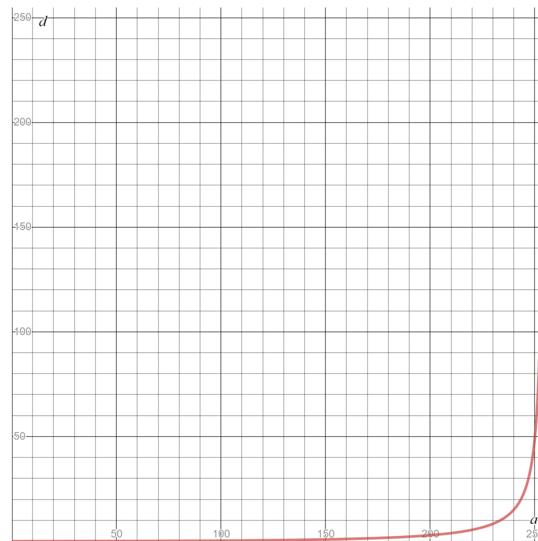


Figure 4.1: Average Pixel Component Information Loss by Overlay Alpha

Plotting a graph of this (Figure 4.1) we can see that the information loss starts really kicks in between an alpha of 220 and 240 and it rapidly spikes. This is good news as it means that even relatively opaque overlays can be used while still maintaining enough of the original images information to reverse engineer its approximate pixel colour components to a decent degree of accuracy.

Once we had shown that restoring the original image should be mathematically possible we needed to implement it in an algorithm. The key to all of this was timing. As multiple threads are involved the current overlay might not be the overlay that was being overlaid in the image currently being processed. There might have been a draw update since then. Attempting to subtract the wrong overlay from an image would not just not work, but would alter the image even more from the original pre-overlaid state. In order to match images with their overlay it was necessary to save a queue of overlay images and the time that they were drawn. Then when processing an image we can work out which overlay image was on screen at the timestamp at which the image was taken. We can also delete all overlays from before that point from the queue and use the selected overlay that was onscreen to obtain the original image. This worked for slower FPS rates but there was an issue with faster models. The time at which an overlay is told to be drawn, is not the time that it is actually drawn. There is a delay.

To solve this we can delay frame processing until the draw update completes by using a Future [16] that would be set itself to the current time via a lambda. This lambda would be posted to the UI thread after instructing the overlay draw and then the futures value would be requested in the image processing thread. This results in a Hard FPS cap (listed for various devices in Figure 4.2). There should be a way to remove this cap by utilising multiple threads but it would get rather fiddly. There were some occasional issues on the S10 and Note10 even with this technique and sometimes the overlay would slip though. This is likely due to the increased FPS of

Device	FPS Limit
Galaxy Note10	6.82
Galaxy S10	8.05
Galaxy Tab S4	1.97
Galaxy S8 *	3.68
Nexus 5X *	1.30

Figure 4.2: Real Time Overlay FPS Limit

these devices leading to an increased likelihood of the race condition triggering.

Once this process was working, in order to achieve performance gains a ByteBuffer was used instead of the array iteration originally taking place. This resulted in a significant speed up but there was still considerable overhead to reversing an overlay. Drawing an overlay is hardware accelerated and can run on the GPU. It might be possible to perform the overlay un-drawing on the GPU as well. We expected it might be possible to approximate the pixel maths using standard draw commands but were ultimately unable to get any of the equations to match.

4.3 Model Implementation

After the basics of the app were built we were able to move onto getting real models working on Android. General models that Google used in TFLite examples were easier to implement so that is where we will start before refocusing on the medical models which we have previously covered.

4.3.1 Image Classification

The first CNN that was hooked into the pipeline was Google's MobileNets for classification of the ImageNet challenge. This model is freely available to download from Android resource pages and there are even helpful demo projects with source code. These show how the model hooks into the output of a devices camera. Using these examples to learn how model interaction worked, it was easy to modify it to instead to use the bitmaps generated from the screen by the processing pipeline implemented earlier. The end result worked perfectly, displaying the classification for the object currently on the screen and validating the concept of running a CNN on the screen contents of a device.

4.3.2 Object Detection

In order to test the work done on drawing over input, an Object Detection model was implemented. Taking a more advanced version of MobileNets (also published by Google) that also incorporated object detection into the mix, the app would not just return a single classification but instead a list of boxes, classifications for the

objects in the boxes, and certainty of each boxes classification. As we also had an example of this model being used with Android it was also fairly easy to configure it to work with this custom pipeline.

The most confident classification was used as the text output for the main box, but in addition to this, a transparent bitmap of the same size as the input was created. Using the android Canvas API to draw semi transparent red boxes with the same coordinates as outputted by the model, it was possible to localize the detected objects. Then, by drawing text on the bitmap beneath the boxes with the box's classification it is possible for us to read the classification of each box. Passing this bitmap as an overlay into the processing pipeline resulted in a complete success.

There was one slight issue with the way that overlapping boxes were drawn. Due to the implementation of overlaps in Android, overlapping boxes caused issues with the formula to obtain the original image, due to the way that the semi transparent boxes were rendered. The PorterDuff.Mode [21] setting of canvas painting in Android describes how images are blended. The default setting also blends alpha values which means that two semi transparent objects become less transparent when they overlap. This throws off the formula to find the original image. By using the "DST_ATOP" mode, we can ensure that the resulting blended alpha is fixed. An example of this model in action can be seen in Figure B.4.

4.3.3 SonoNet

Once these off the shelf models had been hooked into the pipeline the next step was to do the same with the first medical model, SonoNet. SonoNet was different in that it only had one channel input not a three channel input like MobileNets. SonoNet takes a gray-scale image but bitmaps are in the argb (alpha, red, green, blue) format. As gray-scale images should have identical RGB values, it was possible to simply chose to take the red value as the single channel. Previous models had been using the helpful TensorImage class to do most of the buffer manipulation, but it only supported 3 channel input so some of its functions had to be rewritten for 1 channel. After doing this a correct input was able to be passed into TFLite for processing. The output was set to the classification and the model was finally working.

4.3.4 Biometrics

With SonoNet implemented, the next task was implementing a second medical model to show that this project could be more widely useful than a single use case. Taking the converted Biometrics model which had been converted and starting the process of hooking it up to the pipeline. The first steps were very similar to SonoNet in that the input was 1 channel, but for this model it was the output which posed a new challenge.

The Biometrics model outputs two values for each pixel, one representing the confidence it is the brain and one that it is the background. This information needed to somehow be converted into a measurement. By performing similar operations to those done in Python, we can obtain a single channel by taking just one of the two output channels and rounding it to get a 1 or 0. To enable drawing the resulting segmentation a transparent bitmap image was generated and then each of its pixels were set to be either green or not green based on the segmentation. This image could then be passed into the pipeline and drawn on the screen.

The next step was to fit an ellipsis around the segmentation and then calculate major and minor axis lengths. To achieve this OpenCV was used, as had be done in Python. This was challenging though as the OpenCV android library requires a fairly fiddly installation process to allow calling its native API via Java/Kotlin. Once this was done we were able to find the contours of the segmentation map and as with python take the longest contour to represent the brain.

Using OpenCV to fit an ellipse around around this contour allowed the needed radii to be captured. The ellipsis centroid and angle (also provided by OpenCV) were useful to to draw vertical and horizontal lines on the segmentation. This demonstrated the exact measurements being taken, providing confidence to the end user that the metrics were correct (Figure B.6)

As there was no way to know what the isotropic pixel size being displayed would be, it was not possible to directly output HC and BPD measurements. Instead it was best to output the length of the radii as a percentage of the viewfinder box that they were measured in. This would then allow end users to fit the view finder box to their probes output. The user should know the dimensions of the probes output box in millimeters and so this information could be used (given the formulas in the previous chapter) to calculate HC/BPD. A box to allow users to input pixel size so that calculations could be performed by the app might be useful, but finding out how useful this might be in a clinical environment should be researched be fore implementation.

4.3.5 GPU Delegate

While the all intended models were now working, their performance was not yet optimal. To this end the default setting of the pipeline was reconfigured to use a devices GPU if available. Switching to the GPU had massive performance improvements as seen in Figure 3.4. The pipeline was also instructed to use an equivalent number of CPU threads to device cores if running on the GPU was not supported. This also saw a significant performance improvement. Finally reasarch and testing was perfomed on the using the NNAPI delegate rather than directly using the GPU, but according to Google's TFLite benchmarks [20] and supported by the small amount of testing carried out, the NNAPI was not yet more optimal at inference than the GPU.

4.3.6 Style Transfer

While the medical applications of this project's Android app stand on their own feet, it was felt that there was the potential to easily show the potential for future work by implementing a non medical model. As any work would take away time from the main section of the project a few days were scheduled to built something cool that would show of the full capabilities of the app.

Some potential non medical ideas included making a ”treasure hunt” game based on image recognition or possibly some sort of Google Lens style app that was constantly running. In the end a simple style transfer algorithm was chosen as it was readily available online [12] and it was visually appealing. The implementation was largely based off the work previously done on segmentation so did not require a huge amount of alteration of the pipeline. The objective of this tangent was to show that the app could perform a more general task. It is capable of capturing a user selected portion of the (or the whole) screen, running a CNN on it and outputting the results blended with the original image. All while also being able to reverse the overlay on the next frame in order to continue the process without the output in frame. The results were impressive and can be seen in Figure B.8.

4.4 The Final App

Once the SonoNet model was working, there was also a desire to add some key additional features to it. One key objective was to augment the pipeline to automatically tick required images off a checklist as captured. Then once all images had been captured, to put these in an email that could be sent to an expert.

To do this detected classes were tracked and the bitmap corresponding to the highest confidence of that class was stored. A mechanism for users to see the list of captured classes was added. Classes were added to this list once a frame had met a minimum threshold value for that class. Users could then choose to send the images as an email using their favourite email app with a single click of a button. The images would then be added as attachments and the email app opened.

Another additional task was the task of combining the SonoNet and Biometrics models so that the Biometrics model would segment all ”BRAIN-TV” classed planes. Fortunately due to the architecture of the app these two complex models could be cohesively combined with ease. Both models could even use the GPU at the same time so there was no need to relegate one to the CPU.

The non medical portions of the app were also published in an unlisted form to the Google Play Store to allow for easier distribution to potential testers. Google is currently taking a long time to review new apps due to the Covid-19 situation but after a week they approved the app for internal testing.

5. Evaluation

To conduct a holistic evaluation of the project, it was necessary to consider both its individual components in addition to its final manifestation. The evaluation begins with a review of the TFLite conversions in Python. This is followed by an examination of the specifics that comprise the Android wrapper app. Finally, the end to end app is both qualitatively and quantitatively scrutinised.

Unfortunately, due to the Covid-19 outbreak, it was not possible to test the app in a clinical setting. This has resulted in a more limited qualitative evaluation than was originally intended. However, real user feedback from friends and family members has been obtained and a video of the SonoNet app was shown to a group of sonographers to appraise.

5.1 Model Optimization

In this section we will analyse how the converted TF 2.x model recreations (and their derived TFLite models) compare with the models that were evaluated in their original papers. As mentioned in the implementation section, while ideally we would have used the weights used in the original papers, it was not feasible to use them in the time frame of the project. Instead the models were rebuilt in Keras and retrained using as close to the same data as possible. This approach was advantageous as it shows that this process can be done even if the original model weights are not published. Research papers are often lacking useful implementation details that would be very helpful to speed up the optimization process though.

Both models that were converted had the same number of trainable parameters as the template models and the structure of the layers was the same. The only differences might be in the “under the hood” implementation of certain operations, due to the random nature of data selection during training or due to inconsistencies in training data.

