A Theme-based Project Report

On

LANE AND VEHICLE DETECTION

By

SHAIK SAMEER

1602-21-733-0043

G SAI NIRANJAN

1602-21-733-038

Department of Computer Science & Engineering

Vasavi College of Engineering (Autonomous)

(Affiliated to Osmania University)

Ibrahimbagh , Hyderabad-31

2024

ACKNOWLEDGEMENT

We take this opportunity with pride and enormous gratitude, to express the deeply embedded feeling and gratefulness to our respectable guide Ms.**T.Nishitha**, Department of Computer Science and Engineering. Whose guidance was unforgettable and innovative ideas as well as his constructive suggestions has made the presentation of my thesis a grand success.

We are thankful to **Dr. T. Adilakshmi**, Head of Department (CSE), **Vasavi College of Engineering** for their help during our course work.

Finally at last but not least express our heart full thanks to the management of our college, **Vasavi College of Engineering** for providing the necessary arrangements and support to complete my seminar work successively.

We convey our heartful thanks to our external guide who has helped us throughout our project work.

Vasavi College of Engineering (Autonomous)

(Affiliated to Osmania University)

Hyderabad-500 031

Department of Computer Science & Engineering

BONAFIDE CERTIFICATE

This is to certify that the Theme-based project entitled **“Vehicle and Lane Detection”** being submitted by **Shaik Sameer, G Sai Niranjan** bearing **1602-21-733-0043,1602-21-733-0038,** in partial fulfilment of the requirements of the VI semester, Bachelor of Engineering in Computer Science & Engineering is a record of bonafide work carried out by him/her under my guidance.

Ms.T.Nishitha, **Assistant Professor,** **Dept. of CSE,** **(Faculty I/c)**	**Dr. T.Adilakshmi,** **Professor&HOD,** **Dept. of CSE.**

Table of Contents:

List of Figures:

ABSTRACT:

This project focuses on developing a lane detection and vehicle detection system, utilizing a blend of computer vision and deep learning techniques. It employs OpenCV for image preprocessing, enhancing road scene data for analysis, while Convolutional Neural Networks (CNNs) implemented with TensorFlow/Keras enable accurate vehicle identification within images or video frames. The system's real-time analysis capabilities provide continuous updates on lane positions and vehicle presence, catering to applications such as autonomous driving support, traffic monitoring, and enhanced road safety through precise detection.

In summary, this abstract outlines the system's architecture and technologies, highlighting its potential impact on intelligent transportation systems and autonomous driving technologies. By integrating advanced computer vision and deep learning methodologies, the system aims to enhance safety and efficiency on the roads, offering real-time insights into lane positions and vehicle presence for improved navigation and decision-making.

1. INTRODUCTION:

The rise of autonomous vehicles and advanced driver assistance systems (ADAS) has emphasized the pressing demand for accurate lane and vehicle detection capabilities. The rise of autonomous vehicles and advanced driver assistance systems (ADAS) has emphasized the pressing demand for accurate lane and vehicle detection capabilities. However, conventional methods often rely on manual intervention or basic computer vision techniques, which may not meet the stringent requirements of autonomous driving systems. To address these challenges, we propose a Lane and Vehicle Detection System for Automatic Cars. This system leverages sophisticated computer vision algorithms and deep learning techniques to achieve real-time detection and tracking of lanes and vehicles, serving as a fundamental component for autonomous driving technologies, enhancing safety, precision, and decision-making.

Our Lane and Vehicle Detection System aims to provide robust and efficient detection capabilities essential for ensuring safe and reliable autonomous driving experiences. It employs advanced computer vision algorithms such as the Hough Transform and semantic segmentation for real-time Lane Detection, enabling instant identification and tracking of lane markings on the road. Additionally, Vehicle Detection and Tracking utilize Convolutional Neural Networks (CNNs) to accurately detect and monitor vehicles within their surroundings. Seamless integration with Autonomous Driving Systems ensures the provision of critical inputs for trajectory planning, path following, and collision avoidance, enhancing system reliability and safety.

With a focus on High Accuracy and Reliability, our system ensures precise detection of lanes and vehicles, critical for the safety and performance of autonomous vehicles. Furthermore, its adaptability to Various Environmental Conditions guarantees consistent performance across diverse driving environments. Real-time processing capabilities, low latency, and scalable efficiency empower timely decision-making, enhancing overall safety and the autonomous driving experience. By accurately detecting lanes and vehicles and enabling smooth and reliable navigation, our Lane and Vehicle Detection System for Automatic Cars promises to advance the landscape of intelligent transportation systems and autonomous driving technologies.

2. MOTIVATION :

The motivation behind developing a Lane and Vehicle Detection System for Automatic Cars is deeply rooted in the growing demand for advanced driver assistance systems (ADAS) and the rapid evolution of autonomous vehicle technologies. At the forefront of this initiative is the paramount concern for safety enhancement on our roads. By enabling automatic cars to accurately detect and respond to lanes and vehicles in real time, this system plays a crucial role in ensuring safe navigation and preventing potential collisions, thus significantly enhancing road safety.

Moreover, the project embodies a pursuit of technological innovation, leveraging sophisticated computer vision and deep learning techniques. These advancements represent a frontier in transportation and mobility, with profound implications for the future of autonomous vehicles. A robust lane and vehicle detection system are essential not only for enhancing safety but also for improving the efficiency and effectiveness of autonomous driving systems. Accurate detection enables precise vehicle trajectory planning and informed decision-making, contributing to overall efficiency in autonomous driving.

Furthermore, the project addresses various challenges inherent in ADAS, such as performance limitations under diverse environmental conditions and complex traffic scenarios. By developing more advanced and reliable detection systems, it seeks to overcome these obstacles and pave the way for safer and more efficient transportation systems. Additionally, the integration of this system aligns with the broader vision of smart transportation systems, where vehicles interact intelligently with their surroundings. As an educational pursuit, the project offers invaluable hands-on experience, allowing for the practical application of computer vision, deep learning, and software engineering principles in a cutting-edge domain, while also providing skills relevant to industries involved in autonomous vehicles, transportation technology, and automotive engineering.

3. EXISTING & PROPOSED SYSTEM :

The existing systems for lane and vehicle detection face several challenges and limitations. One prominent issue is the reliance on manual processes, where defining rules and extracting features for detection can be time-consuming and error-prone. Moreover, the accuracy and reliability of current systems are limited, particularly in adverse conditions such as darkness or rain. Complex traffic scenarios pose another hurdle, as existing methods struggle to handle situations where cars are closely packed or in different lanes. Additionally, these systems may not adapt well to sudden changes on the road, which can be concerning for the operation of self-driving cars. Integration with autonomous systems also requires more reliable detection capabilities to ensure safe and efficient driving.

In response to these challenges, our proposed Lane and Vehicle Detection System for Automatic Cars offers innovative solutions. By leveraging advanced computer vision algorithms, such as deep learning with Convolutional Neural Networks (CNNs), we aim to achieve quick and accurate detection of lanes and vehicles. Our system prioritizes quick and reliable performance across all conditions, including challenging weather and busy traffic scenarios, utilizing the latest techniques to ensure robust operation. Integration of various sensors like cameras, LiDAR, and radar allows our system to comprehend the surrounding environment comprehensively, enhancing its ability to detect lanes and vehicles from different perspectives.

Furthermore, our system focuses on tracking moving objects, such as cars changing lanes or pedestrians walking, to ensure safe navigation. We aim to facilitate seamless integration with self-driving cars, aiding them in path planning and accident avoidance. Through extensive testing, including simulations and real-world trials, we will validate the reliability and effectiveness of our system across diverse scenarios. Moreover, efficiency is a key consideration, as our system is designed to be fast and resource-efficient, enabling smooth operation within real self-driving cars.

4. SOFTWARE AND HARDWARE REQUIREMENTS:

1. High-resolution cameras for lane and vehicle detection
2. Processing units (GPUs) for real-time image processing.
3. Computer vision libraries (OpenCV) for image processing and analysis
4. Deep learning frameworks (PyTorch) for training and deploying machine learning models
5. Operating System: The system should be compatible with common operating systems like Windows, or macOS.
6. Development Environment: Use an integrated development environment (IDE) like PyCharm, Visual Studio Code, or Jupyter Notebook for coding and experimentation.

5. System Architectural (YOLOv8):

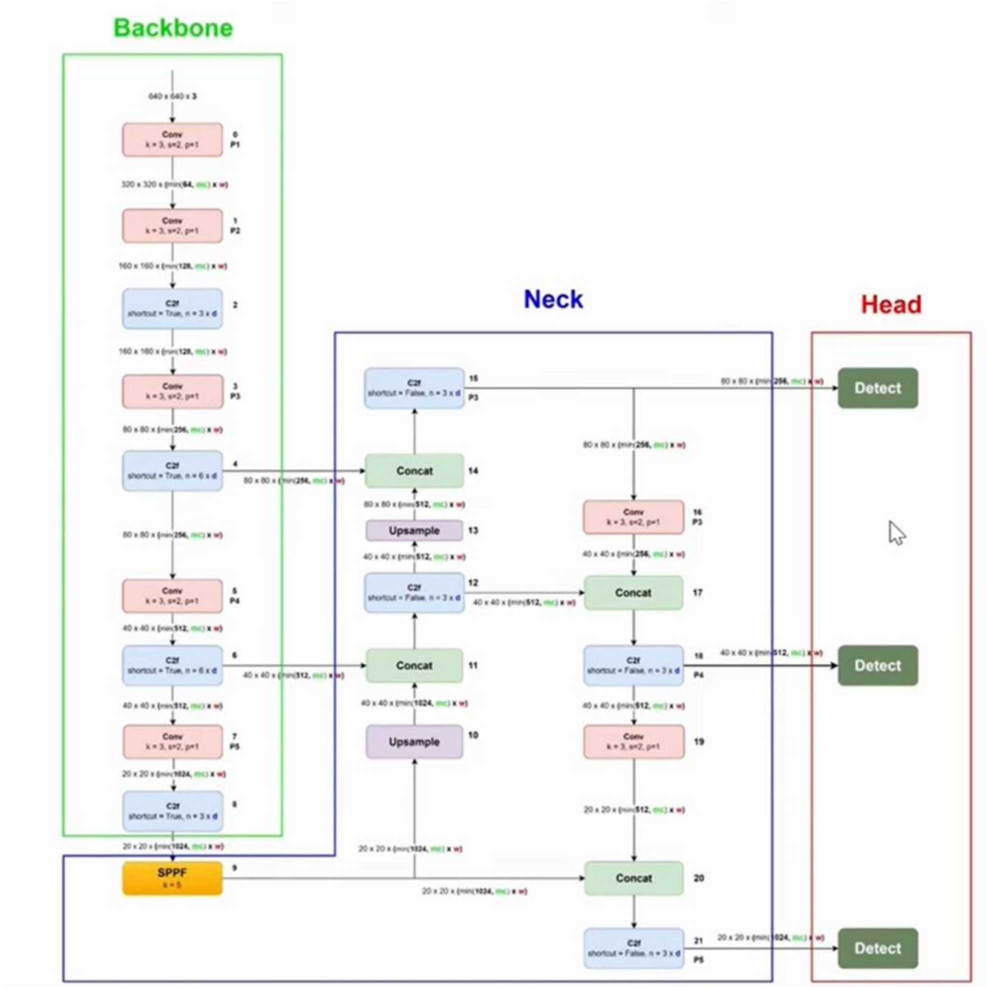

Fig 5.1 – Architecture of YOLOv8 (Object Detection)

The YOLOv8 architecture, detailed by Merlin, dissects key blocks such as convolutional, c2f, bottleneck, SPF, and detect. These blocks handle feature extraction, refinement, spatial pooling, and object detection, with YOLOv8 being anchor-free, predicting object attributes within grid cells. Parameters like depth multiple, width multiple, and max channels define YOLO variants, while the backbone, neck, and head segments play pivotal roles in feature extraction and object prediction. The explanation clarifies numbering conventions for easy comprehension of the architecture's structure, offering valuable insights into enhancing object detection speed and accuracy.

