

Coffee Roasting Assistant App

Comp 8045 – Major Project 1

Robert Moffat – A00916129

January 28, 2022

Table of Contents

1. Introduction	3
1.1. Student Background.....	3
1.2. Project Description.....	3
1.2.1. Essential Problems	3
1.2.2. Goals and Objectives.....	4
2. Body	5
2.1. Background	5
2.2. Project Statement	5
2.3. Possible Alternative Solutions.....	5
2.4. Chosen Solution	6
Image Recognition Neural Network.....	6
2.5. Details of Design and Development.....	7
Deliverables.....	7
Feasibility Assessment	8
Convolutional Neural Network	9
Specific Details and Implementation	13
2.6. Testing Details and Results	43
2.7. Implications of Implementation.....	66
2.7.1. Innovation	66
2.8. Complexity	67
2.9. Research in New Technologies	69
2.10. Future Enhancements.....	70
2.11. Timeline and Milestones.....	71
3. Conclusion.....	74
3.1. Lessons Learned.....	74
3.2. Closing Remarks	74
4. References	75
5. Change Log.....	75
6. Appendix	75
6.1. Project Supervisor Approvals.....	75

6.2. Approved Proposal..... 76

1. Introduction

1.1. Student Background

I became interested in programming when I was a child because of a fascination with video games combined with a desire to build things which could autonomously function. I first learned some BASIC for DOS from a small introductory booklet I had as a child. Unfortunately, that small pamphlet was all I had access to at the time and I quickly learned all I could from it. After that I didn't do much coding until the end of high school when I learned ActionScript for Flash to make some small Flash games and simulations for my physics class. From there I slowly started learning other programming languages and ultimately decided to go to school to pursue it as a career.

I completed 2-year Computer System Technology diploma in the Digital Processing option at BCIT then continued my education in the BTech degree program, doing the first year in Games Development and then finishing in the Wireless and Mobile Applications Development option part-time while working full time roasting coffee.

Through my time in school, I worked on two major projects for outside parties. The first project was a news aggregation website for an esports betting website. In this project I headed database development and the PHP backend. After that, the second project was an Android app designed by a doctor to store personal medical information in coordination with the user's personal doctor. On this project I did all the testing, some database integration, and general Android development.

1.2. Project Description

This project was to develop a mobile Android app with the purpose of assisting in professional and hobby coffee roasting. The app stores information relevant to the user's specific roast profiles. It is set up with the roaster's temperature readout display centered on the app's camera display where the app then uses image recognition to automate temperature data collection during a roasting session. This information is graphed and can be viewed as it changes, as well as reloaded at any time by the user. These graphs can be overlayed to compare changes in bean quality or to visualize the outcome of changing roasting parameters.

1.2.1. Essential Problems

The problems which this project aims to solve are:

- Recording second to second temperature data and storing that information.
- Graphing temperature over time of a coffee roast.
- Automating data collection.
- Storing information pertaining to roasting.

- Backing up or sharing roast information.

The problems of automating constant data collection are solved using a custom designed image recognition system. So, this problem refined to: steadily converting images of a temperature readout to an integer coupled with a time at a speed that keeps up with the changing temperatures.

Manually storing collected roast data can take up a lot of space over time and create clutter. It can also be lost or damaged. The app will store all previous roasts to file and roasting data can be exported to an Excel spreadsheet to either be emailed or uploaded to Google drive.

A slightly more abstract problem which is specific to this project is the problem of converting a temperature reading on the roasting machine into an array of temperature data stored on a computer, in this case a phone. With the solution to this problem being the creation of an image recognition system utilizing a custom-made convolutional neural network which will take in an image of the temperature reading and output a temperature integer to be stored on the phone.

1.2.2. Goals and Objectives

There are three separate pieces of software that were developed for this project. The main one being the android app, and the other two being the desktop neural network training program, and the remote NodeJS server which stores downloadable roast data.

The objective of the desktop neural network training program is, as it sounds, to train the neural network to be used in the program. The goal more specifically is to take a large folder of correctly labeled images of numbers and use them to train a neural network model through backpropagation to identify which of the ten digits are being pictured in each bitmap, or an eleventh output which represents an image which does not contain a digit. The goal of the final model is to obtain >90 accuracy on camera data.

The goal for the Android app is to first and foremost is to load a trained convolutional neural network model and use it to record constant accurate temperature recording with an easy-to-use interface. Next is to have easily accessible data storage. The app also has a checkpoint system with which the user can save predetermined points which they make adjustments to be labelled and noted.

As for the server, the goal is to take HTTP requests and convert them to database queries to select certain information or insert new information.

2. Body

2.1. Background

The coffee roasting process involves applying varying amounts of heat over time to develop the flavours which are desirable for drinking. Beans are heated inside a rotating drum inside the machine. Below the drum is a propane fueled flame which applies the heat to the drum that roasts the beans. The current temperature within the roaster is shown on a seven-segment digital display on the front of the roaster. Over the process of roasting the coffee several adjustments to gas and airflow are made. During the roasting process the single main output of the machine is the digital temperature display on the front of the roaster, although sound is also used to a lesser extent, and the main inputs are the pressure of the burner gas and the airflow through the drum. Temperatures and adjustments are noted along with the time they occurred throughout a roast. Manually recording these is done at limited sample points while the roaster is busy with other duties.

2.2. Project Statement

Roasting data collection can involve a lot of unnecessary work going to and from the computer or notepad to record the varying temperatures and different significant points during the roast. Having a portable phone app to do this would simplify the process and greatly free up the roaster's attention. The app would amplify this even more, by not requiring the roaster's interaction at all for many data collection points, and minimal input for the ones which do require oversight. In addition, the app will have the additional functionality of getting the roaster's attention through the use of warning sound effects when a point requiring input is approaching. This will eliminate the problem of roaster distraction. Aside from these roaster related problems which will be solved, there is also the problem of accurate data collections. Normally temperature data can only reasonably be recorded at a handful of points manually, which can lead to many insights being missed, and requires a lot of attention. This app will solve that by having a steady stream of temperature data throughout the entire roasting process. Having the entirety of the temperature data will give many times greater insights into the performance of the roast and how different variables affect the outcome.

2.3. Possible Alternative Solutions

There are several possible alternative solutions to achieve the outcome desired from this project. Each has varying requirements financially and in difficulty.

The easiest, but most expensive way to achieve the same level of data collection would be to purchase a newer roaster with a built-in computer. There are many newer roasters with built in computers which have an output port to which the user can connect a computer and record roasting data directly from sensors using proprietary software. Of course, the downside of this is the large investment of a new roaster, which is most likely not worth the cost if the only reason is to achieve greater data insights.

A similar solution would be to purchase and manually install another sensor into the roasting machine. This would require skill or hiring someone skilled as well as financial investment. There are ones which exist that have built in USB ports to connect a computer, but if one is purchased without there would also be the added technical difficulty of creating a connection for data collection. This is a good solution, but still requires some level of financial investment and installation.

As for an alternative solution to the problem which could have been implemented in the app would be to utilize a premade machine learning or image recognition library such as TensorFlow or OpenCV. This would do a better job than anything I would be able to create but would take away much of the deep understanding of convolutional neural networks which was learned through the development of the image recognition system from scratch.

2.4. Chosen Solution

The chosen solution was to develop a full specialized image recognition system from scratch without the use of any machine learning libraries. This was used as an opportunity to learn from the ground up how to develop neural networks, and the calculus involved in training them. Creating a way to visually collect temperature data allows for an easy way to obtain this data without the need for any additional equipment aside from a smartphone which is assumed to be possessed by every roaster.

Image Recognition Neural Network

In the proposal of the project, it was mentioned that the image recognition would go down one of two potential avenues being: attempting to develop a convolutional neural network from scratch, or if this was not successful to switch to using a premade library such as TensorFlow or OpenCV. Although it took longer than anticipated, the development of the image recognition from scratch was successful. One of the potential problems with developing a system myself was creating something which was fast enough to keep up with temperature readings as they are provided. Luckily what was created was more than sufficient, and actually has the time to average the outcome of several rounds of forward propagation from different positions, and so ended up being many times faster than the minimum acceptable speed. This acceptable speed being quite slow was also something overlooked during the planning of the project, as during roasting the temperature generally only changes once every few seconds.

From the Ground Up

The image recognition in this project was to be done in one of two ways. Firstly, attempting to design an image recognition system from scratch creating a convolutional neural network using no external libraries. If this was not achievable the project was to fall back on a premade image recognition or machine learning library. Regardless of which option would be used it would provide an excellent learning opportunity to dive into the intricate details of how convolutional neural networks functioned. Fortunately, I was able to develop a functional system which provided an acceptably accurate recognition and data recording, and so no external image recognition or machine learning libraries were utilized.

The image recognition system in this project involves two separate programs: a behind the scenes desktop program for training image recognition models written in C#, and the main Android app which uses the trained models to translate image data to temperature values over time. The desktop training program will only be used by myself during development, and in future if any updates are needed to the neural network models. This program uses backpropagation to train the models and then after every epoch of training automatically writes to disk a custom filetype which is then to be copied over to the Android app to use in live image recognition.

2.5. Details of Design and Development

Deliverables

There are three main deliverables in this project. A desktop convolutional neural network training program written in C#, a remote NodeJS server to hold bean, roast and blend data for easy access, and the main deliverable, the Android coffee roasting app.

The desktop training program is only meant to be used by myself, the developer, during development to train network models to be used for image recognition later when roasting on the Android app. This deliverable must be an application which takes a dataset of number images, trains a convolutional neural network model to an acceptable accuracy (>90%), and then saves that model for use in the Android app.

The NodeJS server deliverable contains a database to store beans, roasts, and blends. It uses HTTPS GET requests to serve all data on beans, roasts and blends. It also has HTTPS POST requests for each to support uploading.

The main deliverable is the Android roasting app. It contains an SQLite database with which it stores data on beans, roast templates, blends, roast data and roasting machine information. All data in the database is creatable and viewable through the user interface. The data previously mentioned on the NodeJS server can be viewed and downloaded to the local database. Any roast template can be used to do and record a live roast. Live roasting utilizes image recognition to record temperature data from the roasting machine to graph and store for later viewing. The image recognition uses the convolutional neural network model from the desktop training program deliverable. The roasting temperature over time data can be stored to a custom file type and later loaded and viewed at any time. When it is loaded it is graphed for easier viewing. All roast data can be exported to an Excel spreadsheet to be emailed or uploaded to Google Drive.

Feasibility Assessment

There are a few different things to consider when assessing the feasibility of this project and will be addressed now.

Installation of App

This is very simple for someone who is technically minded but may be a hurdle to someone who is not experienced in using the android operating system. As it is right now, the app APK is downloaded from the NodeJS server that is used in this project. The app must then be installed manually by the user. This could be a hurdle to the less technically minded, however first and foremost this app was developed to be used at the coffee company which I work at. Because of this I would be available to assist in installation when needed. In addition, I myself do the majority of the roasting, and will be mostly using the app myself. Secondary to this is the use by outside parties. This will be more difficult as the app is not available on the Google Play store and so they will only be able to even know if it's existence through other users. I was also not able to get in contact with other roasters to test the app, and collect training data on their machines, so it is unknown how well the image recognition will work. After this project it is planned to continue development, and have the app added to the Google Play store. This will greatly ease the installation process for users outside of the company.

Image Recognition

As stated previously I only had access to one roasting machine to collect training data and test on during development. Because of this it is unknown how well the app would work on different roasting machines. The project was developed first and foremost with the company I work at in mind, however ideally this app should work on all machines. Most will likely have a similar output so it is feasible that the neural network will still be able to recognize it's readings, but there is no way to know without getting access to another machine. This is something I am hoping to pursue after this project is complete in order to broaden the availability and function of the app.

Physical Setup

The physical setup of the phone during roasting can be a potential difficulty. For the app to be able to recognize the temperature output of the machine, it needs to be placed in a position where its camera is centered on the output. The most reliable way to do this is to use a tripod with a phone attachment, though it is possible to set it up without additional equipment this adds a hurdle to use. At the company where I am working, we have a tripod for this, and since this is the main target for the app, this is not a problem. Having a tripod set up also takes up extra room around the roasting machine which may be needed by the roaster while he is doing other work. This was not a problem in my experience, but again could be outside of this company. Another potential physical feasibility problem is that the phone will need to be dedicated to roasting during the entirety of a roast. This means that if the roaster requires his phone for other things, it will not be available. I have access to a second phone to dedicate to roasting, so this was not a problem. Most users are likely busy working and may not need their phone during the roast anyway. It is a potential feasibility problem though.

Convolutional Neural Network

Firstly, there will be a higher-level overview explaining how convolutional neural networks work before going into more specific details.

A convolutional neural network is a form of neural network which combines standard fully connected layers with layers of convolution. A basic neural network symbolically contains multiple one-dimensional layers of neurons of varying size. Each of these layers are connected to each other using weights. The first layer is the input, and the last layer is the output. The data from the input layer is propagated forward through each layer of the network being manipulated by the strength of the weighted connections and various functions until it reaches the output. The output is compared with what the output should have been, and adjustments are propagated backwards through the network. As the network learns, the strength of each weighted connection is altered to try and move the network towards a state in which the output is on average closer to each desired output.

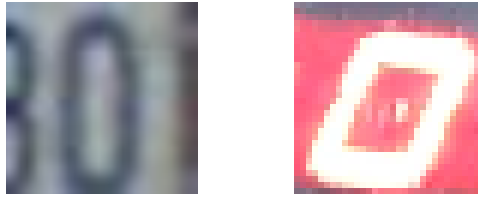
With a convolutional neural network, one or more convolutional layers are added prior to the fully connected layers. These convolutional layers contain different filters whose values are trained in the same way the weights between fully connected layers are trained. Those filters are applied to the two-dimensional image data coming into their layer through a process called convolution. This is the same process which is used to apply an edge detection, or a sharpening filter. Applying these filters to the input helps the network bring out relevant information that helps identify what it is looking for and to connect spatially related data. Since the fully connected layers deal with one-dimensional data, any information about vertically adjacent pixels is much harder to bring out without first having these filtrations.

From here a more detailed explanation of each portion will be given.

Dataset

Through the process of development several different datasets were tried. Some did not end up with an acceptably accurate model, and through trial a hybrid dataset was eventually chosen.

Initially the premade “Street View House Numbers” dataset was used. This set consisted of 32x32 cropped colour images of house number digits 1-9. These digits look quite a bit different from what would be seen on the coffee roaster display, but it was thought to have had enough variation in their presentation that it would translate over to the clearer output of the roaster and fit a wider range of input data. Unfortunately, an acceptable level of accuracy was not achieved using this dataset. This may have been remedied to some extent through different approaches to image preprocessing as the input from the roaster is a lit display having extreme brightness disparity across each image, whereas the house number photos were unlit. The chosen approach to the neural network also only looked at one greyscale colour dimension because of the high contrast that would be present in the live input data. This would be far less effective upon the house number dataset where there is much less contrast between the unlit house numbers and the background of the walls they are on. Additionally, much of the data in this dataset is a darkly coloured digit with a varying, but potentially light background. Whereas the real-world input data would be maximally light digit with a light or dark background, but definitely darker background. Pictured below is a comparison between input data from the “Street View House Number” dataset and a sample of live input data from the roaster.



The next dataset which was attempted was the “MNIST” handwritten digit dataset. This dataset contains 28x28 handwritten monochrome digits. The dataset created a better fit model to the previous dataset but was still not up to an acceptable level for what was required. Although this set was much more like what would be seen when the app is live, there were still some potential reasons why it did not produce an acceptable model. The colour problem from the previous dataset is not present in this one, as well as much of the background noise which likely helped attain the better functioning model. The digits contained are monochrome and white with a black background. So, after converting real images to grayscale, these are much closer to what the app would be presented with. However, there are still obvious problems with this set compared with what is presented during roasting. The largest difference is the way in which handwritten numbers look when compared to the 7-segment style led display which is used on roasters. The digit readout on the roaster is made up entirely of straight lines and right angles. This differs from the handwritten dataset’s more organic curves and handwritten curlicues. There are also differing ways in which numbers can be handwritten when compared to a 7-segment digital readout. Some examples of these differences are pictured below. Although obvious to a human, each one of these example digits differs greatly from what would be seen by the app. Mind you, not all differ this greatly, but these are examples where they are quite different symbols than what would be seen.





Finally, the last dataset attempted was a combination of MNIST and images captured from live roasting. The reason for the combination was to both try to keep the model from being overfitted to the one roasting machine, and also to extend the amount of training data. Manually collecting training to a large enough amount is unfeasible in the time requirements of this project, so the MNIST set was used as filler to bring it up to a large enough training set. In order to further increase the quality of the training set, the app was designed with a hidden collection mode where it would save every image in a folder labeled which digit the app guessed it was. After a roasting session I would go through the folders manually finding incorrectly guessed digits and adding them to the dataset, would retrain a new model and then load it into the app. Using this approach, an acceptable accuracy level was finally achieved. Later a tenth output was added which was meant to represent when a number is not found. The input data used for training it to recognize when it is not a number is an image made of random pixel data generated on the fly. It was not certain that this would work as many things that could be seen have more structure than white noise and would possibly look closer to what it thinks is a number, but it ended up working quite well. Potentially this was enhanced by the preprocessing that is done to the input images.

Input Data

Input data is handled differently between the two programs involved in the image recognition system in this project. Since the two programs have different objectives (the desktop trainer to train, and the Android app to predict) their input data is handled in ways suited to their uses.

For the desktop training program, the data is not preprocessed for clarity on the fly. Instead, any preprocessing cleaning of data is done one time and the file is saved to disk to be used as is from then on. This program had two separate datasets to preprocess, the MNIST dataset and the dataset that was created for this project from recorded roasting images. The MNIST dataset comes in a format suitable for use with python machine learning libraries. To load this file the MNIST.IO C# library was used. Unfortunately, this was found to use a huge amount of memory and had problems running alongside the neural net training. Instead, a separate program was created to be run one time and convert all the images contained in the MNIST dataset into 32x32 PNG files to be loaded on the fly during training. As for the data collected while roasting, it has the preprocessing from the Android app applied to it before saving, so it matches that of the app which is explained in the next paragraph. The preprocessing which is done in the training app during runtime is instead meant to reduce over fitness from the limited dataset (Fitness is how well the model fits the training data. If it becomes overfit it will not be able to handle novel data in the real world well). This involves applying randomized manipulations in size and

positioning of the data at random intervals prior to input into the neural net. Manipulating the data in this way improved the accuracy, increasing the acceptable positioning of the camera. These modifications are applied to every fifth input image.

Neural net input data on the Android app is in the form of images which have been taken from the built-in camera hardware. In order to get the camera image first camera access permission is required. This is checked and requested dynamically if it is not already granted. Once permission is granted the app gets a reference to a Camera object through the Camera.open() function. The camera thread is then started and run every 100ms. The camera thread creates a new PreviewCallback which is set to the camera object's preview callback. In this new PreviewCallback class the onPreviewFrame function is overridden. This function passes a byte array representing the camera data in YCbCr. YCbCr is different from the usual RGB which we are more used to. In this case Y represents the luminosity, Cb represents the blue difference chroma and Cr represents the red difference chroma. Although for this application I was only interested in the luminosity, it was still converted to RGB using built in functions for ease and visual debugging. The full-size image is loaded into the ImageProcessing object where it is cut into relevant subsections for each of the three temperature digits (hundreds, tens and ones). A cut-out line is used to make it clear where the user must position the digits in the camera. These three images are then converted from RGB to just the Y (luminosity) value of YCbCr and then filtered according to the brightness settings chosen in the calibration activity. This converts the image bitmap to monochrome and converts all brightness levels to maximum or minimum using the aforementioned brightness setting as a threshold for what is considered bright enough to be kept. This method was chosen because of the extreme contrast of output on the roasting machine allowing this simple method to remove most noise from the image prior to being fed into the neural network. Pictured below are a few images post sub sectioning and filtering.



Isolated images post-filtering



Below is an example of a bad brightness setting



The final preprocessing that is done to the image data is horizontal translation between averaged network predictions to mitigate any misalignment of captured images.

