

# Vehicle Movement Analysis and Insight Generation Project

## Title

"Vehicle Movement Analysis and Insight Generation in a College Campus using Edge AI"

## Objectives

The primary objectives of this project are as follows:

- **An Effective ANPR System will be developed:** A robust Automatic Number Plate Recognition (ANPR) system will be created, capable of accurately identifying and tracking vehicles in real-time.
- **Parking Management will be enhanced:** A parking space detection and recognition model will be implemented to optimize parking lot utilization and reduce congestion.
- **Edge AI will be leveraged:** Models will be deployed on edge devices using TensorFlow Lite (TFLite) to ensure real-time processing and low latency.
- **Actionable Insights will be generated:** Vehicle movement and parking patterns will be analyzed to provide valuable insights for infrastructure planning and traffic management on the college campus.

## Problem Description

Significant challenges in managing vehicle movement and parking are faced by college campuses due to high traffic volumes and limited parking space availability. Inefficient traffic flow and poorly managed parking can lead to congestion, increased wait times, and frustration among students, staff, and visitors. Traditional traffic management systems often rely on manual monitoring and static solutions, which are insufficient for dynamic and real-time traffic conditions.

To address these issues, the project aims to harness the power of AI and edge computing. By developing an integrated system for ANPR and parking space detection, a real-time, scalable solution will be provided to enhance traffic flow, optimize parking space usage, and generate valuable insights for decision-making. This approach improves the immediate traffic situation and contributes to long-term infrastructure planning and resource allocation on the campus.

## Dataset

A diverse dataset comprising images and videos collected from various parking lots and campus roads was utilized for this project. The dataset includes annotated images for Automatic Number Plate Recognition (ANPR) and parking space detection. Each image is labeled with bounding boxes around vehicles and their license plates, and parking space occupancy status. This dataset was essential for training and validating the models, ensuring accurate identification and analysis of vehicle movements and parking space availability in real-world scenarios.

## Methodology

The methodology involves several key steps:

- **Data Preprocessing:** The dataset was cleaned and annotated to ensure high-quality training data for the models.
- **Model Training:** TensorFlow Object Detection (TFOD) and the my\_ssd\_mobnet model were employed to train the ANPR system. For parking space detection, the SuperGradients library was used with the annotated parking lot dataset.
- **Model Optimization:** The trained models were converted to TensorFlow Lite (TFLite) to facilitate efficient edge deployment, optimizing for speed and accuracy.
- **Edge Deployment:** The optimized models were deployed on edge devices to enable real-time vehicle and parking space analysis with low latency.
- **Evaluation:** The models were rigorously tested against a separate validation dataset to assess their performance, accuracy, and reliability in various conditions.

## Evaluation

To ensure the robustness and effectiveness of the models, a comprehensive evaluation process was conducted:

- **Accuracy:** The accuracy of the ANPR system in detecting and recognizing license plates under various conditions, including different lighting and angles, was measured. The parking space detection model was evaluated based on its ability to correctly identify occupied and vacant spaces.
- **Inference Speed:** The performance of the models was assessed in terms of inference speed, particularly on edge devices. The conversion to TensorFlow Lite (TFLite) aimed to ensure that the models could operate in real-time with minimal latency.
- **Scalability:** The scalability of the system was tested by deploying it in different parking lots and road conditions within the campus. This ensured that the models could handle varying levels of traffic and environmental factors.
- **Reliability:** The models were subjected to rigorous testing to evaluate their reliability and consistency. This involved running the models over extended periods to identify any potential issues or performance degradation.

## Step-by-Step Solution for Parking Space Detection

### Step 1: Initializing the libraries

- Essential libraries such as os, random, torch, requests, and PIL were imported.
- Training, loss, metrics, data loading, and model functions from the super\_gradients library were imported.

```
import os
import random
import torch
import requests
from PIL import Image

from super_gradients.training import Trainer, dataloaders, models
from super_gradients.training.losses import PPYoLoELoss
from super_gradients.training.metrics import DetectionMetrics_050

from super_gradients.training.dataloaders.dataloaders import (
    coco_detection_yolo_format_train,
    coco_detection_yolo_format_val
)

from super_gradients.training.models.detection_models.pp_yolo_e import (
    PPYoLoEPostPredictionCallback
)
```

## Step 2: Initializing the paths

- A configuration class was created to specify paths for saving checkpoints, experiment names, and dataset directories.
- Directory paths for training, validation, and test data (both images and labels) were set.
- The classes and the number of classes for parking space detection were defined.
- Dataloader parameters, including batch size and the number of workers, were specified.
- The model name (yolo\_nas\_l) and the pretrained weights (coco) were chosen.

```

class config:
    #trainer params
    CHECKPOINT_DIR = 'checkpoints' #specify the path you want to save checkpoints to

    EXPERIMENT_NAME = 'race_number' #specify the experiment name
    #dataset params

    DATA_DIR = '/content/datasets' #parent directory to where data lives
    TRAIN_IMAGES_DIR = 'train/images' #child dir of DATA_DIR where train images are
    TRAIN_LABELS_DIR = 'train/labels' #child dir of DATA_DIR where train labels are
    VAL_IMAGES_DIR = 'valid/images' #child dir of DATA_DIR where validation images
are
    VAL_LABELS_DIR = 'valid/labels' #child dir of DATA_DIR where validation labels
are
    # if you have a test set
    TEST_IMAGES_DIR = 'test/images' #child dir of DATA_DIR where test images are
    TEST_LABELS_DIR = 'test/labels' #child dir of DATA_DIR where test labels are

    CLASSES =
['free_parking_space', 'not_free_parking_space', 'partially_free_parking_space']
    NUM_CLASSES = len(CLASSES)

    #dataloader params - you can add whatever PyTorch dataloader params you have
    #could be different across train, val, and test
    DATALOADER_PARAMS={
        'batch_size': 4,
        'num_workers': 2
    }

    # model params
    MODEL_NAME = 'yolo_nas_l' # choose from yolo_nas_s, yolo_nas_m, yolo_nas_l
    PRETRAINED_WEIGHTS = 'coco' #only one option here: coco

```

### Step 3: Initializing the training model

- The Trainer was initialized with the experiment name and checkpoint directory, as specified in the configuration.
- Paths for training data, including directories for images, labels, and classes, were defined.
- Dataloader parameters, such as batch size and number of workers, were set for the training data.
- Paths for validation data, including directories for images, labels, and classes, were defined.
- Dataloader parameters for validation data were set.
- Paths for test data, including directories for images, labels, and classes, were defined.

