A REPORT ON

AUTONOMOUS VEHICLE DRIVING MODEL USING DEEP LEARNING

BY

Sundar Matu - 2022A7PS0165P

AT

CDAC - CINE

A Practice School-I Station of

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

Under the guidance of

MR. FARUKH

21st JUNE 2024

INTRODUCTION:

Introduction

The advent of self-driving car technology marks a transformative era in the automotive and transportation industries. Leveraging advancements in deep learning, self-driving cars aim to enhance road safety, reduce traffic congestion, and provide increased mobility for those unable to drive. This project focuses on the development and simulation of self-driving car systems using deep learning techniques.

Autonomous Vehicles

Autonomous vehicles (AVs) are vehicles capable of sensing their environment and navigating without human input. AVs use artificial intelligence (AI) to perceive their surroundings and make appropriate driving decisions. Through a combination of sensors, such as cameras, lidar, radar, and ultrasonic sensors, AVs gather comprehensive data about their environment. Deep learning algorithms then process this data to identify objects, predict their movements, and make driving decisions that emulate human driving behavior.

Computer Vision

Computer vision enables machines to interpret and understand visual information from the world, using techniques like object detection, image classification, and scene understanding to analyze and make decisions based on image data. In the context of self-driving cars, computer vision plays a crucial role in identifying road signs, traffic signals, pedestrians, other vehicles, and obstacles. By accurately interpreting visual data, the autonomous system can navigate safely and efficiently.

Image Processing

Image processing involves techniques like edge detection, color space transformations, filtering, enhancement, and segmentation to improve image quality and extract useful information for tasks such as object detection. In self-driving car systems, image processing enhances the raw data captured by cameras and other sensors, making it more suitable for analysis by deep learning models. For instance, edge detection can help in identifying lane markings, while segmentation can differentiate between various objects on the road.

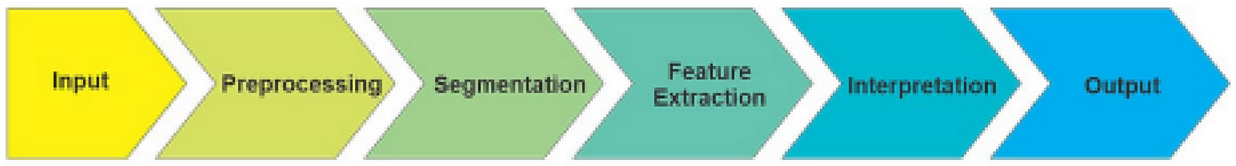


Fig. Steps involved in Image Processing

Basic Objective

The project aims to develop a reliable autonomous driving model that accurately perceives its environment, determines its position, plans optimal routes, and executes safe driving actions using advanced computer vision and image processing techniques. This involves integrating various components to create a cohesive system that mimics human driving behavior and decision-making processes.

Autonomous Driving Technology

Autonomous driving technology involves four critical steps:

1. **Perception (Understanding the environment)**: This step involves collecting data from sensors such as cameras, lidar, radar, and ultrasonic sensors to understand the surrounding environment. Advanced computer vision and image processing techniques are used to detect and classify objects, recognize traffic signs and signals, and interpret road conditions.
2. **Localization (Determining the vehicle's position and its state)**: Localization ensures that the vehicle accurately knows its position and orientation within its environment. This is achieved using a combination of GPS data, sensor fusion, and mapping technologies to maintain precise awareness of the vehicle's location.
3. **Planning (Deciding the plan of action)**: Planning involves generating an optimal route for the vehicle to reach its destination safely and efficiently. This includes path planning, trajectory generation, and decision-making algorithms that take into account dynamic obstacles, traffic rules, and road conditions.
4. **Control (Executing the plans)**: The final step is control, where the vehicle executes the planned actions. This involves steering, accelerating, braking, and maneuvering the vehicle based on the decisions made during the planning phase. Control systems must be highly responsive and accurate to ensure safe navigation.

Key Features

The key features of an autonomous vehicle include:

- **Object Detection and Classification**: Identifying and categorizing various objects such as vehicles, pedestrians, cyclists, and obstacles to ensure safe navigation.
- **Lane Detection**: Recognizing and following lane markings to stay within the correct lane and make safe lane changes.
- **Scene Understanding**: Interpreting the overall driving environment, including road layout, traffic conditions, and potential hazards.
- **Traffic Light Detection**: Accurately detecting and responding to traffic lights to obey traffic signals and ensure smooth traffic flow.

Project Functions

The project functions by training a convolutional neural network (CNN) to process raw pixel data captured from a single front-facing camera mounted on the vehicle. This CNN learns to analyze the visual information in real-time and map it directly to steering commands, allowing the vehicle to navigate autonomously by understanding the road layout, detecting obstacles, and making appropriate steering adjustments based on the visual input.

What is CNN?

A Convolutional Neural Network (CNN) is a specialized deep learning model designed for processing visual data such as images. CNNs are particularly well-suited for tasks like image classification, object detection, and other computer vision applications due to their ability to automatically learn and extract features from raw image data.

Key Components of a CNN

1. **Convolutional Layers**: These layers apply convolution operations to the input image, using filters (also called kernels) to detect various features. Each filter slides over the input image and performs element-wise multiplications followed by summation.

2. **Activation Functions**: Non-linear activation functions such as ReLU (Rectified Linear Unit) are applied to the feature maps produced by convolutional layers. These functions introduce non-linearity into the model, enabling it to learn more complex patterns and representations.

3. **Pooling Layers**: Pooling layers (e.g., max pooling) reduce the spatial dimensions of the feature maps, which helps in reducing the computational load and controls overfitting. Pooling retains the most significant features while discarding less important information.

4. **Fully Connected Layers**: After several convolutional and pooling layers, the output is flattened and fed into fully connected layers. These layers perform the final classification or regression tasks by combining the extracted features into a single output.

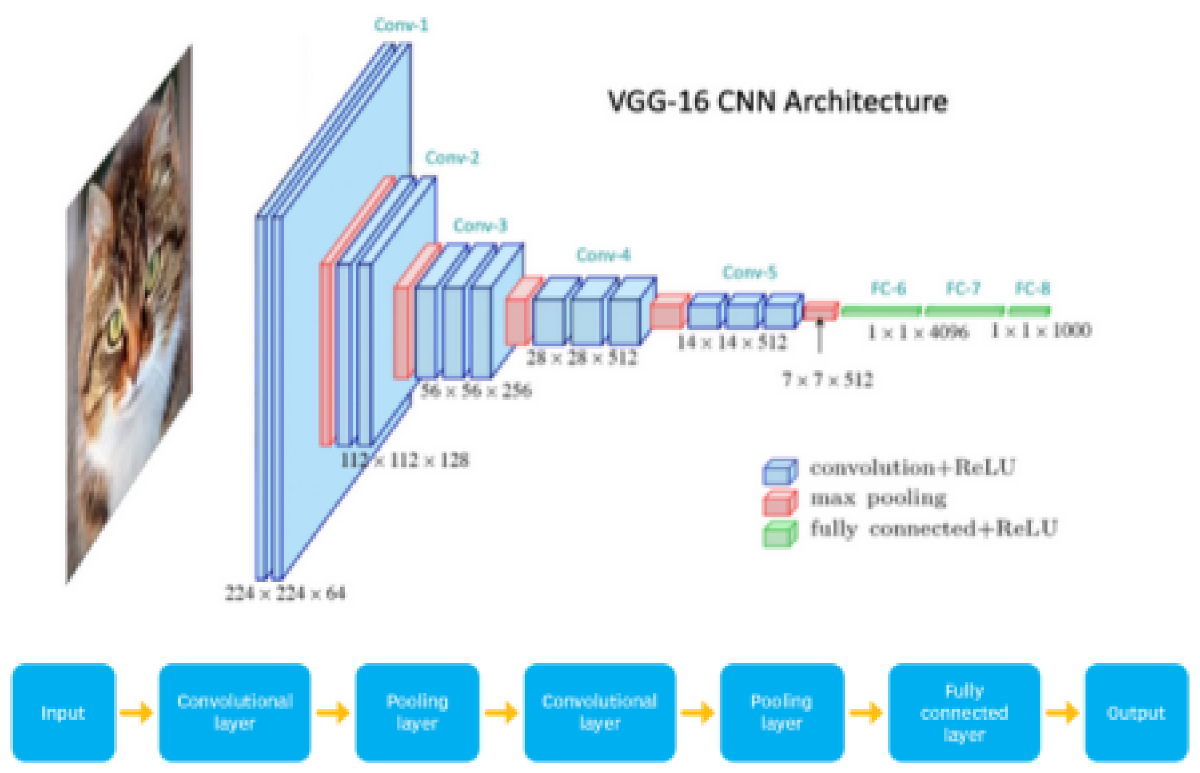
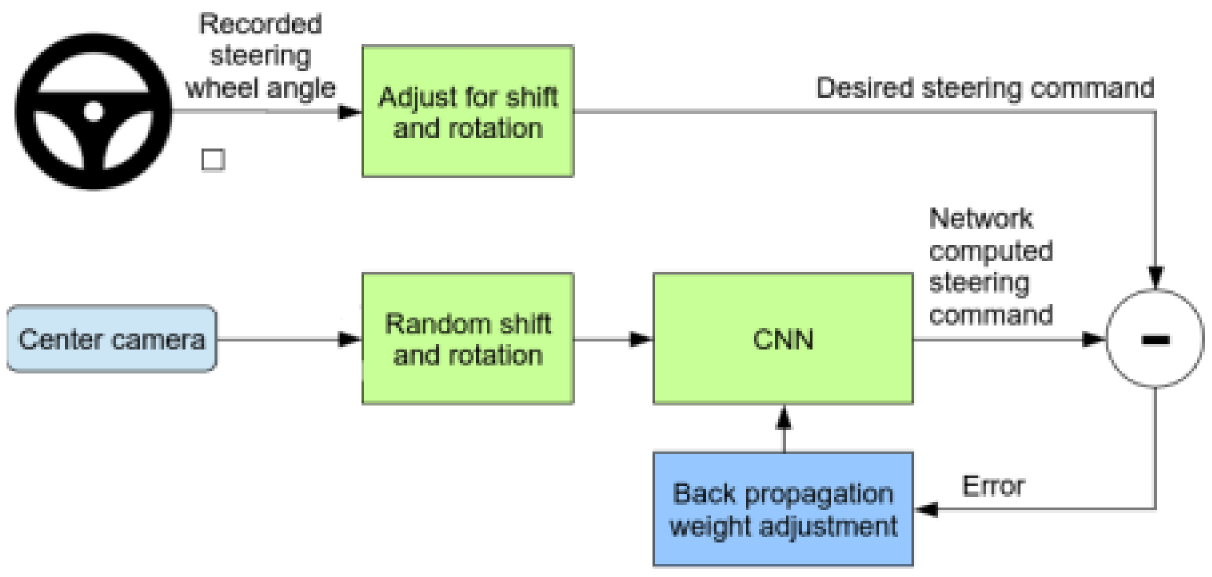