Specific Details and Implementation

The implementation that was used involved two separate programs related to the neural network. There is a desktop model training program coded in C#, and the actual android app which utilizes the model for predicting temperature values.

The convolutional neural network structure, forward and backpropagation algorithms are the same between the desktop and Android app aside from differences in the C# and Java languages. Therefore, an in-depth explanation of them in the desktop version will be explained, and then any parts added in Android will be explained subsequently.

network structure

The network structure evolved somewhat during development, but work was mostly focused on other areas. The initial setup of the network was based on what had been seen in similar networks during studying. The largest modification was when an additional output neuron was added to represent the absence of a number. After this the number of neurons in the final two fully connected layers was doubled. This worked out well, so no further adjustments were made to the structure of the network. Surely what was chosen is not the optimal structure, but there wasn't sufficient time to devote to any more tweaking of structure and what was chosen worked sufficiently well. The final structure in order is:

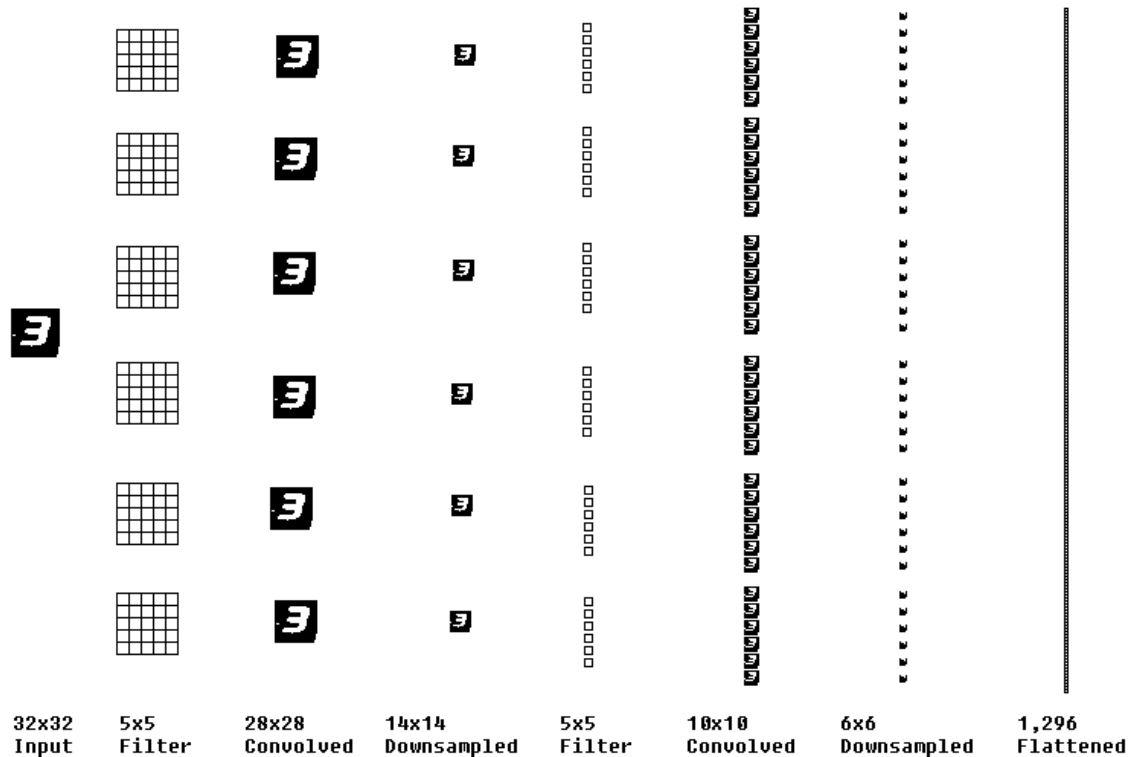
2 layers of convolution with down sampling to half size.

After convolution a 1,296-neuron flattened layer.

2 fully connected hidden layers containing 240 and 200 neurons respectively.

Finally, there are 11 output neurons representing digits 0-9 and a tenth for when a number is not found.

Below is an image to aid in visualizing the convolutional layers ending with flattening.



Network Initialization

Before the network can be trained it needs to be initialized. First the network structure is instantiated according to the settings described previously. Once it is instantiated, the weights, biases, and filters must be set to their random initial values. This value was originally calculated using a function written to generate a gaussian distribution (bell curve) of randomized output centered around 0. This for some reason was causing exploding gradients and NAN outputs. I couldn't figure out what the problem was here. Whether it was in my code to generate a gaussian distribution, or a problem somewhere in the convolutional neural network code. The gaussian function occasionally output numbers that seemed quite high, which may have been the cause of the exploding gradients. Either way, this function was substituted for one which simply generated a random float between -0.5 and 0.5. Perhaps a set of initial weights in a gaussian distribution may have ended in faster training, but this worked well and caused no problems, so it was used instead.

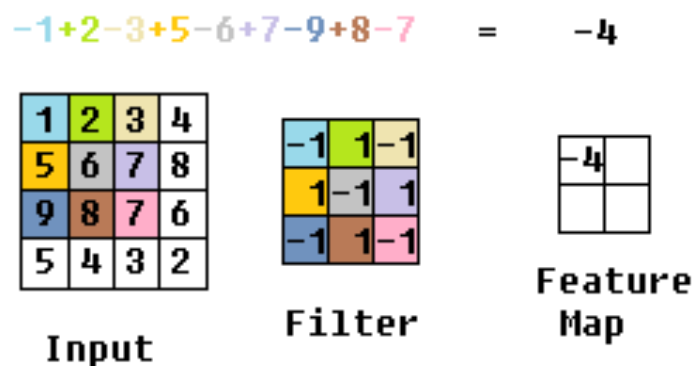
forward propagation

Once the network was initialized, training data was run through it in the same way it would once it is used to classify images of digits. First an image is selected to run through the network for training. A number between 0-10 is picked randomly. If the number 10 is selected, this is meant to represent a situation where the network does not recognize a number. For this training input an image consisting of random black and white pixels is generated. Each pixel has a 50/50 chance of being either fully black or fully white. For the other ten possible numbers that could be picked (0-9), a random image will be

selected from the folder under that label. Every tenth time an input is picked it will instead be an image randomly selected from the MNIST dataset.

convolution

The input of the neural net is set to the image, which is loaded for training, and then it is forward propagated through the network. The first step is applying the first layer of filters. The filters are a smaller matrix of filter values. These filter values are initially randomized from the previous initialization step and will gradually be fine tuned in later steps. The filters are convolved across the input generating new images which are the next convolution layer. The process of convolution can be thought of as overlaying a filter on top of an image and then multiplying pixel values with filter values where they line up. All these multiplications are summed and make the corresponding value in the image in the next layer. In the example below, each matching colour is multiplied to make the coloured numbers above which are summed to make the corresponding value in the feature map.



The filter is then shifted over one pixel to the right on the input matrix and the process is repeated until the horizontal edge of the image is reached. Then the filter is returned to the beginning but shifted down one pixel until the entirety of the image has been filtered. You may optionally pad the edges with zeroes, so that the filter begins centered on 0,0 instead of 1,1 and then surround the input image with pixels equal to the filter width-1 with a value of 0. Doing this will create a filtered image of the same size as the input image. If padding is not used, the convolved image will be 2 pixels smaller in width and in height. In this application there is not any real valuable data near the edges, and a smaller size is preferable, so no padding was used.

After the first filter is applied the 32x32 input image will be output into the next layer as a 28x28 convoluted image. Since the filter is 5x5 this will subtract its width-1. So, $32 - 4 = 28$. After this a constant called a bias is added to each pixel. Now that each pixel has been generated through filtering and adding their biases the next step is to run this value through a chosen activation function. For this step the rectified linear unit (ReLU) was used. ReLU is meant to cause non-linearity in the data and seems to be the current standard activation function for convolution. This function also helps mitigate the vanishing gradient problem that can occur during training in deep neural networks. This is when the error becomes so small as it is propagated deep into the network that it has almost no effect on training.

pooling

The pooling step is used to reduce the amount of data which the net will be required to deal with. It also allows for greater spatial invariance so that the positioning of the input data has a lessened effect on the potential recognition. The standard pooling method, and the one which will be used in this project, is max pooling. This involves taking a subsection of data, comparing the values in that subsection and only taking the maximum value among those values. The positioning of the subsection is then moved based on its stride and the process is repeated. Similar to how the filtering is done. In this case the subsection is 2x2 and the stride is set to 2, so after pooling the new image will be half the width and half the height of the original with only the strongest features coming through. An example of application of this method would be as follows:

1	5	4	1
8	5	4	8
6	3	1	9
5	7	1	2

After Pooling

8	8
7	9

After pooling a second round of convolution, ReLu and pooling are done upon the output from the previous layer. In this layer the convolution is somewhat different from the first layer in that multiple filters are used to create a single feature map.

Once the convolutional layers are complete the data is converted from 2d images into a 1d input array suitable for the fully connected neural network. This step is called flattening.

1	2	5	6	9	10
3	4	7	8	11	12

3 x 2x2 Images

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

1x12 Array

This portion is the part which is the same as a normal neural network, containing weights, biases and neurons. It is made up of a set number of hidden layers containing a set number of neurons and a final layer of output neurons which in this case will represent the integers 0-9 plus an eleventh output representing when the network sees anything other than a digit. This project was designed with 240 neurons in the first fully connected hidden layer, and 200 neurons in the second. Each neuron will be connected to every value in the previous flattened array by an initially randomized weight value. So, with 1,296 flattened input neurons connected to each neuron in the first hidden layer, there are $1,296 \times 250 = 324,000$ weights connecting the two layers, $250 \times 200 = 50,000$ weights connecting the two hidden layers, and $200 \times 11 = 2,200$ weights connecting the second hidden layer to the final output layer. For each neuron the program multiplies each of its weights by their corresponding input value and then sums them. Add to this the bias constant for that neuron and then put this value through the chosen activation function, in this case the sigmoid function, and then you end with the activation for that neuron. This process is repeated for each successive layer, using the activations of the previous neurons as the input for that layer. Once you have calculated the final layer of hidden neurons the output is calculated. This is done in the same way as the other layers. These final neuron's activation values represent the network's confidence in each number being the number shown in the input image. The highest output being its guess, which after training should be as close to 1.0 in the desired number and 0.0 in all others as possible.

At this point forward propagation is complete and the outputs have been obtained. From here the program moves to backpropagation.

backpropagation

Backpropagation is the process of propagating the output error (loss) backwards through the neural network and calculating each point's contribution to the loss, then this contribution is used to adjust towards the desired output. The algorithm making the movement of parameters towards the optimal output is called gradient descent, which adjusts parameters of the network towards an error minimum. Since no machine learning libraries were used, all the math involved had to be researched and coded.

The first step in backpropagation is calculating the error. There are various loss functions which can be used to do this. Initially the SoftMax and Cross Entropy functions were tried. SoftMax is used to convert each individual output to a percentage of total outputs. Then cross entropy is used to calculate the loss of the output. Using this method to calculate loss led to exploding gradient problems, and when the learning rate was reduced to a level that prevented infinities from being generated the network could not achieve any learning or reductions in loss. This was some mistake or misunderstanding on my part, but I could not find what it was. Next the average squared deviation function was used, and learning was finally achieved.

$$\sum_0^n (O_n - T_n)^2$$

This loss function is simply squaring the difference between the desired output and what was produced. Squaring it gives greater weight to larger errors and causes the training to take larger steps at the beginning when the error is greater and to slow down as it approaches the correct answer.

Using the error backpropagation can be calculated. In order to find how much each parameter needs to be adjusted, derivatives must be used to find how much of an effect each parameter has on the output. There is a very large number of derivatives which must be kept track of during backpropagation, which was handled by creating a second neural network object, but instead of holding neuron activations and weights it holds all the derivatives in their relevant positions.

Finding these derivatives is done by starting with the cost function and using the calculus chain rule to work the error back to every adjustable parameter in the network. The chain rule is a formula for calculating the derivative when the output is composed of multiple functions. Say we are trying to find the derivative of one of the weights feeding into the output, this would involve finding the partial derivative of the total error with respect to that weight. To find this the chain rule is utilized to find the derivative of the total error with respect to the output multiplied by the derivative of the output with respect to the net input to that neuron, multiplied by the derivative of the net input with respect to the desired weight.

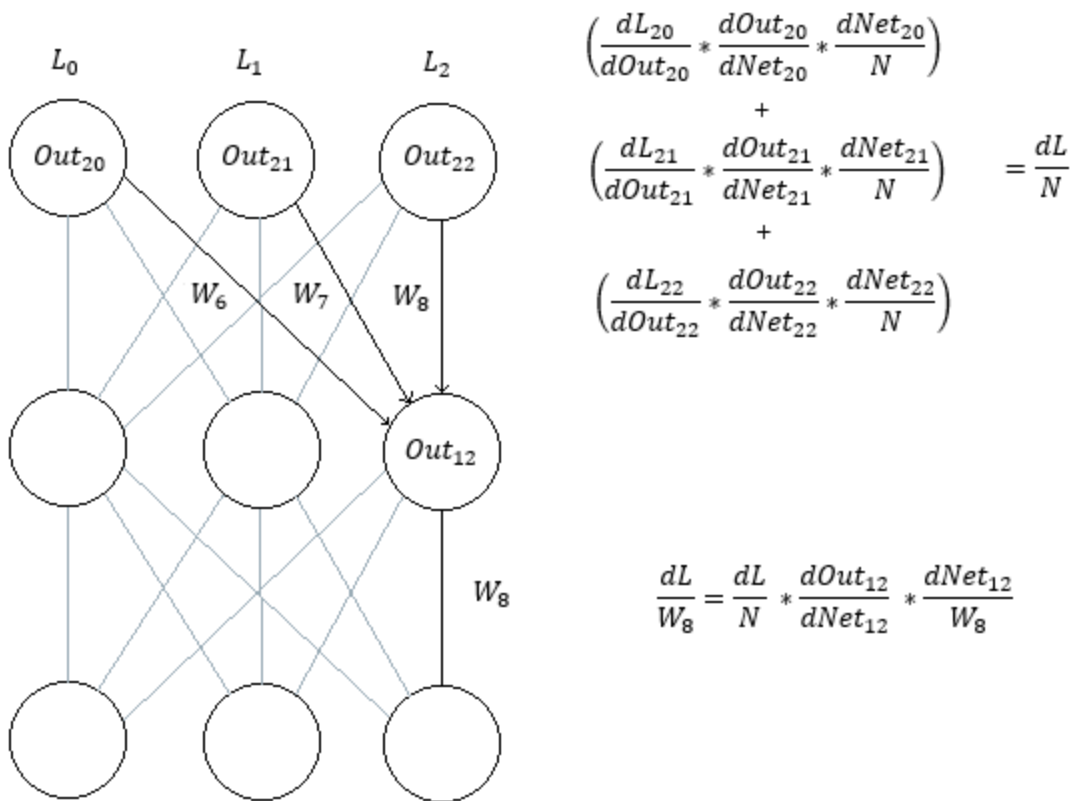
$$\frac{dL}{dW} = \frac{dL}{dOut} * \frac{dOut}{dNet} * \frac{dNet}{dW}$$

The program breaks each of these down to individual derivatives and then multiplies each answer together. It then subtracts this number from the weight to move it towards the desired output. Before subtracting, this value is multiplied by a learning rate to either increase or decrease the strength of this adjustment. For this project many different learning rates were used at different points. For much of development a learning rate of 0.002 was used. However, mistakes in the training code were later found

which, after being fixed, made this an ineffective rate. The final chosen rate was that of a dynamic rate starting at 1. After every training batch is completed the training rate is slightly increased (0.00001). This was found to help counter some slowdown as the model approaches a correct answer and the gradient descent has a shallower slope. When making this adjustment in a real training environment, such as this project, these adjustments are averaged over a batch of training examples. In this project a final batch size of 16 was decided upon. Initially it was 32, but it was found that smaller trained faster. Perhaps this is because this number is closer to the batch size. By averaging like this the network can be trained on many different examples from a given state, which allows it to move towards the best average parameter position. Otherwise, it would see a number, call it x , adjust towards that number and then on the next example, call it y , move towards y and move away from x . If it is trained in this way, it will only ever focus on training a single output and never get any better at any of them. By using the averaging technique, it can see several, adjust to recognize those better and then reassess from that position to what the next best adjustment would be for the whole of what is being learned.

This example also ignored a bias, for simplicity. If there were a bias on the output neuron, the same steps would be applied, but you would skip the last derivative in the chain (net with respect to the weight). Since the bias is additive rather than multiplicative the final derivative that would take its place is 1, so only the first two are used.

There are two more things which must be added to what is explained to train the hidden layers. The depth of multiple layers, and the breadth of each layer's weights and neurons. If this same example above had multiple neurons feeding into multiple outputs with weights connecting the two layers the chain rule must be done for each weight into each neuron.



Then to move to calculating the previous layer's weights (In this case W8) the program sums the error coming in through all the weights affected by it (W6,W7,W8), but in this step before trying to find the derivative of the total error with respect to the weight, you want the derivative of the total error with respect to the neuron. The equation is mostly the same as the previous chain, but with the derivative of the neuron as the last equation and summed across connections.

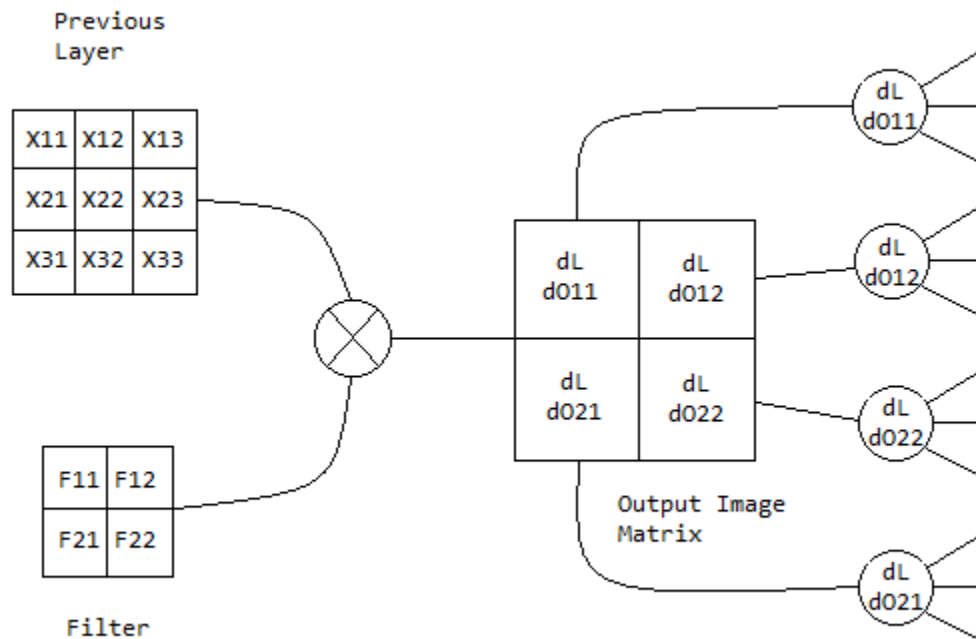
$$\frac{dL}{N} = \sum_0^n \left(\frac{dL_n}{dOut_n} * \frac{dOut_n}{dNet_n} * \frac{dNet_n}{N} \right)$$

From here, this is chained into the desired weight on the next layer:

$$\frac{dL}{W_n} = \frac{dL}{N} * \frac{dOut_i}{dNet_i} * \frac{dNet_i}{W_n}$$

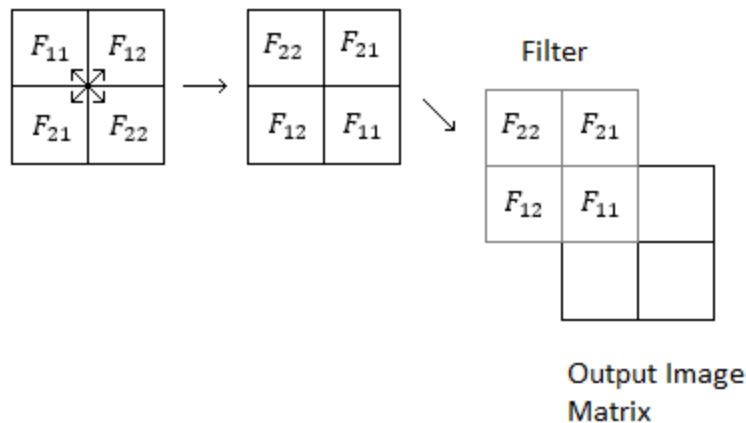
The network structure used contained two hidden layers, so the weights connected to the outputs are calculated, then the error is carried to the last row of hidden neurons summing the error and storing it in each neuron in the error network object. Steps are continued, iterating through the weights between the hidden layers, the first hidden layer, and then the weights to the flattened layer calculating the adjustments to make, multiplying them by the learning rate, averaging them by the size of the batch and then adding them to the network object containing adjustments to be made at the end of the batch.

