Traffic and Road Signs Object Detection Project

By Ishan Palit

Advisor : Prof Srinivasan Radhakrishnan

NUID : 002927588

Abstract

Object detection is a computer vision task that involves identifying and localizing objects of interest in an image or video. The goal is to accurately identify the presence, location, and class of objects in an image. Object detection is an important task in various applications such as self-driving cars, security systems, and medical imaging. It involves several sub-tasks, including image preprocessing, object proposal generation, feature extraction, classification, and localization.
There are several popular deep learning-based approaches to object detection, such as Faster R-CNN, YOLO, and SSD. These methods use convolutional neural networks (CNNs) to extract features from images and then apply additional layers to detect objects. Object detection has significantly improved in recent years due to the advancements in deep learning and the availability of large-scale annotated datasets. The detection of road signs using object detection techniques is an important application of computer vision and has significant practical implications for autonomous driving systems, driver assistance technologies, and intelligent transportation systems.

In the project our aim is to detect road signs from a live video feed as well as still images and classify them into 4 categories - **'trafficlight', 'stop', 'speedlimit' and 'crosswalk'.** We are using TFLite based Deep Learning model for the detection and classification task.

Introduction

Road sign detection using TFLite involves using a deep learning model to detect traffic signs from camera images or video feed captured. TFLite (TensorFlow Lite) is a lightweight version of the TensorFlow deep learning framework, designed for mobile and embedded devices.We are using TFLite because we are executing the model real time on Raspberry Pi 4 Desktop Kit.

The process of road sign detection using TFLite can be broken down into several steps. First, a dataset of images of road signs is used to train a deep learning model. This involves feeding the images into a convolutional neural network (CNN) and adjusting the weights and biases of the network to optimize its performance.

Once the model is trained, it can be converted into a format suitable for deployment on a mobile device using TFLite. The converted model is then integrated into a mobile application that captures live video feed. The images or live video feed are fed into the model, which performs inference to detect any traffic signs present.

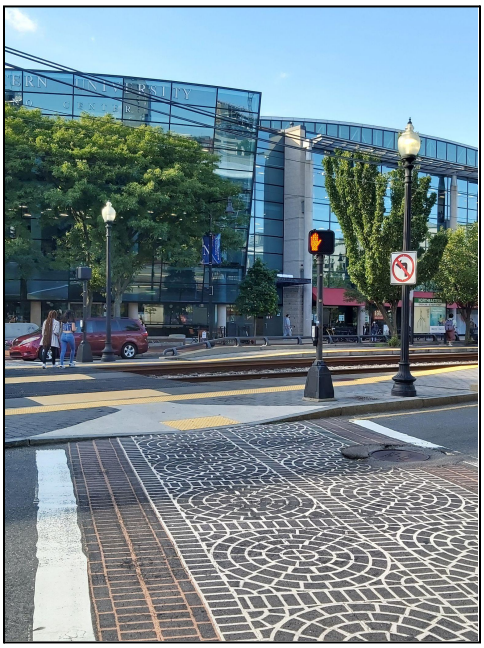
The output of the model is typically a bounding box around the detected sign, along with a classification of the sign's content (e.g. stop sign, speed limit sign, etc.). The application can then take appropriate action based on the detected sign, such as sounding an alert or providing visual feedback to the driver. One of the advantages of using TFLite for road sign detection is its

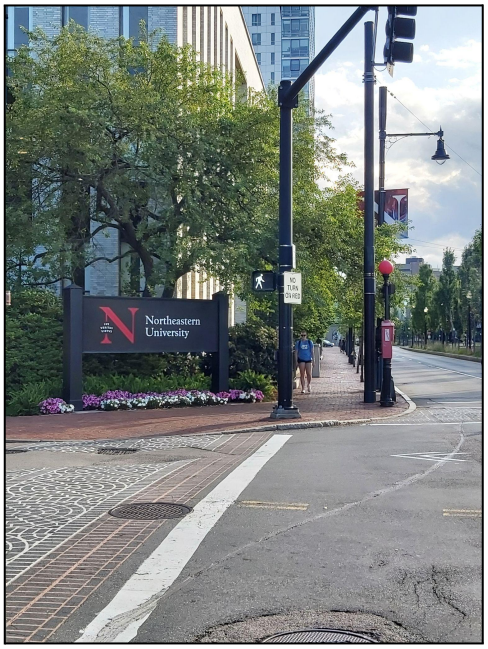
low latency and low power consumption, making it suitable for deployment on mobile and embedded devices. However, the accuracy of the model depends on the quality and quantity of the training data, as well as the design of the CNN architecture. Ongoing research is focused on improving the accuracy and robustness of road sign detection models using TFLite.

Data Collection and Labeling

Data Collection

For the data collection part of the project an image scraping Library is being used - called jmd_imagescraper. Apart from scraping images from the web - 100 images were collected from near the Northeastern University, Boston campus for training. A few sample images are shown -





To construct the training set - a set of 745 (85%) images were used. For the validation set - a set of 132 (15%) images were used and it was made sure that there are no overlaps in the data.