- Dataloader parameters for test data were set.

```
trainer = Trainer(experiment_name=config.EXPERIMENT_NAME,
                  ckpt_root_dir=config.CHECKPOINT_DIR)
train_data = coco_detection_yolo_format_train(
    dataset_params={
        'data_dir': config.DATA_DIR,
        'images_dir': config.TRAIN_IMAGES_DIR,
        'labels_dir': config.TRAIN_LABELS_DIR,
        'classes': config.CLASSES
    },
    dataloader_params=config.DATALOADER_PARAMS
)

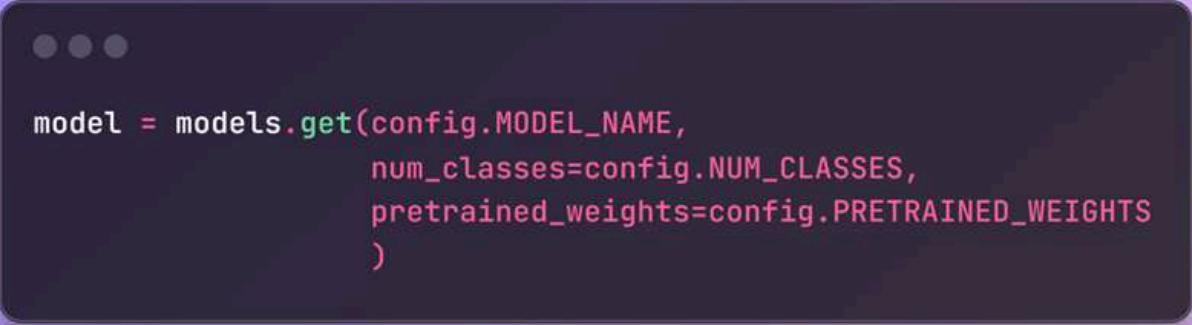
val_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': config.DATA_DIR,
        'images_dir': config.VAL_IMAGES_DIR,
        'labels_dir': config.VAL_LABELS_DIR,
        'classes': config.CLASSES
    },
    dataloader_params=config.DATALOADER_PARAMS
)

test_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': config.DATA_DIR,
        'images_dir': config.TEST_IMAGES_DIR,
        'labels_dir': config.TEST_LABELS_DIR,
        'classes': config.CLASSES
    },
    dataloader_params=config.DATALOADER_PARAMS
)
```

#### Step 4: Setting up the training model

- The model name (yolo\_nas\_l) was chosen.

- The number of classes and pretrained weights (e.g., coco) were specified.



```
model = models.get(config.MODEL_NAME,  
                   num_classes=config.NUM_CLASSES,  
                   pretrained_weights=config.PRETRAINED_WEIGHTS  
                   )
```

### Step 5: Specifying the training parameters

- Averaging of the best models was enabled.
- Learning rate schedules and warmup parameters were defined.
- The optimizer type and its parameters were set.
- Exponential Moving Average (EMA) with its parameters was enabled.
- The maximum number of epochs was set.
- Mixed precision training was enabled.
- The loss function and its parameters were defined.
- Validation metrics and the primary metric to monitor during training (mAP@0.50) were specified.



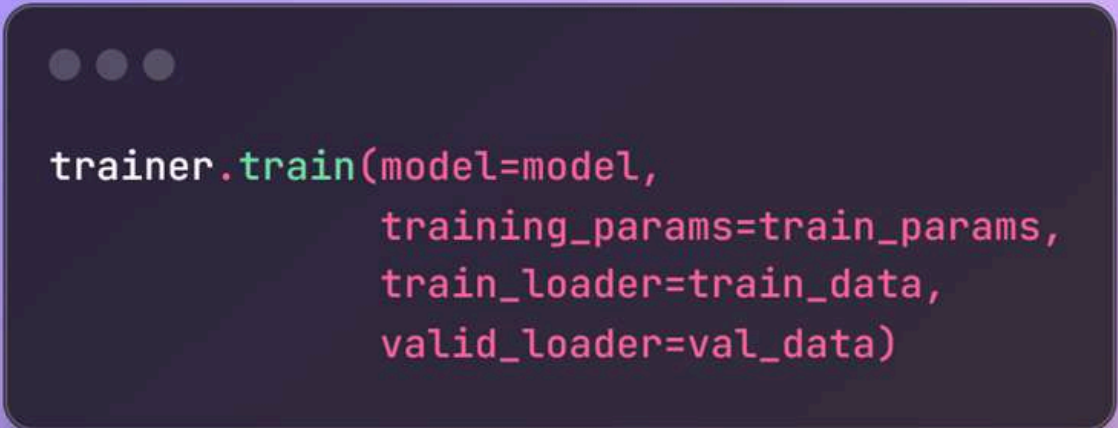
```

train_params = {
    # ENABLING SILENT MODE
    "average_best_models": True,
    "warmup_mode": "linear_epoch_step",
    "warmup_initial_lr": 1e-6,
    "lr_warmup_epochs": 3,
    "initial_lr": 5e-4,
    "lr_mode": "cosine",
    "cosine_final_lr_ratio": 0.1,
    "optimizer": "AdamW",
    "optimizer_params": {"weight_decay": 0.0001},
    "zero_weight_decay_on_bias_and_bn": True,
    "ema": True,
    "ema_params": {"decay": 0.9, "decay_type": "threshold"},
    "max_epochs": 30,
    "mixed_precision": True, #mixed precision is not available for CPU
    "loss": PPYoLoELoss(
        use_static_assigner=False,
        # NOTE: num_classes needs to be defined here
        num_classes=config.NUM_CLASSES,
        reg_max=16
    ),
    "valid_metrics_list": [
        DetectionMetrics_050(
            score_thres=0.1,
            top_k_predictions=300,
            # NOTE: num_classes needs to be defined here
            num_cls=config.NUM_CLASSES,
            normalize_targets=True,
            post_prediction_callback=PPYoLoEPostPredictionCallback(
                score_threshold=0.01,
                nms_top_k=1000,
                max_predictions=300,
                nms_threshold=0.7
            )
        )
    ],
    "metric_to_watch": 'mAP@0.50'
}

```

## Step 6: Start the training process

- The training process was initiated with `trainer.train`.
- The model to be trained was specified with `model=model`.
- Parameters and hyperparameters for training were contained in `training_params=train_params`.
- DataLoader for the training dataset was specified with `train_loader=train_data`.
- DataLoader for the validation dataset was specified with `valid_loader=val_data`.

A code editor window with a dark background and three light gray window control buttons in the top-left corner. It contains a single line of Python code with syntax highlighting: `trainer.train(model=model, training_params=train_params, train_loader=train_data, valid_loader=val_data)`.

```
trainer.train(model=model,  
              training_params=train_params,  
              train_loader=train_data,  
              valid_loader=val_data)
```

## Step 7: Selecting the best model

- The model with the specified configuration was loaded with `models.get`.
- The name of the model architecture to be loaded was specified with `config.MODEL_NAME`.
- The number of classes for the model's output layer was specified with `num_classes=config.NUM_CLASSES`.
- The path to the checkpoint file of the best model was specified with `checkpoint_path`.



```
best_model = models.get(config.MODEL_NAME,  
                        num_classes=config.NUM_CLASSES,  
  
checkpoint_path=os.path.join('/content', config.CHECKPOINT_DIR,  
config.EXPERIMENT_NAME, 'RUN_20240709_102236_971361/average_model.pth'))
```

