EMBEDDED AMERICAN SIGN LETTERS RECOGNITION IN RASPBERRY PI

João Almeida
Joao.SilvadeAlmeida@haw-hamburg.de

HAW Hamburg, Dept. for Computer Science

Abstract

Foundation work for training an object detection model with TensorFlow for deployment on an Embedded Raspberry Pi system for recognition of American sign language letters from the camera's obtained video stream. Recognition is done per frame, without considering consecutive frame relations.

Keywords: computer vision, tensorflow, classification, raspberry pi, python, object detection, machine learning, opencv

Contents

1 Introduction

With the increasing reliance on digital technologies for everyday tasks, machines are expected to cover a bigger range of tasks than ever before. Artificial Intelligence is the field of study of what computers can be taught to do, that they couldn't do before. Object detection is a subclass of this field.

This project provides a proof-of-concept foundation for the application of object detection in the classification of American sign language letters from video obtained by a raspberry pi.

The objective is to obtain an accurate prediction, fast and reliably from the limited hardware provided by the raspberry pi.

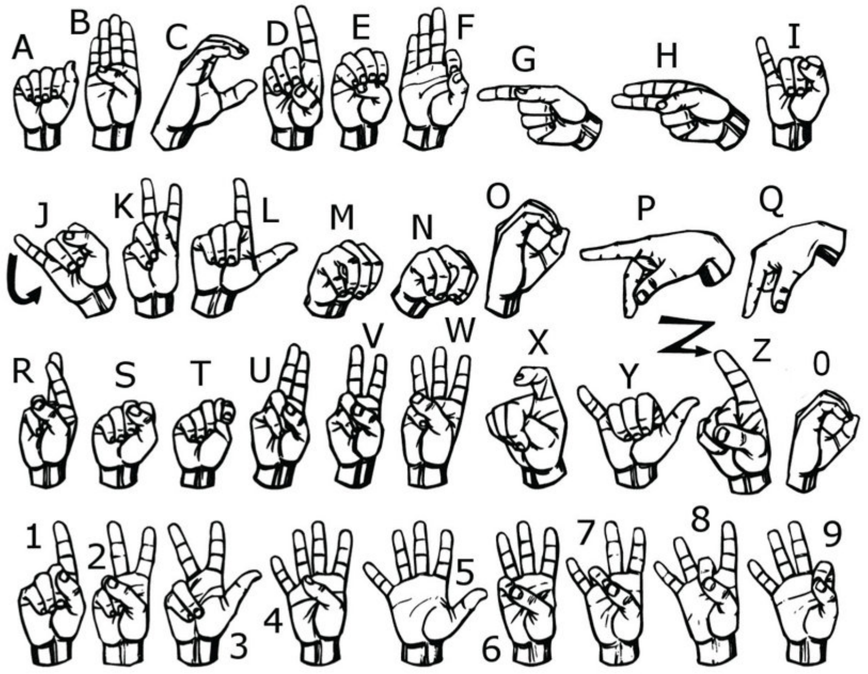

Figure 1: The 26 letters and 10 digits of American sign language

2 Approach and Architecture

2.1 Plan Overview

As with any object detection model training process, following the readthedocs official guide for training a custom object detector [**1**], the following steps had to be followed:

- Create or obtain an image dataset of letters in american sign language.
- Label all the images with information about the position of the hand in the image and the appropriate letter being made.
- Train the model with a dedicated graphics card.
- Training was done with TensorFlow and the SSD ResNet50 V1 FPN 640x640 pre-trained model [**6**].
- Convert the model into a TensorFlow lite model.
- Deploy the model in the embedded raspberry pi hardware.

The training of the model was done in a NVIDIA GeForce RTX 2060 graphics card.

2.2 Tools

As a means to an end, several architectural choices had to be made, here are some of the tools used:

- As a programming language, python was chosen due to its reputation with the machine learning community and its support for the several methods related to these technologies. More specifically, the language of jupyter notebooks was utilized, allowing for a conjunction of markdown and python code allowing a better structuring of the project.
- Conda, the CLI interface for Anaconda, allowing the creation of environments isolated from one another where several machine learning packages could be installed with specific versions, for easier debugging, for the Raspberry Pi OS, the similar tool used for this objective was virtualenv [**2**]
- The machine learning algorithm used was TensorFlow [**3**]
- Due to the large volume of images needed for each letter of the alphabet for a successful classification, in addition to the images obtained and labeled manually, additional images were used from this dataset from roboflow [**4**]
- The main Operating System used was Unix + GNU (Manjaro)

3 Training and Test Data-set

The dataset is split into 26 classes, each one representing a letter of the greek alphabet, from A to Z.