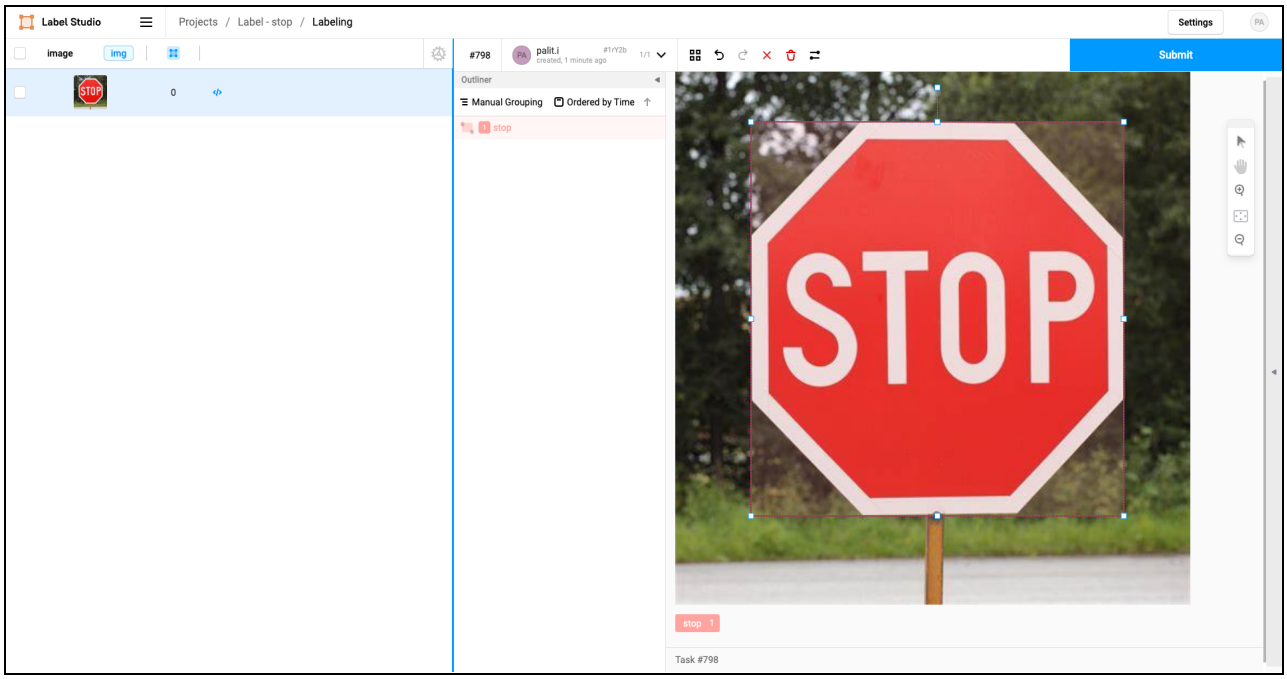
Data Labeling

For image labeling, Label-studio has been used.

Label Studio is an open-source data annotation tool that allows users to label images, videos, text, and other types of data for machine learning and other AI applications. It was created by the developer team at Heartex, and it is designed to be flexible and customizable to fit different data annotation needs.

When it comes to image labeling specifically, Label Studio provides a user-friendly interface that allows annotators to draw bounding boxes, polygons, points, lines, and other shapes around specific objects in the image. This way, the annotator can label objects such as faces, objects, or specific features within an image.

For the current project - bounding boxes have been used to label the images. A preview of the same is given below -



For the image labeling - the data is stored in the Pascal VOC format with the image and the annotation of the image with the same name being stored in XML format.
The PASCAL VOC format defines an XML schema for annotating images. Each annotated image is described in an XML file, which includes the image's filename, size, and a list of objects with their corresponding class labels and bounding boxes.

The format supports annotation of a wide variety of object classes, including people, animals, vehicles, and various objects. For each object in the image, the annotation includes the class label, the bounding box coordinates (x, y, width, and height), and optionally, additional attributes such as pose, occlusion, and truncated ness. The PASCAL VOC format has become a de facto standard for annotating images in the computer vision community, and many popular object detection and segmentation models have been trained on datasets in this format. It has been used in various applications, including object detection, object recognition, and semantic segmentation.

Model Training

The training and validation data is read using - **tflite_model_maker.object_detector.DataLoader** in the pascal VOC format.

- Images in `train_data` are used to train the custom object detection model. Here we use 745 images to train the model.

- Images in `val_data` are used to check if the model can generalize well to new images that it hasn't seen before. Validation is done on 132 images.

```
[ ]  train_data = object_detector.DataLoader.from_pascal_voc(
         'drive/MyDrive/Project_Test_Data/train_jpg',
         'drive/MyDrive/Project_Test_Data/train_jpg',
         ['trafficlight','stop','speedlimit','crosswalk']
     )
     val_data = object_detector.DataLoader.from_pascal_voc(
         'drive/MyDrive/Project_Test_Data/validate_jpg',
         'drive/MyDrive/Project_Test_Data/validate_jpg',
         ['trafficlight','stop','speedlimit','crosswalk']
     )
```

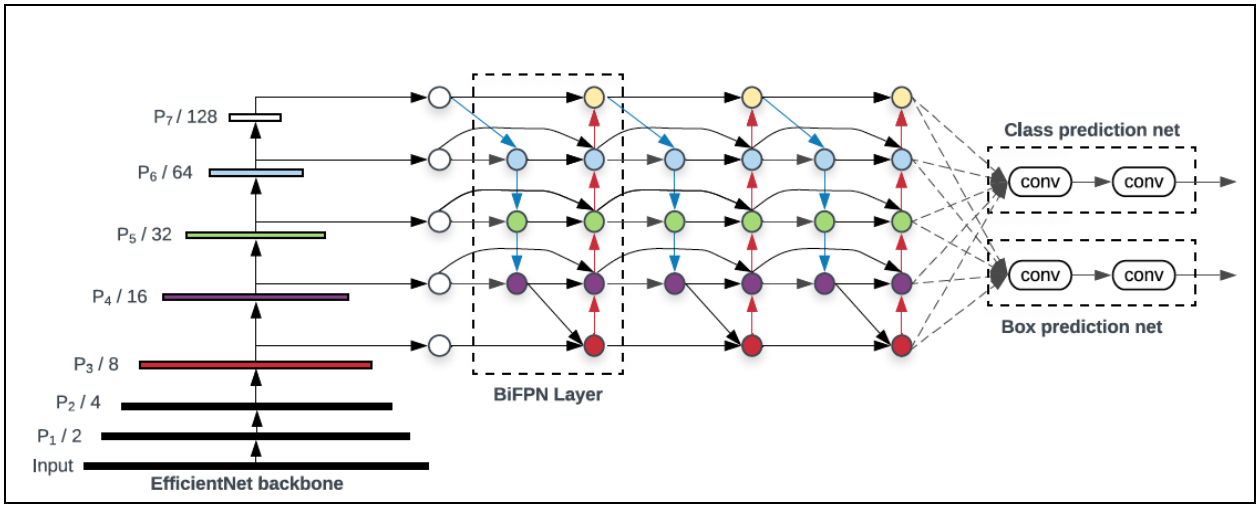
Model Architecture

EfficientDet-Lite[0-4] is a family of mobile/IoT-friendly object detection models derived from the EfficientDet architecture.

