Minor Project Report

Indian Institute of Technology Indore

Lane Detection using Deep Learning

Course Instructor: Dr. Aruna Tiwari

April 20, 2024

Team Members:
Batchu Sonika (210001009)
Gorantla Anuksha (210001018)
Devani Sravya (210001013)

Contents

1 Introduction **2**

1.1 Problem Statement 2

1.2 Necessity of the problem 2

2 Datasets **3**

2.1 Dataset Selection 3

2.2 Dataset Preprocessing 3

3 Model and Environment **7**

3.1 Development Environment 7

3.2 Software Packages and Libraries 7

3.3 Model Architecture 7

4 Implementation **9**

5 Results **9**

5.1 Outputs 9

5.2 Metrics 12

6 Conclusion **15**

7 Acknowledgment **15**

8 References **15**

1 Introduction

1.1 Problem Statement

The project aims to develop an efficient and accurate lane detection system using neural networks to enhance the safety and autonomy of vehicles by providing real-time awareness of lane boundaries on roads. The main challenge is to accommodate varying environmental conditions, such as different lighting conditions, road surfaces, and diverse traffic scenarios, to ensure robust performance across different driving scenarios. The model should be capable of accurately detecting lane markings, including solid, dashed, and curved lines.

1.2 Necessity of the problem

Lane detection is a critical component in various applications, particularly in the realm of autonomous vehicles and advanced driver assistance systems (ADAS).

One of the primary motivations for lane detection is to enhance road safety.Lane detection is crucial for autonomous vehicles to navigate roads safely.Lane detection is a fundamental component of many driver assistance systems, such as lane departure warning (LDW), lane-keeping assist (LKA) and Head-Up Display (HUD). These systems alert drivers when they unintentionally drift out of their lane and can even intervene by providing steering assistance to keep the vehicle within the lane.

By defining the problem accurately, researchers and engineers can tailor solutions to meet the needs of various domains, ultimately contributing to safer roads, improved traffic management, and enhanced user experiences across different applications.

Lane detection can be applied in industrial settings for guiding automated vehicles, robots, and machinery along predefined paths or lanes within warehouses, factories, and logistics centers.It can help detect and track vehicles within lanes, identify traffic violations, and enhance security measures in high-traffic areas.

2 Datasets

2.1 Dataset Selection

After conducting extensive research on various lane detection datasets commonly utilized in academic papers and research, including TU Simple, CULanes, and others, we carefully evaluated each dataset's suitability for our project's requirements. Gathering all the datasets and papers, we created a excel sheet for easy comparison. we identified CurveLanes as the most comprehensive dataset that met our criteria.

The dataset "CurveLanes" which was first introduced in the paper "CurveLane-NAS: Unifying Lane-Sensitive Architecture Search and Adaptive Point Blending". CurveLanes is a new benchmark lane detection dataset with 150K lanes images for difficult scenarios such as curves and multi-lanes in traffic lane detection. It is collected in real urban and highway scenarios in multiple cities in China.

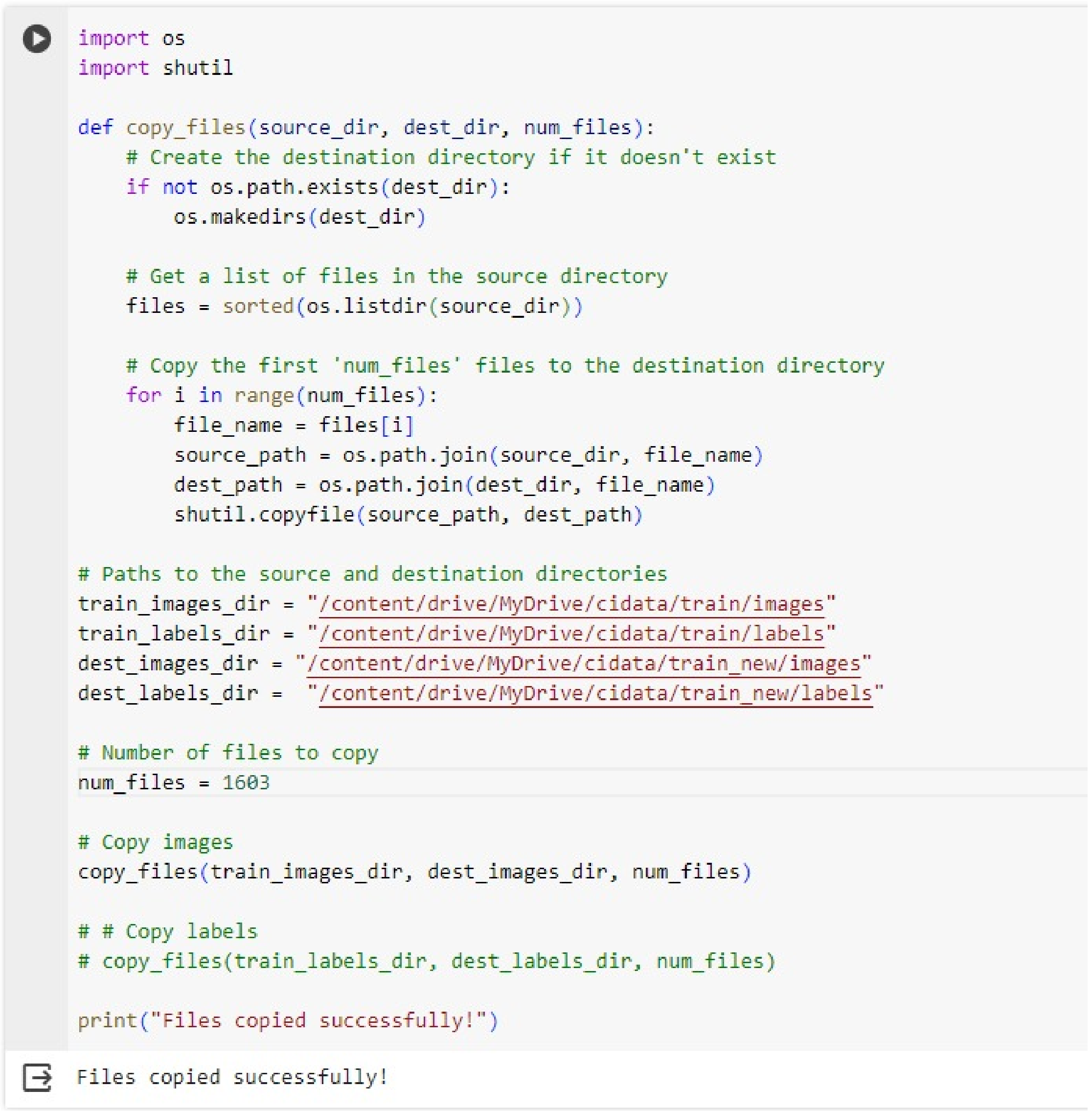
It features pictures of highways in various weather conditions, including cloudy, sunny, and rainy ones, as well as varied lighting. This collection also contains several sorts of curve lanes, such as S-curves and Y-curves.It also has photographs taken from inside the automobile.