As mentioned, a mixed dataset of manually taken photos and already labeled ones from Roboflow[4] has been used.

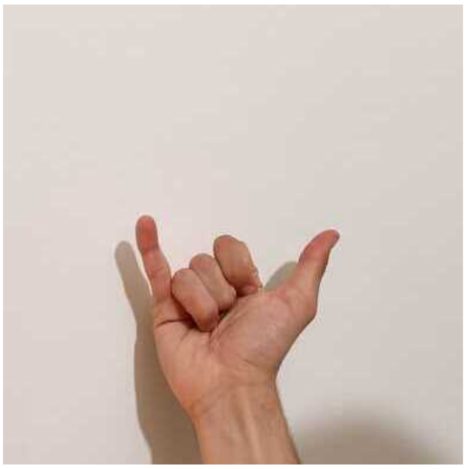

Figure 2: Dataset image "Y" - manually gathered

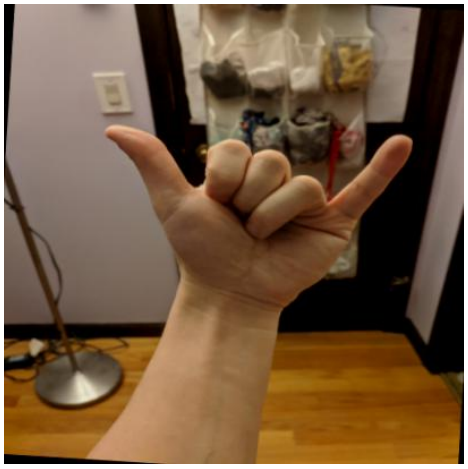

Figure 3: Dataset image "Y" - from Roboflow[4]

The images were gathered with a mobile device, and were downscaled and cropped to 390X390 to be easier to read and speed up the training process. These operations were applied with imagemagick [5] batch commands:

```
mogrify -resize 390 *.jpeg
mogrify -gravity South -chop 0x267 *.jpeg
mogrify -gravity North -chop 0x206 *.jpeg
```

For the labelling process of the manually gathered images, the program labelImg was used as explained in the readthedocs guide [1]

3.1 Image Augmentation

Some image augmentation techniques were used as a way to get more dataset images from the manually gathered ones. The ones taken from the internet had already had these techniques applied to them.

The techniques used for augmentation were horizontal flips of the images, and color conversion to gray scale. The labelling of the new images was done manually for the cases where the bounding box found itself in a different place from the original image.

```
mogrify -flop *.jpeg # to flip images horizontally
mogrify -colorspace Gray *.jpeg # to greyscale
```