Model architecture	Size(MB)*	Latency(ms)**	Average Precision***
EfficientDet-Lite0	4.4	146	25.69%
EfficientDet-Lite1	5.8	259	30.55%
EfficientDet-Lite2	7.2	396	33.97%
EfficientDet-Lite3	11.4	716	37.70%
EfficientDet-Lite4	19.9	1886	41.96%

* Size of the integer quantized models.
** Latency measured on Raspberry Pi 4 using 4 threads on CPU.
*** Average Precision is the mAP (mean Average Precision) on the COCO 2017 validation dataset.

In this project, we use EfficientDet-Lite2 to train our model, as it is at a good tradeoff between Latency and Average Precision.



EfficientDet architecture – It employs EfficientNet as the backbone network, BiFPN as the feature network,and shared class/box prediction network. Both BiFPN layers and class/box net layers are repeated multiple times based on different resource constraints as shown in Table 1

	Input size R_{input}	Backbone Network	BiFPN #channels W_{bifpn}	BiFPN #layers D_{bifpn}	Box/class #layers D_{class}
D0 ($\phi = 0$)	512	B0	64	3	3
D1 ($\phi = 1$)	640	B1	88	4	3
D2 ($\phi = 2$)	768	B2	112	5	3
D3 ($\phi = 3$)	896	B3	160	6	4
D4 ($\phi = 4$)	1024	B4	224	7	4
D5 ($\phi = 5$)	1280	B5	288	7	4
D6 ($\phi = 6$)	1280	B6	384	8	5
D7 ($\phi = 7$)	1536	B6	384	8	5
D7x	1536	B7	384	8	5

Table 1: **Scaling configs for EfficientDet D0-D6 –** ϕ is the compound coefficient that controls all other scaling dimensions; *BiFPN, box/class net, and input size* are scaled up using equation 1, 2, 3 respectively.

Image source : https://arxiv.org/abs/1911.09070

- **Backbone**: EfficientNets is employed as our backbone networks.
- **BiFPN**: BiFPN is proposed, a bi-directional feature network enhanced with fast normalization, which enables easy and fast feature fusion.
- **Scaling**: A single compound scaling factor to govern the depth, width, and resolution for all backbone, feature & prediction networks.

The Augmentation Policy Used is - **policy_v0**

```python
def policy_v0():
  """Autoaugment policy that was used in AutoAugment Detection Paper."""
  # Each tuple is an augmentation operation of the form
  # (operation, probability, magnitude). Each element in policy is a
  # sub-policy that will be applied sequentially on the image.
  policy = [
      [('TranslateX_BBox', 0.6, 4), ('Equalize', 0.8, 10)],
      [('TranslateY_Only_BBoxes', 0.2, 2), ('Cutout', 0.8, 8)],
      [('Sharpness', 0.0, 8), ('ShearX_BBox', 0.4, 0)],
      [('ShearY_BBox', 1.0, 2), ('TranslateY_Only_BBoxes', 0.6, 6)],
      [('Rotate_BBox', 0.6, 10), ('Color', 1.0, 6)],
  ]
  return policy
```

Code source :
https://github.com/tensorflow/examples/blob/master/tensorflow_examples/lite/model_maker/third_party/efficientdet/aug/autoaugment.py

```python
spec = model_spec.get('efficientdet_lite2')
spec.config.autoaugment_policy = 'v0'
```

Training TFModel

- Setting **epochs = 50** , which means it will go through the training dataset **50** times. We will look at the validation accuracy during training and stop when we see validation loss (“val_loss”) stop decreasing to avoid overfitting.
- Set **batch_size = 16** here so you will see that it takes 46 steps to go through the 745 images in the training dataset.
- Set **train_whole_model=True** to fine-tune the whole model instead of just training the head layer to improve accuracy. The trade-off is that it may take longer to train the model.

```python
model = object_detector.create(train_data, model_spec=spec,batch_size = 16,train_whole_model=True, epochs=50, validation_data=val_data)
```

Sample of some epochs -

```
Epoch 46/50
46/46 [==============================] - 46s 994ms/step - det_loss: 0.1834 - cls_loss: 0.1340 - box_loss: 9.8824e-04 - reg_l2_loss: 0.0783 - loss: 0.2617 - learning_rate: 2.5252
Epoch 47/50
46/46 [==============================] - 47s 1s/step - det_loss: 0.1811 - cls_loss: 0.1308 - box_loss: 0.0010 - reg_l2_loss: 0.0783 - loss: 0.2594 - learning_rate: 1.2998e-04 -
Epoch 48/50
46/46 [==============================] - 44s 965ms/step - det_loss: 0.1755 - cls_loss: 0.1285 - box_loss: 9.3995e-04 - reg_l2_loss: 0.0783 - loss: 0.2538 - learning_rate: 4.8007
Epoch 49/50
46/46 [==============================] - 48s 1s/step - det_loss: 0.1828 - cls_loss: 0.1337 - box_loss: 9.8284e-04 - reg_l2_loss: 0.0783 - loss: 0.2611 - learning_rate: 6.9242e-
Epoch 50/50
46/46 [==============================] - 52s 1s/step - det_loss: 0.1803 - cls_loss: 0.1316 - box_loss: 9.7333e-04 - reg_l2_loss: 0.0783 - loss: 0.2586 - learning_rate: 6.9053e-
```

Evaluation on Validation Data

```
model.evaluate(val_data)
```

```
3/3 [==============================] - 24s 3s/step

{'AP': 0.64514923,
 'AP50': 0.78636855,
 'AP75': 0.7598101,
 'APs': 0.48135948,
 'APm': 0.8445239,
 'APl': 0.9590602,
 'ARmax1': 0.58477634,
 'ARmax10': 0.7166675,
 'ARmax100': 0.7193265,
 'ARs': 0.57237744,
 'ARm': 0.8945395,
 'ARl': 0.96666664,
 'AP_/trafficlight': 0.45070502,
 'AP_/stop': 0.66664034,
 'AP_/speedlimit': 0.86242515,
 'AP_/crosswalk': 0.6008265}
```

To evaluate a certain algorithm on instance segmentation, we see the terms mentioned above, namely AP (Average Precision) and mAP (Mean Average Precision).