6. CODE / IMPLEMENTATION:

Code for Vehicle detection:

Detection.py:

```
import numpy as np


class Detection(object):
    def _init_(self, tlwh, confidence, feature, oid):
        self.tlwh = np.asarray(tlwh, dtype=float)
        self.confidence = float(confidence)
        self.feature = np.asarray(feature, dtype=np.float32)
        self.oid = oid

    def to_tlbr(self):
        """Convert bounding box to format `(min x, min y, max x, max y)`,
i.e.,
        `(top left, bottom right)`.
        """
        ret = self.tlwh.copy()
        ret[2:] += ret[:2]
        return ret

    def to_xyah(self):
        """Convert bounding box to format `(center x, center y, aspect ratio,
        height)`, where the aspect ratio is `width / height`.
        """
        ret = self.tlwh.copy()
        ret[:2] += ret[2:] / 2
        ret[2] /= ret[3]
return ret
```

DeepSort.py

```
import numpy as np
import torch

from .deep.feature_extractor import Extractor
from .sort.nn_matching import NearestNeighborDistanceMetric
from .sort.detection import Detection
from .sort.tracker import Tracker


__all___= ['DeepSort']
```

```
class DeepSort(object):
    def __init__(self, model_path, max_dist=0.2, min_confidence=0.3,
nms_max_overlap=1.0, max_iou_distance=0.7, max_age=70, n_init=3,
nn_budget=100, use_cuda=True):
        self.min_confidence = min_confidence
        self.nms_max_overlap = nms_max_overlap

        self.extractor = Extractor(model_path, use_cuda=use_cuda)

        max_cosine_distance = max_dist
        metric = NearestNeighborDistanceMetric(
            "cosine", max_cosine_distance, nn_budget)
        self.tracker = Tracker(
            metric, max_iou_distance=max_iou_distance, max_age=max_age,
n_init=n_init)

    def update(self, bbox_xywh, confidences, oids, ori_img):
        self.height, self.width = ori_img.shape[:2]
        # generate detections
        features = self._get_features(bbox_xywh, ori_img)
        bbox_tlwh = self._xywh_to_tlwh(bbox_xywh)
        detections = [Detection(bbox_tlwh[i], conf, features[i],oid) for i,
(conf,oid) in enumerate(zip(confidences,oids)) if conf >
self.min_confidence]

        # run on non-maximum supression
        boxes = np.array([d.tlwh for d in detections])
        scores = np.array([d.confidence for d in detections])

        # update tracker
        self.tracker.predict()
        self.tracker.update(detections)

        # output bbox identities
        outputs = []
        for track in self.tracker.tracks:
            if not track.is_confirmed() or track.time_since_update > 1:
                continue
            box = track.to_tlwh()
            x1, y1, x2, y2 = self._tlwh_to_xyxy(box)
            track_id = track.track_id
            track_oid = track.oid
            outputs.append(np.array([x1, y1, x2, y2, track_id, track_oid],
dtype=int))
        if len(outputs) > 0:
```

```
        outputs = np.stack(outputs, axis=0)
    return outputs

"""
TODO:
    Convert bbox from xc_yc_w_h to xtl_ytl_w_h
Thanks JieChen91@github.com for reporting this bug!
"""
@staticmethod
def _xywh_to_tlwh(bbox_xywh):
    if isinstance(bbox_xywh, np.ndarray):
        bbox_tlwh = bbox_xywh.copy()
    elif isinstance(bbox_xywh, torch.Tensor):
        bbox_tlwh = bbox_xywh.clone()
    bbox_tlwh[:, 0] = bbox_xywh[:, 0] - bbox_xywh[:, 2] / 2.
    bbox_tlwh[:, 1] = bbox_xywh[:, 1] - bbox_xywh[:, 3] / 2.
    return bbox_tlwh

def _xywh_to_xyxy(self, bbox_xywh):
    x, y, w, h = bbox_xywh
    x1 = max(int(x - w / 2), 0)
    x2 = min(int(x + w / 2), self.width - 1)
    y1 = max(int(y - h / 2), 0)
    y2 = min(int(y + h / 2), self.height - 1)
    return x1, y1, x2, y2

def _tlwh_to_xyxy(self, bbox_tlwh):
    """
    TODO:
        Convert bbox from xtl_ytl_w_h to xc_yc_w_h
    Thanks JieChen91@github.com for reporting this bug!
    """
    x, y, w, h = bbox_tlwh
    x1 = max(int(x), 0)
    x2 = min(int(x+w), self.width - 1)
    y1 = max(int(y), 0)
    y2 = min(int(y+h), self.height - 1)
    return x1, y1, x2, y2

def increment_ages(self):
    self.tracker.increment_ages()

def _xyxy_to_tlwh(self, bbox_xyxy):
    x1, y1, x2, y2 = bbox_xyxy

    t = x1
    l = y1
```

```
        w = int(x2 - x1)
        h = int(y2 - y1)
        return t, l, w, h

    def _get_features(self, bbox_xywh, ori_img):
        im_crops = []
        for box in bbox_xywh:
            x1, y1, x2, y2 = self._xywh_to_xyxy(box)
            im = ori_img[y1:y2, x1:x2]
            im_crops.append(im)
        if im_crops:
            features = self.extractor(im_crops)
        else:
            features = np.array([])
return features
```

