CSE 445.6

Machine Learning

Report

On

Develop a machine learning model for autonomous driving, enabling the car to navigate without human intervention.

Submitted to
MSRB

Submitted by

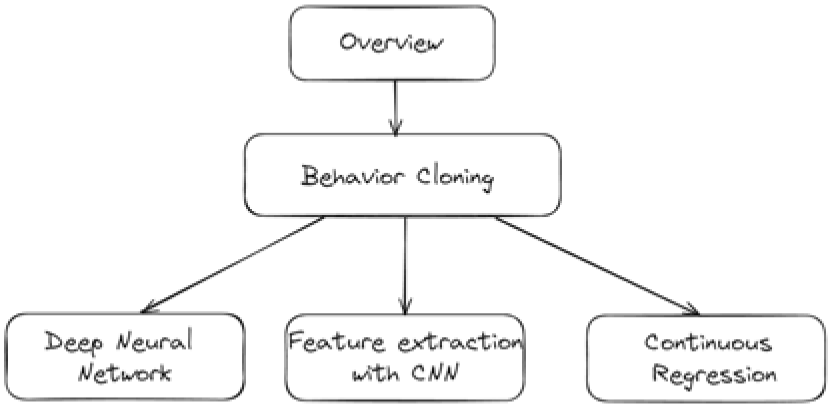
Group 3

Name	ID
MD.Shafat Islam Khan	2121517642
Labiba Faizah	2012230642
Adiba Hossain Lorin	2012039042

Project Title: Develop a machine learning model for autonomous driving, enabling the car to navigate without human intervention.

Introduction:

The method we have used in this project is called Behavior Cloning using NVIDIA model. Behavior cloning in machine learning is a technique where a model learns to imitate the actions of an expert by observing examples of those actions. From this method we have used Deep neural network, for feature extraction we have used Convolutional Neural Network (CNN) and for the last part, usually behavior cloning is a classification problem but for our case it is a continuous regression.



CNN (Convolutional Neural Network)

A Convolutional Neural Network (CNN), or ConvNet, is a specialized deep learning model that is particularly well-suited for image recognition and processing tasks such as image classification, detection, and segmentation. CNNs are utilized in various practical applications, including autonomous vehicles, security camera systems, and more.
It is made up of multiple layers, including convolutional layers, pooling layers, and fully connected layers. The architecture of CNNs is inspired by the visual processing in the human brain, and they are well-suited for capturing hierarchical patterns and spatial dependencies within images.

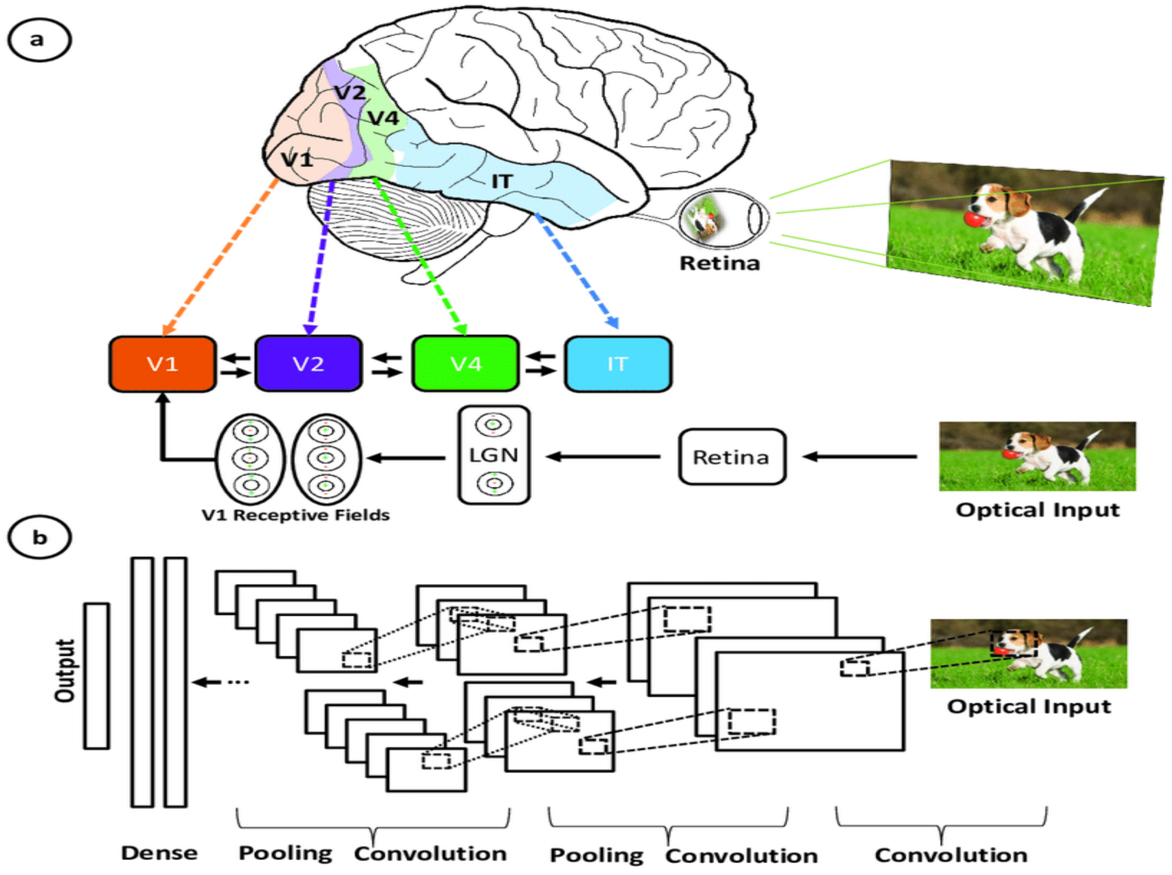


Illustration of the correspondence between the areas associated with the primary visual cortex and the layers in a convolutional neural network

Key components of a Convolutional Neural Network include:

Convolutional Layers: These layers perform convolutional operations on input images using filters (or kernels) to identify features like edges, textures, and more intricate patterns. Convolution operations maintain the spatial relationships between pixels.