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

We know the Precision and Recall formula as stated above where tp = true positive, fp = false positive, fn = false negative as derived from the confusion matrix.

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

Image Source : https://yanfengliux.medium.com/the-confusing-metrics-of-ap-and-map-for-object-detection-3113ba0386ef

The AP(Average Precision) is defined as the area under the precision-recall curve (PR curve). The x-axis is recall, and y-axis is precision. In the case of objection detection or instance segmentation, this is done by changing the score cutoff. Any detection with a score lower than the cutoff is treated as a false positive. At each unique level of classification score that is present in the detection results, we calculate the precision and recall at that point. After we have gone through all the precision-recall value pairs corresponding to each unique score cutoff, we have a precision-recall curve. It might look like the image below:

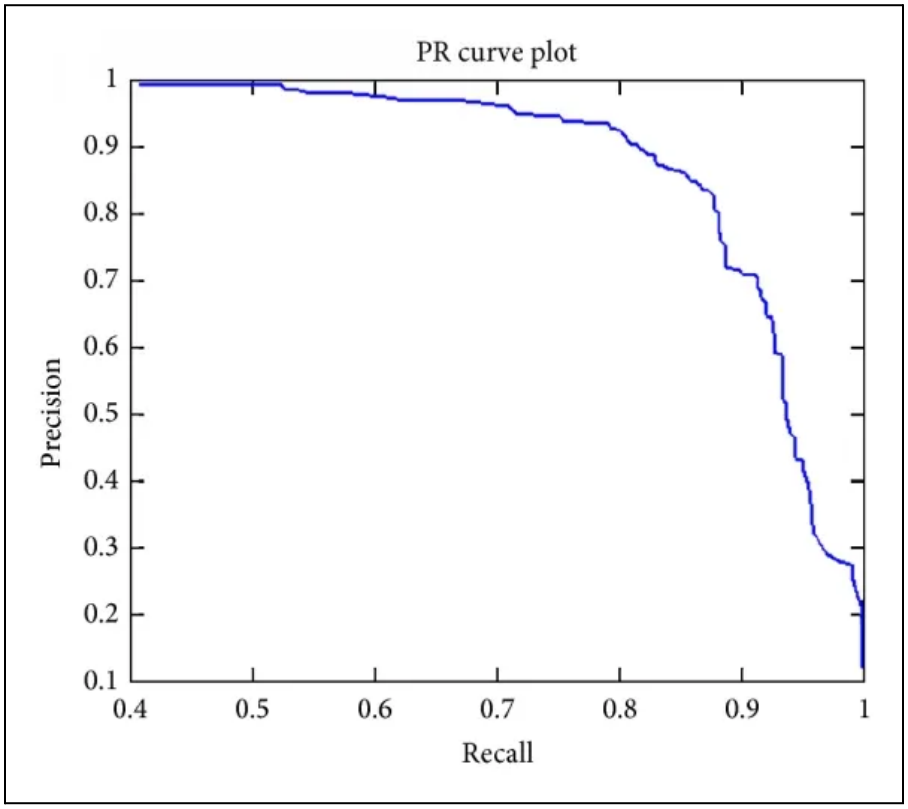


Image Source : https://yanfengliux.medium.com/the-confusing-metrics-of-ap-and-map-for-object-detection-3113ba0386ef

IoU is a good way of measuring the amount of overlap between two bounding boxes or segmentation masks. If the prediction is perfect, IoU = 1, and if it completely misses, IoU = 0. A degree of overlap will produce a IoU value between those two. We have to make a decision about when to call it a good enough prediction, i.e. true positive.

