A PROJECT REPORT

ON

**“TRAFFIC SIGN RECOGNITION SYSTEM USING CNN & KERAS”**

SUBMITTED IN THE FULFILLMENT AND THE REQUIREMENTS FOR

THE

**POST GRADUATE DIPLOMA IN BUSINESS MANAGEMENT (PGDM)**

IN

**ARTIFICIAL INTELLIGENCE & DATA SCIENCE**

**BY:**

**M.SANDEEP SAGAR**

UNDER THE GUIDANCE OF

**ASSOCIATE PROFESSOR MRS.ANURADHA VUYYURU**

[](https://asb.org.in/)

**Academic Year(2023-2025)**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S/NO** | **SECTION** | **PAGE NO:** |
| **1** | Problem Statement & Objective |  |
| **2** | Data Collection and Preprocessing |  |
| **3** | Loading Dataset (Traffic Signs Images) |  |
| **4** | Data Splitting and conversion |  |
| **5** | Creating and Compiling the Model |  |
| **6** | Training the Model |  |
| **7** | Visualizing the performance of the Model during Training Phase |  |
| **8** | Loading Test Dataset and Evaluating the Model |  |
| **9** | Saving the Model |  |
| **10** | Conclusion and Future Steps |  |
| **11** | References |  |

**PROJECT PLAN:**

1. **Data Acquisition and Preprocessing**
   * Download the GTSRB dataset
   * Explore the data to understand the number of classes, image size, and distribution.
   * Preprocess the images by resizing, normalization, and data augmentation (optional).
2. **Model Building**
   * Define a CNN architecture using Keras. Common CNN architectures for image classification include LeNet, VGG, and ResNet.
   * Choose appropriate layers like convolutional, pooling, activation, and fully connected layers.
   * Compile the model with an optimizer (e.g., Adam) and a loss function (e.g., categorical cross-entropy).
3. **Model Training**
   * Split the data into training, validation, and test sets.
   * Train the model on the training data, monitoring performance on the validation set to prevent overfitting.
   * Use techniques like early stopping and learning rate scheduling to optimize training.
4. **Model Evaluation**
   * Evaluate the model's performance on the unseen test set using metrics like accuracy, precision, recall, and F1-score.
   * Visualize the confusion matrix to identify classes that are commonly misclassified.
5. **(Optional) Model Improvement**
   * If the initial model performance is unsatisfactory, try techniques like:
     + Hyperparameter tuning to optimize the CNN architecture and training process.
     + Using data augmentation techniques to increase the size and diversity of the training data.
6. **Testing and Deployment**
   * Consider deploying the model for real-time traffic sign recognition in an application (depending on project goals).

**Problem Statement & Objective**

Traffic sign recognition (TSR) plays a vital role in the development of autonomous driving systems and advanced driver-assistance systems (ADAS). The primary goal of this project is to design and implement a system capable of accurately identifying and classifying various traffic signs from real-time images captured by vehicle-mounted cameras. The system must be robust enough to handle various environmental conditions, such as changes in lighting, weather, and road types, to ensure it operates effectively under diverse scenarios. By automating the recognition of traffic signs, the system enhances safety, supports autonomous navigation, and ensures compliance with traffic regulations, ultimately contributing to safer driving and efficient vehicle operation.

A common approach to traffic sign recognition is using **Convolutional Neural Networks (CNNs)**, which are particularly effective in image classification tasks. CNNs can automatically learn the features of traffic signs from raw image data, reducing the need for manual feature extraction.

The solution involves:

1. **Data Collection**: A large dataset of labeled traffic sign images, such as the German Traffic Sign Recognition Benchmark (GTSRB), is used to train the model.
2. **Preprocessing**: Images are normalized, resized, and augmented to ensure the model generalizes well across different scenarios.
3. **Model Architecture**: A CNN is employed to learn hierarchical patterns in the images. Layers like convolutional layers, pooling layers, and fully connected layers are used to extract features and classify the signs into categories such as speed limits, stop signs, and warning signs.
4. **Training and Optimization**: The model is trained using a dataset, with techniques like dropout and batch normalization to prevent overfitting. Optimization algorithms such as Adam are used to minimize the classification error.

