ATTENDANCE MARKING USING FACE RECOGNITION

Submitted by,

Riya George (223044)

Shifa Nasar V (223048)

Sreya K Raj (223050)

School of Digital Sciences
Kerala University of Digital Sciences Innovation and Technology

(Digital University Kerala)

Technocity Phase 4 Campus, Thiruvananthapuram, Kerala - 695317

BONAFIDE CERTIFICATE

This is to certify that the project report entitled **ATTEDANCE MARKING USING FACE RECOGNITION** submitted by:

Name	**Reg No:**
RIYA GEORGE	223044
SHIFA NASAR V	223048
SREYA K RAJ	223050

In partial fulfillment of the requirements for the award of Master of Science in Computer Science with Specialization in Data Analytics is a Bonafide record of the work carried out at KERALA UNIVERSITY OF DIGITAL SCIENCES, INNOVATION AND TECHNOLOGY under our supervision.

Supervisor

Prof. ASWIN V S
School of Digital Sciences
DUK

Course Coordinator

Prof. ANOOP V S
School of Digital Sciences
DUK

Head of Institution

Prof. SAJI GOPINATH
Vice Chancellor
DUK

DECLARATION

We**, Riya George, Shifa Nasar V** and **Sreya K Raj** students of Master of Science in Computer Science with Specialization in Data Analytics, hereby declare that this report is substantially the result of our own work, and has been carried out during the period September 2023-January 2024.

ACKNOWLEDGEMENT

This project was supervised by Prof. Aswin V S, an associate professor at Digital University Kerala, who provided invaluable guidance and assistance in addressing our queries and locating relevant resources. We extend our heartfelt appreciation to him.

We also wish to convey our deep gratitude to Dr Saji Gopinath, our Vice Chancellor, for creating a conducive and supportive environment that allowed us to work on the project effectively and in a friendly atmosphere.

CONTENTS

ABSTRACT

This project introduces an innovative attendance marking system that leverages the power of TensorFlow Lite for real-time object detection, particularly focused on recognizing and tracking student attendance in educational settings. The system employs a custom-trained model, developed using Python and various supporting libraries, to accurately detect individuals within a camera's field of view.

A key feature of this system is its ability to integrate seamlessly with an automated attendance tracking mechanism. Utilizing facial recognition technology, the system identifies students and updates their attendance status in a structured Excel file. This process includes marking identified students as "Present" for the respective day and dynamically managing the attendance records by adding new date columns when necessary.

The system operates continuously, capturing live frames from a webcam, detecting faces, updating attendance in real-time, and executing other essential functions concurrently. Its design caters to the real-time requirements of attendance tracking in educational institutions and can be adapted for similar use in various other contexts.

In addition to its primary functionality, the system's performance is graphically demonstrated, showing the precision of object detection and the accuracy of facial recognition in real-time. This project not only offers a new approach to automating attendance but also sets a precedent for the application of advanced object detection technologies in practical, everyday scenarios, enhancing efficiency and accuracy in attendance management across diverse domains.

INTRODUCTION

BACKGROUND

Attendance tracking in educational institutes and other organizational settings has traditionally been a labour-intensive and error-prone process. Conventional methods involving manual attendance taking are not only time-consuming but also prone to inaccuracies and inefficiencies. With the increasing demand for streamlined and automated systems, there's a growing need for innovative solutions that can reliably manage attendance records in real-time.

PROBLEM STATEMENT

The existing manual methods of attendance tracking pose significant challenges in terms of accuracy, time consumption, and scalability. These methods are not equipped to handle the complexities of large-scale attendance management and often lead to inconsistency in record-keeping. In educational institutes and other organizational setups, there is a pressing need for an automated and accurate attendance system that can seamlessly integrate with existing workflows.

Moreover, the arrival of advanced technologies such as machine learning and computer vision offers promising avenues for revolutionizing attendance management. However, the development of a reliable, real-time system capable of accurately recognizing and marking attendance for individuals in dynamic environments remains a challenge.

This project addresses these challenges by introducing an automated attendance marking system that harnesses the power of TensorFlow Lite for real-time object detection and facial recognition. By offering a solution that combines advanced technology with practical usability, this project aims to overcome the limitations of traditional attendance tracking methods and provide an efficient and accurate alternative for educational and other institutional settings.

LITERATURE REVIEW

Automated attendance systems, powered by cutting-edge facial recognition technology, have revolutionized the management of attendance in educational institutions by eliminating the inefficiencies of traditional manual methods. This improvement has been made possible through the integration of advanced object detection algorithms like YOLO V3, backpropagation neural network, and region-based convolution network (RCNN), along with the use of facial recognition techniques. Our proposed system takes this innovation a step further by utilizing YOLO V3 for face detection and Microsoft Azure's face API for recognition. Moreover, our system ensures comprehensive attendance records by capturing images at both the beginning and end of each class. Embracing this approach results in a more seamless and accurate attendance tracking experience.