The next step is calculating the derivatives for the convolutional filters. This is quite a bit different than the way the hidden layers are calculated. For this step the flattening step at the end of the last convolution is undone and the one-dimensional flattened layer is put back into the image structures, but with propagated error values in place of the pixel value. There are three portions to each convolutional layer in this project that the loss had to be worked through: max pooling, rectified linear unit, and then the convolutional filtering.



The easiest way to explain how to think about finding the adjustment values for these filters is to visualize taking the output image matrix of derivatives and apply it as though it were a filter to the previous layer values.

Then to find the loss at the input for this layer so that you can carry on to the next can be visualized by crossing the filter values and then applying them to the loss output image matrix with one space of padding:



Then these X values are used as the derivatives of the loss with respect to the output on the next layer. After doing two rounds of this, the program has covered every part of the network and backpropagation is complete.

To keep track of everything the program uses three convolutional neural network objects to hold the data. One is the actual network containing all the weights, biases and activations during forward propagation. The second is used for loss storage as was mentioned previously. So, all the derivatives as they are calculated backwards through the network are stored here instead of the normal weights, biases and activation functions. They are the adjustments to be made in that one training example. The third network object is used to store the averages across the training batch. Every time a weight or bias adjustment is calculated the value is multiplied by the learning rate, then divided by the size of the training batch, and then added to this object containing the averaged weight adjustments. In dividing by the size of the training batch the adjustments are averaged over the batch. Then, when a batch is complete the network of averaged adjustments are subtracted from the main neural networks weights and biases, and the network of averages is reset for another round.

Backpropagation is continued until an epoch is completed. An epoch is one full pass through the training data. Once an epoch is completed the model is tested against separated test data and the accuracy is printed. The model is then saved to a custom file type to be loaded into the Android program for real world use. The program continues through epochs indefinitely, overriding the model file each time until the user manually stops the program when an acceptable accuracy has been reached.

In summary, the desktop training app has the following steps. First the network structure is created and the weights and biases are randomized to values between -0.5 and 0.5. It then selects an image to train. If it is the fifth selected image it makes random manipulations to reduce overfitting. That image is loaded into the convolutional neural network class. The image data is forward propagated through the network. Its error is calculated based on the desired output. This error is back propagated through every layer of the network keeping track of these values in a temporary network structure. Every time the error calculation reaches an adjustable weight or bias it is averaged and added to another temporary structure representing parameter modification averages. This is repeated until one batch is completed, at which point the averaged modification network is subtracted from the convolutional neural network moving it slightly down the descent gradient towards a more accurate model. This is repeated until one

full epoch is completed. At this point the program tests the model against separate test data, and then saves the model to a custom file to be imported into the Android app. The program continues training through epochs until a sufficient accuracy is reached and the user terminates the program.

image recognition postprocessing

Even with very high accuracy of the convolutional neural network there are still some errors. Because the temperature is being graphed, any single error can stand out quite a bit. In order to diminish this even further there are some post-processing techniques applied to the temperature data which is output from the neural net. firstly, for each temperature readout the input image is processed, shifted left, processed again, shifted right and processed a third time. This is then done eight times, and the highest guess from all 24 of these predictions is used. The horizontal translation mitigates is an attempt at improving translational invariance, and the multiple attempts buries most anomalous readings. Then after this, a median filter is applied to the data array to filter out bumps in the data. The median filter iterates through each value in the array, copies a subsection of data in a 100 point window centered on the current value, sorts the temperature values within that window from lowest to highest, then takes the median value and adds that to a filtered array.

Example readings array

1	2	3	40	5	6	7	8	9	10
---	---	---	----	---	---	---	---	---	----

Sorted sample subsample

2	3	5	6	40
---	---	---	---	----

New filtered array

1	2	3	5						
---	---	---	---	--	--	--	--	--	--

The final filtered array is what is used to make the graph that is displayed to the user. The window size used in this technique required some testing and adjustment. The larger the sample window, the better smoothing was achieved, however the larger the window the more curve data was lost from the low point of the roast. Since the temperature drops at the beginning of the roast before it begins increasing in temperature, this is also filtered out with a large enough sample window. 100 samples were found to be a good balance of smoothing and data preservation and was what was used in the final build.

Image Recognition Calibration Activity

Although it wasn't planned in the proposal, it was realized during development the necessity of a camera calibration activity to deal with changing visual environments. This was noticed when the

company I work at moved locations and the image recognition suddenly did not work due to different lighting conditions. This was an unexpected benefit as I did not have access to multiple roasting machines to test the app on, but at least in this case it had multiple lighting conditions. This functionality was something I likely would have overlooked if this did not present itself.

The calibration activity works by utilizing the same classes that the roasting activity uses to run the image recognition. However, this activity uses a dynamic brightness variable when filtering the input images. Starting the brightness multiplier at 0 it increases it by 0.1 every half second. For each brightness setting the preprocessed image is displayed so the user can judge clarity. The image recognition's guess for that brightness is written on a series of buttons at the bottom of the screen. The number guess is written on the next button in the series each time the brightness is increased, and the current brightness is highlighted in bold. This way the user can also see which guess is correct and select that brightness setting. Once the brightness is selected it is stored in the global settings object which also saves the settings to disk each time they are altered. In this way the user's brightness setting is saved between sessions.

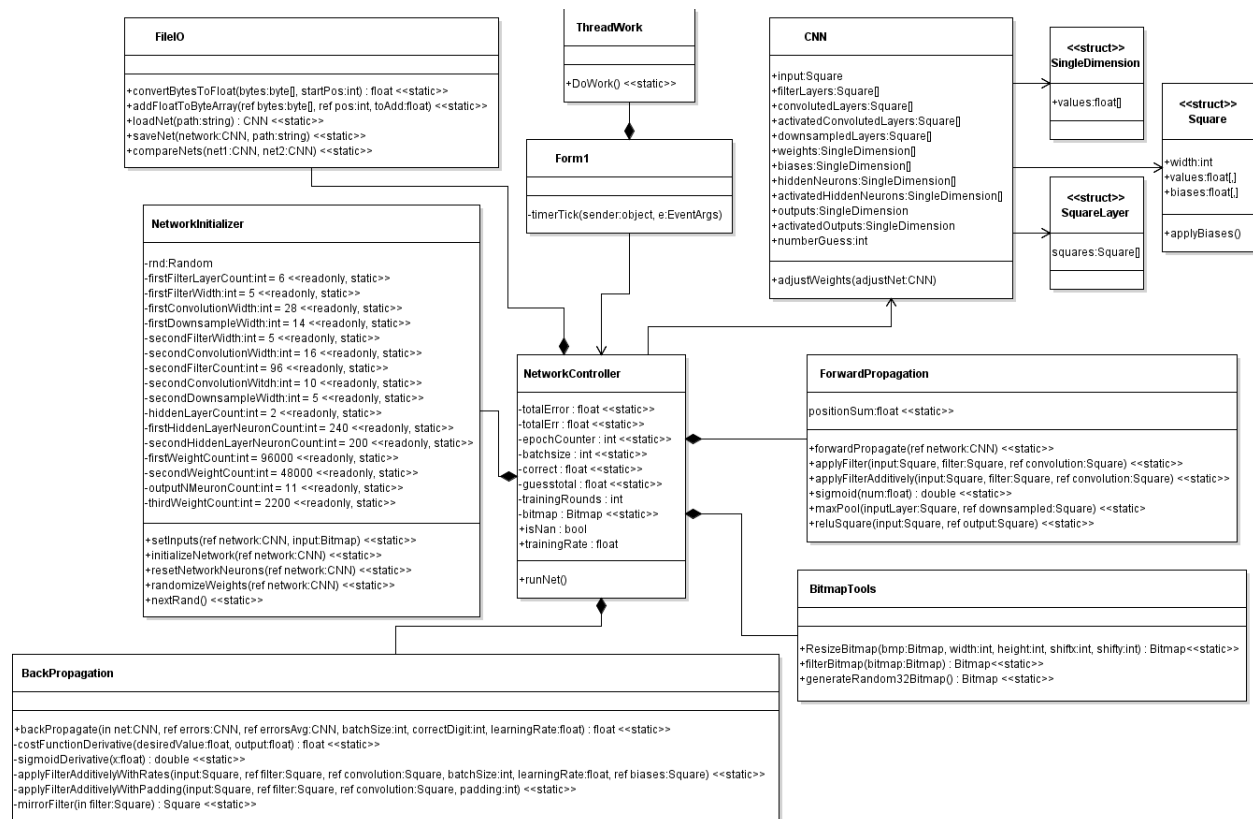
Custom File Types

There are two custom file types which are used in this project. One is for storing the convolutional neural network model to transfer between the desktop training program and the Android roasting app. The other is for storing the large sets of roasting temperature over time data.

The neural net model file type starts with one byte representing the version number. This way the Android application can have a switch statement on loading so that it can be backwards compatible with earlier models if some structural change is made. Next, the program creates a byte array equal to the size of every weight and bias multiplied by 4 (byte size of a float). It then converts every weight and bias in the model from floats to bytes in a specific order and adds them to the byte array. This order is maintained on loading with the Android application so that all the weights and biases are put in the right place. The training program loads the net from the file after saving and compares the original model to the loaded model and throws an exception if a difference is found.

Class Diagrams

Desktop Training Program



Form1: User interface class. Starts **NetworkController** in a second thread on button press. Updates UI output on timer.

NetworkController: Controls everything to do with the neural network. Initializes CNN, sets inputs, forward propagates and backpropagates. Keeps track of current errors and network of adjustments for each batch.

CNN: Convolutional neural network structure. Holds all information about the neural network such as weights, biases, filters. Also used as separate object to store propagated error. Has function **adjustWeights** to subtract averaged adjustments after a training batch is completed.

SingleDimension: Represents a single dimensional layer in the network, such as the hidden layers.

Square: Represents a square of data, such as an image or a filter.

SquareLayer: Represents a layer made up of Squares.

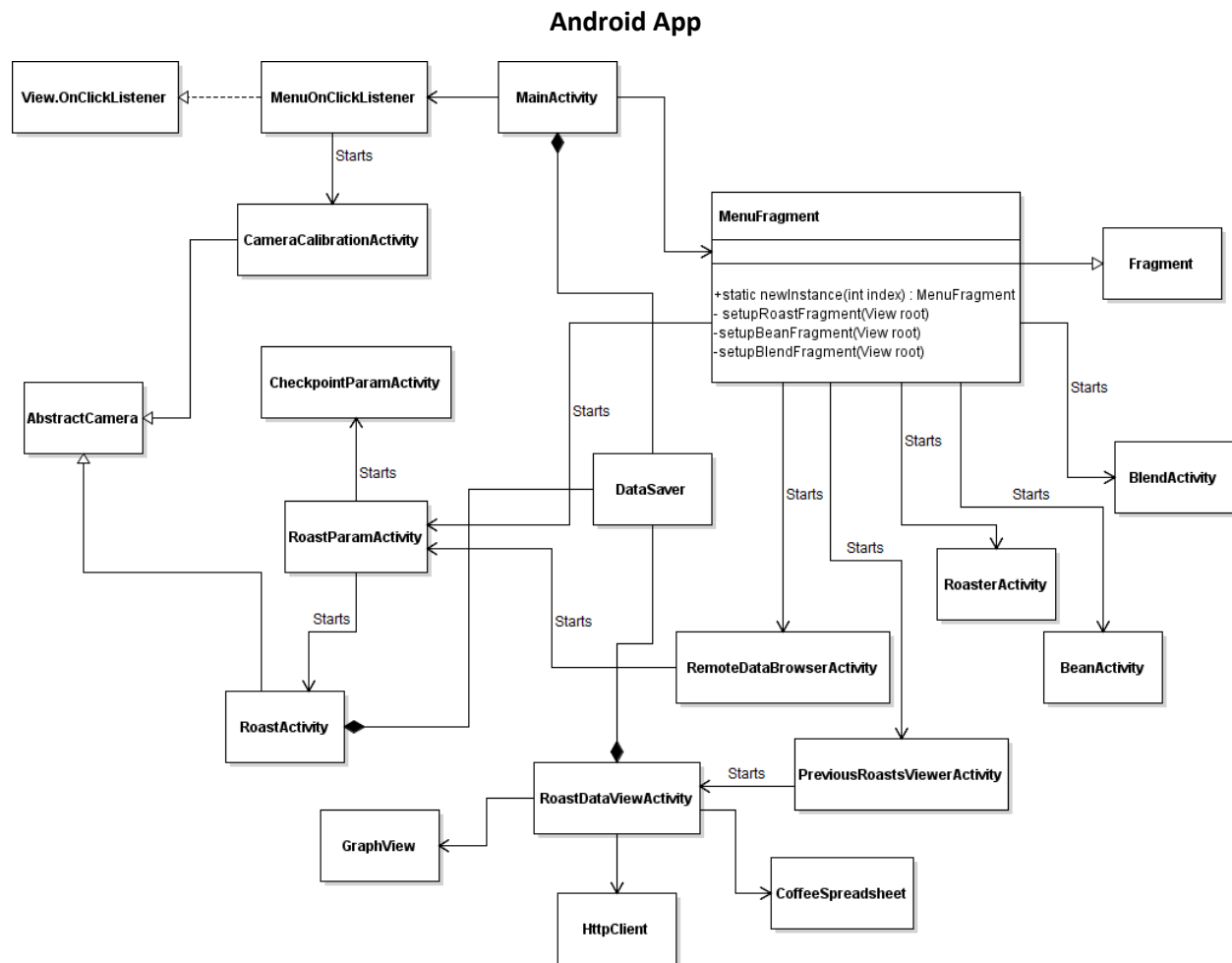
NetworkInitializer: Initializes a passed CNN object according to the set structure and then randomizes weights and biases.

ForwardPropagation: Class that handles propagating input through a passed CNN object that has been initialized. Has functions for all the activation functions and convolutional filtering that is utilized.

BackPropagation: Handles backpropagating error through a CNN object that has been forward propagated. Contains functions to handle all derivatives that are required.

FileIO: Controls file input output. Has functions for saving and loading a CNN object to a file according to the custom file type. Has functions for converting floats to and from bytes and comparing a loaded network to the current network to check for errors.

BitmapTools: All static image manipulation functions that are used on input images.



This class diagram is focused on the high-level structure of the Android app. Showing the different activities and their connections. Areas requiring further focus have their own diagrams only showing classes which are connected to that individual activity.

Most of the classes don't require further explanation that aren't later expanded upon. A few of note in this diagram are:

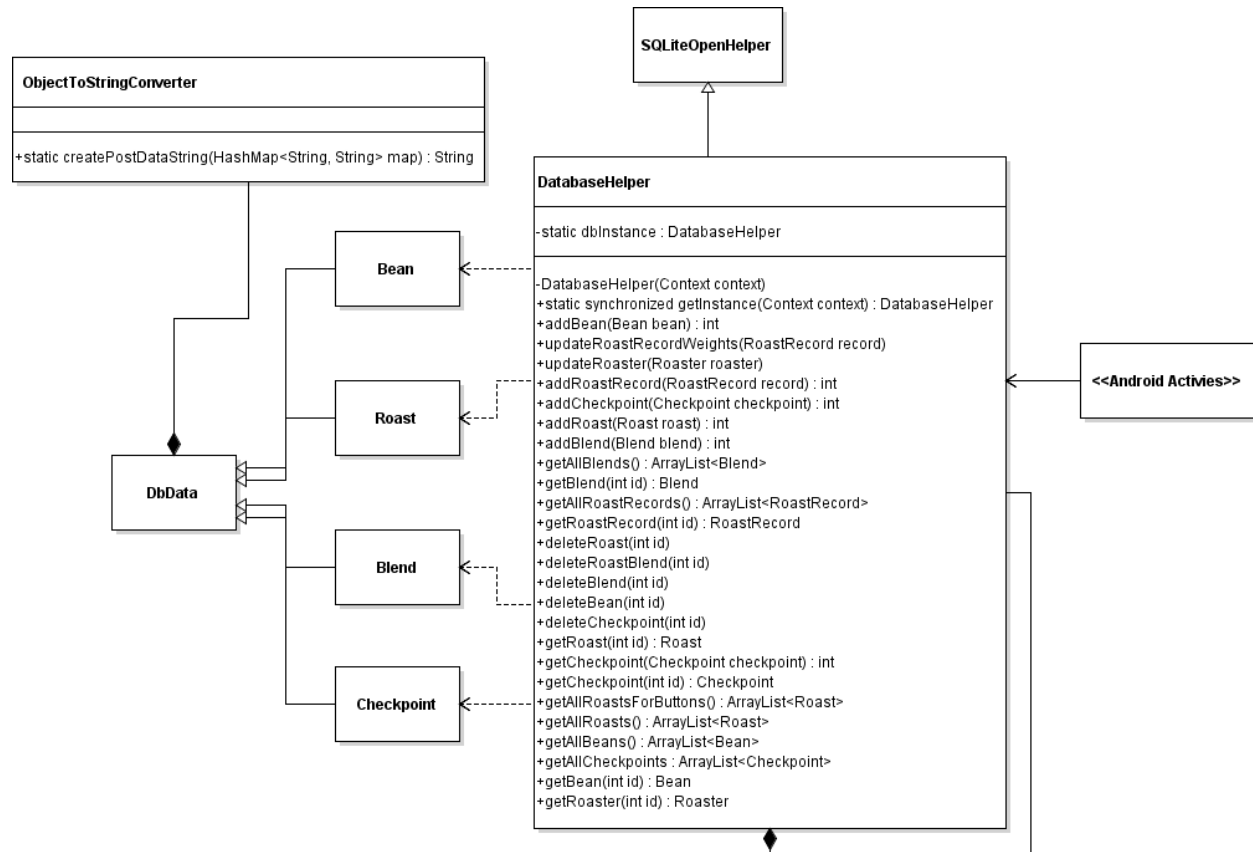
DataSaver: Static class which handles saving and loading of roast data files.

MenuOnClickListener: The UI dropdown settings menu that is accessible from most activities. It is only shown as connected to the **MainActivity** on this diagram, however it is also used by the **RoasterActivity**,

BeanActivity, BlendActivity, RemoteDataActivity and the PreviousRoastsViewerActivity. It wasn't feasible to connect all of these without over cluttering the diagram, so the connections were not shown.

AbstractCamera: Handles all camera hardware access.

GraphView: Controls display of data graphs on the UI.

