***Defence Research and Development Organisation***



***Project Report :2024***

***Behavioral cloning based Self Driving Car***

***Mentor: Mr. Vivek Kumar Kataria***

***Archit Jain***

***GLA University, Mathura***

**INDEX**

1. INTRODUCTION
   * Overview of Behavioral Cloning
   * NVIDIA Architecture for End-to-End Self-Driving
   * Implementation Using Udacity Self-Driving Car Simulator
   * Techniques for Enhancing Model Performance
   * Project Outcomes and Impact
2. ACKNOWLEDGMENT
   * Appreciation and Contributions of Mr. Vivek Kumar Kataria
   * Impact on Project Development and Learning Experience
3. PROPOSED APPROACH
   * Data Collection
     + Downloading and Listing Dataset Files
   * Import Libraries
     + Libraries and Their Uses
   * Data Frame
     + Loading and Inspecting Driving Data
     + Simplifying File Paths
   * Data Visualization and Description
     + Descriptive Statistics
     + Throttle Distribution
     + Correlation Matrix
     + Throttle vs. Speed Scatter Plot
     + Steering Angle Histogram
     + Removing Data Bias
   * Function to Load Images and Steering Angles
     + Preparing Image Paths and Steering Angles
   * Split Data into Training and Validation Sets
     + Data Splitting and Visualization
   * Image Augmentations
     + Zoom Augmentation
     + Pan Augmentation
     + Random Brightness Adjustment
     + Random Flip Augmentation
   * Function to Apply Random Augmentations to Images
   * Images Preprocessing
     + Cropping, Color Space Conversion, Blurring, Resizing, Normalizing
   * Generator Function
     + Batch Generation for Training and Validation
   * Retrieving and Visualizing Sample Images
     + Visualization of Training and Validation Batches
4. NVIDIA BEHAVIORAL CLONING MODEL ARCHITECTURE
   * Model Architecture Overview
   * Input and Convolutional Layers
   * Flatten and Dense Layers
   * Activation Function - ELU
   * Compilation Details
     + Loss Function and Optimizer
   * Model Input and Output Specifications
5. TRAINING MODEL
   * Training Process
     + Model Training with model.fit
     + Batch Generator Parameters
     + Epochs and Validation Data
   * Training Process Details
     + Steps per Epoch
     + Validation Steps
     + Verbose and Shuffle Parameters
     + History Object
6. MODEL EVALUATION
   * Model Evaluation on Validation Data
     + Processing Validation Images
     + Evaluating Mean Squared Error (MSE)
   * Visualization of Predictions
     + Scatter Plot of Predicted vs Actual Values
     + Visualization Details
   * Training and Validation Loss Visualization
     + Plotting Training and Validation Loss
7. CONCLUSION

**INTRODUCTION**

In this project, we delve into the development of a self-driving car by leveraging behavioral cloning, a sophisticated machine learning technique that enables vehicles to navigate autonomously by imitating human driving behavior. Behavioral cloning involves the creation of a model that learns to drive by analyzing a dataset containing images and corresponding driving actions (such as steering angles, throttle, and braking) captured from a human driver. The objective is to train the model to predict the appropriate driving commands when presented with similar visual inputs, thus allowing the car to drive autonomously in a variety of scenarios.

The core of our approach is based on the NVIDIA architecture, a well-regarded convolutional neural network (CNN) model specifically designed for end-to-end self-driving applications. The NVIDIA architecture processes raw images from the car's front-facing camera and directly outputs steering commands, eliminating the need for hand-crafted features or complex pre-processing steps. This architecture consists of a series of convolutional layers that extract relevant features from the input images, followed by fully connected layers that translate these features into precise control signals.

The NVIDIA model's strength lies in its ability to generalize across diverse driving conditions, including different lighting, road types, and weather scenarios. By training on a comprehensive dataset that includes various driving environments, the model can effectively handle real-world driving challenges, such as sharp turns, lane changes, and obstacle avoidance. This capability is crucial for ensuring the reliability and safety of autonomous vehicles on public roads.

In the context of this project, behavioral cloning is implemented using the Udacity self-driving car simulator. The simulator provides a virtual environment where the car can be trained and tested without the risks associated with real-world testing. The driving data collected in the simulator includes video frames from the car's perspective and the corresponding control inputs, which are used to train the model.

Throughout the training process, we employ several techniques to enhance the model's performance and generalization ability. Data augmentation, such as adding random shifts, rotations, and brightness adjustments to the images, is applied to increase the diversity of the training data and prevent overfitting. Additionally, the model is fine-tuned using validation data to ensure that it performs well on unseen driving scenarios.

The result of this project is a self-driving car capable of navigating autonomously in a simulated environment, making decisions in real-time based on visual input. The successful implementation of behavioral cloning using the NVIDIA architecture demonstrates the potential of deep learning in revolutionizing the field of autonomous driving, paving the way for safer and more efficient transportation systems.

ACKNOWLEGMENT

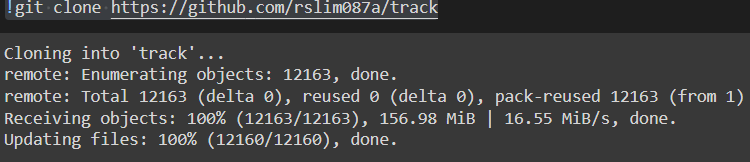
I would like to express my deepest gratitude to Mr. Vivek Kumar Kataria, a distinguished scientist at DRDO, for his invaluable mentorship throughout this project. His guidance and support have been instrumental in every phase of the project, from the initial planning to the final implementation.

Mr. Kataria's expertise in project management was particularly beneficial, as he provided me with the tools and strategies needed to effectively organize and execute the various stages of this project. His advice on analyzing and interpreting research papers was equally crucial, helping me to deepen my understanding of the underlying concepts and apply them successfully to the development of the self-driving car model.

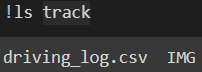
I am immensely grateful for his encouragement, insights, and unwavering support, which have not only contributed to the success of this project but have also enriched my learning experience in profound ways

PROPOSED APPROACH

# Data Collection:



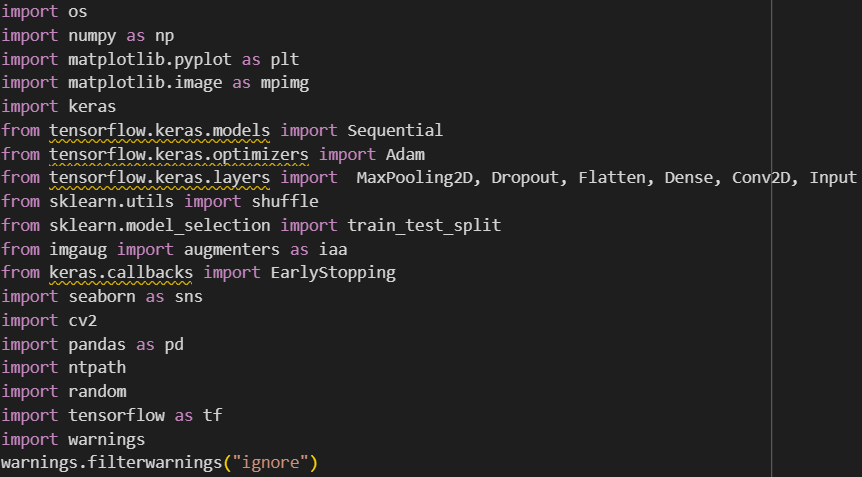
This command downloads the entire track repository from GitHub to your local environment. In this track we have dataset as well as images which are collected while using training mode of the udacity simulator.



This command lists all the files and folders inside the track directory. And gives as an output

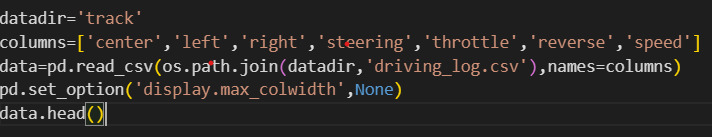
driving\_log.csv (dataset) and IMG folder (images).

