Person Detection using a pretrained model – TF Lite

TEAM M:

Rajaneeshwar AIE19027

Lakshaya K AIE19039

Gokul Prazath AIE19062

Vasudevan KM AIE19067

Introduction to Computer Networks

19AIE211

Acknowledgments

This project would not have been possible without the support and guidance of Mr Premjith B who provided us with sufficient knowledge and guidelines even during these unprecedented and trying times.

We would also like to thank our Computer Science and Engineering (Artificial Intelligence) department for giving us this opportunity to nurture and hone our skills.

Furthermore, we would like to thank the Amrita Vishwa Vidyapeetham management for ample resources to avail our project needs and a platform for communication within our team members and online lectures during this lockdown.

Table of Contents:

Objective

The following report details upon the implementation of person detection using a pretrained model. This pretrained model has been converted to TensorFlow Lite, and various quantization techniques have been applied for optimization. A comparison has also been drawn between the optimized models.

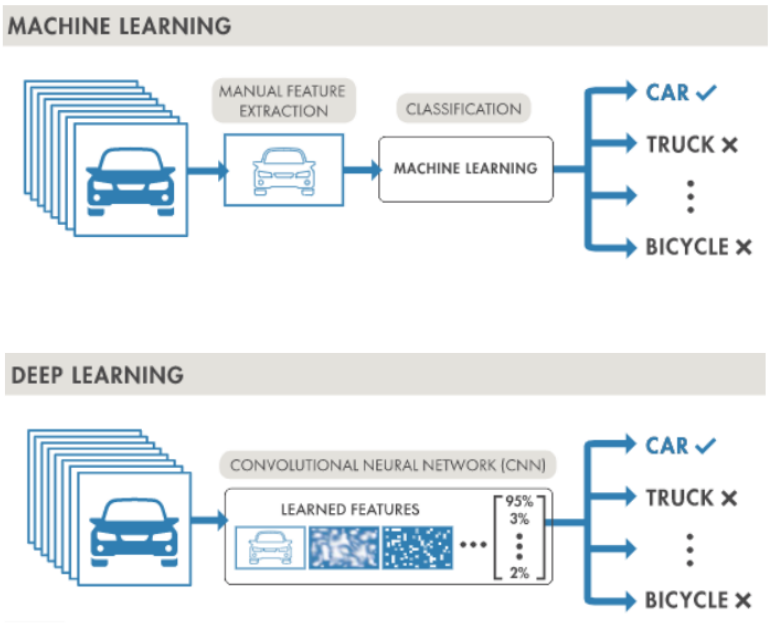
Person Detection:

Person detection falls under a broader category of object detection, which will first be defined.

Object detection involves using computer vision and image processing in order to identify and locate instances of objects within an image or video. Specifically, object detection draws bounding boxes around the detected objects, allowing users to locate the objects in a given frame. Object detection can be divided into the following approaches: machine learning based and deep learning based.

In traditional machine learning-based techniques, certain features of an image are analyzed, such as the color histogram or edges, to identify groups of pixels that may belong to an object. These features are then fed into a regression model that predicts the location of the detected object along with its class label. On the other hand, deep learning-based approaches employ convolutional neural networks (CNNs) to perform end-to-end, unsupervised object detection, in which features don't need to be defined and extracted separately. This project will follow the deep learning route.

While object detection aims to detect every object in the image, person detection specifically only detects humans.



Selecting the pretrained model:

The TensorFlow model zoo is a collection of pretrained object detection models that can be used for inference. All the models in this model zoo contain pretrained parameters for their specific datasets. Some of the models available are as follows:

- Centernet
- EfficientDet
- SSD MobileNet
- ResNet
- R-CNN
- ExtremeNet

> Currently, on-device inference is only optimized with SSD models. Better support for other architectures like CenterNet and EfficientDet is being investigated.

Since only SSD models are currently supported by the TensorFlow lite API, the SSD MobileNet model has been selected for this project.

SSD MobileNet V2:

The SSD MobileNet V2 is an object detection model that has been trained on the COCO 2017 dataset (images scaled to 320x320 resolution). It has been created using the TensorFlow Object Detection API.

The split of the images used is as follows:

Training	118K images / 18GB
Validation	5K images / 1GB
Testing	41K / 6GB

It is essential to note that this model has 90 classes. Since the pretrained models available for TensorFlow only include Object Detection, this project will be manually converting the model to be suitable for person detection in the later steps.

Obtaining the pretrained model:

Step 1: Downloading the model

Code:

```
# Cloning the tensorflow models
!git clone --depth 1 https://github.com/tensorflow/models
```

```
# Install the Object Detection API
%%bash
cd models/research/
protoc object_detection/protos/*.proto --python_out=.
cp object_detection/packages/tf2/setup.py .
python -m pip install .
```

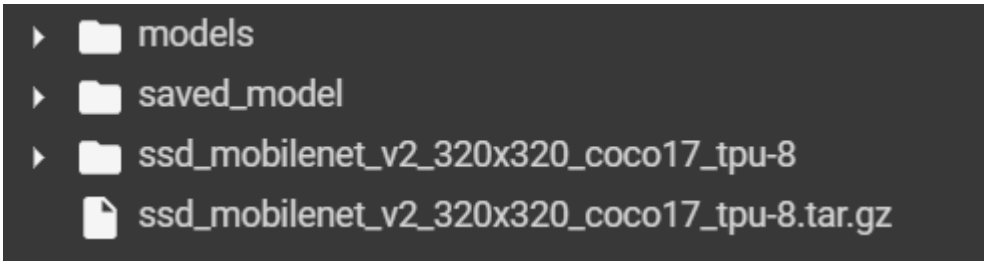
```
# Downloading & extracting the inbuilt SSD-Mobilenet V2 model
!wget http://download.tensorflow.org/models/object_detection/tf2/20200711/ssd_mobilenet_v2_320x320_coco17_tpu-8.tar.gz
!tar -xvf ssd_mobilenet_v2_320x320_coco17_tpu-8.tar.gz
```

Explanation:

The first step of the process is to obtain the pretrained model. In order to this, the *git clone* command has been used to clone the TensorFlow models from GitHub. Following this, the object detection API has been installed. The TensorFlow Object Detection API is an open-source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models.