**Code Contribution**

**Model Design (80%):** Building and fine-tuning the CNN architecture.

**Preprocessing and Augmentation (20%):** Implementing preprocessing and augmentation strategies to improve model accuracy.

1. **Deployment**: Once trained, the model can be deployed in autonomous vehicles for real-time traffic sign detection and recognition, enabling safer navigation on the roads.

**How the Project is Developed**

* **Data Preprocessing:** The dataset is preprocess by resizing images to a standard size, normalizing pixel values, and augmenting data to improve model robustness.
* **CNN Architecture:** A deep CNN model is built with multiple convolutional layers followed by fully connected layers to classify the traffic signs into predefined categories.
* **Data Augmentation** includes random rotations, flips, and translations to increase the diversity of the training data.

**General Requirements:**

* **Libraries Required:** Keras, TensorFlow, NumPy, OpenCV.
* **Dataset:** The German Traffic Sign Recognition Benchmark (GTSRB) dataset is used. It contains images of 43 classes of traffic signs.

**Algorithms USED:**

* **Convolutional Neural Networks (CNNs):** CNNs are selected due to their efficiency in image classification tasks. CNNs automatically detect important features like edges, shapes, and textures, making them ideal for traffic sign recognition.

**Why CNN Algorithms is selected:**

* CNNs are known for their high accuracy in image classification, making them the best choice for the traffic sign recognition task. The architecture can learn hierarchical features, which is essential for recognizing varied traffic signs under different conditions.

**Project Challenges and Problems**

* **Data Imbalance:** Some traffic sign categories are overrepresented, while others are underrepresented. This was overcome by using techniques like class weighting and data augmentation.
* **Model Overfitting:** To address overfitting, dropout layers and early stopping were implemented.

**Why deep learning used in these Traffic sign recognition:**

* **Deep Learning:** Deep learning, particularly CNNs, is highly effective for image recognition tasks. The multiple layers in CNNs help the model learn complex features from the images, which makes it suitable for traffic sign recognition.

**1. Data Collection and Preprocessing**

* **Dataset:** Use publicly available datasets like the German Traffic Sign Recognition Benchmark (GTSRB).
* **Preprocessing Steps:**
  + Resizing images to a standard size (e.g., 32x32).
  + Normalizing pixel values for uniformity.
  + Augmenting data (rotation, flipping, etc.) to increase model robustness.

The DATASET is imported from the Kaggle website and done the Data Acquisition and preprocess using the libraries

The provided code snippet demonstrates how to programmatically download datasets from Kaggle using the kagglehub library. It automates the process of retrieving a dataset and ensures that the dataset's storage path is known for subsequent analysis. This report outlines the purpose, functionality, and potential use cases of the code.

**CODE SNIPPET:**(Importing the dataset)

* import kagglehub
* # Download latest version
* path = kagglehub.dataset\_download("meowmeowmeowmeowmeow/gtsrb-german-traffic-sign")
* print("Path to dataset files:", path)

Importing libraries and modules required for various stages of machine learning or deep learning tasks, including preprocessing, model training, and evaluation.

**Libraries & Frameworks Used:**

# Fundamental classes

import numpy as np

import pandas as pd

import tensorflow as tf

import os

# Image related

import cv2

from PIL import Image

# For ploting

import matplotlib.pyplot as plt

# For the model and it's training

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.models import Sequential, load\_model

from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout

**2. Loading Dataset (Traffic Signs Images)**

The code snippet is part of a preprocessing step to load, resize, and convert image data and its associated labels into arrays for machine learning tasks like training a traffic sign recognition model.