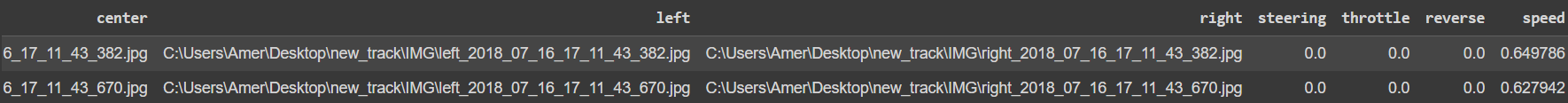
# Import library:



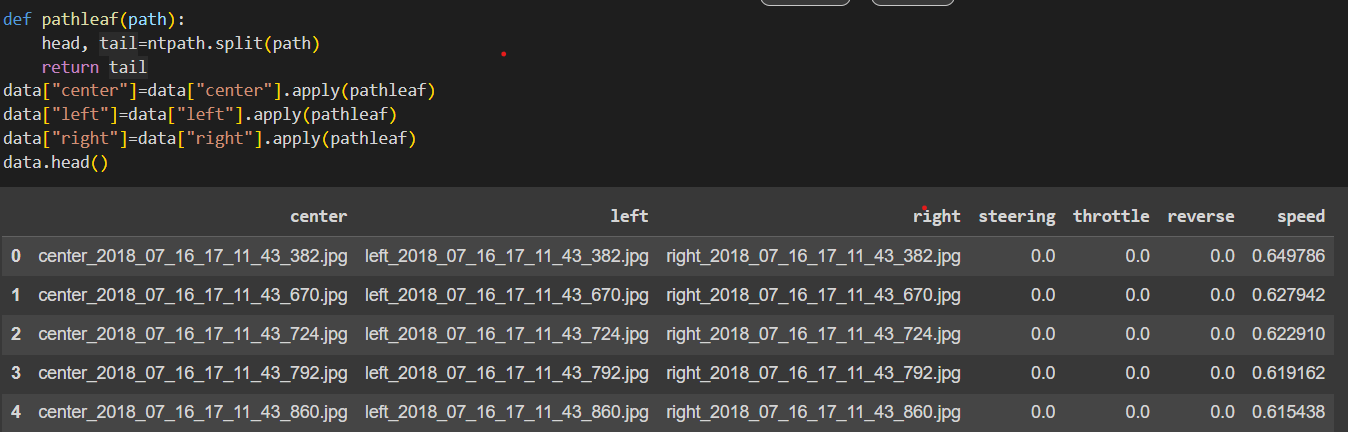
* **os**: It helps in tasks like file and directory management, such as loading images from a directory or saving results.
* **numpy**: It supports the efficient handling of large arrays and matrices of numerical data, which are essential when working with image data and model inputs.
* **matplotlib.pyplot**: Used to visualize images, plot training history, and analyze model performance through graphs like loss curves.
* **matplotlib.image as mpimg**: It loads and displays images, which is useful for inspecting the dataset and visualizing augmented images.
* **keras** and **tensorflow.keras**: Together, they are used to define, compile, and train deep learning models. The Sequential API from Keras is used to build our convolutional neural network (CNN) for predicting driving behavior.
* **Sequential**: Used to create our CNN model by stacking layers sequentially.
* **Adam**: Chosen for optimizing the model, as it efficiently adjusts the learning rate during training.
* **MaxPooling2D, Dropout, Flatten, Dense, Conv2D, Input**: MaxPooling2D: Reduces spatial dimensions, downsampling the input to highlight dominant features. Dropout: Prevents overfitting by randomly setting a fraction of input units to zero during training. Flatten: Converts the matrix output from convolutional layers into a vector for fully connected layers. Dense: Fully connected layer, used for the final prediction. Conv2D: Applies convolutional filters to input images, extracting important features. Input: Specifies the shape of input data.
* **sklearn.utils.shuffle**: Helps in shuffling the dataset to ensure that the model does not learn any unintended patterns related to the order of the data.
* **sklearn.model\_selection.train\_test\_split**: Used to create training and validation sets, ensuring the model is tested on unseen data.
* **imgaug.augmenters as iaa**: Applied to artificially increase the diversity of the training data through techniques like rotation, flipping, and brightness adjustment, which improves model generalization.
* **seaborn**: Used for creating advanced visualizations like heatmaps and correlation matrices, which help in understanding data distributions and relationships.
* **cv2 (OpenCV)**: Used for various image processing tasks, such as resizing, cropping, and converting images, which are essential for preparing the data for the model.
* **pandas**: Handles and processes structured data, such as CSV files containing labels and metadata for images.
* **ntpath**: Used to extract file names and manipulate paths, which is helpful when loading and saving images.
* **random**: Used to randomize various aspects of the dataset, like selecting random images for visualization or augmentation.
* **tensorflow**: TensorFlow serves as the backend for Keras, facilitating the execution of deep learning models on various hardware configurations, including CPUs and GPUs.

# Data frame:



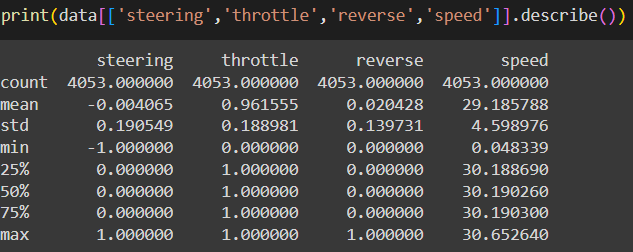


Is responsible for loading and inspecting the driving data collected during the simulation. The driving\_log.csv file contains key information, including image paths and driving commands (steering, throttle, etc.), which will be used to train the model. By setting the correct column names and ensuring that the data is loaded into a Pandas Data Frame, we can efficiently manage and manipulate the data for further processing and model training.

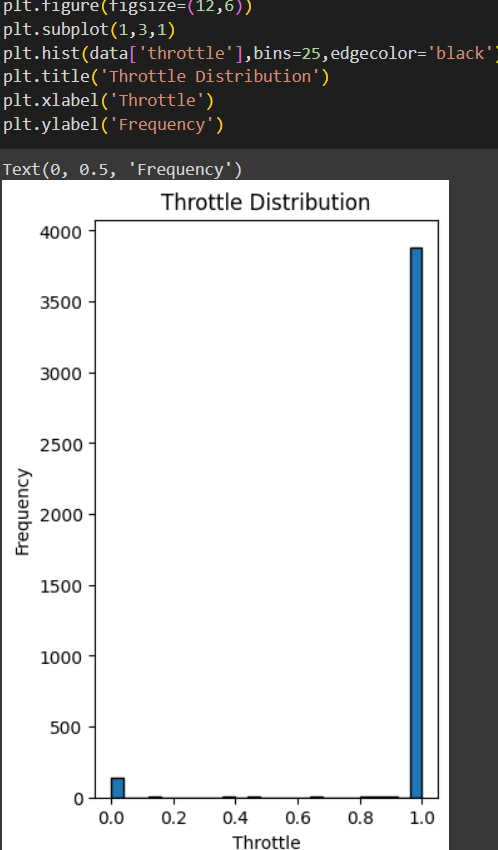


The provided code snippet is designed to simplify file paths in the Data Frame for better readability and management. The pathleaf function extracts and returns the filename from a given file path using ntpath.split(), which splits the path into a head (directory) and a tail (filename). This function is then applied to the center, left, and right columns of the DataFrame, replacing the full file paths with just the filenames. The final line displays the updated DataFrame, showing the filenames rather than the complete paths, which makes it easier to work with the dataset and ensures that only relevant information is presented.

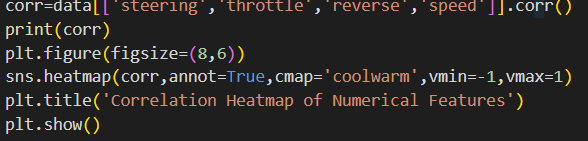
# Data Visualization and Description:

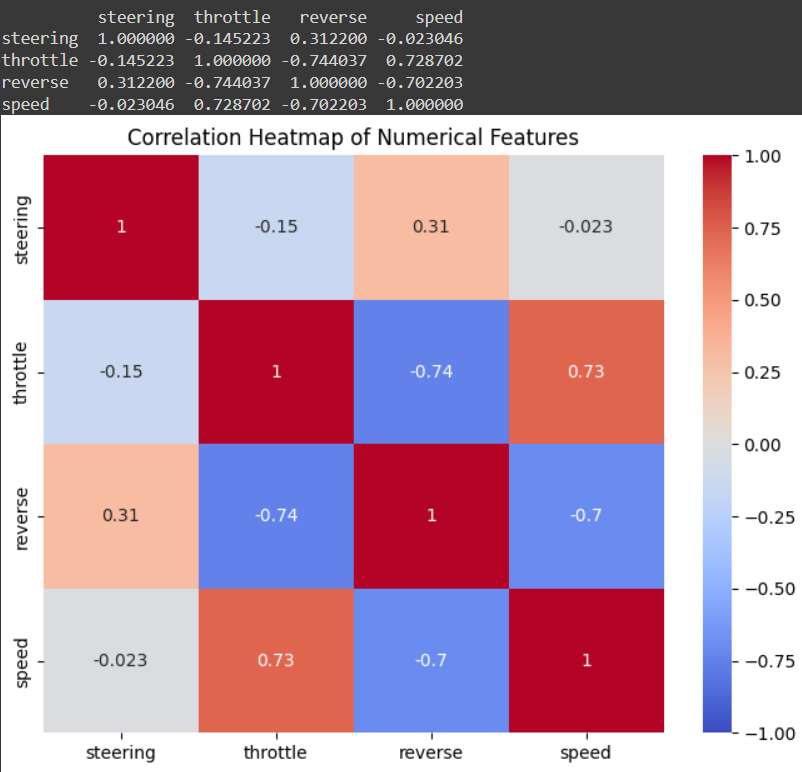


The code print(data[['steering','throttle','reverse','speed']].describe()) generates and prints descriptive statistics for the columns steering, throttle, reverse, and speed from the DataFrame data.