The *wget* command is then used for downloading the SSD MobileNet V2 model from the Internet. Since the format will be that of a zipped file, it'll be extracted with the help of the *tar-xvf* command.

Output:



Step 2: Exporting the TFLite inference graph

Code:

```
# Generates an intermediate SavedModel that can be used with the TFLite Converter
!python /content/models/research/object_detection/export_tflite_graph_tf2.py \
    --trained_checkpoint_dir ssd_mobilenet_v2_320x320_coco17_tpu-8/checkpoint \
    --pipeline_config_path ssd_mobilenet_v2_320x320_coco17_tpu-8/pipeline.config \
    --output_directory `pwd`
```

Explanation:

In order to convert the pretrained model to TensorFlow lite, an intermediate saved model should be generated. To achieve this, a Python file named *export_tflte_graph_tf2.py* should be accessed. This file was downloaded into the working directory during the installation of the Object Detection API.

The parameters of this file are as follows:

- Checkpoint

  The directory to the checkpoint of the pretrained model is passed to this parameter. It is a binary file which contains all the values of the weights, biases, gradients and all the other variables saved.

- Pipeline

  This parameter takes the path of the pipeline configuration. All the settings and required information for training and evaluating the model is given in the *pipeline.config* file.

- Output Directory

  The current working directory is specified as the output directory.

Output:

The output is a folder named *saved_model*. It contains the following subfolders and files:

- Assets

  The assets folder typically contains files used by TensorFlow, for example text files used to initialize vocabulary tables. It is unused in this case so the folder is empty.

- Variables

  This folder contains 2 files: the *variables.data* file contains the model's weights and the *variables.index* file indicates where the weights are stored.

The *saved_model.pb* (where pb stands for protbuf), contains the graph definition, as well as the weights of the model.

Conversion to TFLite:

Without Optimization:

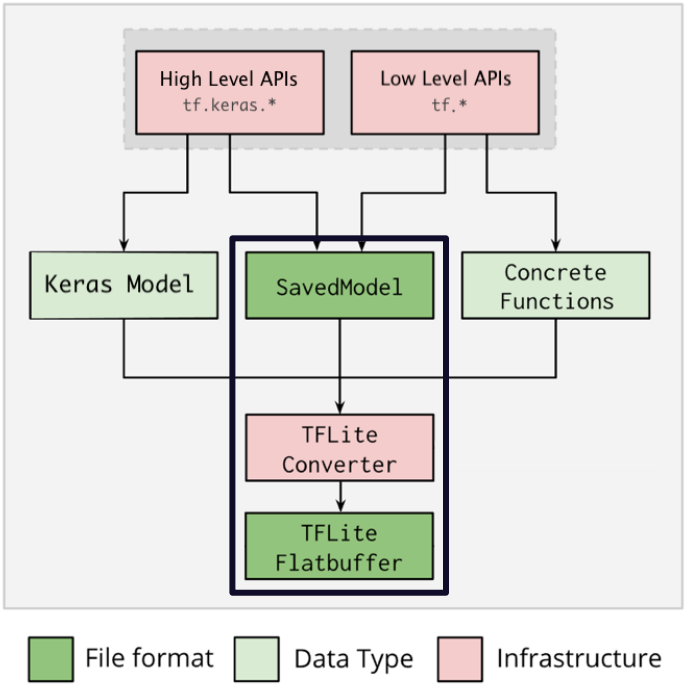
```
dir = '/content/saved_model'
```

```
# Convert the model to TF lite
converter = tf.lite.TFLiteConverter.from_saved_model(dir)
tflite_model = converter.convert()

# Serialize the model
open('tflite_model_nq.tflite', 'wb').write(tflite_model)
file_size = os.path.getsize('tflite_model_nq.tflite')
print("File Size of TFlite non-quantised:", file_size, "bytes")
```

```
File Size of TFlite non-quantised: 24287460 bytes
```

The next step is to convert the saved model into a TensorFlow Lite model. To accomplish this, the TensorFlow Lite converter has been used. The convertor takes a TensorFlow model and generates a TensorFlow Lite model which is an optimized FlatBuffer format that is identified by the .tflite file extension.



Model Optimization:

