GEORGE BROWN COLLEGE

Applied A.I. Solutions Development (T431)

DEEP LEARNING I
GROUP 8 REPORT
TRAFFIC SIGN RECOGNITION PROJECT

Date : June 7 , 2024

Aditya Koshti - 101504567

Ayushi Patel - 101450798

Mahesh Dadheech - 101501168

Sakshi Sareen - 101471328

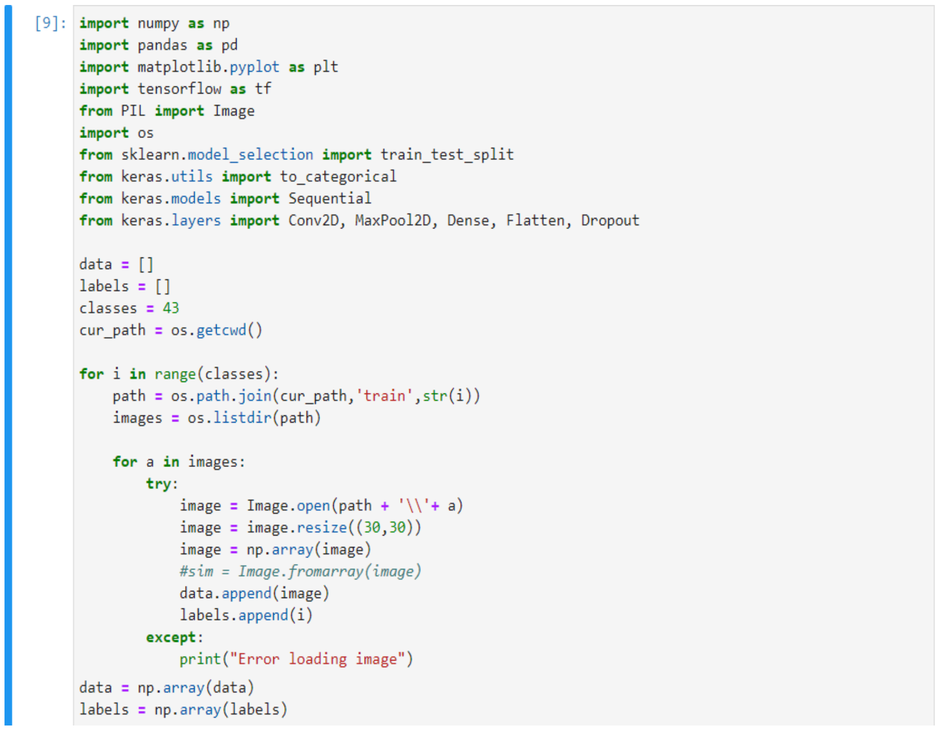
Shivani - 101504534

INDEX

Exploring the Dataset

In our deep learning project on traffic sign recognition, the first step involves understanding and preparing the dataset. We are using a dataset that is organized into a 'train' folder, containing 43 subfolders. Each of these subfolders represents a different traffic sign class, with folder names ranging from 0 to 42. Here's how we handle and preprocess this data. Importing Necessary Libraries To begin with, we need to import several libraries that will help us load, manipulate, and process the images. Here's the code for importing the necessary libraries :

- **NumPy:** For numerical operations and array handling.
- **Pandas:** For data manipulation and analysis.
- **Matplotlib:** For plotting and visualizing data.
- **TensorFlow:** For building and training our deep learning models.
- **PIL (Python Imaging Library):** For opening and manipulating images.
- **OS:** For interacting with the operating system, especially for navigating the file structure.
- **Scikit-learn:** For splitting the data into training and testing sets.



```
[9]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from PIL import Image
import os
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout

data = []
labels = []
classes = 43
cur_path = os.getcwd()

for i in range(classes):
    path = os.path.join(cur_path,'train',str(i))
    images = os.listdir(path)

    for a in images:
        try:
            image = Image.open(path + '\\'+ a)
            image = image.resize((30,30))
            image = np.array(image)
            #sim = Image.fromarray(image)
            data.append(image)
            labels.append(i)
        except:
            print("Error loading image")
data = np.array(data)
labels = np.array(labels)
```

Loading and Preprocessing the Data

Next, we load the images from the dataset and preprocess them so they can be used to train our model. Here is the code that handles this

Detailed Explanation Initialization: We start by creating two empty lists: data for storing the image data and labels for the corresponding labels. We set classes to 43, representing the number of traffic sign categories. cur_path stores the current working directory path.

Iterating Over Classes: Using a for loop, we iterate through each class, ranging from 0 to 42. For each class, we build the path to the respective folder using os.path.join.

Loading Images: We use os.listdir to get a list of all files (images) in the current class folder. Then, we iterate over each image file in the folder. A try-except block is used to handle any potential errors while loading the images.

Processing Images: For each image, we open it using Image.open. We resize the image to 30x30 pixels to standardize the input size using image.resize. We convert the image to a NumPy array using np.array. We append the processed image to the data list and the corresponding label to the labels list.

Handling Errors: If there's an error loading an image, we catch the exception and print an error message, allowing the program to continue processing the remaining images. Converting to NumPy Arrays: Finally, we convert the data and label lists into NumPy arrays for efficient handling and compatibility with our deep learning framework.

Finally, we have stored all the images and their labels into lists (data and labels). We need to convert the list into numpy arrays for feeding to the model. The shape of data is **(39209, 30, 30, 3)** which means that there are 39,209 images of size 30×30 pixels and the last 3 means the data contains colored images (RGB value). With the sklearn package, we use the **train_test_split()** method to split training and testing data. From the keras.utils package, we use the **to_categorical** method to convert the labels present in y_train and t_test into one-hot encoding.