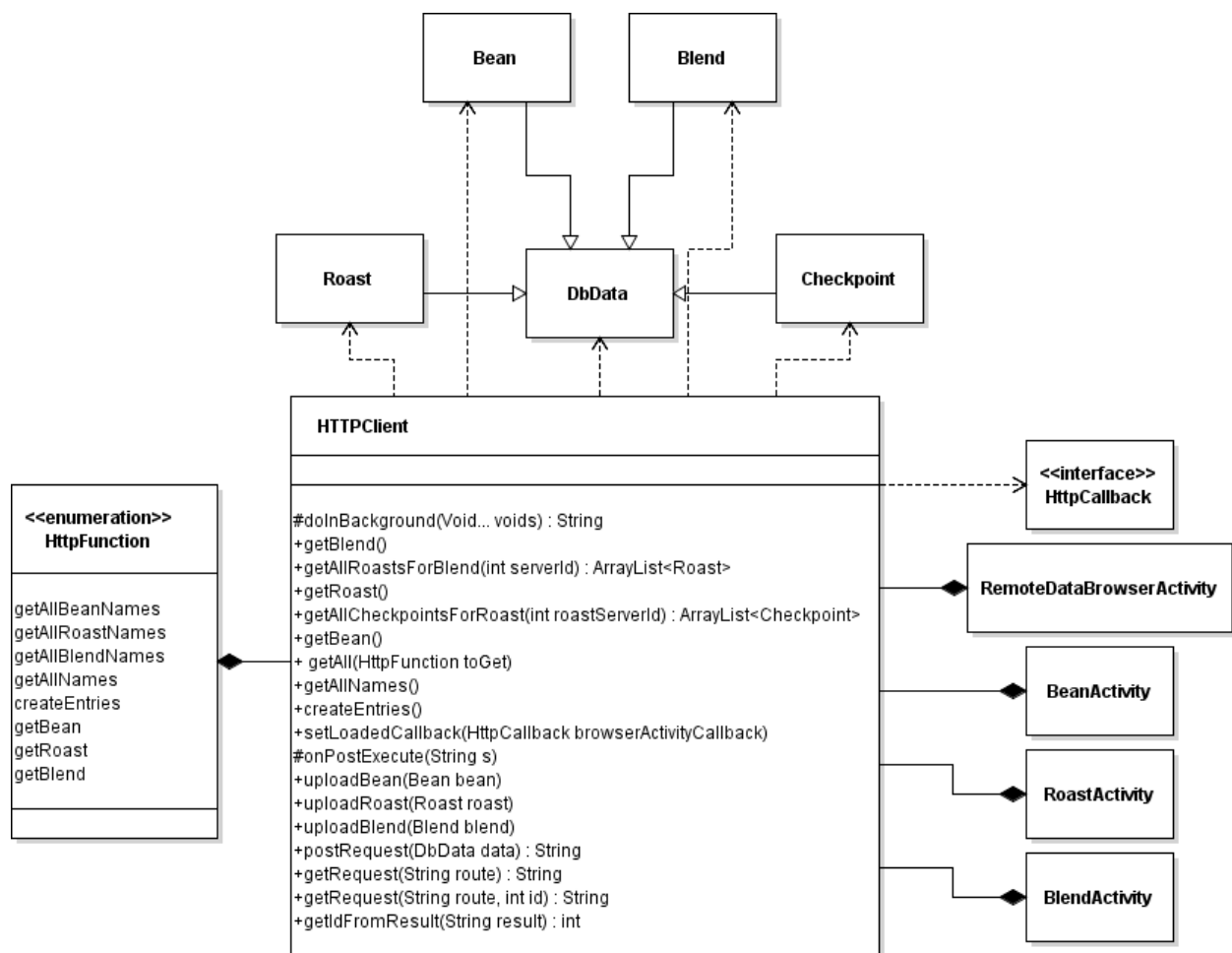


This is a zoomed in look at the DatabaseHelper class. This class was left out of the activities diagram as it is referenced by the MenuFragment, RoastParamActivity and RoastDataViewActivity.

DatabaseHelper: The main static class in this diagram which is referenced by many activities and controls all database access. It is a singleton to ensure that only one copy is instantiated at any time and to make it easily accessible to other classes all over the app. It uses several classes to represent tables in the database so that data can easily be passed between them.

Bean, Roast, Blend, Checkpoint: Represent their respective tables in the database and the data contained therein.

DBData: abstract class to be extended by the previously defined data classes. Contains standard information that will be in all database entries. Is also used for passing information to the NodeJS server.

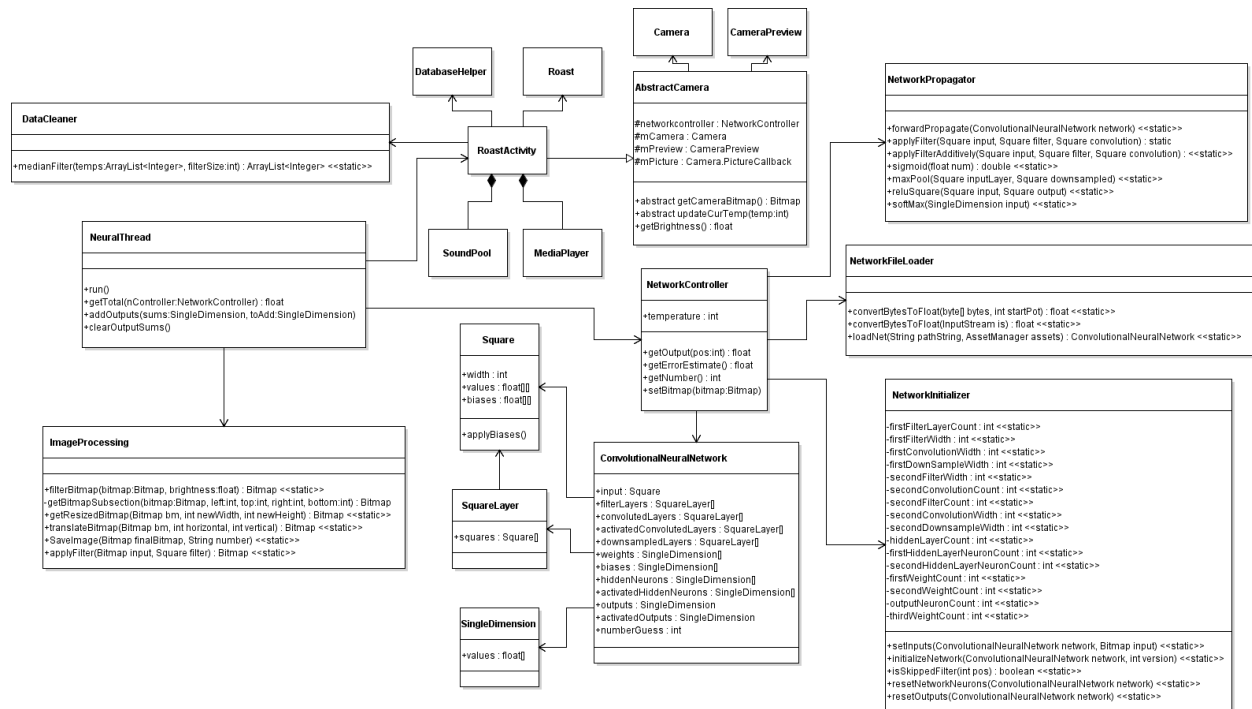


This diagram is focused on the client portion on the networking side of the project. It reuses the same data structure classes as the database section (**Bean**, **Blend**, **Roast**, **Checkpoint**) extending the **DbData** subclass.

HttpFunction: Enumeration containing the names of all server functions which the app passes to the **HttpClient** to communicate the desired function.

HttpCallback: Callback used to communicate when data has been loaded from the server to an activity that has extended **HttpCallback**.

HttpClient: Used by activities to download or to upload data to the server. It is used by setting the id of the specific item to get (if applicable), set the desired function from the **HttpFunction** enum, passing an activity that has extended **HttpCallback** to the `setLoadedCallback` function and then calling the `execute` function to start the thread. When the data has been loaded from the server the activity's callback function will be triggered from which it can display the data in the UI or whatever else is necessary.



This diagram zooms in on all the classes involved in the RoastActivity where most of the image recognition and all the actual roasting is done. The other class utilizing the image recognition is the CameraCalibrationActivity which also extends the AbstractCamera class and uses the NeuralThread class as well.

SoundPool, MediaPlayer: Android classes that are used for playing the alarm sound effect when a roast checkpoint is approaching.

DataCleaner: This class holds the median filtering function used for filtering out any anomalous spikes in temperature data before applying it to the graph. If further filtering techniques were used, they would have been held here.

NeuralThread: This is the thread object which controls all the convolutional neural network's function. It is constantly checking if the input image has been updated, and when it has runs it through the neural network, then saves the temperature prediction in the RoastActivity object.

ImageProcessing: A class which contains all the static functions used for bitmap manipulations.

NetworkController: The class which controls the convolutional neural network. Passes the neural network around to be initialized and propagated forward and back. Sets inputs and reads output from the network object.

ConvolutionalNeuralNetwork: Holds the structure of the neural network. All weights, biases, convolutional filters, neuron activation values, inputs, outputs, and the connections between all of these.

Square, SquareLayer, SingleDimension: Structures used to enhance readability during development. Square is basically a matrix used to represent image or filter data, SquareLayer is an array of squares, and SingleDimension is used to represent a layer of weights, biases or neurons.

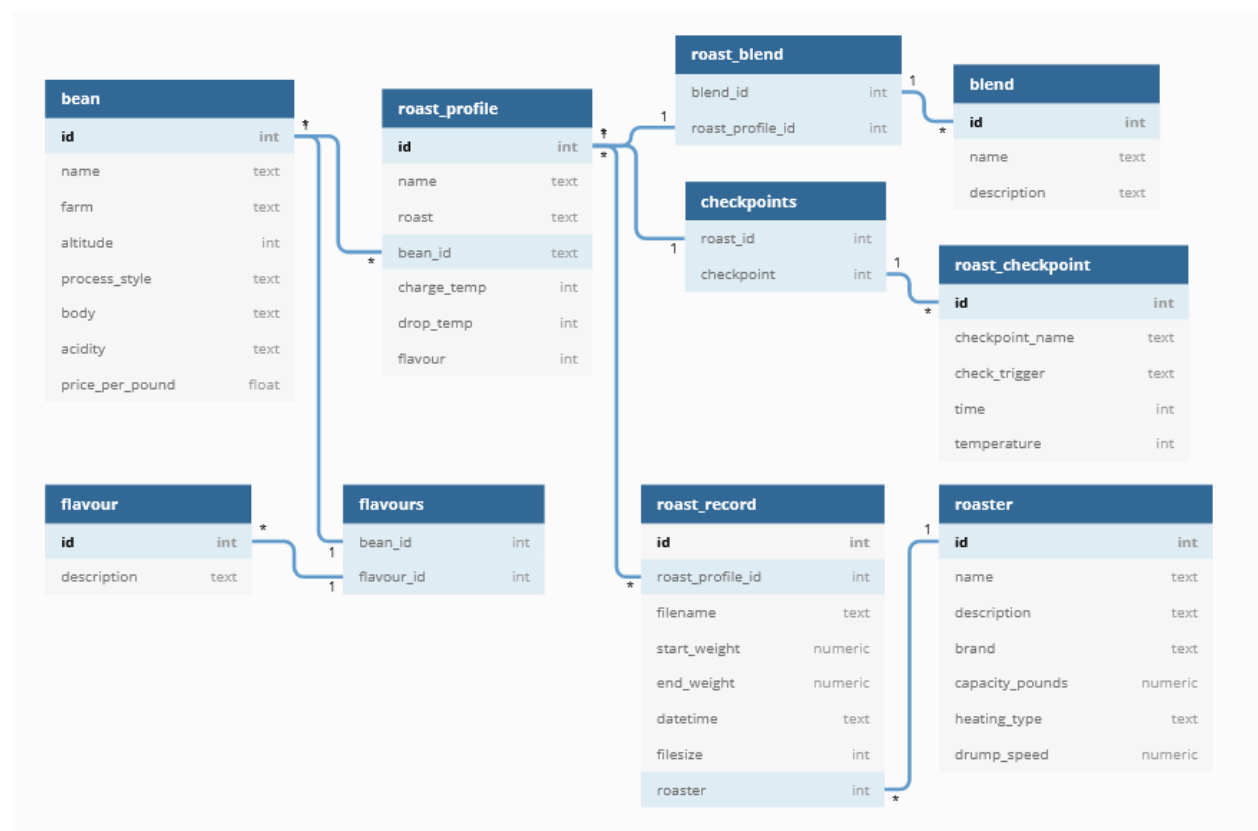
NetworkPropagator: Contains functions for forward propagation and all the activation functions used in that.

NetworkFileLoader: Loads the convolutional neural network model from the custom filetype and converts the byte data to a ConvolutionalNeuralNetwork object.

NetworkInitializer: Sets up the complex structure of the network within a ConvolutionalNeuralNetwork object and initializes all the values contained.

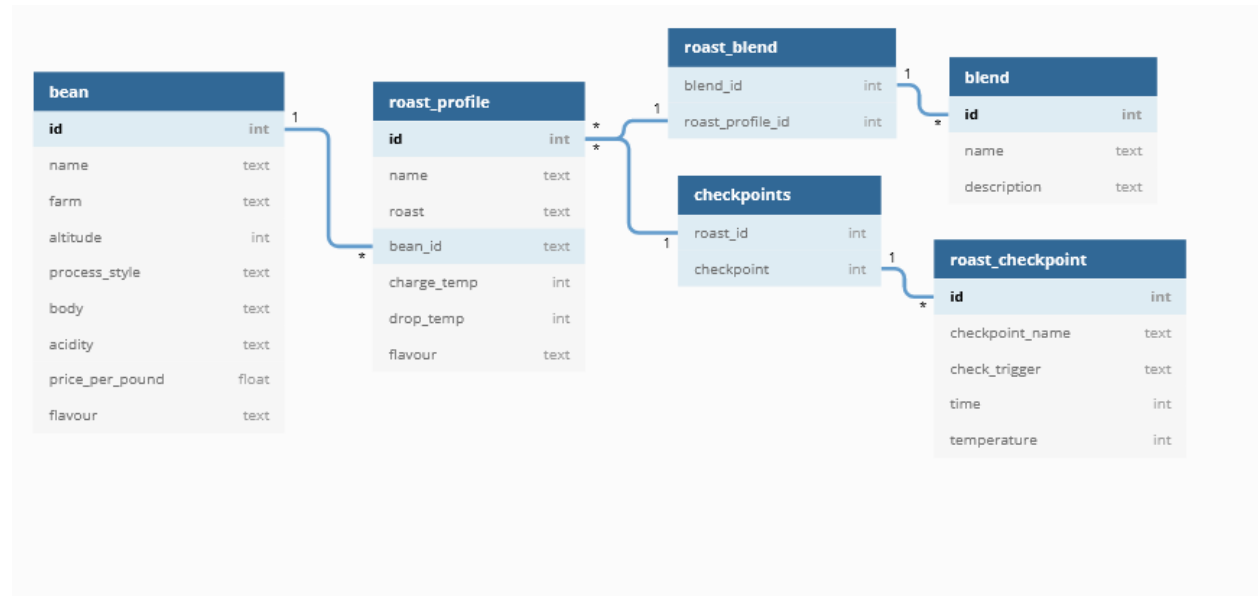
Database Schemas

Android Database



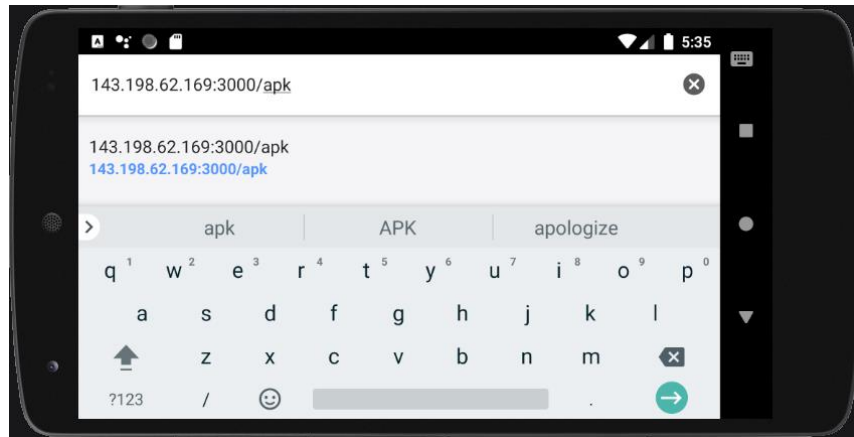
NodeJs Database

This database schema is almost the same as the Android database, however roaster and roast records are not needed. Also, the flavour table with the flavours junction were removed in favour of one string.

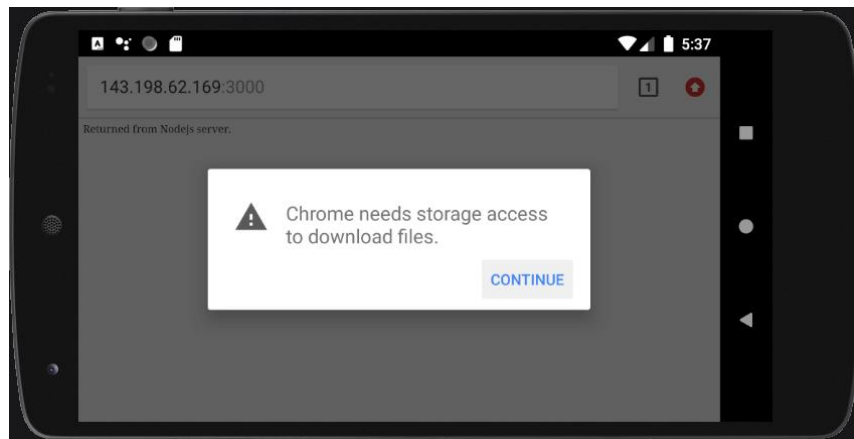


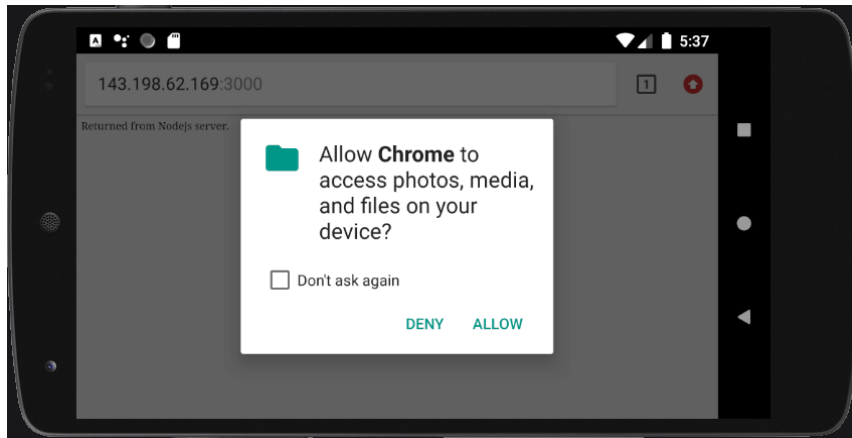
Installation Manual

1. Navigate to 143.198.62.169:3000/apk in your chosen browser on your phone. These instructions will be given using Chrome.

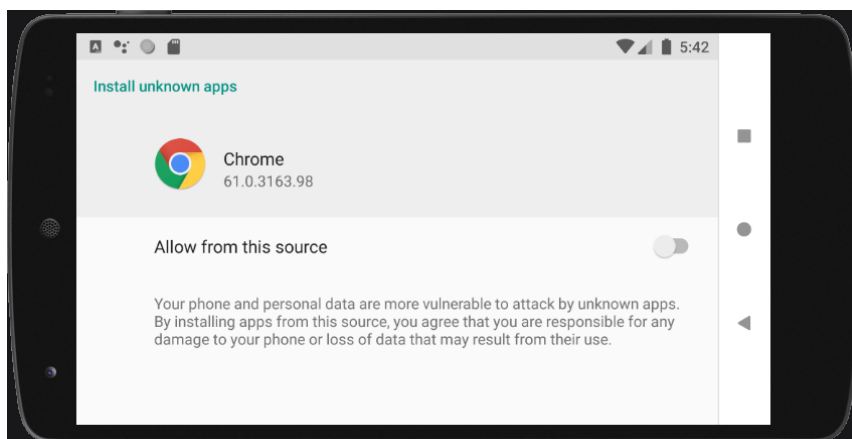
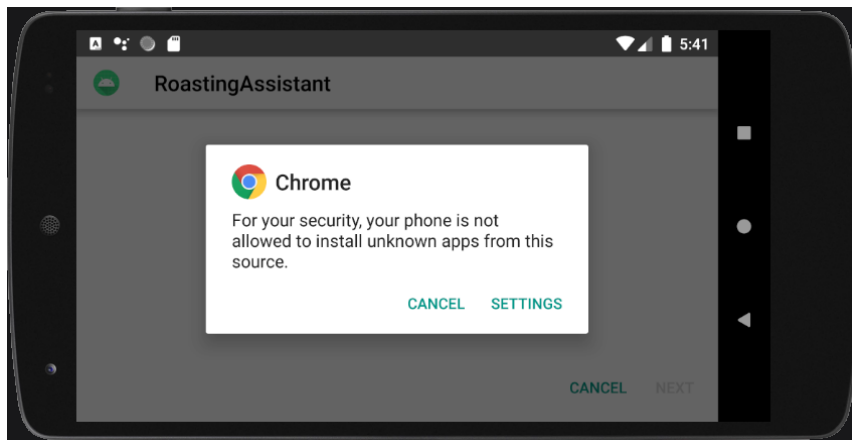


2. Grant Chrome storage access to download files if it does not already have it. When asked for storage access press 'CONTINUE'. Then press 'ALLOW' to allow chrome to access the filesystem. Once the .apk is downloaded, open it.