Pooling Layers: Pooling layers downsample the spatial dimensions of the input, which reduces the computational load and the number of parameters in the network. Max pooling is a common technique, where the highest value from a group of neighboring pixels is selected.

Activation Functions: Non-linear activation functions, such as the Rectified Linear Unit (ReLU), add non-linearity to the model, enabling it to learn more complex data patterns.

Fully Connected Layers: These layers make predictions based on the high-level features identified by earlier layers. They connect every neuron in one layer to every neuron in the subsequent layer.

CNNs are trained on a large collection of labeled images, where the network learns to identify patterns and features linked to particular objects or categories. CNNs are widely used in areas such as image classification, object detection, facial recognition, and medical image analysis. The convolutional layers are the key component of a CNN, where filters are applied to the input image to extract features such as edges, textures, and shapes.

The training process for a CNN involves the following steps:

1. **Data Preparation:** The training images are preprocessed to ensure that they are all in the same format and size.
2. **Loss Function:** A loss function is used to measure how well the CNN is performing on the training data. The loss function is typically calculated by taking the difference between the predicted labels and the actual labels of the training images.
3. **Optimizer:** An optimizer is used to update the weights of the CNN in order to minimize the loss function.
4. **Backpropagation:** It is a technique used to calculate the gradients of the loss function with respect to the weights of the CNN. The gradients are then used to update the weights of the CNN using the optimizer.

The efficiency of a CNN on picture categorization tasks can be evaluated using a variety of criteria. Among the most popular metrics are:

- **Accuracy:** Accuracy is the percentage of test images that the CNN correctly classifies.

- **Precision:** Precision is the percentage of test images that the CNN predicts as a particular class and that are actually of that class.
- **Recall:** Recall is the percentage of test images that are of a particular class and that the CNN predicts as that class.
- **F1 Score:** The F1 Score is a harmonic mean of precision and recall. It is a good metric for evaluating the performance of a CNN on classes that are imbalanced.

We collected data from Udacity's simulation and also created our own training data by taking images while driving a car. For each image, we recorded the corresponding steering angle along with information about the car's speed, throttle, and brake status.

Here’s a simplified explanation of how the Convolutional Neural Network (CNN) processes these images:

5. **Data Collection**: We have images captured from the car's perspective while driving, and each image is tagged with the steering angle and other driving information (speed, throttle, brake).
6. **CNN Processing**: When an image goes through the CNN, the network has many layers of nodes (neurons) that process the image. These nodes perform mathematical operations to detect various features in the image, such as edges, shapes, and textures.
7. **Learning Patterns**: As the CNN processes more and more images, it learns which features are associated with different steering angles. For example, it might learn that a certain combination of road and car features corresponds to a left turn or a right turn.
8. **Prediction**: After training, when a new image is input into the CNN, the network uses what it has learned to predict the steering angle. This means the output of the CNN is the steering angle that the car should use based on what it sees in the image.

The CNN looks at an image, recognizes patterns, and predicts the correct steering angle to navigate the car based on its learned experience from the training data. The steering angle will move to right or left or straight, this will be predicted by convolutional neural network.

Data Pre-Processing:

Firstly, we removed unnecessary information of file path and just kept information of data that are needed like center, left, right.

```
def path_leaf(path):
  head, tail = ntpath.split(path)
  return tail
```

This function path_leaf takes a file path as input and returns the final part (or "leaf") of the path, which is typically the file name.

```
data['center'] = data['center'].apply(path_leaf)
data['left'] = data['left'].apply(path_leaf)
data['right'] = data['right'].apply(path_leaf)
data.head()
```

These lines apply the **path_leaf** function to each entry in the 'center', 'left', and 'right' columns of the **data** DataFrame.

This operation transforms each file path in these columns to just the file name.

Finally, **data.head()** displays the first few rows of the DataFrame to show the changes.

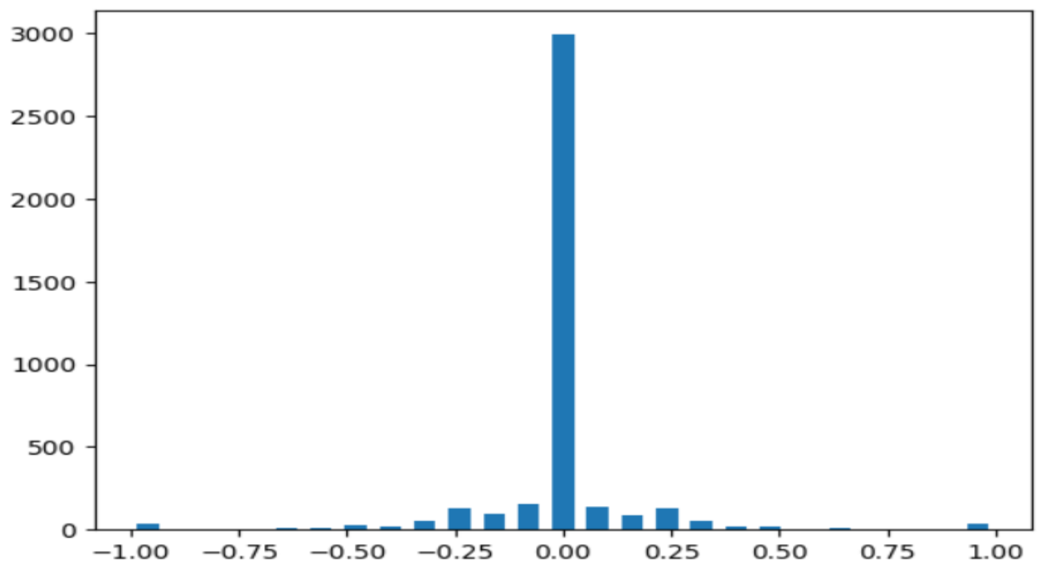
The code defines a function to extract file names from paths and applies this function to specific columns in a DataFrame, replacing full paths with just the file names.