One trend that has gained attention in the realm of attendance management is the incorporation of modern technologies. In particular, there has been a notable integration with smartphones and cloud services. Our proposed system follows suit with this advancement by utilizing faculty members' smartphones and automating communication through monthly emails to students, parents, and faculty. Not only does this increase convenience, but it also simplifies the attendance management process in educational institutions.

Facial recognition, a form of biometric authentication, has become a crucial component of attendance systems. With the integration of Microsoft Azure's face API, our system offers enhanced security and durability for precise attendance records. In addition to this, our system effectively tackles common challenges like proxy attendance and incorrect records through advanced technologies like YOLO V3 for student counting and face recognition to differentiate between familiar and unfamiliar faces. These features result in a comprehensive and dependable solution to the complexities of attendance management.

The literature review highlights the importance of incorporating sophisticated algorithms, such as Personal Component Analysis (PCA) and Eigenface values, for effective face recognition. In line with this trend, our proposed system integrates these techniques to improve the reliability of attendance management. We have also incorporated the Haar Cascade Algorithm to further enhance accuracy, resulting in a user-friendly interface that simplifies image capture, dataset creation, and training. This makes attendance management easily accessible and efficient for educational institutions.

DATA AND EVALUATION

DATASET

The dataset used for this project contains a collection of facial images of 10 students, specifically sourced from our classmates. It encompasses a total of 76 photos of these 10 students. These images were captured using mobile phone cameras and laptop cameras, offering a diverse set of visuals with various angles, lighting conditions and facial expressions. This ensures the robustness and adaptability of the face recognition model.

DATA PRE-PROCESSING

In our exploratory data analysis (EDA) process, our goal was to make the data suitable for model training. The data consists of students' facial images. The data is labelled using Label Studio. It is a tool used for precise annotation and labelling of data. The face of each person is accurately labelled by drawing bounding boxes around their faces, with the person's name as the label.

The labelled facial images, obtained in yolo (You only look once) format from Label Studio were downloaded and processed through Roboflow to refine and augment the dataset for enhanced model training. Roboflow provided a platform to organize, augment, and preprocess the labelled data efficiently.

During the transformation process on Roboflow, various augmentation techniques were employed, including rotation, flipping, and scaling, to diversify the dataset further. Additionally, Roboflow's capabilities allowed for the resizing and normalization of images, ensuring uniformity and consistency in the dataset. The generated dataset from Roboflow, encompassed an expanded collection of facial images, enriched with variations in orientation, scale, and lighting conditions. This augmented dataset aimed to reinforce the model's adaptability to diverse real-world scenarios, improving its accuracy in recognizing facial features across varying conditions.



The processed data from Roboflow, after augmentation and preprocessing, was downloaded in TFRecords format. This format is specifically tailored for TensorFlow, providing an efficient and optimized way to store and access the augmented dataset. TFRecords format ensures streamlined data input into the TensorFlow model, optimizing its performance during training and inference stages.

METHODOLOGIES

The session of this project is to train and deploy a TensorFlow Lite object detection model capable of detecting objects in images. Leveraging the TensorFlow 2 Object Detection API, and culminating in creating a lightweight TensorFlow Lite model for deployment on resource-constrained devices.

The "TF2 object detection zoo" is a valuable resource for the TensorFlow 2 (TF2) community, comprising of a diverse set of pre-trained object detection models and related materials in the Object Detection API. This collection boasts state-of-the-art models trained on large datasets that can be fine-tuned or applied directly for object detection. In this code, the reference to a "zoo" serves as a metaphor for selecting distinct models that are accessible for use. The provided snippet introduces a dictionary (MODELS_CONFIG) that houses configurable settings for various specific models, including SSD MobileNet V2, EfficientDet D0, and others. These widely-utilized models are deemed part of the TF2 Object Detection Zoo due to their public availability and comprehensive usage for object detection tasks.

Using the provided variable chosen_model, you can effortlessly select a specific model from this diverse "zoo" of models, complete with their own unique configurations and pre-trained weights. This dynamic capability empowers users to handpick the most suitable model for their individual needs or objectives, without the burden of manually adjusting every minuscule detail.

The term "ssd-mobilenet-v2" refers to a specific model architecture for object detection, which combines two powerful technologies: the Single Shot Multibox Detector (SSD) and MobileNetV2. SSD is a cutting-edge algorithm that can quickly detect and classify objects in a single pass. By dividing the image into cells and predicting bounding boxes and class scores for each cell, it allows for efficient and real-time object detection. Meanwhile, MobileNetV2 is a lightweight convolutional neural network that is specifically designed for mobile and edge devices. Despite its speed and efficiency, it maintains a high level of accuracy. As a result, it is commonly used as a feature extractor for various computer vision tasks, including object detection.