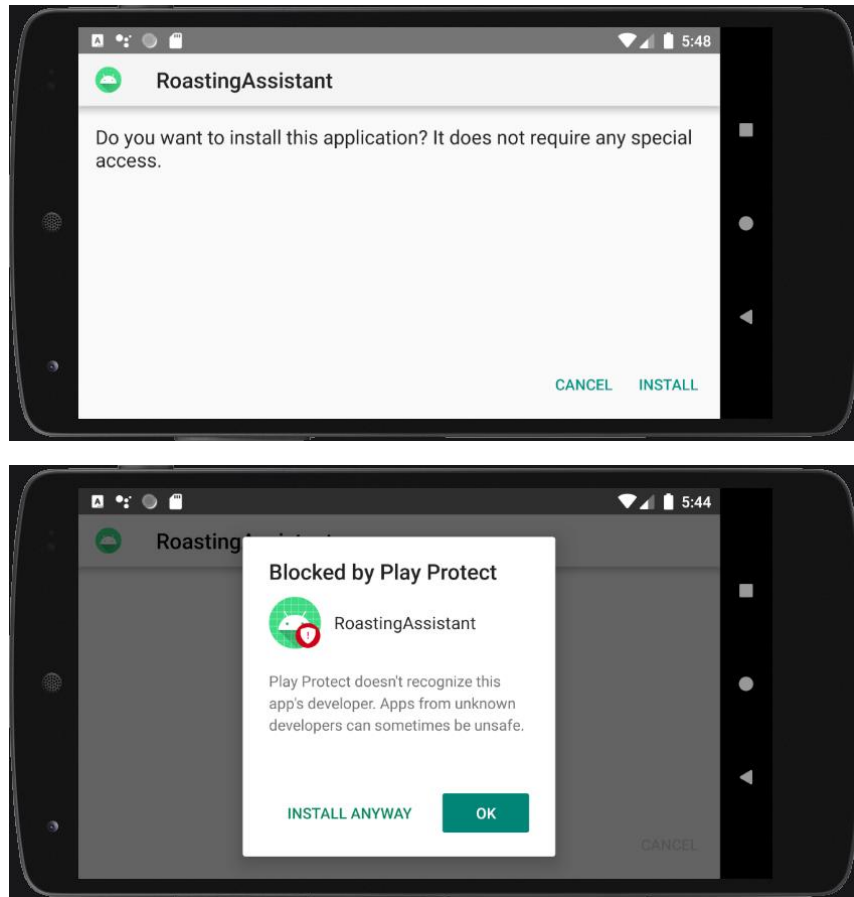




3. If you have not allowed installations from unknown sources chrome will notify you. Press 'SETTINGS'. Then check 'Allow from this source'. Press the back button.



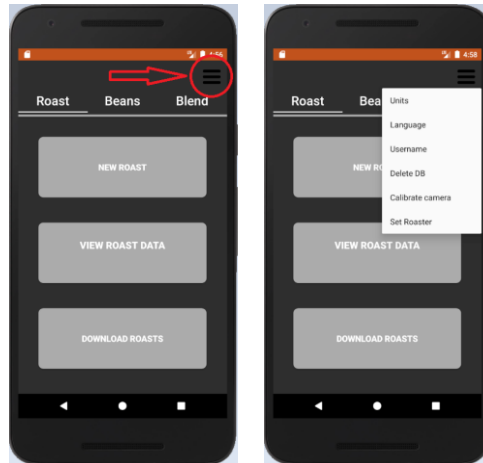
4. You will be prompted if you wish to install the apk. Select 'INSTALL'. The installation will be blocked because the developer is not recognised. Select 'INSTALL ANYWAY'.



5. After installation completes, the app will be available in the normal app menu.

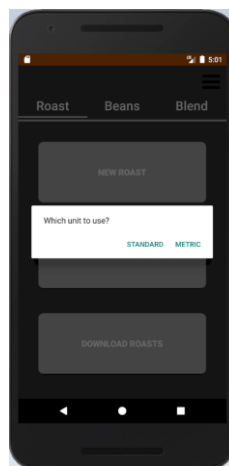
Changing settings

To change settings, tap the three-line button in the upper right corner of the screen.



From here you may change:

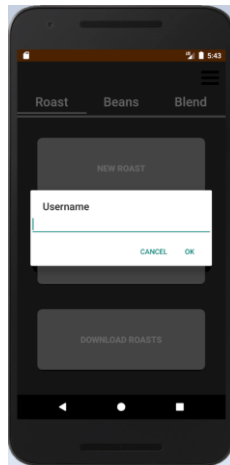
Units between metric and standard.



Language – only English is supported currently.

Username

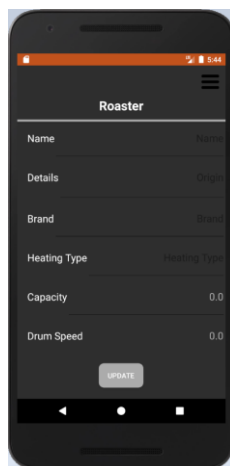
Enter desired username and select 'OK'.



Clear local database – Clears all information stored in the app. WARNING cannot be undone.

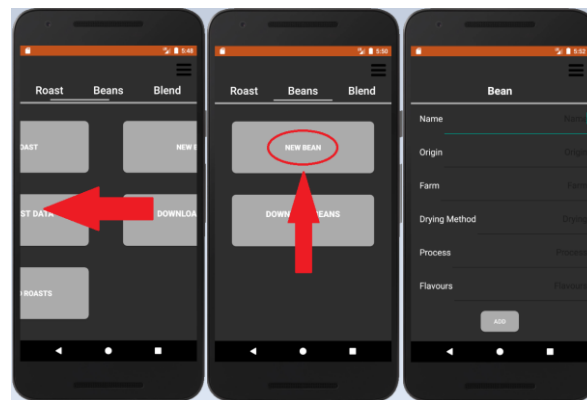
Calibrate Camera – is explained in detail in its own section.

Set Roaster – Enter as much information as is desired about the roaster being used.



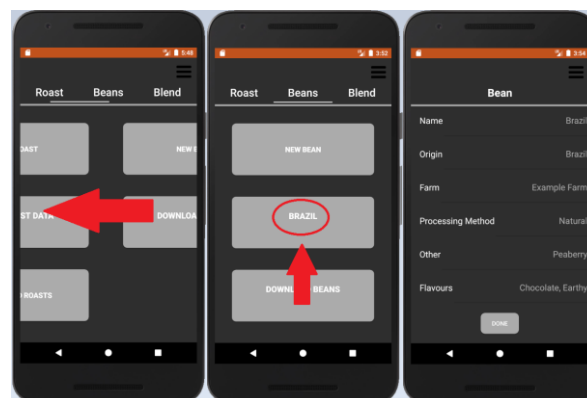
Creating a new bean entry

- 1: From the main screen swipe the menu left to the “beans” section. Then select the button titled “New Bean”.
- 3: Enter whatever details you wish to store. Note: The “Name” field is required.
- 4: Press the “Add” button at the bottom of the screen.



Viewing a previously entered bean

- 1: From the main screen swipe the menu left to the “beans” section. Then select the bean you wish to view. In this example the bean is named “Brazil”.
- This will open the Bean parameter activity from which you can view previous bean information.
- When you are finished viewing bean information tap the “Done” button at the bottom of the screen.

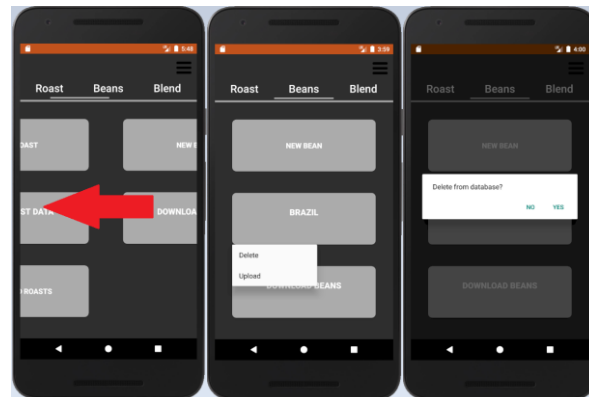


Delete or upload previous Bean entry

1: After navigating to the Bean section as previously described, press and hold on the bean you wish to delete. This will bring up the deletion and upload submenu.

2 A: To delete, tap “Delete” and when asked if you wish to delete from database select “Yes”.

2 B: To upload, tap “Upload”.



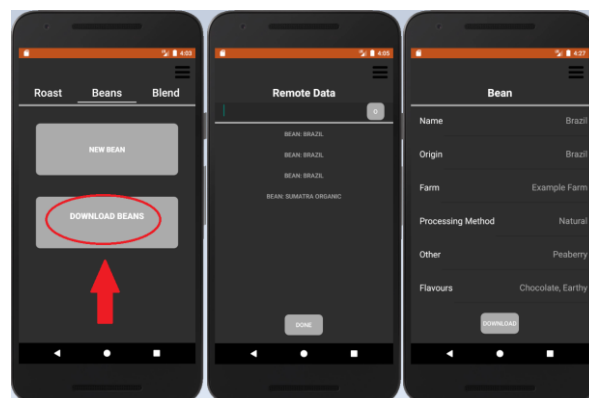
Download bean information from the internet

1: After navigating to the Bean section as previously described, ensure that you are connected to the internet and then tap the “Download Beans” button.

From here an activity will be started displaying all available to download beans

2: Tap an entry to view more information and download. The list has a search bar at the top if you know the name of the bean you wish to find, and scrollable if you wish to browse.

3: Press download and the bean will be added to your locally stored list of beans.



Creating a new Roast

1: From the main screen tap the button titled “New Roast”

3: Enter the name of the roast, then tap “beans” and select the bean which you wish to roast from the dropdown menu. Fill in roast level and drop temp.

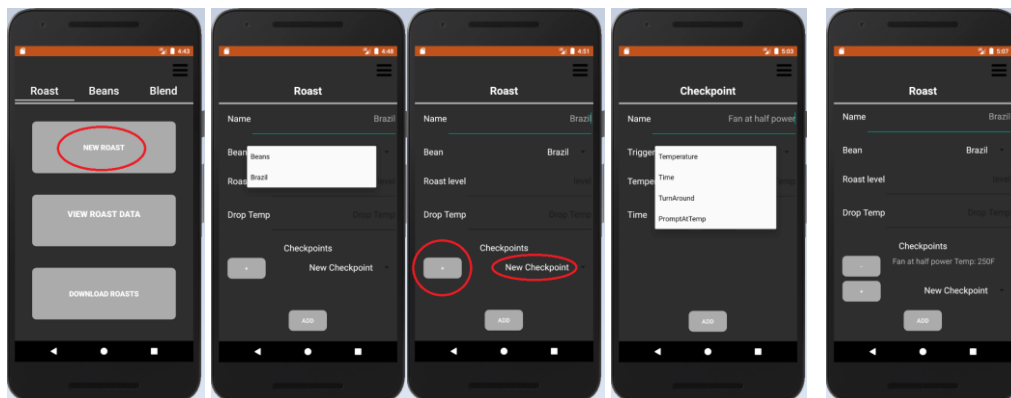
4: Add as many or as few checkpoints to the roast as are desired. To add a new checkpoint, tap the ‘+’ button. To reuse a previously created checkpoint, tap the dropdown labelled “New Checkpoint”.

Step 4a: Enter the title of the checkpoint, and then select the trigger dropdown to select the trigger type you wish to use.

Step 4b: Enter the time if your trigger was Time or enter the temperature for all other trigger types.

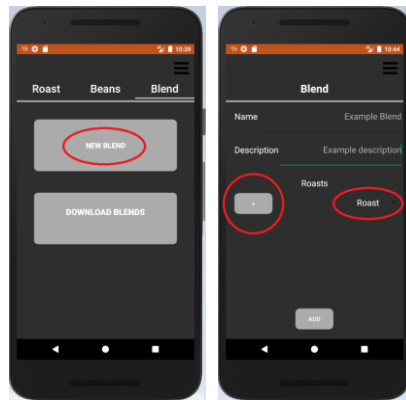
Step 4c: Tap the add button to add the checkpoint to this roast.

5: To Add another checkpoint, repeat step 4. To remove a checkpoint, tap the ‘-’ button. When finished, tap the “Add” button at the bottom of the screen.



Adding new blend

- 1: From the main screen swipe left twice until you are under the “Blend” tab.
- 2: Tap the button titled “New Blend”.
- 3: Enter a name and a description if desired.
- 4: Select a roast from the dropdown menu and tap the ‘+’ button to add it to the blend.
- 5: Repeat step 4 for however many roasts you wish to add to the blend.
- 6: Press the “Add” button to save the blend.



Calibrating Camera

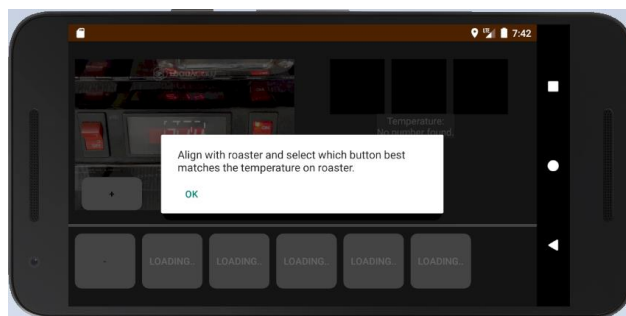
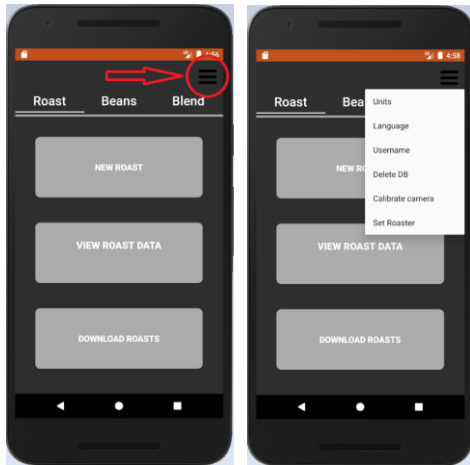
- 1: Tap the three-line button in the upper right corner of the screen.
- 2: Tap “Calibrate Camera” in the dropdown menu.
- 3: Line up the temperature readout on the roaster with the dotted line in the camera window.

The app will gradually adjust the camera settings every few seconds and attempt to read the temperature. The current image to be sent to the neural network is displayed in the three images in the upper right side of the screen. In the example this is showing 2 1 4 in white pixels on a black background. The number guesses are displayed in the six buttons on the bottom of the screen. The currently guessed number is in black text. The “-” character means the app was unable to read a temperature with those settings.

- 4: When the clearest representation of the number is shown and the correct number is displayed on the current button, tap that button to apply those camera settings.

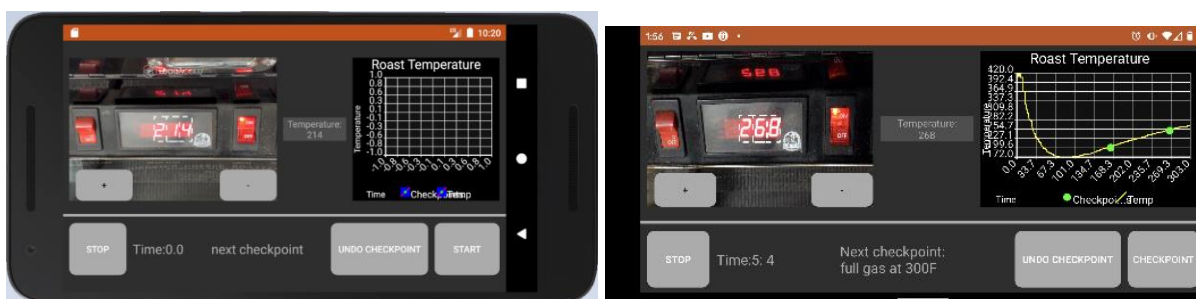
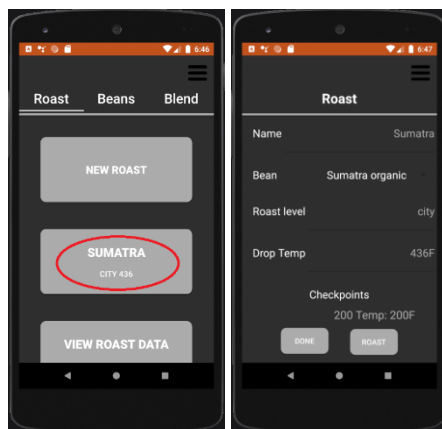
Image A displays bad settings

Image B displays good settings



Roasting

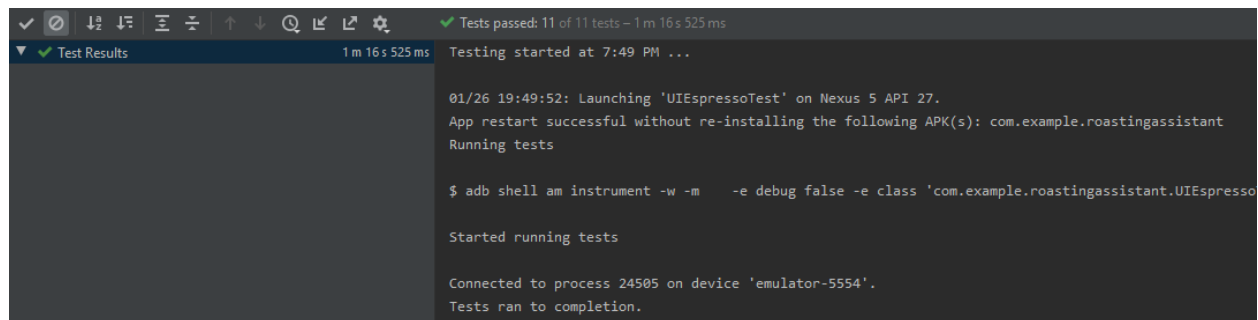
1. Tap a desired roast profile to view from the main menu.
2. From the view menu tap the “roast” button to open the roasting activity.
3. If requested for camera access, grant permission
4. Physically align phone so that the temperature readout on the roaster is within the bounds of the cut-out box which is overlayed in the center of the camera preview.
5. Adjust until the correct temperature is displayed in the center of the screen.
6. When satisfied with the alignment and readout, press the “start” button at the bottom right of the screen.
7. Go about your roast as normal. Any time a checkpoint is reached, the app will automatically mark the point on the graph. As the checkpoint is approached the app will automatically play a warning alarm to remind the user in case some adjustment to the roast is desired.
8. When the roast is complete, or if you desire to end the roast at any time, press the “stop” button in the bottom left of the screen.