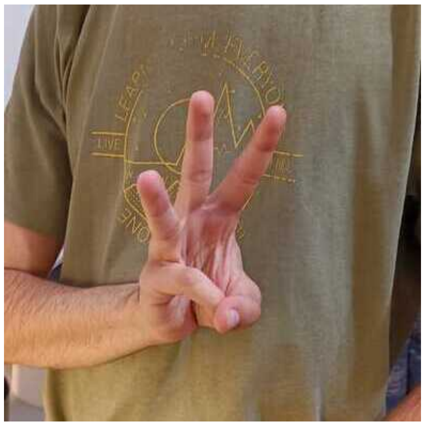

Figure 4: Dataset image "W" - Original image

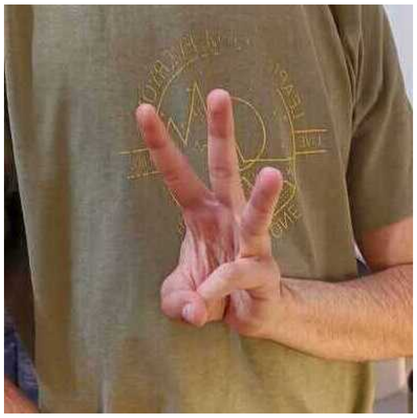

Figure 5: Dataset image "W" - Flipped horizontally

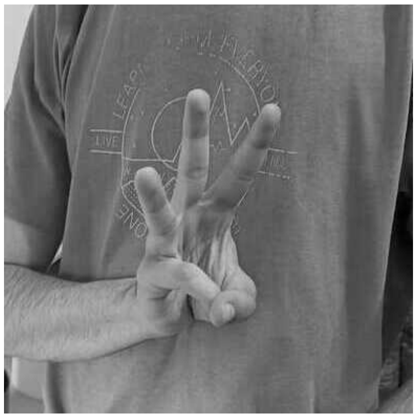

Figure 6: Dataset image "W" - Converted to gray scale

4 Training Process

Due to the limited hardware for training, the batch size was decreased to the maximum allowed, without running into the Out Of Memory error.

These are some of the more relevant parameters used:

- Total images for training: 1668
- batch_size: 4
- num_steps: 25000
- The data set was split 90% for training and 10% for validation

As in TensorFlow (without keras) we don't have a direct number for epochs, we can calculate them manually:

$1\ epoch = \frac{Total\ num\ train\ images}{batch\_size} = \frac{1668}{4} = 417\ steps$

Knowing that 25000 steps were used, and each span of 417 steps represents one epoch, we can calculate the total number of epochs:

$\frac{25000}{417} = 59.95203836930455 \approx 60$

Thus, our final epoch count amounts to roughly 60 epochs.

The loss always decreased steadily indicating a better performance, the learning rate was however not as predictable.

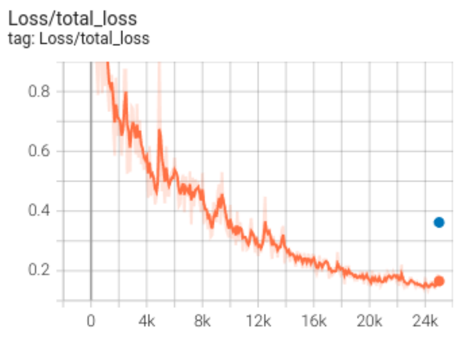

Figure 7: total_loss

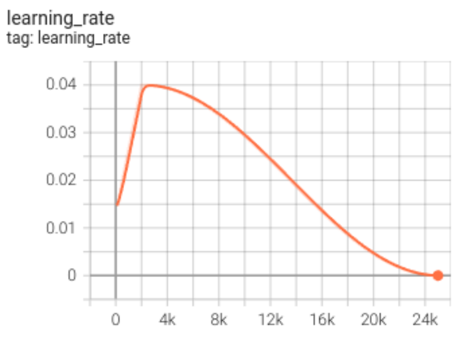

Figure 8: learning_rate

4.1 Testing and Conversion to TFlite

Before deployment, it was important to make sure that the model was performing up to standards, for that, another readthedocs guide was followed [7] and the laptop's web camera was used to test the model. The following images were obtained:

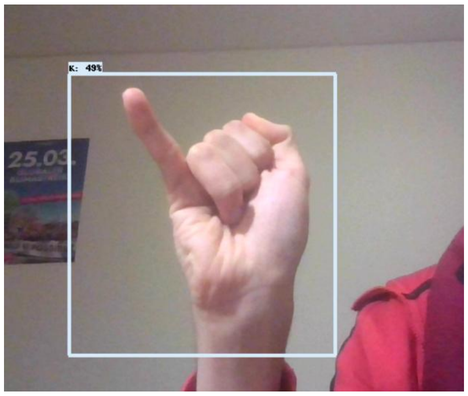

Figure 9: Dataset image "I"

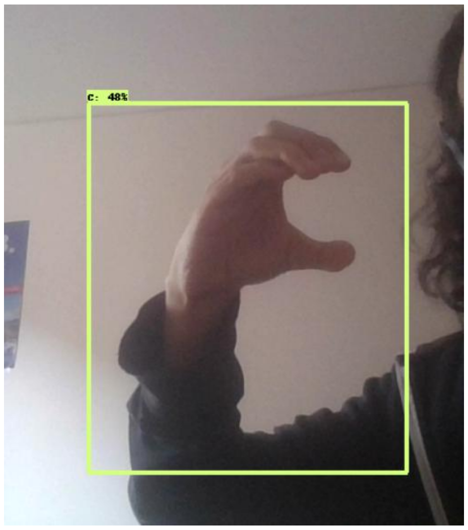

Figure 10: Dataset image "C"

As for the model conversion to TFlite, in order for it to be able to run on an embedded system, the linked guide was followed [8]

5 Deployment on the Raspberry Pi

5.1 Problems

The biggest problems encountered were software wise.