Fig. An example of CNN and the basic flow of a CNN (Convolutional Neural Network)

Flow of the Project

The project utilizes a convolutional neural network (CNN) to interpret front-facing camera footage and steering angle data, making predictions to control the vehicle's steering. Here's a detailed flow of the project:


Recorded steering wheel angle
Adjust for shift and rotation
Desired steering command
Network computed steering command
Center camera
Random shift and rotation
CNN
Back propagation weight adjustment
Error

Data Collection

Data collection for this project was done using the open-source car driving simulator called Udacity in training mode. This simulator offers a variety of roads, locations, and lighting scenarios, providing a diverse dataset for training and testing the autonomous driving model. Here's how the data collection process was carried out:

1. **Simulator Setup**:
   - **Udacity Simulator**: The Udacity simulator is set up in training mode, which allows for controlled collection of driving data. The simulator replicates various real-world driving conditions, offering a rich environment for data generation.
2. **Diverse Driving Scenarios**:
   - **Variety of Roads**: The simulator features multiple road types, including highways, city streets, and rural roads. This variety ensures the model is exposed to different driving scenarios.
   - **Locations**: Different geographical locations are simulated, providing a range of terrains and road layouts.
   - **Lighting Conditions**: The simulator includes various lighting scenarios, such as daytime, nighttime, and varying weather conditions. This diversity helps the model learn to navigate in different visibility conditions.

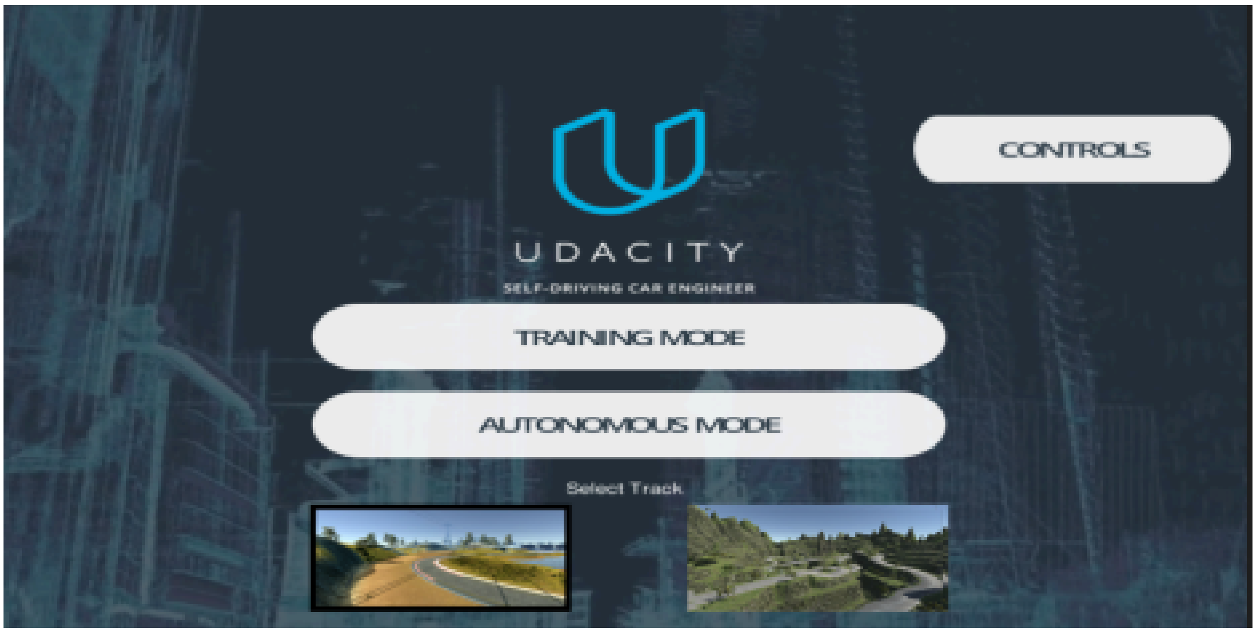
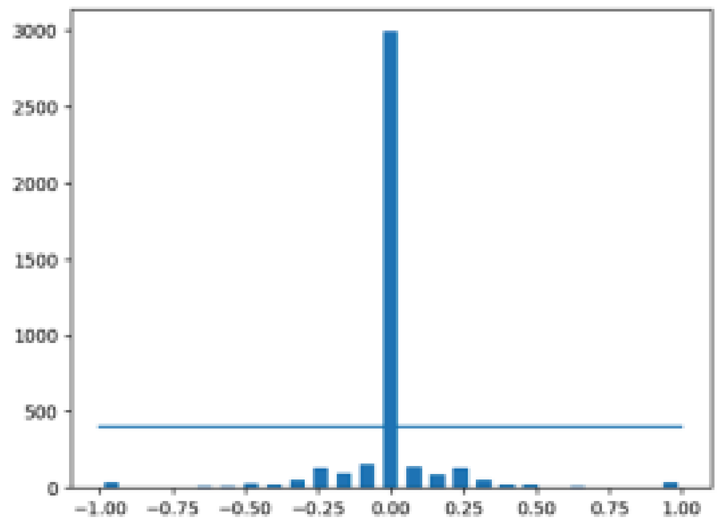


Fig. Picture of the simulator used in the project

Data Balancing

Data balancing involves adjusting the distribution of classes in a dataset to prevent any biases in the machine learning models. In the context of this project, data balancing ensures that the training data accurately represents a variety of driving scenarios and steering commands. This is crucial for developing a robust autonomous driving model that performs well in diverse conditions



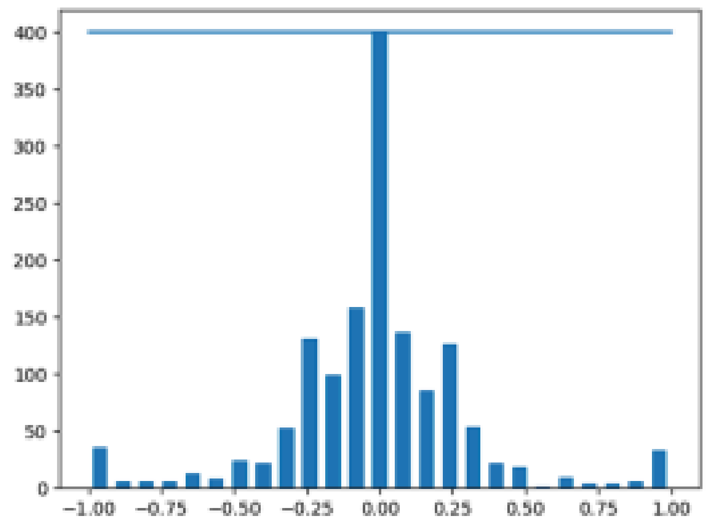


Fig. Data Before Balancing Fig. Data After Balancing

Network Architecture

The neural network architecture designed for this self-driving car project comprises 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. This architecture is tailored to effectively process and interpret visual data from a front-facing camera, mapping it directly to steering commands.

Input Processing

1. **Color Space Conversion**:
    - **YUV Color Planes**: The input image is split into YUV color planes. YUV is chosen because it separates the luminance (brightness) from the chrominance (color), which can improve the model's ability to process and understand visual information in various lighting conditions.
2. **Normalization Layer**:
    - **Image Normalization**: The initial layer normalizes the image. This normalization is predefined and remains unchanged during training. In-network normalization allows for flexibility in the normalization scheme based on the network's architecture and boosts processing speed via GPU acceleration.
3. **Convolutional Layers**:
    - **Layer Configuration**: The architecture includes 5 convolutional layers, focusing on feature extraction. These layers are configured through empirical experimentation to optimize the network's performance.
        - **First Three Layers**: These layers use strided convolutions with a 2×2 stride and a 5×5 kernel. Strided convolutions help reduce the spatial dimensions of the feature maps, allowing the network to learn larger and more complex features efficiently.
        - **Last Two Layers**: These layers employ non-strided convolutions with a 3×3 kernel. Non-strided convolutions help in preserving the spatial dimensions, allowing the network to learn finer details and intricate patterns.
4. **Activation Functions**: Each convolutional layer is followed by a non-linear activation function, typically ReLU (Rectified Linear Unit), which introduces non-linearity into the model, enabling it to learn more complex patterns.
5. **Pooling Layers**: Although not explicitly mentioned, pooling layers (e.g., max pooling) are often used in conjunction with convolutional layers to reduce the spatial dimensions and computational load while retaining the most significant features.
6. **Fully Connected Layers**:
    - **Three Fully Connected Layers**: Following the convolutional layers, there are three fully connected layers. These layers take the high-level features extracted by the convolutional layers and combine them to make final predictions.
    - **Output Layer**: The final layer outputs a control value representing the inverse turning radius, indicating how sharply the vehicle can turn. This control value is crucial for the autonomous vehicle to make precise steering adjustments.