While driving a car at the center, the steering angle of the car is declared as 0.00 in this model. In this data preprocessing, there's an array between –1 to 1 where there can be very minimal values like 0.05, -0.065 but there is no value which is exactly 0. All these minimal values are too close to 0. So, these values have been considered as 0. As a result, all these values are defined as center in steering angle. This implementation has been done through this code:

```
       center = (bins[:-1]+ bins[1:]) * 0.5
```

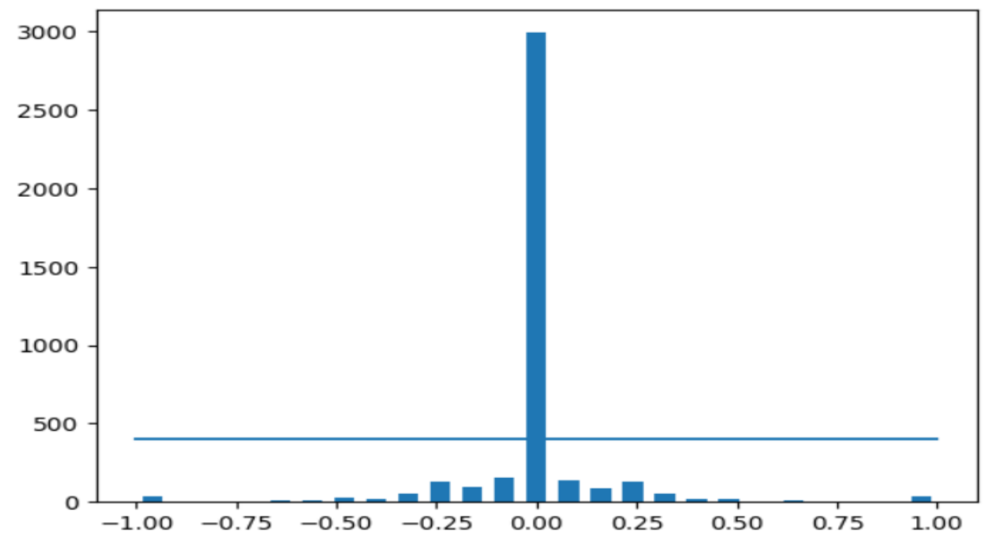
Data Visualization through a Histogram for steering angle:

After plotting all the data in the histogram, we found these data are more biased towards center because the nature of the track we used for training our data, was mostly straight and for this reason, the data we found were inaccurate.



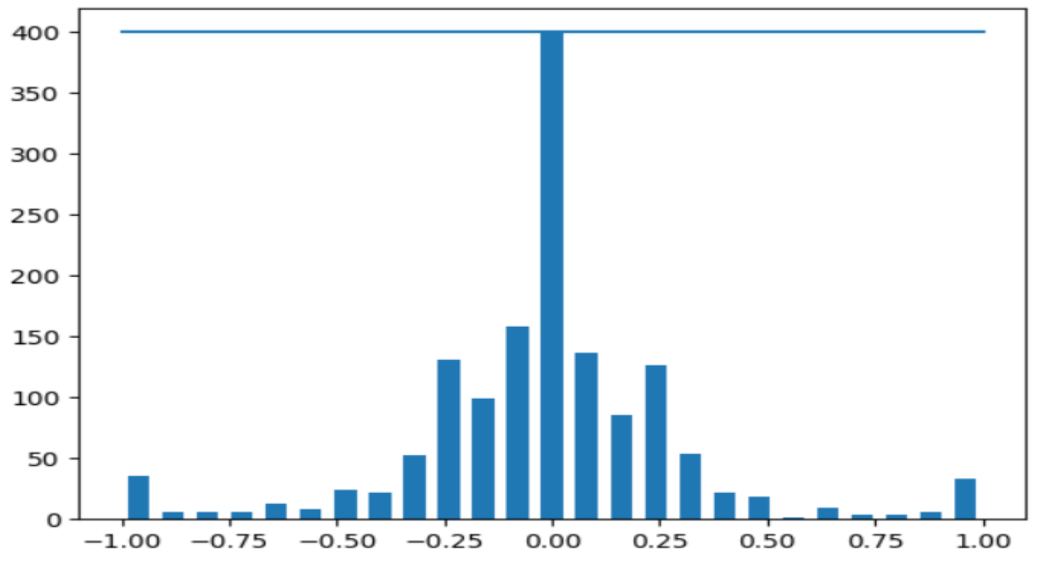
Making data uniform since steering angle is biased towards centre due to the nature of the track:

To solve this problem, we uniformed the data so that the steering angle does not stay biased towards center due to track of the nature. Firstly, we removed the unnecessary data from the histogram that are above 500 range and made the histogram smaller and zoomed. So basically, we have set a threshold of limit and removed the data which are above the range. In this way, unnecessary data have been isolated from the required data.



Isolation of data above the threshold:

But if we do not shuffle the data, every time similar types of data will be generated and as a result again the data will be biased towards the same steering angle. To avoid this circumstances, we shuffle all the bin or sample data continuously and finally we have got our desired histogram.



So the final data we got now:

- total data: 4053
- removed: 2590

- remaining: 1463

Splitting the data into Training and validation set making sure data is uniform:

To access the images corresponding steering angle dats, here the code prepares and augments image and steering angle data for training a driving model. The load_img_steering function takes a directory path and a dataframe, extracts the paths of center, left, and right images along with their steering angles, adjusts the angles for left and right images, and converts the data into numpy arrays. This helps the model learn better by simulating different driving perspectives. Suppose when the steering angle is 0, the three corresponding images (center, left, right) can be accessed through this phase.

Finally we have the data into two sets which are training set and validation set making sure the data is uniform. Then we've split data into a training set and validation set and now the data can be visualized. Here the test data size is 0.2 which means that 20% of data are going for test data and the rest 80% data are going into training and validation set. Finally, data preprocessing ends here.