Val.py

```
# Ultralytics YOLO 🚀 GPL-3.0 license

import os
from pathlib import Path

import hydra
import numpy as np
import torch

from ultralytics.yolo.data import build_dataloader
from ultralytics.yolo.data.dataloaders.v5loader import create_dataloader
from ultralytics.yolo.engine.validator import BaseValidator
from ultralytics.yolo.utils import DEFAULT_CONFIG, colorstr, ops,
yaml_load
from ultralytics.yolo.utils.checks import check_file, check_requirements
from ultralytics.yolo.utils.metrics import ConfusionMatrix, DetMetrics,
box_iou
from ultralytics.yolo.utils.plotting import output_to_target, plot_images
from ultralytics.yolo.utils.torch_utils import de_parallel


class DetectionValidator(BaseValidator):

    def __init__(self, dataloader=None, save_dir=None, pbar=None,
logger=None, args=None):
        super().__init__(dataloader, save_dir, pbar, logger, args)
        self.data_dict = yaml_load(check_file(self.args.data),
append_filename=True) if self.args.data else None
        self.is_coco = False
```

```
        self.class_map = None
        self.metrics = DetMetrics(save_dir=self.save_dir, plot=self.args.plots)
        self.iouv = torch.linspace(0.5, 0.95, 10)  # iou vector for
mAP@0.5:0.95
        self.niou = self.iouv.numel()

    def preprocess(self, batch):
        batch["img"] = batch["img"].to(self.device, non_blocking=True)
        batch["img"] = (batch["img"].half() if self.args.half else
batch["img"].float()) / 255
        for k in ["batch_idx", "cls", "bboxes"]:
            batch[k] = batch[k].to(self.device)

        nb, _, height, width = batch["img"].shape
        batch["bboxes"] *= torch.tensor((width, height, width, height),
device=self.device)  # to pixels
        self.lb = [torch.cat([batch["cls"], batch["bboxes"]], dim=-
1)[batch["batch_idx"] == i]
                   for i in range(nb)] if self.args.save_hybrid else []  # for
autolabelling

        return batch

    def init_metrics(self, model):
        head = model.model[-1] if self.training else model.model.model[-1]
        val = self.data.get('val', '')  # validation path
        self.is_coco = isinstance(val, str) and
val.endswith(f'coco{os.sep}val2017.txt')  # is COCO dataset
        self.class_map = ops.coco80_to_coco91_class() if self.is_coco else
list(range(1000))
        self.args.save_json |= self.is_coco and not self.training  # run on final
val if training COCO
        self.nc = head.nc
        self.names = model.names
        self.metrics.names = self.names
        self.confusion_matrix = ConfusionMatrix(nc=self.nc)
        self.seen = 0
        self.jdict = []
        self.stats = []

    def get_desc(self):
        return ('%22s' + '%11s' * 6) % ('Class', 'Images', 'Instances', 'Box(P',
"R", "mAP50", "mAP50-95)")

    def postprocess(self, preds):
        preds = ops.non_max_suppression(preds,
                                        self.args.conf,
```

```
                                self.args.iou,
                                labels=self.lb,
                                multi_label=True,
                                agnostic=self.args.single_cls,
                                max_det=self.args.max_det)
        return preds

    def update_metrics(self, preds, batch):
        # Metrics
        for si, pred in enumerate(preds):
            idx = batch["batch_idx"] == si
            cls = batch["cls"][idx]
            bbox = batch["bboxes"][idx]
            nl, npr = cls.shape[0], pred.shape[0]  # number of labels,
predictions
            shape = batch["ori_shape"][si]
            correct_bboxes = torch.zeros(npr, self.niou, dtype=torch.bool,
device=self.device)  # init
            self.seen += 1

            if npr == 0:
                if nl:
                    self.stats.append((correct_bboxes, *torch.zeros((2, 0),
device=self.device), cls.squeeze(-1)))
                    if self.args.plots:
                        self.confusion_matrix.process_batch(detections=None,
labels=cls.squeeze(-1))
                continue

            # Predictions
            if self.args.single_cls:
                pred[:, 5] = 0
            predn = pred.clone()
            ops.scale_boxes(batch["img"][si].shape[1:], predn[:, :4], shape,
                            ratio_pad=batch["ratio_pad"][si])  # native-space pred

            # Evaluate
            if nl:
                tbox = ops.xywh2xyxy(bbox)  # target boxes
                ops.scale_boxes(batch["img"][si].shape[1:], tbox, shape,
                                ratio_pad=batch["ratio_pad"][si])  # native-space
labels
                labelsn = torch.cat((cls, tbox), 1)  # native-space labels
                correct_bboxes = self._process_batch(predn, labelsn)
                # TODO: maybe remove these `self.` arguments as they already
are member variable
                if self.args.plots:
```

```
                self.confusion_matrix.process_batch(predn, labelsn)
            self.stats.append((correct_bboxes, pred[:, 4], pred[:, 5],
cls.squeeze(-1)))  # (conf, pcls, tcls)

            # Save
            if self.args.save_json:
                self.pred_to_json(predn, batch["im_file"][si])
            # if self.args.save_txt:
            #    save_one_txt(predn, save_conf, shape, file=save_dir / 'labels' /
f'{path.stem}.txt')

    def get_stats(self):
        stats = [torch.cat(x, 0).cpu().numpy() for x in zip(*self.stats)]  # to
numpy
        if len(stats) and stats[0].any():
            self.metrics.process(*stats)
        self.nt_per_class = np.bincount(stats[-1].astype(int),
minlength=self.nc)  # number of targets per class
        return self.metrics.results_dict

    def print_results(self):
        pf = '%22s' + '%11i' * 2 + '%11.3g' * len(self.metrics.keys)  # print
format
        self.logger.info(pf % ("all", self.seen, self.nt_per_class.sum(),
*self.metrics.mean_results()))
        if self.nt_per_class.sum() == 0:
            self.logger.warning(
                f'WARNING ⚠ no labels found in {self.args.task} set, can not
compute metrics without labels')

        # Print results per class
        if (self.args.verbose or not self.training) and self.nc > 1 and
len(self.stats):
            for i, c in enumerate(self.metrics.ap_class_index):
                self.logger.info(pf % (self.names[c], self.seen,
self.nt_per_class[c],  *self.metrics.class_result(i)))

        if self.args.plots:
            self.confusion_matrix.plot(save_dir=self.save_dir,
names=list(self.names.values()))

    def _process_batch(self, detections, labels):
        """
        Return correct prediction matrix
        Arguments:
            detections (array[N, 6]), x1, y1, x2, y2, conf, class
            labels (array[M, 5]), class, x1, y1, x2, y2
```

```
        Returns:
            correct (array[N, 10]), for 10 IoU levels
        """
        iou = box_iou(labels[:, 1:], detections[:, :4])
        correct = np.zeros((detections.shape[0],
self.iouv.shape[0])).astype(bool)
        correct_class = labels[:, 0:1] == detections[:, 5]
        for i in range(len(self.iouv)):
            x = torch.where((iou >= self.iouv[i]) & correct_class)  # IoU >
threshold and classes match
            if x[0].shape[0]:
                matches = torch.cat((torch.stack(x, 1), iou[x[0], x[1]][:, None]),
                                    1).cpu().numpy()  # [label, detect, iou]
                if x[0].shape[0] > 1:
                    matches = matches[matches[:, 2].argsort()[::-1]]
                    matches = matches[np.unique(matches[:, 1],
return_index=True)[1]]
                    # matches = matches[matches[:, 2].argsort()[::-1]]
                    matches = matches[np.unique(matches[:, 0],
return_index=True)[1]]
                correct[matches[:, 1].astype(int), i] = True
        return torch.tensor(correct, dtype=torch.bool,
device=detections.device)

    def get_dataloader(self, dataset_path, batch_size):
        # TODO: manage splits differently
        # calculate stride - check if model is initialized
        gs = max(int(de_parallel(self.model).stride if self.model else 0), 32)
        return create_dataloader(path=dataset_path,
                                 imgsz=self.args.imgsz,
                                 batch_size=batch_size,
                                 stride=gs,
                                 hyp=dict(self.args),
                                 cache=False,
                                 pad=0.5,
                                 rect=True,
                                 workers=self.args.workers,
                                 prefix=colorstr(f'{self.args.mode}: '),
                                 shuffle=False,
                                 seed=self.args.seed)[0] if self.args.v5loader else \
            build_dataloader(self.args, batch_size, img_path=dataset_path,
stride=gs, mode="val")[0]

    def plot_val_samples(self, batch, ni):
        plot_images(batch["img"],
                    batch["batch_idx"],
                    batch["cls"].squeeze(-1),
```

```
                        batch["bboxes"],
                        paths=batch["im_file"],
                        fname=self.save_dir / f"val_batch{ni}_labels.jpg",
                        names=self.names)

    def plot_predictions(self, batch, preds, ni):
        plot_images(batch["img"],
                        *output_to_target(preds, max_det=15),
                        paths=batch["im_file"],
                        fname=self.save_dir / f'val_batch{ni}_pred.jpg',
                        names=self.names)  # pred

    def pred_to_json(self, predn, filename):
        stem = Path(filename).stem
        image_id = int(stem) if stem.isnumeric() else stem
        box = ops.xyxy2xywh(predn[:, :4])  # xywh
        box[:, :2] -= box[:, 2:] / 2  # xy center to top-left corner
        for p, b in zip(predn.tolist(), box.tolist()):
            self.jdict.append({
                'image_id': image_id,
                'category_id': self.class_map[int(p[5])],
                'bbox': [round(x, 3) for x in b],
                'score': round(p[4], 5)})

    def eval_json(self, stats):
        if self.args.save_json and self.is_coco and len(self.jdict):
            anno_json = self.data['path'] / "annotations/instances_val2017.json"
# annotations
            pred_json = self.save_dir / "predictions.json"  # predictions
            self.logger.info(f'\nEvaluating pycocotools mAP using {pred_json}
and {anno_json}...')
            try:  #
https://github.com/cocodataset/cocoapi/blob/master/PythonAPI/pycocoEva
lDemo.ipynb
                check_requirements('pycocotools>=2.0.6')
                from pycocotools.coco import COCO  # noqa
                from pycocotools.cocoeval import COCOeval  # noqa

                for x in anno_json, pred_json:
                    assert x.is_file(), f"{x} file not found"
                anno = COCO(str(anno_json))  # init annotations api
                pred = anno.loadRes(str(pred_json))  # init predictions api (must
pass string, not Path)
                eval = COCOeval(anno, pred, 'bbox')
                if self.is_coco:
                    eval.params.imgIds = [int(Path(x).stem) for x in
self.dataloader.dataset.im_files]  # images to eval
```

```
                eval.evaluate()
                eval.accumulate()
                eval.summarize()
                stats[self.metrics.keys[-1]], stats[self.metrics.keys[-2]] =
eval.stats[:2]  # update mAP50-95 and mAP50
            except Exception as e:
                self.logger.warning(f'pycocotools unable to run: {e}')
        return stats


@hydra.main(version_base=None,
config_path=str(DEFAULT_CONFIG.parent),
config_name=DEFAULT_CONFIG.name)
def val(cfg):
    cfg.data = cfg.data or "coco128.yaml"
    validator = DetectionValidator(args=cfg)
    validator(model=cfg.model)


if __name___== "__main__":
                    val()
```