A new package, picamera2, was used, to be able to communicate with the camera through python, however, for the time being, installing picamera2 is a hassle due to the transition from the legacy camera to the libcamera stack on the raspberry.

To install it, we needed to build several dependencies, as explained in the raspberry guide followed [9]

The installation was successful as it was able to run google's example TFlite model provided.

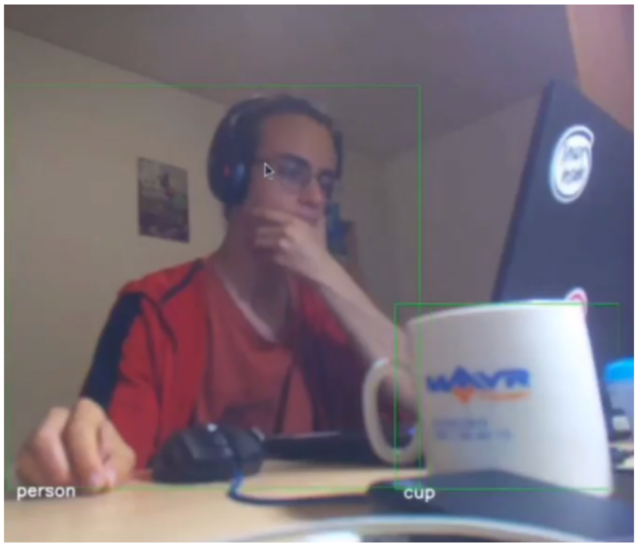

Figure 11: Google's TFlite example model

5.2 Application performance

The application was successfully deployed on the Raspberry Pi embedded system by importing the .lite model to the Raspberry Pi after its creation with the guide mentioned above [8]. It had, however, very high latency and its accuracy had decreased massivley, so much so, that in the present state, it would be hard to apply it in any practical environment.

6 Conclusions

Overall, the project was successful, all the steps of the process were explored and worked on, and the model is able to discern hands from the imagery it receives as input. With it we were able to prove that its possible to run these intricate object detection models even in not very powerful hardware and use them viably.

The biggest problem is the performance and accuracy of the model, especially once converted into the TFlite version for embedded systems, despite several attempts and guides followed to attempt to successfully convert the model to TFlite[**11**][**12**], none yielded desireable results, and more time and experimentation would be needed to find a suitable conversion method for the training methods used. Accuracy could likely also be improved with a better conceived dataset and a more thorough training and validating processes. Still, the final outcome of the project is positive as it works as a first increment and building blocks of what could be a much bigger implementation of the idea.

All project files, libraries used, and several more technical problems faced that were deemed unimportant to tackle in this report can be found in the publicly accessible repository for this project [**10**]

References

[1] Read The Docs. Training Custom Object Detector. (Access date: March 2022) [Online]

[2] Github. EdjeElectronics. How to Run TensorFlow Lite Object Detection Models on the Raspberry Pi (with Optional Coral USB Accelerator). (Access date: March 2022) [Online]

[3] TensorFlow. Machine-learning platform. (Access date: March 2022) [Online]

[4] Roboflow. American Sign Language Letters Dataset. (Access date: March 2022) [Online]

[5] ImageMagick. (Access date: August 2022) [Online]

[6] Github. TensorFlow 2 Detection Model Zoo. (Access date: March 2022) [Online]

[7] Read The Docs. Detect Objects Using Your Webcam. (Access date: March 2022) [Online]

[8] Google Colab. Convert TF Object Detection API model to TFLite. (Access date: March 2022) [Online]

[9] Raspberry Pi. Using the Picamera2 library with TensorFlow Lite. (Access date: March 2022) [Online]

[10] Github. SignLanguageDetection_TenserFlow_RaspberryPi. [Online]

[11] Github. TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi. (Access date: March 2022) [Online]

[12] TensorFlow. Model conversion overview. (Access date: March 2022) [Online]