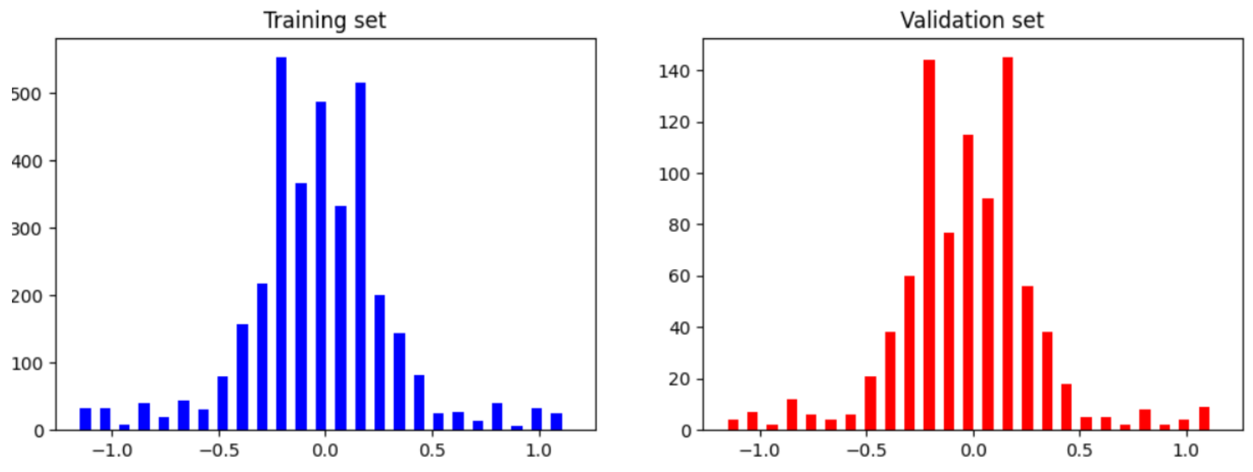


Image Preprocessing:

Each image will be stored as objects. In this model original images are in RGB colors but NVIDIA model that we have used suggests image colors to be in YUV format. Through this format, this model can perform more accurately. Here image can be resized according to the size we need. Such as:

```
    img = img[60:135,:,:]
```

Cropping: Cuts out the part of the image between row 60 and row 135, keeping all columns and color channels.

```
  img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
```

Color Conversion: Converts the image from RGB color space to YUV color space

```
img = cv2.GaussianBlur(img,  (3, 3), 0)
```

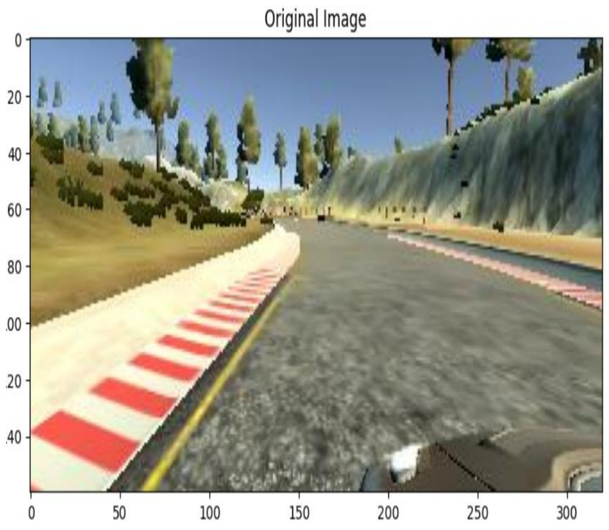
Blurring: Applies a Gaussian blur with a 3x3 kernel to reduce noise and detail.

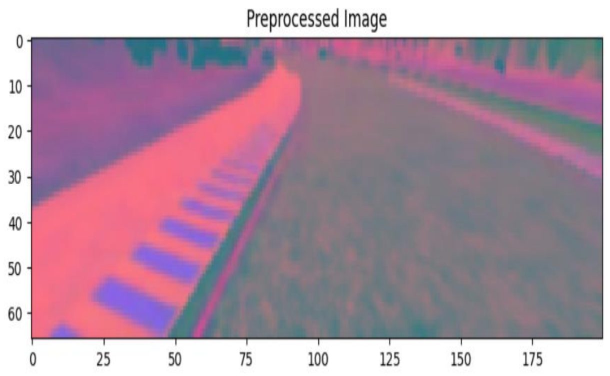
```
   img = cv2.resize(img, (200, 66))
```

- **Resizing**: Resizes the image to 200 pixels wide by 66 pixels high.

```
    img = img/255
```

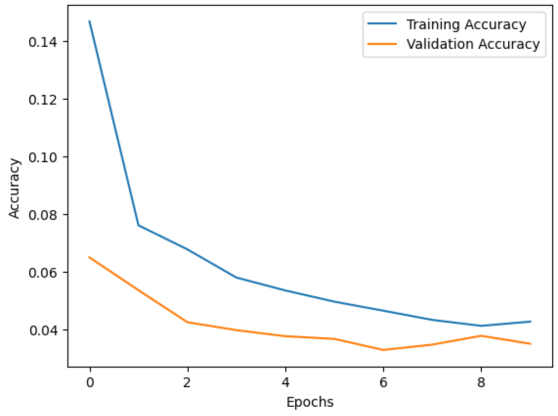
Normalization: Normalizes the image pixel values to the range [0, 1] by dividing by 255.





Augmentation

During the process of making a project we have faced a problem. As we can see in our final results graph the gap between the training and validation is really low which means our model is getting trained properly and the accuracy is good. But initially we could not achieve this accuracy. Our model was inaccurate.