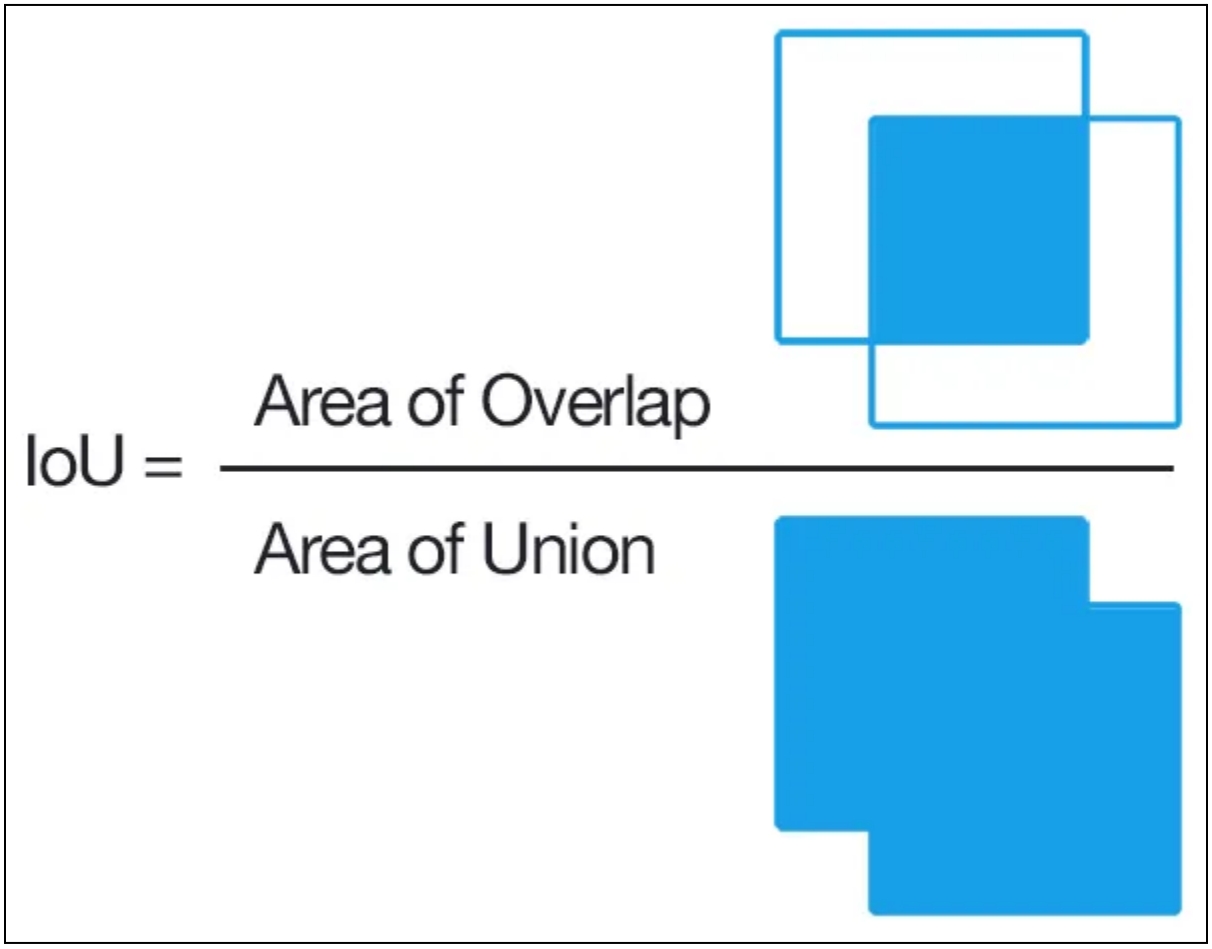


Image Source : https://yanfengliux.medium.com/the-confusing-metrics-of-ap-and-map-for-object-detection-3113ba0386ef

Sometimes the IoU threshold is fixed, for example, at 50% or 75%, which are called AP50 and AP75, respectively. When this is the case, it is simply the AP value with the IoU threshold at that value. We still have to calculate the precision-recall pairs at different score cutoffs.

Mean average precision (mAP) is simply all the AP values averaged over different classes/categories.

Testing Model on a still image to see output with Bounding Box

Setting labels as follows -

```
labels = ['trafficlight','stop','speedlimit','crosswalk']

# Code to use model
interpreter = tf.lite.Interpreter(model_path="road_signs_efficientdet_lite2_aug_v0_b16_ep50.tflite")
interpreter.allocate_tensors()
signature_fn = interpreter.get_signature_runner()
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
```

Setting the confidence level as 0.3 to put bounding boxes -

```
results = []
for i in range(count):
  if scores[i] >= 0.3:        # Setting the confidence level above 0.3 to show bounding boxes
    result = {
      'bounding_box': boxes[i],
      'class_id': classes[i],
      'score': scores[i]
    }
    results.append(result)
```

Code snippet to assign bounding boxes -

```
COLORS = np.random.randint(0, 254, size=(len(classes), 3), dtype=np.uint8)
for obj in results:
    # Convert the object bounding box from relative coordinates to absolute
    # coordinates based on the original image resolution
    ymin, xmin, ymax, xmax = obj['bounding_box']
    xmin = int(xmin * or_image.shape[1])
    xmax = int(xmax * or_image.shape[1])
    ymin = int(ymin * or_image.shape[0])
    ymax = int(ymax * or_image.shape[0])

    # Find the class index of the current object
    class_id = int(obj['class_id'])

    # Draw the bounding box and label on the image
    color = [int(c) for c in COLORS[class_id]]
    cv2.rectangle(or_image, (xmin, ymin), (xmax, ymax), color, 2)
    # Make adjustments to make the label visible for all objects
    y = ymin - 15 if ymin - 15 > 15 else ymin + 15
    label = "{}: {:.0f}%".format(labels[class_id], obj['score'] * 100)
    cv2.putText(or_image, label, (xmin, y),
        cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

original_uint8 = or_image.astype(np.uint8)
```

Sample Outputs

Image - 1

Image - 2

Image -1 is unable to classify the **crosswalk** sign at the back of the stop sign with significant confidence.

Image - 2 has much better classification with all relevant road signs being detected.

Testing Model on Raspberry Pi

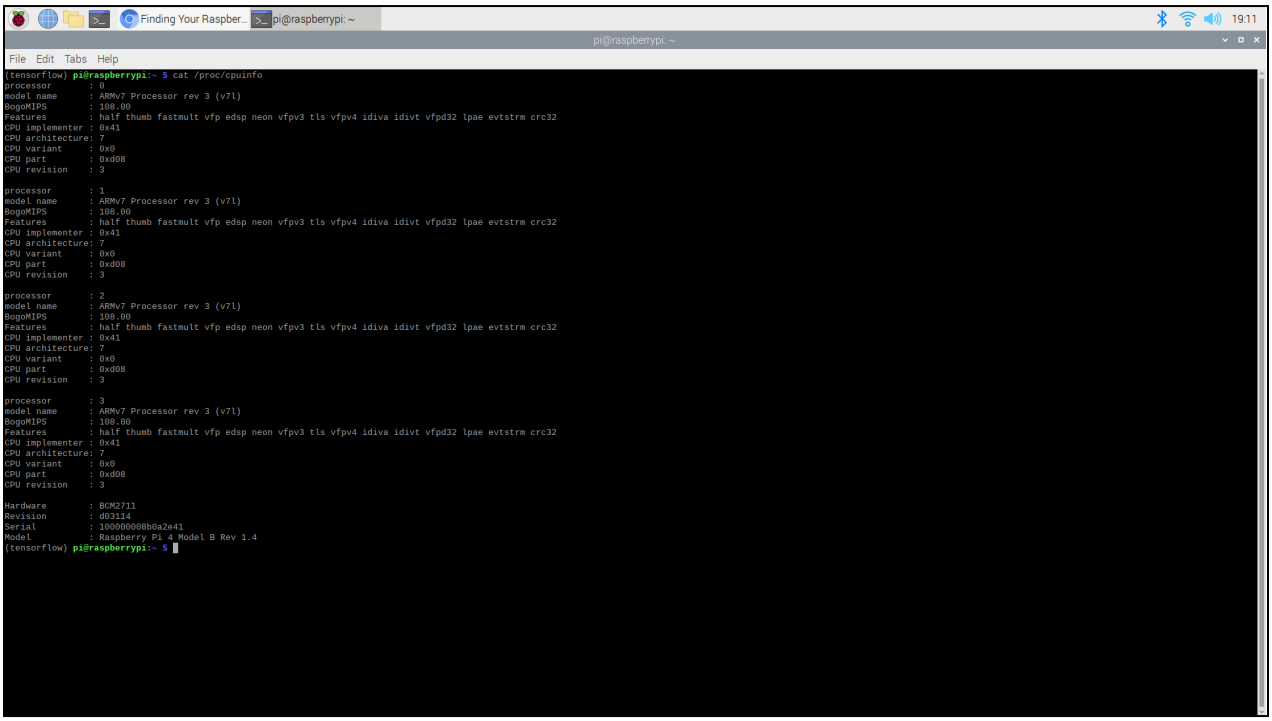
For the model to be tested on raspberry pi - a module is developed to open the camera, and assess the live video feed. Due to limited mobility of the camera - the testing is performed on print out of road signs to see if the object detection task was being successfully performed by the model.