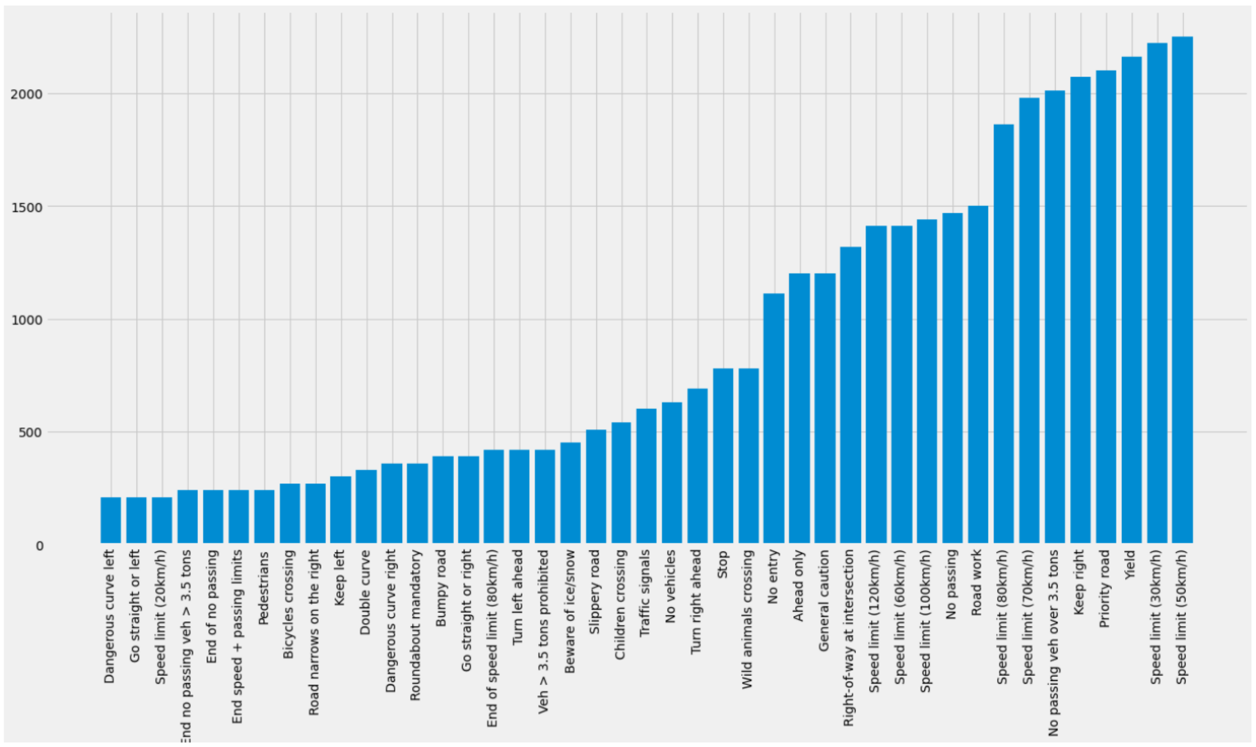
[11]:
```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu', input_shape=X_train.shape[1:]))
model.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))

#Compilation of the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Exploratory Data Analysis

Traffic Sign Class Distribution:

In this section, we conducted an exploratory analysis to understand the distribution of traffic sign classes in our training dataset. The following code snippet demonstrates how we visualized the number of images in each class



Visualization Details

- *Class Distribution:* The bar plot displays the number of images in each traffic sign class. Each bar represents a different class, and the height of the bar indicates the number of images in that class.
- *Sorting:* To visualize the data more clearly, we sorted the classes based on the number of images. This allows us to identify any class imbalances and

understand which classes have the most and least representation in the dataset.

- *X-axis Labels:* The class names are labeled on the x-axis. To prevent overcrowding, we rotated the labels vertically for better readability.

Observations

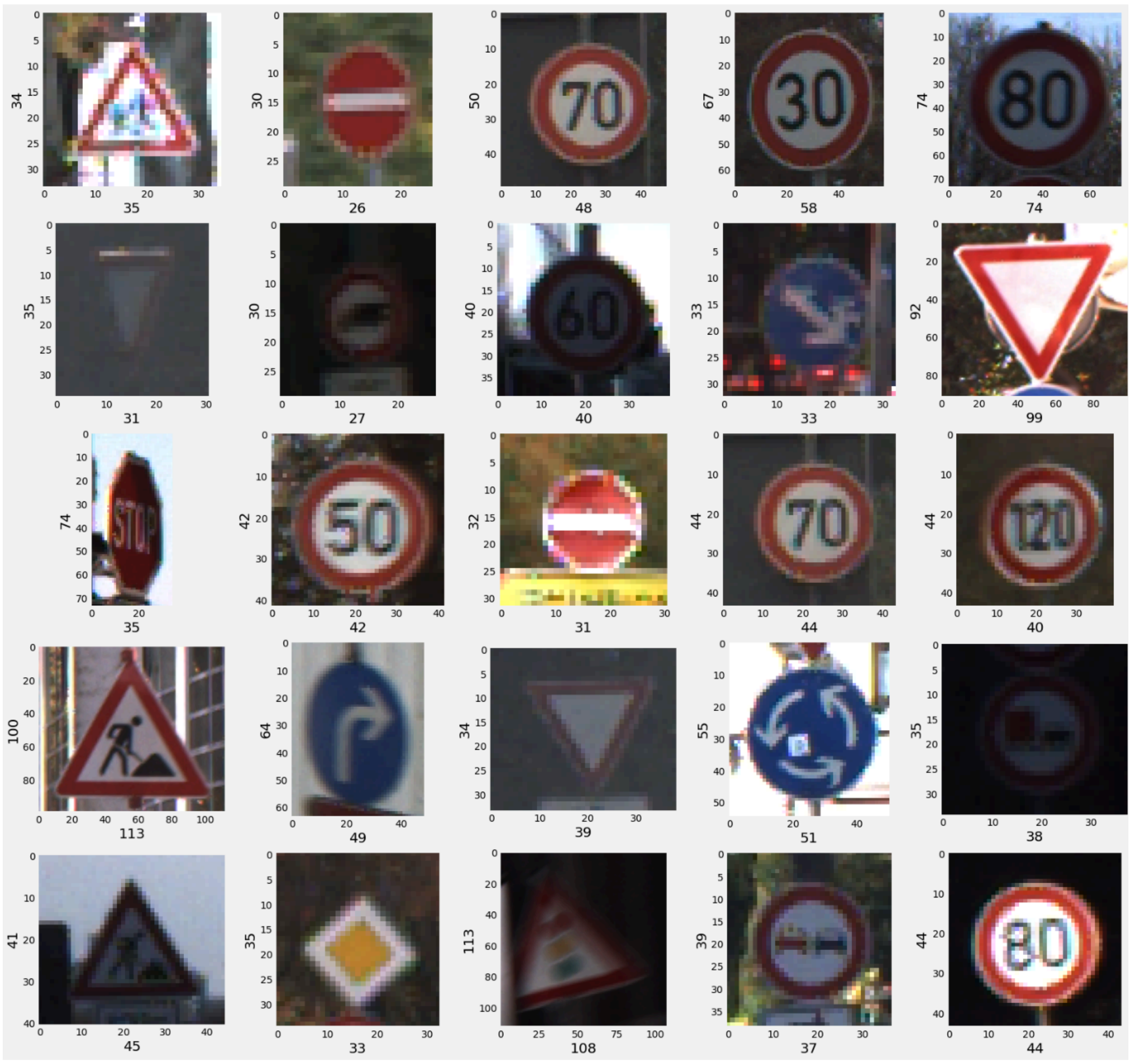
Through this visualization, we gained insights into the distribution of traffic sign classes in our training dataset:

1. **Class Imbalance:** The plot revealed that there is a significant class imbalance, where some classes have a much larger number of images compared to others. This imbalance could potentially affect the model's performance, as it may bias the model towards classes with more samples.

2. **Identifying Underrepresented Classes:** By examining the plot, we could identify classes with fewer images, such as "Dangerous curve left" and "Go straight or left." Addressing the imbalance in these classes may be necessary to ensure that the model learns to recognize all classes effectively.

Visualizing 25 random images from the Test data:

In our exploratory data analysis (EDA), we aimed to get a better understanding of the traffic sign images in our dataset. One useful approach was visualizing a random selection of images from the test set. This helped us to observe the variety and quality of the images, as well as the different classes of traffic signs.



Visualization Details

- *Figure Size:* The visualization is set to a large figure size of 25 x 25 inches to ensure that each image is displayed clearly.
- *Subplots:* We created a grid of 5x5 subplots to display 25 random images from the test dataset.
- Random Image Selection:For each subplot, a random image path is selected from the test data, and the image is read and displayed.
- Image Display: Each image is displayed without a grid to keep the focus on the image content. The width and height of each image are labeled on the

x-axis and y-axis, respectively, to provide additional context about the image dimensions.

Observations

Through this visualization, we observed the following:

1. **Class Diversity:** The random selection of images gave us a snapshot of the diversity of traffic sign classes in the test set. This helped confirm that our test data includes a wide range of traffic signs.

2. **Image Quality**: By visualizing the images, we could assess the quality and resolution of the images in the test dataset. This is important for understanding how well the model might perform with real-world data.

3. **Class Imbalance:** Even with random sampling, we could get a sense of class imbalance by noting which traffic signs appeared more frequently. This reinforced our earlier observation that some classes, such as "Speed limit (50km/h)" and "Speed limit (30km/h)," had many more samples than others like "Dangerous curve left" and "Go straight or left."

MODEL 1 - CNN

Introduction

Convolutional Neural Networks (CNNs) are powerful deep learning models designed specifically for image recognition tasks. They offer several advantages that make them an excellent choice for traffic sign recognition in real-time applications, particularly in scenarios where accuracy and efficiency are crucial, such as in autonomous vehicles.

CNNs automatically learn to detect features in images, making them highly effective for tasks like traffic sign classification. This capability is essential for deployment in autonomous vehicles or mobile applications where quick and accurate processing is essential for safety and user experience.

CNN Architecture and Its Adaptation for Traffic Sign Recognition

Fundamentals of CNNs:

CNNs use convolutional layers to extract features from images, pooling layers to reduce dimensionality, and fully connected layers to make predictions. This architecture helps preserve spatial hierarchies in images and ensures robust feature extraction while maintaining computational efficiency.

Adaptations for Traffic Sign Recognition:

For the specific needs of traffic sign recognition, we adapted a standard CNN model. This adaptation involved modifications to the network's architecture to accommodate the unique characteristics of traffic sign images:

- Base Model: We built a custom CNN model using several convolutional, pooling, and dense layers to extract features and classify traffic signs.
- Custom Classification Layers: Dense and dropout layers were added to fine-tune the model for traffic sign recognition. Specifically, a Dense layer with 256 units and ReLU activation, followed by a Dropout layer with a rate of 0.5, and finally a Dense layer with 43 units and softmax activation were added.

- Pooling and Dropout: Pooling layers were used to reduce the spatial dimensions, and dropout layers were included to prevent overfitting.

Model Training and Initial Results

Training Configuration:

The training was configured with a focus on achieving the best trade-off between accuracy and computational efficiency:

- Batch Size and Epochs: A batch size of 32 and 15 epochs were chosen based on preliminary experiments that indicated these parameters offered the best convergence speed without overfitting.
- Validation Split: 20% of the dataset was reserved for validation purposes to monitor and tune the model's performance independently from the training data.

Initial Model Performance:

Before hyperparameter tuning, the model achieved a validation accuracy of 94.41%. Although this was a promising start, further improvements were necessary to enhance model performance.

CNN - Hyperparameter Tuning with Optuna

Tuning Strategy:

Optuna was utilized for its efficient and flexible framework for hyperparameter optimization. The tuning focused on:

- Dense Layer Units: Finding the optimal number of neurons to process features effectively, with choices between 256 and 512 units.
- Dropout Rates: Determining the best rates to minimize overfitting while retaining model complexity, with rates optimized between 0.3 and 0.7**.**

Results of Tuning:

Post-tuning, the model markedly improved training efficiency and generalization, achieving a validation accuracy of 97%. This enhancement was visually supported by plots showing consistent gains in performance across the tuning trials.

Analysis and Evaluation

Performance Analysis:

The tuned model's performance was extensively analyzed, revealing significant gains in accuracy and better handling of overfitting. The hyperparameter adjustments positively affected the model's ability to generalize to new data.

Discussion of Findings:

The hyperparameter adjustments were critical in balancing model complexity and overfitting. The findings underscore the importance of a tailored approach to model training in the field of traffic sign recognition.

Conclusion and Future Directions

This project illustrates the potential of advanced deep learning models, like CNNs, to transform traffic sign recognition systems. The successful application and tuning of a CNN for this purpose opens avenues for future research, including real-time processing and integration into autonomous vehicle systems. Further exploration could also include the implementation of newer and more complex CNN variants or other advanced deep learning architectures.

MODEL 2 - MobileNetV2(Pre-trained)

Introduction

- MobileNetV2 is a state-of-the-art convolutional neural network architecture that is specifically designed for efficient and fast image recognition tasks. It offers several advantages that make it an excellent choice for traffic sign recognition in real-time applications, particularly in scenarios where computational resources are limited, such as in autonomous vehicles or mobile devices.
- MobileNetV2 is specifically designed to run on mobile and edge devices with limited computational power. This is particularly important for deployment in autonomous vehicles or mobile applications where quick and efficient processing is essential for safety and user experience.
- MobileNetV2 has been extensively used and validated in various real-world applications, demonstrating its robustness and reliability. Its adoption in our project ensures that we are building on a foundation of well-tested technology.