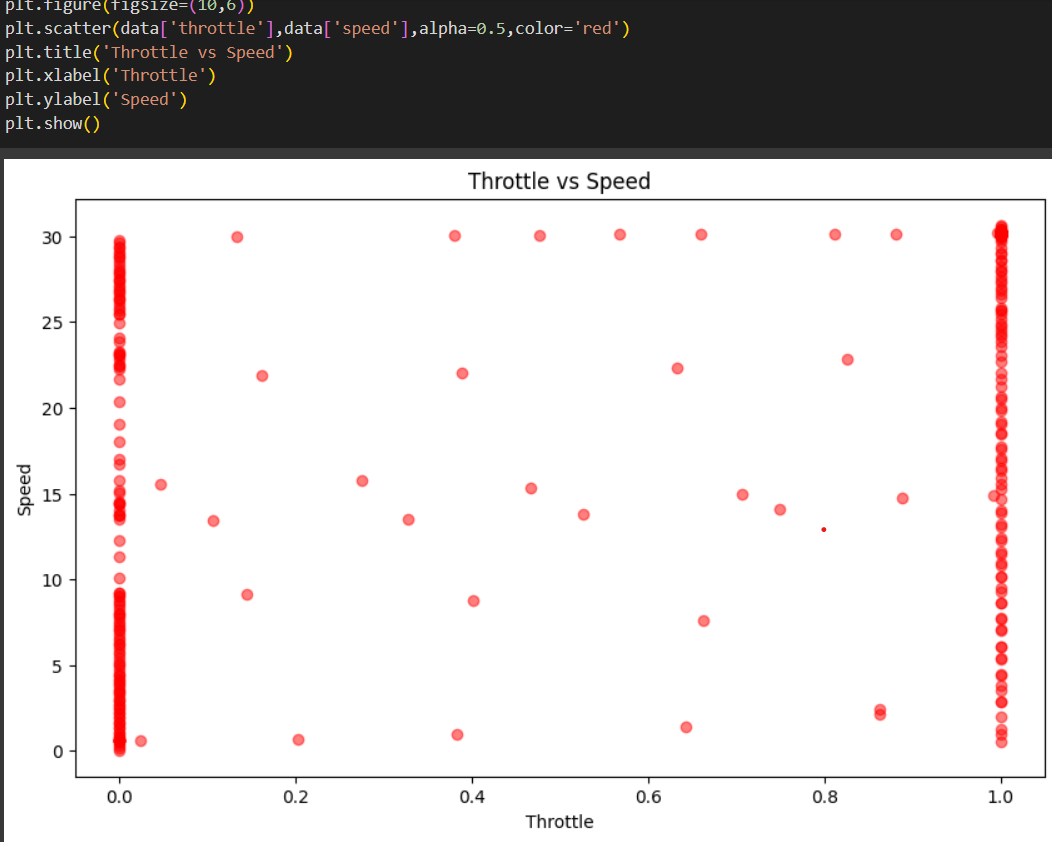


Distribution of the throttle on the dataset with its frequency.





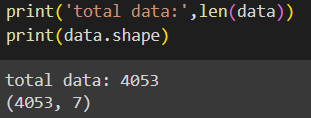
The code calculates and prints the correlation matrix for the steering, throttle, reverse, and speed columns to show how these features are related. It then visualizes this correlation matrix using a heatmap, providing a clear graphical representation of the relationships between the numerical features with colour coding to indicate the strength and direction of the correlations.



The code creates a scatter plot to visualize the relationship between throttle and speed from the DataFrame. It plots throttle on the x-axis and speed on the y-axis, using red dots to represent data points with some transparency (alpha=0.5). The plot helps in understanding how throttle input affects vehicle speed, with the title and axis labels providing context.



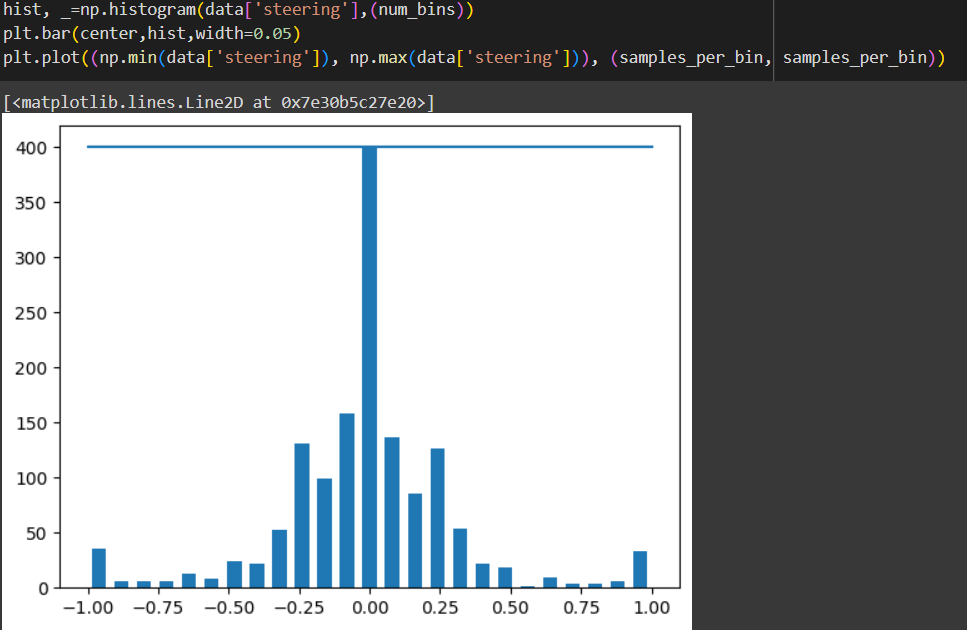
The code creates a histogram to visualize the distribution of steering angles. It divides the steering angle data into 25 bins and calculates the number of samples in each bin. The histogram is then plotted as a bar chart, with a horizontal line indicating a reference value of samples\_per\_bin. The plot displays how steering angles are distributed across different ranges, with the x-axis representing steering angles and the y-axis showing the number of samples per bin.



Total length of the dataset and corresponding shape of the data using data.shape.

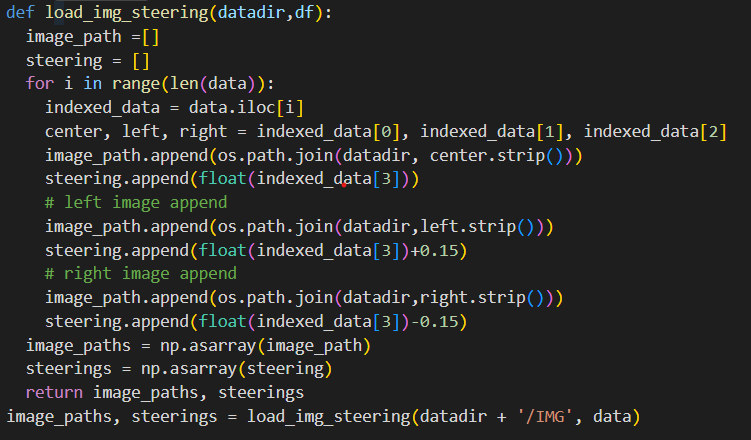
# Removing the biasing of the certain angle in the data:

The code addresses a bias in the steering angle data by removing excess samples to ensure a more balanced dataset. It first initializes a list to track which samples to remove. For each range of steering angles (bin), it collects all indices where the steering angles fall within that bin, shuffles these indices, and selects samples exceeding a specified limit (samples\_per\_bin) to be removed. After processing all bins, the code drops these indices from the dataset, thereby reducing bias towards particular steering angles and improving the model's ability to generalize, especially on curves. The code prints the number of samples removed and remaining to confirm the adjustment.



After removing biasing we can see that certain angle around 0.00 can also be considered for the car and can better generalises on the curves.

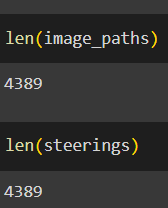
# Function to load images and their steering angles:



The code defines a function load\_img\_steering to prepare image paths and corresponding steering angles for training a model. Here’s how it works:

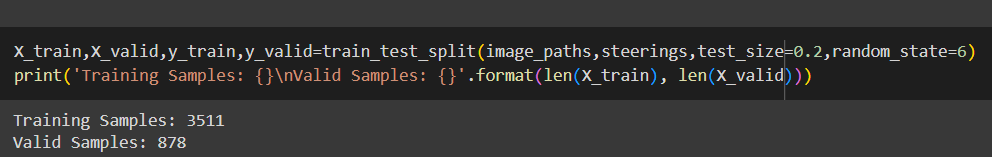
1. **Initialization**: The function initializes two empty lists, image\_path and steering, to store image file paths and their associated steering angles.
2. **Processing Data**: It iterates over each row in the DataFrame df (named data here):
   * **Extract Image Paths**: For each row, it extracts paths for the center, left, and right camera images and appends them to image\_path.
   * **Associate Steering Angles**: For each image path, it appends the steering angle with a slight adjustment to simulate different driving scenarios:
     + The center image uses the recorded steering angle.
     + The left image uses the steering angle increased by 0.15.
     + The right image uses the steering angle decreased by 0.15.
3. **Convert to Arrays**: After processing all rows, the lists image\_path and steering are converted to NumPy arrays.
4. **Return**: The function returns the arrays image\_paths and steerings, which can be used for training the model.