## Step 8: Specifying the testing parameters

- The testing process was initiated with `trainer.test`.
- The best model obtained from training was specified with `model=best_model`.
- `DataLoader` for the test dataset was specified with `test_loader=test_data`.
- Metrics for evaluating the model, with thresholds and top-k predictions, were specified with `test_metrics_list=DetectionMetrics_050`.
- The number of classes for the model's output layer was specified with `num_cls=config.NUM_CLASSES`.
- Normalizing the targets before evaluation was enabled with `normalize_targets=True`.
- Post-processing steps for the predictions, including Non-Maximum Suppression (NMS) parameters, were specified with `post_prediction_callback=PPYoloEPostPredictionCallback`.

```
trainer.test(model=best_model,
             test_loader=test_data,
             test_metrics_list=DetectionMetrics_050(score_thres=0.1,
             top_k_predictions=300,
             num_cls=config.NUM_CLASSES,
             normalize_targets=True,
             post_prediction_callback=PPYoloEPostPredictionCallback(
                 score_threshold=0.01,
                 nms_top_k=1000,
                 max_predictions=300,
                 nms_threshold=0.7)
             ))
```

### Step 9: Testing the model

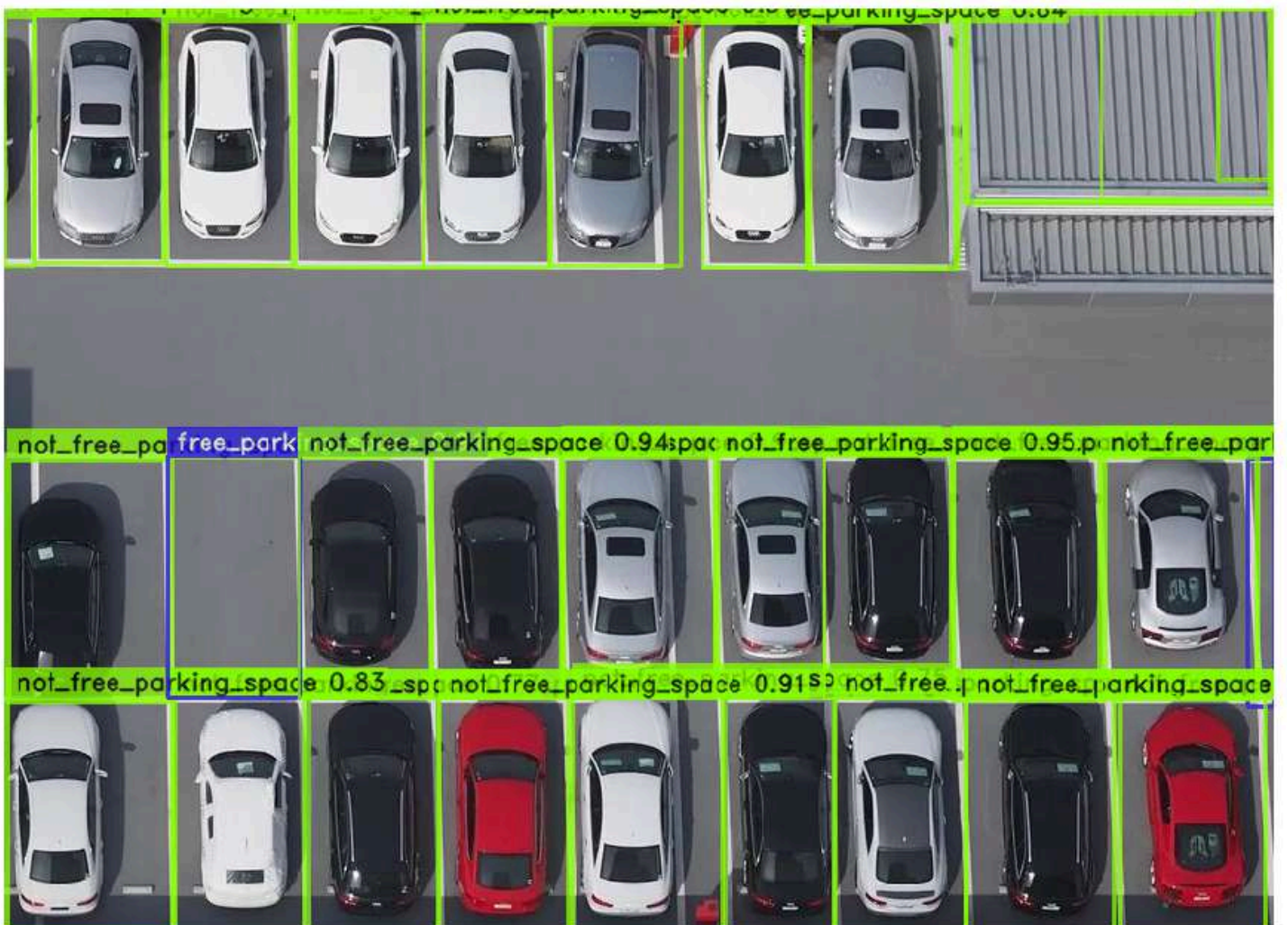
- An empty list called tpaths was initialized.
- The directory /content/datasets/test/images was traversed recursively using os.walk.
- For each filename in the directory, the full path was constructed by joining dirname and filename and added to the tpaths list.
- The number of image paths collected in the tpaths list was printed.
- Predictions on the images in tpaths were made using the best-trained model with a confidence threshold of 0.4, and the results were displayed with best\_model.predict(tpaths, conf=0.4).show().

```

tpaths=[]
for dirname, _, filenames in os.walk('/content/datasets/test/images'):
    for filename in filenames:
        tpaths+=(os.path.join(dirname, filename))
print(len(tpaths))
best_model.predict(tpaths, conf=0.4).show()