When the terms 'ssd-mobilenet-v2' are used together, they are referring to a powerful object detection model that makes use of the SSD algorithm for accurate bounding box predictions

and MobileNetV2 as the backbone for extracting features. This tandem is especially well-suited for real-time situations and environments with restricted processing abilities, such as those found in mobile and embedded devices.

efficientdet-d0" is a term used to describe a specific model architecture known as EfficientDet. With its focus on balancing accuracy and efficiency, EfficientDet is a prominent member of the EfficientDet family, including variants like "d0," "d1," and more. By leveraging different scale levels, EfficientDet has proven its capability to deliver high levels of accuracy with computational efficiency. Thus, the "d0" denotes a particular scale level with its unique architecture features, such as layer numbers, width, and other parameters.

To sum up, the term "efficientdet-d0" is used to describe a specific version of the EfficientDet object detection model, distinguished by its "d0" scale level of hyperparameters. This model is carefully crafted to strike a balance between efficiency and accuracy in detecting objects.

The term "ssd-mobilenet-v2-fpnlite-320" specifically refers to a variant of the SSD object detection model. This variant incorporates the MobileNetV2 architecture as its backbone and includes Feature Pyramid Network (FPN) Lite, using an input size of 320x320 pixels. SSD, a highly efficient and real-time object detection algorithm, is able to perform both detection and classification in a single pass. By splitting the input image into a grid of cells, it generates bounding boxes and class scores for each cell. MobileNetV2, on the other hand, is a lightweight convolutional neural network designed for mobile and edge devices to achieve accuracy while prioritizing speed and efficiency.

FPN Lite is a specialized version of the innovative Feature Pyramid Network, tailor-made to tackle the challenge of varying object sizes in detection tasks. By integrating features from multiple levels of the network, FPN Lite ramps up the network's ability to accurately detect objects at different scales. Meanwhile, the designation "320" corresponds to the predetermined resolution for training and inference images, specifically 320x320 pixels. Thus, the name "ssd-mobilenet-v2-fpnlite-320" encompasses the unique combination of the efficient MobileNetV2 and FPN Lite models, optimized for scale variation, and utilizing an input image size of 320x320 pixels. This particular configuration is often the top choice for achieving top-notch performance.

```
# Change the chosen_model variable to deploy different models available in the TF2 object detection zoo
chosen_model = 'ssd-mobilenet-v2-fpnlite-320'

MODELS_CONFIG = {
    'ssd-mobilenet-v2': {
        'model_name': 'ssd_mobilenet_v2_320x320_coco17_tpu-8',
        'base_pipeline_file': 'ssd_mobilenet_v2_320x320_coco17_tpu-8.config',
        'pretrained_checkpoint': 'ssd_mobilenet_v2_320x320_coco17_tpu-8.tar.gz',
    },
    'efficientdet-d0': {
        'model_name': 'efficientdet_d0_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d0_512x512_coco17_tpu-8.config',
        'pretrained_checkpoint': 'efficientdet_d0_coco17_tpu-32.tar.gz',
    },
    'ssd-mobilenet-v2-fpnlite-320': {
        'model_name': 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8',
        'base_pipeline_file': 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.config',
        'pretrained_checkpoint': 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz',
    },
    # The centernet model isn't working as of 9/10/22
    #'centernet-mobilenet-v2': {
    #    'model_name': 'centernet_mobilenetv2fpn_512x512_coco17_od',
    #    'base_pipeline_file': 'pipeline.config',
    #    'pretrained_checkpoint': 'centernet_mobilenetv2fpn_512x512_coco17_od.tar.gz',
    #}
}

model_name = MODELS_CONFIG[chosen_model]['model_name']
pretrained_checkpoint = MODELS_CONFIG[chosen_model]['pretrained_checkpoint']
base_pipeline_file = MODELS_CONFIG[chosen_model]['base_pipeline_file']
```

Firstly, a directory labeled "mymodel" is created in the designated Google Drive location. Next, the working directory is updated to reflect the new "mymodel" folder. The URL for accessing the pre-trained model weights is then generated. The tar archive containing these weights is successfully obtained from the provided URL. In order to organize the files, the contents of the tar archive are extracted and placed into the previously created "mymodel" folder. Furthermore, the training configuration file for the model is also acquired. To do so, the URL for downloading this file is constructed and the file is successfully downloaded and saved. Overall, this code establishes a structured system for managing files related to a customized object detection model. It secures the necessary pre-trained weights and training configuration file.

```
# Create "mymodel" folder for holding pre-trained weights and configuration files
%mkdir /content/gdrive/MyDrive/TFlowLite/models/mymodel/
%cd /content/gdrive/MyDrive/TFlowLite/models/mymodel/

# Download pre-trained model weights
import tarfile
download_tar = 'http://download.tensorflow.org/models/object_detection/tf2/20200711/' + pretrained_checkpoint
!wget {download_tar}
tar = tarfile.open(pretrained_checkpoint)
tar.extractall()
tar.close()

# Download training configuration file for model
download_config = 'https://raw.githubusercontent.com/tensorflow/models/master/research/object_detection/configs/tf2/' + base_pipeline_file
!wget {download_config}
```