2.6. Testing Details and Results

User Interface

The app user interface was tested using the Android Espresso testing tool. This was done by creating tests and assertions to assure that all parts of the user interface are behaving as desired. The goal of this section of testing is to ensure that the user will have the desired outcome from each interaction with the user interface. It is difficult to show the UI tests working without video, however a screenshot of all espresso tests being passed in Android Studio is shown below. The greatest difficulty in the UI testing was just getting the testing software working as desired. There were no problems of interest, so their coverage will be brief.



The tests were as follows:

Test Case ID	Test Scenario	Test Steps	Expected Results
UIT01	Scroll Menu to the left.	Trigger swipe to the right.	Menu switches to the previous fragment. No change if on first fragment.
Results			
The menu swiped between fragments as desired.			
Test Case ID	Test Scenario	Test Steps	Expected Results
UIT02	Scroll Menu to the right.	Trigger swipe to the left.	Menu switches to the next fragment. No change if on last fragment.
Results			
The menu swiped between fragments as desired.			

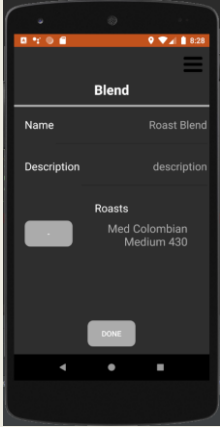
Test Case ID	Test Scenario	Test Steps	Expected Results
UIT03	Scroll Menu down.	Trigger swipe up.	Menu fragment scrolls data upwards.
Results			
The menu data was scrolled as desired.			

Test Case ID	Test Scenario	Test Steps	Expected Results
UIT04	Scroll Menu up.	Trigger swipe to the down.	Menu fragment scrolls data upwards.
Results			
The menu data was scrolled as desired			

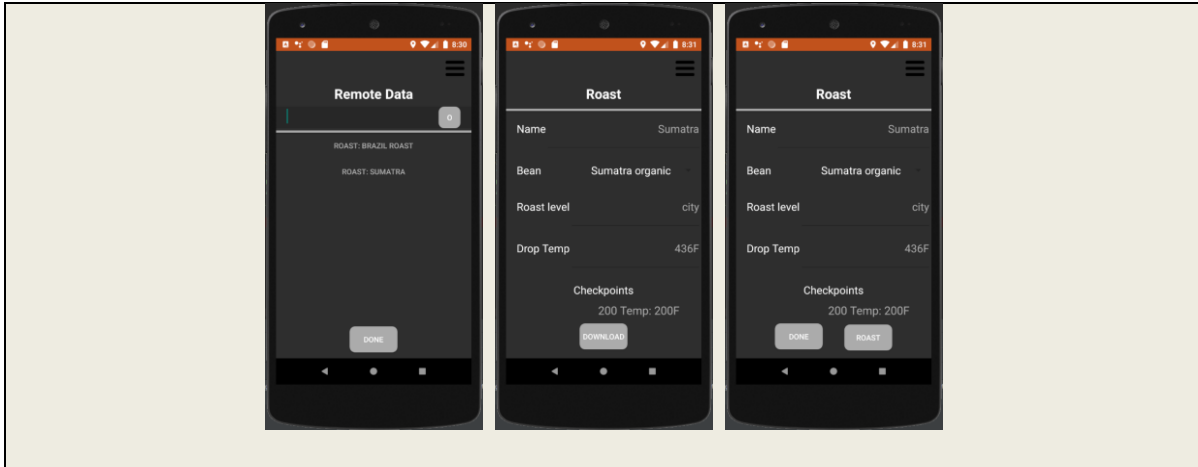
Test Case ID	Test Scenario	Test Steps	Expected Results
UIT05	Open settings menu.	Trigger press on settings button.	Settings dropdown is displayed.
Results			
The settings menu dropdown is displayed when pressed as desired.			

Test Case ID	Test Scenario	Test Steps	Expected Results
UIT06	Create new bean entry	<ol style="list-style-type: none"> 1. Scroll menu right to beans. 2. Press "New Bean" 3. Enter name 4. Enter origin 5. Enter drying method 6. Enter flavour 7. Enter process 8. Enter farm 9. Press create bean 	A complete bean entry will be filled out and ready for database entry.
Test Data			
Name = Columbian Supremo Origin = Columbia Drying method = sun Flavour = chocolate, honey, nutty Process = natural Farm = Anei S.N.			
Results			
All menu maneuvering and data input was done correctly with no errors. <div data-bbox="698 1140 914 1566" data-label="Image"> </div>			

Test Case ID	Test Scenario	Test Steps	Expected Results
UIT07	Create new roast entry	<ol style="list-style-type: none"> 1. Press "New Roast" button. 2. Enter Roast Name 3. Enter Roast 4. Press "Add Checkpoint" 5. Set Checkpoint Name 6. Set type to temperature 7. Enter temperature 8. Press "Add Checkpoint" Press "Add Roast" 	A complete roast entry will be filled out and ready for database entry.
Test Data			
Roast Name = Med Colombian Bean = Colombian Supremo Roast = Medium Checkpoint Name = Drop time Temperature = 430			
Results			
All menu maneuvering and data input was done correctly with no errors. <div data-bbox="698 1159 914 1585" data-label="Image"> </div>			

Test Case ID	Test Scenario	Test Steps	Expected Results
UIT08	Create new blend.	<ol style="list-style-type: none"> 1. Scroll right 2. Scroll right 3. Press "Add Blend" 4. Press Add Bean 5. Select Colombia 6. Press Save Blend 	A complete blend entry will be filled out and ready for database entry.
Results			
All menu maneuvering and data input was done correctly with no errors.			
			

Test Case ID	Test Scenario	Test Steps	Expected Results
UIT09	Test remote data browser	<ol style="list-style-type: none"> 1. Press download roast 2. Press top roast 3. Press download 	Remote data browser will be opened, and the first roast entry will be displayed.
Results			
All menu maneuvering and data input was done correctly with no errors. Sumatra roast is viewed and downloaded from server.			



Test Case ID	Test Scenario	Test Steps	Expected Results
UIT10	Test opening roasting activity.	1. Press existing roast 2. Press Use Roast Profile	Roasting activity will be started and display roasting interface.
Results			
The roasting activity is maneuvered to, the interface displayed, and the camera hardware loaded and displayed.			

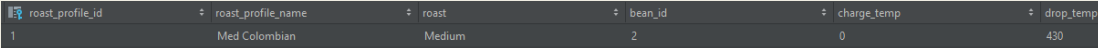
Test Case ID	Test Scenario	Test Steps	Expected Results
UIT11	Test stop roast.	1. Run test UIT10 2. Press start Roast 3. Press Stop roast	Roast will be started, and then stopped successfully.
Results			
The roasting activity is maneuvered to, the interface displayed, and the camera hardware loaded and displayed. A roast is started, the timer and data collection starts. The stop button is pressed and the timer and data collection stops.			

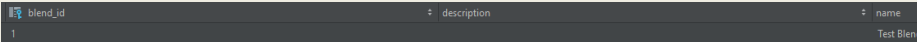
Database Testing

Database testing was also done through Espresso. A lot of time was spent trying to get unit tests to work, but I was unable to get access to a Context to start the database with. Instead, a hidden activity which was only used for testing was created. This contained all the database functions which were to be tested.

Test Case ID	Test Scenario	Test Steps	Expected Results
DBT01	Create database and generate tables.	1. Create new database 2. Fill in tables according to database schema	Database will be created and match schema.
Results			
Database was created matching schema.			

Test Case ID	Test Scenario	Test Steps	Expected Results																				
DB02	Add new bean entry in database.	1. Add new bean entry 2. Add dummy flavour to bean entry	Database will have an entry matching the test data.																				
Test Data																							
Name = Columbian Supremo Origin = Columbia Drying method = sun Flavour = chocolate, honey, nutty Process = natural Farm = Anei S.N.																							
Results																							
An entry is added to the database matching the test data.																							
<table><tr><th>bean_id</th><th>bean_name</th><th>origin</th><th>farm</th><th>altitude</th><th>process_style</th><th>flavour</th><th>body</th><th>acidity</th><th>drying_method</th></tr><tr><td>1</td><td>Colombian Supremo</td><td>Colombia</td><td>An So.N.</td><td></td><td>natural</td><td>chocolate, honey, nutty</td><td></td><td></td><td>Sun</td></tr></table>				bean_id	bean_name	origin	farm	altitude	process_style	flavour	body	acidity	drying_method	1	Colombian Supremo	Colombia	An So.N.		natural	chocolate, honey, nutty			Sun
bean_id	bean_name	origin	farm	altitude	process_style	flavour	body	acidity	drying_method														
1	Colombian Supremo	Colombia	An So.N.		natural	chocolate, honey, nutty			Sun														

Test Case ID	Test Scenario	Test Steps	Expected Results
DB03	Add new Roast entry to database.	1. Add new bean entry 2. Add dummy checkpoint to roast entry 3. Add Roast entry	Database will have an entry matching the test data.
Test Data			
Roast Name = Med Colombian Bean = Colombian Supremo Roast = Medium Checkpoint Name = Drop time Temperature = 430			
Results			
An entry is added to the database matching the test data.			
			

Test Case ID	Test Scenario	Test Steps	Expected Results
DB04	Add new Blend entry to database.	1. Add new bean entry 2. Add dummy checkpoint to roast entry 3. Add Roast entry 4. Add blend entry	Database will have a blend entry matching the test data.
Test Data			
Blend name = Test Blend Roasts = Med Colombian			
Results			
An entry is added to the database matching the test data.			
			

Test Case ID	Test Scenario	Test Steps	Expected Results														
DB05	Add new Roaster entry to database.	1. Add new roaster entry to database	Database will have a roaster entry matching the test data.														
Test Data																	
Roaster name = Test roaster Roaster description = Test																	
Results																	
An entry is added to the database matching the test data.																	
<table><tr><th>roaster_id</th><th>roaster_name</th><th>roaster_description</th><th>brand</th><th>capacity_pounds</th><th>heating_type</th><th>drum_speed</th></tr><tr><td>1</td><td>Test Roaster</td><td>Test</td><td></td><td>0</td><td></td><td>0</td></tr></table>				roaster_id	roaster_name	roaster_description	brand	capacity_pounds	heating_type	drum_speed	1	Test Roaster	Test		0		0
roaster_id	roaster_name	roaster_description	brand	capacity_pounds	heating_type	drum_speed											
1	Test Roaster	Test		0		0											

```

Tests passed: 4 of 4 tests - 13 s 901 ms
Test Results 13 s 901 ms Testing started at 10:46 PM ...

01/26 22:46:27: Launching 'DatabaseTest' on Nexus 5 API 27.
App restart successful without requiring a re-install.
Running tests

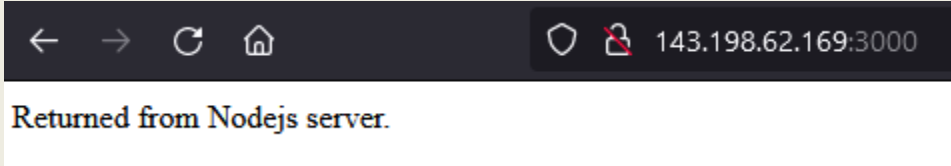
$ adb shell am instrument -w -m -e debug false -e class 'com.example.roasti
Connected to process 3283 on device 'emulator-5554'.

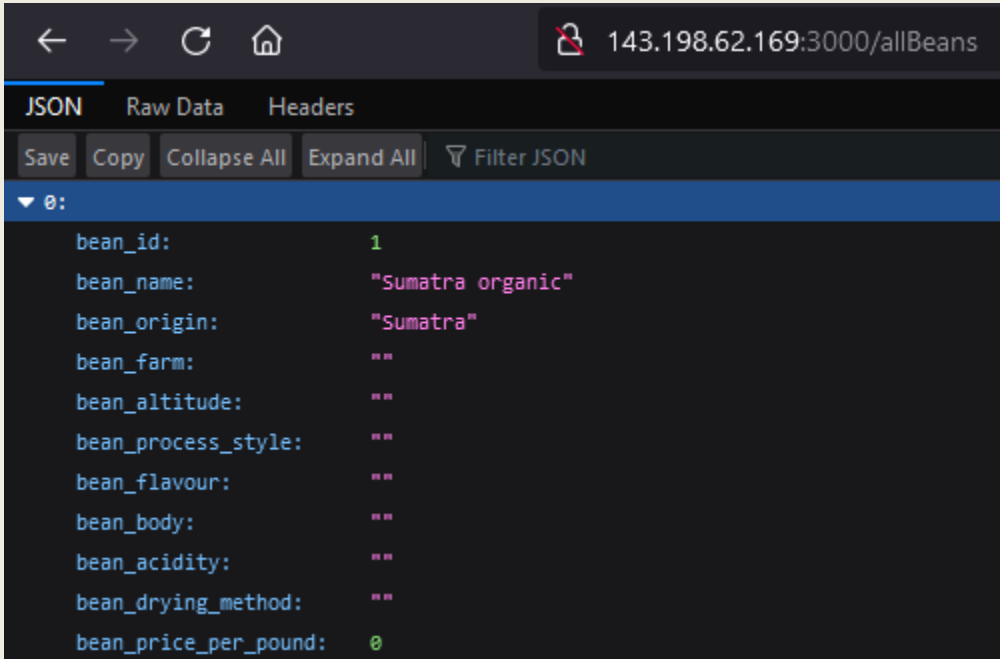
Started running tests

Tests ran to completion.

```

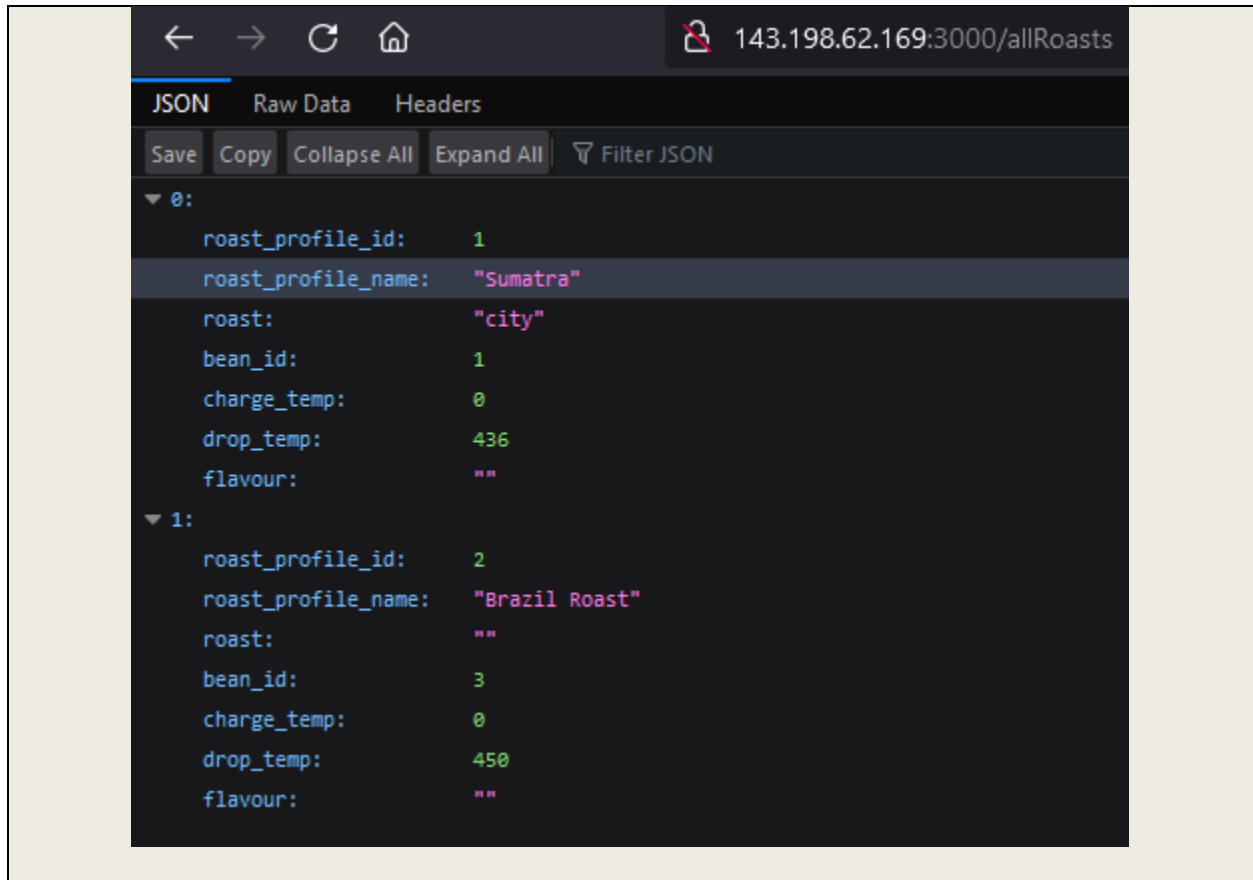
NodeJS Testing

Test Case ID	Test Scenario	Test Steps	Expected Results
NDT01	Connect to NodeJS server.	1. Enter IP address and port of server.	Server will return the string "Returned from Nodejs server."
Results			
"Returned from Nodejs server." Was returned by the server.			
			

Test Case ID	Test Scenario	Test Steps	Expected Results
NDT02	Request bean list from server.	1. Navigate to http://143.198.62.169:3000/allBeans	Server will return a JSON list representing stored bean data.
Results			
JSON list of bean data was returned			
			

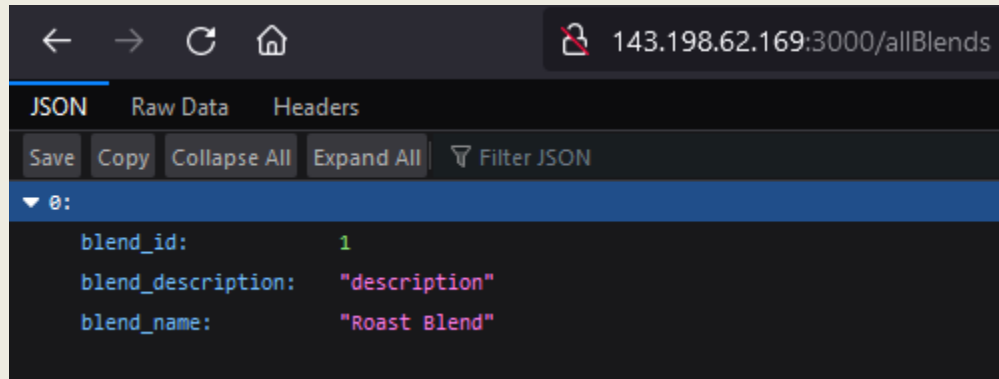
```
▼ 1:
  bean_id:          2
  bean_name:        "Brazil"
  bean_origin:      "Brazil"
  bean_farm:         ""
  bean_altitude:    ""
  bean_process_style: ""
  bean_flavour:     ""
  bean_body:         ""
  bean_acidity:     ""
  bean_drying_method: ""
  bean_price_per_pound: 0
▼ 2:
  bean_id:          3
  bean_name:        "Brazil"
  bean_origin:      "Brazil"
  bean_farm:         ""
  bean_altitude:    ""
  bean_process_style: ""
  bean_flavour:     ""
  bean_body:         ""
  bean_acidity:     ""
  bean_drying_method: ""
  bean_price_per_pound: 0
```