```

## Output



# Step-by-Step Solution for Vehicle Movement Analysis using ANPR

## step 1: Setup Paths

- **Essential variables and libraries were imported:**
  - os was imported for directory and path management.
  - The custom model name was set to 'my\_ssd\_mobnet'.
  - The pretrained model name was set to 'ssd\_mobilenet\_v2\_fpn-lite\_320x320\_coco17\_tpu-8'.
  - The pretrained model URL was set to 'http://download.tensorflow.org/models/object\_detection/tf2/20200711/ssd\_mobilenet\_v2\_fpn-lite\_320x320\_coco17\_tpu-8.tar.gz'.
  - The TF record script name was set to 'generate\_tfrecord.py'.
  - The label map name was set to 'label\_map.pbtxt'.
- **Paths were initialized using os.path.join:**
  - The workspace path was set to os.path.join('Tensorflow', 'workspace').
  - The scripts path was set to os.path.join('Tensorflow','scripts').
  - The API model path was set to os.path.join('Tensorflow','models').
  - The annotation path was set to os.path.join('Tensorflow', 'workspace','annotations').
  - The image path was set to os.path.join('Tensorflow', 'workspace','images').
  - The model path was set to os.path.join('Tensorflow', 'workspace','models').
  - The pretrained model path was set to os.path.join('Tensorflow', 'workspace','pre-trained-models').
  - The checkpoint path was set to os.path.join('Tensorflow', 'workspace','models', CUSTOM\_MODEL\_NAME).
  - The output path was set to os.path.join('Tensorflow', 'workspace','models', CUSTOM\_MODEL\_NAME, 'export').
  - The TFJS export path was set to os.path.join('Tensorflow', 'workspace','models', CUSTOM\_MODEL\_NAME, 'tfjsexport').
  - The TFLite export path was set to os.path.join('Tensorflow', 'workspace','models', CUSTOM\_MODEL\_NAME, 'tfliteexport').
  - The Protobuf compiler path was set to os.path.join('Tensorflow', 'protoc').
- **File paths were set using os.path.join:**
  - The pipeline config file path was set to os.path.join('Tensorflow', 'workspace','models', CUSTOM\_MODEL\_NAME, 'pipeline.config').
  - The TF record script file path was set to os.path.join(paths['SCRIPTS\_PATH'], TF\_RECORD\_SCRIPT\_NAME).
  - The label map file path was set to os.path.join(paths['ANNOTATION\_PATH'], LABEL\_MAP\_NAME).
- **Directory paths were iterated over using paths.values():**
  - For each path, it was checked if the directory exists using os.path.exists(path).
  - If the directory did not exist:
    - For POSIX systems, the directory was created using !mkdir -p {path}.
    - For NT systems, the directory was created using !mkdir {path}.



```

import os

CUSTOM_MODEL_NAME = 'my_ssd_mobnet'
PRETRAINED_MODEL_NAME = 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8'
PRETRAINED_MODEL_URL =
'http://download.tensorflow.org/models/object_detection/tf2/20200711/ssd_mobilenet_v2_fpnlite
8.tar.gz'
TF_RECORD_SCRIPT_NAME = 'generate_tfrecord.py'
LABEL_MAP_NAME = 'label_map.pbtxt'

paths = {
    'WORKSPACE_PATH': os.path.join('Tensorflow', 'workspace'),
    'SCRIPTS_PATH': os.path.join('Tensorflow', 'scripts'),
    'APIMODEL_PATH': os.path.join('Tensorflow', 'models'),
    'ANNOTATION_PATH': os.path.join('Tensorflow', 'workspace', 'annotations'),
    'IMAGE_PATH': os.path.join('Tensorflow', 'workspace', 'images'),
    'MODEL_PATH': os.path.join('Tensorflow', 'workspace', 'models'),
    'PRETRAINED_MODEL_PATH': os.path.join('Tensorflow', 'workspace', 'pre-trained-models'),
    'CHECKPOINT_PATH': os.path.join('Tensorflow', 'workspace', 'models', CUSTOM_MODEL_NAME),
    'OUTPUT_PATH': os.path.join('Tensorflow', 'workspace', 'models', CUSTOM_MODEL_NAME, 'export'),
    'TFJS_PATH': os.path.join('Tensorflow', 'workspace', 'models', CUSTOM_MODEL_NAME, 'tfjs'),
    'TFLITE_PATH': os.path.join('Tensorflow', 'workspace', 'models', CUSTOM_MODEL_NAME, 'tflite'),
    'PROTOC_PATH': os.path.join('Tensorflow', 'protoc')
}

files = {
    'PIPELINE_CONFIG': os.path.join('Tensorflow', 'workspace', 'models', CUSTOM_MODEL_NAME, 'pipeline_config.pbtxt'),
    'TF_RECORD_SCRIPT': os.path.join(paths['SCRIPTS_PATH'], TF_RECORD_SCRIPT_NAME),
    'LABELMAP': os.path.join(paths['ANNOTATION_PATH'], LABEL_MAP_NAME)
}

for path in paths.values():
    if not os.path.exists(path):
        if os.name == 'posix':
            !mkdir -p {path}
        if os.name == 'nt':
            !mkdir {path}

```

## Step 2: Download TF Models Pretrained Models from Tensorflow Model Zoo and Install TFOD

- **Windows Setup:** wget was installed and imported.

- **Repository Clone:** The TensorFlow models repository was cloned if the object\_detection directory did not exist.
- **Object Detection API Installation:**
- **POSIX:** Protobuf compiler was installed, protocol buffers were compiled, and the setup script was executed.
- **Windows:** Protobuf compiler was downloaded, extracted, added to PATH, protocol buffers were compiled, and the setup script was executed.
- **Installation Verification:** The installation was verified using the model\_builder\_tf2\_test.py script.
- **Pretrained Model Download:**
- **POSIX:** Model was downloaded, moved, and extracted.
- **Windows:** Model was downloaded, moved, and extracted.



```

if os.name=='nt':
    %pip install wget
    import wget

if not os.path.exists(os.path.join(paths['APIMODEL_PATH'], 'research',
'object_detection')):
    !git clone https://github.com/tensorflow/models {paths['APIMODEL_PATH']}

# Install Tensorflow Object Detection
if os.name=='posix':
    !sudo apt-get install protobuf-compiler
    !cd Tensorflow/models/research && protoc object_detection/protos/*.proto --
python_out=. && cp object_detection/packages/tf2/setup.py . && python -m pip install
.

if os.name=='nt':

url="https://github.com/protocolbuffers/protobuf/releases/download/v3.15.6/protoc-
3.15.6-win64.zip"
wget.download(url)
!move protoc-3.15.6-win64.zip {paths['PROTOC_PATH']}
!cd {paths['PROTOC_PATH']} && tar -xf protoc-3.15.6-win64.zip
os.environ['PATH'] += os.pathsep +
os.path.abspath(os.path.join(paths['PROTOC_PATH'], 'bin'))
!cd Tensorflow/models/research && protoc object_detection/protos/*.proto --
python_out=. && copy object_detection\\packages\\tf2\\setup.py setup.py && python
setup.py build && python setup.py install
!cd Tensorflow/models/research/slim && pip install -e .

VERIFICATION_SCRIPT = os.path.join(paths['APIMODEL_PATH'], 'research',
'object_detection', 'builders', 'model_builder_tf2_test.py')
# Verify Installation
!python {VERIFICATION_SCRIPT}

if os.name == 'posix':
    !wget {PRETRAINED_MODEL_URL}
    !mv {PRETRAINED_MODEL_NAME+'.tar.gz'} {paths['PRETRAINED_MODEL_PATH']}
    !cd {paths['PRETRAINED_MODEL_PATH']} && tar -zxvf
{PRETRAINED_MODEL_NAME+'.tar.gz'}
if os.name == 'nt':
    wget.download(PRETRAINED_MODEL_URL)
    !move {PRETRAINED_MODEL_NAME+'.tar.gz'} {paths['PRETRAINED_MODEL_PATH']}
    !cd {paths['PRETRAINED_MODEL_PATH']} && tar -zxvf
{PRETRAINED_MODEL_NAME+'.tar.gz'}

```

### Step 3: Create Label Map

- A list of labels was defined as `labels = [{'name':'licence', 'id':1}]`.
- Writing to the file specified by `files['LABELMAP']`, each label was formatted with:
  - The label name, such as 'licence'.
  - The corresponding identifier, like 1.

```
labels = [{'name':'licence', 'id':1}]

with open(files['LABELMAP'], 'w') as f:
    for label in labels:
        f.write('item { \n')
        f.write('\tname:\'{ }\'\n'.format(label['name']))
        f.write('\tid:{ }\n'.format(label['id']))
        f.write('}\n')
```