The number of training steps is set to 40000, which represents the duration of the model training process. During this time, the model will be updated using the labeled training data. Adjusting this parameter can impact the effectiveness and length of the training process. In this case, the value is 40,000 steps. Based on the chosen model, the batch size is determined. If the chosen model is 'efficientdet-d0,' the batch size is set to 4. This batch size refers to the number of training samples processed in each iteration. When working with models that have limited computational resources, smaller batch sizes are typically used. If a model other than 'efficientdet-d0' is chosen, the batch size is instead set to 16. For these models, larger batch sizes can be utilized during training.

```
# Set training parameters for the model
num_steps = 40000

if chosen_model == 'efficientdet-d0':
  batch_size = 4
else:
  batch_size = 16
```

This code is essential for customizing the training process for a specialized object detection model within our project. It sets up crucial file paths, specifically for the training configuration file (named base_pipeline_file) and the fine-tune checkpoint, both located in the designated "mymodel" directory. The training configuration file holds all the necessary parameters and configurations for training the model, while the fine-tune checkpoint contains pre-trained weights that act as a starting point for further enhancements. Additionally, this code defines a function that determines the total number of classes in the label map file (label_map_pbtxt_fname). This label map file connects class names to unique numeric identifiers and is vital for training an accurate object detection model. Understanding and configuring these aspects are pivotal steps in the preparation and training phases of the custom object detection model, contributing to the overall success of the project.

```
# Set file locations and get number of classes for config file
pipeline_fname = '/content/gdrive/MyDrive/TFlowLite/models/mymodel/' + base_pipeline_file
fine_tune_checkpoint = '/content/gdrive/MyDrive/TFlowLite/models/mymodel/' + model_name + '/checkpoint/ckpt-0'

def get_num_classes(pbtxt_fname):
    from object_detection.utils import label_map_util
    label_map = label_map_util.load_labelmap(pbtxt_fname)
    categories = label_map_util.convert_label_map_to_categories(
        label_map, max_num_classes=90, use_display_name=True)
    category_index = label_map_util.create_category_index(categories)
    return len(category_index.keys())
num_classes = get_num_classes(label_map_pbtxt_fname)
print('Total classes:', num_classes)
```

To properly initialize our custom object detection model, this code snippet establishes the necessary file paths and retrieves the total number of classes. The variable pipeline_fname represents the path to the configuration file for training, while fine_tune_checkpoint points to the checkpoint file that holds pretrained weights. Our code also includes the function get_num_classes, which calculates the number of classes present in the label map file, designated by the variable label_map_pbtxt_fname. By utilizing the convenient utility functions within TensorFlow's Object Detection API, this function loads the label map, converts it into categories, and determines the number of unique classes. The resulting number is then displayed on the console, providing crucial insight for accurately configuring the model during training. Overall, this code is crucial in the setup and preparation phase, ensuring successful initialization of the model.

In this segment of code, a crucial step towards configuring the custom object detection model is taken, greatly contributing to the success of our project. The code dynamically generates a custom configuration file called "pipeline_file.config," specifically tailored to meet the unique requirements of the training process. Working within the directory where the model files are stored, the code reads and modifies the content of the base pipeline file, carefully adjusting key parameters. These modifications include setting the path to the fine-tune checkpoint, specifying paths for the training and validation datasets, defining the label map path, configuring the batch size, determining the number of training steps, setting the number of classes, and ensuring that the fine-tune checkpoint type is aligned with the project. It is worth noting that our approach involves making model-specific modifications to optimize performance. For instance, to prevent any potential concerns, we decrease the learning rates for the 'ssd-mobilenet-v2' model. Furthermore, when the 'efficientdet-d0' model is selected, we fine-tune certain parameters to ensure compatibility with TensorFlow Lite. Combining all of these adjustments, we create a customized configuration file that serves as a personalized roadmap for training the object detection model on our dataset. This level of fine-tuning is crucial for achieving the highest level of performance and accuracy during the training phase.

```
# Create custom configuration file by writing the dataset, model checkpoint, and training parameters into the base pipeline file
import re

%cd /content/gdrive/MyDrive/TFlowLite/models/mymodel
print('writing custom configuration file')

with open(pipeline_fname) as f:
    s = f.read()
with open('pipeline_file.config', 'w') as f:

    # Set fine_tune_checkpoint path
    s = re.sub('fine_tune_checkpoint: ".*?"',
               'fine_tune_checkpoint: "{}"'.format(fine_tune_checkpoint), s)

    # Set tfrecord files for train and test datasets
    s = re.sub(
        '(input_path: ".*?)(PATH_TO_BE_CONFIGURED/train)(.*?")', 'input_path: "{}"'.format(train_record_fname), s)
    s = re.sub(
        '(input_path: ".*?)(PATH_TO_BE_CONFIGURED/val)(.*?")', 'input_path: "{}"'.format(val_record_fname), s)

    # Set label_map_path
    s = re.sub(
        'label_map_path: ".*?"', 'label_map_path: "{}"'.format(label_map_pbtxt_fname), s)

    # Set batch_size
    s = re.sub('batch_size: [0-9]+',
               'batch_size: {}'.format(batch_size), s)

    # Set training steps, num_steps
    s = re.sub('num_steps: [0-9]+',
               'num_steps: {}'.format(num_steps), s)

    # Set number of classes num_classes
    s = re.sub('num_classes: [0-9]+',
               'num_classes: {}'.format(num_classes), s)
```

```
# Change fine-tune checkpoint type from "classification" to "detection"
s = re.sub(
    'fine_tune_checkpoint_type: "classification"', 'fine_tune_checkpoint_type: "{}"'.format('detection'), s)

# If using ssd-mobilenet-v2, reduce learning rate (because it's too high in the default config file)
if chosen_model == 'ssd-mobilenet-v2':
  s = re.sub('learning_rate_base: .8',
             'learning_rate_base: .08', s)

  s = re.sub('warmup_learning_rate: 0.13333',
             'warmup_learning_rate: .026666', s)

# If using efficientdet-d0, use fixed_shape_resizer instead of keep_aspect_ratio_resizer (because it isn't supported by TFLite)
if chosen_model == 'efficientdet-d0':
  s = re.sub('keep_aspect_ratio_resizer', 'fixed_shape_resizer', s)
  s = re.sub('pad_to_max_dimension: true', '', s)
  s = re.sub('min_dimension', 'height', s)
  s = re.sub('max_dimension', 'width', s)

f.write(s)
```