The final line calls this function, providing the path to the image directory (datadir + '/IMG') and the DataFrame data, and stores the results in image\_paths and steerings.



Length of images and steering angle are interconnected with each other and this we wanted so that we can predict the steering angle for the particular image.

# Split the data into training and validation sets:

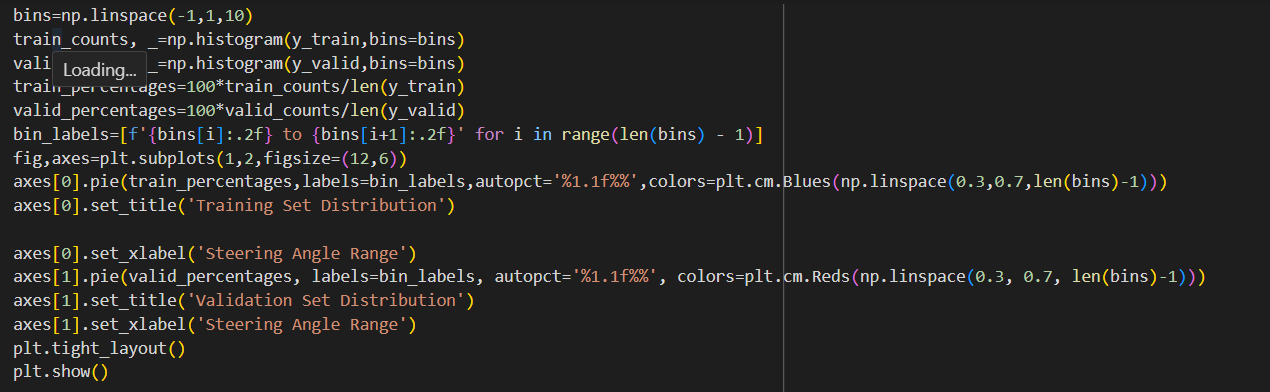


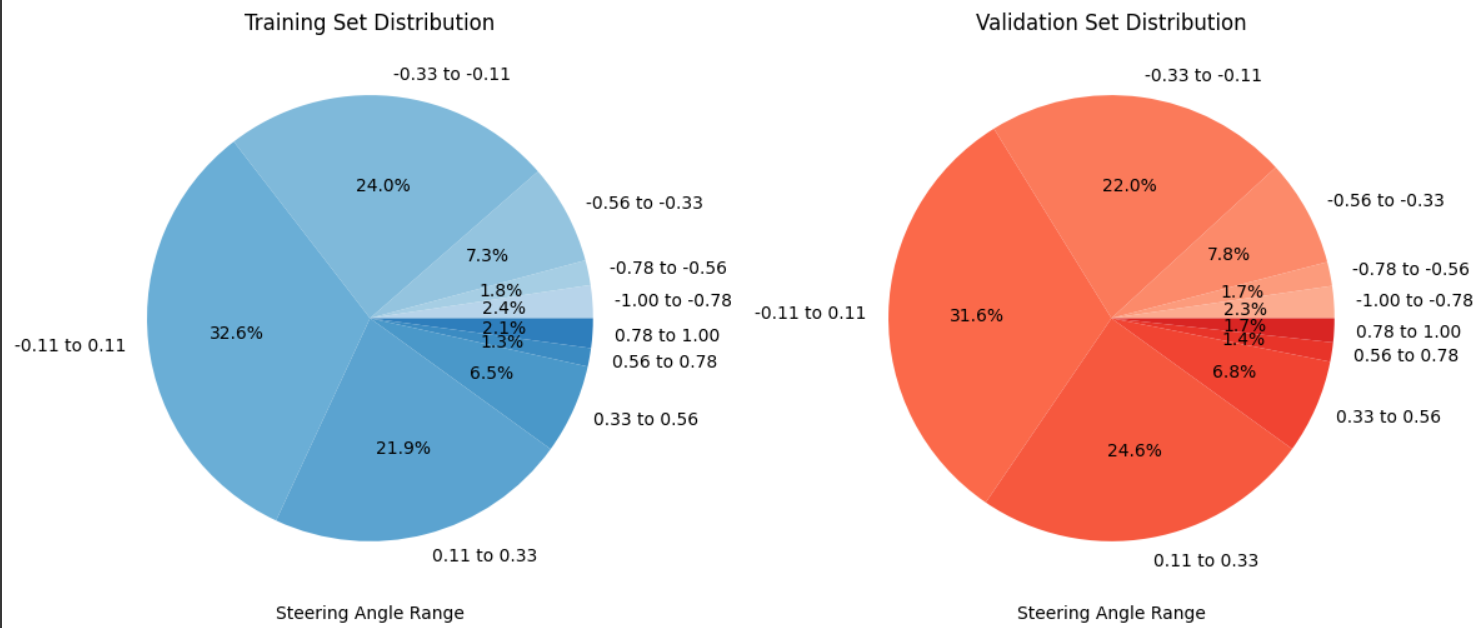
Split the training and validation data test size=0.2 and training samples are :3511 and valid samples are :878.

# Visualize the training set and validation set:



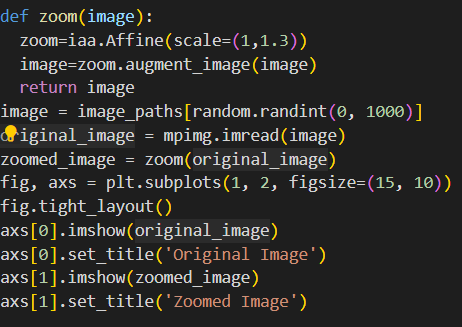
The code plots two histograms side by side: one for the steering angles in the training set (in blue) and one for the validation set (in red). Each histogram shows the distribution of steering angles with the same bin size and width, complete with titles and axis labels for easy comparison.

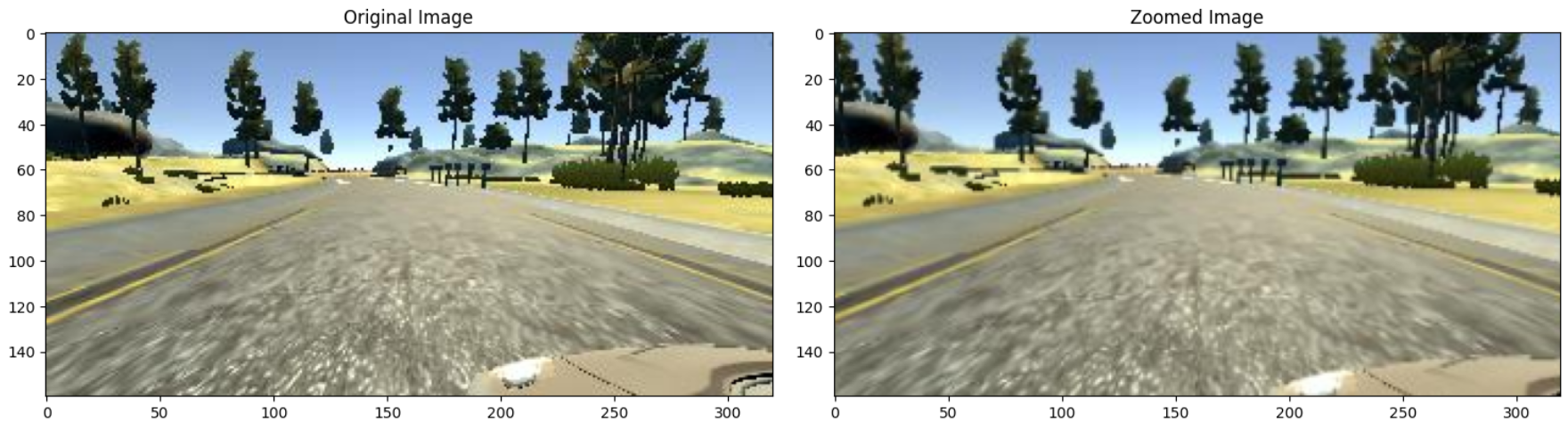




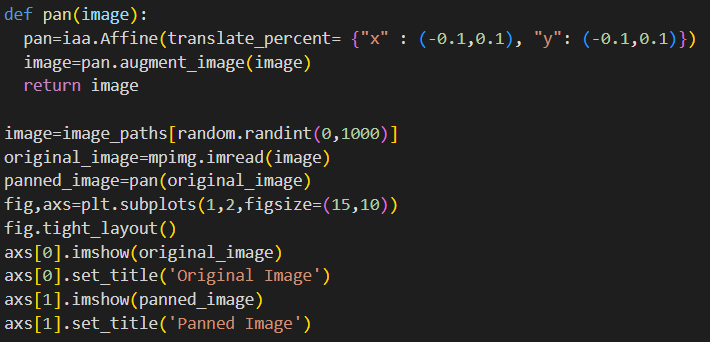
The code generates two pie charts to compare the distribution of steering angles in the training and validation sets. It calculates the percentage of samples within each steering angle range for both sets and creates pie charts to visualize these distributions. The training set is shown in shades of blue, while the validation set uses shades of red. Each pie chart includes percentage labels and is presented side by side for easy comparison.