#### Step 4: Create TF records

- Checking for the existence of `ARCHIVE_FILES` at `{paths['IMAGE_PATH']}/archive.tar.gz`.
- If found, extracting files using `tar -zxvf {ARCHIVE_FILES}`.
- Cloning the repository from <https://github.com/nicknochnack/GenerateTFRecord> to `{paths['SCRIPTS_PATH']}` if `{files['TF_RECORD_SCRIPT']}` does not exist.
- Executing the TFRecord script `{files['TF_RECORD_SCRIPT']}`:
- Generating train.record from images at `{os.path.join(paths['IMAGE_PATH'], 'train')}` labeled with `{files['LABELMAP']}`, outputting to `{os.path.join(paths['ANNOTATION_PATH'], 'train.record')}`.
- Generating test.record from images at `{os.path.join(paths['IMAGE_PATH'], 'test')}` labeled with `{files['LABELMAP']}`, outputting to `{os.path.join(paths['ANNOTATION_PATH'], 'test.record')}`.

```

## OPTIONAL IF RUNNING ON COLAB
ARCHIVE_FILES = os.path.join(paths['IMAGE_PATH'], 'archive.tar.gz')
if os.path.exists(ARCHIVE_FILES):
    !tar -zxvf {ARCHIVE_FILES}
#####

if not os.path.exists(files['TF_RECORD_SCRIPT']):
    !git clone https://github.com/nicknochnack/GenerateTFRecord
    {paths['SCRIPTS_PATH']}

!python {files['TF_RECORD_SCRIPT']} -x {os.path.join(paths['IMAGE_PATH'], 'train')}
-l {files['LABELMAP']} -o {os.path.join(paths['ANNOTATION_PATH'], 'train.record')}
!python {files['TF_RECORD_SCRIPT']} -x {os.path.join(paths['IMAGE_PATH'], 'test')} -
l {files['LABELMAP']} -o {os.path.join(paths['ANNOTATION_PATH'], 'test.record')}

```

## Step 5: Copy Model Config to Training Folder

- If the operating system is POSIX:
  - The pipeline configuration file located at {os.path.join(paths['PRETRAINED\_MODEL\_PATH'], PRETRAINED\_MODEL\_NAME, 'pipeline.config')} was copied to {os.path.join(paths['CHECKPOINT\_PATH'])}.
- If the operating system is Windows:
  - The pipeline configuration file located at {os.path.join(paths['PRETRAINED\_MODEL\_PATH'], PRETRAINED\_MODEL\_NAME, 'pipeline.config')} was copied to {os.path.join(paths['CHECKPOINT\_PATH'])}.

```

if os.name == 'posix':
    !cp {os.path.join(paths['PRETRAINED_MODEL_PATH'], PRETRAINED_MODEL_NAME,
    'pipeline.config')} {os.path.join(paths['CHECKPOINT_PATH'])}
if os.name == 'nt':
    !copy {os.path.join(paths['PRETRAINED_MODEL_PATH'], PRETRAINED_MODEL_NAME,
    'pipeline.config')} {os.path.join(paths['CHECKPOINT_PATH'])}

```

## Step 6: Update Config For Transfer Learning

- TensorFlow and necessary modules were imported.
- Configuration settings were fetched from the pipeline file at files['PIPELINE\_CONFIG'].
- The pipeline configuration was initialized and parsed.
- Adjustments were made to the pipeline configuration:
  - The number of classes in the SSD model was set to match the count of labels.
  - Batch size for training was configured to 4.
  - The fine-tune checkpoint path was set to {os.path.join(paths['PRETRAINED\_MODEL\_PATH'], PRETRAINED\_MODEL\_NAME, 'checkpoint', 'ckpt-0')}.
  - Checkpoint type was specified as "detection".
  - Training and evaluation input paths were updated to use TFRecord files located at {os.path.join(paths['ANNOTATION\_PATH'], 'train.record')} and {os.path.join(paths['ANNOTATION\_PATH'], 'test.record')}, respectively.
- Configuration changes were converted to text format and written back to files['PIPELINE\_CONFIG'].



```

import tensorflow as tf
from object_detection.utils import config_util
from object_detection.protos import pipeline_pb2
from google.protobuf import text_format

config = config_util.get_configs_from_pipeline_file(files['PIPELINE_CONFIG'])

pipeline_config = pipeline_pb2.TrainEvalPipelineConfig()
with tf.io.gfile.GFile(files['PIPELINE_CONFIG'], "r") as f:
    proto_str = f.read()
    text_format.Merge(proto_str, pipeline_config)

pipeline_config.model.ssd.num_classes = len(labels)
pipeline_config.train_config.batch_size = 4
pipeline_config.train_config.fine_tune_checkpoint =
os.path.join(paths['PRETRAINED_MODEL_PATH'], PRETRAINED_MODEL_NAME, 'checkpoint',
'ckpt-0')
pipeline_config.train_config.fine_tune_checkpoint_type = "detection"
pipeline_config.train_input_reader.label_map_path= files['LABELMAP']
pipeline_config.train_input_reader.tf_record_input_reader.input_path[:] =
[os.path.join(paths['ANNOTATION_PATH'], 'train.record')]
pipeline_config.eval_input_reader[0].label_map_path = files['LABELMAP']
pipeline_config.eval_input_reader[0].tf_record_input_reader.input_path[:] =
[os.path.join(paths['ANNOTATION_PATH'], 'test.record')]

config_text = text_format.MessageToString(pipeline_config)
with tf.io.gfile.GFile(files['PIPELINE_CONFIG'], "wb") as f:
    f.write(config_text)

```

## Step 7: Train the model

- Defined TRAINING\_SCRIPT as {os.path.join(paths['APIMODEL\_PATH'], 'research', 'object\_detection', 'model\_main\_tf2.py')}.
- Constructed a command string for training:
  - python {} --model\_dir={} --pipeline\_config\_path={} --num\_train\_steps=10000.
- Printed the constructed command string for verification.
- Executed the training command using !{command}.

```

TRAINING_SCRIPT = os.path.join(paths['APIMODEL_PATH'], 'research',
                                'object_detection', 'model_main_tf2.py')

command = "python {} --model_dir={} --pipeline_config_path={} --
num_train_steps=10000".format(TRAINING_SCRIPT,
                                paths['CHECKPOINT_PATH'], files['PIPELINE_CONFIG'])

print(command)

!{command}

```

## Step 8: Evaluate the Model

- Constructed a command string for training:
  - `python {} --model_dir={} --pipeline_config_path={} --checkpoint_dir={}`.
- Printed the constructed command string for verification.
- Executed the training command using `!{command}`.

```

command = "python {} --model_dir={} --pipeline_config_path={} --checkpoint_dir=
{}".format(TRAINING_SCRIPT, paths['CHECKPOINT_PATH'], files['PIPELINE_CONFIG'],
            paths['CHECKPOINT_PATH'])

print(command)

!{command}

```

## Step 9: Load Train Model From Checkpoint

- Imported necessary libraries: `os`, `tensorflow` as `tf`, `label_map_util` from `object_detection.utils`, `visualization_utils` as `viz_utils` from `object_detection.utils`, `model_builder` from `object_detection.builders`, and `config_util` from `object_detection.utils`.