2.2 Dataset Preprocessing

In order to facilitate processing within Google Colab, we aimed to reduce the size of the CurveLanes dataset, which originally amounted to 68 gigabytes. From this extensive dataset, we selected 1.6k images for training purposes, along with 300 images each for both validation and testing.

Due to the restricted RAM and GPU usage on colab, inorder to increase the number of training epochs to improve the accuracy we again split the dataset into half i.e 800 images in training dataset and 150 images for both validation and test datasets.

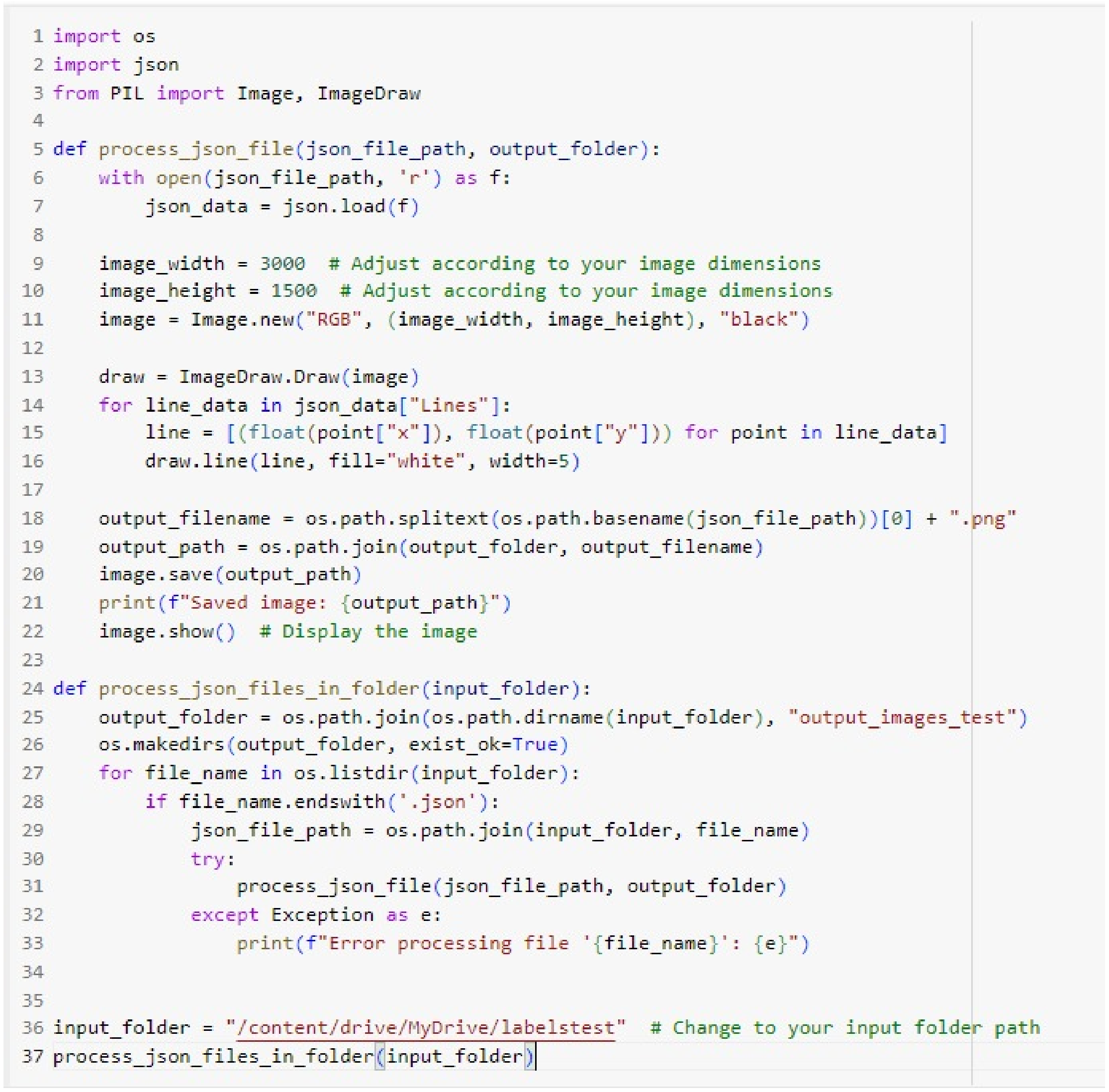
We trained our model for both 8 epochs and 16 epochs to see the variations in the metrics.



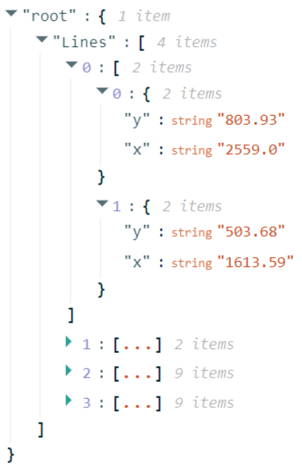
```python
import os
import shutil

def copy_files(source_dir, dest_dir, num_files):
    # Create the destination directory if it doesn't exist
    if not os.path.exists(dest_dir):
        os.makedirs(dest_dir)

    # Get a list of files in the source directory
    files = sorted(os.listdir(source_dir))

    # Copy the first 'num_files' files to the destination directory
    for i in range(num_files):
        file_name = files[i]
        source_path = os.path.join(source_dir, file_name)
        dest_path = os.path.join(dest_dir, file_name)
        shutil.copyfile(source_path, dest_path)

# Paths to the source and destination directories
train_images_dir = "/content/drive/MyDrive/cidata/train/images"
train_labels_dir = "/content/drive/MyDrive/cidata/train/labels"
dest_images_dir = "/content/drive/MyDrive/cidata/train_new/images"
dest_labels_dir =  "/content/drive/MyDrive/cidata/train_new/labels"

# Number of files to copy
num_files = 1603

# Copy images
copy_files(train_images_dir, dest_images_dir, num_files)

# # Copy labels
# copy_files(train_labels_dir, dest_labels_dir, num_files)

print("Files copied successfully!")
```

```
Files copied successfully!
```