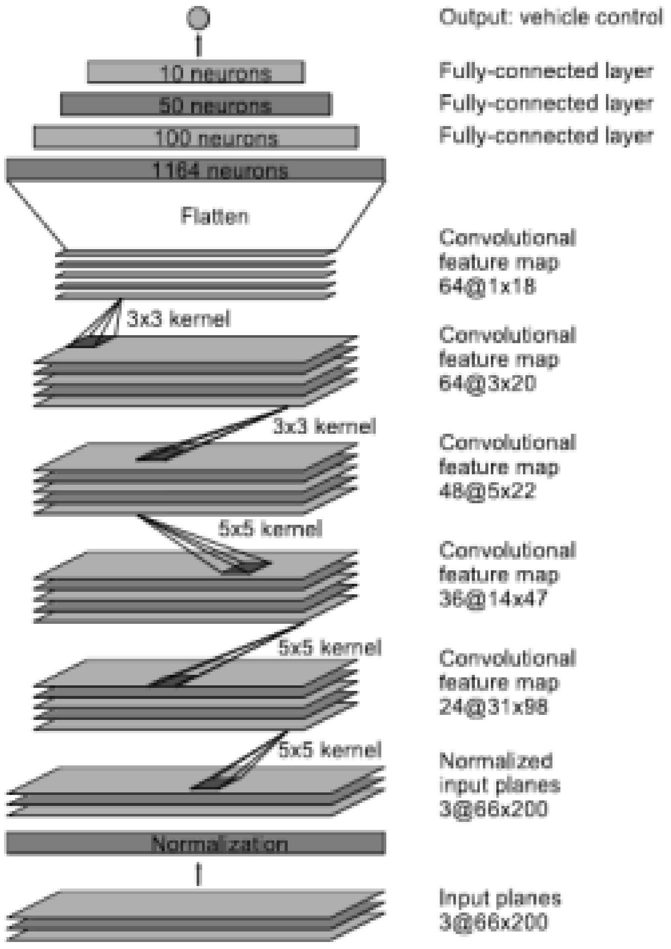


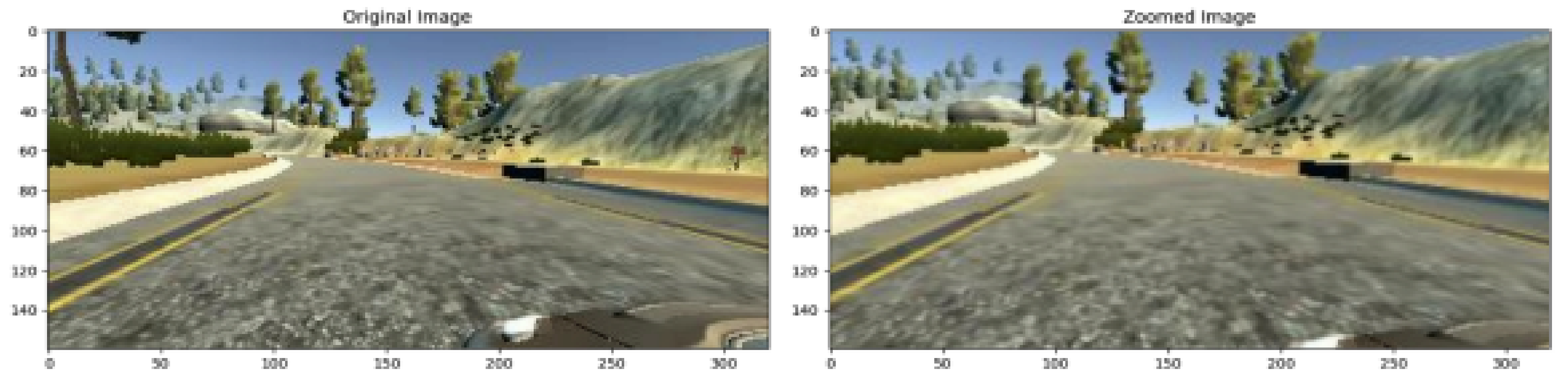
Fig. Diagram representing the network architecture used to train the model

Image Augmentation

Augmentation is a crucial technique in image processing and computer vision, especially in scenarios where the available dataset is limited or lacks diversity. By applying various random transformations to images, augmentation helps create a more robust and generalized model by exposing it to a wider range of variations that may
occur in real-world scenarios. Here's a bit more detail on common augmentation techniques:

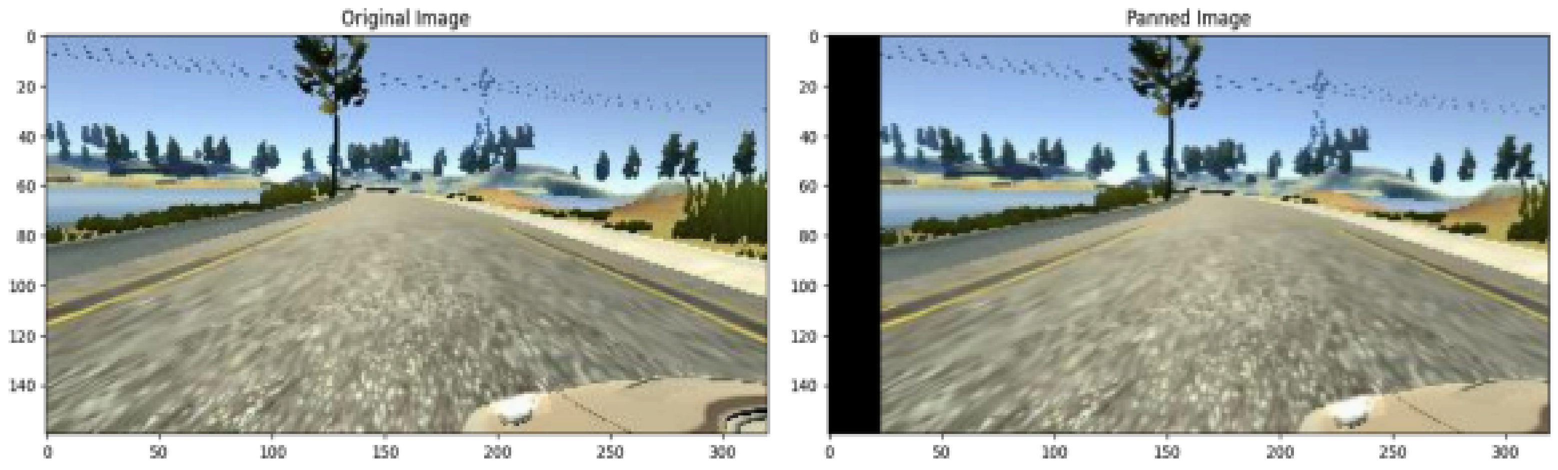
1. Zooming :

It involves adjusting the scale of an image, either by zooming in (enlarging a portion of the image) or zooming out (shrinking the entire image)



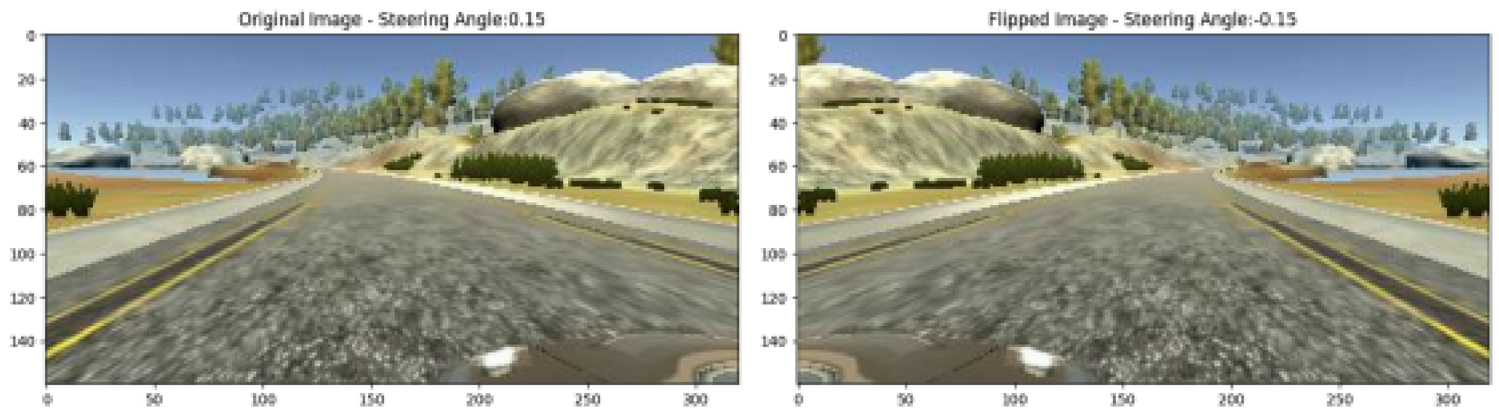
2. Panning:

Shifting an image horizontally or vertically can simulate changes in camera perspective or object position, enhancing the model's ability to generalize.