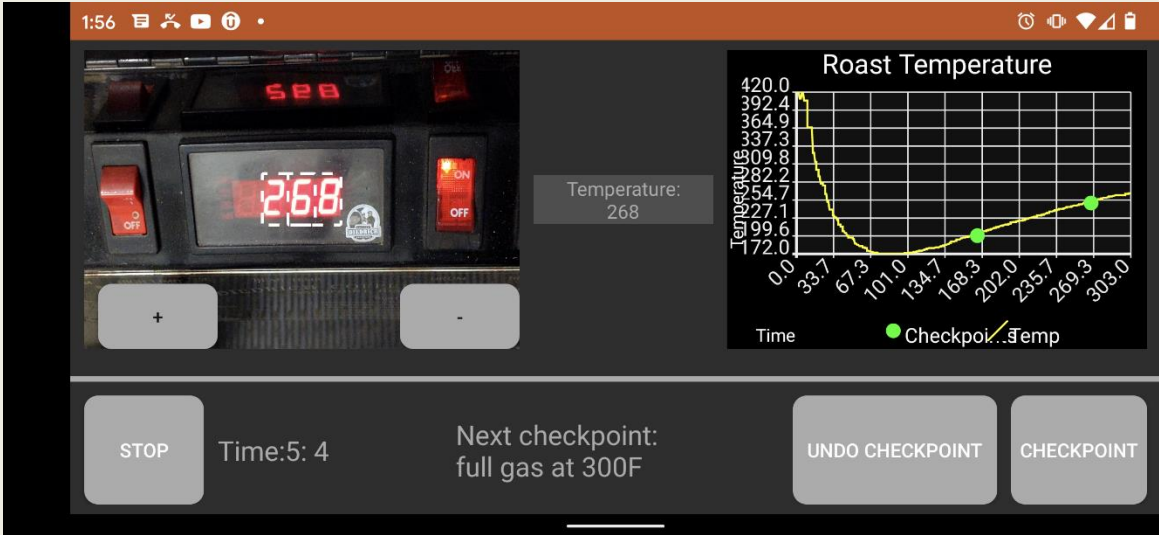
Test Case ID	Test Scenario	Test Steps	Expected Results
NDT03	Request roast list from server.	1. Navigate to http://143.198.62.169:3000/allRoasts	Server will return a JSON list representing stored roast data.
Results			
JSON list of roast data was returned			



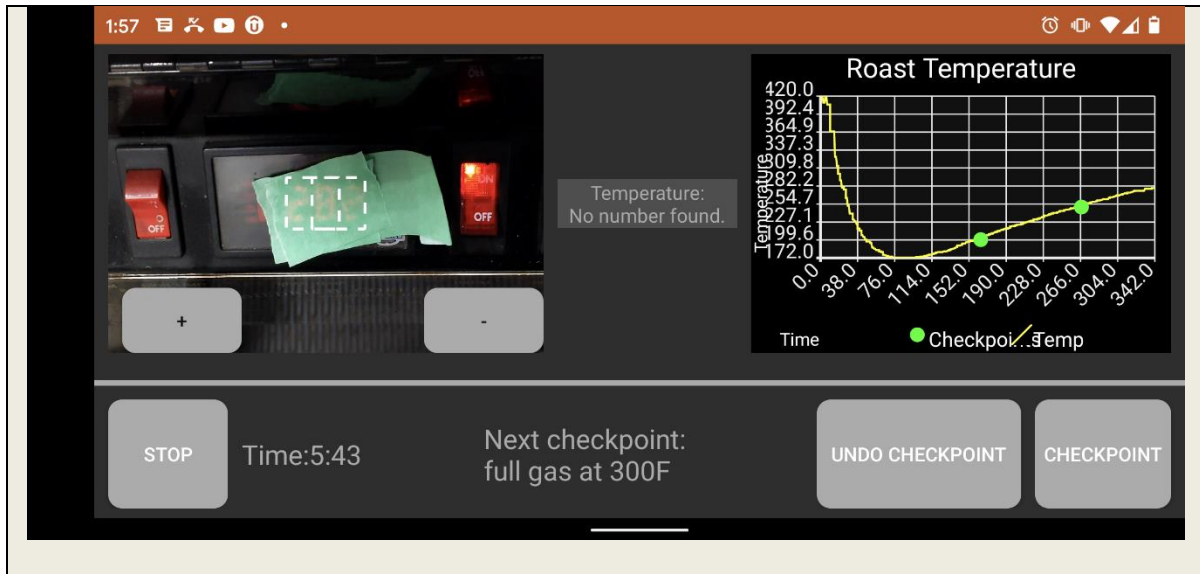
Test Case ID	Test Scenario	Test Steps	Expected Results
NDT04	Request blend list from server.	1. Navigate to <code>http://143.198.62.169:3000/allBlends</code>	Server will return a JSON list representing stored blend data.
Results			

JSON list of blend data was returned



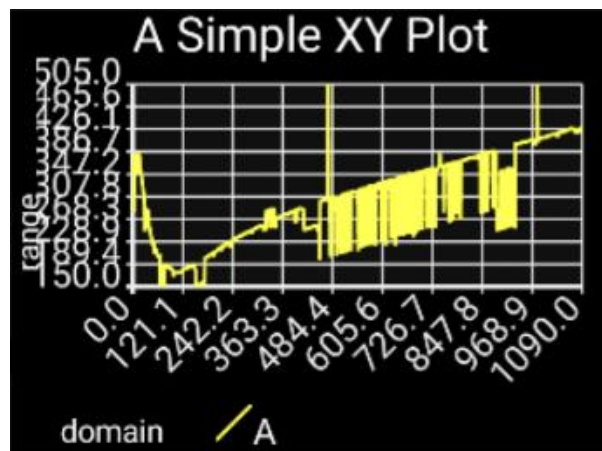
Test Case ID	Test Scenario	Test Steps	Expected Results
NNT01	Convert temperature image to integer.	1. Start roast 2. Align camera with temperature	The temperature string in the center of the view will read 268
Results			
The temperature readout in the center of the activity reads 268.			
			

Test Case ID	Test Scenario	Test Steps	Expected Results
NNT02	Recognize a lack of digits in the image.	1. Start roast 2. Align camera with temperature 3. Put masking tape over the temperature readout on the roaster	The temperature string in the center of the view will read "No number found"
Results			
The temperature readout in the center of the activity reads "No number found".			



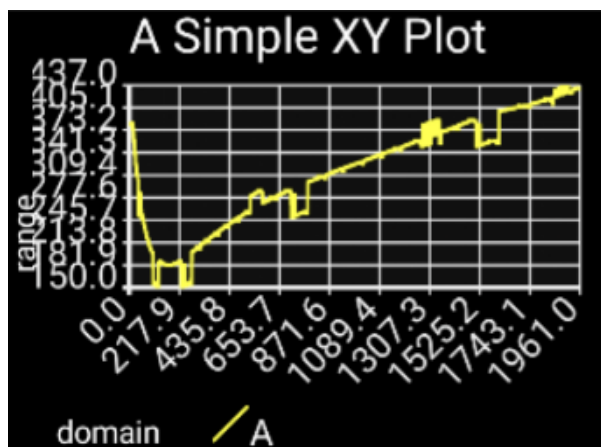
There was constant testing of newly trained models and post-processing techniques while working on other parts of the project. This was documented through screenshots at different points along the way as it was going on daily at work over months. Below are some examples of the evolution of the training over time.

This was one of the first graphed full coffee roasts using the app. As you can see there is a lot of noise and errors on the part of the image recognition. A recognizable general curve can be seen, but this level of error was not acceptable as a final product. This required a better trained model as well as post-processing techniques which were implemented later.



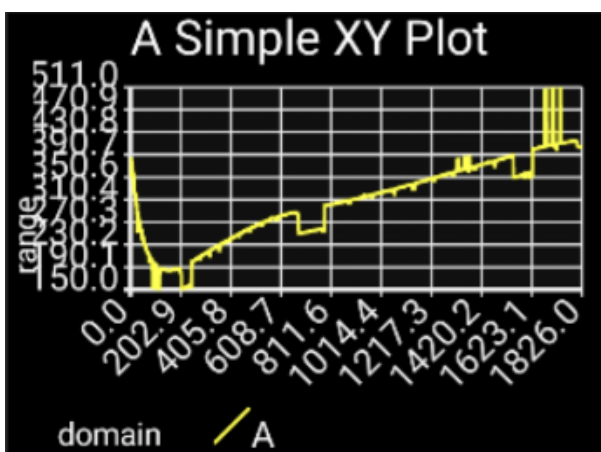
In this example the quality of the model was greatly improved upon. This was done through better training data. The downloaded data sets were supplemented by actual images of the roasting

temperature readout. Despite the large increase in quality, it is still plain to see that there is a large degree of errors present.



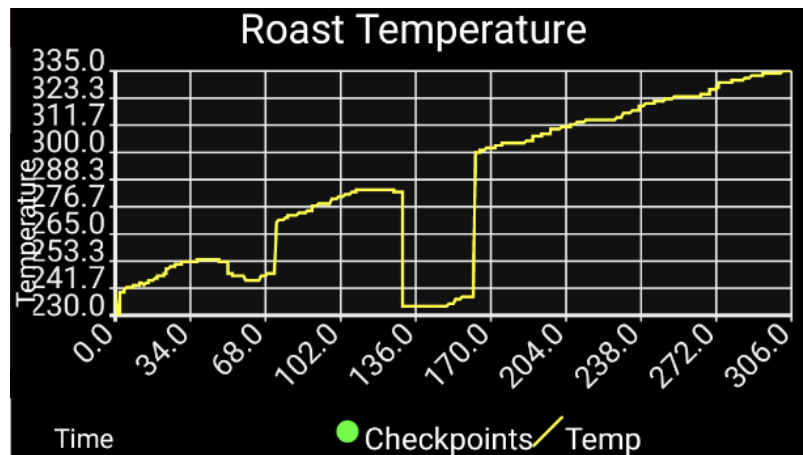
At this point a post processing technique was implemented where the image is sent through the neural network three times from a shifted left position, normal position and a shifted right position.

These three guesses are then averaged to mitigate any potential positional problems with the recognition. This was repeated eight times from different samples and averaged again. Since the neural net processing speed was much faster than needed, running it three times per guess was not a problem. Also, there was tuning, and many adjustments made to the training of the neural network model.

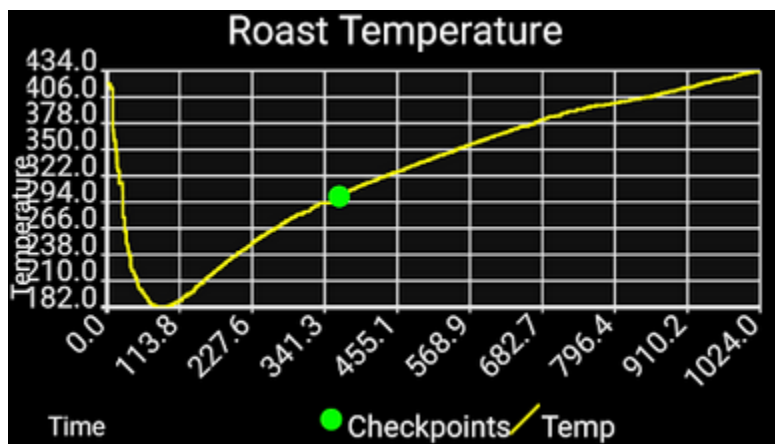


At this point, in addition to further work on the training of the network model, a new post-processing technique was implemented to increase accuracy. The technique used was to apply a median filter to the raw temperature data prior to displaying it on the graph. This involves taking a point in the graph, then copying a subset of data centered on that point, ordering that data from lowest to highest and then replacing the point with the median value of that subset. In doing this, any spikes are pushed to the beginning or end of the subset and then thrown away. There is still a

large problem in this data when the network mistakenly predicts a number in the ten's column. Rather than causing a brief spike in the graph, this causes a large flat dip as is seen below.



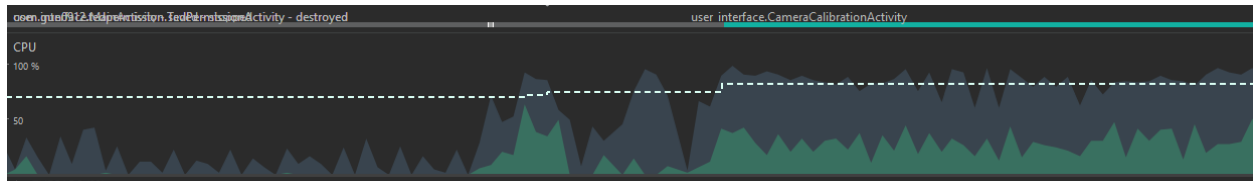
At this point a final bit of post-processing was attempted. The network model was making predictable mistakes with similar digits, mostly 8 and 0. This was likely because aside from the horizontal line across the center they are the same digit on this readout. The attempted solution was to check if the previous digit was an 8 and next a 0 was predicted to set it to a 9 instead, and vice versa. This had partial success but was not perfect. Eventually, some mistakes in the math training the network were discovered and both training time and quality were greatly increased, leading to the final graph shown below. At this point the recognition was deemed acceptable and work on the image recognition was concluded.



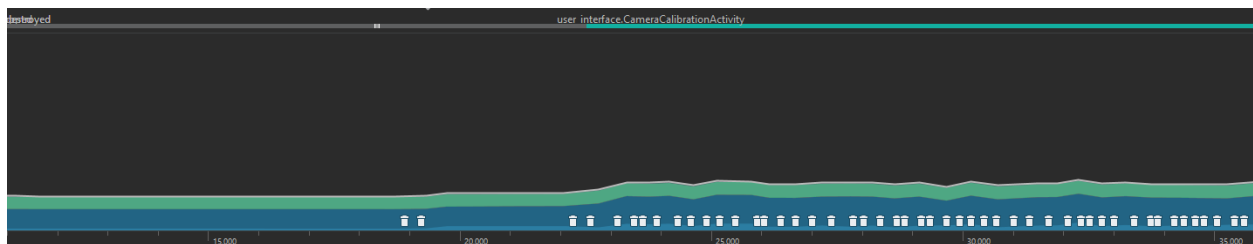
The following are readouts from hardware monitoring while the image recognition is running. There was a worry that the neural network may consume too much processing power, memory, and with those battery power.

This is the CPU monitoring before and after running the CameraCalibrationActivity which utilizes the same image recognition class as the roasting activity. The upper bar, although hard to read, is showing the main menu which transitions to the CameraCalibrationActivity when it turns green. Below is the

graphed CPU usage, of which the green is usage by the app and the darker rear is by background processes on the phone. It is not clear why the background processes increase when the image processing begins, but it is likely the OS handling the camera hardware. As you can see the app CPU usage is well within a reasonable range, and below that of even the background processes. In fact, the time before the activity begins, when the OS is likely starting the activity actually is spiking the CPU usage more than the image recognition ever does.



Pictured here is the memory monitoring before and after running the CameraCalibrationActivity. As with the CPU usage, there is minimal and acceptable increase in memory usage after the activity has started. On average memory usage increases ~20mb when the image recognition is active. With the average ram on a cellphone today being around 4GB this is definitely an acceptable usage. Due to all the images being processed there is quite a bit of garbage collection being done by the OS as you can see pictured by the white garbage can symbol at the bottom of the graph. All this garbage collection could increase CPU usage. This is something that may be addressed in future iterations, but as the CPU usage is well within acceptable range it was not addressed in this project.



The GPU was not monitored as the neural network is processed by the CPU. Creating everything involved with the convolutional neural network was the primary concern without the addition of learning GPU libraries to speed up processing time.

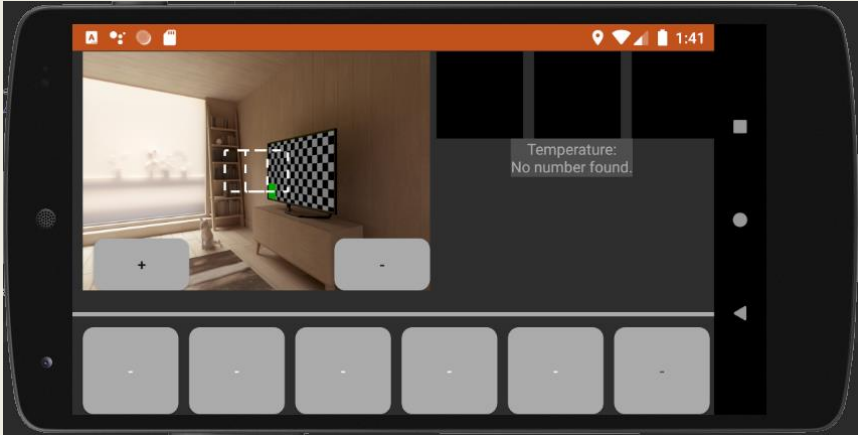
The execution time of the neural network was also monitored. The following are 10 execution time samples converted to seconds:

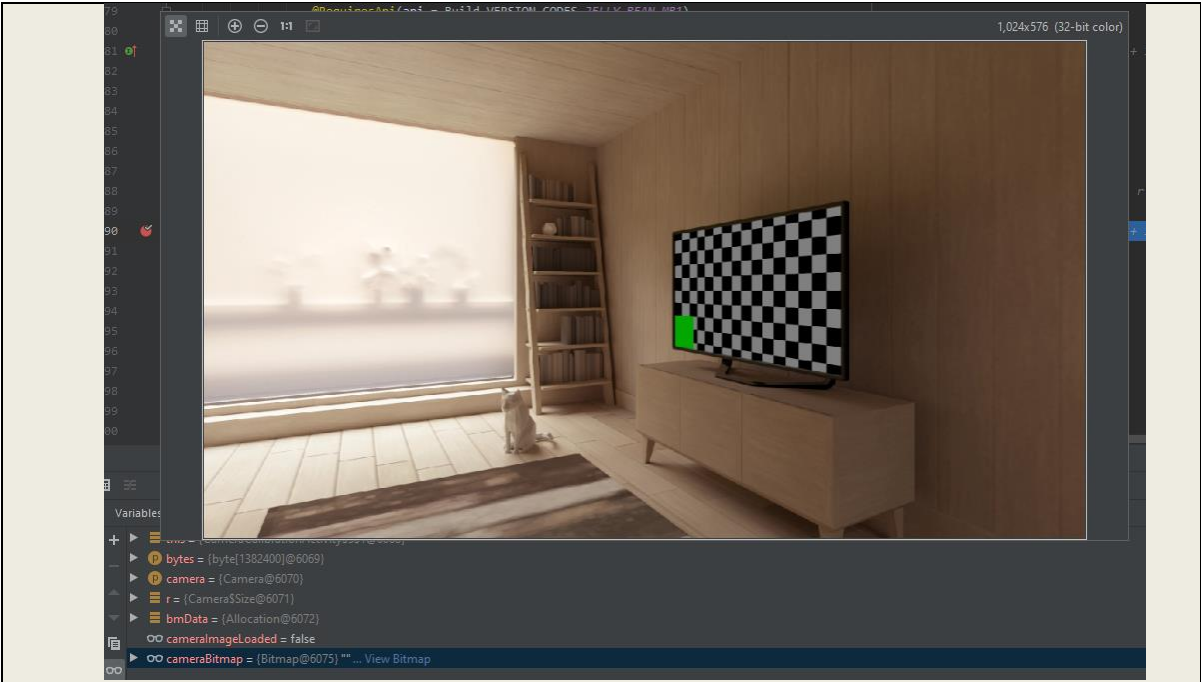
0.01, 0.028, 0.035, 0.073, 0.013, 0.017, 0.072, 0.011, 0.034, 0.064

This averages to an execution time of 0.036 seconds. The average time per degree of temperature increase after temperature turn around is ~2 seconds, making this speed more than 55 times faster than what is minimally required.

Test Case ID	Test Scenario	Test Steps	Expected Results
CMT01	Gain access to camera hardware.	<ol style="list-style-type: none"> 1. Get camera permission 2. Acquire camera object through Camera.open() 	Get a non-null camera object to work with.
Results			
<p>Android camera permission was requested and accepted. Camera.open() returned a Camera object which was stored in mCamera and went from null to a working camera object.</p> <pre> ▶ this = {CameraCalibrationActivity@6000} mCamera = null mPreview = null ▶ this = {CameraCalibrationActivity@6000} ▶ mCamera = {Camera@6055} mPreview = null </pre>			

Test Case ID	Test Scenario	Test Steps	Expected Results
CMT02	Convert image to proper format for preprocessing.	<ol style="list-style-type: none"> 1. Create new PreviewCallback class and override onPreviewFrame function to access camera image byte array. 2. Use a render script to convert byte array to an Allocation object representing bitmap data. 	A bitmap object which matches what is shown in the camera preview window.