# Image Augmentations:

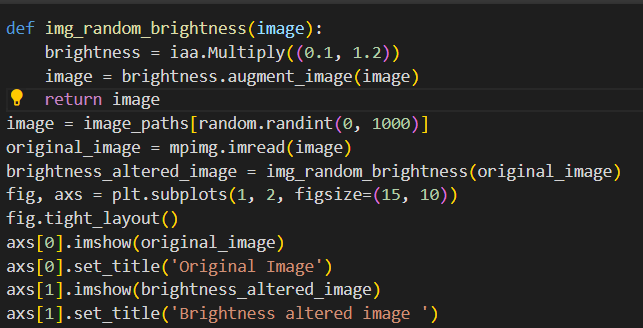


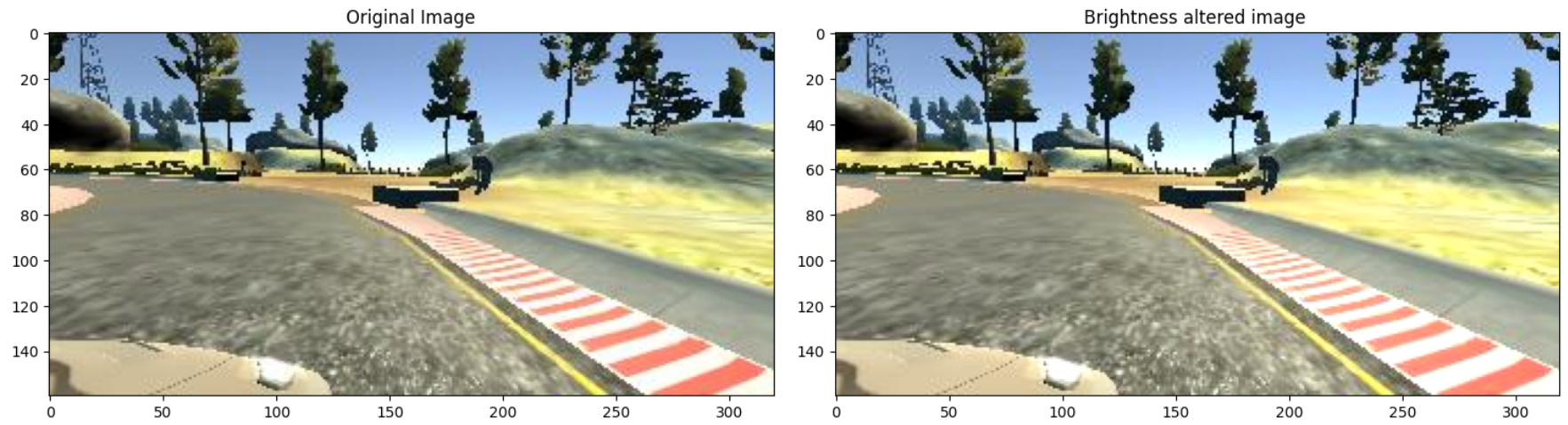


Applied Zoom augmentation. The code defines a function zoom that applies a zoom augmentation to an image using the imgaug library. It randomly selects an image from the image\_paths list, reads the image, and then applies the zoom effect. Finally, it displays the original and zoomed images side by side for comparison using Matplotlib.

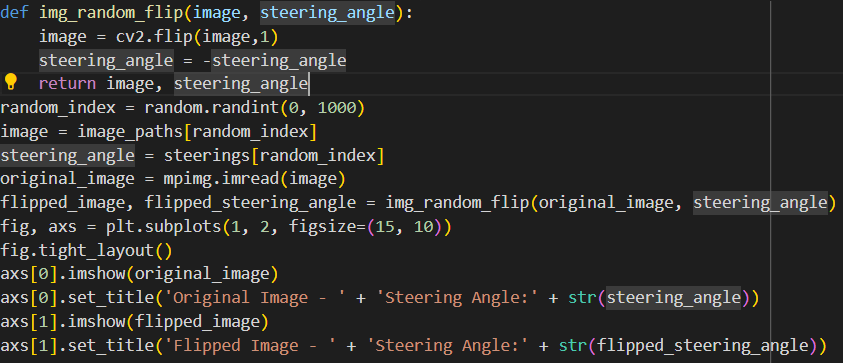


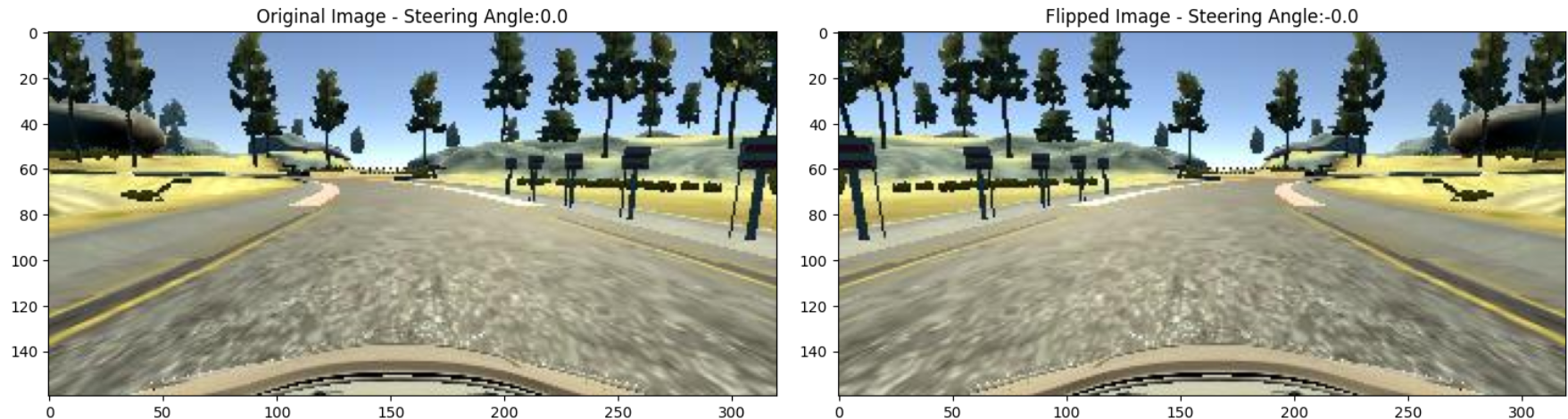
Applied pan augmentation. The code defines a pan function that applies a panning transformation to an image using imgaug. It randomly selects an image from the image\_paths list, reads the image, and applies horizontal and vertical shifts. The original and panned images are then displayed side by side using Matplotlib for comparison.



****

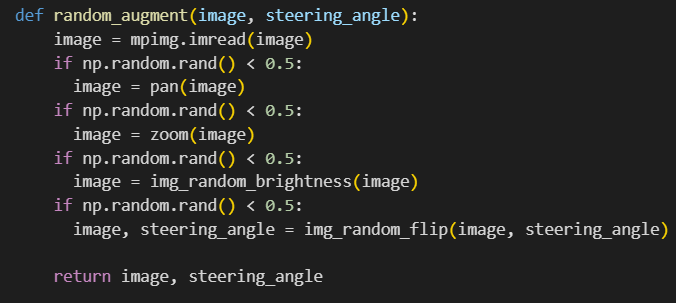
The code defines a function img\_random\_brightness that adjusts the brightness of an image using imgaug. It selects a random image from the image\_paths list, reads it, and applies a brightness change with a random factor between 0.1 and 1.2. The original and brightness-adjusted images are then displayed side by side using Matplotlib for visual comparison.

****

****

The code defines a function img\_random\_flip that flips an image horizontally and adjusts the steering angle accordingly. When an image is flipped, the steering angle must be inverted (negated) to reflect the change in direction. The function is applied to a randomly selected image and its corresponding steering angle. Both the original and flipped images, along with their steering angles, are displayed side by side using Matplotlib for comparison.

# Function to apply random augmentations to images:

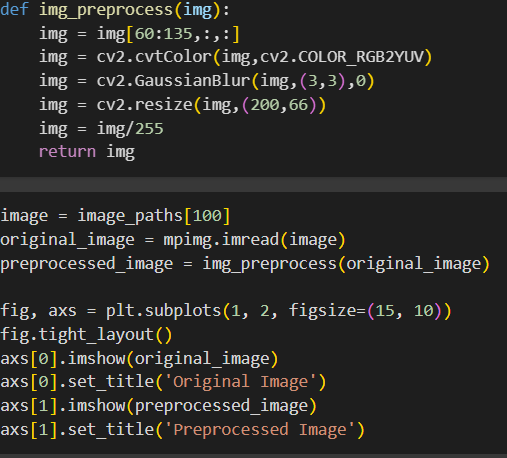


The random\_augment function performs a series of random image augmentations on a given image and its associated steering angle. Here’s a detailed breakdown:

1. **Read the Image**: The function starts by loading the image from the provided file path using mpimg.imread(image). This function reads the image file and returns it as an array.
2. **Random Pan Augmentation**: With a 50% probability (np.random.rand() < 0.5), the function applies a panning transformation to the image using the pan function. This may shift the image horizontally or vertically, simulating slight movements of the camera.
3. **Random Zoom Augmentation**: Similarly, with a 50% probability, it applies a zoom effect to the image using the zoom function. This transformation scales the image, creating the effect of zooming in or out.
4. **Random Brightness Adjustment**: With another 50% chance, the function adjusts the brightness of the image using the img\_random\_brightness function. This alteration varies the brightness level, simulating different lighting conditions.
5. **Random Flip Augmentation**: Finally, with a 50% probability, the image is flipped horizontally using the img\_random\_flip function. This function also inverts the steering angle to reflect the change in the direction the vehicle would need to steer after flipping.