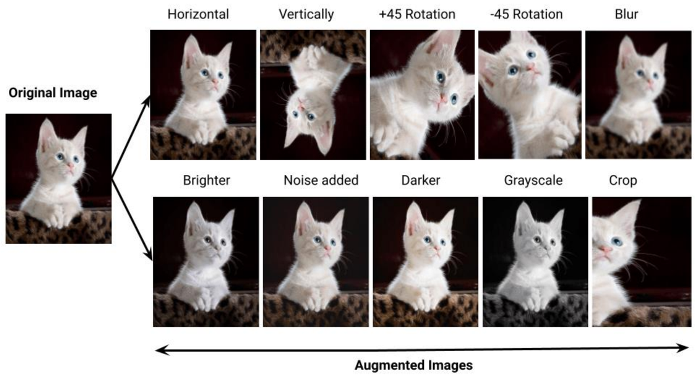
We solved the problem through a method called **Augmentation.**

What is Augmentation?

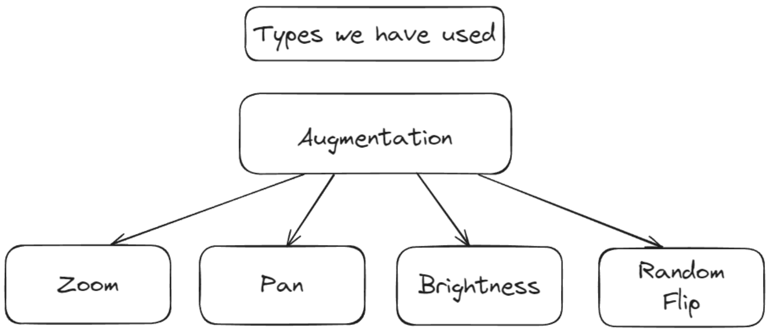
Augmentation in machine learning refers to techniques used to increase the diversity of a training dataset without collecting new data. This is done by making small, random changes to the existing data. For example, in image processing, augmentation might include rotating, flipping, or cropping images to create slightly different versions of the same image. This helps improve the model's ability to generalize and perform better on new, unseen data.

We gathered a lot of data (approximately 4k) but to train a model 4k data are not enough. We needed more data. So, we used augmentation technique which generates more datasets from the existing datasets.

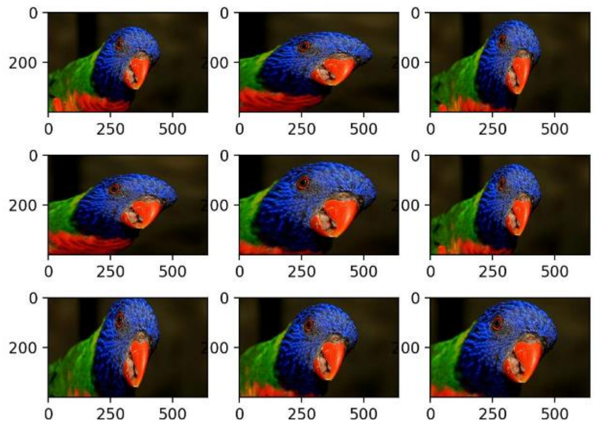
For example



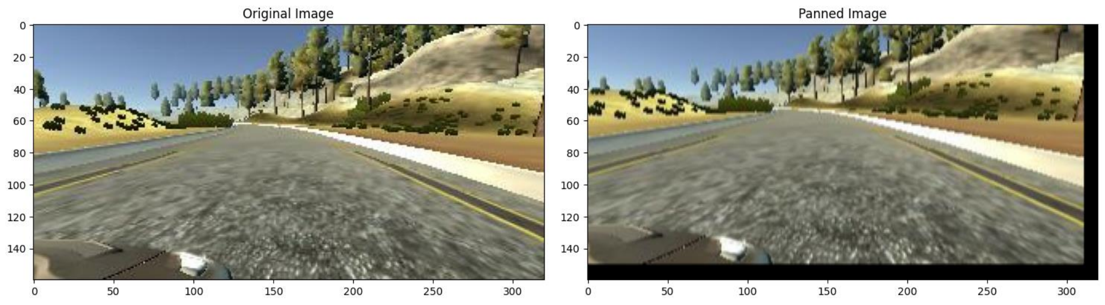
There are some different techniques to use this method. We have used 4 techniques.



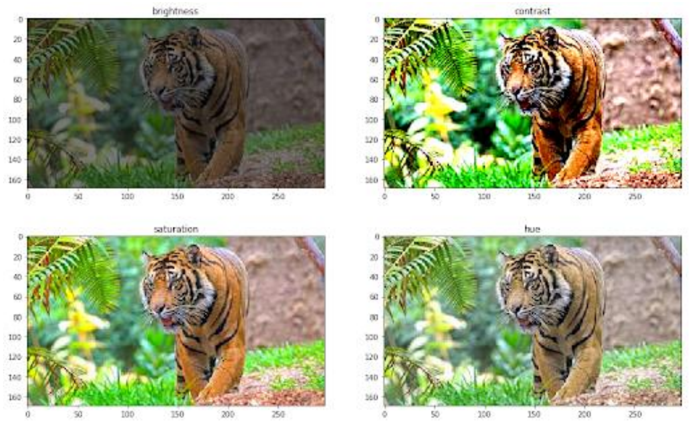
1. **Zoom:** This involves either cropping the image to create a zoomed-in effect or zooming out to include more of the surrounding area. This helps the model learn to recognize objects at different scales.



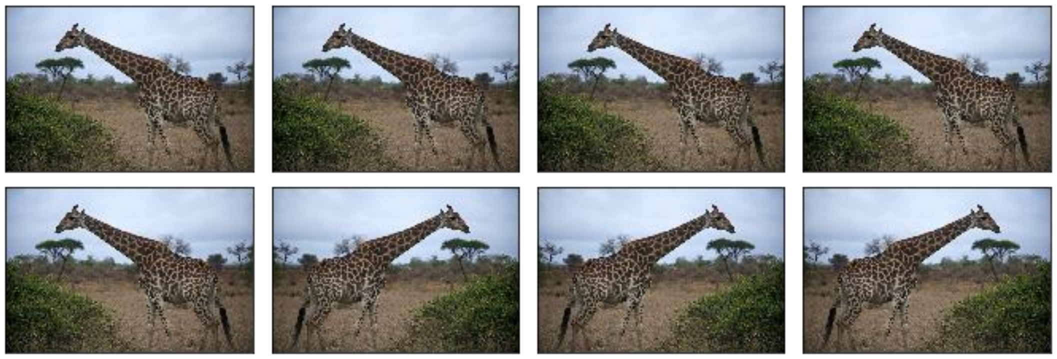
2. **Pan:** This refers to shifting the image horizontally or vertically. It helps the model become invariant to the position of objects in the image, making it better at recognizing objects regardless of where they appear.