5.1.1 SonoNet TF 2.X

We start the evaluation with a comparison between the Theano/Lasagne SonoNet model and this project’s TF 2.x model. The results can be seen in Figure 5.1 and Figure 5.2 (see Figure 2.9 for the paper’s F1 score). At first glance the TF 2.x model seems to perform much better across the board then the original model, with an F1 score of 0.93 vs the paper’s 0.80. This was an issue as such as large difference in accuracy could not be explained by simply a different framework. Both models were of the same architecture. We might at first glance assume that a mistake had been made in calculating these metrics, but closer inspection of the confusion matrices reveals the key difference between the two trained models.

Both models have a very similar shaped confusion matrix with a strong diagonal and almost no erroneous classifications outside of 2 key areas. The first area being the 3VV and RVOT classes. Neither model is able to distinguish between these two views of the heart very well. This makes perfect sense as these are very visually similar views. The second area where the models fall down is the background. If either model miss-classifies, it is most likely to be related to the background class. This also makes sense due to the nature of ultrasound scans. Often background images may still contain components of the standard planes as the probe is moved around.

It is with the background class that the discrepancy in F1 score can be seen. Due to differences in data the background set that the TF 2.x version of SonoNet was trained and tested on is considerably smaller than the original papers background set. The result of this is that the original paper's model is going to include a fair amount more images labeled as background which are very close to those of the other classes. This will have effected both training and testing with the end result being both more background class images classed as standard planes and more standard planes classed as background. This would impact both precision and recall, resulting in a reduced F1 score.

As is elaborated on in the implementation section we can see this discrepancy in the data from the outset. In the end however, it can be seen from the confusion matrix that the TF 2.x model was just as good at classifying the non-background standard planes. If the input data was not representative enough of the background then the result would be an over eagerness to classify a plane and an under-classification of the background. This would ideally not have too large an impact as the class likelihood of a real plane of that class should be greater than that of a background plane so the images selected to save from the scan would still be the correct images. As the model could easily be trained with new data in the future and the statistical impact on the end result was not too broad, more time was not spent looking at this.

In addition to accuracy, we can also compare the frame rates of the three models we can now evaluate, in order to evaluate differences in framework performance. To generate the data for this comparison, a lab machine running a GeForce GTX Titan Black GPU with the Nvidia 435.21 Driver was used. To obtain the Frame Rate for the original paper's models their sampling code was modified to also output a frame rate. As well as this, we need to ensure that images are processed sequentially rather than in batches as this is the scenario in which the models would be used. The results can be seen in Figure 5.3.

This data shows some interesting variation. Theano and TF 1.x appear faster than TF 2.x on the GPU (possibly due to the eager execution changes in TF 2.x) however TF 2.x outperforms on the CPU. This comparison is insightful and shows just how much the framework (or at least a framework's default configuration) can effect FPS.

Class	Precision	Recall	F1-score	# Images
Brain (Cb.)	0.90	0.96	0.93	549
Brain (Tv.)	0.86	0.98	0.92	764
Profile	0.46	0.91	0.61	92
Lips	0.88	0.91	0.89	496
Abdominal	0.93	0.90	0.92	474
Kidneys	0.77	0.77	0.77	166
Femur	0.87	0.93	0.90	471
Spine (cor.)	0.72	0.94	0.81	81
Spine (sag.)	0.60	0.87	0.71	156
4CH	0.81	0.73	0.77	306
3VV	0.68	0.59	0.63	287
RVOT	0.60	0.58	0.59	284
LVOT	0.82	0.74	0.78	317
Background	1.00	0.99	0.99	104722

	precision	recall	f1-score	support
BRAIN-CB	0.98	1.00	0.99	684
BRAIN-TV	0.99	1.00	1.00	949
PROFILE	0.91	0.96	0.94	114
LIPS	0.88	0.98	0.93	576
ABDOMINAL	0.92	1.00	0.96	603
KIDNEYS	0.94	0.86	0.90	213
FEMUR	0.95	1.00	0.97	570
SPINE-CORONAL	0.96	1.00	0.98	88
SPINE-SAGITTAL	0.96	0.99	0.97	176
4CH	0.96	0.94	0.95	359
3VV	0.79	0.68	0.73	348
RVOT	0.68	0.80	0.74	346
LVOT	0.89	0.92	0.91	383
BACKGROUND	0.98	0.86	0.91	1447
accuracy			0.93	6856
macro avg	0.91	0.93	0.92	6856
weighted avg	0.93	0.93	0.93	6856

Figure 5.1: Old and New SonoNet 32 Stats

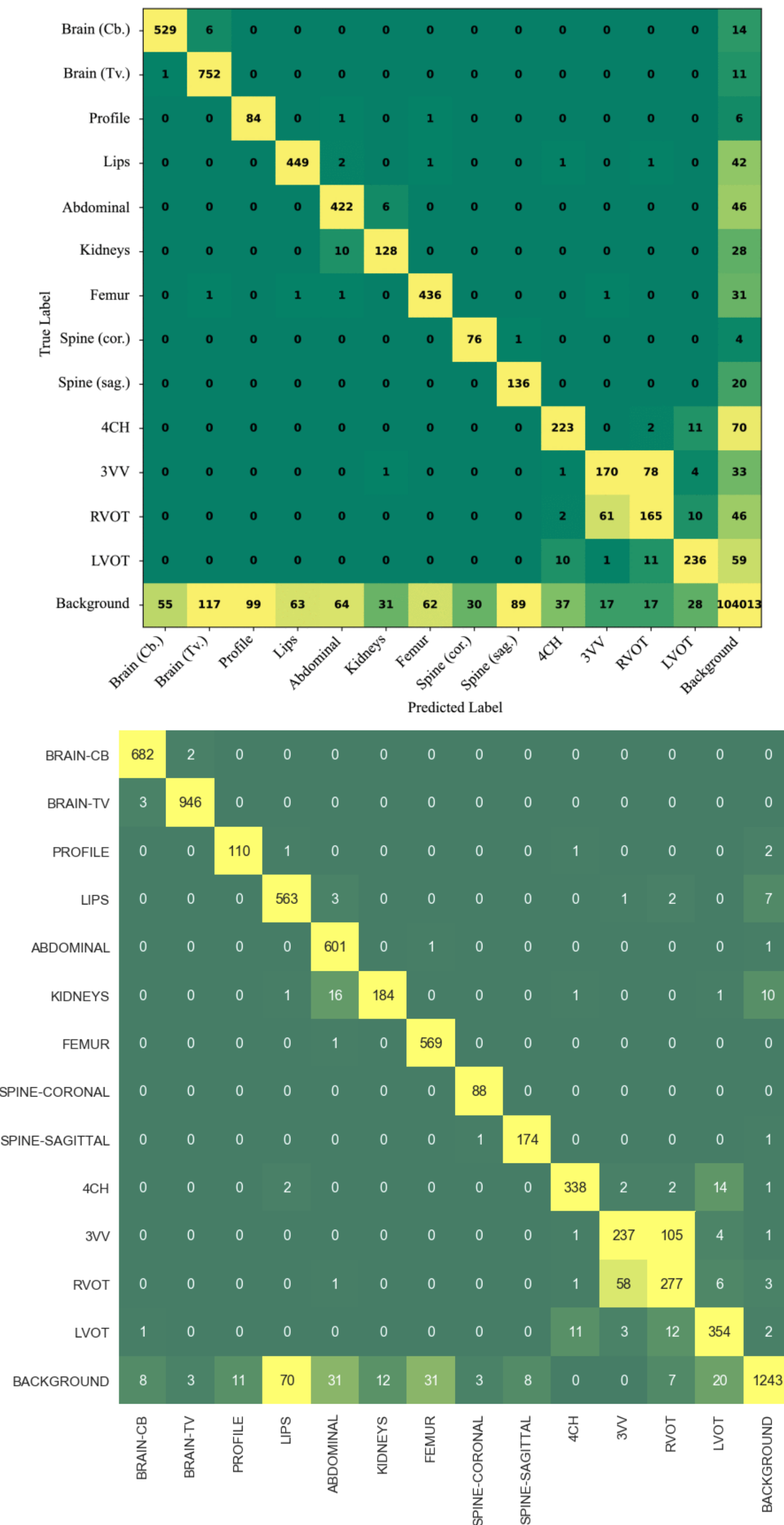


Figure 5.2: Old and New SonoNet 32 Confusion Matrices

	Classification FPS	
	GPU	CPU
Theano	106.1	2.13
TF 1.0	103.6	6.55
TF 2.0	63.9	8.66
TFLite	N/A	15.30

Figure 5.3: SonoNet 32 Desktop Inference FPS

5.1.2 SonoNet TFLite

Looking again at Figure 5.3 we can see that TFLite runs approximately twice as fast on a desktop CPU as the TF 2.x model does. However, notably the TFLite python interpreter does not support running on a GPU and this means that a fair comparison between the two frameworks on a desktop simply is not possible. The GPU (as expected) vastly outperformed all CPU runs. In the final section of this chapter we will cover runtime performance on device and so we will not touch on it here.

We find accuracy of the TF 2.x and TFLite model's to be identical as expected without quantization and the effects of quantization have already been evaluated during its implementation in order to find the ideal parameters.

5.1.3 Biometrics TF 2.x

The other model that was converted and optimized for Android was the Biometrics model. Unlike SonoNet which was concerned with classification this model is ultimately attempting to output a continuous length. To this end it was evaluated differently and so we will also evaluate the converted in this way by using mean absolute error (MAE), mean error (ME) and dice score.

First looking at the results in table form (Figure 5.4), BPD and Dice Scores seem to be fairly similar between the models, but HC seems to be much less accurate for the TF 2.x Model. Looking at the Bland-Altman plots confirm this (Figures 5.5 and 5.6), with the HC plots being much more spread out than the BPD plots.

When we look closely at the results for BPD and HC we can see that the mean absolute error and mean errors are almost identical. This implies that the measurement is off by a constant factor. When we pair this with the fact that the original and TF 2.x BPD standard deviations are fairly similar as well as the similarities in dice score: we can conclude that the TF 2.x is slightly but consistently over-segmenting in a manor which could be compensated for by a fixed offset. The reason why the HC errors are significantly higher in the TF 2.x model is likely due to this constant factor being exaggerated by HC formula; specifically the h component which uses the square of the difference between major and minor axis radii. This also has a similar knock on effect on standard deviation that can be seen in the table.

	Original Paper	TensorFlow Model
HC MAE (mm)	1.80 (1.49)	4.35 (4.91)
HC ME (mm)	0.54 (2.28)	4.34 (4.92)
BPD MAE (mm)	0.68 (0.62)	1.01 (0.89)
BPD ME (mm)	0.13 (0.91)	0.99 (0.92)
Dice Score	0.981 (0.007)	0.982 (0.021)

Figure 5.4: Biometrics Comparison. Value (Standard Deviation)

Even with this correction however, the TF 2.x model undoubtedly has more outliers, both in BPD and HC metrics. We can be fairly certain that the training data used was the same for this model and that the models were the same architecture. We can therefore hypothesis that it was likely the training procedure which caused the difference with the original paper using weights that had been pre-trained on ImageNet, something which was not implemented in the TF 2.x model.

5.1.4 Biometrics TFLite

As with SonoNet, conducting a fair evaluation of the TFLite model's performance outside of its intended setting on device would be ineffective. It will be left to the final subsection of this chapter to discus device performance. Accuracy was again the same between TF 2.x and TFLite models as was to be expected based on what we have seen with SonoNet.