"This optional code snippet serves the purpose of displaying the customized configuration file, named "pipeline_file.config," on the console. The shell command !cat is utilized to concatenate and show the content of a specified file. Specifically, it prints the content of the designated custom configuration file located at "/content/gdrive/MyDrive/TFlowLite/models/mymodel/pipeline_file.config." This step can greatly benefit from manual inspection and validation of the tweaks made to the configuration file. By doing so, you can ensure that the updated parameter values, file paths, and other settings accurately correspond to the changes made in the previous code section. Ultimately, displaying the content provides transparency surrounding the modifications.

```
# (Optional) Display the custom configuration file's contents
!cat  /content/gdrive/MyDrive/TFlowLite/models/mymodel/pipeline_file.config
```

Let's go over these two lines of code which are crucial for customizing our model's configuration and checkpoint storage The first line,pipeline_file = '/content/gdrive/MyDrive/TFlowLite/models/mymodel/pipeline_file.config', holds the path to the custom configuration file, aptly named "pipeline_file.config". This path leads to the specific location inside the "mymodel" directory where we previously created and modified the configuration file. Moving onto the second line, model_dir = '/content/gdrive/MyDrive/TFlowLite/training/', we set the directory where our training checkpoints will be saved to with the variable model_dir. These checkpoints serve as snapshots of our model's progress during training, being stored in this designated location.

```
# Set the path to the custom config file and the directory to store training checkpoints in
pipeline_file = '/content/gdrive/MyDrive/TFlowLite/models/mymodel/pipeline_file.config'
model_dir = '/content/gdrive/MyDrive/TFlowLite/training/'
```

This code snippet marks a pivotal moment in our project as it initiates the training process for our custom object detection model. By utilizing TensorFlow's powerful model_main_tf2.py script, this command is executed as a shell command and orchestrates the entire training workflow. The script is directed towards the custom configuration file, 'pipeline_file.config,' which houses all the tailored settings essential for the model's training. The location for storing our training checkpoints, labeled as 'model_dir,' is also clearly specified. To enable real-time monitoring, the 'alsologtostderr' flag is utilized, which effectively logs messages to both log files and standard error. The 'num_steps' parameter is a crucial aspect as it determines the total number of training steps our model will undergo, thus defining the depth and extent of the learning process.

```
# Run training!
!python /content/gdrive/MyDrive/TFlowLite/models/research/object_detection/model_main_tf2.py \
    --pipeline_config_path={pipeline_file} \
    --model_dir={model_dir} \
    --alsologtostderr \
    --num_train_steps={num_steps} \
    --sample_1_of_n_eval_examples=1
```

This code segment creates a directory called 'custom_model_lite' within a designated Google Drive path to store the trained TensorFlow Lite (TFLite) model. The variable 'last_model_path' is then assigned the path to the training directory