Train.py:

```
# Ultralytics YOLO 🚀 GPL-3.0 license

from copy import copy

import hydra
import torch
import torch.nn as nn

from ultralytics.nn.tasks import DetectionModel
from ultralytics.yolo import v8
from ultralytics.yolo.data import build_dataloader
from ultralytics.yolo.data.dataloaders.v5loader import create_dataloader
from ultralytics.yolo.engine.trainer import BaseTrainer
from ultralytics.yolo.utils import DEFAULT_CONFIG, colorstr
from ultralytics.yolo.utils.loss import BboxLoss
from ultralytics.yolo.utils.ops import xywh2xyxy
from ultralytics.yolo.utils.plotting import plot_images, plot_results
from ultralytics.yolo.utils.tal import TaskAlignedAssigner, dist2bbox,
make_anchors
from ultralytics.yolo.utils.torch_utils import de_parallel


# BaseTrainer python usage
```

```
class DetectionTrainer(BaseTrainer):

    def get_dataloader(self, dataset_path, batch_size, mode="train", rank=0):
        # TODO: manage splits differently
        # calculate stride - check if model is initialized
        gs = max(int(de_parallel(self.model).stride.max() if self.model else 0),
32)
        return create_dataloader(path=dataset_path,
                                 imgsz=self.args.imgsz,
                                 batch_size=batch_size,
                                 stride=gs,
                                 hyp=dict(self.args),
                                 augment=mode == "train",
                                 cache=self.args.cache,
                                 pad=0 if mode == "train" else 0.5,
                                 rect=self.args.rect,
                                 rank=rank,
                                 workers=self.args.workers,
                                 close_mosaic=self.args.close_mosaic != 0,
                                 prefix=colorstr(f'{mode}: '),
                                 shuffle=mode == "train",
                                 seed=self.args.seed)[0] if self.args.v5loader else \
            build_dataloader(self.args, batch_size, img_path=dataset_path,
stride=gs, rank=rank, mode=mode)[0]

    def preprocess_batch(self, batch):
        batch["img"] = batch["img"].to(self.device,
non_blocking=True).float() / 255
        return batch

    def set_model_attributes(self):
        nl = de_parallel(self.model).model[-1].nl  # number of detection layers
(to scale hyps)
        self.args.box *= 3 / nl  # scale to layers
        # self.args.cls *= self.data["nc"] / 80 * 3 / nl  # scale to classes and
layers
        self.args.cls *= (self.args.imgsz / 640) ** 2 * 3 / nl  # scale to image
size and layers
        self.model.nc = self.data["nc"]  # attach number of classes to model
        self.model.args = self.args  # attach hyperparameters to model
        # TODO: self.model.class_weights =
labels_to_class_weights(dataset.labels, nc).to(device) * nc
        self.model.names = self.data["names"]

    def get_model(self, cfg=None, weights=None, verbose=True):
        model = DetectionModel(cfg, ch=3, nc=self.data["nc"],
verbose=verbose)
```

```
        if weights:
            model.load(weights)

        return model

    def get_validator(self):
        self.loss_names = 'box_loss', 'cls_loss', 'dfl_loss'
        return v8.detect.DetectionValidator(self.test_loader,
                                             save_dir=self.save_dir,
                                             logger=self.console,
                                             args=copy(self.args))

    def criterion(self, preds, batch):
        if not hasattr(self, 'compute_loss'):
            self.compute_loss = Loss(de_parallel(self.model))
        return self.compute_loss(preds, batch)

    def label_loss_items(self, loss_items=None, prefix="train"):
        """
        Returns a loss dict with labelled training loss items tensor
        """
        # Not needed for classification but necessary for segmentation &
detection
        keys = [f"{prefix}/{x}" for x in self.loss_names]
        if loss_items is not None:
            loss_items = [round(float(x), 5) for x in loss_items]  # convert
tensors to 5 decimal place floats
            return dict(zip(keys, loss_items))
        else:
            return keys

    def progress_string(self):
        return ('\n' + '%11s' *
                (4 + len(self.loss_names))) % ('Epoch', 'GPU_mem',
*self.loss_names, 'Instances', 'Size')

    def plot_training_samples(self, batch, ni):
        plot_images(images=batch["img"],
                    batch_idx=batch["batch_idx"],
                    cls=batch["cls"].squeeze(-1),
                    bboxes=batch["bboxes"],
                    paths=batch["im_file"],
                    fname=self.save_dir / f"train_batch{ni}.jpg")

    def plot_metrics(self):
        plot_results(file=self.csv)  # save results.png
```

```
# Criterion class for computing training losses
class Loss:

    def __init__(self, model):  # model must be de-paralleled

        device = next(model.parameters()).device  # get model device
        h = model.args  # hyperparameters

        m = model.model[-1]  # Detect() module
        self.bce = nn.BCEWithLogitsLoss(reduction='none')
        self.hyp = h
        self.stride = m.stride  # model strides
        self.nc = m.nc  # number of classes
        self.no = m.no
        self.reg_max = m.reg_max
        self.device = device

        self.use_dfl = m.reg_max > 1
        self.assigner = TaskAlignedAssigner(topk=10, num_classes=self.nc,
alpha=0.5, beta=6.0)
        self.bbox_loss = BboxLoss(m.reg_max - 1,
use_dfl=self.use_dfl).to(device)
        self.proj = torch.arange(m.reg_max, dtype=torch.float, device=device)

    def preprocess(self, targets, batch_size, scale_tensor):
        if targets.shape[0] == 0:
            out = torch.zeros(batch_size, 0, 5, device=self.device)
        else:
            i = targets[:, 0]  # image index
            _, counts = i.unique(return_counts=True)
            out = torch.zeros(batch_size, counts.max(), 5, device=self.device)
            for j in range(batch_size):
                matches = i == j
                n = matches.sum()
                if n:
                    out[j, :n] = targets[matches, 1:]
            out[..., 1:5] = xywh2xyxy(out[..., 1:5].mul_(scale_tensor))
        return out

    def bbox_decode(self, anchor_points, pred_dist):
        if self.use_dfl:
            b, a, c = pred_dist.shape  # batch, anchors, channels
            pred_dist = pred_dist.view(b, a, 4, c //
4).softmax(3).matmul(self.proj.type(pred_dist.dtype))
            # pred_dist = pred_dist.view(b, a, c // 4,
4).transpose(2,3).softmax(3).matmul(self.proj.type(pred_dist.dtype))
```

```
        # pred_dist = (pred_dist.view(b, a, c // 4, 4).softmax(2) *
self.proj.type(pred_dist.dtype).view(1, 1, -1, 1)).sum(2)
        return dist2bbox(pred_dist, anchor_points, xywh=False)

    def __call__(self, preds, batch):
        loss = torch.zeros(3, device=self.device)  # box, cls, dfl
        feats = preds[1] if isinstance(preds, tuple) else preds
        pred_distri, pred_scores = torch.cat([xi.view(feats[0].shape[0], self.no,
-1) for xi in feats], 2).split(
            (self.reg_max * 4, self.nc), 1)

        pred_scores = pred_scores.permute(0, 2, 1).contiguous()
        pred_distri = pred_distri.permute(0, 2, 1).contiguous()

        dtype = pred_scores.dtype
        batch_size = pred_scores.shape[0]
        imgsz = torch.tensor(feats[0].shape[2:], device=self.device,
dtype=dtype) * self.stride[0]  # image size (h,w)
        anchor_points, stride_tensor = make_anchors(feats, self.stride, 0.5)

        # targets
        targets = torch.cat((batch["batch_idx"].view(-1, 1), batch["cls"].view(-
1, 1), batch["bboxes"]), 1)
        targets = self.preprocess(targets.to(self.device), batch_size,
scale_tensor=imgsz[[1, 0, 1, 0]])
        gt_labels, gt_bboxes = targets.split((1, 4), 2)  # cls, xyxy
        mask_gt = gt_bboxes.sum(2, keepdim=True).gt_(0)

        # pboxes
        pred_bboxes = self.bbox_decode(anchor_points, pred_distri)  # xyxy,
(b, h*w, 4)

        _, target_bboxes, target_scores, fg_mask, _ = self.assigner(
            pred_scores.detach().sigmoid(), (pred_bboxes.detach() *
stride_tensor).type(gt_bboxes.dtype),
            anchor_points * stride_tensor, gt_labels, gt_bboxes, mask_gt)

        target_bboxes /= stride_tensor
        target_scores_sum = target_scores.sum()

        # cls loss
        # loss[1] = self.varifocal_loss(pred_scores, target_scores,
target_labels) / target_scores_sum  # VFL way
        loss[1] = self.bce(pred_scores, target_scores.to(dtype)).sum() /
target_scores_sum  # BCE

        # bbox loss
```

```
        if fg_mask.sum():
            loss[0], loss[2] = self.bbox_loss(pred_distri, pred_bboxes,
anchor_points, target_bboxes, target_scores,
                                              target_scores_sum, fg_mask)

        loss[0] *= self.hyp.box  # box gain
        loss[1] *= self.hyp.cls  # cls gain
        loss[2] *= self.hyp.dfl  # dfl gain

        return loss.sum() * batch_size, loss.detach()  # loss(box, cls, dfl)


@hydra.main(version_base=None,
config_path=str(DEFAULT_CONFIG.parent),
config_name=DEFAULT_CONFIG.name)
def train(cfg):
    cfg.model = cfg.model or "yolov8n.yaml"
    cfg.data = cfg.data or "coco128.yaml"  # or
yolo.ClassificationDataset("mnist")
    # trainer = DetectionTrainer(cfg)
    # trainer.train()
    from ultralytics import YOLO
    model = YOLO(cfg.model)
    model.train(**cfg)


if __name__ == "__main__":
    """
    CLI usage:
    python ultralytics/yolo/v8/detect/train.py model=yolov8n.yaml
data=coco128 epochs=100 imgsz=640

    TODO:
    yolo task=detect mode=train model=yolov8n.yaml data=coco128.yaml
epochs=100
    """
    train()
```