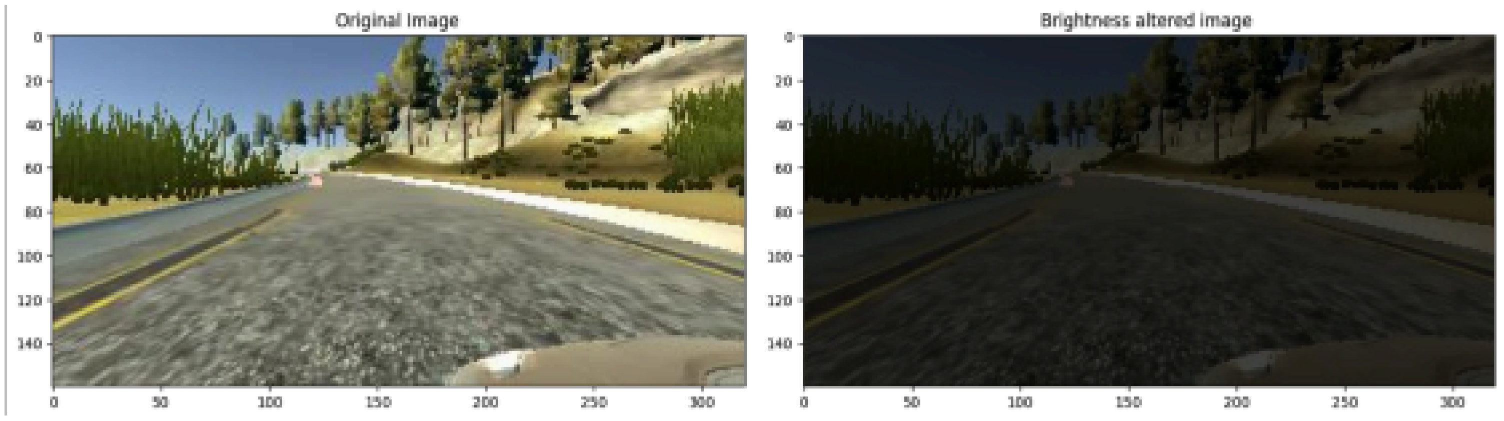
3. Image Flipping:

Horizontal and vertical flips can be applied to images, which aids in teaching the model that objects may appear in different orientations.


Original Image - Steering Angle:0.15
Flipped Image - Steering Angle:-0.15

4. Color Adjustments:

Altering brightness, contrast, saturation, and hue can simulate changes in lighting conditions, which are common in real-world scenarios.



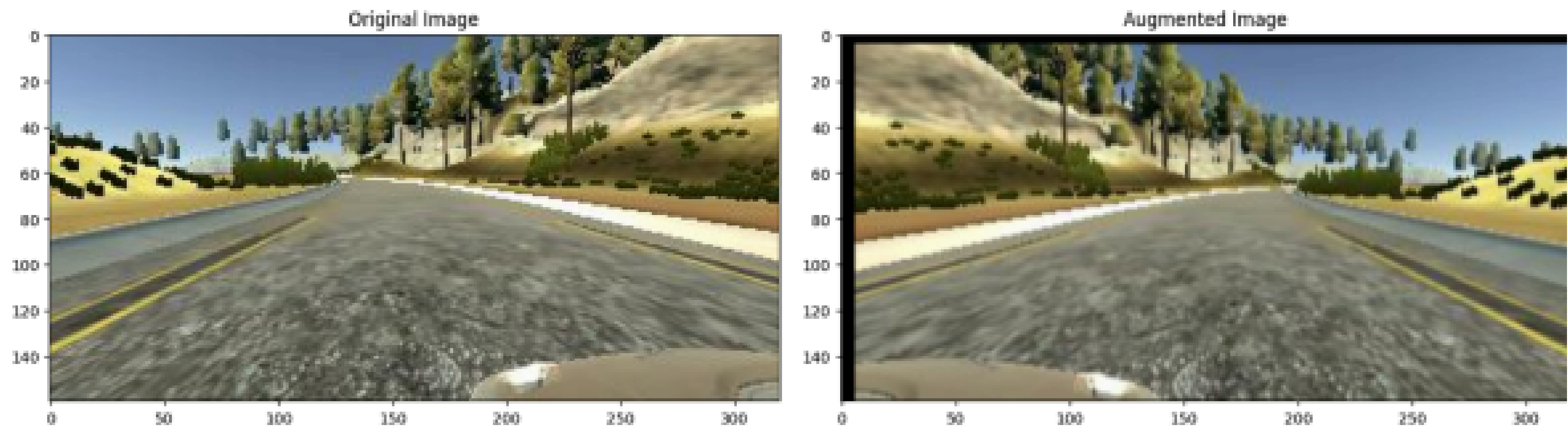


Fig. Image with all augmentation randomly applied over a loop

Image Pre-processing

Image preprocessing involves a series of techniques applied to raw image data to enhance its quality and make it suitable for further analysis or use in machine learning models. In this case, to meet the conditions of the CNN model provided by NVIDIA,

1. The image was cropped to a size (200 x 66).
2. Color format was changed from RGB to YUV format.
3. A Gaussian blur was applied to the image.

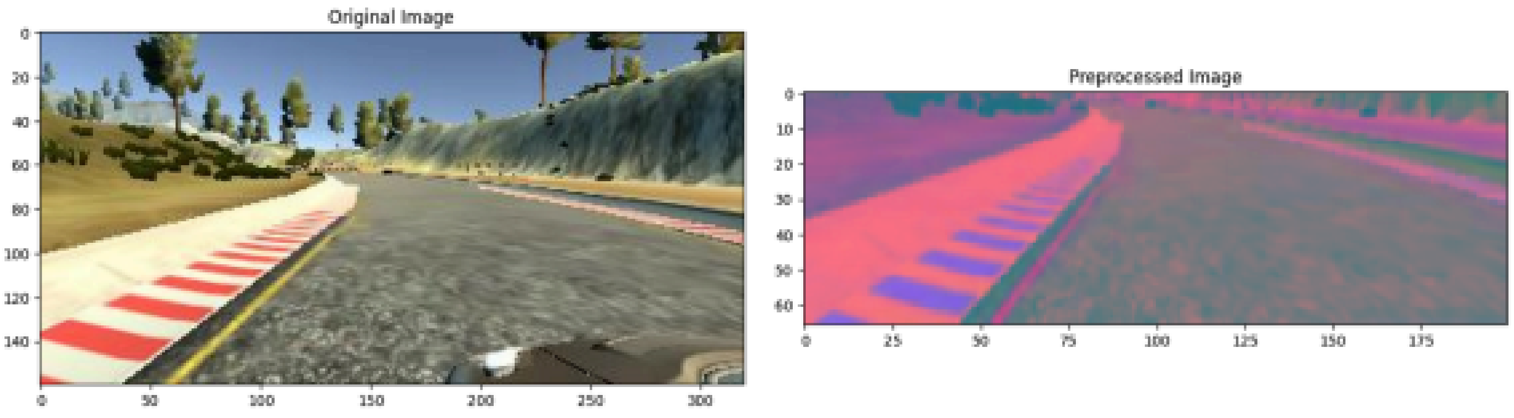


Fig. Representation of an image before and after image pre-processing

Model Training

Nvidia Model Architecture

The Nvidia model architecture for end-to-end learning of self-driving cars consists of several convolutional layers followed by fully connected layers. This architecture is designed to process images from a front-facing camera and output steering commands. This model includes 4 Convolutional layers and 5 Fully connected layers. This architecture is particularly effective for autonomous driving because it directly learns the mapping from raw pixels to control commands.

Training Data vs. Validation Data

Training Data:

This subset of data is used to train the model, allowing it to learn patterns and make predictions. The training data consists of images captured from the vehicle’s front camera and the corresponding steering angles, throttle, brake, and speed data.

It is essential to have a diverse and comprehensive training dataset that includes various driving conditions, such as different weather conditions, road types, and lighting conditions. This helps the model generalize better to unseen scenarios.

Validation Data:

This separate subset of data is used to evaluate the model's performance and generalization ability during the training process. It provides an unbiased evaluation of the model fit during the training phase.

The validation data is not used for training but is critical for hyperparameter tuning and preventing overfitting. By monitoring the model's performance on the validation data, we can make decisions such as adjusting learning rates, modifying model architecture, and applying regularization techniques.

Loss Function

The loss function quantifies the difference between the model's predictions and the actual target values. It is a crucial component of the training process as it guides the model by providing feedback on its performance. In the context of autonomous driving, the loss function helps the model learn to predict steering angles accurately based on input images.

For this project, the Mean Squared Error (MSE) loss function is used. MSE is defined as the average of the squares of the errors, where the error is the difference between the predicted and actual values.

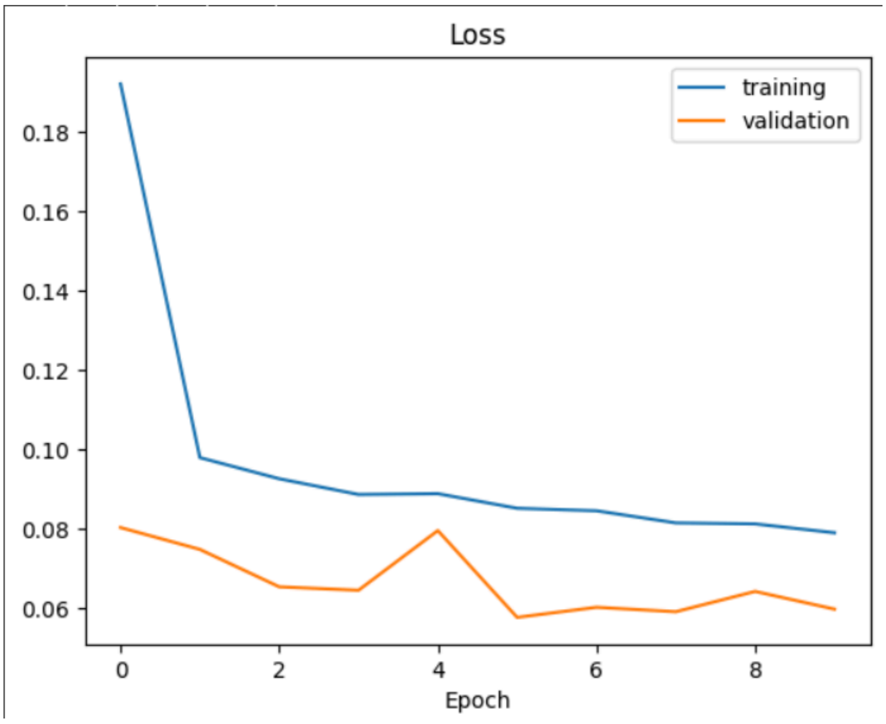
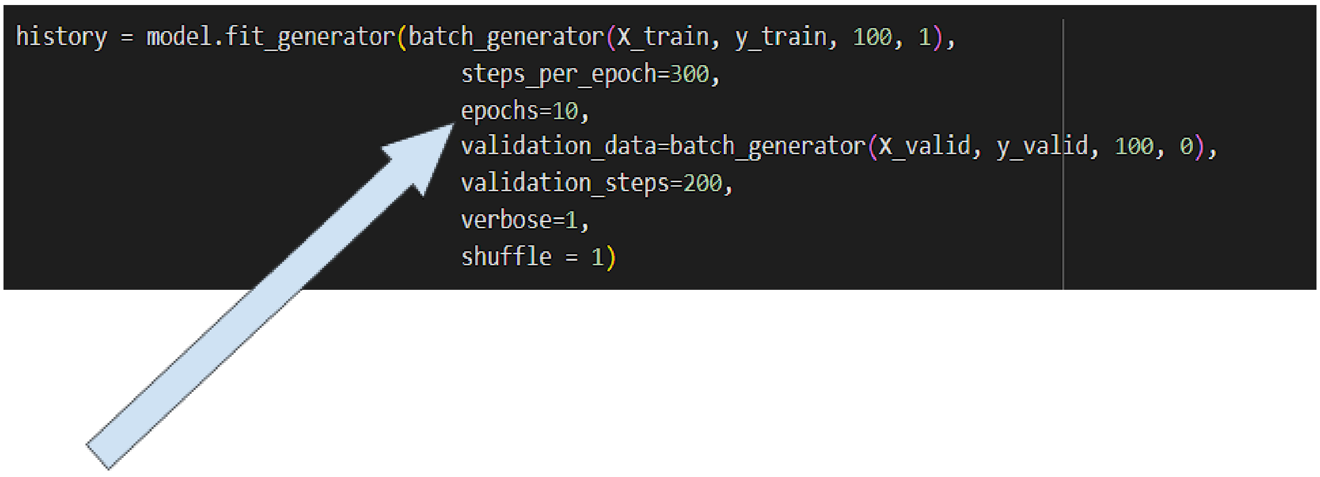