(/content/gdrive/MyDrive/TFlowLite/training). Using the prefix '!python,' the script 'export_tflite_graph_tf2.py' from the TensorFlow Object Detection API is executed as a shell command. This script is specifically designed to convert a trained TensorFlow model into TFLite format, and takes parameters such as the trained checkpoint directory ('trained_checkpoint_dir') and the output directory for the TFLite model ('output_directory'). With a single command, the trained object detection model transforms into a TFLite format, ready for deployment on devices with limited resources like mobile phones or edge devices. Keep the resulting TFLite model in the 'custom_model_lite' directory for smooth deployment and inference later on.

```
# Make a directory to store the trained TFLite model
!mkdir /content/gdrive/MyDrive/TFlowLite/custom_model_lite
output_directory = '/content/gdrive/MyDrive/TFlowLite/custom_model_lite'

# Path to training directory (the conversion script automatically chooses the highest checkpoint file)
last_model_path = '/content/gdrive/MyDrive/TFlowLite/training'

!python /content/gdrive/MyDrive/TFlowLite/models/research/object_detection/export_tflite_graph_tf2.py \
    --trained_checkpoint_dir {last_model_path} \
    --output_directory {output_directory} \
    --pipeline_config_path {pipeline_file}
```

At the core of this crucial section of code lies the transformation of the exported TensorFlow model. Nestled within the 'custom_model_lite' directory, the model is originally in the SavedModel format. However, leveraging the powerful TFLiteConverter in TensorFlow, the script deftly converts the model into a more petite and easily deployable TensorFlow Lite (TFLite) version. The conversion process is initiated by initializing the converter, which ingeniously processes the SavedModel and produces a TFLite model. The final result is then saved as a 'detect.tflite' binary file in the same directory. This conversion is vital for effectively deploying the object detection model on devices with limited resources such as mobile phones and edge devices, where the efficiency of TFLite models is paramount. The resulting 'detect.tflite' file is a distilled and optimized version of the trained model, ready for deployment.

```
# Convert exported graph file into TFLite model file
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model('/content/gdrive/MyDrive/TFlowLite/custom_model_lite/saved_model')
tflite_model = converter.convert()

with open('/content/gdrive/MyDrive/TFlowLite/custom_model_lite/detect.tflite', 'wb') as f:
  f.write(tflite_model)
```

The code imports essential modules from the tflite_support library that specifically handle metadata for TensorFlow Lite models. Metadata not only provides vital details about a model's properties, but also aids in documenting and interpreting the model effectively. The

object_detector submodule is specifically designed to assist in writing metadata for object detection models, such as categorical labels, input/output configurations, and other pertinent information. As for the writer_utils submodule, it likely serves as a comprehensive set of handy functions that can be utilized for writing metadata across various types of models. With these modules at hand, the TensorFlow Lite object detection model becomes highly interpretable and seamlessly integrated into diverse applications.

```
from tflite_support.metadata_writers import object_detector
from tflite_support.metadata_writers import writer_utils
```

In this specific code, the tflite_support library plays a crucial role in generating essential metadata for a TensorFlow Lite object detection model. By doing so, the model becomes more interpretable and can be easily integrated into a variety of applications. To achieve this, the ObjectDetectorWriter class from the object_detector submodule within the library is utilized for the creation of metadata. The designated TensorFlow Lite model, which was previously converted and saved as "detect.tflite," is specified through the _MODEL_PATH variable. Furthermore, the label file "labelmap.txt," containing a list of classes for the object detection model, is also referenced with the help of the _LABEL_FILE variable. As a result, the final model is enriched with normalization parameters and class labels, and is saved as "detect_metadata.tflite" at the dedicated path.

```
ObjectDetectorWriter = object_detector.MetadataWriter
_MODEL_PATH = "/content/gdrive/MyDrive/TFlowLite/custom_model_lite/detect.tflite"
# Task Library expects label files that are in the same format as the one below.
_LABEL_FILE = "/content/labelmap.txt"
_SAVE_TO_PATH = "/content/gdrive/MyDrive/TFlowLite/custom_model_lite/detect_metadata.tflite"
# Normalization parameters is required when reprocessing the image. It is
# optional if the image pixel values are in range of [0, 255] and the input
# tensor is quantized to uint8. See the introduction for normalization and
# quantization parameters below for more details.
# https://www.tensorflow.org/lite/models/convert/metadata#normalization_and_quantization_parameters)
_INPUT_NORM_MEAN = 127.5
_INPUT_NORM_STD = 127.5

# Create the metadata writer.
writer = ObjectDetectorWriter.create_for_inference(
    writer_utils.load_file(_MODEL_PATH), [_INPUT_NORM_MEAN], [_INPUT_NORM_STD],
    [_LABEL_FILE])

# Verify the metadata generated by metadata writer.
print(writer.get_metadata_json())

# Populate the metadata into the model.
writer_utils.save_file(writer.populate(), _SAVE_TO_PATH)
```