3. **Brightness:** This technique involves adjusting the brightness of the image, either making it lighter or darker. It helps the model handle variations in lighting conditions.



4. **Random Flip:** This involves randomly flipping the image horizontally (left to right) or vertically (top to bottom). It helps the model learn to recognize objects in different orientations.



These generated images we only used for the training datasets because for model accuracy if we use these augmented images, our accuracy will not be good. We made sure that only 50% of the augmented images will be used and it is randomized. It will be added only to the training datasets.

.

Why are not using the augmented images in Validation set?

If we use augmented images in validation set, our model will be more generalized which we do not want.

Also, Augmented images are typically not used in the validation set for a few key reasons:

1. **Realistic Performance Assessment:** The validation set is used to evaluate the model's performance on data it hasn't seen during training. Using unaltered images ensures that this evaluation reflects how the model will perform on real-world, unseen data.
2. **Consistency:** Keeping the validation set consistent and unmodified allows for a fair comparison between different models and training runs. Augmenting validation images could introduce variability that makes it harder to gauge true model improvements.

3. **Overfitting Check:** The validation set helps in monitoring for overfitting. If augmented images were included in the validation set, it could obscure the signs of overfitting because the validation data would be more similar to the training data.

4. **Hyperparameter Tuning:** The validation set is often used to tune hyperparameters. Accurate tuning requires a stable and realistic evaluation set, free from artificial alterations.

In summary, the validation set should remain representative of real-world data to provide a reliable measure of the model's performance and generalization capability.

Model Architecture:

The model used is based on the NVIDIA model architecture for self-driving cars. In our model, we have used the Exponential Linear Unit (ELU) as the activation function for the neural network layers. ELU helps by introducing non-linearity, which allows the model to learn complex patterns more effectively. Additionally, its smooth transition to negative values aids in faster convergence and better performance compared to other activation functions like ReLU.

Key factors to use **ELU**:

People use **ELU** (Exponential Linear Unit) over other activation functions for several key reasons:

1. Smooth Transition to Negative Values: Unlike ReLU (Rectified Linear Unit), which can become zero for negative inputs (leading to "dead neurons"), ELU allows for negative values. This smooth transition helps keep the mean activation closer to zero, which can make learning more efficient.

2. Zero-Centered Output: ELU outputs can be both positive and negative, which helps in balancing the data. This zero-centered property can improve the convergence of the neural network during training.

3. Avoiding Vanishing Gradient Problem: Like ReLU, ELU helps mitigate the vanishing gradient problem in deep networks, ensuring that gradients are sufficiently large for neurons to keep learning.

4. Faster and More Accurate Learning: Studies and experiments have shown that models using ELU can converge faster and achieve better performance compared to models using ReLU or other activation functions. The non-zero mean of the gradients can help in speeding up the training process.

Here's a succinct way to explain why ELU is preferred:

People use ELU because it provides a smooth transition to negative values, maintains a zero-centered output, helps avoid the vanishing gradient problem, and often leads to faster and more accurate learning compared to other activation functions like ReLU.

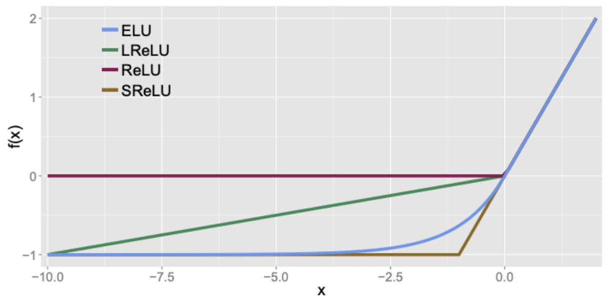
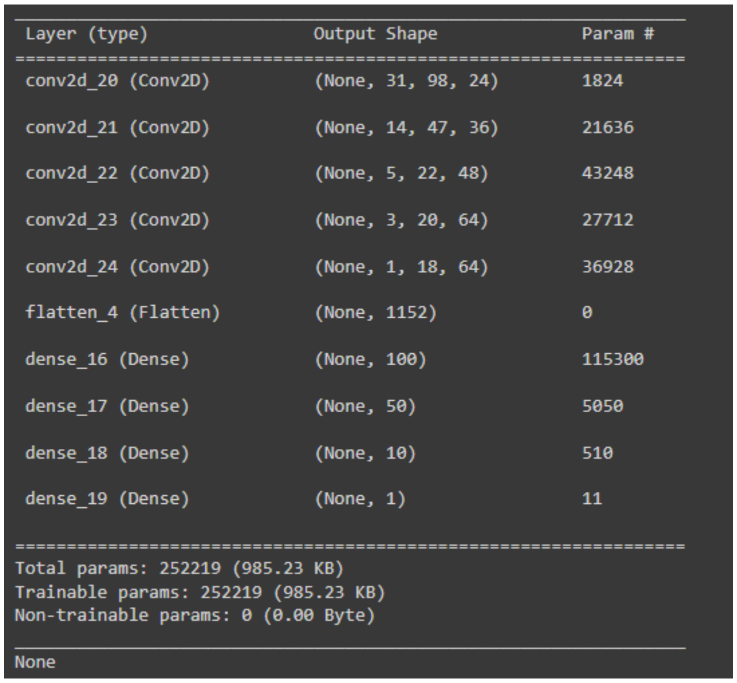


Figure 1: The rectified linear unit (ReLU), the leaky ReLU (LReLU, $\alpha = 0.1$), the shifted ReLUs (SReLUs), and the exponential linear unit (ELU, $\alpha = 1.0$).

