Embedded Facial Detection and Identification :

Deployment of Tensorflow Lite and OpenCV on the Raspberry Pi 3 Model B+

Andrew O'Shei
Master 1 Informatique
Université Paris 8

12/1/2022

Abstract

With improvements in efficiency over the past few years it has finally become possible to deploy machine learning algorithms on low powered and embedded systems. The ability to run machine learning inference models on these systems has opened the door to many new innovations across a variety of applications in IoT, Self-Driving Vehicles, Robotics, Agriculture and Industry. However, deploying machine learning models on embedded systems is not without its challenges. In this report we will be presenting a practical deployment of machine learning models (Haar Cascade and ResNet50) for performing facial detection and identification on a low powered system, the Raspberry Pi 3 Model B+. We will present the challenges we encountered as well as some solutions and considerations for those who are considering building similar applications.

1 Introduction

With a 1.4GHz quad-core SoC, 1Gb DDR2 RAM and integrated Wifi card the Raspberry Pi 3 Model B+ has maintained its place as a very capable consumer-grade embedded platform since its launch in 2018. Its capability and relatively low cost has led to wide adoption among hobbyists, universities and commercial industry. As such there are no shortage of interesting applications using this platform. However, despite its ubiquity, the need for embedded AI and

machine learning models has seen the release of platforms such as the NVidia Jetson, Google's Coral Dev Board and the Raspberry Pi 4 which are beginning to supersede the Raspberry Pi 3 Model B+ due to their expanded capabilities in this domain. That being said, The Raspberry Pi 3B+ is still likely the most affordable option for prototyping embedded machine learning and edge computing applications. We will demonstrate through our project that for many applications the additional capabilities (and cost) of newer platforms may not be necessary in order to get your Machine Learning-based projects off the ground.

The application we chose for this project was dictated by the materials available. We were provided with a HC-SR04 Ultrasonic sensor and a Raspberry Pi Camera module so naturally we decided to implement a computer vision based application. Some of the requirements for this project also came under the directive of our Professor.

The design phase our application had four core requirements :

1. Interface with the hardware components
2. Implements a machine learning model
3. Read and Write data to a database
4. Run a server that allows remote clients to access the data

2 The Application

Our application is based around the concept of a security device that will raise an alert when it detects the presence of an individual and simultaneously determine if the individual has the proper credentials to access the location under surveillance. In order to achieve this we will use the ultrasonic sensor for initial object detection. When the ultrasonic sensor detects an object in range the camera will then capture an image of the detected object. The captured image is then processed first for face detection and then, if a face is detected, for face identification. The results are then logged on a database. In a separate process a web server is running which allows for remote clients to connect and read from the database (See Figure 1).