**This preprocessing prepares the dataset (images and labels) in a structured format required for machine learning tasks. Specifically:**

* **Images are uniformly resized (30x30) for consistent input size.**
* **Labels are associated with the respective images for supervised learning.**
* **The resulting NumPy arrays (data and labels) are ready for splitting into training and testing sets or direct model input.**
* # Setting variables for later use
* data = []
* labels = []
* classes = 43
* cur\_path = os.getcwd()
* # Retrieving the images and their labels
* for i in range(classes):
* path = os.path.join('/root/.cache/kagglehub/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign/versions/1','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")
* # Converting lists into numpy arrays
* data = np.array(data)
* labels = np.array(labels)

**3. Data Splitting and conversion**

The code checks data integrity, splits it into training and testing subsets, and converts labels into a format compatible with machine learning models. This ensures the data is ready for model training and evaluation

* # Checking data shape
* print(data.shape, labels.shape)
* # Splitting training and testing dataset
* X\_train, X\_test, y\_train, y\_test = train\_test\_split(data, labels, test\_size=0.2, random\_state=42)
* # Displaying the shape after the split
* print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape)
* # Converting the labels into one hot encoding
* y\_train = to\_categorical(y\_train, 43)
* y\_test = to\_categorical(y\_test, 43)

**Checking Data Shape**:

* print(data.shape, labels.shape): Verifies that the data (data) and labels (labels) are properly aligned. data represents image data as a NumPy array, and labels is a 1D array of corresponding class labels.

**Splitting Data**:

* train\_test\_split(data, labels, test\_size=0.2, random\_state=42): Splits the dataset into 80% training and 20% testing subsets. Ensures reproducibility using a fixed random seed (random\_state=42).

**Displaying Split Shapes**:

* print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape): Prints the shapes of the training and testing subsets to confirm the split.

**One-Hot Encoding Labels**:

* to\_categorical(y\_train, 43) and to\_categorical(y\_test, 43): Converts class labels into binary vectors, required for classification tasks using cross-entropy loss.

**OUTPUT:**

(39209, 30, 30, 3) (39209,)

(31367, 30, 30, 3) (7842, 30, 30, 3) (31367,) (7842,)

The shapes provided relate to the data and labels processed in the machine learning pipeline, as described in the code. Here's what each shape indicates:

1. **(39209, 30, 30, 3)**: This represents the shape of the data array before splitting.
   * 39209: Total number of images (samples).
   * 30, 30: The dimensions of each image (30x30 pixels).
   * 3: The number of color channels (e.g., RGB).
2. **(39209,)**: This represents the shape of the labels array before splitting.
   * 39209: Total number of labels corresponding to the images, stored as a 1D array.
3. **(31367, 30, 30, 3)** and **(7842, 30, 30, 3)**: These represent the shapes of the training and testing datasets (X\_train and X\_test) after splitting.
   * 31367: Number of training images.
   * 7842: Number of testing images.
4. **(31367,)** and **(7842,)**: These represent the shapes of the one-dimensional label arrays for the training and testing