5.1.5 Pruning

In the implementation section, pruning was touched upon and after performing a quick evaluation, it was found to have no impact on performance. Despite this, pruning did have an impact on the SonoNet model's accuracy and size. The results in Figure 5.7 show these impacts. For SonoNet we can evaluate 50% ,75% and 90% model sparsity and it appears that accuracy is only very slightly diminished for all levels of pruning. The counter-intuitive part is that all three levels of sparsity produced models of approximately the same size. We can see reference to this anomaly in a GitHub issue [19], but the TensorFlow developers have yet to provide us with an answer. As such we should treat these results with a high degree of skepticism, it is highly likely that this is one of the areas where the TensorFlow framework is still maturing.

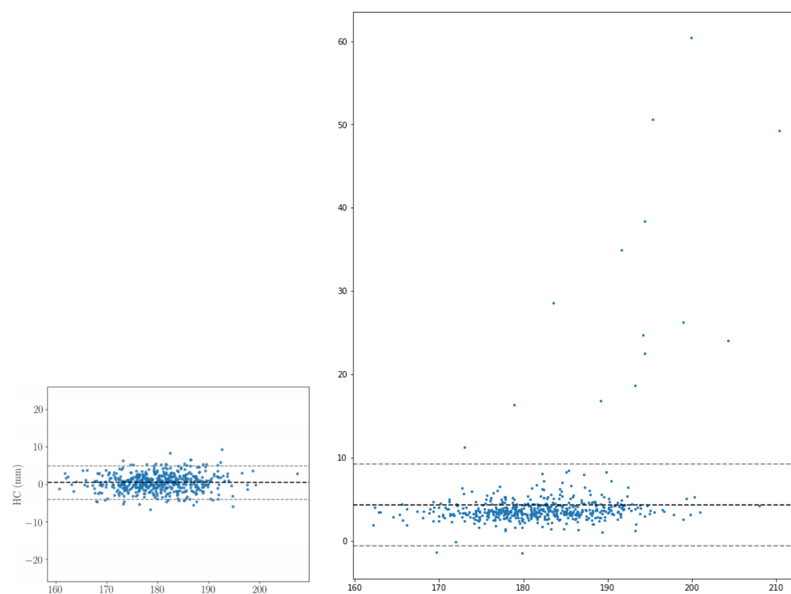


Figure 5.5: HC Bland Altman Plots (Original and TF 2.x)

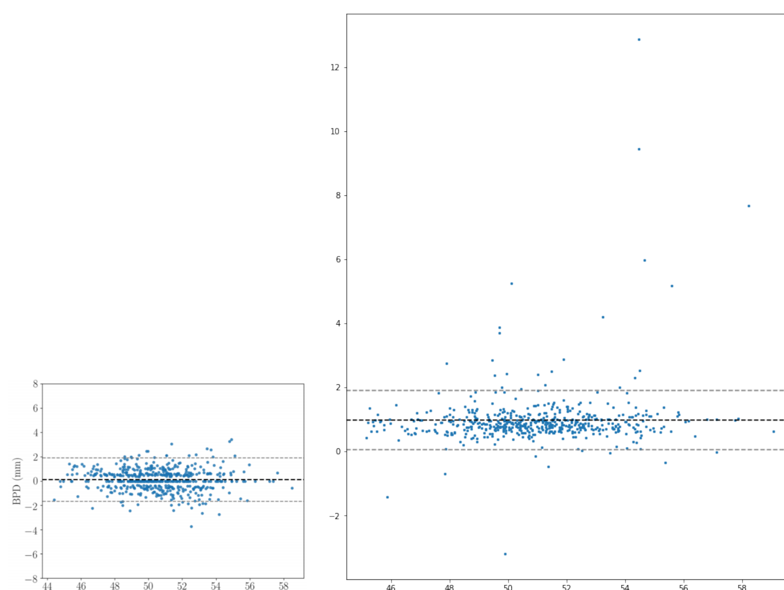


Figure 5.6: BPD Bland Altman Plots (Original and TF 2.x)

Pruning %	Model .h5 Size	Average F1
No Pruning	44755KB	0.929
50%	14996KB	0.919
75%	14997KB	0.921
90%	14997KB	0.926

Figure 5.7: TF 2.X Pruning Effects

5.2 The Android Wrapper

This section deals with the evaluation of the pure Android side of the project without looking at potential models. It analyses various aspects of the app and and looks at how the app might operate with theoretically perfect models that do not make mistake and return results instantly.

5.2.1 Software Compatibility

Android is one of the most widely available OSs in the world, but it is also very fragmented as an ecosystem. Google releases a new Android version each year, often adding or depreciating popular APIs. When building this app a decision was made to limit it to more modern Android versions (Android Oreo (2017)) onward. While it would have been possible to support older devices, it would have required refactoring some key components, such as how screen recording and screen overlay are initiated.

The majority of the changes that make backwards compatibility difficult on Android are based around adding additional required permissions to features in later versions of Android. If in the future it became clear that there was demand to extend this app to older devices then it might be worth doing the necessary work but for this project it was deemed out of scope.

Finally if the user is using a screen overlay such as Twilight that draws over the screen, this would be invariably captured by any running models. This has the potential to reduce accuracy or completely throw off models that might be running. As such users should be warned to turn off all other overlays when using this app.

5.2.2 Device Selection

In order to evaluate how my project performed we need to chose a selection of devices to get a broad view of performance. In order to achieve this we use 4 cloud devices (real devices with their screen being broadcast) to evaluate this project, as well as a local Galaxy S8 and Nexus 5X. We use the Samsung Remote Test Lab [55] to source the cloud devices as it allows multiple hours of manual app tests for free.

We needed to be aware of potential overhead in using an online service but but as can be seen in Figure 5.9 in the comparison between my local S8 and the cloud S8: the performance impact, while measurable is not enormous. The range of free devices in the farm are somewhat limited so the variety of devices chosen for evaluation is not as broad as it might have been. We also aim to choose devices that are certified with the Lumify portable ultrasound scanner and fortunately this largely encompassed Samsung devices anyway.

Full details on the devices chosen can be seen in Figure 5.8. Screen sizes range

Name	Manufacturer	OS	API Version	CPU	GPU	Screen Size	Screen Resolution	Lumify Certified
Galaxy Note10	Samsung	10	29	Exynos 9825	Mali-G76 MP12	6.3”	1080 x 2220	Yes
Galaxy S10	Samsung	9	28	Exynos 9820	Mali-G76 MP12	6.1”	1440 x 3040	Yes
Galaxy Tab S4	Samsung	9	28	Snapdragon 835	Adreno 540	10.5”	1600 x 2560	Yes
Galaxy S8	Samsung	9	28	Exynos 8895	Mali-G71 MP20	5.8”	1440 x 2960	No
Galaxy S8 *	Samsung	9	28	Exynos 8895	Mali-G71 MP20	5.8”	1440 x 2960	No
Nexus 5X *	Google	8.1	27	Snapdragon 808	Adreno 418	5.2”	1080 x 1920	No

Figure 5.8: Android Evaluation Devices

from 5.2” to 10.5” and screen resolutions are varied. A range of operating system versions from 8.1 - 10 are present and a mixture of CPU and GPU chip-sets.

5.2.3 Pipeline Performance

Once we have decided on our devices we can move on to analyse the theoretical maximum performance that a generic model could achieve on these devices. We do this by looking at the base overhead that capturing and overlaying the screen impose on performance.

In order to conduct this evaluation we utilize the flexibly of the code base to build a simple ”model” that returns a single static prediction. We then augmented the code with metrics capturing the frame rate and the frame processing time. We might expect that Frame Processing Time $= \frac{1}{FPS}$ but this is not the case as there is a significant section of the frame processing time that is not captured by these metrics as it is either screen capture code or overlay update code that is not able to be measured by app code.

To calculate this frame overhead we can run the base model on a random YouTube video. If frames are not updating it can slow the FPS as the ImageReader does not see new frames to capture so the input must be moving. By timing for two minutes and then recording the FPS over this period as well as the average Frame Process Time of my code, it is then possible for us to calculate Frame Time by $\frac{1}{FPS}$ and work out the frame overhead. We get this by subtracting Frame Process Time from Frame Time. The results are show in Figure 5.9.

Given how close the FPS rate is to the screen refresh rate for the majority of the devices tested, it is likely that the processing of frames is able to be carried out in realtime on each frame. This means that performance on a theoretically instant inference model could be carried out in realtime at the same frame rate as the devices refresh rate.

5.2.4 On Screen Overlay

While as mentioned in the implementation section there is potential for performance improvements in the on screen overlay area in this section we will quickly evaluate the quantitative effects that these improvements can make by looking at a flame

	FPS	Frame Process Time	Frame Time	Frame Overhead
Galaxy Note10	59.9	2.90ms	16.7ms	13.8ms
Galaxy S10	60.0	2.44ms	16.7ms	14.3ms
Galaxy Tab S4	50.1	4.78ms	20.0ms	15.2ms
Galaxy S8	45.4	5.33ms	22.0ms	16.7ms
Galaxy S8 *	59.3	4.47ms	16.9ms	12.4ms
Nexus 5X *	30.5	8.34ms	32.8ms	24.5ms

Figure 5.9: Android Frame Rates

graph of the current frame processing loop running on a Galaxy S8 (Figure 5.10).

The slowest portion of the code (almost half the loop or 150ms per frame) was spent reversing the screen overlay. The bottlenecks seems to be reading from the ByteBuffer and performing the unsigned integer comparison necessary to work out if a pixel needs to be corrected. In order to optimize this, we would need to move a large chunk of this code to native C++ that would be able to operate much more efficiently on the ByteBuffer. While this might take some time the payoff would be considerable.

In addition to this it seems that creating bitmap images is rather inefficient, specifically creating a cropped bitmap from an input. This in general took 25% of processing time or 75ms. In order to optimize this we would need to research why specifically this was taking so long and look to use another API or custom code to work around it.

Finally waiting for the UI to update with the Future get request that ensures that an update has been drawn before processing the next frame, only adds about 10% or 30ms to the loop. Given that at 60FPS the expected delay to draw a frame would be at least 17ms this seems to be in the right region. As such we can anticipate that if this code was able to moved out of the main loop into a separate thread and if it turned out that 30ms intervals were the highest degree of specificity for the currently drawn overlay image then an overlay update rate of 30FPS would be technically possible while maintaining the information about the original image.

By using a technique such as a single coloured pixel in the corner of the screen who's colour would indicate the overlay frame number modulus a constant, it should be possible to even achieve a frame rate equal to that of the refresh rate (60FPS+) if the two bottlenecks already mentioned were to be solved.

5.3 The Final App

In this section we evaluate the end result of combining optimized models with the Android wrapper to form a fully functional app. We approach this task with both

Figure 5.10: On Screen Overlay Flame Graph

a quantitative approach to performance and qualitative techniques to gain feedback from potential users. Before reading this section reviewing the User Guide (Appendix A) might be helpful.

5.3.1 Device Specific Performance

As mentioned in earlier sections for all the models used: accuracy was not noticeably impacted by the transition to Android. Given that the Android models are just as accurate as their desktop counterparts the key to determining their usability is their speed. TFLite does provide a bench-marking program that can be used over the ADB (Android Device Bridge) to profile models directly, but as TFLite now provides a native inference time property it is possible to get the direct model inference time without any overhead. This also allows more representative data to be easily passed as input, so we will solely us this approach for evaluation.