Predict.py:

```
# Ultralytics YOLO 🚀 GPL-3.0 license

import hydra
import torch
import argparse
import time
```

```
from pathlib import Path
import math
import cv2
import torch
import torch.backends.cudnn as cudnn
from numpy import random
from ultralytics.yolo.engine.predictor import BasePredictor
from ultralytics.yolo.utils import DEFAULT_CONFIG, ROOT, ops
from ultralytics.yolo.utils.checks import check_imgsz
from ultralytics.yolo.utils.plotting import Annotator, colors, save_one_box

import cv2
from deep_sort_pytorch.utils.parser import get_config
from deep_sort_pytorch.deep_sort import DeepSort
from collections import deque
import numpy as np
palette = (2 ** 11 - 1, 2 ** 15 - 1, 2 ** 20 - 1)
data_deque = {}

deepsort = None

object_counter = {}

object_counter1 = {}

line = [(100, 500), (1050, 500)]

speed_line_queue={}

def estimatespeed(Location1,Location2):
    d_pixel=math.sqrt(math.pow(Location2[0]-
Location1[0],2)+math.pow(Location2[1]-Location1[1],2))
    ppm=8
    d_meters=d_pixel/ppm
    time_constant=15*3.6

    speed=d_meters+time_constant
    return int(speed)


def init_tracker():
    global deepsort
    cfg_deep = get_config()

cfg_deep.merge_from_file("deep_sort_pytorch/configs/deep_sort.yaml")

    deepsort= DeepSort(cfg_deep.DEEPSORT.REID_CKPT,
```

```
                    max_dist=cfg_deep.DEEPSORT.MAX_DIST,
min_confidence=cfg_deep.DEEPSORT.MIN_CONFIDENCE,

nms_max_overlap=cfg_deep.DEEPSORT.NMS_MAX_OVERLAP,
max_iou_distance=cfg_deep.DEEPSORT.MAX_IOU_DISTANCE,
                    max_age=cfg_deep.DEEPSORT.MAX_AGE,
n_init=cfg_deep.DEEPSORT.N_INIT,
nn_budget=cfg_deep.DEEPSORT.NN_BUDGET,
                    use_cuda=True)
##########################################################################################
def xyxy_to_xywh(*xyxy):
    """" Calculates the relative bounding box from absolute pixel values. """
    bbox_left = min([xyxy[0].item(), xyxy[2].item()])
    bbox_top = min([xyxy[1].item(), xyxy[3].item()])
    bbox_w = abs(xyxy[0].item() - xyxy[2].item())
    bbox_h = abs(xyxy[1].item() - xyxy[3].item())
    x_c = (bbox_left + bbox_w / 2)
    y_c = (bbox_top + bbox_h / 2)
    w = bbox_w
    h = bbox_h
    return x_c, y_c, w, h

def xyxy_to_tlwh(bbox_xyxy):
    tlwh_bboxs = []
    for i, box in enumerate(bbox_xyxy):
        x1, y1, x2, y2 = [int(i) for i in box]
        top = x1
        left = y1
        w = int(x2 - x1)
        h = int(y2 - y1)
        tlwh_obj = [top, left, w, h]
        tlwh_bboxs.append(tlwh_obj)
    return tlwh_bboxs

def compute_color_for_labels(label):
    """
    Simple function that adds fixed color depending on the class
    """
    if label == 0: #person
        color = (85,45,255)
    elif label == 2: # Car
        color = (222,82,175)
    elif label == 3:  # Motobike
        color = (0, 204, 255)
    elif label == 5:  # Bus
        color = (0, 149, 255)
```

```
    else:
        color = [int((p * (label ** 2 - label + 1)) % 255) for p in palette]
    return tuple(color)

def draw_border(img, pt1, pt2, color, thickness, r, d):
    x1,y1 = pt1
    x2,y2 = pt2
    # Top left
    cv2.line(img, (x1 + r, y1), (x1 + r + d, y1), color, thickness)
    cv2.line(img, (x1, y1 + r), (x1, y1 + r + d), color, thickness)
    cv2.ellipse(img, (x1 + r, y1 + r), (r, r), 180, 0, 90, color, thickness)
    # Top right
    cv2.line(img, (x2 - r, y1), (x2 - r - d, y1), color, thickness)
    cv2.line(img, (x2, y1 + r), (x2, y1 + r + d), color, thickness)
    cv2.ellipse(img, (x2 - r, y1 + r), (r, r), 270, 0, 90, color, thickness)
    # Bottom left
    cv2.line(img, (x1 + r, y2), (x1 + r + d, y2), color, thickness)
    cv2.line(img, (x1, y2 - r), (x1, y2 - r - d), color, thickness)
    cv2.ellipse(img, (x1 + r, y2 - r), (r, r), 90, 0, 90, color, thickness)
    # Bottom right
    cv2.line(img, (x2 - r, y2), (x2 - r - d, y2), color, thickness)
    cv2.line(img, (x2, y2 - r), (x2, y2 - r - d), color, thickness)
    cv2.ellipse(img, (x2 - r, y2 - r), (r, r), 0, 0, 90, color, thickness)

    cv2.rectangle(img, (x1 + r, y1), (x2 - r, y2), color, -1, cv2.LINE_AA)
    cv2.rectangle(img, (x1, y1 + r), (x2, y2 - r - d), color, -1, cv2.LINE_AA)

    cv2.circle(img, (x1 +r, y1+r), 2, color, 12)
    cv2.circle(img, (x2 -r, y1+r), 2, color, 12)
    cv2.circle(img, (x1 +r, y2-r), 2, color, 12)
    cv2.circle(img, (x2 -r, y2-r), 2, color, 12)

    return img

def UI_box(x, img, color=None, label=None, line_thickness=None):
    # Plots one bounding box on image img
    tl = line_thickness or round(0.002 * (img.shape[0] + img.shape[1]) / 2) +
1  # line/font thickness
    color = color or [random.randint(0, 255) for _ in range(3)]
    c1, c2 = (int(x[0]), int(x[1])), (int(x[2]), int(x[3]))
    cv2.rectangle(img, c1, c2, color, thickness=tl, lineType=cv2.LINE_AA)
    if label:
        tf = max(tl - 1, 1)  # font thickness
        t_size = cv2.getTextSize(label, 0, fontScale=tl / 3, thickness=tf)[0]

        img = draw_border(img, (c1[0], c1[1] - t_size[1] -3), (c1[0] + t_size[0],
c1[1]+3), color, 1, 8, 2)
```

```
        cv2.putText(img, label, (c1[0], c1[1] - 2), 0, tl / 3, [225, 255, 255], thickness=tf, lineType=cv2.LINE_AA)


def intersect(A,B,C,D):
    return ccw(A,C,D) != ccw(B,C,D) and ccw(A,B,C) != ccw(A,B,D)

def ccw(A,B,C):
    return (C[1]-A[1]) * (B[0]-A[0]) > (B[1]-A[1]) * (C[0]-A[0])


def get_direction(point1, point2):
    direction_str = ""

    # calculate y axis direction
    if point1[1] > point2[1]:
        direction_str += "South"
    elif point1[1] < point2[1]:
        direction_str += "North"
    else:
        direction_str += ""

    # calculate x axis direction
    if point1[0] > point2[0]:
          direction_str += "East"
      elif point1[0] < point2[0]:
         direction_str += "West"
    else:
        direction_str += ""

    return direction_str
def draw_boxes(img, bbox, names,object_id, identities=None, offset=(0, 0)):
    cv2.line(img, line[0], line[1], (46,162,112), 3)

    height, width, _ = img.shape
    # remove tracked point from buffer if object is lost
    for key in list(data_deque):
      if key not in identities:
        data_deque.pop(key)

    for i, box in enumerate(bbox):
        x1, y1, x2, y2 = [int(i) for i in box]
        x1 += offset[0]
        x2 += offset[0]
        y1 += offset[1]
```

```
            y2 += offset[1]

            # code to find center of bottom edge
            center = (int((x2+x1)/ 2), int((y2+y2)/2))

            # get ID of object
            id = int(identities[i]) if identities is not None else 0

            # create new buffer for new object
            if id not in data_deque:
                speed_line_queue[id]=[]
                data_deque[id] = deque(maxlen= 64)
            color = compute_color_for_labels(object_id[i])
            obj_name = names[object_id[i]]
            label = '{}{:d}'.format("", id) + ":"+ '%s' % (obj_name)

            # add center to buffer
            data_deque[id].appendleft(center)
            if len(data_deque[id]) >= 2:
              direction = get_direction(data_deque[id][0], data_deque[id][1])
              object_speed=estimatespeed(data_deque[id][1],data_deque[id][0])
              speed_line_queue[id].append(object_speed)
              if intersect(data_deque[id][0], data_deque[id][1], line[0], line[1]):
                  cv2.line(img, line[0], line[1], (255, 255, 255), 3)
                  if "South" in direction:
                    if obj_name not in object_counter:
                        object_counter[obj_name] = 1
                    else:
                        object_counter[obj_name] += 1
                  if "North" in direction:
                    if obj_name not in object_counter1:
                        object_counter1[obj_name] = 1
                    else:
                        object_counter1[obj_name]  +=  1
            try:
                label=label+"
"+str(sum(speed_line_queue[id])//len(speed_line_queue[id])) +"kmph"
            except:
                pass
            UI_box(box, img, label=label, color=color, line_thickness=2)
            # draw trail
            for i in range(1, len(data_deque[id])):
                # check if on buffer value is none
                if data_deque[id][i - 1] is None or data_deque[id][i] is None:
                    continue
                # generate dynamic thickness of trails
                thickness = int(np.sqrt(64 / float(i + i)) * 1.5)
```

```
            # draw trails
            cv2.line(img, data_deque[id][i - 1], data_deque[id][i], color,
thickness)

    #4. Display Count in top right corner
        for idx, (key, value) in enumerate(object_counter1.items()):
            cnt_str = str(key) + ":" +str(value)
            cv2.line(img, (width - 500,25), (width,25), [85,45,255], 40)
            cv2.putText(img, f'Number of Vehicles Entering', (width - 500, 35),
0, 1, [225, 255, 255], thickness=2, lineType=cv2.LINE_AA)
            cv2.line(img, (width - 150, 65 + (idx*40)), (width, 65 + (idx*40)),
[85, 45, 255], 30)
            cv2.putText(img, cnt_str, (width - 150, 75 + (idx*40)), 0, 1, [255,
255, 255], thickness = 2, lineType = cv2.LINE_AA)

        for idx, (key, value) in enumerate(object_counter.items()):
            cnt_str1 = str(key) + ":" +str(value)
            cv2.line(img, (20,25), (500,25), [85,45,255], 40)
            cv2.putText(img, f'Numbers of Vehicles Leaving', (11, 35), 0, 1,
[225, 255, 255], thickness=2, lineType=cv2.LINE_AA)
            cv2.line(img, (20,65+ (idx*40)), (127,65+ (idx*40)), [85,45,255],
30)
            cv2.putText(img, cnt_str1, (11, 75+ (idx*40)), 0, 1, [225, 255, 255],
thickness=2, lineType=cv2.LINE_AA)



    return img


class DetectionPredictor(BasePredictor):

    def get_annotator(self, img):
        return Annotator(img, line_width=self.args.line_thickness,
example=str(self.model.names))

    def preprocess(self, img):
        img = torch.from_numpy(img).to(self.model.device)
        img = img.half() if self.model.fp16 else img.float()  # uint8 to fp16/32
        img /= 255  # 0 - 255 to 0.0 - 1.0
        return img

    def postprocess(self, preds, img, orig_img):
        preds = ops.non_max_suppression(preds,
                                        self.args.conf,
                                        self.args.iou,
                                        agnostic=self.args.agnostic_nms,
```

```
                                        max_det=self.args.max_det)

        for i, pred in enumerate(preds):
            shape = orig_img[i].shape if self.webcam else orig_img.shape
            pred[:, :4] = ops.scale_boxes(img.shape[2:], pred[:, :4],
shape).round()

        return preds

    def write_results(self, idx, preds, batch):
        p, im, im0 = batch
        all_outputs = []
        log_string = ""
        if len(im.shape) == 3:
            im = im[None]  # expand for batch dim
        self.seen += 1
        im0 = im0.copy()
        if self.webcam:  # batch_size >= 1
            log_string += f'{idx}: '
            frame = self.dataset.count
        else:
            frame = getattr(self.dataset, 'frame', 0)

        self.data_path = p
        save_path = str(self.save_dir / p.name)  # im.jpg
        self.txt_path = str(self.save_dir / 'labels' / p.stem) + ('' if
self.dataset.mode == 'image' else f'_{frame}')
        log_string += '%gx%g ' % im.shape[2:]  # print string
        self.annotator = self.get_annotator(im0)

        det = preds[idx]
        all_outputs.append(det)
        if len(det) == 0:
            return log_string
        for c in det[:, 5].unique():
            n = (det[:, 5] == c).sum()  # detections per class
            log_string += f"{n} {self.model.names[int(c)]}{'s' * (n > 1)}, "
        # write
        gn = torch.tensor(im0.shape)[[1, 0, 1, 0]]  # normalization gain whwh
        xywh_bboxs = []
        confs = []
        oids = []
        outputs = []
        for *xyxy, conf, cls in reversed(det):
            x_c, y_c, bbox_w, bbox_h = xyxy_to_xywh(*xyxy)
            xywh_obj = [x_c, y_c, bbox_w, bbox_h]
            xywh_bboxs.append(xywh_obj)
```

```
            confs.append([conf.item()])
            oids.append(int(cls))
        xywhs = torch.Tensor(xywh_bboxs)
        confss = torch.Tensor(confs)

        outputs = deepsort.update(xywhs, confss, oids, im0)
        if len(outputs) > 0:
            bbox_xyxy = outputs[:, :4]
            identities = outputs[:, -2]
            object_id = outputs[:, -1]

            draw_boxes(im0,                 bbox_xyxy,                 self.model.names,
object_id,identities)

        return log_string


@hydra.main(version_base=None,
config_path=str(DEFAULT_CONFIG.parent),
config_name=DEFAULT_CONFIG.name)
def predict(cfg):
    init_tracker()
    cfg.model = cfg.model or "yolov8n.pt"
    cfg.imgsz = check_imgsz(cfg.imgsz, min_dim=2)  # check image size
    cfg.source = cfg.source if cfg.source is not None else ROOT / "assets"
    predictor = DetectionPredictor(cfg)
    predictor()


if __name___== "__main__":
    predict()
```