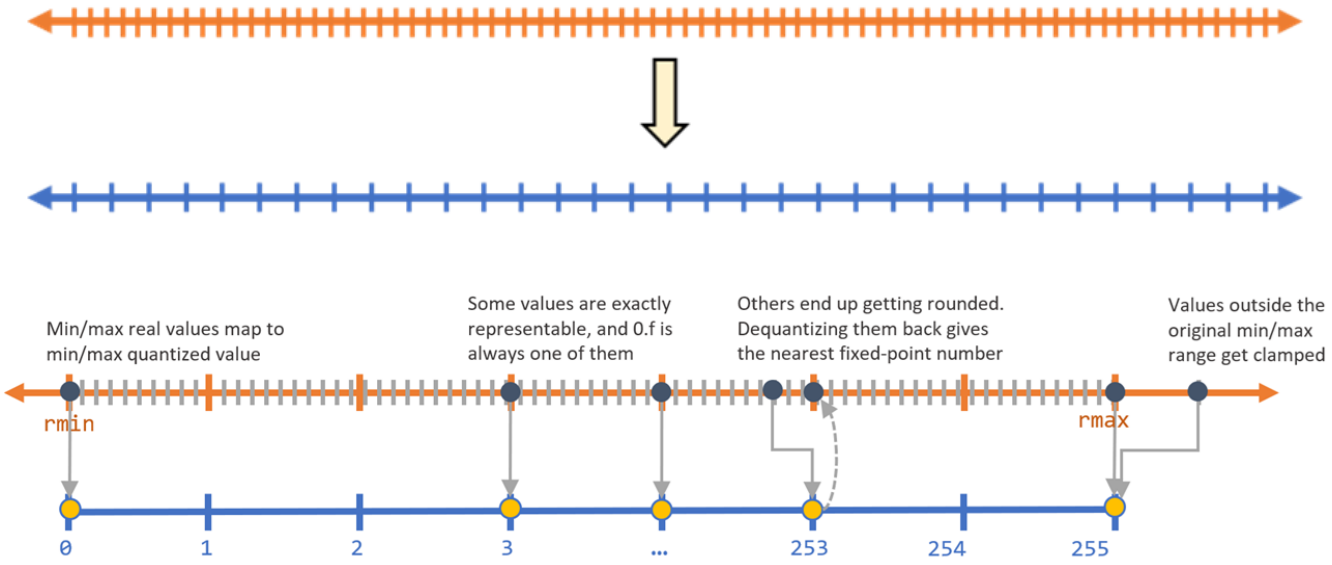
Since TensorFlow Lite models can be deployed on edge devices, it is essential to optimize the models with respect to the hardware requirements of the devices. The efficiency of the model's inference is a serious concern during the deployment of machine learning models due to reasons such as latency, memory utilization, and even power consumption.

The benefits of optimization are as follows:

- Smaller memory footprint
- Makes inference more efficient
- Allows the model work with lower precision
- Uses lesser energy in inference

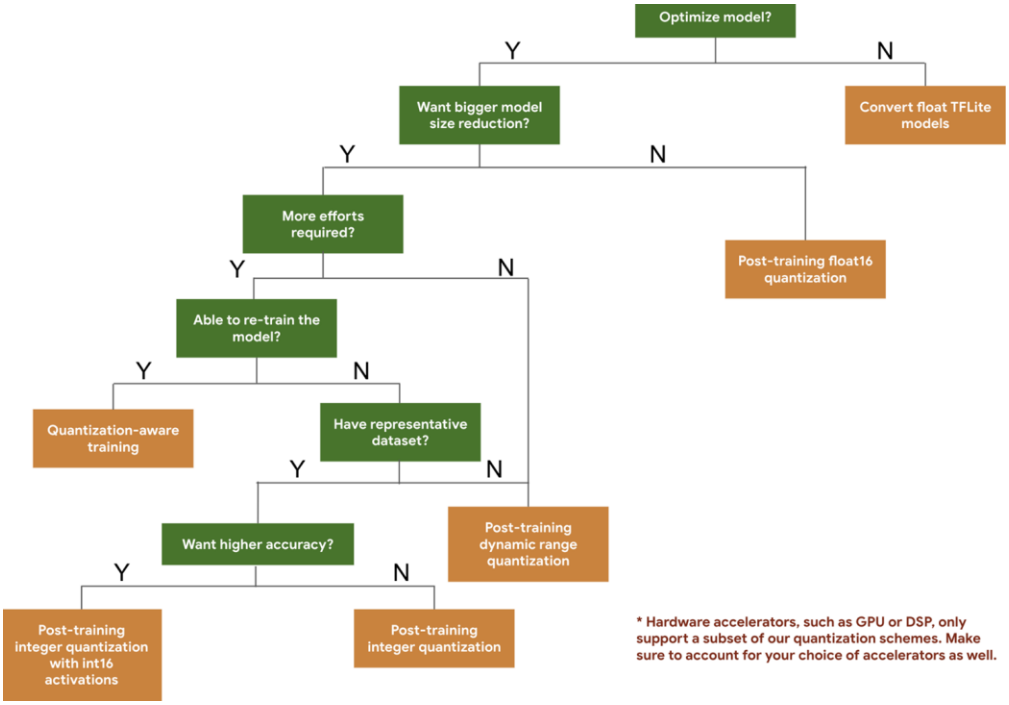
Quantization:

Computers can only utilize a finite number of bits to represent infinite real numbers. The accuracy of representation is decided by how many bits are used - 32-bit floating point being the default for most applications, including deep learning. Deep Neural Networks can work with smaller datatypes, with less precision, such as 8-bit integers. The process of quantization is the discretization of numbers to some specific values, which can then be represented using integers instead of floating-point numbers.



Selecting the Quantization technique:

The following decision tree can be used to decide which quantization technique to employ based upon the requirements of the model at hand.



Since the model used won't be retrained, quantization-aware training won't be performed. The following parts of this project will focus on post-training quantization.

Post-Training Quantization:

Post-training quantization involves techniques that reduce CPU and hardware accelerator latency, processing power, and model size with little degradation in model accuracy. These techniques can be applied on a float TensorFlow model during the conversion to TensorFlow Lite. They are enabled as options in the TensorFlow Lite converter.

The following four quantization techniques will be performed and detailed below:

- Dynamic Range
- Full Integer
- Float Fallback
- Float16

Dynamic Range Quantization:

Dynamic range quantization is a technique wherein the model weights get quantized from the float32 format to the int8 format. Due to this, the model size reduces 4 times. At inference, weights are converted from a precision of 8-bits to floating point and computed using floating-point kernels. This conversion is done once and cached to reduce latency.