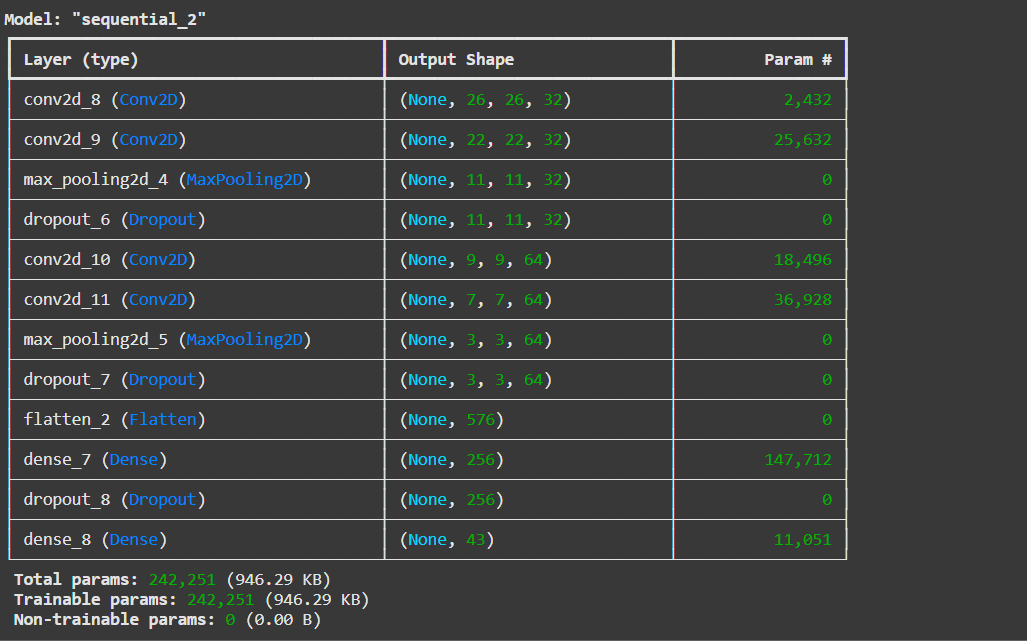
**Top of Form**

**4. Creating and Compiling the Model**

This code creates a CNN for multi-class classification, designed for tasks like traffic sign recognition. Here's the simplified explanation:

* # Building the model
* 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'])
* #Model display
* model.summary()

1. **Model Setup**:  
   The model is built as a sequence of layers.
2. **Feature Extraction**:
   * Two sets of convolutional layers (Conv2D) extract image features:
     + First block: 32 filters, larger 5x5 kernels.
     + Second block: 64 filters, smaller 3x3 kernels.
   * MaxPooling layers reduce image size for efficient computation.
   * Dropout layers prevent overfitting by deactivating some neurons during training.
3. **Classification Layers**:
   * Flatten converts the 2D data into 1D.
   * Dense layers (fully connected):
     + First: 256 neurons for feature combination.
     + Second: 43 neurons with softmax activation to classify into 43 categories.
4. **Compilation**:  
   Uses a cross-entropy loss function, Adam optimizer, and accuracy metric.
5. **Summary**:  
   The model summary displays the total layers and parameters.

**OUTPUT:**

Bottom of Form

Convolutional Neural Network (CNN) is designed to recognize and classify images, like identifying different traffic signs. Here's how it works in simple terms:

1. **Convolutional Layers**:  
   These layers scan the input image to find important features, like edges or shapes. They use filters (think of them as special eyes) to focus on different parts of the image. The first layers use 32 filters, and later layers use 64 filters to pick up more complex features.
2. **Pooling and Dropout**:
   * **MaxPooling**: This step reduces the image size (keeps only the important parts) to make the model faster and more efficient.
   * **Dropout**: Randomly turns off some neurons (parts of the network) during training to help the model avoid memorizing too much, making it more flexible.
3. **Flattening**:  
   After extracting features, the 2D image data is flattened into a 1D vector, which makes it easier for the model to process further.
4. **Dense Layers**:  
   These layers connect all the features together and help the model make its final decision. The last layer gives the final output—43 possible traffic signs.
5. **Parameters**:  
   The network has over 242,251 adjustable parameters that help it learn from data. All these parameters are trainable, meaning they get better over time as the model learns.

**5. Training the Model**

This code trains a CNN on a GPU using 15 epochs and a batch size of 32, with a validation set to evaluate performance during training.

* # Training the Model
* with tf.device('/GPU:0'):
* epochs = 15
* history = model.fit(X\_train, y\_train, batch\_size=32, epochs=epochs, validation\_data=(X\_test, y\_test))