MobileNetV2 Architecture and Its Adaptation for Traffic Sign Recognition

Fundamentals of MobileNetV2:

The MobileNetV2 architecture introduces an efficient and lightweight approach to deep neural networks through the use of depthwise separable convolutions and inverted residual connections with linear bottlenecks. These innovations help in preserving model accuracy while significantly reducing the computational complexity and model size. This design is particularly well-suited for mobile and embedded vision applications, where computational resources are limited

Adaptations of MobileNetV2:

For the specific needs of traffic sign recognition, we adapted the standard MobileNetV2 model. This adaptation involved modifications to the network's architecture to accommodate the unique characteristics of traffic sign images:

- *Base Model Freezing:* The pre-trained MobileNetV2 base model, loaded with ImageNet weights, was used without the top (classification) layer. Freezing these layers means that their weights will not be updated during training, leveraging the robust feature extraction capabilities of MobileNetV2 while focusing on training the new layers added on top.

- *Custom Classification Layers:* Custom dense and dropout layers were added on top of the base model to fine-tune it for the specific task of traffic sign recognition. Specifically, a Dense layer with 512 units and ReLU activation, followed by a Dropout layer with a rate of 0.5, a Dense layer with 256 units and ReLU activation, another Dropout layer with a rate of 0.5, and finally a Dense layer with 43 units and softmax activation were added.

- *Global Average Pooling:* This layer was utilized to reduce the dimensionality of the feature maps, thus decreasing the computational load and helping to prevent overfitting.

- *Output Layer Configuration*: The final layer was customized to classify 43 distinct traffic sign categories, employing a softmax activation function to output probabilities across the classes.

Model Training and Initial Results

Training Configuration:

The training was configured with a focus on achieving the best trade-off between accuracy and computational efficiency:

- *Batch Size and Epochs*: A batch size of 32 and 10 epochs were chosen based on preliminary experiments that indicated these parameters offered the best convergence speed without overfitting.
- *Validation Split:* 20% of the dataset was reserved for validation purposes to monitor and tune the model’s performance independently from the training data.

Initial Model Performance

Before hyperparameter tuning, the model achieved a validation accuracy of 50%. Although this was a promising start, further improvements were necessary to enhance model performance.

MobileNetV2 - Hyperparameter Tuning with Optuna

Tuning Strategy:

Optuna was utilized for its efficient and flexible framework for hyperparameter optimization. The tuning focused on:

- *Dense Layer Units:* Finding the optimal number of neurons to process features effectively, with choices between 256 and 512 units.
- *Dropout Rates:* Determining the best rates to minimize overfitting while retaining model complexity, with rates optimized between 0.3 and 0.7.

Results of Tuning

Post-tuning, the model markedly improved training efficiency and generalization, achieving a validation accuracy of 87.26%. This enhancement was visually supported by plots showing consistent gains in performance across the tuning trials.

Analysis and Evaluation

Performance Analysis:

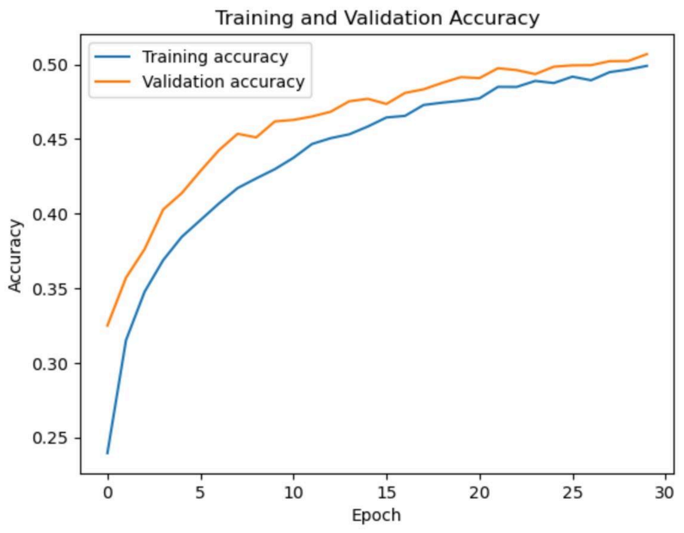
The tuned model's performance was extensively analyzed, revealing significant gains in accuracy and a better handling of overfitting. The hyperparameter adjustments were shown to positively affect the model's ability to generalize to new data.

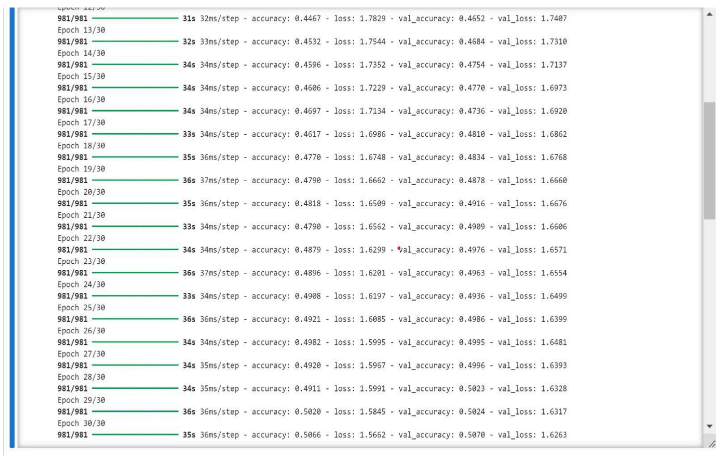
Discussion of Findings:

The hyperparameter adjustments were critical in balancing model complexity and overfitting. The findings underscore the importance of a tailored approach to model training in the field of traffic sign recognition.

__Conclusion and Future Directions:__

This project illustrates the potential of advanced deep learning models, like MobileNetV2, to transform traffic sign recognition systems. The successful application and tuning of MobileNetV2 for this purpose opens avenues for future research, including real-time processing and integration into autonomous vehicle systems. Further exploration could also include the implementation of newer and more complex MobileNet variants or other advanced deep-learning architectures.