Raspberry Pi Specification

The system specification used for Raspberry Pi (raspberry pi 4 model b rev 1.4) are as follows -

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
- 1GB, 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE

- Gigabit Ethernet
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
- 2 × micro-HDMI ports (up to 4kp60 supported)
- 2-lane MIPI DSI display port
- 2-lane MIPI CSI camera port
- 4-pole stereo audio and composite video port
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
- OpenGL ES 3.1, Vulkan 1.0
- Micro-SD card slot for loading operating system and data storage
- 5V DC via USB-C connector (minimum 3A*)
- 5V DC via GPIO header (minimum 3A*)
- Power over Ethernet (PoE) enabled (requires separate PoE HAT)
- Operating temperature: 0 – 50 degrees C ambient



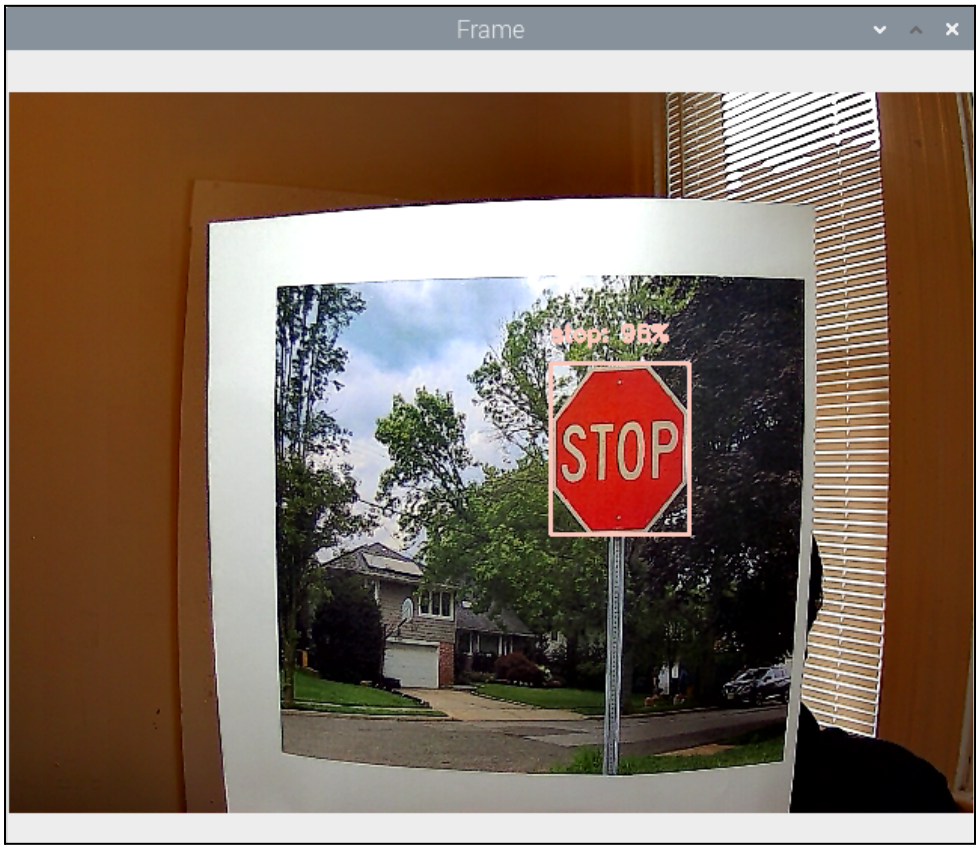
Live Video Feed

To capture the live video feed a few lines of code was needed to be run on Raspberry Pi -

```
#Code to switch on video feed from camera
print('[INFO] Starting video stream ...')
vs = VideoStream(src=0).start()
time.sleep(2.0)
fps = FPS().start()

while True:
    frame = vs.read()
    rframe = cv2.resize(frame, resize_dim)
    rframe = np.array(rframe, dtype=input_data_type)
    #rframe = np.expand_dims(rframe, axis=0)
    #print(rframe.shape)

    count, scores, classes, boxes = detect(interpreter, rframe)

    results = []
    for i in range(count):
        result = None
        if scores[i] > float(args['confidence']):
            result = {
                    'bounding_box': boxes[i],
                    'class_id': classes[i],
                    'score': scores[i]
                }
        results.append(result)
        for obj in results:
            if obj is None:
                continue
            ymin, xmin, ymax, xmax = obj['bounding_box']
            xmin = int(xmin * frame.shape[1])
            xmax = int(xmax * frame.shape[1])
            ymin = int(ymin * frame.shape[0])
            ymax = int(ymax * frame.shape[0])

            class_id = int(obj['class_id'])

            color = [int(c) for c in COLORS[class_id]]
            cv2.rectangle(frame, (xmin, ymin), (xmax, ymax), color, 2)
            y = ymin - 15 if ymin - 15 > 15 else ymin + 15
            label = "{}: {:.0f}%".format(CLASSES[class_id], obj['score']*100)
            cv2.putText(frame, label, (xmin, y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
    cv2.imshow("Frame", frame)
    key = cv2.waitKey(1) & 0xFF
```