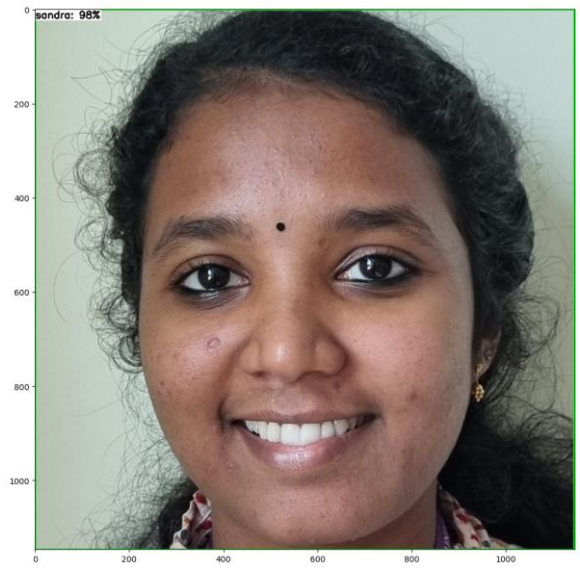
In this whole code segment, we expertly carried out the necessary tasks to create and deploy a TensorFlow Lite model for object detection. To start, we seamlessly enabled access to data within the Colab environment by setting up Google Drive. We then obtained the TensorFlow Models repository and skillfully configured paths for our training data and label maps to ensure precise training and evaluation. We carefully considered selecting the appropriate object detection model, and we skillfully downloaded pre-trained weights and configurations for efficient transfer learning. Essential parameters for training, such as the number of steps and batch size, were thoughtfully chosen. Our final steps included customizing the model's configuration and generating metadata to enhance interpretability and enable deployment on resource-constrained devices. This succinct yet comprehensive code segment encapsulates our thorough approach.

RESULTS

In our project, we built a TensorFlow Lite (TFLite) model to detect objects in test photos. The goal was to use a custom-trained model to detect objects in a number of test images. To get accurate object detection, this was implemented using Python code and various libraries and functionalities.

The Python code has been created to run a custom TFLite model on a number of test images to detect faces. The code begins by importing the required libraries, such as OpenCV, NumPy, and TensorFlow Lite. The main part of the code is the tflite_detect_images function, which handles model inference on test images. It accepts parameters such as the model path, image path, label path, minimum confidence threshold, number of test images, and results save path. The function loads the TFLite model into memory and retrieves information such as input and output data. It then loops through a set number of randomly selected test photos, detects faces, and returns detection results. On the photos, the code generates bounding boxes around identified faces and displays labels and confidence percentages.

Subsequently, the code executed the model inference on the selected test images, leveraging the TensorFlow Lite interpreter. Object detection was performed, and bounding boxes, object labels, and confidence scores were annotated on the images. This allowed visual verification of the model's performance in detecting objects within the test images.



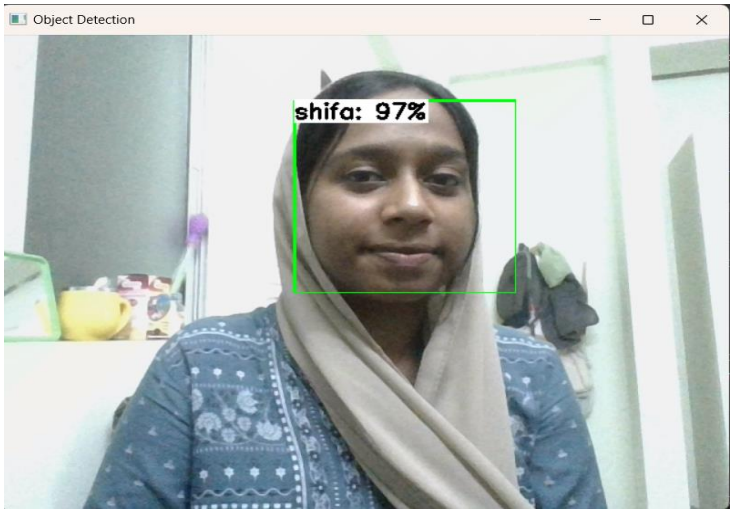
TensorFlow Lite is used in real-time object recognition by the code which includes object detection from a camera stream with attendance monitoring. In order to recognize objects, the tflite_detect_webcam function loads a pre-trained TensorFlow Lite model and starts webcam capture. Additionally, it makes use of a Haar Cascade classifier to recognize faces in the camera frames.