To further improve latency, "dynamic-range" operators dynamically quantize activations based on their range to 8-bits and perform computations with 8-bit weights and activations. This optimization provides latencies close to fully fixed-point inference. Nevertheless, the outputs are still stored using floating point. As a result of this, the speedup with dynamic-range quantization is lesser compared to a full fixed-point computation.

While models get smaller with dynamic range quantization, the possibility of running the model for inference on a GPU or TPU is lost. A CPU has to be utilized instead.

Code Snippet:

```
converter = tf.lite.TFLiteConverter.from_saved_model(dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model_dr = converter.convert()
open('tflite_model_dr.tflite', 'wb').write(tflite_model_dr)
```

Full Integer Quantization:

Full integer quantization enables users to use an already-trained floating-point model and fully quantize it to only use 8-bit signed integers (int8). By using this quantization scheme, reasonable quantized model accuracy can be obtained across many models without resorting to retraining a model with quantization-aware training. Models experience a 4x size reduction, and will see even greater CPU speed-ups. Fixed point hardware accelerators, such as Edge TPUs, are also able to run these models.

Generating the Representative Dataset:

For full integer quantization, there is a need to calibrate or estimate the range of all floating-point tensors in the model. Unlike constant tensors such as weights and biases, variable tensors such as model input, activations (outputs of intermediate layers) and model output cannot be calibrated unless a few inference cycles are run. As a result, the converter requires a representative dataset for calibration. This dataset can be a small subset of the training or validation data.

```
def rep_data_gen():
  a = []
  for img in glob.glob("/content/sample/*.jpg"):
    data = cv2.imread(img)
    img = cv2.resize(data, (300, 300))
    img = img / 255.0
    img = img.astype(np.float32)
    a.append(img)
    a1 = np.array(a)
    img = tf.data.Dataset.from_tensor_slices(a1).batch(1)
    for i in img:
        yield [i]
```

Code Snippet:

```
# In onder to do full integer quantization, a range should be estimated of all the floating point tensors in the model
# For variable tensors, the convertor needs a representative dataset with a few inference cycles to calibrate
# Loop over every image and perform detection
converter_int = tf.lite.TFLiteConverter.from_saved_model(dir)
converter_int.representative_dataset = rep_data_gen
converter_int.optimizations = [tf.lite.Optimize.DEFAULT] # Using the default optimisation strategy
converter_int.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8] # Ensures that TFlite interpreter only uses integers
converter_int.inference_input_type = tf.uint8  # Setting input type to be int8
converter_int.inference_output_type = tf.uint8  # Setting output type to be int8
tflite_quant_model_int = converter_int.convert() # Converts a saved model into TensorFlow Lite model
open('tflite_quant_model_int.tflite', 'wb').write(tflite_quant_model_int)
```

Float-Fallback Quantization:

Float fallback quantization is used in order to fully integer quantize a model, but resorts to float operators when an integer implantation isn't available. This method ensures that conversion occurs smoothly. However, a drawback is that float-fallback quantized models won't be compatible with integer-only devices. This is due to the fact that the input and output of the model still remains float in order to have the same interface as the original float only model.

Code Snippet:

```
# Path to the directory of the saved model
converter_ff = tf.lite.TFLiteConverter.from_saved_model(dir)
# Assigning the representative dataset generator
converter_ff.representative_dataset = rep_data_gen
# Using the default optimisation strategy
converter_ff.optimizations = [tf.lite.Optimize.DEFAULT]

# Converts a saved model into TensorFlow Lite model
tflite_quant_model_ff = converter_ff.convert()
open('tflite_quant_model_ff.tflite', 'wb').write(tflite_quant_model_ff)
```

Float16 Quantization:

Float16 quantization reduces TensorFlow Lite model sizes by 2x, while sacrificing very little accuracy. It quantizes model constants (like weights and bias values) from full precision floating point (32-bit) to a reduced precision floating point data type (IEEE FP16).

However, it doesn't reduce latency as much as a quantization to fixed point math. Furthermore, a float16 quantized model that is deployed on a CPU dequantizes the weights values to float32 by default.

```
converter = tf.lite.TFLiteConverter.from_saved_model(dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_types = [tf.float16]
tflite_quant_model_f16 = converter.convert()
open('tflite_quant_model_f16.tflite', 'wb').write(tflite_quant_model_f16)
```

Comparison:

Model	Dynamic Range	Full Integer	Float16
Size	4x reduction	4x reduction	2x reduction
Speed-up	2x-3x speedup	3x+ speedup	GPU acceleration
Accuracy	Accuracy Loss	Smaller Accuracy Loss	Insignificant Accuracy Loss
Hardware	CPU	CPU, Edge TPU, Microcontrollers	CPU, GPU
Data Requirements	No data	Representative Sample	No data

Visualization:

Code:

```
# Importing necessary packages
import tensorflow as tf
import pandas as pd
import os
import importlib.util
import numpy as np
import tensorflow as tf
import cv2
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
from scipy import ndimage
```

```
# Helper function
def person_detection(model, images):
  interpreter = tf.lite.Interpreter(model) # Passing the model
  interpreter.allocate_tensors() #Allocates memory for the model's input Tensors

  # Get model details
  input_details = interpreter.get_input_details()
  output_details = interpreter.get_output_details()

  # Getting height & width  -- 300x300 since our model was trained on 300x300 images
  height = input_details[0]['shape'][1]
  width = input_details[0]['shape'][2]

  # Boolean operation to check if model is float or int
  floating_model = (input_details[0]['dtype'] == np.float32)
  uint8_model = (input_details[0]['dtype'] == np.uint8)

  # Uploading the labels & storing in a list
  with open("labels.txt", 'r') as f:
    labels = [line.strip() for line in f.readlines()]

  # Parameter values of the input image in MobileNet
  input_mean = 127.5
  input_std = 127.5

  # Specifying a minimum confidence threshold
  min_conf_threshold = 0.55
```

```
ourclasses = []
ourscores = []
count_hum = []
# Loop over every image and perform detection
for image_path in images:
    # Load image and resize to expected shape
    image = cv2.imread(image_path)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    imH, imW, _ = image.shape
    image_resized = cv2.resize(image_rgb, (width, height))
    input_data = np.expand_dims(image_resized, axis=0)

    # Normalize pixel values if using a floating model (i.e. if model is non-quantized)
    if floating_model:
      input_data = (np.float32(input_data) - input_mean) / input_std

    # Perform the actual detection by running the model with the image as input
    interpreter.set_tensor(input_details[0]['index'], input_data)
    interpreter.invoke()

    # Retrieve detection results
    boxes = interpreter.get_tensor(output_details[0]['index'])[0] # Bounding box coordinates of detected objects
    classes = interpreter.get_tensor(output_details[1]['index'])[0] # Class index of detected objects
    scores = interpreter.get_tensor(output_details[2]['index'])[0] # Confidence of detected objects
```

```
 # IF UINT8 MODEL: Obtaining true value from int_8 value
 # real_value = (int8_value - zero_point)*scale
 if uint8_model:
   scale = interpreter.get_input_details()[0]["quantization"][0] # Getting the scale from the model's input details
   zero_point = interpreter.get_input_details()[0]["quantization"][1] # Getting the zero point from the model's input details
   # Scaling the values correspondingly
   boxes = boxes*scale
   scores = scores*scale
   classes = classes*scale

 # Loop over all detections and draw detection box if confidence is above minimum threshold
 for i in range(len(scores)):
   # If anything other than a human is detected --> score is set to 0
     if classes[i] != 0:
       scores[i] = 0
     if (scores[i] > min_conf_threshold): # Only considering detections above threshold confidence

         # Get bounding box coordinates and draw box
         # Interpreter can return coordinates that are outside of image dimensions, need to force them to be within image using max() and min()
         ymin = int(max(1,(boxes[i][0] * imH)))
         xmin = int(max(1,(boxes[i][1] * imW)))
         ymax = int(min(imH,(boxes[i][2] * imH)))
         xmax = int(min(imW,(boxes[i][3] * imW)))

         cv2.rectangle(image, (xmin,ymin), (xmax,ymax), (10, 255, 0), 2) # Drawing the rectangular box
```

```
          # Draw label
          object_name = labels[int(classes[i])] # Look up object name from "labels" array using class index
          label = '%s: %d%%' % (object_name, (scores[i])*100) # Example: 'person: 72%'
          labelSize, baseLine = cv2.getTextSize(label, cv2.FONT_HERSHEY_SIMPLEX, 0.7, 2) # Get font size
          label_ymin = max(ymin, labelSize[1] + 10) # Make sure not to draw label too close to top of window
          # Draw white box to put label text in
          cv2.rectangle(image, (xmin, label_ymin-labelSize[1]-10), (xmin+labelSize[0], label_ymin+baseLine-10), (255, 255, 255), cv2.FILLED)
          cv2.putText(image, label, (xmin, label_ymin-7), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2) # Draw label text

 cv2_imshow(image) # Displaying the image
```

```python
# Displaying output in terms of a DF
def visualise_output(ourscores, ourclasses, count_hum):
  fin_scores=[]
  fin_class=[]
  # Converting all values to string
  for i in range(len(ourclasses)):
    temp_scoreStr = ' '.join([str(elem) for elem in ourscores[i]])
    temp_classStr = ' '.join([str(elem) for elem in ourclasses[i]])
    fin_scores.append(temp_scoreStr)
    fin_class.append(temp_classStr)
  return pd.DataFrame(list(zip(fin_class,fin_scores,count_hum )),columns =['classes','scores','count'])
```

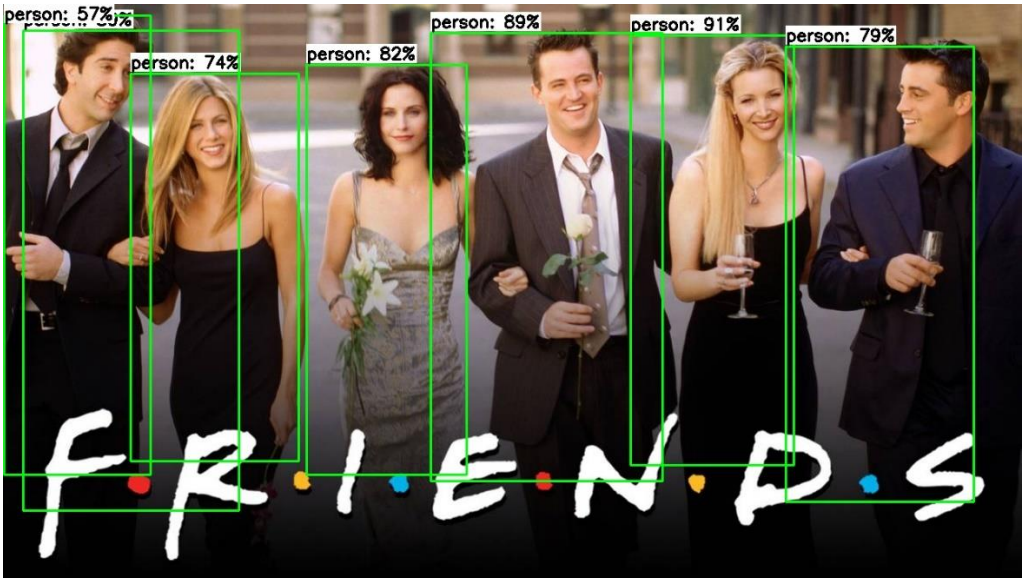
```python
# Main function
# Input parameters: models, images
# Outputs: Resultant DF for image detection for all the models
def person_detection_outputs(models, images):
  output_dfs = []
  for i in models:
    scores, classes, humans = person_detection(i, images)
    df = visualise_output(scores, classes, humans)
    output_dfs.append(df)

  result = pd.concat(output_dfs,  keys=['TFLite - No Quantisation', 'TFLite - Dynamic Range', 'TFLite - Float Fallback', "TFLite - Integer Only"], axis='columns')
  result.index = images
  return result
```