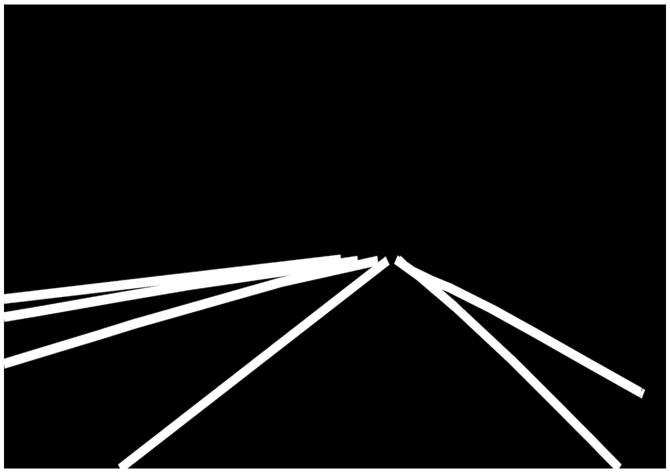
The dataset provided labels in JSON format, containing information about lane markings and other relevant annotations. To streamline our workflow, we converted these JSON-formatted labels into corresponding images.



```python
import os
import json
from PIL import Image, ImageDraw

def process_json_file(json_file_path, output_folder):
    with open(json_file_path, 'r') as f:
        json_data = json.load(f)

    image_width = 3000  # Adjust according to your image dimensions
    image_height = 1500  # Adjust according to your image dimensions
    image = Image.new("RGB", (image_width, image_height), "black")

    draw = ImageDraw.Draw(image)
    for line_data in json_data["Lines"]:
        line = [(float(point["x"]), float(point["y"])) for point in line_data]
        draw.line(line, fill="white", width=5)

    output_filename = os.path.splitext(os.path.basename(json_file_path))[0] + ".png"
    output_path = os.path.join(output_folder, output_filename)
    image.save(output_path)
    print(f"Saved image: {output_path}")
    image.show()  # Display the image

def process_json_files_in_folder(input_folder):
    output_folder = os.path.join(os.path.dirname(input_folder), "output_images_test")
    os.makedirs(output_folder, exist_ok=True)
    for file_name in os.listdir(input_folder):
        if file_name.endswith('.json'):
            json_file_path = os.path.join(input_folder, file_name)
            try:
                process_json_file(json_file_path, output_folder)
            except Exception as e:
                print(f"Error processing file '{file_name}': {e}")


input_folder = "/content/drive/MyDrive/labelstest"  # Change to your input folder path
process_json_files_in_folder(input_folder)
```

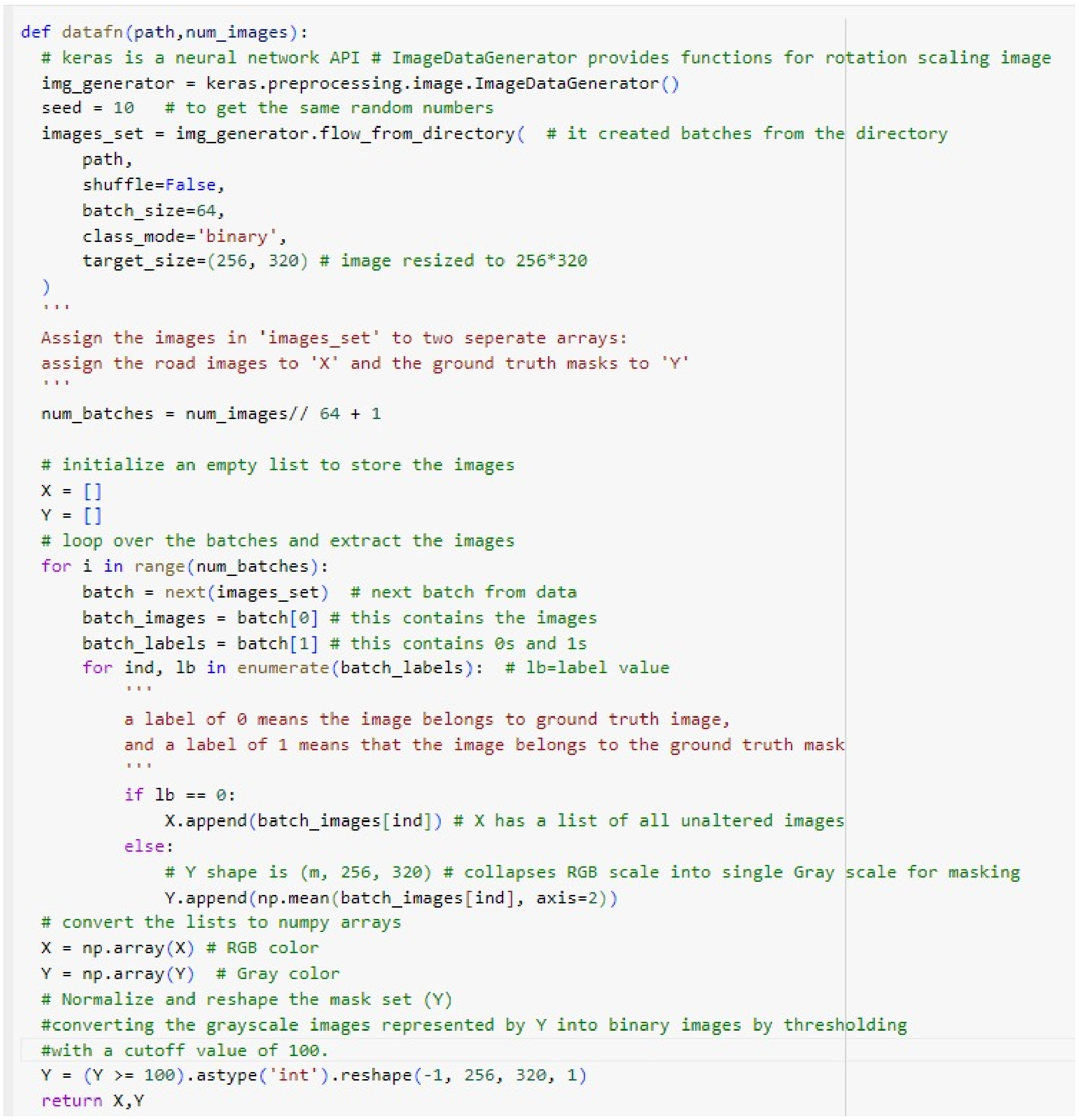


```
▼"root" : { 1 item
  ▼"Lines" : [ 4 items
    ▼0 : [ 2 items
      ▼0 : { 2 items
        "y" : string "803.93"
        "x" : string "2559.0"
      }
      ▼1 : { 2 items
        "y" : string "503.68"
        "x" : string "1613.59"
      }
    ]
    ▶1 : [...] 2 items
    ▶2 : [...] 9 items
    ▶3 : [...] 9 items
  ]
}
```



Furthermore, to ensure uniformity and optimize computational efficiency, we resized all selected images to a standardized resolution of 256 pixels in height and 320 pixels in width.