Fig: Blue line represents the training data loss and the yellow line represents the validation data loss and both are plotted against the epoch number.

Epoch Number

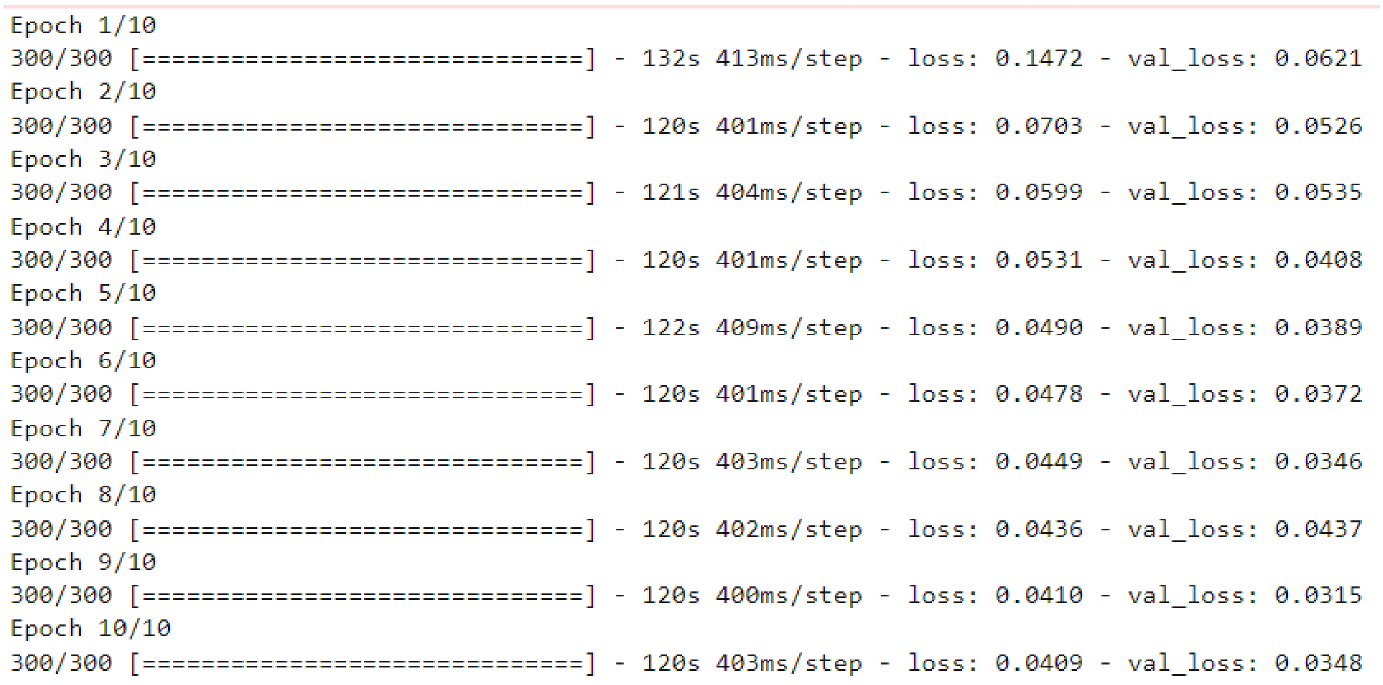
At this point, you might be curious as to what epoch number actually means. Epoch number refers to one complete pass through the entire training dataset by the learning algorithm. During training, an epoch is the unit of measurement that indicates how many times the model has seen the entire dataset .

- Multiple epochs allow the model to iterate over the training data multiple times, improving learning and performance.
- Increasing the number of epochs can help the model converge to a lower error rate, but too many epochs can lead to overfitting, where the model performs well on training data but poorly on new data.



```
history = model.fit_generator(batch_generator(X_train, y_train, 100, 1),
                              steps_per_epoch=300,
                              epochs=10,
                              validation_data=batch_generator(X_valid, y_valid, 100, 0),
                              validation_steps=200,
                              verbose=1,
                              shuffle = 1)
```

(Here we have defined the number of epoch's to be 10)



```
Epoch 1/10
300/300 [==============================] - 132s 413ms/step - loss: 0.1472 - val_loss: 0.0621
Epoch 2/10
300/300 [==============================] - 120s 401ms/step - loss: 0.0703 - val_loss: 0.0526
Epoch 3/10
300/300 [==============================] - 121s 404ms/step - loss: 0.0599 - val_loss: 0.0535
Epoch 4/10
300/300 [==============================] - 120s 401ms/step - loss: 0.0531 - val_loss: 0.0408
Epoch 5/10
300/300 [==============================] - 122s 409ms/step - loss: 0.0490 - val_loss: 0.0389
Epoch 6/10
300/300 [==============================] - 120s 401ms/step - loss: 0.0478 - val_loss: 0.0372
Epoch 7/10
300/300 [==============================] - 120s 403ms/step - loss: 0.0449 - val_loss: 0.0346
Epoch 8/10
300/300 [==============================] - 120s 402ms/step - loss: 0.0436 - val_loss: 0.0437
Epoch 9/10
300/300 [==============================] - 120s 400ms/step - loss: 0.0410 - val_loss: 0.0315
Epoch 10/10
300/300 [==============================] - 120s 403ms/step - loss: 0.0409 - val_loss: 0.0348
```

The above figure represents the training data loss and the validation loss and the time taken to train the model at the specific epoch number.

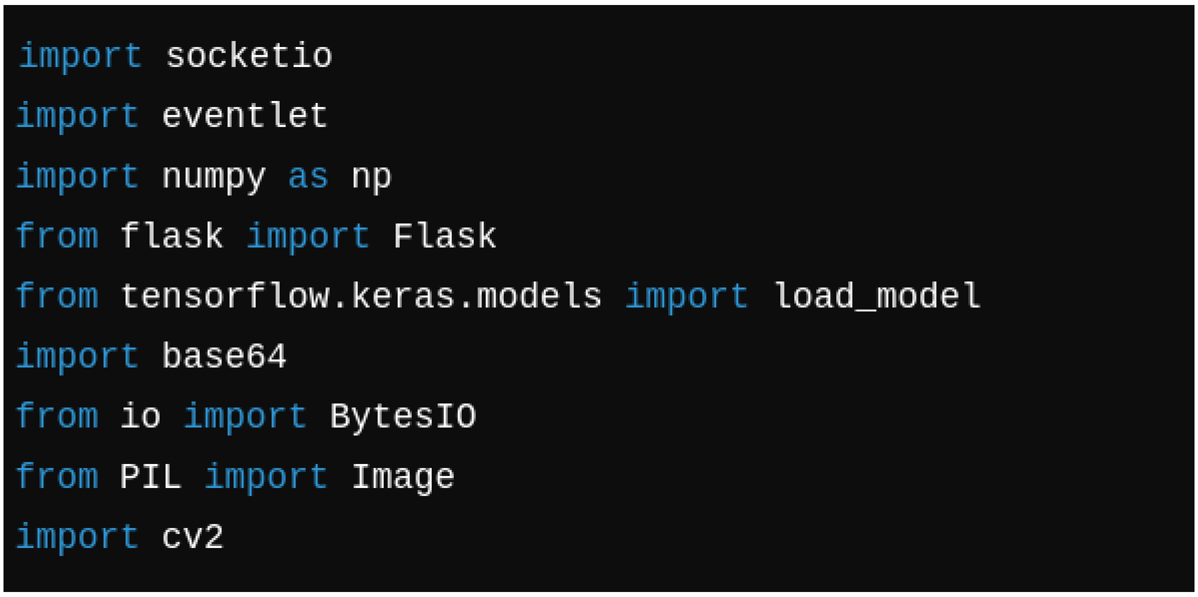
Model Deployment

Now that we have a trained and working model, we need to connect it to our simulator so that they can send and receive data between each other. To achieve this we host a web-server and deploy the model on it, which then connects to the simulator via the server.