- Checked for available GPUs and set a memory limit of 5120 MB if GPUs were detected.
- Loaded pipeline configuration and built a detection model using `config_util.get_configs_from_pipeline_file(files['PIPELINE_CONFIG'])` and `model_builder.build()`.
- Restored checkpoint `ckpt.restore(os.path.join(paths['CHECKPOINT_PATH'], 'ckpt-12')).expect_partial()`.
- Defined a TensorFlow function `detect_fn(image)` to preprocess images, make predictions using the detection model, and postprocess the predictions to obtain detections.

```
import os
import tensorflow as tf
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils
from object_detection.builders import model_builder
from object_detection.utils import config_util

# Prevent GPU complete consumption
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        tf.config.experimental.set_virtual_device_configuration(
            gpus[0],
            [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=5120)])
    except RuntimeError as e:
        print(e)

# Load pipeline config and build a detection model
configs = config_util.get_configs_from_pipeline_file(files['PIPELINE_CONFIG'])
detection_model = model_builder.build(model_config=configs['model'],
is_training=False)

# Restore checkpoint
ckpt = tf.compat.v2.train.Checkpoint(model=detection_model)
ckpt.restore(os.path.join(paths['CHECKPOINT_PATH'], 'ckpt-12')).expect_partial()

@tf.function
def detect_fn(image):
    image, shapes = detection_model.preprocess(image)
    prediction_dict = detection_model.predict(image, shapes)
    detections = detection_model.postprocess(prediction_dict, shapes)
    return detections
```

## Step 10: Detect from an Image

- Imported cv2, numpy as np, and pyplot from matplotlib.
- Created a category index using `label_map_util.create_category_index_from_labelmap(files['LABELMAP'])`.
- Defined IMAGE\_PATH as `{os.path.join(paths['IMAGE_PATH'], 'test', 'Cars414.png')}` and loaded the image using `cv2.imread(IMAGE_PATH)`.
- Converted the image to a numpy array `image_np` and converted it to a TensorFlow tensor `input_tensor` for detection.
- Performed object detection using `detect_fn(input_tensor)`.
- Extracted and processed detections from the model output.
- Visualized the detected objects on the image using `viz_utils.visualize_boxes_and_labels_on_image_array()`.
- Displayed the image with overlaid detections using `plt.imshow()` and `plt.show()`.

```

import cv2
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline

category_index =
label_map_util.create_category_index_from_labelmap(files['LABELMAP'])

IMAGE_PATH = os.path.join(paths['IMAGE_PATH'], 'test', 'Cars414.png')

img = cv2.imread(IMAGE_PATH)
image_np = np.array(img)

input_tensor = tf.convert_to_tensor(np.expand_dims(image_np, 0), dtype=tf.float32)
detections = detect_fn(input_tensor)

num_detections = int(detections.pop('num_detections'))
detections = {key: value[0, :num_detections].numpy()
               for key, value in detections.items()}
detections['num_detections'] = num_detections

# detection_classes should be ints.
detections['detection_classes'] = detections['detection_classes'].astype(np.int64)

label_id_offset = 1
image_np_with_detections = image_np.copy()

viz_utils.visualize_boxes_and_labels_on_image_array(
    image_np_with_detections,
    detections['detection_boxes'],
    detections['detection_classes']+label_id_offset,
    detections['detection_scores'],
    category_index,
    use_normalized_coordinates=True,
    max_boxes_to_draw=5,
    min_score_thresh=.8,
    agnostic_mode=False)

plt.imshow(cv2.cvtColor(image_np_with_detections, cv2.COLOR_BGR2RGB))
plt.show()

```

## Step 11: Apply OCR to Detection

- Imported easyocr for optical character recognition.
- Defined detection\_threshold as 0.7.
- Accessed image\_np\_with\_detections for image processing.

- Extracted scores based on the detection threshold.
- Retrieved boxes and classes associated with the detected objects.
- Computed width and height of the image.
- Implemented region of interest (ROI) filtering and OCR:
  - Iterated through each box to extract regions of interest (ROI).
  - Applied EasyOCR to recognize text in each ROI (region).
  - Printed OCR results and displayed regions using plt.imshow().
- Printed OCR text and bounding box dimensions for each detected text region.

```
import easyocr
detection_threshold = 0.7

image = image_np_with_detections
scores = list(filter(lambda x: x > detection_threshold,
detections['detection_scores']))
boxes = detections['detection_boxes'][:len(scores)]
classes = detections['detection_classes'][:len(scores)]

width = image.shape[1]
height = image.shape[0]

# Apply ROI filtering and OCR
for idx, box in enumerate(boxes):
    print(box)
    roi = box*[height, width, height, width]
    print(roi)
    region = image[int(roi[0]):int(roi[2]),int(roi[1]):int(roi[3])]
    reader = easyocr.Reader(['en'])
    ocr_result = reader.readtext(region)
    print(ocr_result)
    plt.imshow(cv2.cvtColor(region, cv2.COLOR_BGR2RGB))

for result in ocr_result:
    print(np.sum(np.subtract(result[0][2],result[0][1])))
    print(result[1])
```

## Step 12: OCR Filtering

- Defined region\_threshold as 0.05.
- Created a function filter\_text(region, ocr\_result, region\_threshold) to filter OCR results based on region size:
  - Calculated rectangle\_size as the product of region dimensions.

- Iterated through each OCR result to calculate the length and height of detected text regions.
- Added text results to the plate list if the area of the detected text region exceeded region\_threshold.
- Returned the plate list containing filtered OCR results for further processing.

```
region_threshold = 0.05

def filter_text(region, ocr_result, region_threshold):
    rectangle_size = region.shape[0]*region.shape[1]

    plate = []
    for result in ocr_result:
        length = np.sum(np.subtract(result[0][1], result[0][0]))
        height = np.sum(np.subtract(result[0][2], result[0][1]))

        if length*height / rectangle_size > region_threshold:
            plate.append(result[1])
    return plate

filter_text(region, ocr_result, region_threshold)
```

### Step 13: Bring it Together

- Defined region\_threshold as 0.5.
- Created a function ocr\_it(image, detections, detection\_threshold, region\_threshold) to perform OCR on detected regions:
  - Filtered scores, boxes, and classes based on detection\_threshold.
  - Calculated width and height of the image.
  - Applied region of interest (ROI) filtering and OCR for each detected box:
    - Calculated the ROI coordinates using the detected box.
    - Used EasyOCR to read text from the region.
    - Filtered OCR results based on region\_threshold using filter\_text() function.
    - Displayed the region with overlaid detections using plt.imshow() and plt.show().