```
981/981 ──────────────── 31s 32ms/step - accuracy: 0.4467 - loss: 1.7829 - val_accuracy: 0.4652 - val_loss: 1.7407
Epoch 13/30
981/981 ──────────────── 32s 33ms/step - accuracy: 0.4532 - loss: 1.7544 - val_accuracy: 0.4684 - val_loss: 1.7310
Epoch 14/30
981/981 ──────────────── 34s 34ms/step - accuracy: 0.4596 - loss: 1.7352 - val_accuracy: 0.4754 - val_loss: 1.7137
Epoch 15/30
981/981 ──────────────── 34s 34ms/step - accuracy: 0.4606 - loss: 1.7229 - val_accuracy: 0.4770 - val_loss: 1.6973
Epoch 16/30
981/981 ──────────────── 34s 34ms/step - accuracy: 0.4697 - loss: 1.7134 - val_accuracy: 0.4736 - val_loss: 1.6920
Epoch 17/30
981/981 ──────────────── 33s 34ms/step - accuracy: 0.4617 - loss: 1.6986 - val_accuracy: 0.4810 - val_loss: 1.6862
Epoch 18/30
981/981 ──────────────── 35s 36ms/step - accuracy: 0.4770 - loss: 1.6748 - val_accuracy: 0.4834 - val_loss: 1.6768
Epoch 19/30
981/981 ──────────────── 36s 37ms/step - accuracy: 0.4790 - loss: 1.6662 - val_accuracy: 0.4878 - val_loss: 1.6660
Epoch 20/30
981/981 ──────────────── 35s 36ms/step - accuracy: 0.4818 - loss: 1.6509 - val_accuracy: 0.4916 - val_loss: 1.6676
Epoch 21/30
981/981 ──────────────── 33s 34ms/step - accuracy: 0.4790 - loss: 1.6562 - val_accuracy: 0.4909 - val_loss: 1.6606
Epoch 22/30
981/981 ──────────────── 34s 34ms/step - accuracy: 0.4879 - loss: 1.6299 - val_accuracy: 0.4976 - val_loss: 1.6571
Epoch 23/30
981/981 ──────────────── 36s 37ms/step - accuracy: 0.4896 - loss: 1.6201 - val_accuracy: 0.4963 - val_loss: 1.6554
Epoch 24/30
981/981 ──────────────── 33s 34ms/step - accuracy: 0.4908 - loss: 1.6197 - val_accuracy: 0.4936 - val_loss: 1.6499
Epoch 25/30
981/981 ──────────────── 36s 36ms/step - accuracy: 0.4921 - loss: 1.6085 - val_accuracy: 0.4986 - val_loss: 1.6399
Epoch 26/30
981/981 ──────────────── 34s 34ms/step - accuracy: 0.4982 - loss: 1.5995 - val_accuracy: 0.4995 - val_loss: 1.6481
Epoch 27/30
981/981 ──────────────── 34s 35ms/step - accuracy: 0.4920 - loss: 1.5967 - val_accuracy: 0.4996 - val_loss: 1.6393
Epoch 28/30
981/981 ──────────────── 34s 35ms/step - accuracy: 0.4911 - loss: 1.5991 - val_accuracy: 0.5023 - val_loss: 1.6328
Epoch 29/30
981/981 ──────────────── 36s 36ms/step - accuracy: 0.5020 - loss: 1.5845 - val_accuracy: 0.5024 - val_loss: 1.6317
Epoch 30/30
981/981 ──────────────── 35s 36ms/step - accuracy: 0.5066 - loss: 1.5662 - val_accuracy: 0.5070 - val_loss: 1.6263
```

MODEL 3 - RESNET

Introduction

- For our project, we selected ResNet-50 due to its excellent balance between complexity and performance. Given the task of traffic sign recognition, ResNet-50 stands out for its ability to distinguish subtle differences between various sign categories effectively.
- This model's choice was influenced by its demonstrated capabilities in handling complex image recognition tasks, making it an optimal solution for the challenges presented by traffic sign recognition.
- Our approach leverages the deep residual learning framework of ResNet to enhance accuracy and ensure robust recognition capabilities across diverse and challenging environmental conditions.

ResNet Architecture and Its Adaptation for Traffic Sign Recognition

Fundamentals of ResNet:

The ResNet architecture introduces a novel approach to training deep neural networks through the use of residual connections. These connections help preserve the gradient flow through the network, allowing for the successful training of networks that are much deeper than those previously possible. This design mitigates the vanishing gradient problem commonly observed in deep networks without residual connections.