Setting Up The Server

For this task multiple libraries were used such as :

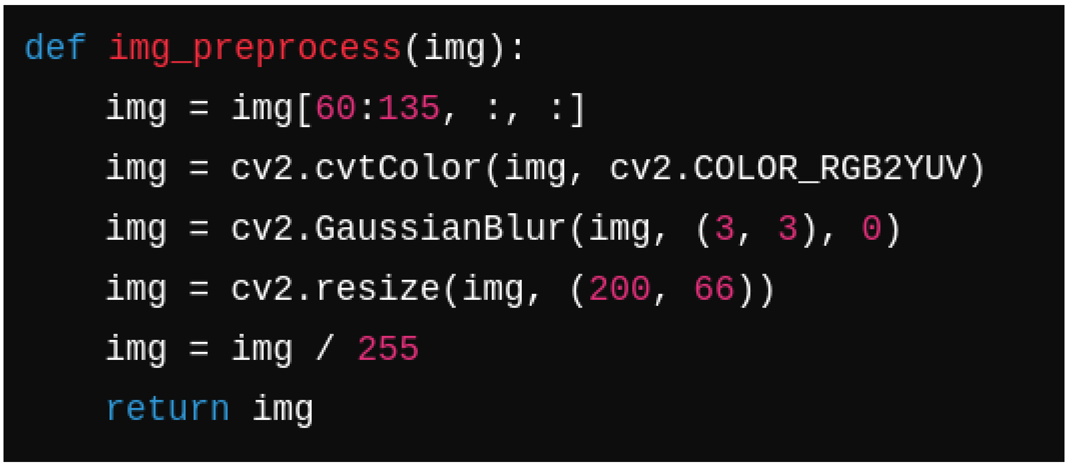
- **Socketio** and **Eventlet** for real-time communication.
- **Flask** for hosting the web server
- **TensorFlow / Keras** for loading the trained model
- Additional libraries include **base64**, **BytesIO**, **PIL** (Python Imaging Library), and **cv2** (OpenCV) for image processing.



```
import socketio
import eventlet
import numpy as np
from flask import Flask
from tensorflow.keras.models import load_model
import base64
from io import BytesIO
from PIL import Image
import cv2
```

Image Preprocessing

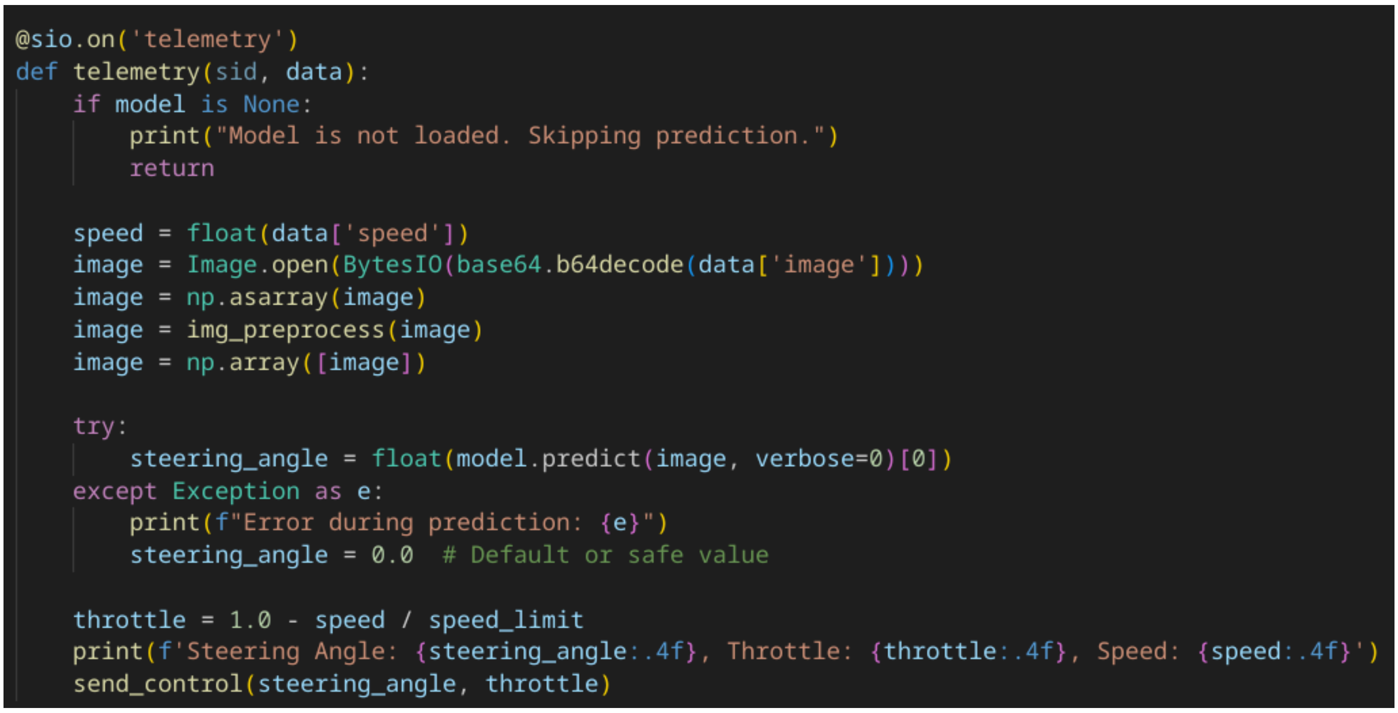
We preprocess incoming images to make them suitable for model prediction. This involves cropping the image, converting it to YUV color space, applying Gaussian blur, resizing, and normalizing the pixel values.



```python
def img_preprocess(img):
    img = img[60:135, :, :]
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    img = cv2.GaussianBlur(img, (3, 3), 0)
    img = cv2.resize(img, (200, 66))
    img = img / 255
    return img
```

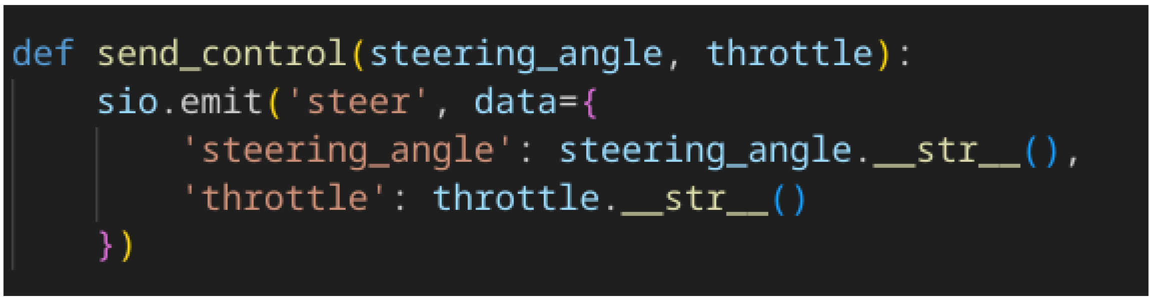
Telemetry Function

This function is the core of our server. It receives data (throttle speed, steering angle, image etc) from the simulator and calls the preprocessing function and then makes a steering angle prediction using the model, and calculates the throttle based on the speed provided.



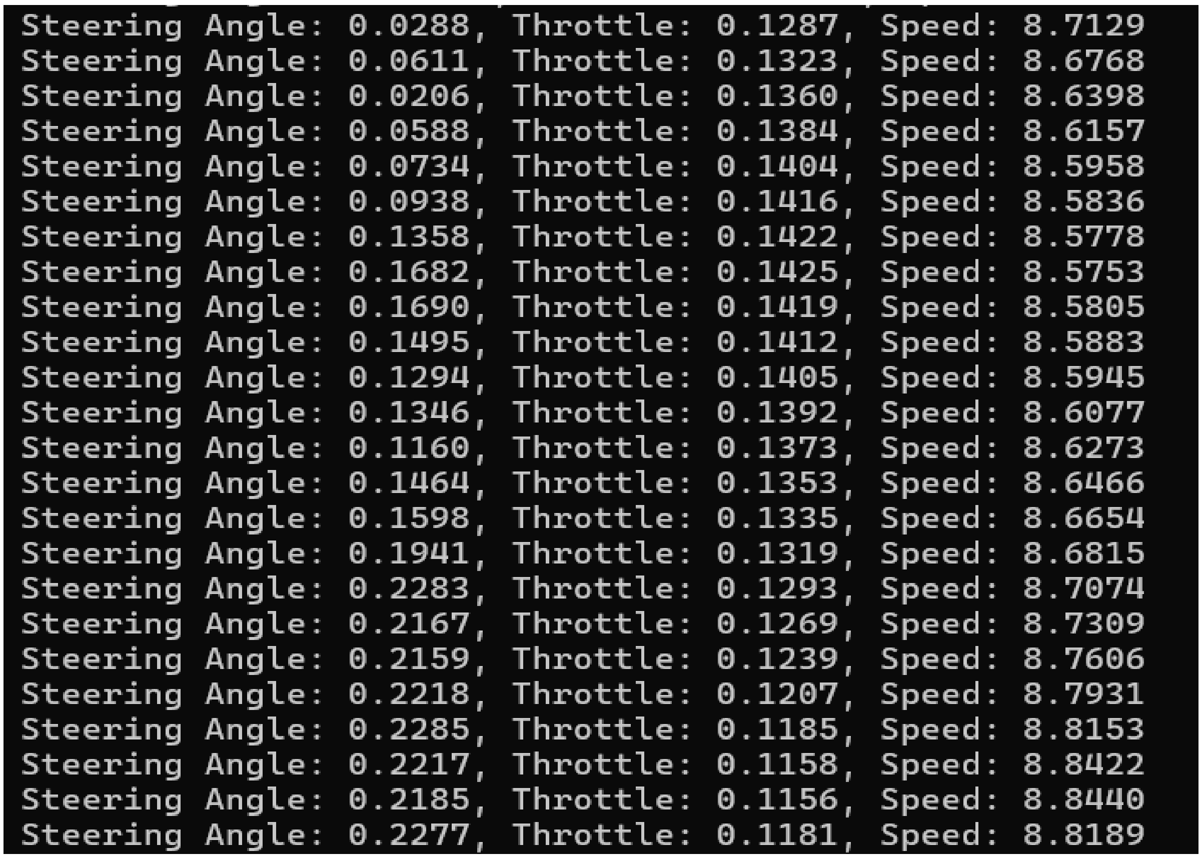
```python
@sio.on('telemetry')
def telemetry(sid, data):
    if model is None:
        print("Model is not loaded. Skipping prediction.")
        return

    speed = float(data['speed'])
    image = Image.open(BytesIO(base64.b64decode(data['image'])))
    image = np.asarray(image)
    image = img_preprocess(image)
    image = np.array([image])

    try:
        steering_angle = float(model.predict(image, verbose=0)[0])
    except Exception as e:
        print(f"Error during prediction: {e}")
        steering_angle = 0.0  # Default or safe value

    throttle = 1.0 - speed / speed_limit
    print(f'Steering Angle: {steering_angle:.4f}, Throttle: {throttle:.4f}, Speed: {speed:.4f}')
    send_control(steering_angle, throttle)
```