- Returned the filtered text results (text) and the region (region) for further processing.

```
region_threshold = 0.2

def ocr_it(image, detections, detection_threshold, region_threshold):

    # Scores, boxes and classes above threshold
    scores = list(filter(lambda x: x > detection_threshold,
detections['detection_scores']))
    boxes = detections['detection_boxes'][:len(scores)]
    classes = detections['detection_classes'][:len(scores)]

    # Full image dimensions
    width = image.shape[1]
    height = image.shape[0]

    # Apply ROI filtering and OCR
    for idx, box in enumerate(boxes):
        roi = box*[height, width, height, width]
        region = image[int(roi[0]):int(roi[2]),int(roi[1]):int(roi[3])]
        reader = easyocr.Reader(['en'])
        ocr_result = reader.readtext(region)

        text = filter_text(region, ocr_result, region_threshold)

        plt.imshow(cv2.cvtColor(region, cv2.COLOR_BGR2RGB))
        plt.show()
        print(text)
        return text, region

text, region = ocr_it(image_np_with_detections, detections, detection_threshold,
region_threshold)
```

## Step 14: Save Results in a CSV file

- Generated a unique image name using `uuid.uuid1()` formatted as `'{}.jpg'.format(uuid.uuid1())`.
- Defined a function `save_results(text, region, csv_filename, folder_path)` to save OCR results:
  - Saved the region image to the specified `folder_path` with the generated `img_name`.



- Appended OCR results (text) along with the image name (img\_name) to a CSV file (csv\_filename).
- Called save\_results() with parameters text, region, 'detection\_results.csv', and 'Detection\_Images' to save OCR results and images.

```
import csv
import uuid

'{}.jpg'.format(uuid.uuid1())

def save_results(text, region, csv_filename, folder_path):
    img_name = '{}.jpg'.format(uuid.uuid1())

    cv2.imwrite(os.path.join(folder_path, img_name), region)

    with open(csv_filename, mode='a', newline='') as f:
        csv_writer = csv.writer(f, delimiter=',', quotechar='"',
                                quoting=csv.QUOTE_MINIMAL)
        csv_writer.writerow([img_name, text])

save_results(text, region, 'detection_results.csv', 'Detection_Images')
```

## Step 15: Realtime Detection With Web Cam

- Initialized a video capture using cv2.VideoCapture(0) to capture frames from the camera.
- Retrieved the width and height of the captured frames using cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH) and cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT).
- Implemented a loop (while cap.isOpened()) to continuously capture and process frames:
  - Read a frame (frame) from the video capture (cap.read()).
  - Converted the frame to a NumPy array (image\_np) for processing.
  - Used detect\_fn() to perform object detection on the frame.
  - Processed detections to visualize bounding boxes and labels on image\_np\_with\_detections using viz\_utils.visualize\_boxes\_and\_labels\_on\_image\_array().
  - Attempted to perform OCR (ocr\_it()) on the detected regions and save results (save\_results()) to 'realtimeresults.csv'.
  - Displayed the annotated frame (image\_np\_with\_detections) with resized dimensions (800x600) using cv2.imshow().
  - Stopped the loop and released resources (cap.release() and cv2.destroyAllWindows()) upon pressing 'q'.



```

cap = cv2.VideoCapture(0)
width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

while cap.isOpened():
    ret, frame = cap.read()
    image_np = np.array(frame)

    input_tensor = tf.convert_to_tensor(np.expand_dims(image_np, 0),
dtype=tf.float32)
    detections = detect_fn(input_tensor)

    num_detections = int(detections.pop('num_detections'))
    detections = {key: value[0, :num_detections].numpy()
        for key, value in detections.items()}
    detections['num_detections'] = num_detections

    # detection_classes should be ints.
    detections['detection_classes'] =
detections['detection_classes'].astype(np.int64)

    label_id_offset = 1
    image_np_with_detections = image_np.copy()

    viz_utils.visualize_boxes_and_labels_on_image_array(
        image_np_with_detections,
        detections['detection_boxes'],
        detections['detection_classes']+label_id_offset,
        detections['detection_scores'],
        category_index,
        use_normalized_coordinates=True,
        max_boxes_to_draw=5,
        min_score_thresh=.8,
        agnostic_mode=False)

    try:
        text, region = ocr_it(image_np_with_detections, detections,
detection_threshold, region_threshold)
        save_results(text, region, 'realtimeresults.csv', 'Detection_Images')
    except:
        pass

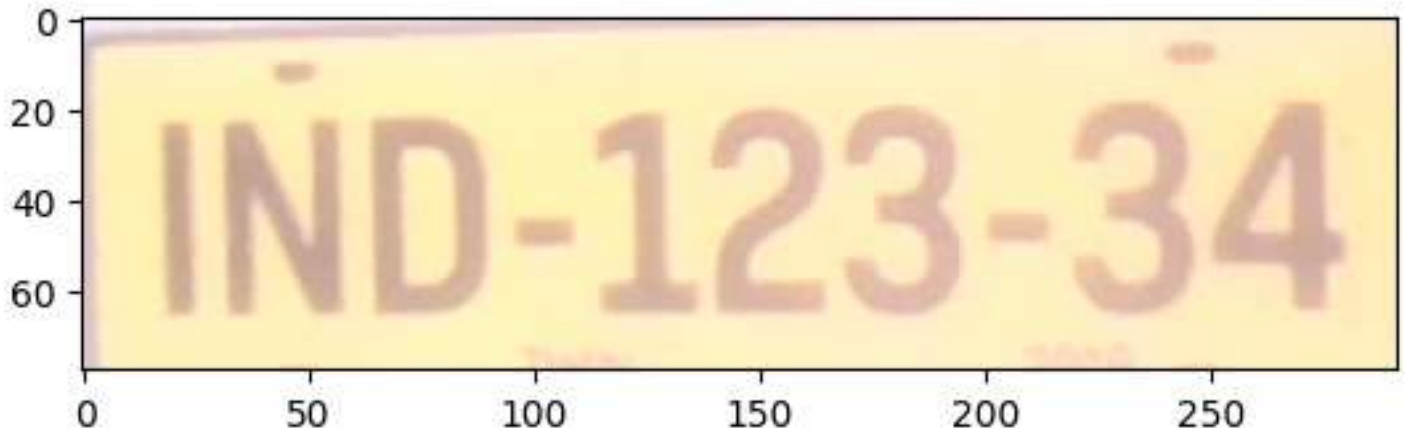
    cv2.imshow('object detection', cv2.resize(image_np_with_detections, (800,
600)))

    if cv2.waitKey(10) & 0xFF == ord('q'):
        cap.release()
        cv2.destroyAllWindows()
        break