LANE DETECTION :

```
import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import numpy as np

import cv2

import os

import glob
```

```
from moviepy.editor import VideoFileClip

%matplotlib inline

def list_images(images, cols = 2, rows = 5, cmap=None):

 """

 Display a list of images in a single figure with matplotlib.

 Parameters:

 images: List of np.arrays compatible with plt.imshow.

 cols (Default = 2): Number of columns in the figure.

 rows (Default = 5): Number of rows in the figure.

 cmap (Default = None): Used to display gray images.


 """

 plt.figure(figsize=(10, 11))

 for i, image in enumerate(images):

 plt.subplot(rows, cols, i+1)

 #Use gray scale color map if there is only one channel

 cmap = 'gray' if len(image.shape) == 2 else cmap

 plt.imshow(image, cmap = cmap)

 plt.xticks([])

 plt.yticks([])

 plt.tight_layout(pad=0, h_pad=0, w_pad=0)

 plt.show()

#Reading in the test images

test_images = [plt.imread(img) for img in
```

glob.glob('test_images/*.jpg')]

list_images(test_images)

2. Color Selection

Lane lines in the test images are in white and yellow. We need to choose the most suitable color

space, that clearly highlights the lane lines.

Original RGB color selection

I will apply color selection to the test_images in the original RGB format. We will try to retain

as much of the lane lines as possible, while blacking out most of the other stuff.

```
def RGB_color_selection(image):
 """
 Apply color selection to RGB images to blackout everything except
for white and yellow lane lines.
 Parameters:
 image: An np.array compatible with plt.imshow.
 """
 #White color mask
 lower_threshold = np.uint8([200, 200, 200])
 upper_threshold = np.uint8([255, 255, 255])
 white_mask = cv2.inRange(image, lower_threshold, upper_threshold)

 #Yellow color mask
 lower_threshold = np.uint8([175, 175, 0])
 upper_threshold = np.uint8([255, 255, 255])
```

```
 yellow_mask = cv2.inRange(image, lower_threshold, upper_threshold)


 #Combine white and yellow masks

 mask = cv2.bitwise_or(white_mask, yellow_mask)

 masked_image = cv2.bitwise_and(image, image, mask = mask)


 return masked_image
```

Applying color selection to test_images in the RGB color space.

```
list_images(list(map(RGB_color_selection, test_images)))
```

a) HSV color space

Wikipedia: HSV is an alternative representation of the RGB color model. The HSV representation

models the way colors mix together, with the saturation dimension resembling various shades

of brightly colored paint, and the value dimension resembling the mixture of those paints with

varying amounts of black or white.

```
def convert_hsv(image):

 """

 Convert RGB images to HSV.

 Parameters:

 image: An np.array compatible with plt.imshow.

 """

 return cv2.cvtColor(image, cv2.COLOR_RGB2HSV)

list_images(list(map(convert_hsv, test_images)))
```

```
def HSV_color_selection(image):
 """
 Apply color selection to the HSV images to blackout everything
except for white and yellow lane lines.
 Parameters:
 image: An np.array compatible with plt.imshow.
 """
 #Convert the input image to HSV
 converted_image = convert_hsv(image)

 #White color mask
 lower_threshold = np.uint8([0, 0, 210])
 upper_threshold = np.uint8([255, 30, 255])
 white_mask = cv2.inRange(converted_image, lower_threshold,
upper_threshold)

 #Yellow color mask
 lower_threshold = np.uint8([18, 80, 80])
 upper_threshold = np.uint8([30, 255, 255])
 yellow_mask = cv2.inRange(converted_image, lower_threshold,
upper_threshold)

 #Combine white and yellow masks
 mask = cv2.bitwise_or(white_mask, yellow_mask)
```

masked_image = cv2.bitwise_and(image, image, mask = mask)

return masked_image

Applying color selection to test_images in the HSV color space.

list_images(list(map(HSV_color_selection, test_images)))

c) HSL color space

Wikipedia: HSL is an alternative representation of the RGB color model. The HSL model

attempts to resemble more perceptual color models such as NCS or Munsell, placing fully

saturated colors around a circle at a lightness value of 1/2, where a lightness value of 0 or 1 is

fully black or white, respectively.

def convert_hsl(image):

""""

Convert RGB images to HSL.

Parameters:

image: An np.array compatible with plt.imshow.

""""

return cv2.cvtColor(image, cv2.COLOR_RGB2HLS)

list_images(list(map(convert_hsl, test_images)))

def HSL_color_selection(image):

""""

Apply color selection to the HSL images to blackout everything

except for white and yellow lane lines.

Parameters:

image: An np.array compatible with plt.imshow.

""""

#Convert the input image to HSL

converted_image = convert_hsl(image)

#White color mask

lower_threshold = np.uint8([0, 200, 0])

upper_threshold = np.uint8([255, 255, 255])

white_mask = cv2.inRange(converted_image, lower_threshold, upper_threshold)

#Yellow color mask

lower_threshold = np.uint8([10, 0, 100])

upper_threshold = np.uint8([40, 255, 255])

yellow_mask = cv2.inRange(converted_image, lower_threshold, upper_threshold)

#Combine white and yellow masks

mask = cv2.bitwise_or(white_mask, yellow_mask)

masked_image = cv2.bitwise_and(image, image, mask = mask)

return masked_image

Applying color selection to test_images in the HSL color space.

list_images(list(map(HSL_color_selection, test_images)))

Using HSL produces the clearest lane lines of all color spaces. We will use them for the next

steps.

color_selected_images = list(map(HSL_color_selection, test_images))

3. Canny Edge Detection

Wikipedia: The Canny edge detector is an edge detection operator that uses a multi-stage

algorithm to detect a wide range of edges in images.

a) Gray scaling the images

The Canny edge detection algorithm measures the intensity gradients of each pixel. So, we need

to convert the images into gray scale in order to detect edges.

```
def gray_scale(image):

 """

 Convert images to gray scale.

 Parameters:

 image: An np.array compatible with plt.imshow.

 """

 return cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

gray_images = list(map(gray_scale, color_selected_images))

list_images(gray_images)
```

b) Applying Gaussian smoothing

Wikipedia: Since all edge detection results are easily affected by image noise, it is essential to filter out the noise to prevent false detection caused by noise. To smooth the image, a Gaussian

filter is applied to convolve with the image. This step will slightly smooth the image to reduce

the effects of obvious noise on the edge detector.

def gaussian_smoothing(image, kernel_size = 13):

""""

Apply Gaussian filter to the input image.

Parameters:

image: An np.array compatible with plt.imshow.

kernel_size (Default = 13): The size of the Gaussian

kernel will affect the performance of the detector.

It must be an odd number (3, 5, 7, ...).

""""

return cv2.GaussianBlur(image, (kernel_size, kernel_size), 0)

blur_images = list(map(gaussian_smoothing, gray_images))

list_images(blur_images)

c) Applying Canny Edge Detection

Wikipedia: The Process of Canny edge detection algorithm can be broken down to 5 different

steps:

1. Find the intensity gradients of the image

2. Apply non-maximum suppression to get rid of spurious response to edge detection.

3. Apply double threshold to determine potential edges.

4. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other

edges that are weak and not connected to strong edges.

****If an edge pixel's gradient value is higher than the high threshold value, it is marked as a**

strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and

larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is

smaller than the low threshold value, it will be suppressed. The two threshold values are

empirically determined and their definition will depend on the content of a given input image.*

```
def canny_detector(image, low_threshold = 50, high_threshold = 150):

 """

 Apply Canny Edge Detection algorithm to the input image.

 Parameters:

 image: An np.array compatible with plt.imshow.

 low_threshold (Default = 50).

 high_threshold (Default = 150).

 """

 return cv2.Canny(image, low_threshold, high_threshold)

edge_detected_images = list(map(canny_detector, blur_images))

list_images(edge_detected_images)
```

4. Region of interest

We're interested in the area facing the camera, where the lane lines are found. So, we'll apply

region masking to cut out everything else.

```
def region_selection(image):
 """
 Determine and cut the region of interest in the input image.
 Parameters:
 image: An np.array compatible with plt.imshow.
 """
 mask = np.zeros_like(image)
 #Defining a 3 channel or 1 channel color to fill the mask with
depending on the input image
 if len(image.shape) > 2:
 channel_count = image.shape[2]
 ignore_mask_color = (255,) * channel_count
 else:
 ignore_mask_color = 255
 #We could have used fixed numbers as the vertices of the polygon,
#but they will not be applicable to images with different
dimesnions.
 rows, cols = image.shape[:2]
 bottom_left = [cols * 0.1, rows * 0.95]
 top_left = [cols * 0.4, rows * 0.6]
 bottom_right = [cols * 0.9, rows * 0.95]
 top_right = [cols * 0.6, rows * 0.6]
 vertices = np.array([[bottom_left, top_left, top_right,
```

bottom_right]], dtype=np.int32)

 cv2.fillPoly(mask, vertices, ignore_mask_color)

 masked_image = cv2.bitwise_and(image, mask)

 return masked_image

masked_image = list(map(region_selection, edge_detected_images))

list_images(masked_image)

5. Hough Transform

The Hough transform is a technique which can be used to isolate features of a particular shape

within an image. I'll use it to detected the lane lines in selected_region_images.

def hough_transform(image):

 """

 Determine and cut the region of interest in the input image.

 Parameters:

 image: The output of a Canny transform.

 """

 rho = 1 #Distance resolution of the accumulator in

pixels.

 theta = np.pi/180 #Angle resolution of the accumulator in

radians.

 threshold = 20 #Only lines that are greater than threshold

will be returned.

 minLineLength = 20 #Line segments shorter than that are

rejected.

maxLineGap = 300 #Maximum allowed gap between points on the

same line to link them

return cv2.HoughLinesP(image, rho = rho, theta = theta, threshold

= threshold,

minLineLength = minLineLength, maxLineGap =

maxLineGap)

hough_lines contains the list of lines detected in the selected region. Now, we will draw these

detected lines onto the original test_images.

hough_lines = list(map(hough_transform, masked_image))

def draw_lines(image, lines, color = [255, 0, 0], thickness = 2):

""""

Draw lines onto the input image.