Sending Control Commands



```python
def send_control(steering_angle, throttle):
    sio.emit('steer', data={
        'steering_angle': steering_angle.__str__(),
        'throttle': throttle.__str__()
    })
```

Results



```
Steering Angle: 0.0288, Throttle: 0.1287, Speed: 8.7129
Steering Angle: 0.0611, Throttle: 0.1323, Speed: 8.6768
Steering Angle: 0.0206, Throttle: 0.1360, Speed: 8.6398
Steering Angle: 0.0588, Throttle: 0.1384, Speed: 8.6157
Steering Angle: 0.0734, Throttle: 0.1404, Speed: 8.5958
Steering Angle: 0.0938, Throttle: 0.1416, Speed: 8.5836
Steering Angle: 0.1358, Throttle: 0.1422, Speed: 8.5778
Steering Angle: 0.1682, Throttle: 0.1425, Speed: 8.5753
Steering Angle: 0.1690, Throttle: 0.1419, Speed: 8.5805
Steering Angle: 0.1495, Throttle: 0.1412, Speed: 8.5883
Steering Angle: 0.1294, Throttle: 0.1405, Speed: 8.5945
Steering Angle: 0.1346, Throttle: 0.1392, Speed: 8.6077
Steering Angle: 0.1160, Throttle: 0.1373, Speed: 8.6273
Steering Angle: 0.1464, Throttle: 0.1353, Speed: 8.6466
Steering Angle: 0.1598, Throttle: 0.1335, Speed: 8.6654
Steering Angle: 0.1941, Throttle: 0.1319, Speed: 8.6815
Steering Angle: 0.2283, Throttle: 0.1293, Speed: 8.7074
Steering Angle: 0.2167, Throttle: 0.1269, Speed: 8.7309
Steering Angle: 0.2159, Throttle: 0.1239, Speed: 8.7606
Steering Angle: 0.2218, Throttle: 0.1207, Speed: 8.7931
Steering Angle: 0.2285, Throttle: 0.1185, Speed: 8.8153
Steering Angle: 0.2217, Throttle: 0.1158, Speed: 8.8422
Steering Angle: 0.2185, Throttle: 0.1156, Speed: 8.8440
Steering Angle: 0.2277, Throttle: 0.1181, Speed: 8.8189
```

Fig: The receiving, processing and output of data by the model

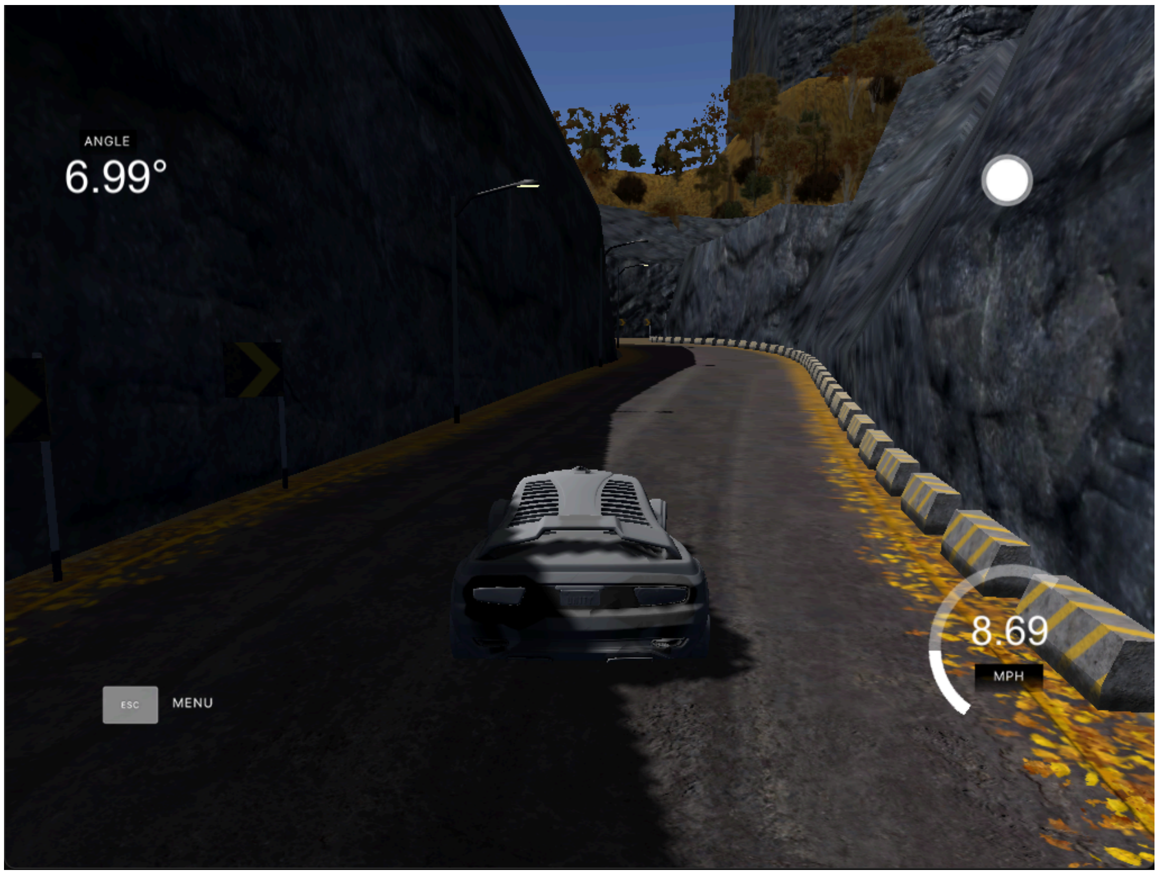


Fig: Autonomous Vehicle

Conclusion

In conclusion, our project successfully integrated data collection, model training, and deployment to create an effective autonomous driving system. We used computer vision and image processing techniques to preprocess images and train a robust model. The deployment utilized a Flask server with Socket.IO for real-time communication with the driving simulator, allowing for accurate control of the vehicle. The model performed well in various scenarios, demonstrating reliable predictions and control commands. Future improvements will focus on enhancing model performance in complex environments and transitioning to real-world testing

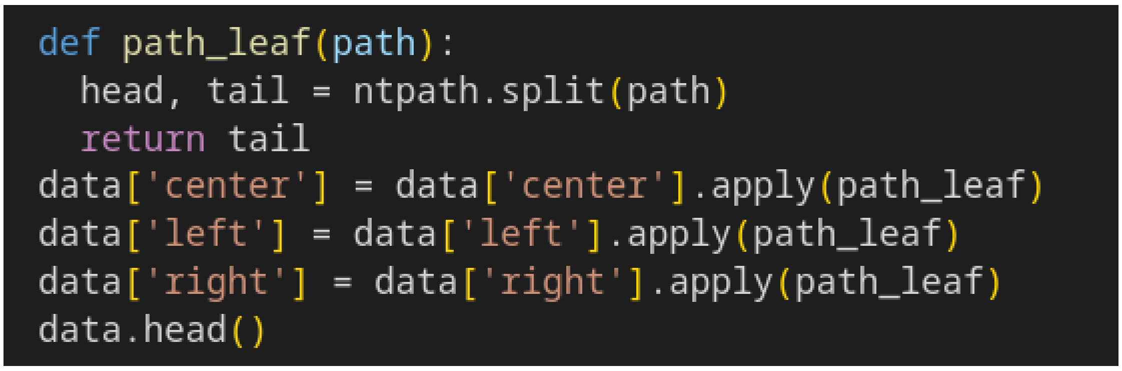
Conclusion

In conclusion, our project successfully integrated data collection, model training, and deployment to create an effective autonomous driving system. We used computer vision and image processing techniques to preprocess images and train a robust model. The deployment utilized a Flask server with Socket.IO for real-time communication with the driving simulator, allowing for accurate control of the vehicle. The model performed well in various scenarios, demonstrating reliable predictions and control commands. Future improvements will focus on enhancing model performance in complex environments and transitioning to real-world testing.