If we take the output of this method then we can use it to calculate the TFLite FPS (Figure 5.12). In this table Real FPS is the FPS recorded by the app, TFLite Time is the inference time reported by the TFLite library. Frame Time is the reciprocal of Real FPS, Calculated Overhead is Frame Time - TFLite Time and TFLite FPS is the reciprocal of TFLite Time. We can make a quick comparison between TFLite FPS rates and those of a desktops CPU (Figure 5.3). By doing this we can see that the most modern devices GPUs are capable of a similar FPS rate to a desktop CPU when running SonoNet. Compared to the desktop GPUs the model seems to be 3-5x slower.

During the testing the Nexus 5X device had an issue with running models on its GPU. By doing a quick test on a Galaxy S7 edge running the same OS version the issue being OS related was ruled out so it can be assumed to be the hardware. It was likely to be an error in the OpenGL TFLite implementation, a theory supported by an issue open on GitHub [30]. In order to remedy this it was necessary to add a fallback mechanism to use the CPU (and so also the integer quantized model) when running on the GPU threw an error (Listing 5.1).

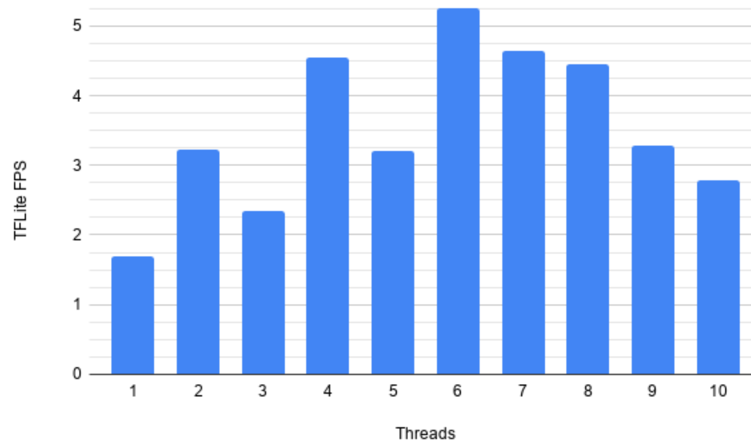


Figure 5.11: TFLite FPS by Thread Count

Looking at Figures 5.13 and 5.14 we can also see again the effect that overlaying directly on the input has on performance with the TFLite FPS rates being almost identical but there being 2-3x as much non TFLite overhead in the direct overlay configuration. In the original biometric paper a desktop GPU rate of 15 FPS was quoted on a Nvidia Titan Xp GPU. This matches the SonoNet model in that top of the line devices are about 3-5x worse in terms of performance.

```
java.lang.IllegalArgumentException: Internal error: Failed to run on the given
    Interpreter: OpenCL library not loaded - dlopen failed: library "libOpenCL-
    pixel.so" not found
    Falling back to OpenGL
    TfLiteGpuDelegate Invoke: Write to buffer failed. Source data is larger than
        buffer.
    Node number 49 (TfLiteGpuDelegateV2) failed to invoke.'''
```

Listing 5.1: The error thown when attemping to use the GPU on a Nexus 5X

Finally, while the difference between CPU and GPU were presented in the implementation section dealing with quantization (Figure 3.4) we only briefly looked at the impact of using multiple threads. In order to complete this evaluation we can measure FPS with varying thread counts on a Galaxy S8 and plot the results for the SonoNet model (Figure 5.11). From this graph it can be seen that 6 threads seemed to be the optimal amount for performance. Any more than this decreases performance. Up until 6 FPS there seems to be a difference between an odd or even thread count, we can conclude this is down to the architecture of the CPU. Finally while the device only had 8 physical cores it is still possibly to use 9+ threads but this just appears to add overhead and reduce performance.

5.3.2 APK Structure and Size

This is good point to spend some time evaluating the contents of a generated app and its size. Using the Android Studio APK analysis tool we can get a breakdown (Figure 5.15). While the full APK is close to 200MB the actual download size for

	Real FPS	TFLite Time	Frame Time	Calculated Overhead	TFLite FPS
Galaxy Note10	9.16	64.7ms	109.2ms	44.5ms	15.46
Galaxy S10	11.22	49.5ms	89.1ms	39.6ms	20.20
Galaxy Tab S4	6.26	72.9ms	159.7ms	107.5ms	13.72
Galaxy S8 *	4.45	93.7ms	224.7ms	136.3ms	10.67
Nexus 5X **	0.81	1021.6ms	1234.5ms	212.9ms	0.98

* Local Device, ** Local Device using CPU

Figure 5.12: SonoNet TFLite Android Frame Rates

	Real FPS	TFLite Time	Frame Time	Calculated Overhead	TFLite FPS
Galaxy Note10	1.60	312ms	625ms	313ms	3.21
Galaxy S10	1.59	305ms	629ms	324ms	3.28
Galaxy Tab S4	0.75	524ms	1333ms	809ms	1.91
Galaxy S8 *	0.77	836ms	1299ms	463ms	1.20
Nexus 5X **	0.12	6906ms	8333ms	1427ms	0.14

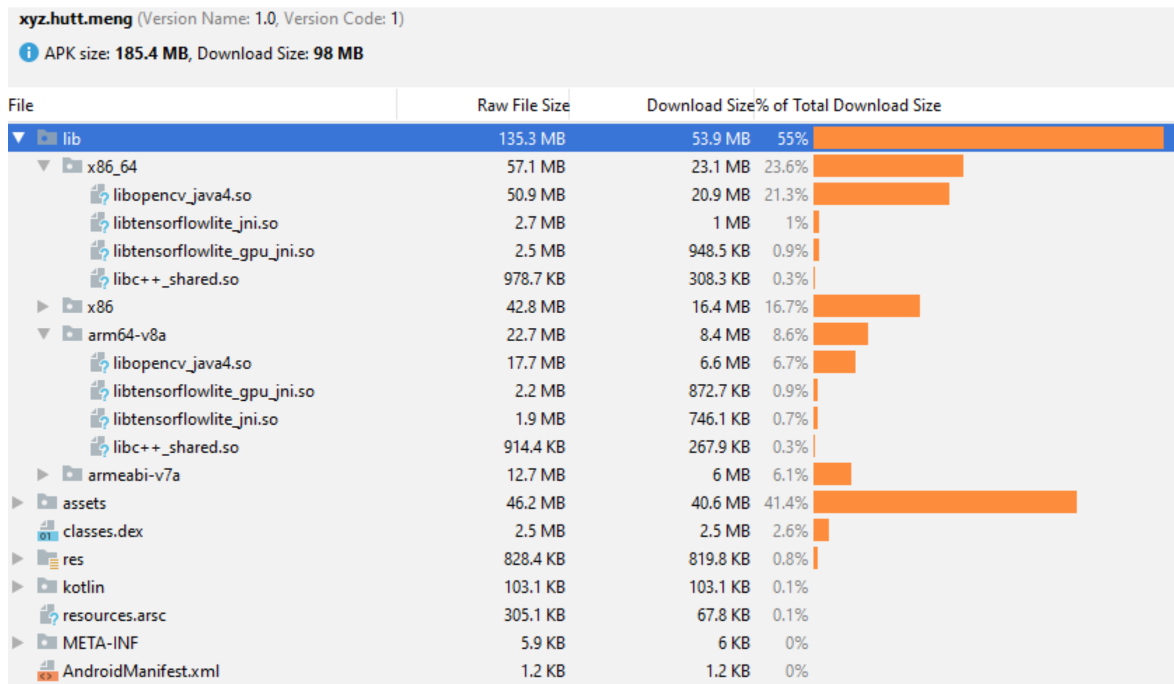
* Local Device, ** Local Device using CPU

Figure 5.13: Segmentation TFLite Android Frame Rates

	Real FPS	TFLite Time	Frame Time	Calculated Overhead	TFLite FPS
Galaxy Note10	2.37	317ms	422ms	105ms	3.15
Galaxy S10	2.57	304ms	389ms	85ms	3.29
Galaxy Tab S4	1.41	518ms	709ms	191ms	1.93
Galaxy S8 *	1.00	822ms	1000ms	178ms	1.22
Nexus 5X **	0.14	6454ms	7143ms	689ms	0.15

* Local Device, ** Local Device using CPU

Figure 5.14: Segmentation TFLite Android Frame Rates (No Overlay)


xyz.hutt.meng (Version Name: 1.0, Version Code: 1)
APK size: 185.4 MB, Download Size: 98 MB

File	Raw File Size	Download Size	% of Total Download Size
lib	135.3 MB	53.9 MB	55%
x86_64	57.1 MB	23.1 MB	23.6%
libopencv_java4.so	50.9 MB	20.9 MB	21.3%
libtensorflowlite_jni.so	2.7 MB	1 MB	1%
libtensorflowlite_gpu_jni.so	2.5 MB	948.5 KB	0.9%
libc++_shared.so	978.7 KB	308.3 KB	0.3%
x86	42.8 MB	16.4 MB	16.7%
arm64-v8a	22.7 MB	8.4 MB	8.6%
libopencv_java4.so	17.7 MB	6.6 MB	6.7%
libtensorflowlite_gpu_jni.so	2.2 MB	872.7 KB	0.9%
libtensorflowlite_jni.so	1.9 MB	746.1 KB	0.7%
libc++_shared.so	914.4 KB	267.9 KB	0.3%
armeabi-v7a	12.7 MB	6 MB	6.1%
assets	46.2 MB	40.6 MB	41.4%
classes.dex	2.5 MB	2.5 MB	2.6%
res	828.4 KB	819.8 KB	0.8%
kotlin	103.1 KB	103.1 KB	0.1%
resources.arsc	305.1 KB	67.8 KB	0.1%
META-INF	5.9 KB	6 KB	0%
AndroidManifest.xml	1.2 KB	1.2 KB	0%

Figure 5.15: APK Components

a user using an arm chip would be much closer to 55MB due to the need to only download the arm native binary files for TFLite and OpenCV as well as the benefits of compression. When this is taken into account about 75% of the download size would be the models (in the asset directory). This includes both integer and float quantized versions of the SonoNet and Biometrics models (which are about 75% of the models by size) as well as a single float quantized version of the MobileNets, Object Detection, and Style Transfer Models. The actual source code and other resources make up 4% of the app.

While this app is certainly on the larger side and does unfortunately break the Google Play store file size limit; this could be worked around by only supporting the arm64 architecture, which is realistically going to be most modern devices. This would bring the raw APK size down to around 75MB while still maintaining broad app compatibility. The other alternative which was utilized for obtaining user feedback is using flavours (customizable subsets of assets and code) to only include a subset of models for each release.

5.3.3 User Feedback

Unfortunately due to the effects of the Covid-19 pandemic it was not possible to evaluate this app in a clinical setting. While we were able to successfully able to evaluate the success of network translation though quantitative means, the feel and experience of the overall app requires a more qualitative approach. While some feedback was gained from sonographers, it is also important to gain a more detailed view of how a potential user might interact with the app. That could only be gained

from watching users use the app themselves. To gain feedback from, this viewpoint friends and family locally and via video call were asked to participate in a short study.

While willing to participate, these non medical users would neither understand or be interested in ultrasound scans. To work around this we instead gain feedback on the more general models running on the app, the TFLite, Object Detection and Style Transfer models. These models are fun to use and don not require any prior medical knowledge. They also solved a distribution issue in that without OpenCV (needed for ellipsis calculation in the brain segmentation model) the app size was under the 100MB required by the play store and so could be distributed by an internal release build.

The obvious disadvantages of taking this approach is that we can not critique direct feedback on the usability of the medical models specifically, but we can still draw on non medical users to obtain general feedback. 5 people interacted with the non medical flavour of the app, talking though their thought processes as they went though the user guide, with myself asking them questions along the way.