Parameters:

image: An np.array compatible with plt.imshow.

lines: The lines we want to draw.

color (Default = red): Line color.

thickness (Default = 2): Line thickness.

""""

image = np.copy(image)

for line in lines:

for x1,y1,x2,y2 in line:

cv2.line(image, (x1, y1), (x2, y2), color, thickness)

return image

line_images = []

for image, lines in zip(test_images, hough_lines):

line_images.append(draw_lines(image, lines))

list_images(line_images)

6. Averaging and extrapolating the lane lines

We have multiple lines detected for each lane line. We need to average all these lines and draw a

single line for each lane line. We also need to extrapolate the lane lines to cover the full lane line

length.

def average_slope_intercept(lines):

"""

Find the slope and intercept of the left and right lanes of each

image.

Parameters:

lines: The output lines from Hough Transform.

"""

left_lines = [] #(slope, intercept)

left_weights = [] #(length,)

right_lines = [] #(slope, intercept)

right_weights = [] #(length,)

for line in lines:

for x1, y1, x2, y2 in line:

if x1 == x2:

```
continue
slope = (y2 - y1) / (x2 - x1)
intercept = y1 - (slope * x1)
length = np.sqrt(((y2 - y1) ** 2) + ((x2 - x1) ** 2))
if slope < 0:
left_lines.append((slope, intercept))
left_weights.append((length))
else:
right_lines.append((slope, intercept))
right_weights.append((length))
left_lane = np.dot(left_weights, left_lines) /
np.sum(left_weights) if len(left_weights) > 0 else None
right_lane = np.dot(right_weights, right_lines) /
np.sum(right_weights) if len(right_weights) > 0 else None
return left_lane, right_lane
def pixel_points(y1, y2, line):
"""
Converts the slope and intercept of each line into pixel points.
Parameters:
y1: y-value of the line's starting point.
y2: y-value of the line's end point.
line: The slope and intercept of the line.
"""
if line is None:
```

```
return None
slope, intercept = line
x1 = int((y1 - intercept)/slope)
x2 = int((y2 - intercept)/slope)
y1 = int(y1)
y2 = int(y2)
return ((x1, y1), (x2, y2))
def lane_lines(image, lines):
"""
Create full lenght lines from pixel points.
Parameters:
image: The input test image.
lines: The output lines from Hough Transform.
"""
left_lane, right_lane = average_slope_intercept(lines)
y1 = image.shape[0]
y2 = y1 * 0.6
left_line = pixel_points(y1, y2, left_lane)
right_line = pixel_points(y1, y2, right_lane)
return left_line, right_line

def draw_lane_lines(image, lines, color=[255, 0, 0], thickness=12):
"""
Draw lines onto the input image.
```

Parameters:

image: The input test image.

lines: The output lines from Hough Transform.

color (Default = red): Line color.

thickness (Default = 12): Line thickness.

"""

line_image = np.zeros_like(image)

for line in lines:

if line is not None:

cv2.line(line_image, *line, color, thickness)

return cv2.addWeighted(image, 1.0, line_image, 1.0, 0.0)

lane_images = []

for image, lines in zip(test_images, hough_lines):

lane_images.append(draw_lane_lines(image, lane_lines(image, lines)))

list_images(lane_images)

7. Apply on video streams

Now, we'll use the above functions to detect lane lines from a video stream.

#Import everything needed to edit/save/watch video clips

from moviepy import *

from IPython.display import HTML

```
from IPython.display import Image
def frame_processor(image):
 """
 Process the input frame to detect lane lines.
 Parameters:
 image: Single video frame.
 """
 color_select = HSL_color_selection(image)
 gray = gray_scale(color_select)
 smooth = gaussian_smoothing(gray)
 edges = canny_detector(smooth)
 region = region_selection(edges)
 hough = hough_transform(region)
 result = draw_lane_lines(image, lane_lines(image, hough))
 return result
def process_video(test_video, output_video):
 """
 Read input video stream and produce a video file with detected
lane lines.
 Parameters:
 test_video: Input video.
 output_video: A video file with detected lane lines.
 """
 input_video = VideoFileClip(os.path.join('test_videos',
```

```
test_video), audio=False)
 processed = input_video.fl_image(frame_processor)
processed.write_videofile(os.path.join('output_videos',
output_video), audio=False)
%time process_video('solidWhiteRight.mp4',
'solidWhiteRight_output.mp4')
HTML("""
<video width="960" height="540" controls>
 <source src="{0}">
</video>
""".format("output_videos\solidWhiteRight_output.mp4"))
Moviepy - Building video output_videos\solidWhiteRight_output.mp4.
Moviepy - Writing video output_videos\solidWhiteRight_output.mp4
Moviepy - Done !
Moviepy - video ready output_videos\solidWhiteRight_output.mp4
CPU times: total: 6.66 s
Wall time: 7.91 s
<IPython.core.display.HTML  object>
%time process_video('solidYellowLeft.mp4',
'solidYellowLeft_output.mp4')
HTML("""
<video width="960" height="540" controls>
 <source src="{0}">
</video>
```

""".format("output_videos\solidYellowLeft_output.mp4"))

Moviepy - Building video output_videos\solidYellowLeft_output.mp4.

Moviepy - Writing video output_videos\solidYellowLeft_output.mp4

Moviepy - Done !

Moviepy - video ready output_videos\solidYellowLeft_output.mp4

CPU times: total: 27.5 s

Wall time: 27.7 s

<IPython.core.display.HTML object>

%time process_video('challenge.mp4', 'challenge_output.mp4')

HTML("""

<video width="960" height="540" controls>

<source src="{0}">

</video>

""".format("output_videos\challenge_output.mp4"))

Moviepy - Building video output_videos\challenge_output.mp4.

Moviepy - Writing video output_videos\challenge_output.mp4

Moviepy - Done !

Moviepy - video ready output_videos\challenge_output.mp4

CPU times: total: 14.6 s

Wall time: 15.5 s

<IPython.core.display.HTML object8. CONCLUSION & FUTURE SCOPE :

7.RESULTS / SCREENSHOTS :