**Return Augmented Image and Steering Angle**: After applying the augmentations, the function returns the modified image and updated steering angle.

# Images Preprocessing:



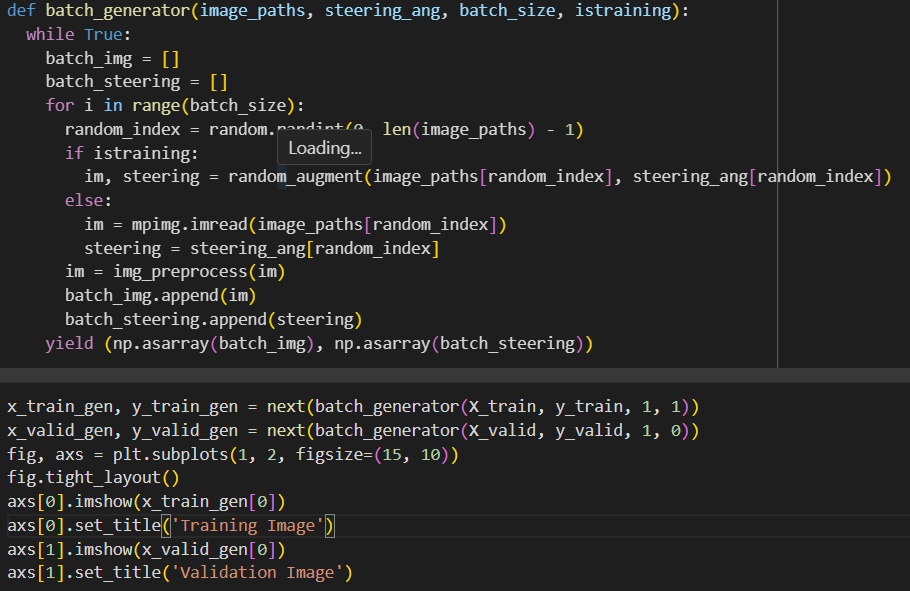


The img\_preprocess function prepares images for training by performing the following steps:

1. **Crop**: Focuses on the region of interest by cropping the top 60 pixels and retaining 75 pixels in height.
2. **Convert Color Space**: Changes the image from RGB to YUV color space to separate brightness from color information.
3. **Apply Gaussian Blur**: Smooths the image to reduce noise and enhance feature detection.
4. **Resize**: Standardizes image dimensions to 200x66 pixels, ensuring consistent input size for the model.
5. **Normalize**: Scales pixel values to the range [0, 1] to improve model learning and convergence.

These preprocessing steps ensure that the images are well-prepared, focusing on relevant features and standardizing inputs for effective training in the self-driving car simulator.

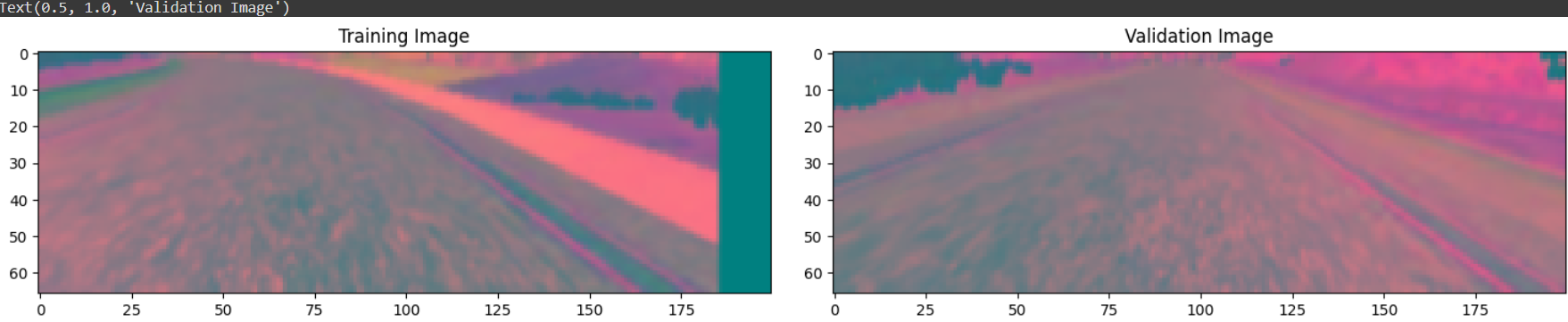
# Generator Function:



The batch\_generator function is designed to efficiently provide batches of images and corresponding steering angles for model training or validation. Here’s a detailed overview:

1. **Infinite Loop**: The function operates in an infinite loop, continuously generating batches of data. This is particularly useful for training models over multiple epochs, ensuring that the model receives a consistent flow of training examples.
2. **Batch Initialization**: It initializes empty lists to store the images and steering angles for the current batch.
3. **Batch Creation**:
   * **Random Sampling**: For each batch, the function selects random indices to fetch images and steering angles from the provided dataset.
   * **Augmentation**: During training (istraining=True), it applies random augmentations to the images to enhance the diversity of the training data and improve model robustness. This includes transformations such as panning, zooming, adjusting brightness, and flipping.
   * **Direct Loading**: During validation or testing (istraining=False), the function loads images directly without applying augmentations.
4. **Image Preprocessing**: Each image is preprocessed to ensure consistency in the input data. This involves cropping, converting to the YUV color space, applying Gaussian blur, resizing to a standard dimension, and normalizing pixel values.
5. **Batch Assembly**: The preprocessed images and steering angles are collected into lists and then converted into NumPy arrays. This enables efficient data handling and feeding into the model.
6. **Yielding Batches**: The function yields the batch of images and steering angles as NumPy arrays, facilitating seamless integration with model training or evaluation processes.

This approach ensures that the model is trained on diverse and well-preprocessed data, improving its generalization and performance.

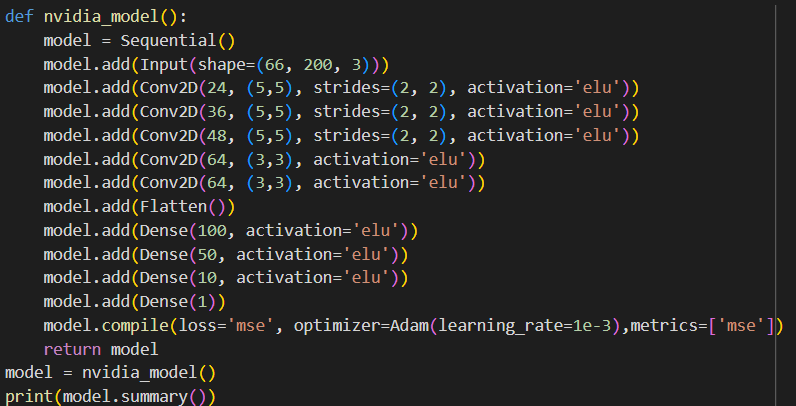


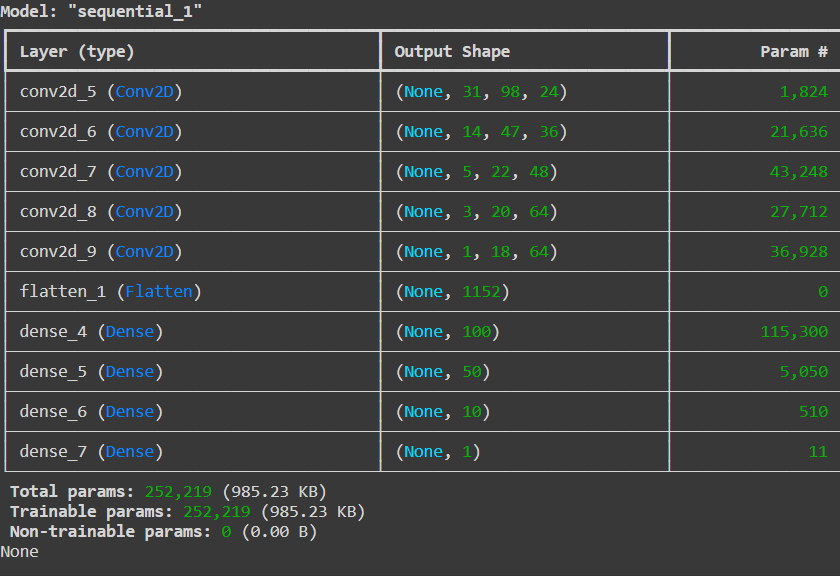
In this section, we demonstrate the process of retrieving and visualizing sample images from the training and validation datasets.