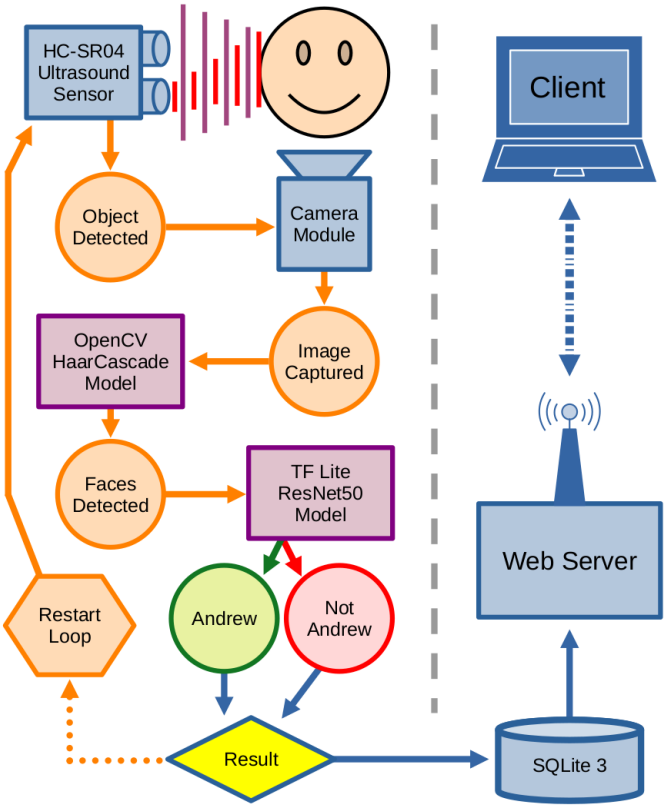

Figure 1: Application program loop

3 Machine Learning

For facial detection and identification our application employs two machine learning algorithms, the Haar Cascade model (FrontalFace_Alt2) and a ResNet50 image classification model. We had considered using a single model however, using the Haar Cascade to isolate and detect faces not only improves the quality of images fed to the image classification model but it also allows us to more easily classify images containing multiple faces. This also potentially improves performance while simultaneously freeing additional resources as each model can be loaded into memory independently and the discarded after finishing its task. However, this does add more latency to the program loop. Functionally, the Haar Cascade model is used to detect individual faces, which are then automatically cropped and scaled before being sent to the ResNet50 model for identification.

3.1 Haar Cascade Classifier

The Haar Cascade Classifier algorithm is a machine learning model for classifying elements within an image. It operates by dividing images into negative and positive space with positive space containing the detected feature and negative space containing everything else. It is a powerful technique for extracting features from an image[4].

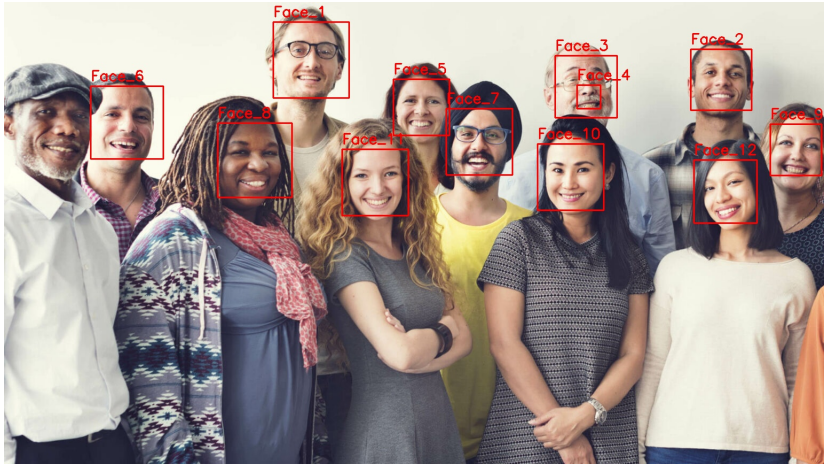

Figure 2: Faces detected using the Haar Cascade *FrontalFace_Alt2* Classifier, Original image source: martech.org

The Haar Cascade Classifier used in our application is the pre-trained model available with the OpenCV image processing library. Specifically we deployed the *FrontalFace_Alt2* model for its generally good performance with low false positives across a range of images. One issue with this specific model in the context of our application is that, as its name implies, the model only detects faces that are positioned frontwards towards the camera. This means faces at angles or images of heads in profile will not be identified by this model. Normally, this problem is overcome by deploying multiple Haar Cascade models and then aggregating the results of each. The problem is that using multiple models is memory intensive. On an embedded system one must decide to either compromise speed, by loading each classifier individually and then discarding them when it has finished, or compromise in feature detection. For our application we chose to only employ a single Haar Cascade model for speed.

3.2 ResNet50 Image Classifier

The ResNet series of deep learning models were originally published in 2015 by a team from Microsoft Research, led by Kaiming He. They also won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) of that year with this model (Specifically, the ResNet152). ResNet is short for Residual Convolutional Neural Network. The primary way ResNet differs from a traditional CNN is that it adds an additional mechanism to the model called a Residual Learning Block.

A common problem in many neural network models is that dataset features, which appear with a high frequency, are often over-represented in the final trained model. Dropout algorithms, for example, are one approach to addressing this problem. The Residual Learning Block is another approach, it operates in a similar fashion to the Long-Term Short Memory Cells found in some Recurrent Neural Network models. The basic idea is to provide a mechanism which allows less prominent features to carry forward through the layers of the neural network thus maintaining a balance in the feature representation of a trained model[1].