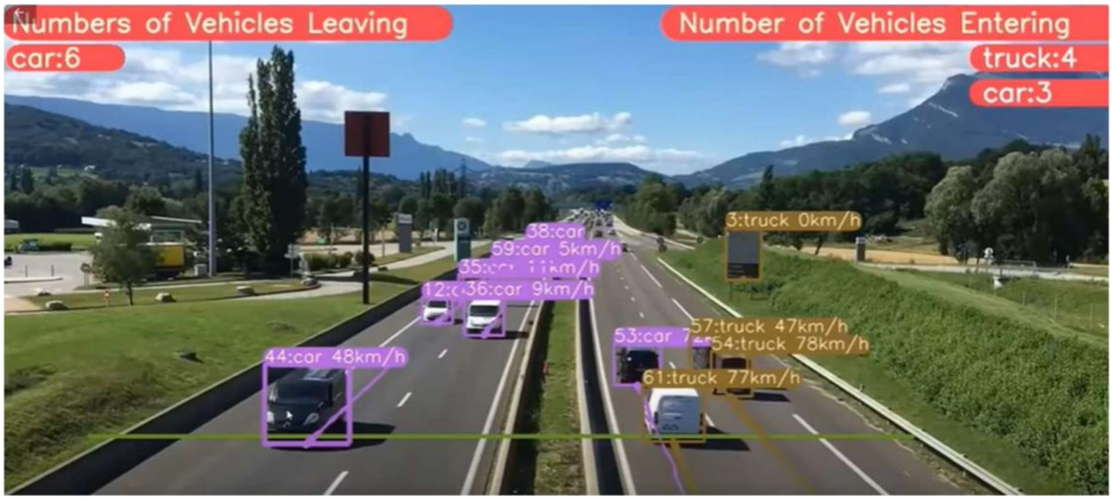

FIG : 7.1 Counting the vehicles and estimation of speed

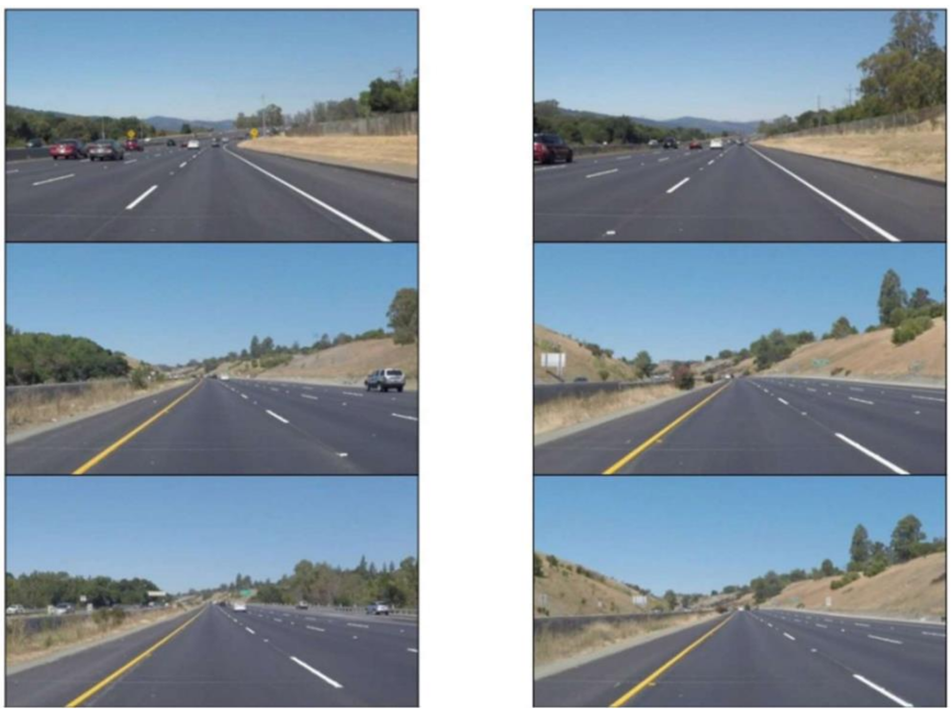

FIG : 7.2 Loading The Test Images

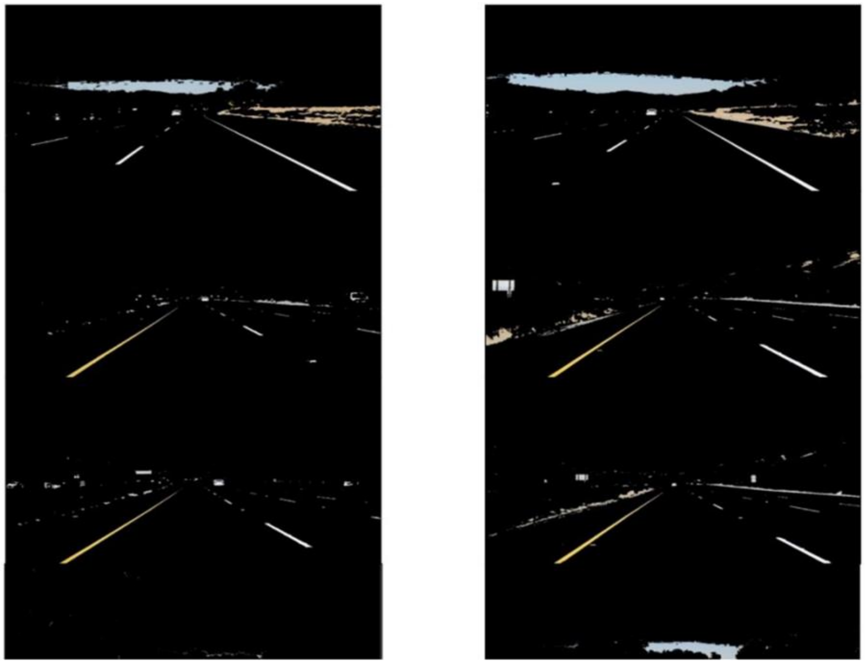

FIG :7.3 HSV Color Space Output

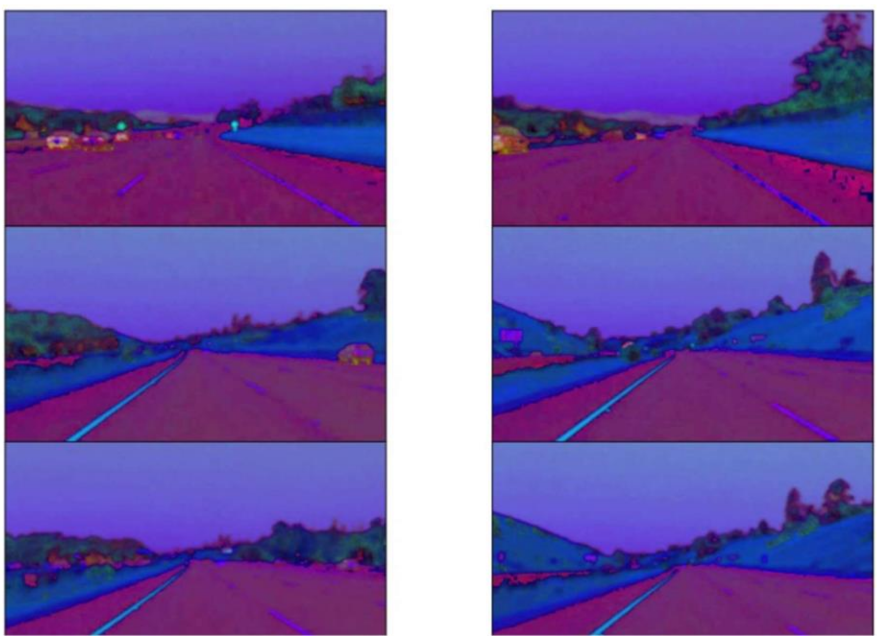

FIG : 7.4 Input Image To HSV

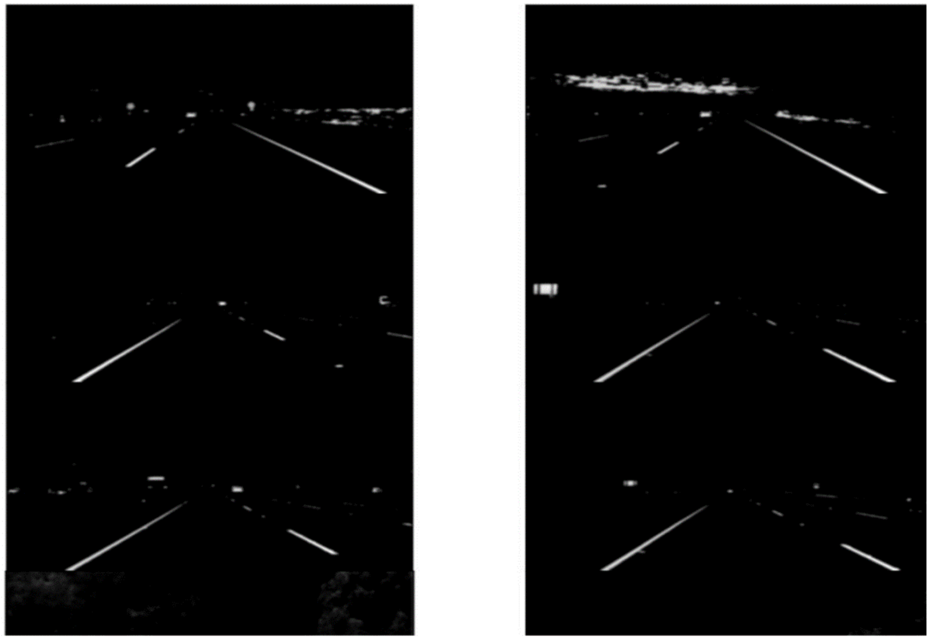

FIG : 7.5 Canny Edge Detection Output

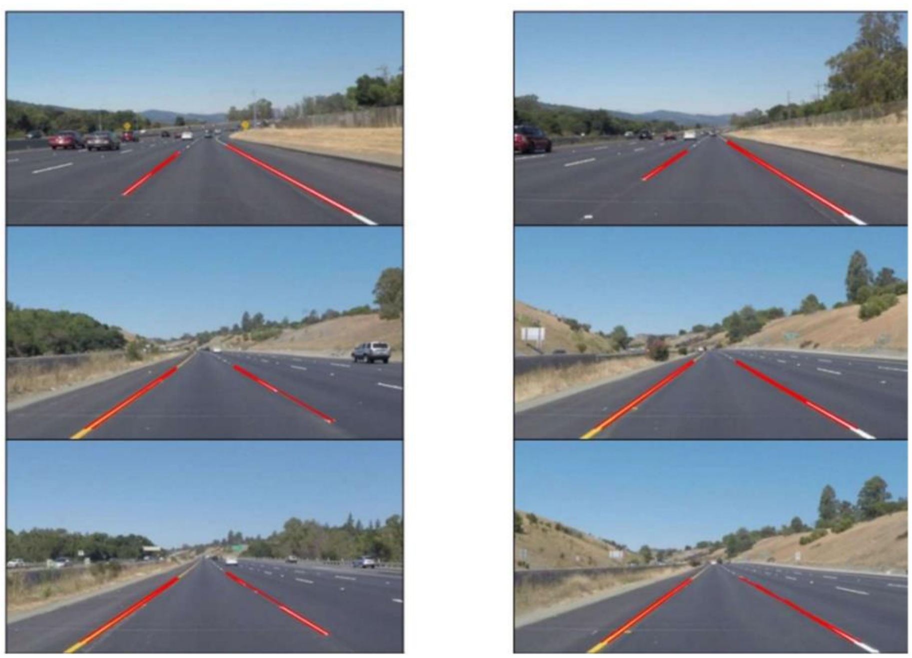

FIG : 7.6 Hough Transform Output

8. CONCLUSION & FUTURE SCOPE :

In conclusion, the Lane and Vehicle Detection System for Automatic Cars presents a comprehensive solution to the critical need for accurate and reliable detection capabilities in autonomous driving and advanced driver assistance systems (ADAS). By leveraging sophisticated computer vision algorithms and deep learning techniques, the system enables real-time detection and tracking of lanes and vehicles on the road, enhancing safety, precision, and decision-making capabilities.

The system's architecture incorporates advanced features such as real-time lane detection, vehicle detection and tracking, seamless integration with autonomous driving systems, high accuracy and reliability, adaptability to various environmental conditions, low latency performance, scalability, and efficiency. These features collectively contribute to an enhanced autonomous driving experience and improved road safety.

Advancements in lane and vehicle detection systems aim to address diverse challenges in modern transportation. Enhanced Environmental Adaptability improves detection performance in challenging conditions like rain and fog, ensuring consistent reliability for road safety. Integration with V2X Communication enhances connectivity between vehicles and infrastructure, fostering a comprehensive understanding of the environment for safer navigation. Behavior Prediction and Intent Recognition enable proactive decision-making by anticipating the behavior of nearby vehicles and pedestrians, contributing to smoother traffic flow. Multi-modal Sensor Fusion integrates various sensor types to enhance detection accuracy, while Real-time Map Updates ensure navigation systems remain current. Continual Learning enables the system to adapt and improve performance over time, ensuring effectiveness in evolving driving environments.

9. References :

1) Al-Rashed, Mohammad A., Anwaar Ulhaq, Usama Owais, and Abdul Salam. "A Survey of Vision-Based Lane Recognition Algorithms." Journal of Physics: Conference Series 2547, no. 1 (2020): 012015. https://doi.org/10.1088/1742-6596/2547/1/012015.

2) Zou, Zhengxia, et al. "Deep Learning for Object Detection: A Survey." arXiv (2020): 1907.09408. https://arxiv.org/abs/1907.09408.

3) Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 779-788. Las Vegas, NV: IEEE, 2016. https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf.

4) Kaur, Lovedeep, Manish Gupta, and Devanjali Dey. "Real-Time Lane Detection System Using Hough Transform Technique." Proceedings of 2014 Fourth International Conference on Communication Systems and Network Technologies, 871-874. Bhopal, India: IEEE, 2014. https://ieeexplore.ieee.org/document/5512220.

5) Mohan, Akshay. "Lane Detection and Tracking in Challenging Scenarios." Master's thesis, University of California, San Diego, 2018. https://escholarship.org/content/qt50n0c8cg/qt50n0c8cg.pdf?t=lnpy3h.

6) Yu, Fisher, et al. "BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling." Nuscenes. https://www.nuscenes.org/nuscenes.

7) Geiger, Andreas, Philip Lenz, and Raquel Urtasun. "Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite." CVPR 2012. https://www.cvlibs.net/datasets/kitt/.

8) OpenCV. "Open Source Computer Vision Library." https://opencv.org/.

9) PyTorch. "An Open Source Machine Learning Framework That Accelerates the Path from Research Prototyping to Production Deployment." https://pytorch.org/.