In general the feedback was widely positive. Users were able to quickly perform tasks as instructed. General feedback was that the app was in general smooth and well structured with a clean aesthetic. One user suggested switching to NHS blue for the medical variant of the app.

There are certainly areas where improvement could be made. One common theme was about making the reasoning behind the permission requests more clear. While users were on the whole OK with granting overlay access, many paused for thought when granting screen record permissions especially as Android does not allow a more fine grained approach and it is an all or nothing request. Suggestions on how to increase their trust when granting the permission were generally aimed towards providing a deeper explanation. Another common issue was the button icons. The use of the same icon with different colours (while clean), confused a few users. It was not clear to them which specific combination of icons would best represent the implied actions. More work is needed to study this in a larger sample of potential users.

Another piece of feedback was the ability to change the viewfinder aspect ratio. This had been fixed as most of the models this project uses required a fixed aspect ratio and we wanted to avoid skewing, but there is certainly an argument to allow a more variable aspect ratio. While users found the user guide helpful, one suggested a popup tutorial explaining everything in app and this could work well. A few users suggested additional non medical uses such as emulating popular camera filters such as hair colour changers.

5.3.4 SonoNet Specific Feedback

While this user feedback was very useful in the broad case we want to also get feedback on the more specific SonoNet use case. Due to the Covid-19 pandemic there was limited opportunity for interaction with busy medical professionals. However, it was possible to make a video demonstrating SonoNet's features [37] and then ask them three questions based using the Likert scale. We will evaluate the answers to these three questions from 10 sonographers while also looking at the small amount of additional feedback provided.

First we need to get a picture of how sonographers viewed the potential for mobile ultrasound scanners to disrupt current tech and improve patient experience. With mobile ultrasound scanners comes the benefits of additionally processing scans on device (the use case for SonoNet). As seen in Figure 5.16, 90% of sonographers either agreed or strongly agreed that mobile scanners could improve patient experience with 1 being neutral. This reaffirms the need for an app like SonoNet to help with processing.

Secondly it would be useful to gauge if sonographers would like the option to capture additional images or video as well as the standard planes. This is not something that is implemented at the moment but it is something that could easily be implement in the future (Figure 5.17). This question received a more mixed response, although half of the sonographers questioned felt it would be useful. As something like this might be personal to each sonographer, adding the option to capture additional images/video would be a good idea.

Finally it would be helpful to know if sonographers saw potential for this app in a training setting (Figure 5.18). There is the potential to add features that might help during training such as additional hints and a realtime showing of confidence for each plane. As with the first question, 90% of sonographers saw potential in this with 1 being neutral. Based on this the addition of a "Training Mode" could be a good future feature.

Five of the sonographers also gave additional feedback on the video that they were shown. Some additional requested features were the ability to select a specific image from a selection rather than 100% relying on the CNN and also defaulting the view-finding box to a standard location for the scan. Two sonographers emphasized the impact that this app could have on tele-medicine by allowing experts to review scans remotely.

Overall feedback was very positive. In the future an extended app based on this project would have a real possibility of making an impact in the area of fetal ultrasound.

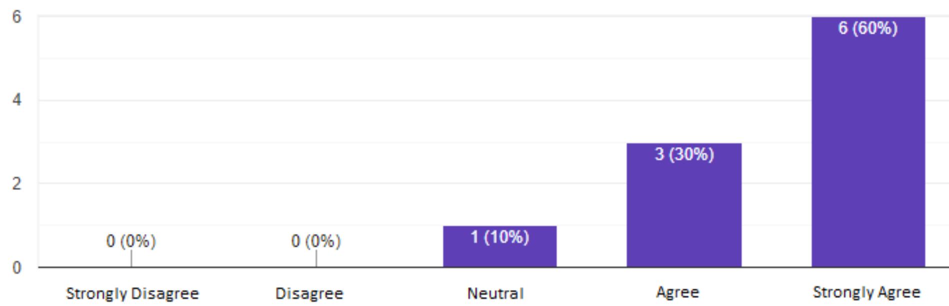


Figure 5.16: Mobile ultrasound scanners combined with intelligent software have the potential to improve patient experience

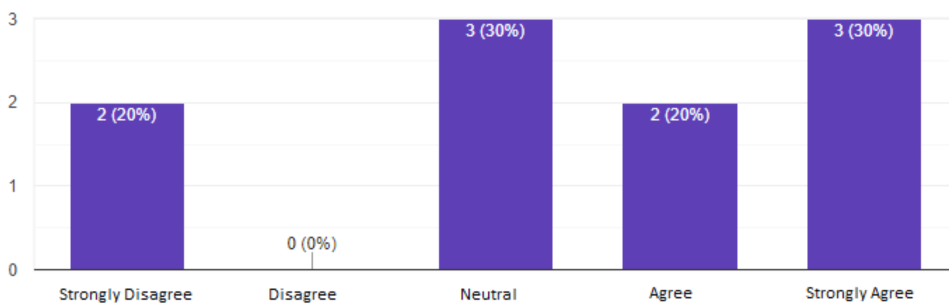


Figure 5.17: The ability to capture additional images or video beyond standard planes is important to me.

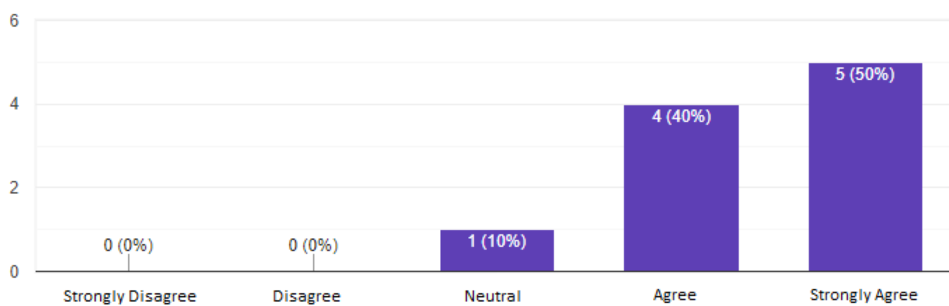


Figure 5.18: An app like this could be helpful in a training setting.

6. Conclusion

6.1 Summary

This project has demonstrated that medical CNN architectures can be ported to work efficiently on Android devices. It has also built an original application to run models on the contents of a phone's screen and overlay information on top of this input in realtime.

During quantitative evaluation, mobile model performance and accuracy have been shown as comparable to the paper versions which they were based on. Throughout qualitative evaluation, valuable feedback has been obtained, confirming the end result was both useful and usable.

Despite the project's many successes, a potential weakness is the failure to find a method to directly translate models to TFLite. As explained previously, machine learning frameworks are simply moving at too quick a pace to allow a sustained degree of backwards compatibility. Unfortunately, this proved infeasible to work around. However, by overcoming significant challenges in documentation and data availability, converting and optimising both SonoNet and the Biometrics model was still accomplished.

Overall this project has met and exceeded its original objectives. It has shown that the power of image analysis and augmentation is not just limited to powerful desktop PCs and I am excited to see where the ideas and implementations developed in this project might lead.

6.2 Future Work

Working on this project was an enjoyable experience and it is interesting to consider its potential to act as a foundation for future impactful developments. Some of these prospects are listed below:

SonoNet

There is the potential to further improve and evaluate the SonoNet model's use in a clinical settings. Some possible improvements include updating the older SonoNet architecture to a more modern design, reevaluating the training data used and evaluating the app in greater depth within a clinical context.

Medical Models

While two medical models were covered by this project, there is the potential to look at converting other medical models that might benefit from a similar app. Research

in this area is advancing at a tremendous pace and opening up other new models for use on mobile platforms could prove to be very impactful. Mobile analysis of ultrasound images was enabled by the commercialisation of mobile ultrasound scanners and so as other imaging methods become more mobile, the ability to process their outputs offline will become more important.

"Fun" Models

Finally, the work done in this project has the potential for a variety of fun and interesting applications. As shown with the Style Transfer model, it is possible to alter the contents of the screen using a model. This could easily be taken even further. By implementing a model such as StarGAN [5], it could be possible to alter the appearance of every person appearing on your device, be it on YouTube or Instagram. While the FPS rate of this might be limited to single digits today, as hardware and models improve it is likely this will be possible in realtime in the next few years. In the future, users might choose how they want to view the world and their smartphone will cater to that preference despite the ethical concerns this ability might raise today.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695, 2016. pages 16

[2] Apple. Replaykit — apple developer documentation. `https://developer.apple.com/documentation/replaykit`. (Accessed on 05/28/2020). pages 12

[3] Christian F. Baumgartner, Konstantinos Kamnitsas, Jacqueline Matthew, Tara P. Fletcher, Sandra Smith, Lisa M. Koch, Bernhard Kainz, and Daniel Rueckert. Sononet: Real-time detection and localisation of fetal standard scan planes in freehand ultrasound. *IEEE Trans. Med. Imaging*, 36(11):2204–2215, 2017. pages 6, 7, 9, 21, 22

[4] Caffe. Caffe — deep learning framework. `https://caffe.berkeleyvision.org/`. (Accessed on 01/23/2020). pages 8

[5] Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8789–8797, 2018. pages 61

[6] Tech Crunch. Kotlin is now google's preferred language for android app development — techcrunch. `https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/`. (Accessed on 05/28/2020). pages 34

[7] deeplearning.net. Welcome — theano 1.0.0 documentation. `http://deeplearning.net/software/theano/`. (Accessed on 06/01/2020). pages 15

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. pages 15

[9] Public Health England. Nhs fetal anomaly screening programme handbook, 2018. pages 20

[10] José Faria, João Almeida, Maria João M Vasconcelos, and Luís Rosado. Automated mobile image acquisition of skin wounds using real-time deep neural networks. In *Annual Conference on Medical Image Understanding and Analysis*, pages 61–73. Springer, 2019. pages 9

[11] Andrew Fitzgibbon, Maurizio Pilu, and Robert B Fisher. Direct least square fitting of ellipses. *IEEE Transactions on pattern analysis and machine intelligence*, 21(5):476–480, 1999. pages 23

[12] Google. Artistic style transfer with tensorflow lite. https://www.tensorflow.org/lite/models/style_transfer/overview. (Accessed on 06/01/2020). pages 6, 16, 42

[13] Google. Batchnorm not in pruneregistry · issue #386 · tensorflow/model-optimization. https://github.com/tensorflow/model-optimization/issues/386. (Accessed on 06/05/2020). pages 28

[14] Google. Behavior changes: all apps. https://developer.android.com/about/versions/pie/android-9.0-changes-all. (Accessed on 01/21/2020). pages 11

[15] Google. Effective tensorflow 2. https://www.tensorflow.org/guide/effective_tf2. (Accessed on 06/02/2020). pages 17

[16] Google. Future. https://developer.android.com/reference/kotlin/java/util/concurrent/Future. (Accessed on 06/01/2020). pages 38

[17] Google. Intents and intent filters. https://developer.android.com/guide/components/intents-filters. (Accessed on 06/01/2020). pages 11

[18] Google. Machine learning foundations: Ep #7 - image augmentation and overfitting. https://www.youtube.com/watch?v=QWdYWwW6OAE. (Accessed on 06/01/2020). pages 16

[19] Google. model_pruning: Why 50% and 90% zeros of the stripped models are the same size? · issue #32805 · tensorflow/tensorflow. https://github.com/tensorflow/tensorflow/issues/32805. (Accessed on 06/11/2020). pages 48

[20] Google. Performance benchmarks. https://www.tensorflow.org/lite/performance/benchmarks. (Accessed on 06/01/2020). pages 41

[21] Google. Porterduff.mode. https://developer.android.com/reference/android/graphics/PorterDuff.Mode#SCREEN. (Accessed on 06/10/2020). pages 40