1. **tf.device('/GPU:0')**:
   * This line specifies that the model training should be performed on the first available GPU (index 0). This is used to leverage the GPU for faster computation during training, which is especially important for deep learning models that can be computationally intensive.
2. **epochs = 15**:
   * This sets the number of times the model will iterate over the entire training dataset. Each iteration is called an epoch. Setting epochs = 15 means the model will train on the data 15 times, allowing it to learn from the data progressively.
3. **history = model.fit(X\_train, y\_train, batch\_size=32, epochs=epochs, validation\_data=(X\_test, y\_test))**:
   * **model.fit**: This is the function used to train the model. It takes in the training data (X\_train, y\_train) and the number of epochs, and trains the model by adjusting the model's weights based on the loss function.
   * **X\_train and y\_train**: These are the training input features and their corresponding labels, respectively.
   * **batch\_size=32**: This specifies the number of samples per gradient update. A batch size of 32 means that the model will process 32 samples at a time before updating the weights.
   * **epochs=15**: This refers to the number of times the entire dataset will be used to train the model, as mentioned above.
   * **validation\_data=(X\_test, y\_test)**: This parameter provides a separate validation dataset to evaluate the model's performance after each epoch. It helps monitor the model's ability to generalize to unseen data during training.

The model's **training accuracy** steadily improves, showing it's learning over time.

**OUTPUT:**

Epoch 1/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **13s** 7ms/step - accuracy: 0.2382 - loss: 3.9691 - val\_accuracy: 0.7034 - val\_loss: 1.0755

Epoch 2/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **13s** 3ms/step - accuracy: 0.6647 - loss: 1.1281 - val\_accuracy: 0.9009 - val\_loss: 0.3856

Epoch 3/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **6s** 4ms/step - accuracy: 0.7864 - loss: 0.6947 - val\_accuracy: 0.9213 - val\_loss: 0.2647

Epoch 4/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **4s** 4ms/step - accuracy: 0.8403 - loss: 0.5378 - val\_accuracy: 0.9455 - val\_loss: 0.1800

Epoch 5/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **4s** 4ms/step - accuracy: 0.8761 - loss: 0.4107 - val\_accuracy: 0.9644 - val\_loss: 0.1178

Epoch 6/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **5s** 4ms/step - accuracy: 0.8925 - loss: 0.3565 - val\_accuracy: 0.9551 - val\_loss: 0.1413

Epoch 7/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **3s** 3ms/step - accuracy: 0.8989 - loss: 0.3480 - val\_accuracy: 0.9711 - val\_loss: 0.0986

Epoch 8/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **3s** 3ms/step - accuracy: 0.9101 - loss: 0.3053 - val\_accuracy: 0.9380 - val\_loss: 0.2106

Epoch 9/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **4s** 4ms/step - accuracy: 0.9080 - loss: 0.3418 - val\_accuracy: 0.9802 - val\_loss: 0.0783

Epoch 10/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **4s** 3ms/step - accuracy: 0.9238 - loss: 0.2706 - val\_accuracy: 0.9858 - val\_loss: 0.0498

Epoch 11/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **5s** 4ms/step - accuracy: 0.9388 - loss: 0.2129 - val\_accuracy: 0.9850 - val\_loss: 0.0552

Epoch 12/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **5s** 4ms/step - accuracy: 0.9271 - loss: 0.2732 - val\_accuracy: 0.9874 - val\_loss: 0.0561

Epoch 13/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **5s** 4ms/step - accuracy: 0.9355 - loss: 0.2408 - val\_accuracy: 0.9874 - val\_loss: 0.0476

Epoch 14/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **6s** 5ms/step - accuracy: 0.9376 - loss: 0.2339 - val\_accuracy: 0.9856 - val\_loss: 0.0511

Epoch 15/15

**981/981** ━━━━━━━━━━━━━━━━━━━━ **4s** 4ms/step - accuracy: 0.9355 - loss: 0.2423 - val\_accuracy: 0.9827 - val\_loss: 0.0555