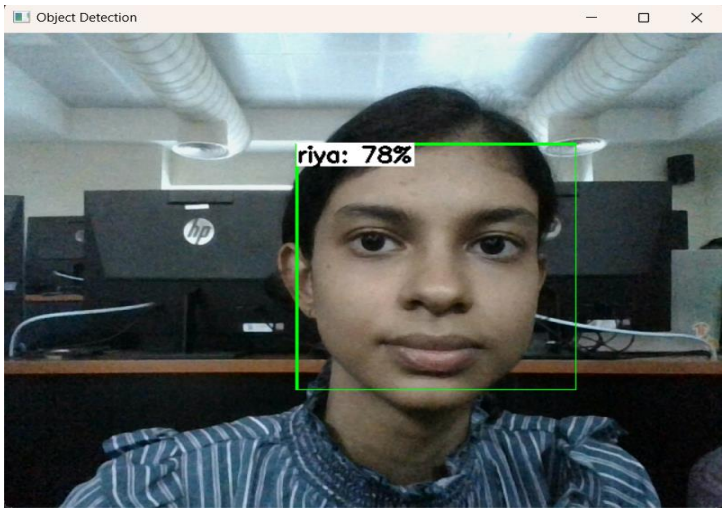
The method preprocesses and resizes the face regions to match the input size of the model once faces have been discovered. These scaled face regions are then subjected to object detection, whereby bounding boxes are drawn around identified items and their corresponding confidences are labeled. This enables the system to recognize and categorize things contained within the webcam's captured faces.

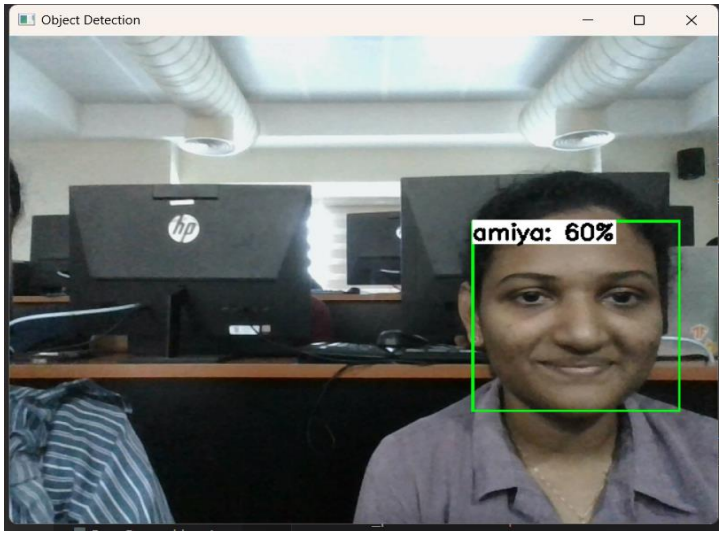
Integration with attendance tracking is one of its notable characteristics. The code tries to update the attendance for the person it detects when it recognizes their face likely that of a student. It manages attendance data using an Excel file called "students.xlsx," designating the recognized student as "Present" for the current day. It changes the attendance if the date column is present; if not, a new column is created for the current date and adds the attendance details under that date.

The code runs continuously, gathering frames from the webcam, identifying faces and objects, changing attendance when necessary, and performing other tasks. Because of its real-time capabilities, it can be used in situations where it's necessary to track attendance, like in educational institutions or event management systems.

The results from the objection detection with the labels as the name of person and the accuracy of that prediction in real time is given in the figure.






Object Detection
amiya: 60%

CONCLUSION

In conclusion, the development and implementation of the automated attendance marking system using TensorFlow Lite represent a significant leap towards efficient and accurate attendance management in educational and organizational environments. The project successfully demonstrates the potential of advanced technologies like machine learning and computer vision in solving real-world challenges.

By integrating real-time object detection and facial recognition capabilities, the system offers a seamless solution for attendance tracking, significantly reducing manual efforts and potential errors associated with traditional methods. Its ability to dynamically update attendance records, recognize students, and mark their presence in an Excel file showcases the system's practicality and adaptability for diverse settings.

In essence, this project serves as a demonstration to the transformative impact of technology in simplifying complex tasks. By providing an automated solution for attendance marking, it not only enhances operational efficiency but also lays the groundwork for future innovations in leveraging machine learning for everyday applications. As technology continues to evolve, the system stands as a demonstration to the possibilities of smart, automated systems in enhancing productivity and accuracy across diverse domains.

REFERENCES

1. Kar, N., Debbarma, M. K., Saha, A., & Pal, D. R. (2012). Study of implementing automated attendance system using face recognition technique. International Journal of computer and communication engineering, 1(2), 100-103.
2. Lukas, S., Mitra, A. R., Desanti, R. I., & Krisnadi, D. (2016, October). Student attendance system in classroom using face recognition technique. In 2016 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 1032-1035). IEEE.
3. Sawhney, S., Kacker, K., Jain, S., Singh, S. N., & Garg, R. (2019, January). Real-time smart attendance system using face recognition techniques. In 2019 9th international conference on cloud computing, data science & engineering (Confluence) (pp. 522-525). IEEE.
4. Jadhav, A., Jadhav, A., Ladhe, T., & Yeolekar, K. (2017). Automated attendance system using face recognition. International Research Journal of Engineering and Technology (IRJET), 4(1), 1467-1471.
5. Patil, A., & Shukla, M. (2014). Implementation of classroom attendance system based on face recognition in class. International Journal of Advances in Engineering & Technology, 7(3), 974.