Adaptation Details:

For the specific needs of traffic sign recognition, we adapted the standard ResNet-50 model. This adaptation involved modifications to the network's architecture to accommodate the unique characteristics of traffic sign images:

- *Input Layer Adjustments*: Given the varying sizes and resolutions of traffic sign images in the dataset, the input layer was adjusted to normalize and resize incoming images to a uniform dimension.

- *Convolutional Layers*: These layers were finely tuned to capture the intricate details and features of the traffic signs, essential for accurate classification.

- *Global Average Pooling*: This layer was utilized to reduce the dimensionality of the feature maps, thus decreasing the computational load and helping to prevent overfitting.

- *Output Layer Configuration*: The final layer was customized to classify 43 distinct traffic sign categories, employing a softmax activation function to output probabilities across the classes.

Dataset Preparation and Preprocessing

Dataset Description:

- The TSRB dataset, integral to our training and validation process, includes a wide array of traffic signs, captured under various conditions and angles, making it ideal for robust training.
- This dataset features over 50,000 images distributed across more than 40 classes, providing a comprehensive range of traffic signs.

Preprocessing Techniques:

To optimize the dataset for effective model training, several preprocessing steps were implemented:

- *Resizing and Normalization*: Images were resized to a standard dimension and normalized to ensure uniformity in input data and to aid the convolutional network in learning more effectively.

- *Data Augmentation*: Techniques such as rotation, zoom, and horizontal flipping were employed to artificially expand the training dataset, which helps the model generalize better to new, unseen data.

Model Training and Initial Results

Training Configuration:

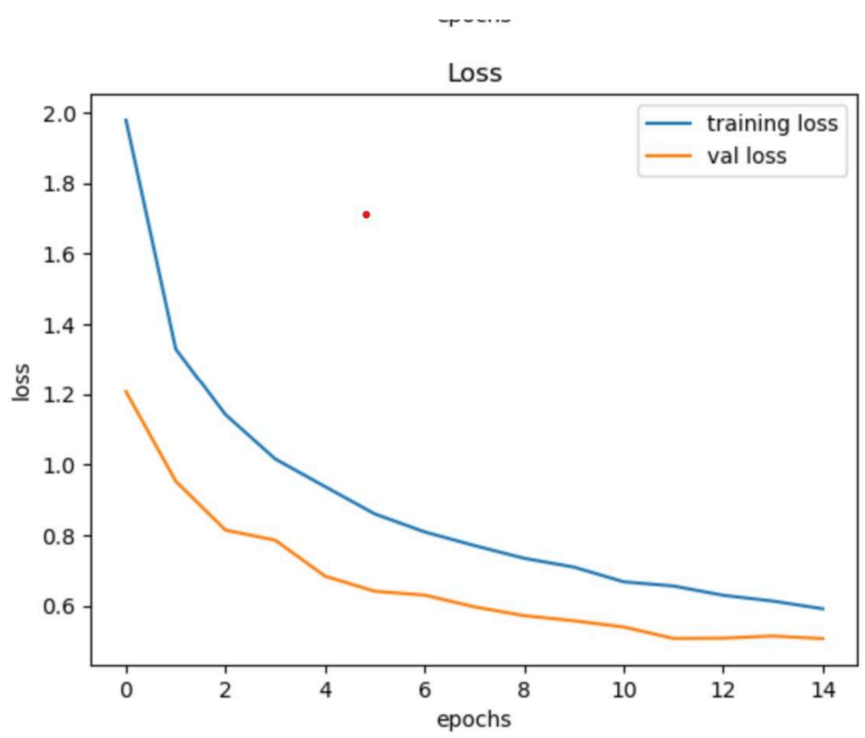
The training was configured with a focus on achieving the best trade-off between accuracy and computational efficiency:

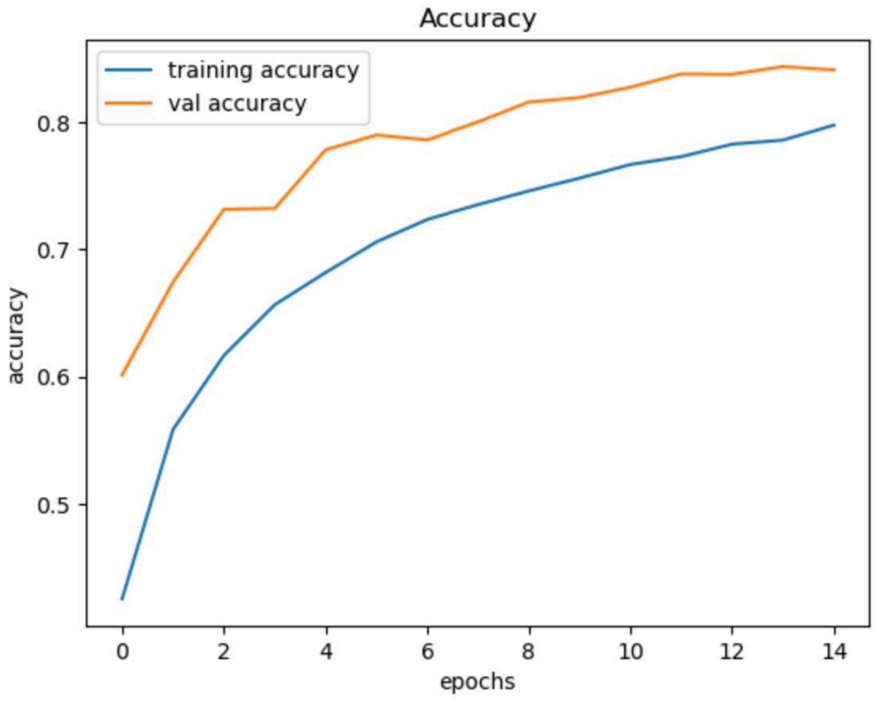
- *Batch Size and Epochs*: A batch size of 32 and 15 epochs were chosen based on preliminary experiments that indicated these parameters offered the best convergence speed without overfitting.

- *Validation Split:* 20% of the dataset was reserved for validation purposes to monitor and tune the model's performance independently from the training data.

Initial Model Performance

Before hyperparameter tuning, the model achieved a validation accuracy of 84.15%. Although promising, the training and validation loss trends indicated potential overfitting, prompting further tuning.





RESNET - Hyperparameter Tuning with Optuna

Tuning Strategy:

Optuna was utilized for its efficient and flexible framework for hyperparameter optimization. The tuning focused on:

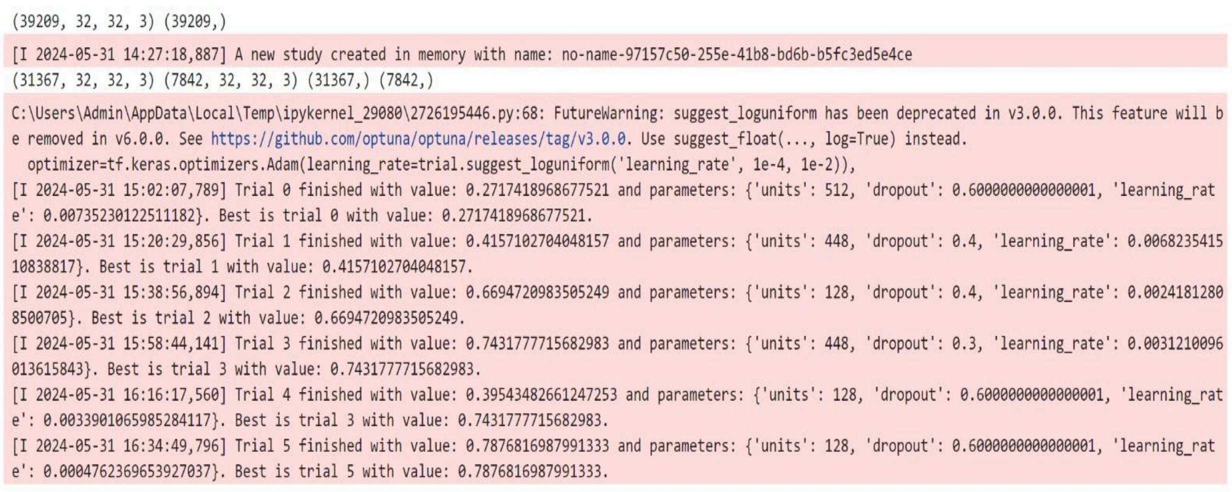
- *Dense Layer Units*: Finding the optimal number of neurons to process features effectively.

- *Dropout Rates*: Determining the best rates to minimize overfitting while retaining model complexity.

- *Learning Rates*: Adjusting the learning rate to optimize the training speed and model convergence.

Results of Tuning

Post-tuning, the model markedly improved training efficiency and generalization, achieving a validation accuracy of 78.76%. This enhancement was visually supported by plots showing consistent gains in performance across the tuning trials.