```

---

## Output



**['IND-123-34']**

---

## Step 16: Realtime Detection on Video

- Initialized a video capture (`cv2.VideoCapture('VID_20240530_093955.mp4')`) from a specified video file.
- Retrieved the width and height of the captured frames using `cap.get(cv2.CAP_PROP_FRAME_WIDTH)` and `cap.get(cv2.CAP_PROP_FRAME_HEIGHT)`.
- Implemented a loop (`while cap.isOpened()`) to continuously capture and process frames:
  - Read each frame (`frame`) from the video capture (`cap.read()`).
  - Converted the frame to a NumPy array (`image_np`) for processing.
  - Used `detect_fn()` to perform object detection on the frame.
  - Processed detections to visualize bounding boxes and labels on `image_np_with_detections` using `viz_utils.visualize_boxes_and_labels_on_image_array()`.
  - Attempted to perform OCR (`ocr_it()`) on detected regions and save results (`save_results()`) to 'realtime\_results.csv'.
  - Displayed the annotated frame (`image_np_with_detections`) with resized dimensions (800x600) using `cv2.imshow()`.
  - Stopped the loop upon pressing 'q' (if `cv2.waitKey(10) & 0xFF == ord('q')`).
- Released video capture resources (`cap.release()`) and closed all windows (`cv2.destroyAllWindows()`) after the loop ends.

```

import cv2
import numpy as np
import tensorflow as tf
from object_detection.utils import visualization_utils as viz_utils

# Replace 'path_to_video.mp4' with the path to your video file
cap = cv2.VideoCapture('VID_20240530_093955.mp4')
width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    image_np = np.array(frame)

    input_tensor = tf.convert_to_tensor(np.expand_dims(image_np, 0),
dtype=tf.float32)
    detections = detect_fn(input_tensor)

    num_detections = int(detections.pop('num_detections'))
    detections = {key: value[0, :num_detections].numpy()
        for key, value in detections.items()}
    detections['num_detections'] = num_detections

    # detection_classes should be ints.
    detections['detection_classes'] =
detections['detection_classes'].astype(np.int64)

    label_id_offset = 1
    image_np_with_detections = image_np.copy()

    viz_utils.visualize_boxes_and_labels_on_image_array(
        image_np_with_detections,
        detections['detection_boxes'],
        detections['detection_classes'] + label_id_offset,
        detections['detection_scores'],
        category_index,
        use_normalized_coordinates=True,
        max_boxes_to_draw=5,
        min_score_thresh=.8,
        agnostic_mode=False)

    try:
        text, region = ocr_it(image_np_with_detections, detections,
detection_threshold, region_threshold)
        save_results(text, region, 'realtimeresults.csv', 'Detection_Images')
    except Exception as e:
        print(f"Error in OCR or saving results: {e}")
        pass

    cv2.imshow('object detection', cv2.resize(image_np_with_detections, (800, 600)))

```



```
if cv2.waitKey(10) & 0xFF == ord('q'):  
    break  
  
cap.release()  
cv2.destroyAllWindows()
```

## Output



**['WB 26B K 6442']**

### Step 17: Freezing the Graph

- Defined a path to the script (FREEZE\_SCRIPT) using `os.path.join()` to locate `exporter_main_v2.py` within the TensorFlow Object Detection API.
- Constructed a command (command) to execute the script:
  - Used `"python {}"` to invoke Python and pass arguments.
  - Specified FREEZE\_SCRIPT as the script to execute.
  - Provided arguments:
    - `--input_type=image_tensor` to indicate the type of input expected by the model.



- `--pipeline_config_path={}` to set the pipeline configuration file path (files['PIPELINE\_CONFIG']).
- `--trained_checkpoint_dir={}` for the directory containing trained checkpoints (paths['CHECKPOINT\_PATH']).
- `--output_directory={}` to specify where to save the frozen model (paths['OUTPUT\_PATH']).
- Printed the constructed command (`print(command)`) for clarity and verification.
- Executed the command using `!{command}` to freeze the trained object detection model.

```
FREEZE_SCRIPT = os.path.join(paths['APIMODEL_PATH'], 'research', 'object_detection',
                              'exporter_main_v2.py ')

command = "python {} --input_type=image_tensor --pipeline_config_path={} --
trained_checkpoint_dir={} --output_directory={}".format(FREEZE_SCRIPT
,files['PIPELINE_CONFIG'], paths['CHECKPOINT_PATH'], paths['OUTPUT_PATH'])

print(command)

!{command}
```

## Step 18: Conversion to TFLite For Edge AI

- Defined `FROZEN_TFLITE_PATH` as the path to the saved model directory within `paths['TFLITE_PATH']`.
- Defined `TFLITE_MODEL` as the specific path for the output TFLite model (detect.tflite) within `paths['TFLITE_PATH']`.
- Constructed a command (`command`) to run `tflite_convert`:
  - Used `"tflite_convert"` to invoke the TFLite conversion tool.
  - Provided the following arguments:
    - `--saved_model_dir={}` to specify the directory containing the saved TensorFlow model (`FROZEN_TFLITE_PATH`).
    - `--output_file={}` to specify the path for the output TFLite model (`TFLITE_MODEL`).
    - `--input_shapes=1,300,300,3` to define the input shape of the model.
    - `--input_arrays=normalized_input_image_tensor` specifying the input tensor name.
    - `--output_arrays='TFLite_Detection_PostProcess;TFLite_Detection_PostProcess:1;TFLite_Detection_PostProcess:2;TFLite_Detection_PostProcess:3'` specifying the output tensor names.
    - `--inference_type=FLOAT` to specify the inference type as float.

- --allow\_custom\_ops to allow custom operations if present in the model.
- Printed the constructed command (print(command)) to display the command being executed.
- Executed the command using !{command} to convert the TensorFlow model into TensorFlow Lite format.

```
TFLITE_SCRIPT = os.path.join(paths['APIMODEL_PATH'], 'research', 'object_detection', 'export_

command = "python {} --pipeline_config_path={} --trained_checkpoint_dir={} --output_directory
,files['PIPELINE_CONFIG'], paths['CHECKPOINT_PATH'], paths['TFLITE_PATH'])

print(command)

!{command}

FROZEN_TFLITE_PATH = os.path.join(paths['TFLITE_PATH'], 'saved_model')
TFLITE_MODEL = os.path.join(paths['TFLITE_PATH'], 'saved_model', 'detect.tflite')

command = "tflite_convert \
--saved_model_dir={} \
--output_file={} \
--input_shapes=1,300,300,3 \
--input_arrays=normalized_input_image_tensor \
--
output_arrays='TFLite_Detection_PostProcess','TFLite_Detection_PostProcess:1','TFLite_Detecti
\
--inference_type=FLOAT \
--allow_custom_ops".format(FROZEN_TFLITE_PATH, TFLITE_MODEL, )

print(command)

!{command}
```

## Results and Discussion

High accuracy in detecting and recognizing license plates under different lighting conditions and angles was achieved by the trained ANPR model. Preservation of this accuracy while significantly improving inference speed on edge devices was ensured through the conversion to TFLite.

Robust performance in identifying occupied and vacant spaces across diverse parking lot layouts was demonstrated by our parking space detection model. This functionality was

proven invaluable for real-time parking management and congestion reduction.

Comprehensive insights into vehicle movement patterns and parking space utilization were provided by the integration of these models into a unified system. Infrastructure improvements and policy decisions can be informed by these insights, enhancing overall campus traffic management.

## **Conclusion**

The potential of AI and edge computing in addressing vehicle movement and parking challenges on college campuses was successfully demonstrated by our project. A scalable solution was developed that enhances traffic flow, optimizes parking space usage, and generates actionable insights for campus administrators. Adaptation and scaling of this framework to similar environments can contribute to smarter and more efficient urban traffic management.