Additionally, considering that lane detection typically involves grayscale analysis, we converted the RGB-formatted labels into grayscale images. This conversion simplifies subsequent processing steps while retaining the essential lane detection information.



```
def datafn(path,num_images):
  # keras is a neural network API # ImageDataGenerator provides functions for rotation scaling image
  img_generator = keras.preprocessing.image.ImageDataGenerator()
  seed = 10   # to get the same random numbers
  images_set = img_generator.flow_from_directory(  # it created batches from the directory
      path,
      shuffle=False,
      batch_size=64,
      class_mode='binary',
      target_size=(256, 320) # image resized to 256*320
  )
  '''
  Assign the images in 'images_set' to two seperate arrays:
  assign the road images to 'X' and the ground truth masks to 'Y'
  '''
  num_batches = num_images// 64 + 1

  # initialize an empty list to store the images
  X = []
  Y = []
  # loop over the batches and extract the images
  for i in range(num_batches):
      batch = next(images_set)  # next batch from data
      batch_images = batch[0] # this contains the images
      batch_labels = batch[1] # this contains 0s and 1s
      for ind, lb in enumerate(batch_labels):  # lb=label value
          '''
          a label of 0 means the image belongs to ground truth image,
          and a label of 1 means that the image belongs to the ground truth mask
          '''
          if lb == 0:
              X.append(batch_images[ind]) # X has a list of all unaltered images
          else:
              # Y shape is (m, 256, 320) # collapses RGB scale into single Gray scale for masking
              Y.append(np.mean(batch_images[ind], axis=2))
  # convert the lists to numpy arrays
  X = np.array(X) # RGB color
  Y = np.array(Y)  # Gray color
  # Normalize and reshape the mask set (Y)
  #converting the grayscale images represented by Y into binary images by thresholding
  #with a cutoff value of 100.
  Y = (Y >= 100).astype('int').reshape(-1, 256, 320, 1)
  return X,Y
```

3 Model and Environment

3.1 Development Environment

We used Google Colab as the primary development environment for coding and implementation. We chose colab because of the various facilities that it provides.

One of the key advantages of using Google Colab was its integration with Google Drive, allowing seamless access to datasets and saved models stored in the cloud. The availability of free GPU and TPU resources on Google Colab significantly accelerated the training process, enabling faster experimentation with deep learning models. Google Colab's collaborative features facilitated teamwork by allowing multiple users to work on the same notebook simultaneously, enhancing productivity and collaboration.

3.2 Software Packages and Libraries

For our project, we utilized a variety of Python packages and libraries, including NumPy for numerical computing, Matplotlib for data visualization, Pandas for data manipulation and analysis, OpenCV for image processing, TensorFlow with Keras for deep learning model development, and scikit-learn for evaluating model performance.

3.3 Model Architecture

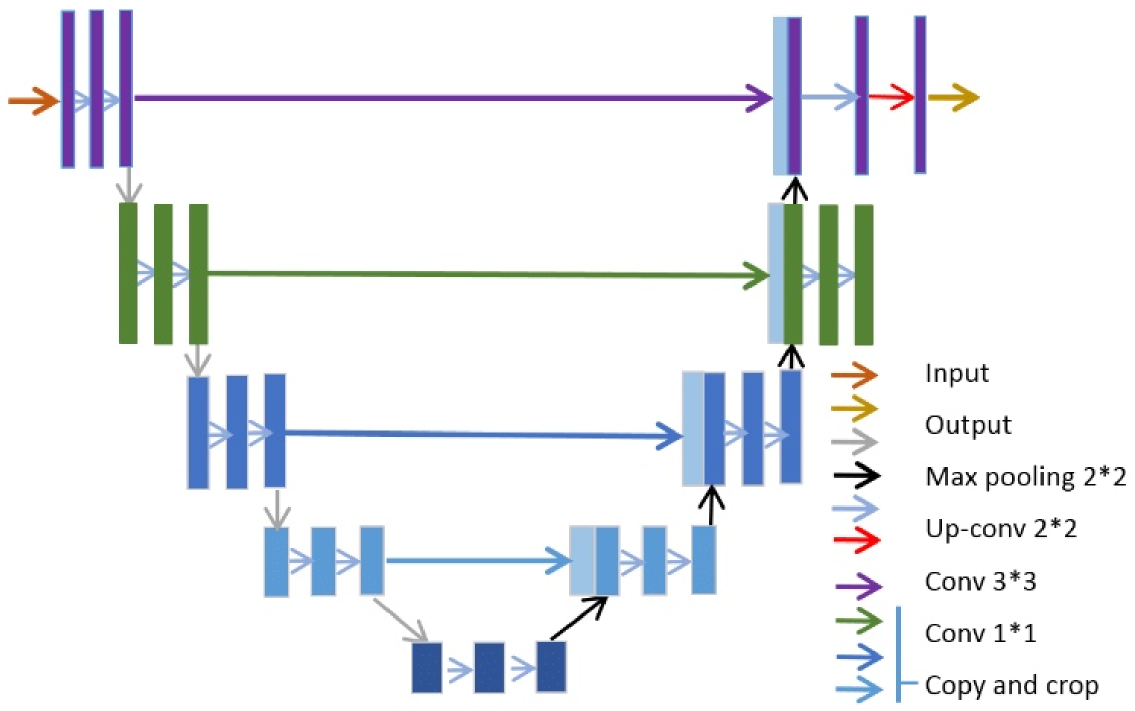
UNet: A Deep Learning Architecture for Semantic Segmentation.

UNet is a convolutional neural network architecture designed for semantic segmentation tasks. The UNet architecture comprises a contracting path that captures context information through a series of convolutional and pooling layers. These layers gradually reduce the spatial dimensions of the input image while extracting features at multiple scales. Each convolutional block consists of two convolutional layers followed by rectified linear unit (ReLU) activation functions, which introduce non-linearity into the model.

At the bottom of the network lies a bottleneck, where the spatial dimensions are reduced to a minimum. This bottleneck helps in capturing rich feature representations by utilizing a large number of filters.

Following the bottleneck, the UNet architecture includes an expansive path responsible for precise localization.

The final layer of the UNet architecture is a convolutional layer with a sigmoid activation function, producing a binary segmentation mask indicating the presence or absence of objects in each pixel of the input image.