1. **Batch Retrieval**:
   * We use the batch\_generator function to obtain a single batch of images and corresponding steering angles from both the training and validation datasets. For the training dataset, we specify that the batch should include data augmentation. For the validation dataset, no augmentation is applied.
2. **Visualization**:
   * The retrieved images from both datasets are visualized using Matplotlib. We present two side-by-side images:
     + **Training Image**: This image represents a sample from the training dataset, showcasing the preprocessing and augmentation applied to the data.
     + **Validation Image**: This image shows a sample from the validation dataset, demonstrating the raw preprocessing without additional augmentation.

This visualization helps in understanding the impact of preprocessing and augmentation on the training and validation data, ensuring that the images fed into the model are well-prepared and representative of the conditions the model will encounter.

# NVIDIA Behavioral Cloning Model Architecture:





The NVIDIA architecture for the behavioral cloning model is a Convolutional Neural Network (CNN) designed to predict steering angles based on images. Here's a simplified explanation of each layer and its role in the model:

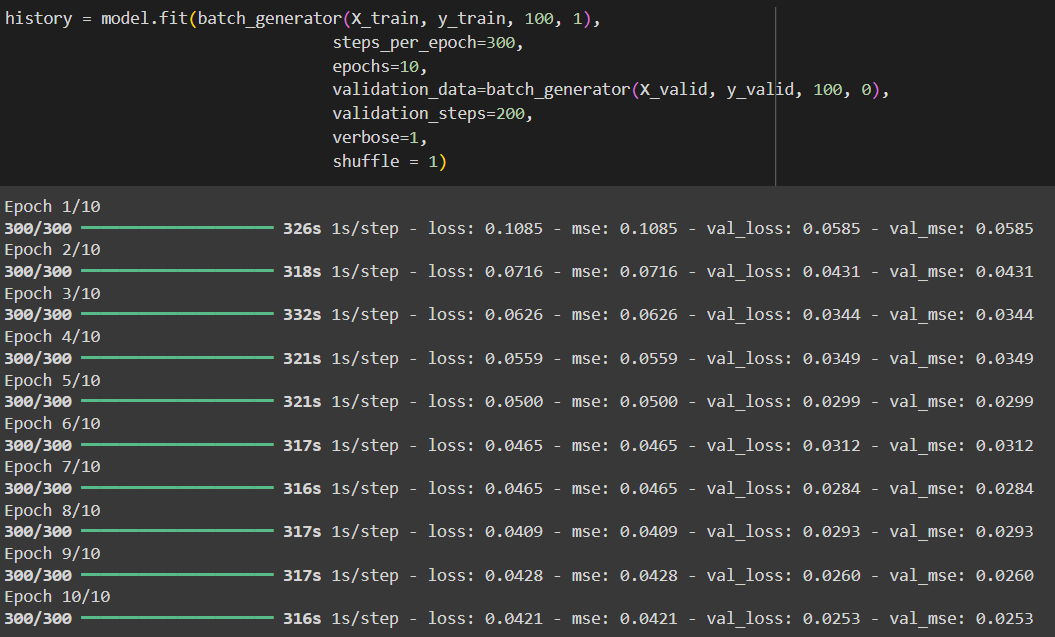
1. **Input Layer**:
   * **Shape**: (66, 200, 3)
   * **Purpose**: Accepts input images of size 66x200 pixels with 3 color channels (RGB).
2. **Convolutional Layers**:
   * **Conv2D Layers**:
     + **First Layer**: Applies 24 filters of size 5x5 with a stride of 2x2, using the ELU activation function.
     + **Second Layer**: Applies 36 filters of size 5x5 with a stride of 2x2, also using ELU activation.
     + **Third Layer**: Applies 48 filters of size 5x5 with a stride of 2x2, with ELU activation.
     + **Fourth Layer**: Applies 64 filters of size 3x3 with ELU activation.
     + **Fifth Layer**: Applies another 64 filters of size 3x3 with ELU activation.
   * **Purpose**: Convolutional layers detect features from the input images. The filters help in identifying patterns such as edges, shapes, and textures. The use of strides reduces the spatial dimensions while preserving important features.
3. **Flatten Layer**:
   * **Purpose**: Converts the 2D feature maps from the convolutional layers into a 1D vector. This prepares the data for the fully connected (dense) layers.
4. **Dense Layers**:
   * **First Dense Layer**: 100 neurons with ELU activation.
   * **Second Dense Layer**: 50 neurons with ELU activation.
   * **Third Dense Layer**: 10 neurons with ELU activation.
   * **Output Dense Layer**: 1 neuron without activation, predicting the steering angle.
   * **Purpose**: Dense layers interpret the features extracted by the convolutional layers. The final output layer provides the predicted steering angle.
5. **Activation Function - ELU**:
   * **Purpose**: ELU (Exponential Linear Unit) helps avoid the problem of dead neurons, where some neurons can become inactive and not contribute to the model's learning. ELU allows negative values, which helps the network learn better and reduces dead neuron issues compared to ReLU (Rectified Linear Unit).
   * **Dead ReLU Problem**: ReLU can lead to dead neurons where gradients are zero, making them inactive. ELU mitigates this by allowing small negative outputs and thus maintaining gradient flow.
6. **Compilation**:
   * **Loss Function**: Mean Squared Error (MSE)
     + **Purpose**: Measures the average squared difference between the predicted steering angles and the actual values. It is suitable for regression tasks like predicting steering angles.
   * **Optimizer**: Adam with a learning rate of 0.001
     + **Purpose**: Adam is an adaptive learning rate optimizer that adjusts the learning rate during training, helping to converge faster and more reliably.

**Input and Output**

* **Input**: The model takes images of size 66x200 pixels with 3 color channels (RGB).
* **Output**: The model predicts a single steering angle as a continuous value.

This architecture and the associated configurations (activation functions, loss function, and optimizer) are designed to efficiently process images and predict accurate steering angles, making it suitable for training a self-driving car model.

# Training Model:



The model.fit function is used to train a machine learning model.

1. **batch\_generator(X\_train, y\_train, 100, 1)**:

* **Purpose**: Provides the training data in batches.
* **Function**: batch\_generator is a function that yields batches of images and steering angles. Here, it is used to generate batches of size 100 for training.
* **Parameters**:
  + X\_train: Training images.
  + y\_train: Corresponding steering angles.
  + 100: Batch size.
  + 1: Indicates that data augmentation should be applied (training mode).

1. **steps\_per\_epoch=300**:

* **Purpose**: Defines the number of batches to be processed in each epoch.
* **Function**: The model will process 300 batches per epoch. Each batch contains 100 samples, so it processes a total of 30,000 samples per epoch.

1. **epochs=10**:

* **Purpose**: Specifies the number of times the entire dataset will be passed through the model.
* **Function**: The model will be trained for 10 complete passes through the training data.

1. **validation\_data=batch\_generator(X\_valid, y\_valid, 100, 0)**:

* **Purpose**: Provides validation data for evaluating the model’s performance during training.
* **Function**: Similar to the training data generator, but here it generates batches of validation data.
* **Parameters**:
  + X\_valid: Validation images.
  + y\_valid: Corresponding steering angles.
  + 100: Batch size.
  + 0: Indicates no data augmentation (validation mode).

1. **validation\_steps=200**:

* **Purpose**: Defines the number of validation batches to be processed at the end of each epoch.
* **Function**: The model will evaluate 200 batches of validation data per epoch.

1. **verbose=1**:

* **Purpose**: Controls the level of detail shown during training.
* **Function**: With verbose=1, the training progress will be shown in the console, including the loss and accuracy metrics for each epoch.

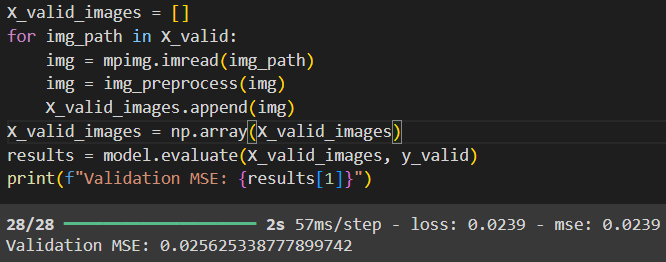
1. **shuffle=1**:

* **Purpose**: Determines whether the training data should be shuffled before each epoch.
* **Function**: Setting shuffle=1 (or True) means that the training data will be shuffled before each epoch, which helps in improving the model's ability to generalize.