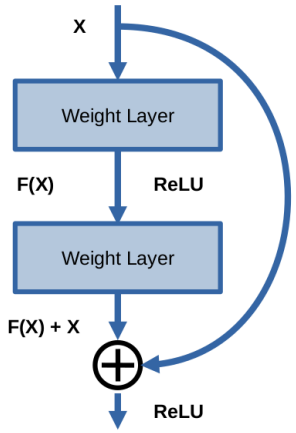

Figure 3: Residual Learning Block

The specific model we used was the ResNet50, which as its name suggests, is a 50-layer residual neural network. The ResNet50 was selected due to its good performance when compared to the other models we tested. However, the ResNet50 is not necessarily the ideal model for use in an embedded system as it is more memory intensive than traditional neural networks. The ResNet we used was trained using Tensorflow. For deployment on the Raspberry Pi we converted the trained model to Tensorflow Lite, which employs several optimizations in order to deploy models on low power and resource constrained systems.

4 The Dataset

The ResNet50 was trained on a dataset consisting of 368 imagers split into two classes, "Andrew" and "Not Andrew". Similar to the training method used in the Haar Cascade model, the dataset was split into positive and negative classes of images. The positive class images were images of myself, as the desired goal was to train the network to distinguish myself from a potential intruder. There are 100 unique images representing the class "Andrew" and 268 images representing the negative class "Not Andrew". Each image in the dataset is 250 x 250 pixels and contains an image of a face. Images represent a variety of face angles and include examples of faces wearing glasses, hats and masks. There are also images both in color and grayscale. Although I used some images from my personal collection, most of the images in the negative class were obtained from Google's *Facial Expression Comparison Dataset*[2].

Figure 4: Dataset example: Positive Class "Andrew" (Top), Negative Class "Not Andrew" (Bottom)

4.1 Dataset Augmentation

Although a few hundred images might seem like a good sized dataset we were unable to obtain good results when training models on the base dataset of 368 images. The models I trained were too prone to over-fitting and and gradient descent. As such they did not produce good results when tested against real world detection samples. Generally the models gave all positive or all negative results regardless of the input. We struggled with this for some time, however after regarding the Tensorflow documentation on Image classification (https://www.tensorflow.org/tutorials/images/classification) and a few real world examples, such as D. Nikolaiev's self-published "me-not_me" face detector[3], we were introduced to the idea of dataset augmentation. Augmentation of an image based dataset, by manipulation of images through basic image processing techniques, can in some circumstances enhance results in situations where it is simply not possible to go out and collect more data to include in the dataset. In our case it greatly improved the end result.

Figure 5: Dataset Augmentation example: Original image at top left

Data augmentation of image-based datasets involves employing operations such as, flip, crop, scale, skew and rotate, to create multiple samples from a single data point. Tensorflow has a built in operation called a generator, which allows the automatic application of these techniques when training a network. However, we had better results after applying these image augmentation techniques

manually via an OpenCV script. Image manipulations were applied at random and then empty space was filled by using the nearest neighbor algorithm on the image's pixels adjacent to the empty space. However, it is not a perfect solution. If you take a look at the image on the bottom right in Figure 5, you will see an example of an undesirable result. Artifacts such as this should be cleaned up as it creates an unnatural distortion in the image that could negatively impact training results. As you may have noticed, this artifact was created because the original image was poorly cropped.

Data augmentation can also go wrong in a few other ways. The first is that it is difficult to create a validation set from the dataset of augmented images. This is because a random split will likely include multiple versions of the same image in both the training set and validation set. The commonality between features can bias the final result of the trained model. This bias can lead to overfitting of the model.

5 Training Results

The final dataset with augmentation came to 1668 images, with 600 images of *Andrew* and 1068 images of *Not Andrew*. The final dataset was split 50/50 into training and validation sets, with a random selection of images for each set. Using the Tensorflow early stopping mechanisim was used to stop training when the validation loss bottoms out. The best result averaged out to around 50 epochs. The results of the ResNet50 model are as follows :