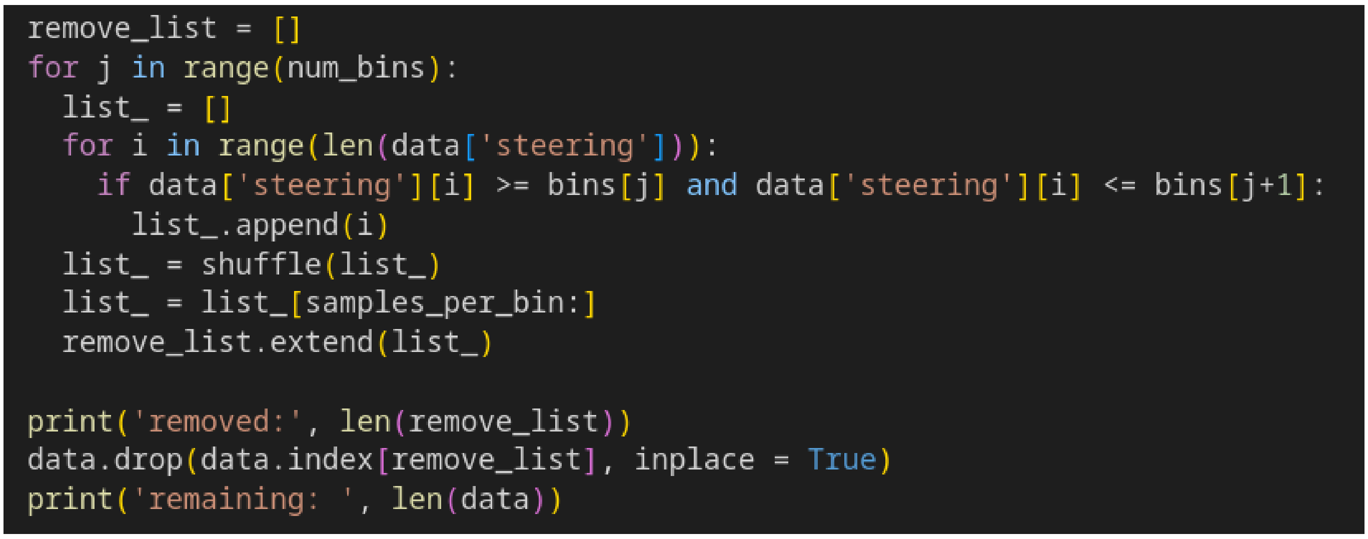
Project Github Repository Link -
https://github.com/radnus321/Autonomous-Vehicle-Driving-Program

Appendix :

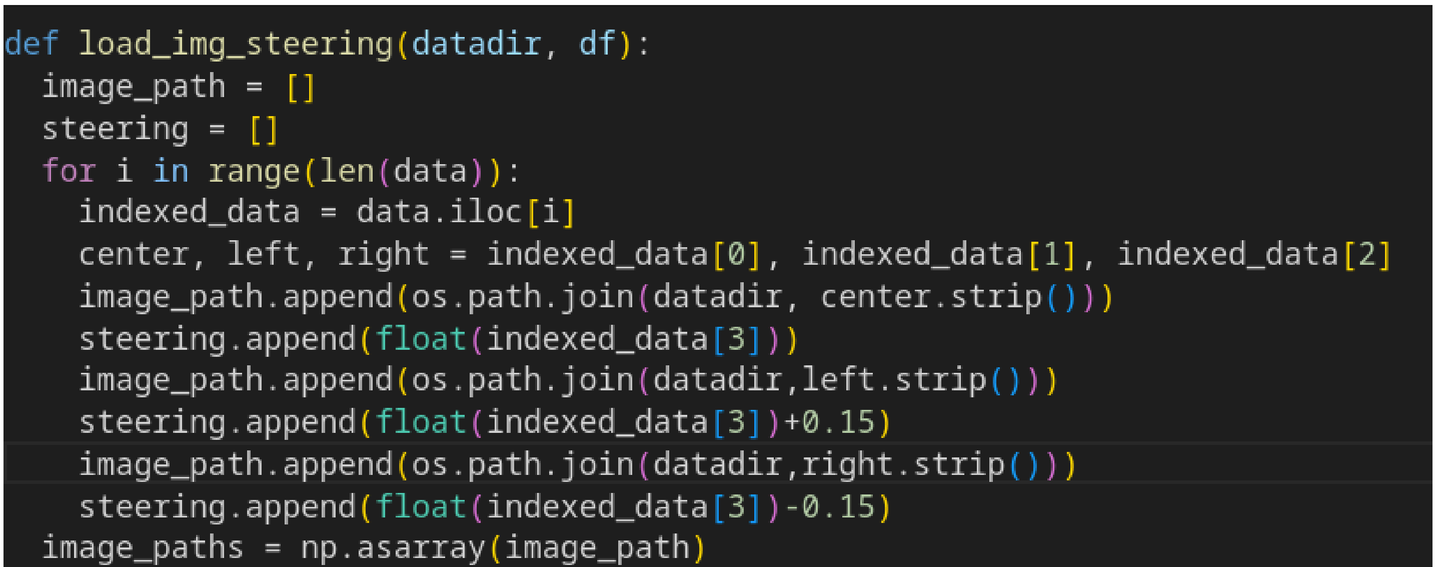
Important code blocks used in the project



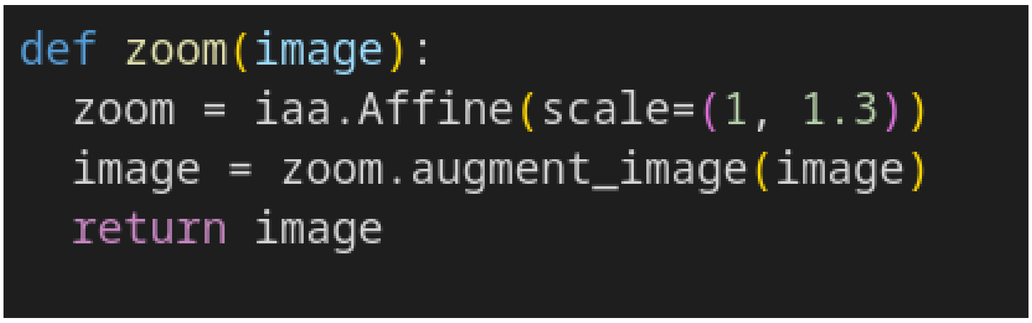
```
def path_leaf(path):
  head, tail = ntpath.split(path)
  return tail
data['center'] = data['center'].apply(path_leaf)
data['left'] = data['left'].apply(path_leaf)
data['right'] = data['right'].apply(path_leaf)
data.head()
```



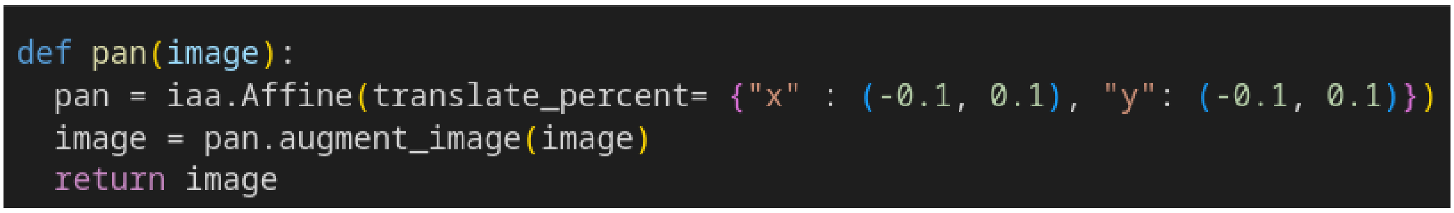
```
remove_list = []
for j in range(num_bins):
  list_ = []
  for i in range(len(data['steering'])):
    if data['steering'][i] >= bins[j] and data['steering'][i] <= bins[j+1]:
      list_.append(i)
  list_ = shuffle(list_)
  list_ = list_[samples_per_bin:]
  remove_list.extend(list_)

print('removed:', len(remove_list))
data.drop(data.index[remove_list], inplace = True)
print('remaining: ', len(data))
```



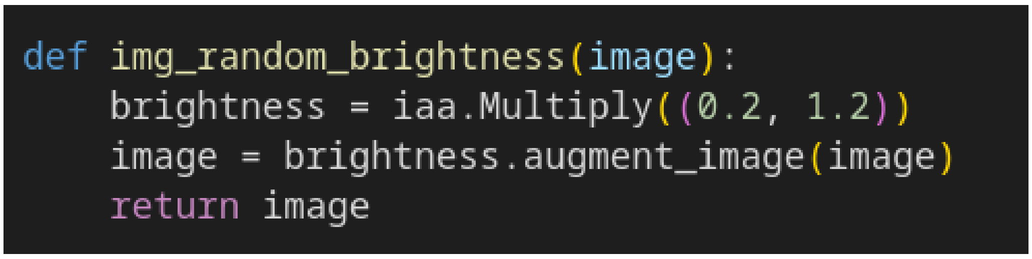
```
def load_img_steering(datadir, df):
  image_path = []
  steering = []
  for i in range(len(data)):
    indexed_data = data.iloc[i]
    center, left, right = indexed_data[0], indexed_data[1], indexed_data[2]
    image_path.append(os.path.join(datadir, center.strip()))
    steering.append(float(indexed_data[3]))
    image_path.append(os.path.join(datadir,left.strip()))
    steering.append(float(indexed_data[3])+0.15)
    image_path.append(os.path.join(datadir,right.strip()))
    steering.append(float(indexed_data[3])-0.15)
  image_paths = np.asarray(image_path)
```



```python
def zoom(image):
  zoom = iaa.Affine(scale=(1, 1.3))
  image = zoom.augment_image(image)
  return image
```



```python
def pan(image):
  pan = iaa.Affine(translate_percent= {"x" : (-0.1, 0.1), "y": (-0.1, 0.1)})
  image = pan.augment_image(image)
  return image
```



```python
def img_random_brightness(image):
    brightness = iaa.Multiply((0.2, 1.2))
    image = brightness.augment_image(image)
    return image
```

```
def batch_generator(image_paths, steering_ang, batch_size, istraining):

  while True:
    batch_img = []
    batch_steering = []

    for i in range(batch_size):
      random_index = random.randint(0, len(image_paths) - 1)

      if istraining:
        im, steering = random_augment(image_paths[random_index], steering_ang[random_index])

      else:
        im = mpimg.imread(image_paths[random_index])
        steering = steering_ang[random_index]

      im = img_preprocess(im)
      batch_img.append(im)
      batch_steering.append(steering)
    yield (np.asarray(batch_img), np.asarray(batch_steering))
x_train_gen, y_train_gen = next(batch_generator(X_train, y_train, 1, 1))
x_valid_gen, y_valid_gen = next(batch_generator(X_valid, y_valid, 1, 0))
```

References

1. https://arxiv.org/abs/1604.07316