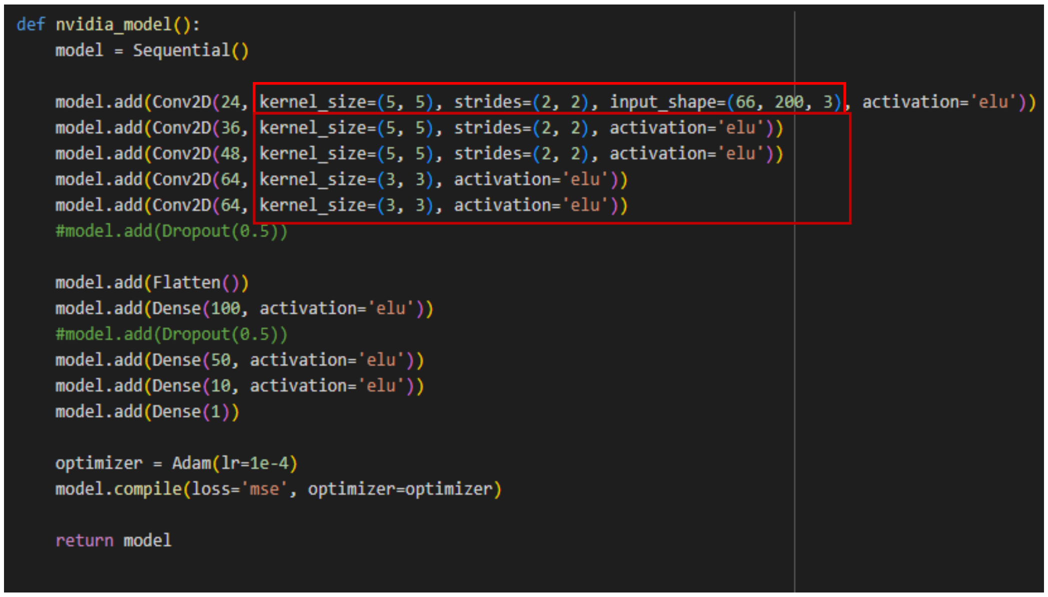
In our model, we have used **dropout** to solve overfitting because our datasets became more generalized means our model became biased towards the training data.

Data Summary:



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_20 (Conv2D)          (None, 31, 98, 24)        1824

 conv2d_21 (Conv2D)          (None, 14, 47, 36)        21636

 conv2d_22 (Conv2D)          (None, 5, 22, 48)         43248

 conv2d_23 (Conv2D)          (None, 3, 20, 64)         27712

 conv2d_24 (Conv2D)          (None, 1, 18, 64)         36928

 flatten_4 (Flatten)         (None, 1152)              0

 dense_16 (Dense)            (None, 100)               115300

 dense_17 (Dense)            (None, 50)                5050

 dense_18 (Dense)            (None, 10)                510

 dense_19 (Dense)            (None, 1)                 11

=================================================================
Total params: 252219 (985.23 KB)
Trainable params: 252219 (985.23 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

The Convolutional Neural Network (CNN) architecture includes five convolutional layers, a flatten layer, and four dense layers. Key layers and their parameters are: conv2d_20 (1,824 params), conv2d_21 (21,636 params), conv2d_22 (43,248 params), conv2d_23 (27,712 params), conv2d_24 (36,928 params), dense_16 (115,300 params), dense_17 (5,050 params), dense_18 (510 params), and dense_19 (11 params). The network has a total of 252,219 trainable parameters (985.23 KB).



```
def nvidia_model():
    model = Sequential()

    model.add(Conv2D(24, kernel_size=(5, 5), strides=(2, 2), input_shape=(66, 200, 3), activation='elu'))
    model.add(Conv2D(36, kernel_size=(5, 5), strides=(2, 2), activation='elu'))
    model.add(Conv2D(48, kernel_size=(5, 5), strides=(2, 2), activation='elu'))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='elu'))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='elu'))
    #model.add(Dropout(0.5))

    model.add(Flatten())
    model.add(Dense(100, activation='elu'))
    #model.add(Dropout(0.5))
    model.add(Dense(50, activation='elu'))
    model.add(Dense(10, activation='elu'))
    model.add(Dense(1))

    optimizer = Adam(lr=1e-4)
    model.compile(loss='mse', optimizer=optimizer)

    return model
```

Since we are using NVIDIA model, these are the suggested values for our model.

Optimization:

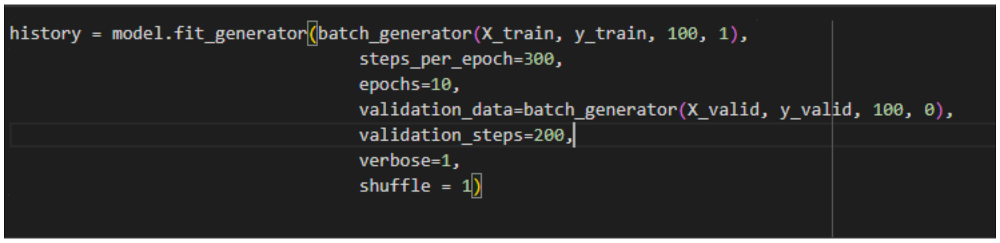
Certainly! Let's break down each component of the model configuration:

1. **Optimizer (Adam):**
    - Adam optimizer is a popular choice for training neural networks. It combines the benefits of two other extensions of stochastic gradient descent: AdaGrad and RMSProp.
    - Adam maintains a separate learning rate for each parameter and adapts those rates based on the first and second moments of the gradients.
    - The learning rate of 0.001 is relatively small, which helps in stabilizing the training process and prevents the model from making large updates to the weights, which might lead to overshooting or instability.
2. **Loss Function (Mean Squared Error - MSE):**
    - Mean Squared Error is a common choice for regression problems, like predicting steering angles in this case.
    - It measures the average squared difference between the predicted and actual values.

- By minimizing the MSE loss, the model aims to reduce the discrepancy between its predictions and the true steering angles.