[22] Google. Post-training quantization. https://www.tensorflow.org/lite/performance/post_training_quantization. (Accessed on 01/22/2020). pages 19

[23] Google. Processes and application lifecycle. https://developer.android.com/guide/components/activities/process-lifecycle. (Accessed on 01/21/2020). pages 11

[24] Google. Quantization aware training comprehensive guide. https://www.tensorflow.org/model_optimization/guide/quantization/training_comprehensive_guide. (Accessed on 06/06/2020). pages 20

[25] Google. Services overview. https://developer.android.com/guide/components/services. (Accessed on 01/21/2020). pages 6, 11, 12

[26] Google. Sparsity runtime integration with tf/tflite for latency improvements · issue #173 · tensorflow/model-optimization. https://github.com/tensorflow/model-optimization/issues/173. (Accessed on 06/05/2020). pages 28

[27] Google. Tensorflow lite. https://www.tensorflow.org/lite. (Accessed on 01/15/2020). pages 17

[28] Google. Tensorflow lite roadmap. https://www.tensorflow.org/lite/guide/roadmap. (Accessed on 05/28/2020). pages 18

[29] Google. Tensorflow model optimization. https://www.tensorflow.org/model_optimization. (Accessed on 06/05/2020). pages 28

[30] Google. tflitegpudelegate invoke write to buffer failed source data is larger than buffer · issue #33410 · tensorflow/tensorflow. https://github.com/tensorflow/tensorflow/issues/33410. (Accessed on 05/28/2020). pages 53

[31] Google. Understand the activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle. (Accessed on 01/21/2020). pages 6, 12

[32] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. pages 19

[33] healthitanalytics.com. Top 5 use cases for artificial intelligence in medical imaging. https://healthitanalytics.com/news/top-5-use-cases-for-artificial-intelligence-in-medical-imaging. (Accessed on 06/06/2020). pages 7

[34] Heuritech. https://web.archive.org/web/20160305154644/http://blog.heuritech.com/2016/brief-report-of-the-heuritech-deep-learning-meetup-5/. https://web.archive.org/web/20160305154644/http://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/. (Accessed on 01/22/2020). pages 6, 14

[35] GD Hill, JR Block, JB Tanem, and MA Frommelt. Disparities in the prenatal detection of critical congenital heart disease. *Prenatal diagnosis*, 35(9):859–863, 2015. pages 7

[36] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. pages 6, 15

[37] Isaac Hutt. Ultrasound app evaluation - youtube. `https://www.youtube.com/watch?v=spbX4Ulr3qY&feature=youtu.be`. (Accessed on 06/13/2020). pages 58

[38] Loc Nguyen Huynh, Rajesh Krishna Balan, and Youngki Lee. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, pages 25–30, 2016. pages 9

[39] IDC. Smartphone market share - os. `https://www.idc.com/promo/smartphone-market-share/os`. (Accessed on 01/21/2020). pages 11

[40] idownloadblog.com. How to enable facebook chat heads anywhere in ios 7. `https://www.idownloadblog.com/2014/03/31/messagebox-ios-7/`. (Accessed on 05/28/2020). pages 13

[41] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. Ai benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 0–0, 2018. pages 9

[42] imgaug.readthedocs.io. imgaug — imgaug 0.4.0 documentation. `https://imgaug.readthedocs.io/en/latest/`. (Accessed on 06/01/2020). pages 27

[43] Ajit Jaokar. Tensorflow 1.x vs 2.x. – summary of changes - data science central. `https://www.datasciencecentral.com/profiles/blogs/tensorflow-1-x-vs-2-x-summary-of-changes`. (Accessed on 06/02/2020). pages 17

[44] Keras. Home - keras documentation. `https://keras.io/`. (Accessed on 01/20/2020). pages 16

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. pages 13

[46] lasagne.readthedocs.io. Welcome to lasagne — lasagne 0.2.dev1 documentation. `https://lasagne.readthedocs.io/en/latest/`. (Accessed on 06/01/2020). pages 15

[47] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 1201–1205, 2016. pages 9

[48] Jiayi Liu, Jayanta Dutta, Nanxiang Li, Unmesh Kurup, and Mohak Shah. Usability study of distributed deep learning frameworks for convolutional neural networks. *Deep Learning Day at KDD*, 2018. pages 18

[49] Microsoft. Microsoft and facebook create open ecosystem for ai model interoperability. `https://azure.microsoft.com/en-us/blog/microsoft-and-facebook-create-open-ecosystem-for-ai-model-interoperability/`. (Accessed on 01/22/2020). pages 18

[50] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019. pages 19

[51] NHS. Ultrasound scan - nhs. `https://www.nhs.uk/conditions/ultrasound-scan/`. (Accessed on 05/30/2020). pages 20

[52] ONNX. Onnx. `https://onnx.ai/`. (Accessed on 01/20/2020). pages 18

[53] Philips. Portable ultrasound machine — philips lumify. `https://www.philips.co.uk/healthcare/sites/lumify`. (Accessed on 01/21/2020). pages 11, 20

[54] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015. pages 14

[55] Samsung. Remote test lab - distribute — samsung developers. `https://developer.samsung.com/remote-test-lab`. (Accessed on 06/12/2020). pages 50

[56] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. pages 14

[57] Matthew Sinclair, Christian F Baumgartner, Jacqueline Matthew, Wenjia Bai, Juan Cerrolaza Martinez, Yuanwei Li, Sandra Smith, Caroline L Knight, Bernhard Kainz, Jo Hajnal, et al. Human-level performance on automatic head biometrics in fetal ultrasound using fully convolutional neural networks. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 714–717. IEEE, 2018. pages 6, 9, 22, 23, 24, 31

[58] Burhanudin Syamsuri and Gede Putra Kusuma. Plant disease classification using lite pretrained deep convolutional neural network on android mobile device. *International Journal of Innovative Technology and Exploring Engineering*, 9(2):2278–3075, 2019. pages 9

[59] Ryosuke Tanno and Keiji Yanai. Caffe2c: A framework for easy implementation of cnn-based mobile applications. In *Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services*, pages 159–164, 2016. pages 8

[60] Pallavi Vaish, R Bharath, Pachamuthu Rajalakshmi, and Uday B Desai. Smartphone based automatic abnormality detection of kidney in ultrasound images. In *2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pages 1–6. IEEE, 2016. pages 9

[61] Ysh329. deep-learning-model-convertor — the convertor/conversion of deep learning models for different deep learning frameworks/softwares. `https://ysh329.github.io/deep-learning-model-convertor/`. (Accessed on 05/30/2020). pages 18

A. User Guide

Once the app has been installed (Figure B.1) and opened for the first time, permission must be given for the app to operate. The app will automatically open in the settings window. You must allow the app permission to appear on top of other apps for the app to function (Figure B.3). Once this permission has been given you can click the back button to return to the app.

Next you can click one of the prototype models to start experimenting (Figure B.2). This will bring up a second permission dialog asking to allow access to capture the screen (Figure B.3). This is essential for the app to be able to process other app's content. Clicking start now allows the app to start up its analysis engine.

Once this is done you are free to exit the app as you would normally and the bottom UI element will still stay in place over other apps. To get rid of it and exit hold down the right hand camera shaped button.

To start the analysis press the right hand button to start and press it again to stop. The left hand button brings up a box that can be dragged around the screen (Figure B.5). It can also be resized by dragging your finger left/right outside of the box. This box determines where on the screen the app will analyse (if the orange box is not pressed it will analyse the whole screen). To use the new capture area simply press the left button again to hide the box then stop and start the recording.

A small amount of additional information is given for each model as follows:

TFLite

This model is a simple MobileNets model that will classify images presented to it into 1000 classes.

Detect

This model is a step above TFLite in that it can detect multiple objects and will draw boxes around them with labels (Figure B.4).

Style

This model performs artistic style transfer on the section of the screen being analysed (Figure B.8).

SonoNet

This model will classify the fetal plane in the view. It also stores the best match for each plane and holding down the orange button will bring up a checklist of planes

collected. Clicking send will then allow these best images to be attached to an email (Figure B.5).

Segmentation

This model will draw a brain/background segmentation over the Brain-TV fetal plane. It will also draw the major and minor axis on the ellipse generated by this segmentation and output the percentage of the view taken up by the major/minor axis lines (Figure B.6).