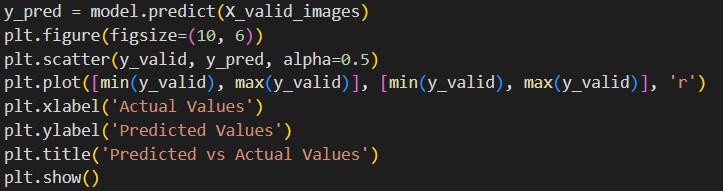
1. **history**:

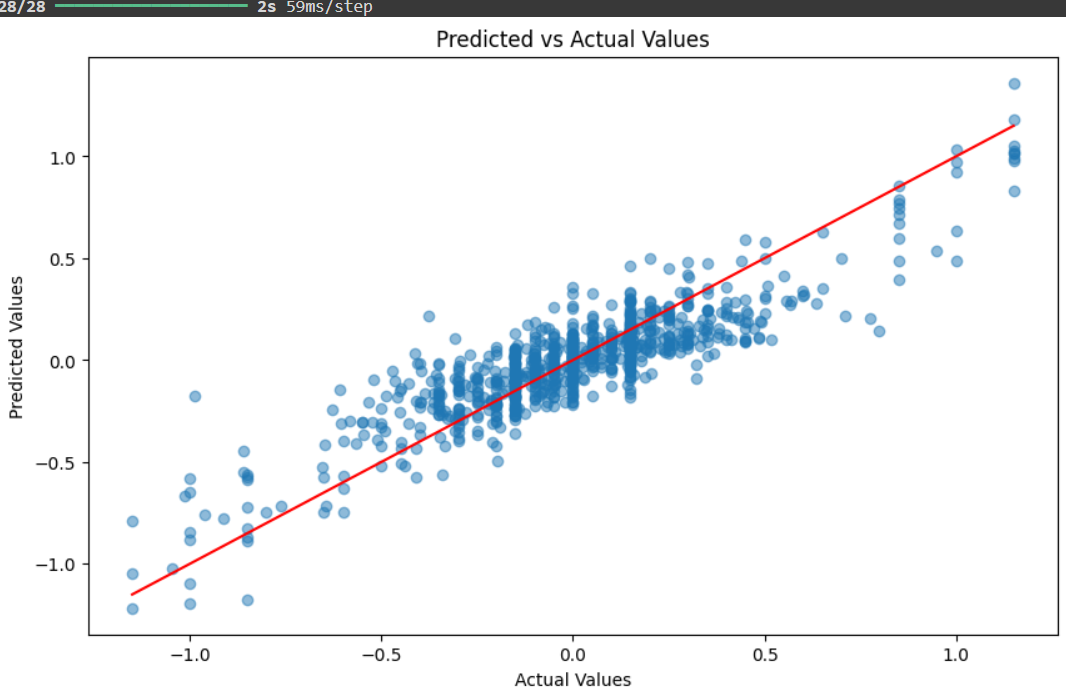
* **Purpose**: Stores the training history.
* **Function**: The history object contains information about the training process, such as the loss and accuracy metrics for each epoch. This can be used for further analysis or visualization of training performance.

# Model Evaluation:



The code snippet processes and evaluates the validation data for the model. It begins by initializing an empty list X\_valid\_images to store preprocessed validation images. For each image path in X\_valid, the code reads the image using mpimg.imread, preprocesses it with the img\_preprocess function, and appends the processed image to the list. Once all images are processed, they are converted into a NumPy array X\_valid\_images. The model's performance is then assessed using the model.evaluate function, which calculates the mean squared error (MSE) between the predicted and actual steering angles on this validation dataset. Finally, the validation MSE is printed, providing an indication of the model's accuracy on unseen data.

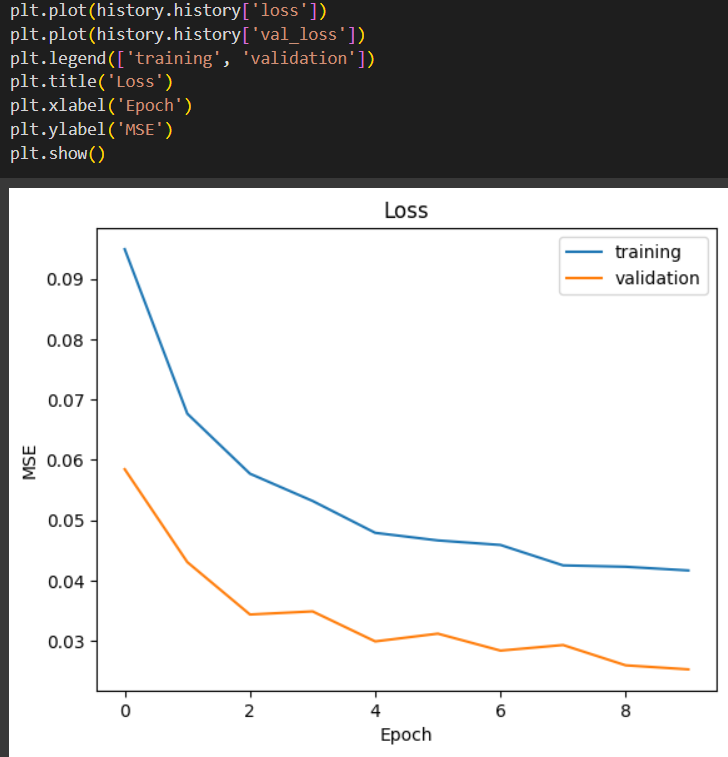




This code snippet visualizes the model's predictions against the actual steering angles for the validation set.

1. **Prediction**: The line y\_pred = model.predict(X\_valid\_images) uses the trained model to generate predictions (y\_pred) for the validation images (X\_valid\_images).
2. **Visualization**:
   * **Plot Setup**: plt.figure(figsize=(10, 6)) sets up the size of the plot.
   * **Scatter Plot**: plt.scatter(y\_valid, y\_pred, alpha=0.5) creates a scatter plot where the x-axis represents the actual steering angles (y\_valid), and the y-axis represents the predicted steering angles (y\_pred). The alpha=0.5 argument makes the points slightly transparent.
   * **Diagonal Line**: plt.plot([min(y\_valid), max(y\_valid)], [min(y\_valid), max(y\_valid)], 'r') adds a red diagonal line representing where predicted values perfectly match actual values. This line helps visualize the accuracy of predictions: points close to this line indicate good predictions.
   * **Labels and Title**: plt.xlabel('Actual Values') and plt.ylabel('Predicted Values') label the x and y axes, respectively, while plt.title('Predicted vs Actual Values') sets the plot title.
3. **Display Plot**: plt.show() renders the plot, providing a visual comparison between the actual and predicted steering angles.

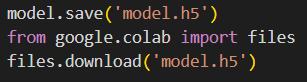
This visualization helps assess how well the model's predictions align with the true values, with points ideally clustering around the red diagonal line.



This code snippet is used to visualize the training and validation loss over epochs during the model's training process.

1. **Plot Training Loss**: plt.plot(history.history['loss']) plots the training loss values recorded during each epoch. The history.history['loss'] contains the loss values for the training data after each epoch.
2. **Plot Validation Loss**: plt.plot(history.history['val\_loss']) plots the validation loss values recorded during each epoch. The history.history['val\_loss'] contains the loss values for the validation data after each epoch.
3. **Add Legend**: plt.legend(['training', 'validation']) adds a legend to the plot to differentiate between the training and validation loss curves.
4. **Set Title and Labels**:
   * plt.title('Loss') sets the title of the plot to "Loss".
   * plt.xlabel('Epoch') labels the x-axis as "Epoch", indicating the number of epochs over which the model was trained.
   * plt.ylabel('MSE') labels the y-axis as "MSE", representing the Mean Squared Error of the loss.
5. **Display Plot**: plt.show() renders the plot.

# Saving And Downloading Model:



This first saves the trained model using model.save('model.h5'), which stores the model’s architecture and weights in a file named model.h5 using the HDF5 format. This format is commonly used for its efficiency in handling large datasets and its ability to store complex data structures. Next, the code imports the files module from Google Colab and uses files.download('model.h5') to initiate the download of the saved model file to the local machine. This process allows you to transfer the model from the cloud environment in Colab to your local storage, enabling you to preserve the trained model for future use or sharing with others.

CONCLUSION

In this project, we undertook the development of a sophisticated behavioral cloning model leveraging the NVIDIA architecture to accurately predict steering angles for autonomous vehicles. We began by assembling a comprehensive dataset comprised of driving logs, which included images captured from the vehicle’s perspective alongside their corresponding steering angles. This dataset served as the foundation for training and validating our model. To enhance the model’s robustness and generalization capabilities, we implemented a series of preprocessing and augmentation techniques. These methods included image cropping, resizing, and applying transformations such as zooming, panning, and brightness adjustments to simulate various driving conditions and improve the model's performance under different scenarios.

For training, we employed Mean Squared Error (MSE) as the loss function, which measures the average squared difference between the predicted and actual steering angles, making it suitable for this regression task. The Adam optimizer was chosen with a learning rate of 0.001 to facilitate efficient and reliable convergence of the model. This combination of loss function and optimizer allowed for effective learning and adaptation during training. The results, as reflected in the loss curves and visualization of predictions versus actual values, demonstrated the model's ability to generalize well from training to validation data. Additionally, the adoption of the Exponential Linear Unit (ELU) activation function played a crucial role in addressing the problem of dead neurons, which can occur with traditional ReLU activation. By mitigating this issue, ELU ensured a smoother training process and contributed to more effective learning, ultimately enhancing the model’s overall performance and reliability.

Looking ahead, there are several avenues to enhance and expand this project. Incorporating more sophisticated augmentation techniques and additional sensor data, such as radar or LiDAR, could improve the model’s robustness and adaptability to various driving conditions. Exploring more advanced neural network architectures, like those integrating recurrent layers for better temporal understanding, could further refine the model’s accuracy. Additionally, implementing real-time testing and deployment in actual driving scenarios could validate the model's performance in practical applications. Future work could also involve integrating reinforcement learning to enable the model to learn from live driving experiences, potentially leading to more autonomous and adaptive driving systems.