Why Unet?

1.**Robustness to Limited Data:** UNet's architecture, with its skip connections and data augmentation techniques, helps in training robust models even with limited annotated data. This is advantageous in scenarios where acquiring large-scale annotated datasets is challenging or expensive.

2.**Efficient Training and Inference:** Despite its deep architecture, UNet can be trained efficiently, especially with modern deep learning frameworks and hardware accelerators. Inference time is also relatively fast, making it suitable for real-time or near-real-time applications.

3. **Interpretable Results:** UNet produces segmentation masks at the pixel level, providing interpretable results that can be easily understood and analyzed. This is particularly beneficial in applications where detailed understanding of image regions is required.

4. **Preservation of Spatial Information:** The skip connections in UNet facilitate the direct flow of information from early layers to later layers in the expansive path. This helps mitigate information loss during downsampling in the contracting path, preserving spatial details crucial for accurate segmentation.

UNet stands as a powerful architecture for semantic segmentation tasks, offering a balance between context understanding and precise localization. Its modular design and simplicity make it a popular choice for image segmentation challenges.

4 Implementation

To access the code implementation of our UNet model in Google Colab:

1.6k training images and 300 validation and testing images with 6 epochs :Link.

800 training images and 150 validation and testing images with 8 epochs :Link.

800 training images and 150 validation and testing images with 16 epochs :Link.

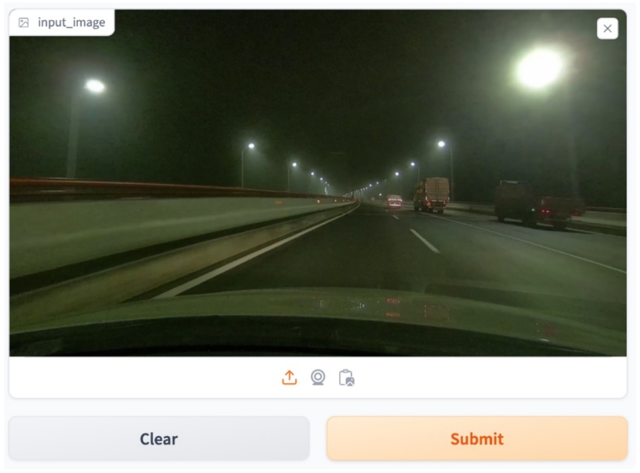
Inferencing code with user interfaceLink.

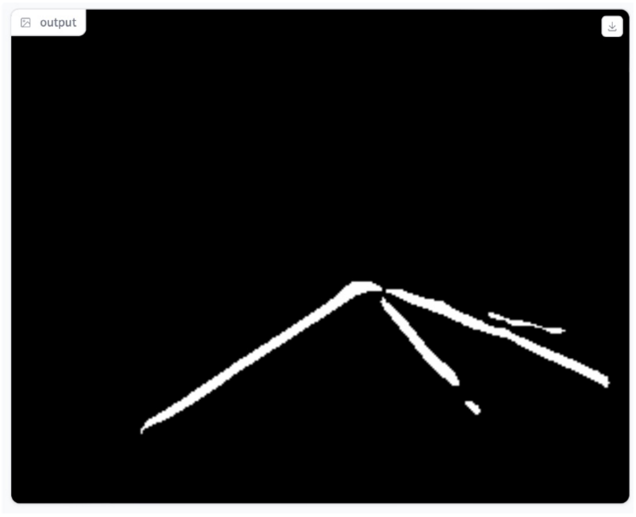
5 Results

5.1 Outputs

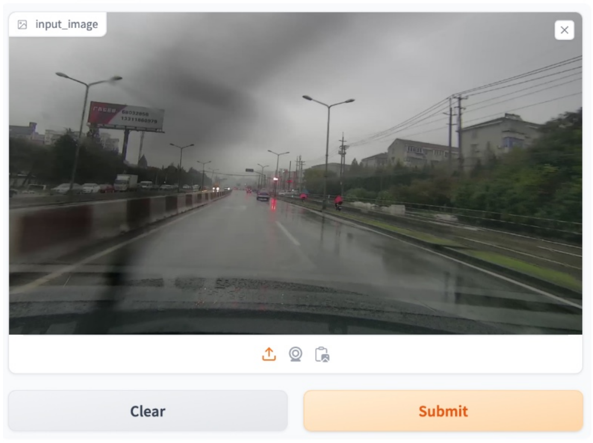
Below are some of the test outputs that our model predicted.

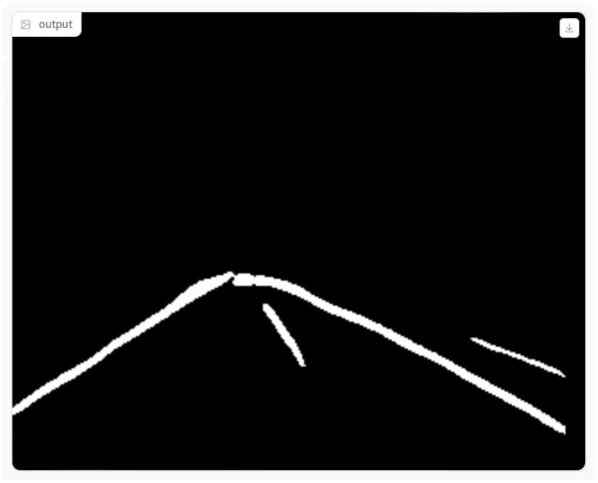
Night:



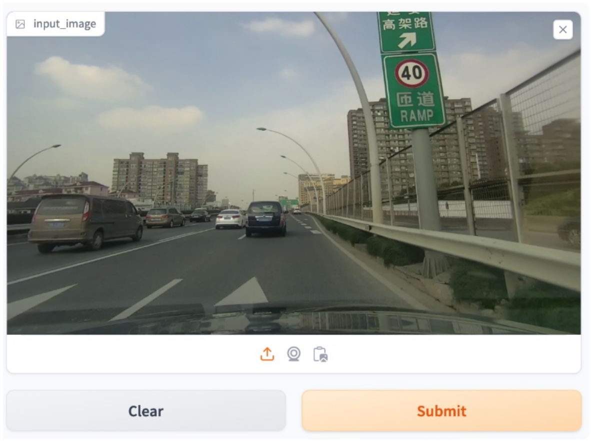


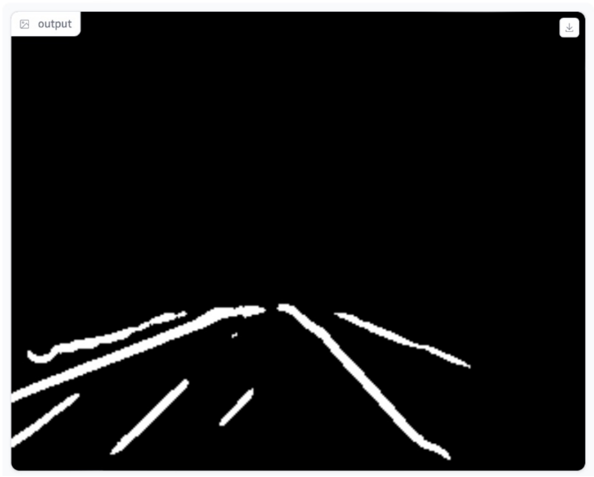
With windshield:



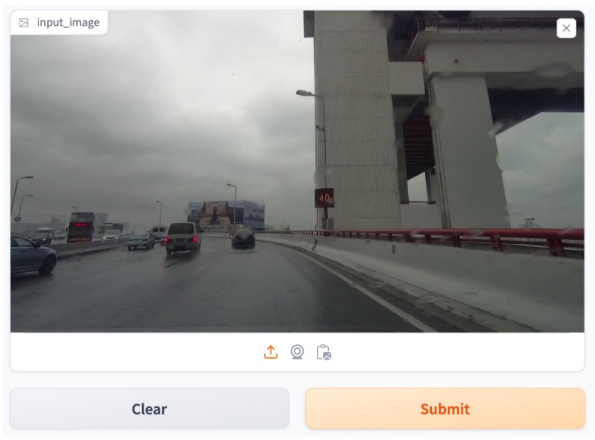


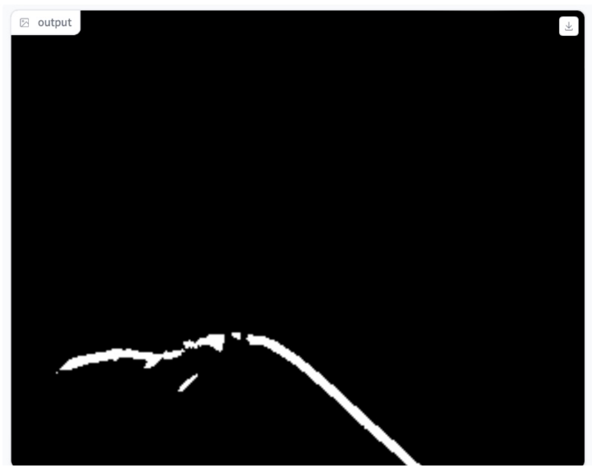
Sunny:



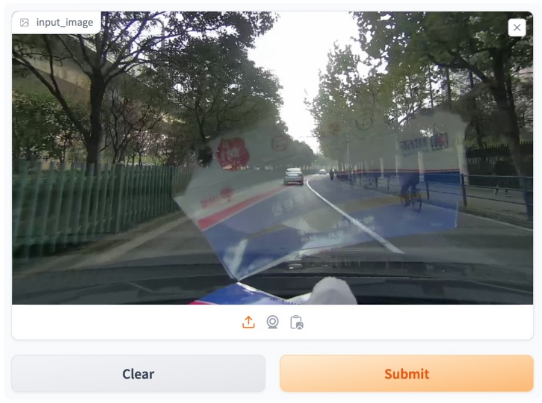


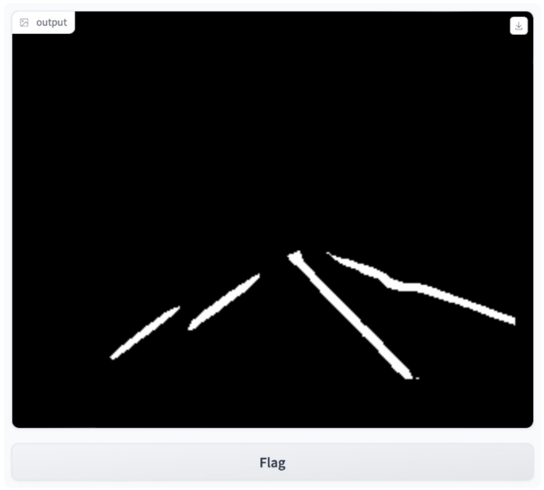
Cloudy:



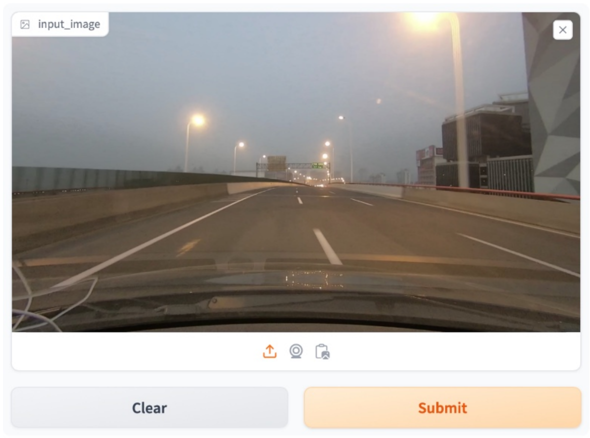


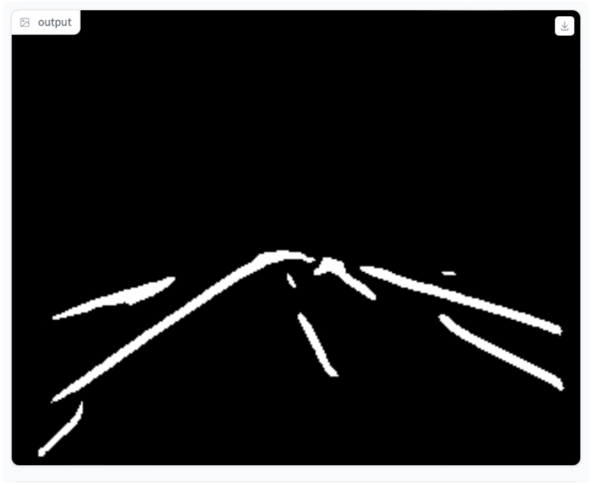
With reflection:



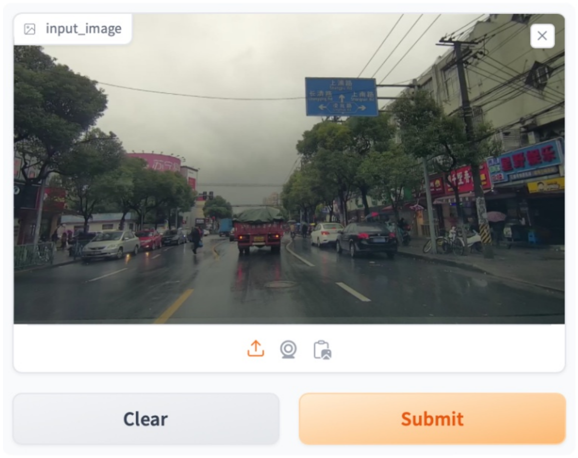


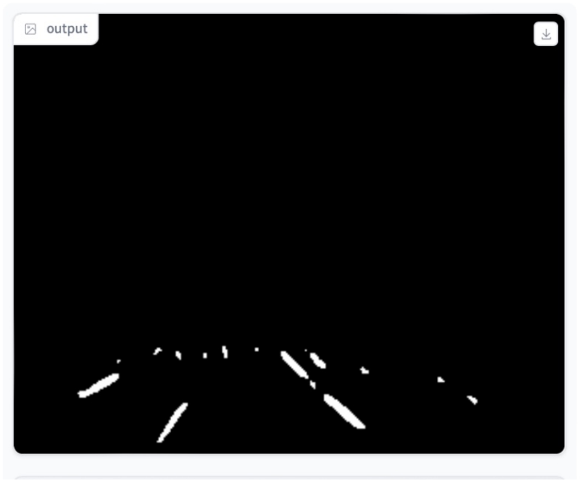
Evening:





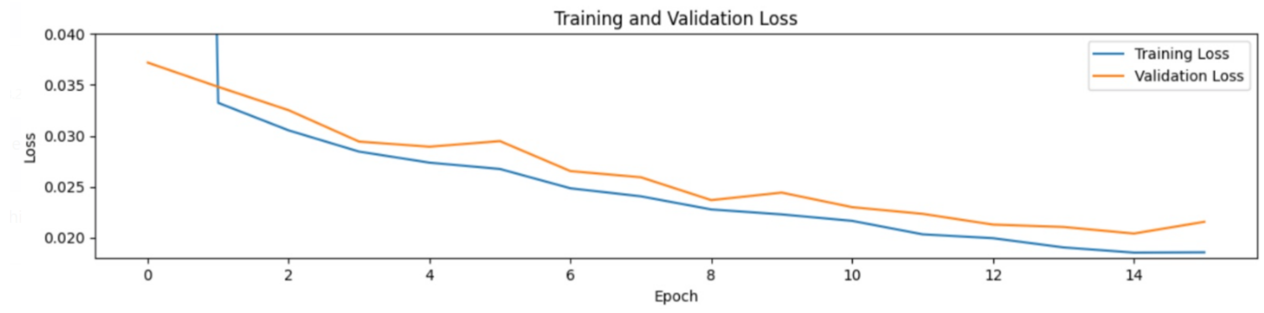
Rainy:



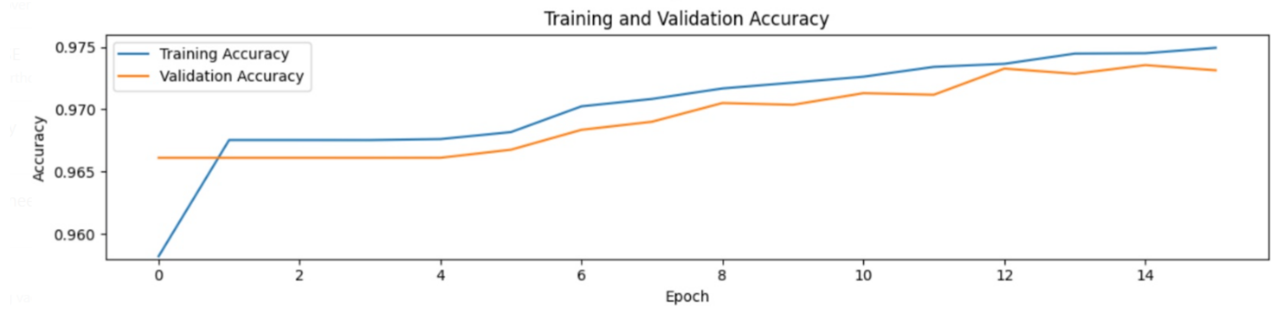


5.2 Metrics

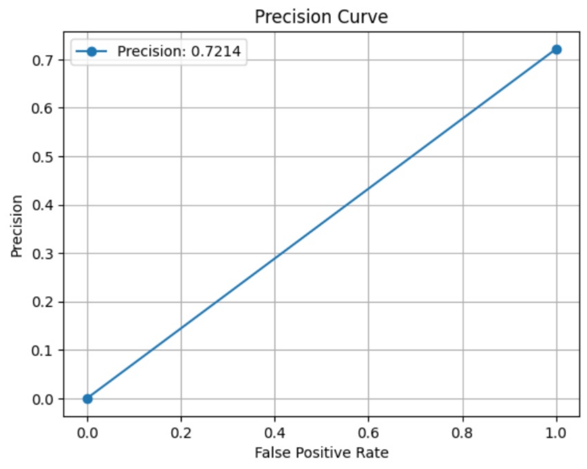
Mean Squared Error (MSE) loss: Measures the squared difference between the predicted lane positions and the ground truth positions.



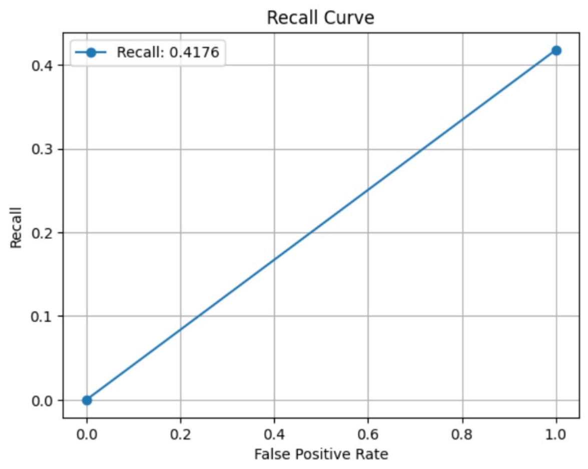
Accuracy: Accuracy, in the context of lane detection, measures the proportion of correctly classified pixels (both lane and non-lane) out of the total number of pixels in the image.