B. App Screenshots

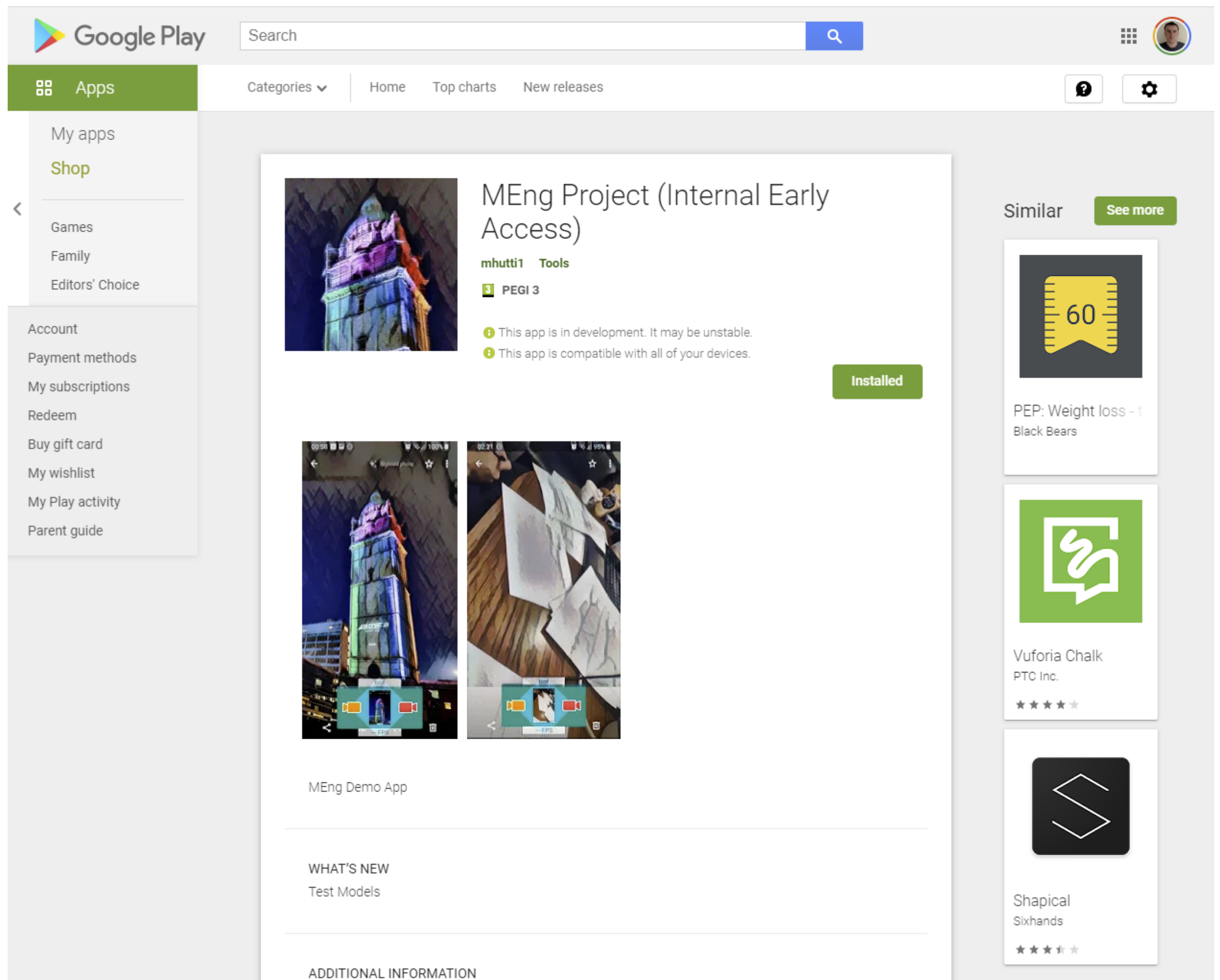


Figure B.1: MEng Project Play Store Listing

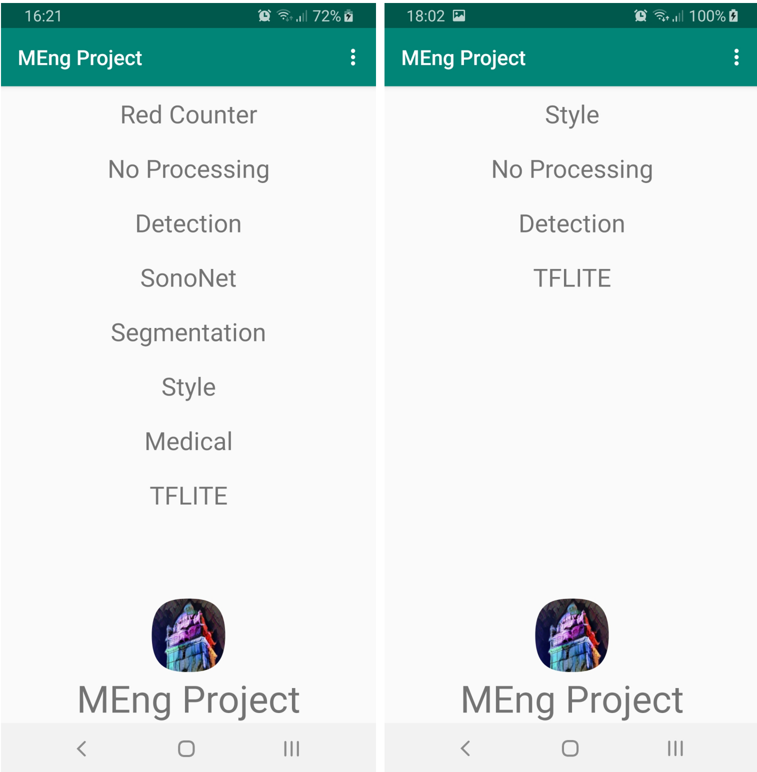


Figure B.2: Model Selection in Full and Generic Flavours

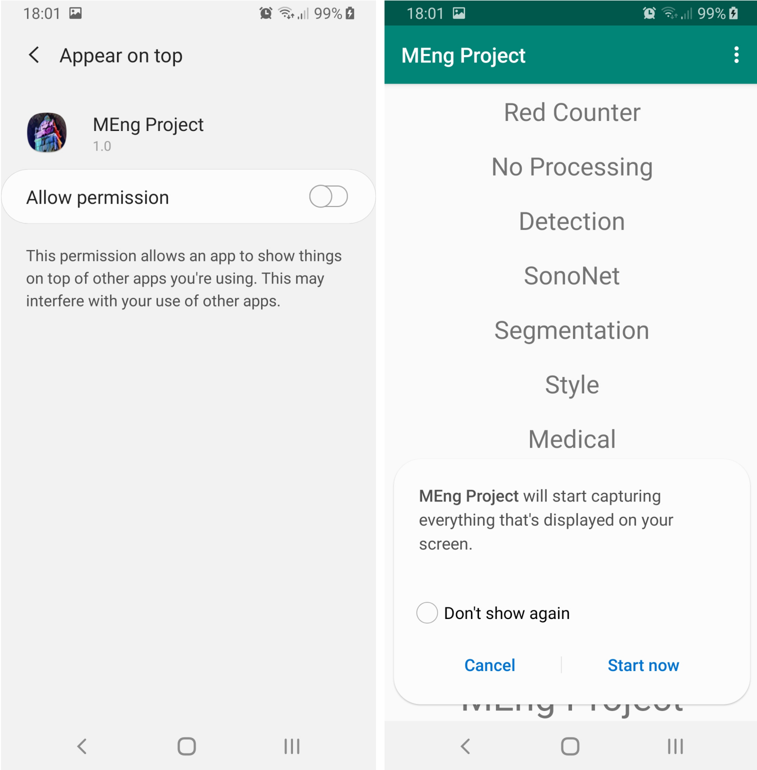


Figure B.3: System Permission Dialogs

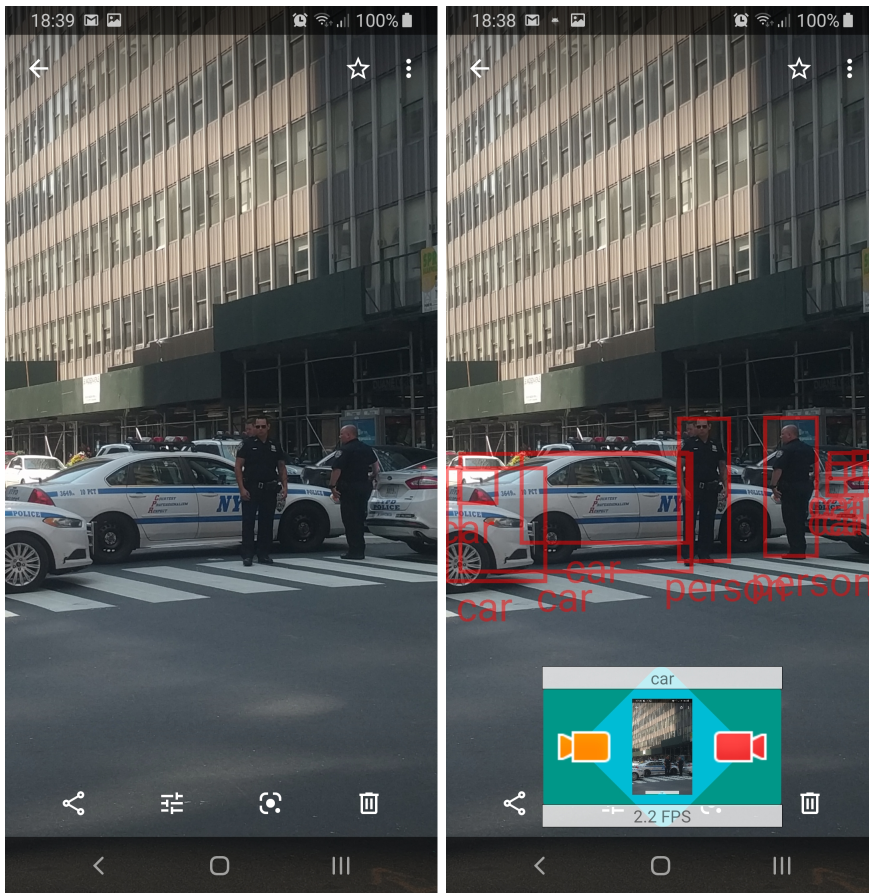


Figure B.4: Object Detection

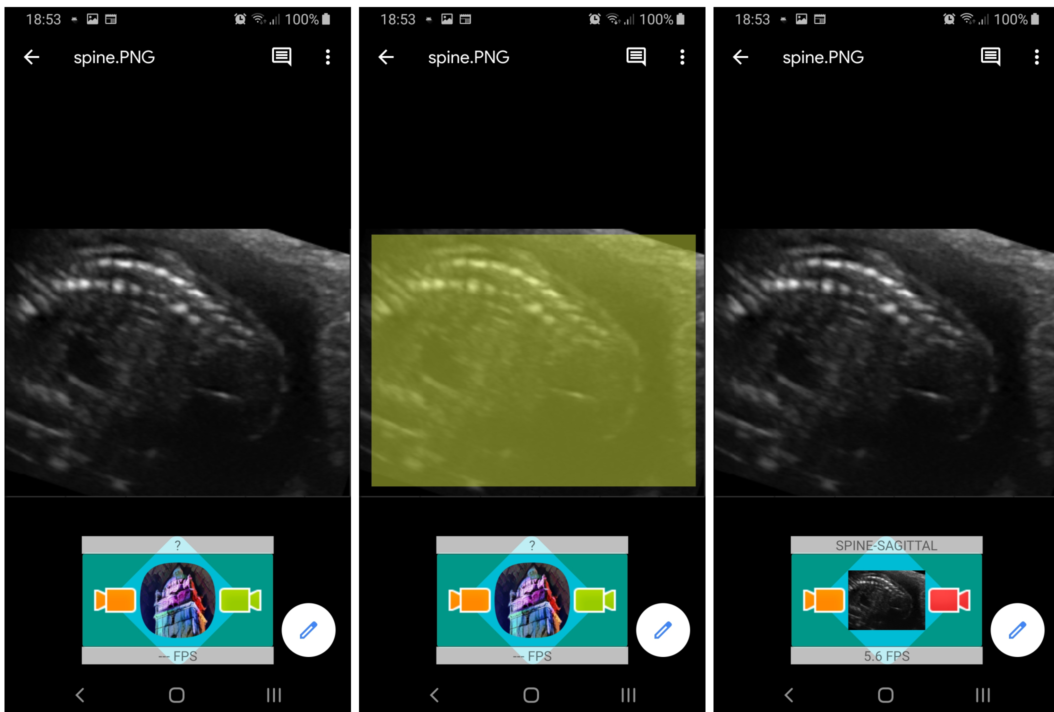


Figure B.5: Viewfinding over an Image of an ultrasound scan

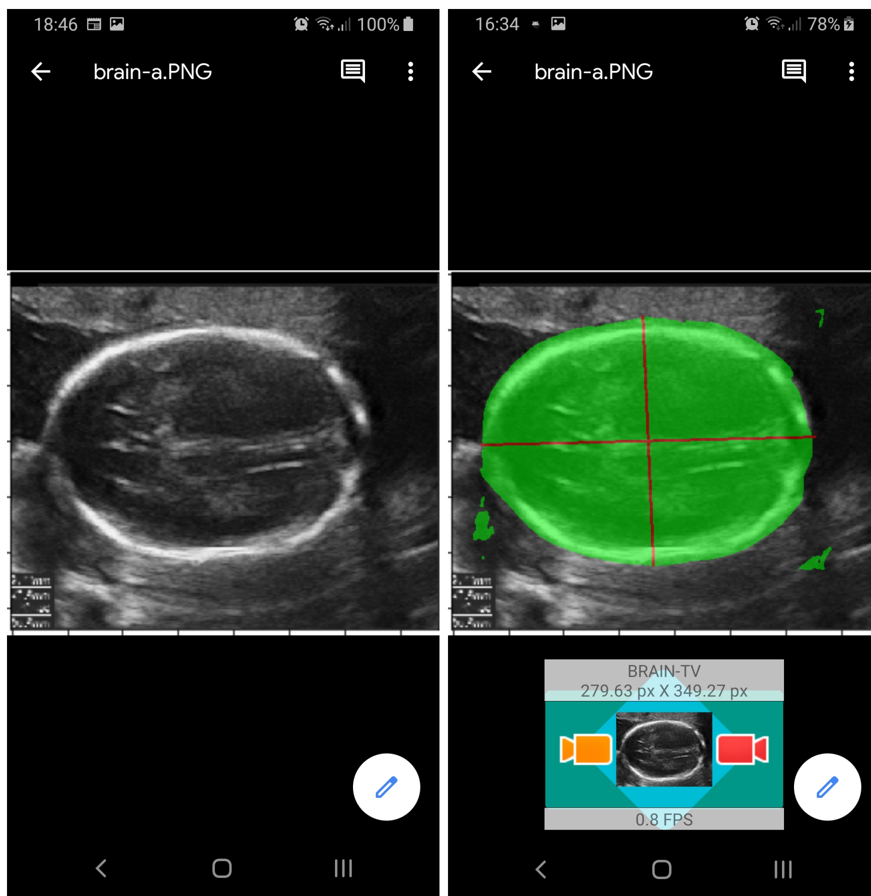


Figure B.6: Brain Segmentation and Metric Calculation

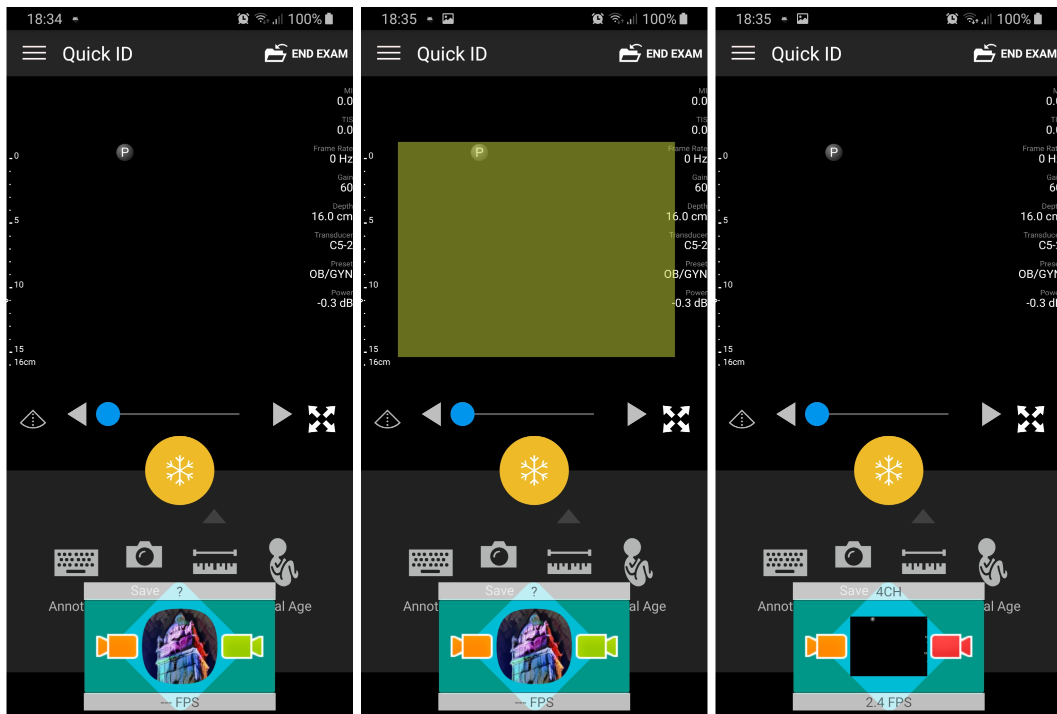


Figure B.7: View-finding over A Real Ultrasound App

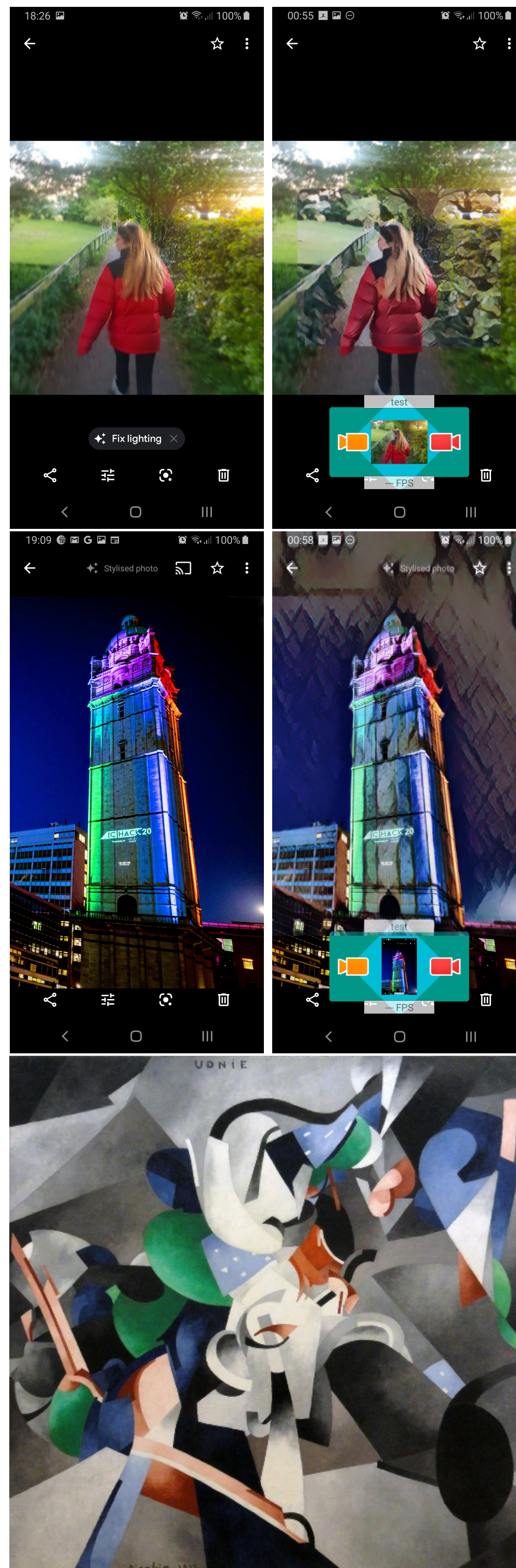


Figure B.8: Style Transfer Template and Examples

C. Additional Tables

	precision	recall	f1-score	support
BRAIN-CB	0.98	1.00	0.99	684
BRAIN-TV	0.99	1.00	1.00	949
PROFILE	0.92	0.96	0.94	114
LIPS	0.88	0.98	0.93	576
ABDOMINAL	0.92	1.00	0.95	603
KIDNEYS	0.95	0.86	0.90	213
FEMUR	0.95	1.00	0.97	570
SPINE-CORONAL	0.97	1.00	0.98	88
SPINE-SAGITTAL	0.96	0.99	0.97	175
4CH	0.95	0.94	0.95	359
3VV	0.79	0.68	0.73	348
RVOT	0.68	0.79	0.73	346
LVOT	0.89	0.92	0.90	383
BACKGROUND	0.98	0.86	0.92	1447
accuracy			0.93	6855
macro avg	0.92	0.93	0.92	6855
weighted avg	0.93	0.93	0.93	6855

	BRAIN-CB	BRAIN-TV	PROFILE	LIPS	ABDOMINAL	KIDNEYS	FEMUR	SPINE-CORONAL	SPINE-SAGITTAL	4CH	3VV	RVOT	LVOT	BACKGROUND
BRAIN-CB	682	2	0	0	0	0	0	0	0	0	0	0	0	0
BRAIN-TV	3	946	0	0	0	0	0	0	0	0	0	0	0	0
PROFILE	0	0	109	1	0	0	0	0	0	1	0	0	0	3
LIPS	0	0	0	563	3	0	0	0	0	0	1	2	0	7
ABDOMINAL	0	0	0	0	601	0	1	0	0	0	0	0	0	1
KIDNEYS	0	0	0	2	16	184	0	0	0	1	0	0	0	10
FEMUR	0	0	0	0	1	0	569	0	0	0	0	0	0	0
SPINE-CORONAL	0	0	0	0	0	0	0	88	0	0	0	0	0	0
SPINE-SAGITTAL	0	0	0	0	0	0	0	1	173	0	0	0	0	1
4CH	0	0	0	2	0	0	0	0	0	339	2	2	13	1
3VV	0	0	0	0	0	0	0	0	0	1	238	104	4	1
RVOT	0	0	0	0	1	0	0	0	0	2	59	275	6	3
LVOT	1	0	0	0	0	0	0	0	0	12	3	13	352	2
BACKGROUND	7	4	9	71	34	10	28	2	8	0	0	7	20	1247

Figure C.1: SonoNet Integer Quantization Results

D. Model Listings

```
tf.keras.backend.set_image_data_format("channels_last")

model = tf.keras.Sequential()
model.add(layers.Conv2D(32, 3, padding='same', use_bias=False, input_shape=(height,width, 1)))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(32, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.MaxPool2D(2, 2))
model.add(layers.Conv2D(64, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(64, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.MaxPool2D(2, 2))
model.add(layers.Conv2D(128, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(128, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(128, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.MaxPool2D(2, 2))
model.add(layers.Conv2D(256, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(256, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(256, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.MaxPool2D(2, 2))