3. **Training with a Generator Function (batch_generator):**
   - In machine learning, especially when dealing with large datasets that don't fit into memory, generators are used to load and preprocess data on the fly in batches.
   - The **batch_generator** function yields batches of preprocessed and augmented images along with their corresponding steering angles.
   - Preprocessing may involve tasks like resizing images, normalizing pixel values, or any other necessary transformations to prepare the data for training.
   - Augmentation involves applying various transformations to the images (such as rotation, flipping, or shifting) to artificially increase the diversity of the training data. This helps in improving the model's robustness and generalization ability.
   - Using a generator function allows for efficient utilization of computational resources and enables training on datasets that are too large to fit entirely into memory.

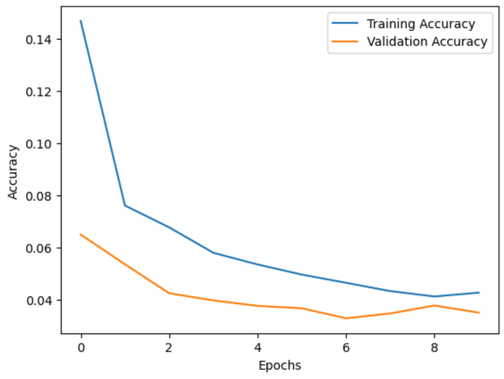
In summary, this configuration sets up the model for efficient training using the Adam optimizer with a small learning rate, minimizing Mean Squared Error loss, and leveraging a generator function to handle data loading, preprocessing, and augmentation during training.

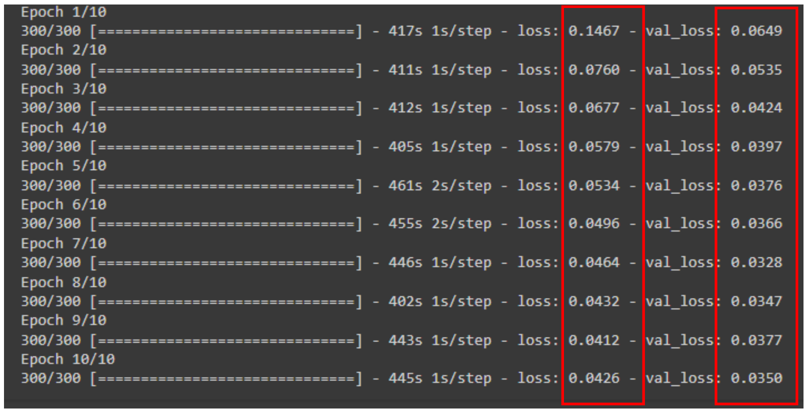


```
history = model.fit_generator(batch_generator(X_train, y_train, 100, 1),
                                  steps_per_epoch=300,
                                  epochs=10,
                                  validation_data=batch_generator(X_valid, y_valid, 100, 0),
                              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

Here, this code trains a neural network using batches of data. It sets how many times the model will see the whole training data (epochs) and how many batches of data it will see each time (steps per epoch). It also sets up validation, where the model checks its performance on separate data. The 'verbose' setting controls how much progress information is shown during training. Additionally, it shuffles the training data to make sure the model doesn't learn any order patterns. This helps the model learn effectively from large datasets and evaluate its performance accurately.

After fitting the model, we get our final training and validation graph:





```
Epoch 1/10
300/300 [==============================] - 417s 1s/step - loss: 0.1467 - val_loss: 0.0649
Epoch 2/10
300/300 [==============================] - 411s 1s/step - loss: 0.0760 - val_loss: 0.0535
Epoch 3/10
300/300 [==============================] - 412s 1s/step - loss: 0.0677 - val_loss: 0.0424
Epoch 4/10
300/300 [==============================] - 405s 1s/step - loss: 0.0579 - val_loss: 0.0397
Epoch 5/10
300/300 [==============================] - 461s 2s/step - loss: 0.0534 - val_loss: 0.0376
Epoch 6/10
300/300 [==============================] - 455s 2s/step - loss: 0.0496 - val_loss: 0.0366
Epoch 7/10
300/300 [==============================] - 446s 1s/step - loss: 0.0464 - val_loss: 0.0328
Epoch 8/10
300/300 [==============================] - 402s 1s/step - loss: 0.0432 - val_loss: 0.0347
Epoch 9/10
300/300 [==============================] - 443s 1s/step - loss: 0.0412 - val_loss: 0.0377
Epoch 10/10
300/300 [==============================] - 445s 1s/step - loss: 0.0426 - val_loss: 0.0350
```

Here the loss value is too less in both training and validation set which is a good sign means our model is getting trained properly. From the above graph we can see the gap between the training and validation is really low which show the model's accuracy is good.

Conclusion:

In this project, we successfully implemented Behavior Cloning using an NVIDIA model to predict steering angles for autonomous driving. By leveraging a Convolutional Neural Network (CNN), we were able to process images captured from a car's perspective and accurately predict the necessary steering angles. The model architecture, inspired by the visual processing in the human brain, included convolutional layers for feature extraction and fully connected layers for prediction.

Key steps in our process included:

1. **Data Preprocessing:** Standardizing the image data and steering angles, removing irrelevant data, and ensuring uniform distribution of steering angles to prevent bias.
2. **Image Preprocessing:** Converting images to YUV color space, resizing, cropping, applying Gaussian blur, and normalizing pixel values.
3. **Data Augmentation:** Enhancing the dataset through techniques such as zooming, panning, adjusting brightness, and random flipping to improve the model's generalization capability.
4. **Model Architecture**: Using a CNN with the ELU activation function and dropout to prevent overfitting.
5. **Optimization and Training:** Employing the Adam optimizer with a small learning rate, minimizing Mean Squared Error (MSE) loss, and using a generator function for efficient data handling during training.

The results demonstrated that our model was trained effectively, as evidenced by the low loss values in both the training and validation sets and the minimal gap between them, indicating high accuracy and robust performance. This approach showcases the potential of CNNs in autonomous driving applications, where accurate and real-time predictions are crucial for safe navigation.