Outputs:

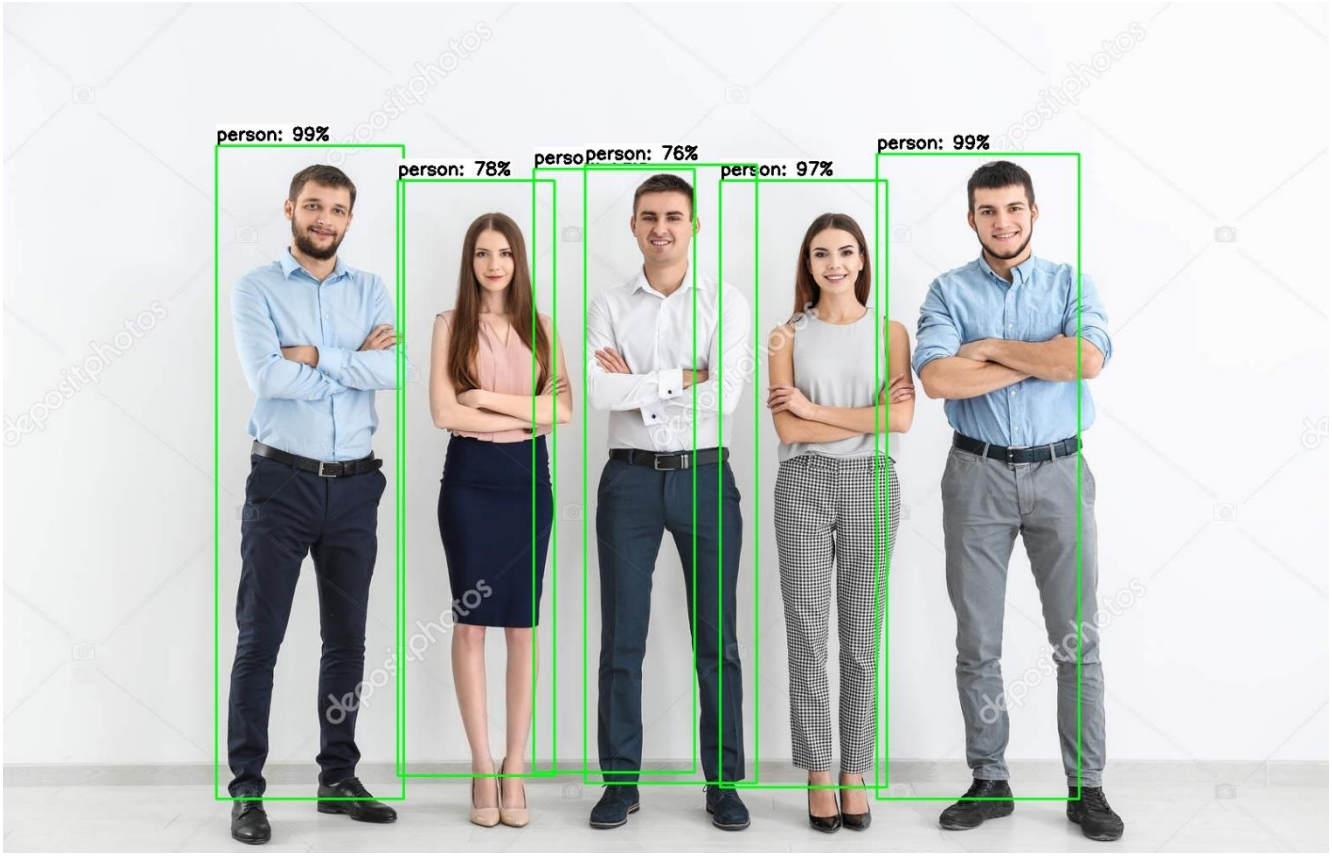
These outputs correspond to the Integer-Quantized TFLite model.