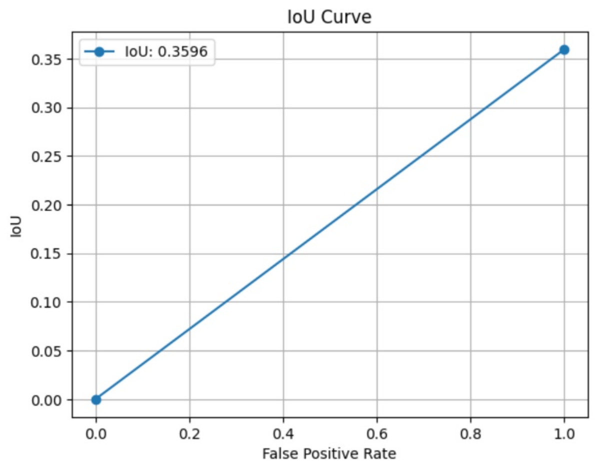
Precision: High precision is generally desirable in lane detection to ensure that the detected lanes are accurate and reliable. False positive lane detections could lead to erroneous decisions by an autonomous vehicle or misinterpretations by a driver assistance system, potentially causing safety hazards.



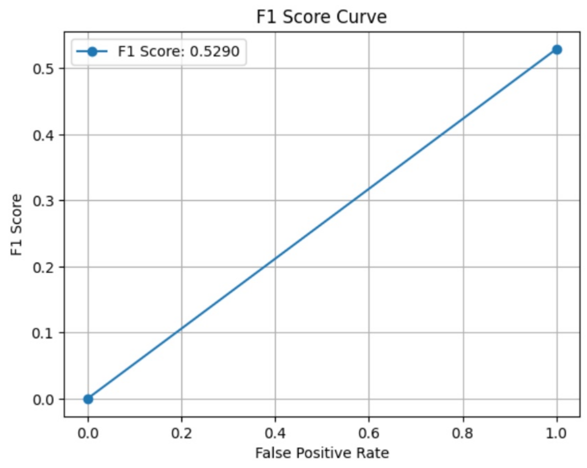
Recall: While high precision is crucial, recall should not be sacrificed to the extent that significant portions of the lane markings are missed. It's important to detect as much of the actual lane markings as possible to provide comprehensive guidance for autonomous vehicles or driver assistance systems.



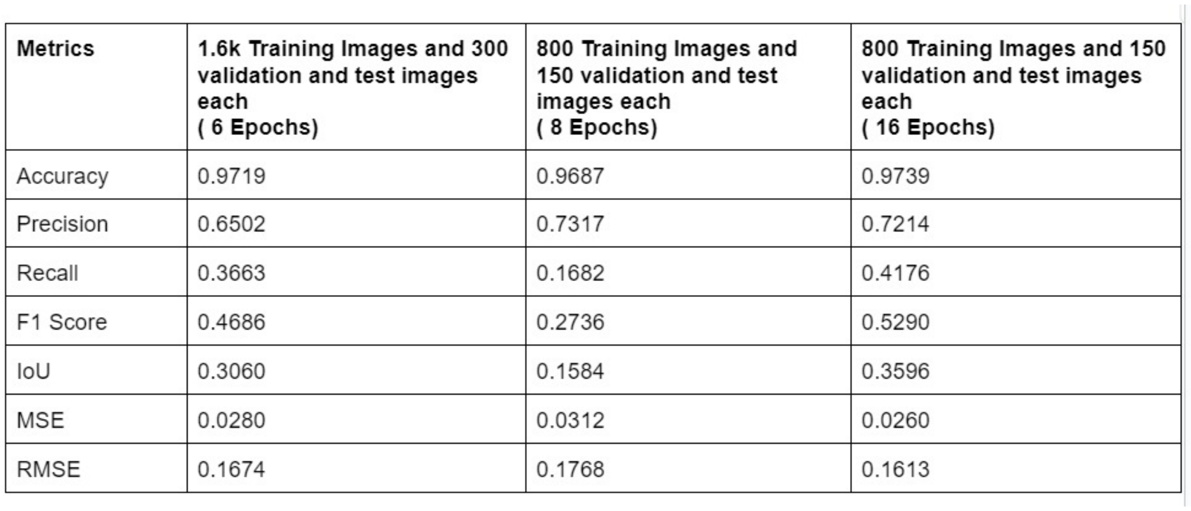
IoU: It is a metric commonly used to evaluate the accuracy of segmentation tasks, such as delineating lane markings on roads. IoU measures the overlap between the predicted lane markings and the ground truth annotations. It quantifies how well the predicted lane markings align with the actual lane markings.



F1 score: F1 score is the harmonic mean of precision and recall, providing a single metric that balances both false positives and false negatives.



Variation of metrics for varying dataset sizes and epochs.



Metrics	**1.6k Training Images and 300 validation and test images each (6 Epochs)**	**800 Training Images and 150 validation and test images each (8 Epochs)**	**800 Training Images and 150 validation and test images each (16 Epochs)**
Accuracy	0.9719	0.9687	0.9739
Precision	0.6502	0.7317	0.7214
Recall	0.3663	0.1682	0.4176
F1 Score	0.4686	0.2736	0.5290
IoU	0.3060	0.1584	0.3596
MSE	0.0280	0.0312	0.0260
RMSE	0.1674	0.1768	0.1613

From the above table, we can observe that if we increase the dataset size or increase the number of epochs we can see improvement in metrics.

6 Conclusion

In conclusion, the training iterations conducted on varying datasets and epochs have provided valuable insights into the performance improvements of our lane detection model. Beginning with a dataset of 1.2 GB size and 6 epochs, we observed promising initial results, indicating the model's capability to learn and adapt to lane detection tasks.

Subsequent experiments with datasets of reduced sizes but increased epochs demonstrated a trend towards enhanced performance. Specifically, training on a dataset half the size but for 8 epochs showed further improvement, while extending training to 16 epochs on the same dataset yielded even more notable enhancements. These findings strongly suggest that continued training on larger datasets for more epochs could lead to significant advancements in model accuracy and robustness.

However, despite these promising indications, the pursuit of further improvements was constrained by limitations in RAM and GPU utilizations, particularly evident in the Google Colab environment. These resource constraints hindered our ability to scale up the training process to larger datasets and extended epochs.

7 Acknowledgment

We express our gratitude to Dr. Aruna Tiwari for delivering the course on COMPUTATIONAL INTELLIGENCE with exceptional skill and expertise. As a computer science student, this course is essential for any professional route we aspire to pursue, and we are delighted to have learned it with such a high degree of clarity. Furthermore, this project, along with several other lab assignments, has not only improved our ability to apply theoretical knowledge but has also provided us with practical coding experience through real-world task-oriented questions. We express our appreciation to our lecturer and teaching assistants for ensuring the seamless progress of our education.

8 References

CurveLane-NAS: Unifying Lane-Sensitive Architecture Search and Adaptive Point Blending.

Lane Road Segmentation Based on Improved UNet Architecture for Autonomous Driving.

Dataset.