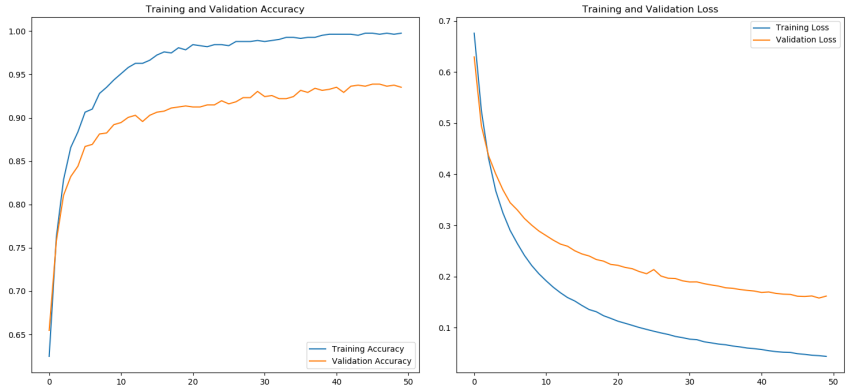

Figure 6: Training results after 50 Epochs

I was unsure of the result so I prepared a small post-validation set of 40 images (20 of *Andrew*, 20 of *Not Andrew*) that were not present in the initial training

and validation sets. After testing the trained model I received the following results :

- Post Validation Accuracy = 90%
- Post Validation Recall = 1.0
- Post Validation Precision = 83%
- Post Validation Accuracy = 0.9

Although we believe these results are good, considering the constraints, the post-validation dataset is too small to make strong claims about the performance of the model. With the validation set used to train the model we received a slightly higher accuracy with a score averaging about 93% after 50 epochs and 5 training runs. This result is also suspect as the validation set used in training likely contained multiple versions of the same image in both the training and validation sets. With all those disclaimers out of the way the model still appears to work pretty well in practice, although as of writing this we've only been able to test the application on three other people.

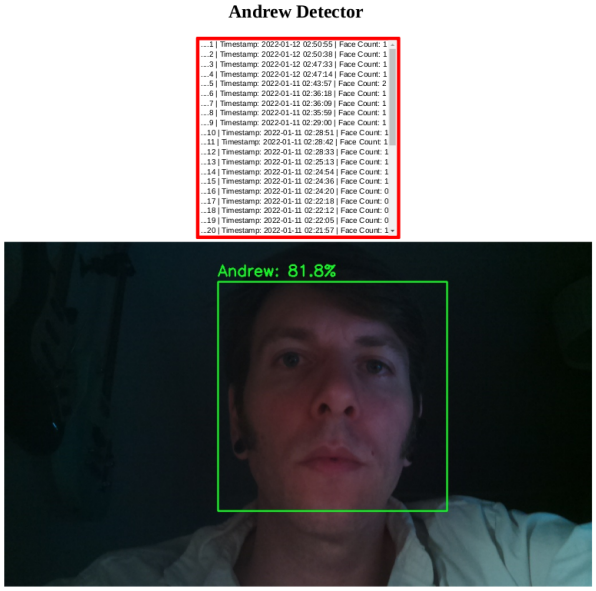

Figure 7: Application client side interface

6 Conclusion

Although we did encounter a few difficulties in the process of implementing this application on the Raspberry Pi, now that all the components are in place the real-world performance is surprisingly good. We believe that it is a a good test case for those considering this platform for use in a future machine learning based project. Given more time we would have preferred to build a more robust dataset and train a model that is more efficient than the ResNet50. There is still latency of around 2-3 secounds for the program to load the model. Additionally the project includes a mix of programs written in C and Python, moving some of the Python functionality over to C would likely further improve performance of the application. However, we believe a 2-3 second delay for model inference is perfectly acceptable for most applications. Maybe not for safety critical applications, but applications that deal with long term monitoring over time or that do not require real-time performance would have no problem with this level of latency. We will certainly consider the Raspberry Pi 3 Model B+ as a viable embedded machine learning platform in the future.

Sources:

[1] He K, Zhang X, Ren S and Sun J, *Deep Residual Learning for Image Recognition*, Microsoft Research 2015.

[2] Vemulapalli R, Agarwala A, *A Compact Embedding for Facial Expression Similarity*, CoRR, abs/1811.11283, 2018.
https://research.google/tools/datasets/google-facial-expression/

[3] Nikolaiev D, *Real-time 'me-not_me' Face Detector*, September 2021.
https://www.linkedin.com/pulse/real-time-me-notme-face-detector-dmytro-nikolaiev/

[4] Padilla R, Costa Filho CFF and Costa M, *Evaluation of Haar Cascade Classifiers for Face Detection*, ICDIP: International Conference on Digital Image Processing Vol. 6, April 2012.