Picture 1:

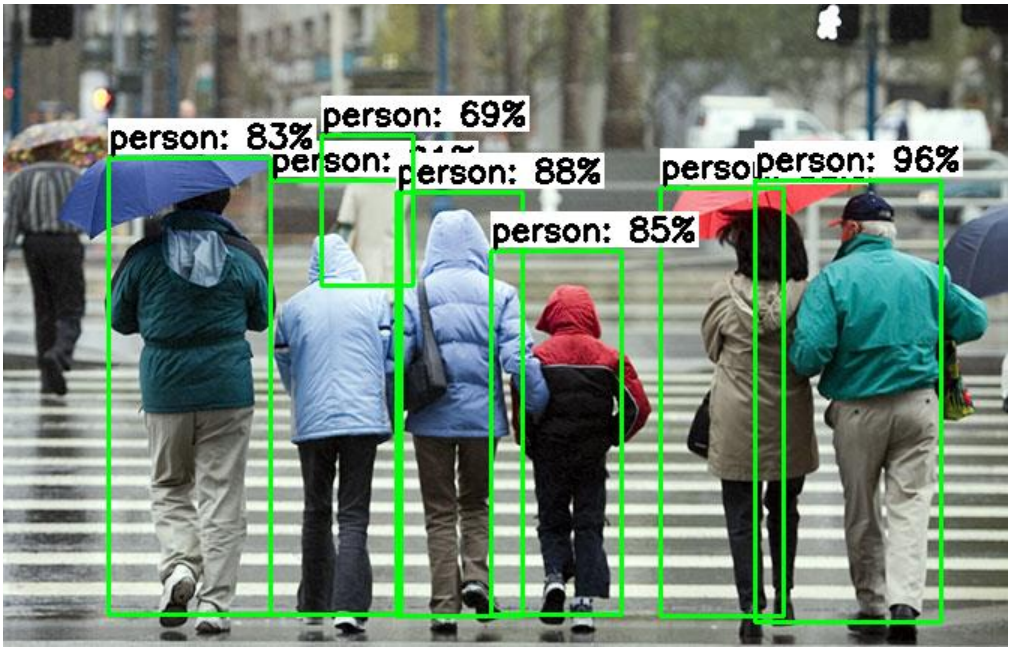


Picture 2:

Picture 3:



Picture 4:



Analysis of Output:

	classes				scores				count			
	TFLite - No Quantisation	TFLite - Dynamic Range	TFLite - Float Fallback	TFLite - Integer Only	TFLite - No Quantisation	TFLite - Dynamic Range	TFLite - Float Fallback	TFLite - Integer Only	TFLite - No Quantisation	TFLite - Dynamic Range	TFLite - Float Fallback	TFLite - Integer Only
friends.jpg	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0.7705091 0.7236502 0.67431027 0.65949464 0.56751066 0.5469954 0.5201002	0.779381 0.72387934 0.6738368 0.6590019 0.57136786 0.53069705 0.52315074	0.77973557 0.7249824 0.6747454 0.6592864 0.57025576 0.53223056 0.5238272	0.77973557 0.7249824 0.6747454 0.6592864 0.57025576 0.53223056 0.5238272	7	7	7	7
mh.jpg	0	0	0	0	0.7949883	0.7910958	0.7912071	0.7912071	1	1	1	1
catdog.jpg	1	1	1	1	1	1	1	1	0	0	0	0
people.jpg	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0.85559523 0.84071386 0.769765 0.6487295 0.64104736	0.8569325 0.8402587 0.7811003 0.65594566 0.63789284	0.856844 0.84028184 0.7808938 0.6561438 0.63772786	0.856844 0.84028184 0.7808938 0.6561438 0.63772786	5	5	5	5
crosswalk.jpg	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0.8193802 0.79376197 0.746917 0.7177604 0.7061164 0.701836 0.6579395	0.81821287 0.7916248 0.7545775 0.7215593 0.7095437 0.70602185 0.6516773	0.8184087 0.7919562 0.75471103 0.7216512 0.7101569 0.7058741 0.6513487	0.8184087 0.7919562 0.75471103 0.7216512 0.7101569 0.7058741 0.6513487	7	7	7	7

The output of the program gives the following information:

Note: Columns that are highlighted represent a difference in values.

- Classes

In the case that a human is detected, a zero (0) is appended to the classes array of the corresponding image. If there are no humans in the input picture, a one (1) is appended instead. As visible from the data-frame above, the *catdog.jpg* has a class label of 1 since there are no humans present in the image.

- Scores

The confidence of the model in detecting each person is displayed in the scores array. In the case that the image contains no humans, a complete score of 1 is appended to the array. Since the columns of the scores are highlighted, it is conclusive that there is a difference in confidences when using various techniques to quantize the model.

- Count

This is used to display the number of humans detected in each picture.

Analysis of Quantized Models:

Code:

```
# Finding the sizes of the models
nq_size = os.path.getsize('/content/tflite_model_nq.tflite')
dr_size = os.path.getsize('/content/tflite_model_dr.tflite')
int_size = os.path.getsize('/content/tflite_quant_model_int.tflite')
ff_size = os.path.getsize('/content/tflite_quant_model_ff.tflite')
f16_size = os.path.getsize('/content/tflite_quant_model_f16.tflite')
```

```
# Visualising the sizes in the form of a bargraph
import numpy as np
import matplotlib.pyplot as plt

data = {'Non-Quantised':nq_size, 'Dynamic-Range':dr_size, 'Integer Only':int_size,
        'Float-Fallback':ff_size, 'Float16': f16_size}

models = list(data.keys())
sizes = list(data.values())
fig = plt.figure(figsize = (10, 5))

# creating the bar plot
plt.bar(models, sizes, color ='salmon', width = 0.4)
plt.xlabel("Models")
plt.ylabel("Sizes (in bytes)")
plt.title("Size Comparision of Models")
plt.show()
```

```
# Finding the percentage model size reduction
percentages = list(map(lambda x: x/sizes[0], sizes))
```

```
# Displaying in the form of a DF
comparision = pd.DataFrame(list(zip(models, sizes, percentages)),
                           columns =['Models','Sizes','Percentage of Unquantized Model'])
comparision
```

Outputs:

Figure 1:

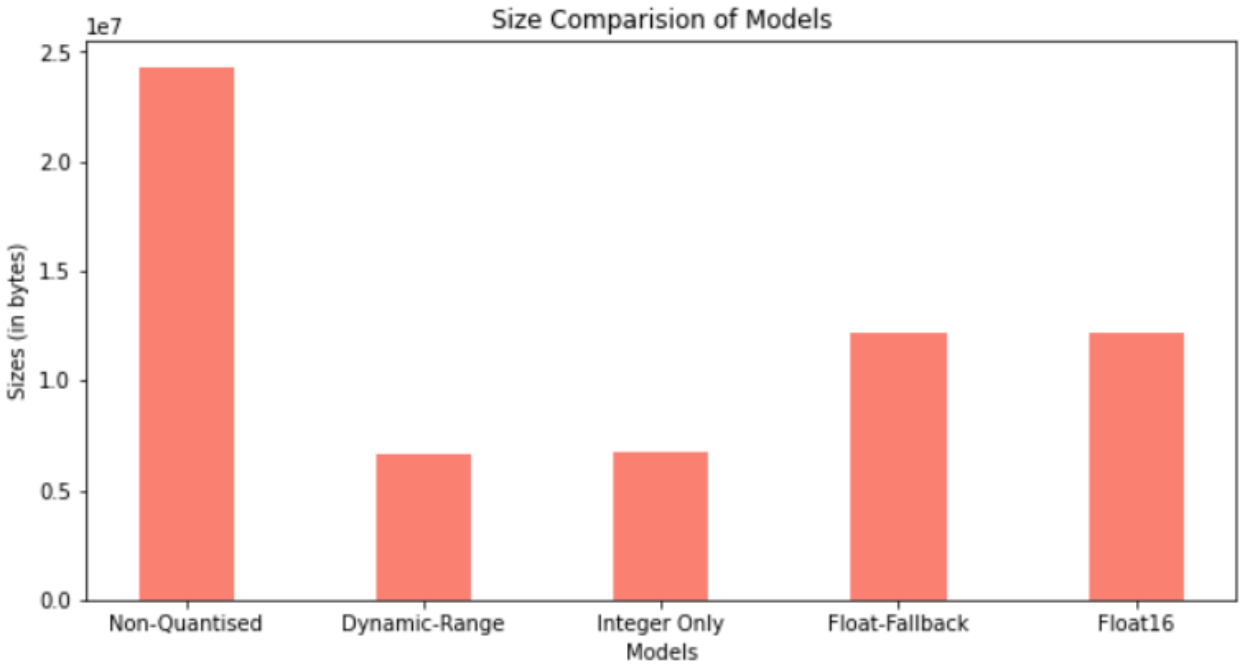
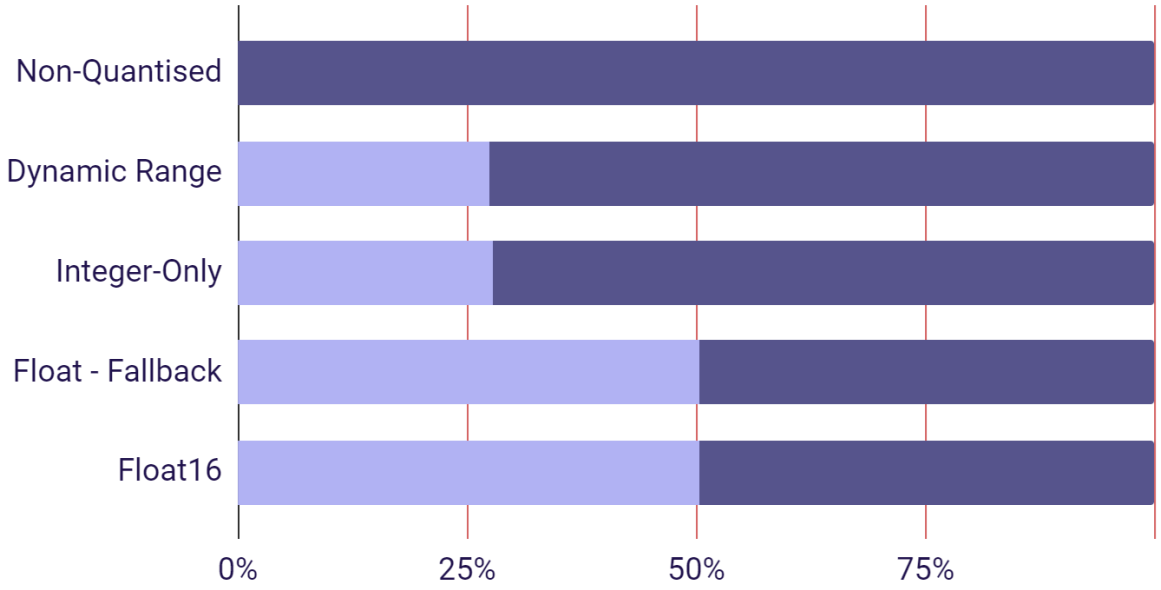


Figure 2:

	Models	Sizes	Percentage of Unquantized Model
0	Non-Quantised	24287460	1.000000
1	Dynamic-Range	6657712	0.274121
2	Integer Only	6744728	0.277704
3	Float-Fallback	12215056	0.502937
4	Float16	12215056	0.502937

Conclusion:

As seen in figures 1 and 2, there is an obvious reduction in size from the non-quantized model to the quantized models. To further illustrate this difference, a plot has been created to compare the differences.



In order to verify that the sizes of the obtained models are accurate, it has been cross-verified with the official size reduction guide.

Model	Dynamic Range	Full Integer	Float
Size	4x reduction	4x reduction	2x reduction

Since the percentages of the bar graph and the table correspond correctly, it can be concluded that the models have been quantized accurately.