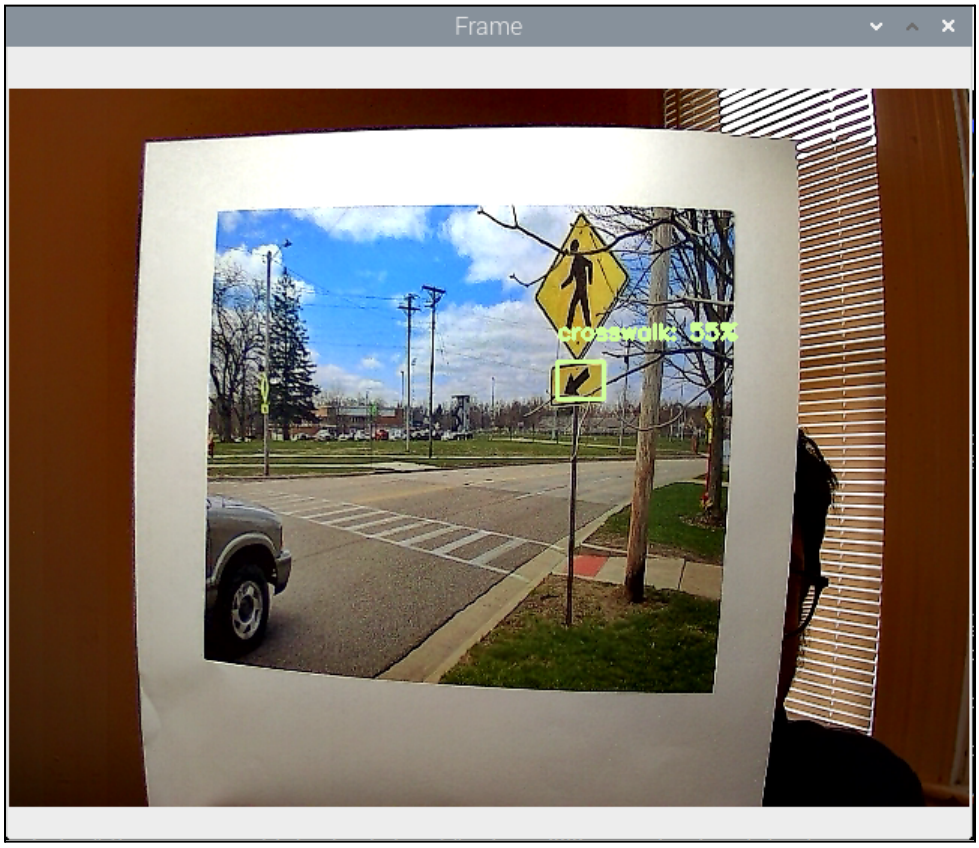
Outputs of Video Feed

From the testing of the video feed some of the samples are presented below -

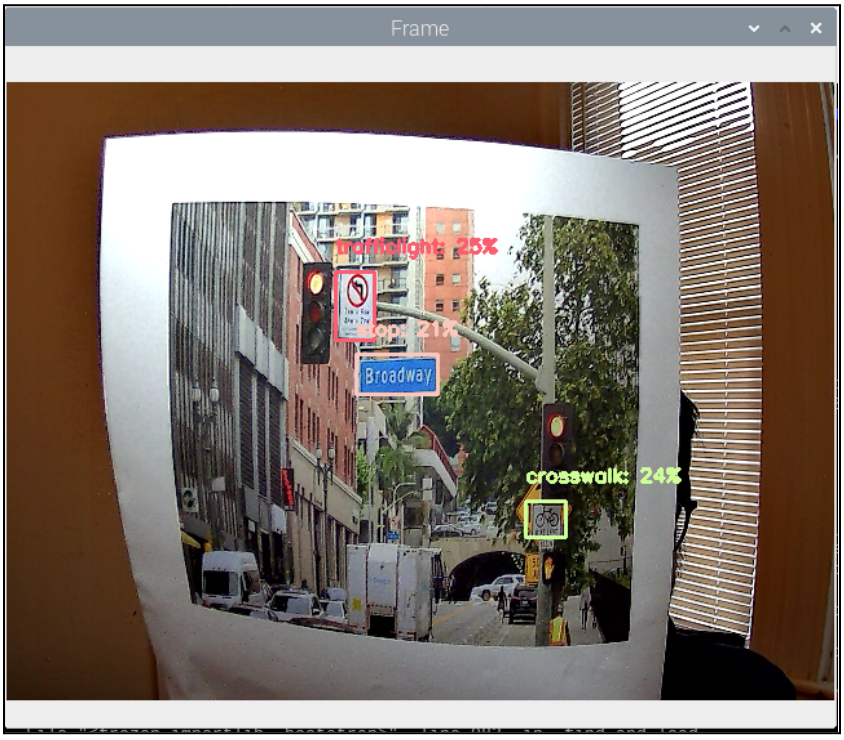


Stop signs are detected with very high confidence.