model.add(layers.Conv2D(256, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(256, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(256, 3, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())

model.add(layers.Conv2D(128, 1, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))
model.add(layers.ReLU())
model.add(layers.Conv2D(num_classes, 1, padding='same', use_bias=False))
model.add(layers.BatchNormalization(center=False, scale=False))

model.add(layers.GlobalAvgPool2D())

model.add(layers.Softmax())

model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
              metrics=(['accuracy', tf.keras.metrics.CategoricalAccuracy()]))
```

Figure D.1: A Simple Sequential Implementation of SonoNet 32 in Keras

```
from tensorflow.keras.models import Sequential

def double_conv(out_channels):
    return Sequential([
            layers.Conv2D(out_channels, 3, padding='same'),
            layers.ReLU(),
            layers.BatchNormalization(center=False, scale=False),
            layers.Conv2D(out_channels, 3, padding='same'),
            layers.ReLU(),
            layers.BatchNormalization(center=False, scale=False),
            ])

class UNet():
    def __init__(self):
        self.dconv_down1 = Sequential([
            layers.Conv2D(32, 3, padding='same'),
            layers.ReLU(),
            layers.Conv2D(32, 3, padding='same'),
            layers.ReLU()
        ])

        self.dconv_down2 = double_conv(128)
        self.dconv_down3 = double_conv(256)
        self.dconv_down4 = double_conv(512)

        self.maxpool = layers.MaxPool2D(2, 2)
        self.upsample = layers.UpSampling2D(size=(2, 2), interpolation='bilinear')

        self.dconv_up3 = double_conv(256)
        self.dconv_up2 = double_conv(128)
        self.dconv_up1 = double_conv(64)

        self.conv_last = layers.Conv2D(2, 1, padding='same')
        self.dropout = layers.Dropout(0.6)
        self.softmax = layers.Softmax()

    def forward(self, x):
        conv1 = self.dconv_down1(x)
        x = self.maxpool(conv1)

        conv2 = self.dconv_down2(x)
        x = self.maxpool(conv2)

        conv3 = self.dconv_down3(x)
        x = self.maxpool(conv3)

        x = self.dropout(x)
        x = self.dconv_down4(x)

        x = self.upsample(x)
        x = layers.concatenate([x, conv3], axis=-1)
        x = self.dconv_up3(x)
        x = self.upsample(x)
        x = layers.concatenate([x, conv2], axis=-1)
        x = self.dconv_up2(x)
        x = self.upsample(x)
        x = layers.concatenate([x, conv1], axis=-1)

        x = self.dconv_up1(x)
        return self.softmax(self.conv_last(x))
net = UNet()
input1 = layers.Input((height,width, 1))
model = Model([input1], [net.forward(input1)])
model.compile(optimizer=tf.keras.optimizers.Adam(0.00001),
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=False),
              metrics=(['accuracy']))
```

Figure D.2: A Non Sequential Keras Model for Segmentation in Head Biometrics