		3. Use Allocation.copyTo to copy Allocation data into a Bitmap object.	
Results			
<p>This took a lot of time to get working. The allocation script which converts the image byte array to bitmap data was taken from this post on StackOverflow.</p> <p>https://stackoverflow.com/questions/4768165/convert-preview-frame-to-bitmap</p> <p>Using this script, a Bitmap object is generated which matches the camera preview that is shown.</p>			
			



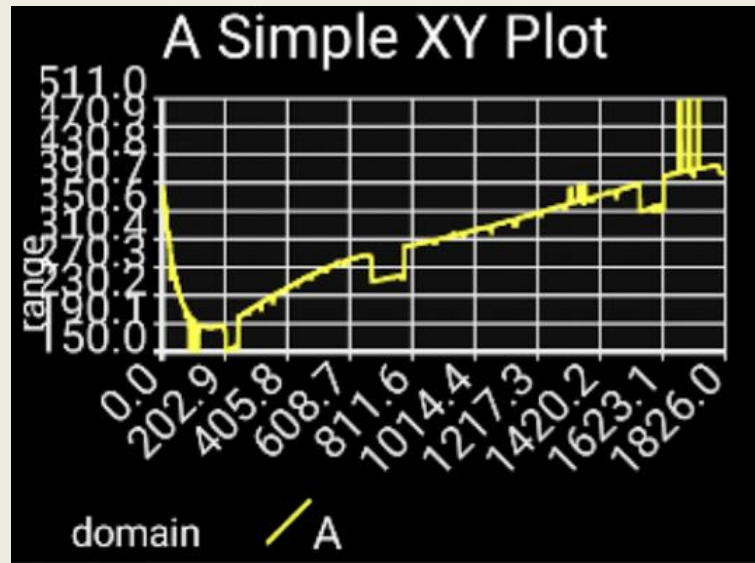
Note: The image previewed is from the built in Android emulator’s emulated camera hardware.

Graphing

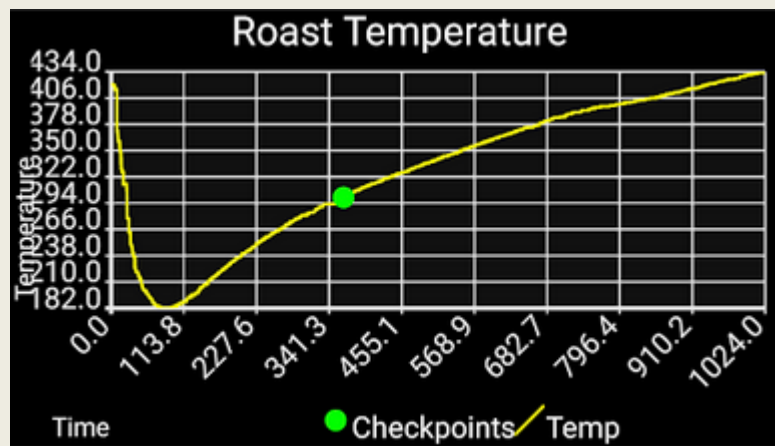
Test Case ID	Test Scenario	Test Steps	Expected Results
GRT01	Create a graph from specified data and visually inspect for correctness.	<div>1. Fully set up app and run through recording a real roast.</div> <div>2. Visually inspect graph that curve follows what is expected and is free from noise.</div>	A smooth graph dropping down and then slowly curving up.
Results			

After a lot of attempts with different neural network models and post-processing techniques a smooth graph free of spikes was created.

Early failed attempt



Final Passed Attempt



Convolutional Neural Network Trainer

The testing on the desktop training network was a little more informal as probably hundreds of trained models were created over the course of the project. There was a lot of tinkering involved in the process to get to the final product that was developed. The largest factors which affected the model's fitness were the training data used, and several mistakes which were discovered in the math and coding along the way.

The first point of note was when training was actually achieved. Initially the network would just hover around 10% accuracy, which would be the expected starting place with 10 digits. The training rate was increased until it caused exploding gradients and the error would approach infinity. Any time this happened the network was ruined, and training needed to be started again.

Eventually the loss calculation was changed from Cross Entropy to Average Squared Deviation and learning was finally achieved, however slow. At this point the dataset used was the house number set, which would eventually get up to %70-%80 accuracy with very long training times. The trainer would have to be left for hours before it would get to this point, and sometimes during that time would hit infinities, throwing away all the time investment. Aside from this being poor accuracy on the house number test data, it was even worse on the real-world temperature readout and was totally unusable.

During training it was discovered that increasing the learning rate over time was possible. If the learning rate was any greater than 0.002 it would lead to exploding gradients and infinite errors. However, it was found that over time the learning rate could be increased without this happening. Increasing the learning rate increased the speed at which the network would train, but the trainer had to be careful not to increase the learning rate too much or the errors would explode. Overall, this sped up training time quite a bit, but now required a user to babysit the program throughout training. Eventually it was changed so that the learning rate would automatically increase slowly over time. The amount was played with for a while until a good balance of speed and safety was reached.

In order to deal with the exploding gradients causing infinite errors and damaging networks during training, an auto-reload feature was developed. If a point was reached where any of the numbers became NAN, the program would automatically reload the last saved network model and turn down the learning rate. At this point the long learning could be totally autonomous.

Eventually while combing the network structure, it was discovered there were some coding mistakes in the backpropagation that was causing sections of the network to not be trained. This dead weight was causing much of the networks training slowdown as the rest of the network would have to slowly learn around these sections. Once this was fixed, training time was decreased to something closer to an hour whereas it was several hours prior.

After this the dataset was changed to the MNIST dataset combined with the data harvested from the app. This had a much smaller decrease between accuracy on the training data and the real-world roasting data. However, the higher the proportion of data came from MNIST the more the real-world accuracy would drop off. This was theorized previously in the report on being because of the sometimes-massive differences between the handwritten digit and the 7-segment digitally created digit. Sometimes these digits would be entirely different symbols, as with a handwritten 2

and a 7-segment 2 which more resembles a backwards s. In these cases, the training data was useless and actually detracted from accuracy. Once a good balance of training data was reached accuracy was eventually brought up to ~90%. This was where real-world useability was eventually achieved using post-network error correction. Even at 90% this was still around 1 in 10 number guesses being incorrect, which could heavily distort the graph.

Finally, there were some mistakes in the calculus used in the error where it wasn't being divided across inputs correctly, and the activation function wasn't being applied on the output neurons. After this was fixed starting learning rate could be turned up to 1. Training time and accuracy were significantly increased. Over 90% accuracy could be achieved in minutes instead of an hour, with 50% achieved in under 30 seconds. Batch size was also decreased to 16. After these final fixes and optimizations, a final accuracy of 98% was achieved.

2.7. Implications of Implementation

Implications of this project are:

Any user will be required to use an Android device. Though it would be ideal to have the app available on both main cellular operating systems it was not feasible in this project to produce both. Although I have some experience developing on IOS there would have been a substantial amount of learning involved in producing the app in both. From what I have read online Android is by far the most used cellular operating system, but unfortunately the segment of the population using IOS will be lost as potential users. This is aimed to be developed in the future if work on the project is continued significantly.

Roasters outside of the company which I will be required to have a sufficiently similar output on their roasting machine. How similar is unknown as I did not have access to other machines to test. I was able to get decent readings off the time display on my microwave and oven even despite the ':' symbol added, so the image recognition will have some amount of cross useability. It is definitely planned in the future to expand the recognition either through increased datasets from interested users, and/or converting the project to using higher quality widely used image recognition libraries.

Roasters wishing to use the app will be required to find some way to have their camera aligned with the readout on their roasting machine. I leave this up to the individual user but may be difficult in some situations.

2.7.1. Innovation

While coffee roasting apps already exist, they are quite basic. This project adds the innovation of computer vision for accurate, constant and automated temperature/time graphing, as well as the functionality to upload and share roasting information. Some newer roasters have a computer built in for this sort of data collection, but many roasters, including the one which I work, with do not. This will give accurate temperature data throughout the entire roast, and cover if the roaster misses any data collection point. Through my research I could not find an existing app for coffee roasting which utilized image recognition to wirelessly record temperature information from the roaster's

temperature readout. In addition to temperature data, this will allow the ability to overlay actions taken during roasting upon that data. Having this information will also allow the roasters better context within the temperature graph and how the graph changes at different times based on adjustments to temperature, airflow, or drum speed.

This project was taken as an opportunity to explore convolutional neural networks from start to finish. This was something that I've desired to delve into for a long time, and this gave me an opportunity to take the large time investment that was required. This was somewhat experimental as it was not known if completing it from start to finish was feasible within the time frame of this project.

2.8. Complexity

The problem is difficult to complete for multiple reasons. Firstly, because it is using a technology which is new to me and quite complicated. Secondly it is being coded from the beginning with no premade libraries, all math, data structures and pre and post-processing of data is done wholly by me from scratch. Finally, it is also difficult because it is not a clear-cut task. While the objective of turning an image into an integer is clear, and the use of machine learning is also clear, the process within that chosen solution is not, and requires a great deal of tinkering to find something that works adequately.

This project is out of the range of a diploma level student mainly because of the scope of its entirety and the complexity of the design and development of the image recognition's convolutional neural network from the ground up. While a diploma level student could complete the server, database and UI of the app, bundling all these together may be too much for a lone student to complete on their own.

Convolutional Neural Network: The convolutional neural network used in the image recognition would be too difficult to create by a diploma student without the help of outside libraries. Calculus was not really covered when I took my diploma, and even in the degree was covered quite briefly. Aside from the calculus there is the huge amount of complexity in the interconnectedness of the structure of the network which I believe would be too much for a diploma level student. On top of this it must be asynchronous in order to access camera hardware and run the image recognition without crippling the UI and without errors in data access.

The convolutional neural network can be broken into several individually complex components as well:

- Acquiring a useable data set
- the preprocessing of the images and figuring out what works best to enhance image readability
- Forward and back propagation and all the math and variables within that
- Data post-processing and error correction

Networking: The app utilizes a remote server accepts and distributes information used in roasting. This has a separate database from the one in the android app, so two databases were required to be developed. App data was stored in an object structure, converted to a map, which is converted to a string, posted to the server where it is then converted into a SQL query and finally stored in the database. Server data is pulled from the database, downloaded, placed into a JSON object, then each of the fields are copied into the applicable object representing that table, which is then passed to the local database. Some of these uploads and downloads contained data that spanned several tables which needed to be joined together. For example, a Blend consists of data from the blend table, multiple entries from the roast table connected through a junction table, each roast entry contains a reference to a bean table entry and potentially many checkpoints which also have a junction table. All this had to be kept organised for both upload and download on server and client side.

File types: Custom file types were created to store and transfer the neural network model, and for saving roast temperature data. These required conversions to and from byte arrays representing the float datatype that was used to store both the temperature and network weights. These byte arrays order had to be very strictly adhered to as it represented the positioning throughout the neural network and any change in order could completely ruin the accuracy of the network model. Endianness was something that also required attention. The files coming from the C# program were little endian and had to be converted to big endian for the android app to utilize them.

Concurrency: Multiple threads were required for the roasting and image recognition. One thread was the main thread used for the user interface, one thread for the camera hardware, one for the image recognition, and one for everything else.

The camera thread would loop checking if the last image was unloaded, and if it was, it would load a new one, convert it to a bitmap, set the shared image to it, and set the flag to loaded.

The thread handling the image recognition would call a function that would check if a new camera image was loaded. If it was, it would copy that image and then flag it as not loaded. This flagging prevents the two threads from altering the image data and causing errors. After that the image was preprocessed, separated into three digits, load each into the neural net, forward propagate each three times, average them, combine them back into a single three-digit number, this number is saved in an AtomicInteger object which is shared with the main class. AtomicInteger forces threads to wait if another is currently reading or writing, in case they are both reading and writing simultaneously.

The third thread handles keeping track of time and updating the temperature text, graphing, warning alarm and checkpoint triggering. It is the other thread which has access to the temperature AtomicInteger. This thread runs every half second, at which it adds the current temperature and time to the roast temperature array, checks this temperature against the next approaching checkpoint, sets off the warning alarm if it is within five degrees, applies the median filtering to the graph data and then refreshes the graph data.

2.9. Research in New Technologies

For this project I had to fully learn how convolutional neural networks worked and the math behind them. I hadn't created anything with machine learning prior to this project and only had a high-level idea of how machine learning worked. I was not aware of how the convolutional portion worked, only that it was generally what was used in image recognition. Some studying had been done during the writing of the project proposal, but there was a great deal of gaps in knowledge which had to be filled during development. Creating the neural network was hard as it is so difficult to debug any mistakes which may be in any part of the large process of forward and back propagation. I haven't been exposed to very much calculus, and there were often small, overlooked parts of the math which had to be cobbled together from many different explanations which each brushed over different portions of the explanation.

I had many problems with exploding gradients while tuning the training. Adjusting the image recognition was very time consuming as I had to make adjustments, potentially retrain models, bring the app to my job to test it and then wait until I was off work before I was able to change anything in the code. I eventually managed to do some testing at home but was having a lot of trouble getting an image from a computer monitor that looked the same as the real roaster's readout.

A great deal of research into the math and concepts behind convolutional neural networks was done for this project. Starting with almost no knowledge I am now very familiar with convolution, forward propagation and backpropagation. It was very difficult to get my head around all these concepts with just theory, so an excel version of a neural network was created not as a deliverable, but to exercise and work through my knowledge prior to designing the actual software.

				Bias	0.35	0.6									
								Backpropagation							
								ne#1/w1	ne#1/w2	ne#2/w3	ne#2/w4				
								0.1493152639	0.1988325279	0.2491948599	0.2983897197				

2.10. Future Enhancements

Future enhancements would mainly include improving and widening the ability of the image recognition. In addition, making the app available to the other roasters online to provide feedback on which features are missing or require improvement. Unfortunately, there was no access to other roasters locally, so user testing was kept in the area of user interface and bug discovery without the insights of useful additions. As for improving and widening the ability of the neural network, that would likely involve switching to a more widely reliable image recognition library which would hopefully do a better job on various roasting machines which I wouldn't be able to personally test on and fine tune. Though it is not known how well the current state would work on other machines until it is tested and may be acceptable. Depending on the library this may also require asking users to submit images of digits from their roasters to be used in training, although more likely a pretrained image recognition library specifically designed for digit recognition would be used.

2.11. Timeline and Milestones

Section	Work Performed	Hours Worked
1	User Interface	55
	Create Main Menu UI	10
	Roast, Bean, and Blend Parameter UI's	9
	Remote Data Browser UI	4
	Roasting Activity	12
	Previous Roasts and Previous Roast Data Viewer	5
	Checkpoint Activity and Classes	4
	Created Options Menu	5
	Camera Calibration Activity	6
2	MYSQL Database	69
	Implemented Database Schema	13
	Added Android Functions to Insert and Get from DB Tables	11
	Created Android Classes to represent DB Objects for Sending and Pulling From DB	6
	Created Android Functions for Database Interactions	9
	Roast Metadata Table and Android Functions Added	8
	Converting DB Roast Data to Buttons on UI	6
	Check If New Checkpoint Entry Already Exists in DB Prior to Insertion	2
	Created DB Table and Android Functions for Roasting Machine	5
	Drew Up New Database Schema with Previously Overlooked Data	9
3	NodeJS Server	44
	Bought and Set Up Linux Server	6
	Learned NodeJS and Created Skeleton of Request Functions	23

	Fully Implemented Request Functions	15
4	App Implementation of Server	35
	Created Android Classes Representing Server Data with Functions Translating them to HTTP Requests	12
	Final Server Work and Debugging	23
5	Desktop Neural Net Trainer	182
	Further Studied CNN's and Created Simple Excel Neural Network to work through Math	28
	Wrote Neural Net Structural Functions	46
	Studying, training and Restructuring Until Network Learning Was Achieved	60
	Creating File Structure and Saving of Network Model for Use in Android App	8
	Tweaking and Adjusting Network Structure and Training	24
	Restructured. Retrained and Tweaked Network Model with Eleventh Output Representing "Not a Number"	16
6	Android Temperature Recognition	117
	Created Network Model Classes and File Loading Functions	12
	Obtained Android Camera Hardware Preview and Bitmap	23
	Created Handler to Asynchronously Update UI, Temperature List, and Temperature Post Processing	7
	Created NeuralThread to Asynchronously Handle Temperature Image Recognition	15
	First Working Iteration of Neural Network	12
	Camera Hardware Zoom	5
	Image Data Preprocessing	16
	Checkpoint Triggers and Storage	6
	Tweaking, Optimizing and Output Filtering	21
7	Graphing	5
	Imported and Implemented Graphing Library	3

	Graph Overlay of Multiple Roasts	2
8	Saving Roast Data to Disk	7
	Planned and Implemented Custom Filetype	3
	Loaded Data from File Displaying on Graph UI	4
9	Export .xls Spreadsheet	10
	Select Spreadsheet Library and Troubleshoot	8
	Create Spreadsheet Output Format and Input Relevant Data	2
10	Options and Menus	9
	Deletion and Upload Submenus on Main Menu Entries	3
	Options menu	2
	Global Settings Singleton Class and Settings Implementation Throughout App	4
11	Camera Calibration	12
	Rewriting NeuralThread Class to Be Useable by Both Roast Activity and CameraCalibration Activity	7
	Create Handler to Adjust Image Input Over Time and Display Adjusted Input and Guesses	5
	Total Hours	545

3. Conclusion

Development expanded upon and brought together many different skills and areas of software development into one. Both desktop and Android development were utilized. Databases, networking, image manipulation, machine learning, calculus, concurrency, and custom file output were all covered and expanded upon what was known.

3.1. Lessons Learned

This project allowed me to apply and greatly expand upon what I have learned during my time at BCIT. My chosen specialization was the Wireless and Mobile Applications Development option. This project being mainly developed in Android it came into use quite a lot.

During the degree we practiced putting together and connecting to a Tomcat server. This was expanded upon during the project in the development of the NodeJS server. We used HTTP requests in class, and the same was done with the NodeJS server. This was however a new server technology and used JavaScript which I did not have much familiarity with. It also gave me the opportunity to deploy a server to an externally hosted server which I had never done before.

We had some experience working with Android hardware, using the gyros and other things. In a similar area this project required access to the Android camera hardware. Getting access to the hardware itself was not difficult but turning the data from it into something useable was quite challenging and a good learning experience.

Before the part-time degree option, I was in the Games Development option full time. Even from this the calculus learned was applied and expanded upon.

A great deal was learned about the way neural networks function. I had a general conceptual idea, but this project gave me a reason to learn the math and small details that go into creating an image recognition system. I had wanted to learn these small details for a while but couldn't commit to the time required to really learn it well until this project.

3.2. Closing Remarks

This project was a difficult but fulfilling experience. It was very stressful trying to get the image recognition working, and at times I wasn't sure if I would be able to complete it but, in the end, I am glad that I did and learned a lot through the process. I've been wanting to explore machine learning for a long time and was happy to have an opportunity to do this.

4. References

First understanding of neural networks was obtained from this video series by 3Blue1Brown:

<https://www.youtube.com/watch?v=tIeHLnjs5U8&t=174s>

Deeper understanding of the specific math involved was obtained through this tutorial:

<https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

Backpropagation of convolutional layers was learned from this tutorial:

<https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c>

5. Change Log

- Paragraph structure throughout document: Added indentation and spacing between paragraphs.
- Removed all images not created by myself.
- Page 3 section 1.2 Project Description, essential problems, goals and objectives were rewritten to be more succinct and clearer.
- Page 4 section 2 Body, Background, Project Statement trimmed of unnecessary information and adjusted to be clearer.
- Added studying references.
- Page 17 Backpropagation Simplified and abbreviated the explanation of the backpropagation. Removed much of the mathematical explanation.

6. Appendix

6.1. Project Supervisor Approvals

To the Major Projects Committee:

I have seen the demo for Robert's project, and reviewed the report. This is an interesting project, and it is noteworthy that he implemented the learning software from scratch – which is a worthwhile and challenging task for the project. It is my view that he has done a nice job, and he has satisfied all of the requirements of the Major Project. I therefore recommend that he forward the report to the committee for final approval.

Aaron

6.2. Approved Proposal