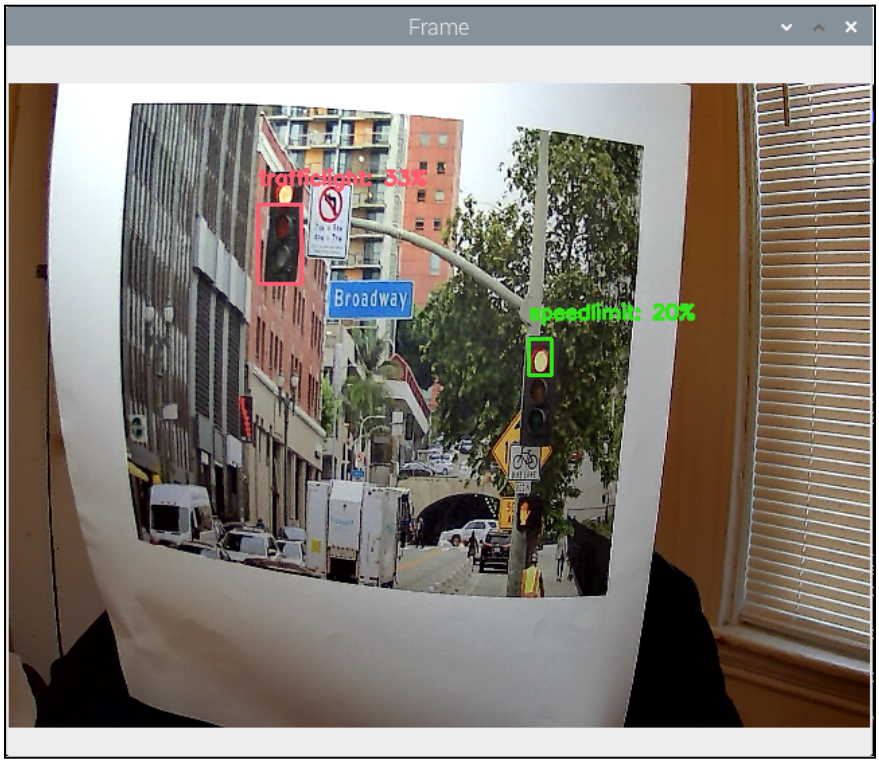
The Speed Limit sign is detected but the confidence is not high.



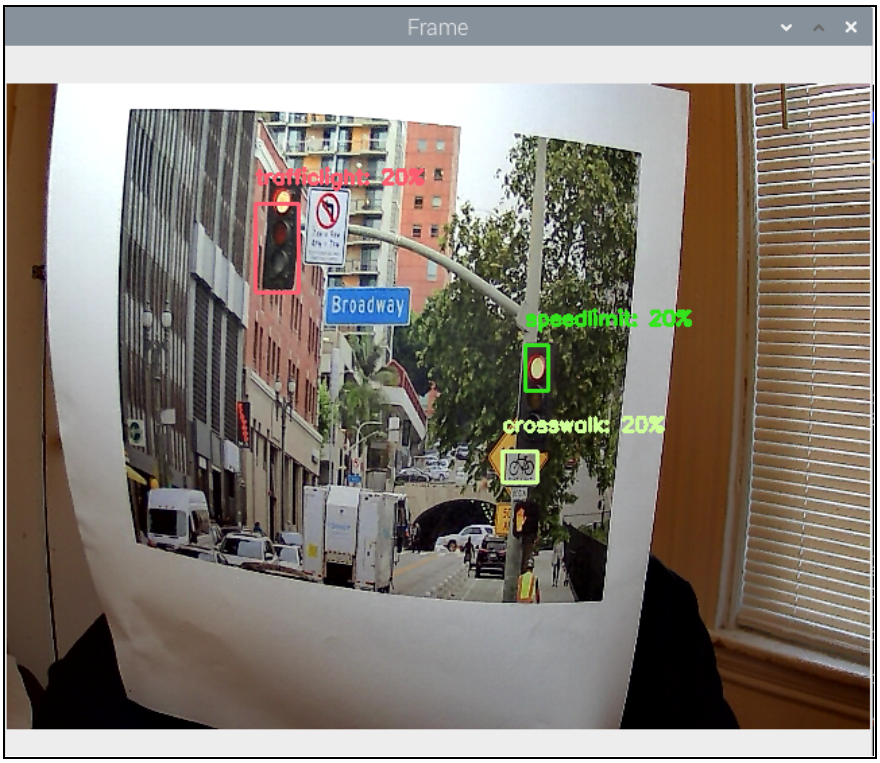
The Crosswalk sign is detected but the confidence level is moderate.



Some misclassifications are there in the frame of this image and the confidence is low.



In the next frame the traffic light is correctly detected.



There is misclassification in the speed limit and it is a wrong detection.

Further Improvements

The further improvements in the project could be using different forms of augmentation techniques. The model can be trained with more examples - using more edited images to cut out additional noise from the images.

The model can also be trained with a larger batch size - 32 and above but it could not be done due to Hardware limitations.

Also, EfficientDet-Lite3 and EfficientDet-Lite4 architecture can be used for training so that we can get more accuracy. However the frame rate would drop but model performance would be higher.

Acknowledgements

For the project, I would like to thank Prof Srinivasan Radhakrishnan for his guidance in every step and support with the hardware and logistics. I would also like to thank my flatmates Devroop Kar (MS CS Northeastern University and Incoming PhD CS at Rochester Institute of Technology) and Baidik Chandra (PhD CS at Northeastern University) for their invaluable inputs

in the project. Without their inputs at specific parts of the project especially data collection and testing the model on Raspberry Pi - this project would not be possible.

References

https://arxiv.org/abs/1911.09070

https://github.com/tensorflow/examples/blob/master/tensorflow_examples/lite/model_maker/third_party/efficientdet/aug/autoaugment.py

https://www.tensorflow.org/lite/api_docs/python/tf/lite

https://paperswithcode.com/task/object-detection

https://towardsdatascience.com/coco-data-format-for-object-detection-a4c5eaf518c5

https://www.tensorflow.org/lite/api_docs/python/tflite_model_maker/model_spec

http://host.robots.ox.ac.uk/pascal/VOC/

https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00444-8
https://yanfengliux.medium.com/the-confusing-metrics-of-ap-and-map-for-object-detection-3113ba0386ef

https://pypi.org/project/opencv-python/

https://labelstud.io/guide/

https://www.youtube.com/watch?v=yqkISICHH-U&t=12s

https://www.youtube.com/watch?v=-ZyFYniGUsw&t=235s

https://github.com/tensorflow/examples/tree/master/tensorflow_examples/lite/model_maker

https://joedockrill.github.io/jmd_imagescraper/

Github Repo

https://github.com/palit-ishan/Traffic-and-Road-Signs-Object-Detection-Project.git