```
(39209, 32, 32, 3) (39209,)
[I 2024-05-31 14:27:18,887] A new study created in memory with name: no-name-97157c50-255e-41b8-bd6b-b5fc3ed5e4ce
(31367, 32, 32, 3) (7842, 32, 32, 3) (31367,) (7842,)
C:\Users\Admin\AppData\Local\Temp\ipykernel_29080\2726195446.py:68: FutureWarning: suggest_loguniform has been deprecated in v3.0.0. This feature will b
e removed in v6.0.0. See https://github.com/optuna/optuna/releases/tag/v3.0.0. Use suggest_float(..., log=True) instead.
  optimizer=tf.keras.optimizers.Adam(learning_rate=trial.suggest_loguniform('learning_rate', 1e-4, 1e-2)),
[I 2024-05-31 15:02:07,789] Trial 0 finished with value: 0.2717418968677521 and parameters: {'units': 512, 'dropout': 0.6000000000000001, 'learning_rat
e': 0.00735230122511182}. Best is trial 0 with value: 0.2717418968677521.
[I 2024-05-31 15:20:29,856] Trial 1 finished with value: 0.4157102704048157 and parameters: {'units': 448, 'dropout': 0.4, 'learning_rate': 0.0068235415
10838817}. Best is trial 1 with value: 0.4157102704048157.
[I 2024-05-31 15:38:56,894] Trial 2 finished with value: 0.6694720983505249 and parameters: {'units': 128, 'dropout': 0.4, 'learning_rate': 0.0024181280
8500705}. Best is trial 2 with value: 0.6694720983505249.
[I 2024-05-31 15:58:44,141] Trial 3 finished with value: 0.7431777715682983 and parameters: {'units': 448, 'dropout': 0.3, 'learning_rate': 0.0031210096
013615843}. Best is trial 3 with value: 0.7431777715682983.
[I 2024-05-31 16:16:17,560] Trial 4 finished with value: 0.39543482661247253 and parameters: {'units': 128, 'dropout': 0.6000000000000001, 'learning_rat
e': 0.0033901065985284117}. Best is trial 3 with value: 0.7431777715682983.
[I 2024-05-31 16:34:49,796] Trial 5 finished with value: 0.7876816987991333 and parameters: {'units': 128, 'dropout': 0.6000000000000001, 'learning_rat
e': 0.0004762369653927037}. Best is trial 5 with value: 0.7876816987991333.
```

Analysis and Evaluation

Performance Analysis:

The tuned model's performance was extensively analyzed, revealing significant gains in accuracy and a better handling of overfitting. The hyperparameter adjustments were shown to positively affect the model's ability to generalize to new data.

Discussion of Findings:

The hyperparameter adjustments were critical in balancing model complexity and overfitting. The findings underscore the importance of a tailored approach to model training in the field of traffic sign recognition.

Conclusion and Future Directions:

This project illustrates the potential of advanced deep learning models, like ResNet, to transform traffic sign recognition systems. The successful application and tuning of ResNet-50 for this purpose opens avenues for future research, including real-time processing and integration into autonomous vehicle systems. Further exploration could also include the implementation of newer and more complex ResNet variants or other advanced deep-learning architectures.

Challenges:

Developing a reliable traffic sign recognition system comes with its fair share of challenges. These obstacles can significantly affect how well the model performs, and overcoming them requires thoughtful strategies. Here's a closer look at some of the main challenges we encountered during the project:

1. Class Imbalance
One of the first hurdles we faced was class imbalance in our dataset. Imagine trying to learn about different types of birds, but most of your examples are of

sparrows, with only a few instances of eagles and parrots. Similarly, our dataset had plenty of certain traffic signs but very few of others. This imbalance can skew the model, making it great at recognizing common signs but struggling with rare ones. To tackle this, we had to get creative with techniques like generating synthetic data and tweaking the model to pay more attention to the less common signs.

2. Variability in Lighting and Weather Conditions
Another big challenge was dealing with images taken under various lighting and weather conditions. Think about how different a stop sign looks in bright sunlight versus a foggy morning. These differences can trip up our model, making it less reliable. We worked on this by augmenting our training data with images simulating different conditions and teaching the model to be more adaptable. This way, whether it's sunny, rainy, or foggy, the model learns to recognize traffic signs accurately.

3. Similarity between Classes
Traffic signs often look quite similar to each other. For instance, speed limit signs for 30 km/h and 50 km/h can look nearly identical except for the number. This similarity can easily confuse the model, leading to mistakes. To improve its accuracy, we focused on enhancing the model's ability to pick up subtle differences and using more advanced training methods. This helped the model distinguish between signs that look almost the same at first glance.

4. Overfitting
Lastly, we had to deal with the issue of overfitting. This happens when the model becomes too good at recognizing the training data, including its quirks and noise, but then struggles with new, unseen data. It's like memorizing answers for a test rather than understanding the material. We used several strategies to prevent this, such as adding regularization techniques, using dropout layers to make the model less sensitive to specific details, and validating our model rigorously to ensure it generalizes well. Additionally, having a diverse training dataset helped the model learn more broadly applicable features.

Solutions:

During the development of our traffic sign recognition system, we encountered several challenges that required thoughtful approaches and practical solutions. Here’s an in-depth look at how we addressed these issues:

1. Class Imbalance

Initially, we were concerned about class imbalance in our dataset. Class imbalance occurs when some traffic sign classes are underrepresented compared to others, potentially leading to a biased model. However, upon closer inspection, we found that our dataset was not highly imbalanced. The distribution of traffic sign classes was sufficiently even, and our model's performance across different classes was quite good. As a result, we decided not to implement any specific techniques to address class imbalance. This decision saved us time and computational resources, allowing us to focus on other aspects of model development.

2. Variability in Lighting and Weather Conditions

Traffic signs can appear differently depending on the lighting and weather conditions when the photos are taken. This variability can affect the model’s ability to accurately recognize signs. To mitigate this, we resize all images to a fixed size of either 30x30 or 32x32 pixels, depending on the requirements of the specific model we were using. This resizing step helped to standardize the input images, reducing the impact of varying lighting conditions. By ensuring that all images were uniform in size, we made it easier for the model to learn relevant features without being distracted by differences in image dimensions or quality.

3. Similarity between Classes

Some traffic signs look very similar to each other, which can make it challenging for the model to distinguish between them. For example, speed limit signs for 30 km/h and 50 km/h are almost identical except for the numbers. To address this, we relied on the inherent strengths of our Convolutional Neural Network (CNN) architecture. CNNs are particularly good at extracting features from images through their convolutional and pooling layers. These layers help the model learn the subtle differences between similar classes. Thanks to the effectiveness of the

CNN in feature extraction, we did not need to implement additional techniques to specifically address the similarity between classes.

4. Overfitting

Overfitting is a common issue in machine learning where the model performs well on training data but poorly on unseen data. To prevent overfitting, we employed several strategies:

- *Dropout Layers:* We included dropout layers in our model architecture. Dropout works by randomly setting a fraction of the output units to zero during training. This prevents the model from becoming too reliant on any single feature and encourages it to learn more robust patterns. Dropout is a powerful regularization technique that helped improve our model's generalization ability.

- *Fixed Number of Epochs:* Due to computational limitations, we trained our model for a fixed number of epochs rather than using early stopping, which would have allowed the training process to stop once performance on a validation set stopped improving. While early stopping is often beneficial, our fixed training schedule was a practical compromise given our resources.

- *Hyperparameter Tuning with Optuna:* To further combat overfitting and optimize our model's performance, we used Optuna, a hyperparameter optimization framework. Optuna helped us find the best combination of hyperparameters, such as learning rate, batch size, and the number of layers, which enhanced the model's performance while mitigating overfitting.

Traffic Sign Classification GUI:

Purpose

One of the other objectives of this project was to develop a graphical user interface (GUI) that allows users to upload images of traffic signs and classify them using a pre-trained neural network model. This tool aims to simplify the process of traffic sign recognition, making it accessible and user-friendly.

Tools Used

To achieve our goal, we utilized a variety of tools and libraries:

- *Tkinter:* Used for creating the GUI, providing a simple and efficient way to design the user interface.
- *PIL (Pillow):* Employed for handling image operations, such as opening, resizing, and displaying images within the GUI.
- *NumPy:* Utilized for numerical operations on image data, crucial for preparing images for model input.
- *Keras (TensorFlow):* Used for loading and using the pre-trained neural network model to classify traffic signs.

Key Features

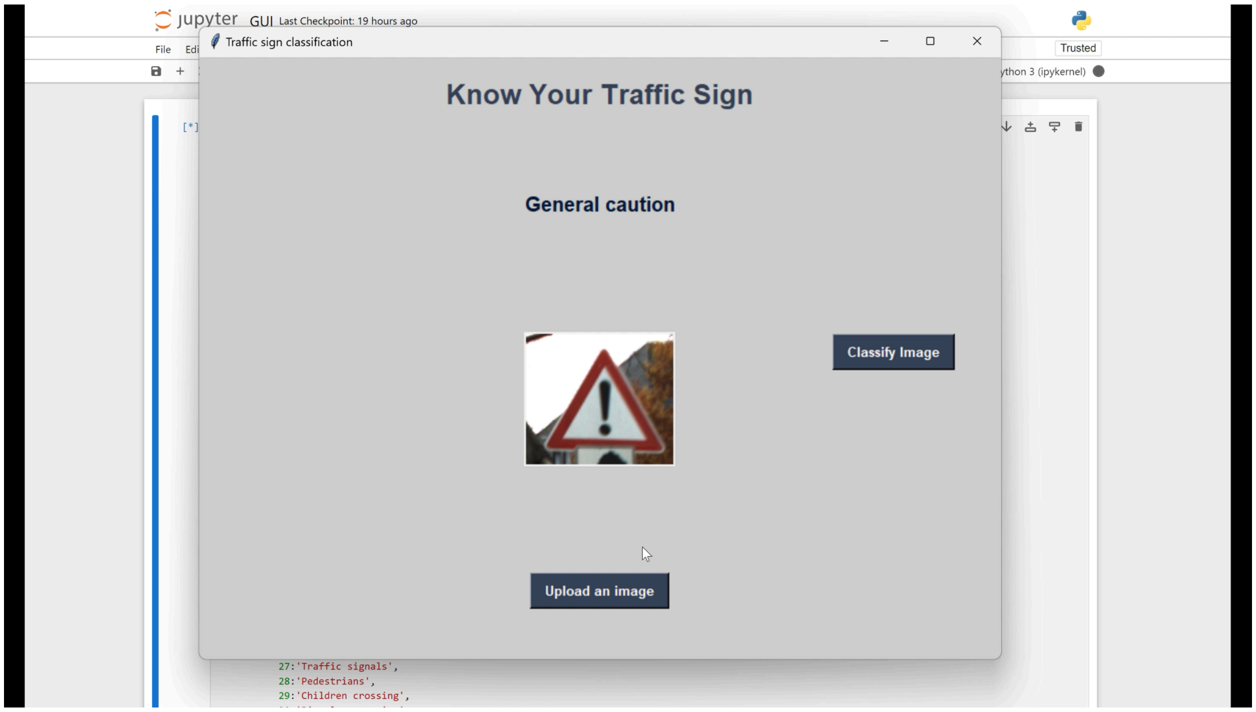
The GUI includes several essential features to ensure functionality and ease of use:

- *Upload Button:* This button allows users to upload an image of a traffic sign from their local system.
- *Classify Button:* Once an image is uploaded, this button classifies the image using the pre-trained model.
- *Result Display:* After classification, the GUI displays the label of the classified traffic sign, informing the user about the detected sign.

Explanation of Key Components

- *GUI Initialization:* The GUI is initialized using Tkinter, setting the window size, title, and background color. Labels and buttons are created for user interaction.
- *Model Loading:* The pre-trained traffic sign classification model is loaded using Keras, enabling the classification of uploaded images.
- *Image Classification:* When an image is uploaded, it is resized and processed to fit the model's input requirements. The model then predicts the class of the traffic sign, and the result is displayed in the GUI.
- *User Interaction:* Users can upload an image using the "Upload an image" button. Once an image is uploaded, the "Classify Image" button appears, allowing users to classify the uploaded image. The result is then displayed on the GUI.

GUI Snapshot:



```
27:'Traffic signals',
28:'Pedestrians',
29:'Children crossing',
```

Future Scope:

The traffic sign recognition project has immense potential for future enhancements and applications. Here are some areas where we envision expanding and improving the system:

1. Mobile Application Integration
One of the most exciting prospects for this model is its integration into a mobile application. By leveraging the device's camera, the app can detect traffic signs in real-time. This would be particularly useful for drivers, providing them with immediate recognition and alerts about traffic signs they might miss otherwise. The app could also utilize GPS data to provide contextual information about the signs, enhancing navigation systems and improving road safety.

2. Text-to-Speech Feature
To make the GUI more accessible and user-friendly, we plan to incorporate a text-to-speech feature. This would allow the system to read out the classified traffic sign labels aloud, which can be particularly helpful for visually impaired users or drivers who need to keep their eyes on the road. The text-to-speech feature would enhance the usability of the application by providing audible feedback, ensuring users are always informed about the detected traffic signs without needing to look at the screen.

3. Language Translation
Another significant enhancement we plan to introduce is a language translation feature within the GUI. This would enable users to translate the classified traffic sign labels into their desired language. Such a feature would be incredibly beneficial for travelers in foreign countries, helping them understand traffic signs and navigate safely even if they do not speak the local language. The translation feature would make the system more versatile and user-friendly, catering to a global audience.

Conclusion:

In this project, we've delved into the fascinating realm of traffic sign recognition, exploring the potential of deep learning models to enhance road safety and navigation. It's been quite a journey, filled with countless hours of training and evaluating different models like MobileNet, ResNet, and our very own custom Convolutional Neural Network (CNN).

Imagine meticulously scrutinizing each model against a vast dataset of traffic signs, aiming to find the one that performs best in real-world scenarios. It's like searching for the perfect puzzle piece to complete the picture of efficient traffic sign recognition.

Our findings have revealed an exciting story: while MobileNet and ResNet performed decently with accuracies around 87.26% and 84.15% respectively, our custom CNN stole the show with an impressive 96.2% accuracy. It's like finding the hidden gem among a sea of options!

This significant improvement with our custom CNN highlights the importance of tailoring models to specific tasks. By fine-tuning our model specifically for traffic sign recognition, we've unlocked its full potential, making roads safer for everyone.

Looking ahead, our focus shifts to refining and optimizing our winning model. Picture us fine-tuning the engine of a race car, aiming to squeeze out every bit of performance. We envision seamlessly integrating our model into practical applications like mobile apps and advanced driver assistance systems, making navigation smoother and safer for people worldwide.

As we embark on the next phase of our journey, our commitment remains unwavering: to create a safer, smarter future for transportation. It's like setting sail towards uncharted waters, driven by our passion to innovate and make a positive impact. Through our continued efforts, we're not just advancing technology; we're building bridges to a more connected and secure world.