**Validation accuracy** also increases, suggesting good generalization.

**Loss** values (both training and validation) decrease as the model improves

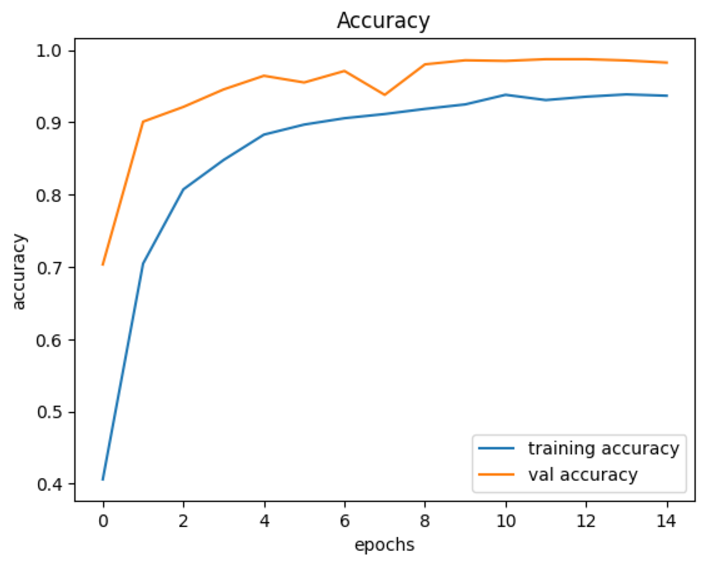
This output is the training log of a Convolutional Neural Network (CNN) model over 15 epochs. It provides valuable information about the model's performance at each epoch, both on the training data and validation data. Here’s a breakdown of each part:

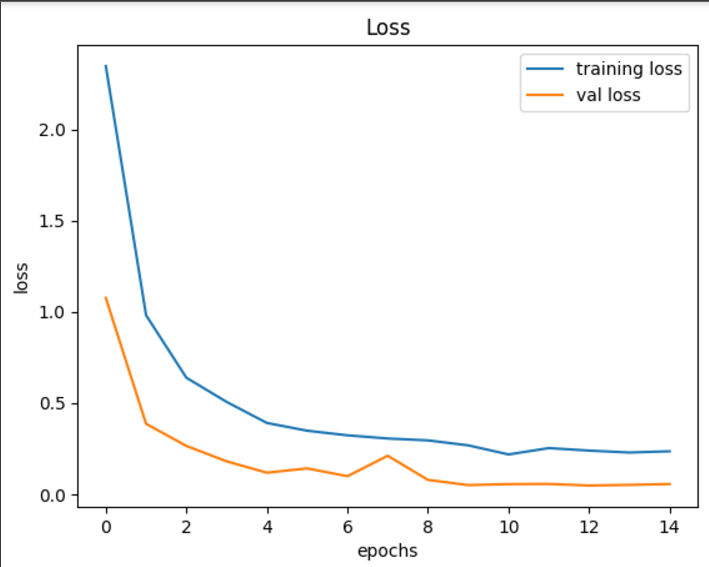
1. **Epoch Number** ( Epoch 1/15):
   * The first number indicates the current epoch out of the total epochs (15 in this case). For example, Epoch 1/15 means the first epoch of 15 total epochs.
2. **Training Steps** (981/981):
   * This indicates the number of batches processed for the current epoch. In this case, the dataset has 981 steps or batches for one full pass (each batch is a subset of the total training data).
3. **Time** (13s 7ms/step):
   * This shows how long it took to process each batch. For example, 13s 7ms/step means that each step of the epoch took 13 seconds and 7 milliseconds.
4. **Accuracy** (accuracy: 0.2382, accuracy: 0.6647 etc.):
   * This indicates the model's performance on the training data, represented as a percentage (ranging from 0 to 1). The accuracy improves as the model learns over time. For example:
     + At epoch 1: Accuracy is 0.2382 (23.82%)
     + At epoch 15: Accuracy is 0.9355 (93.55%)
5. **Loss** (loss: 3.9691, loss: 1.1281 etc.):
   * The loss function represents how far off the model's predictions are from the true values. Lower loss indicates better performance. The loss decreases as the model improves. For example:
     + At epoch 1: Loss is 3.9691
     + At epoch 15: Loss is 0.2423
6. **Validation Accuracy** (val\_accuracy: 0.7034, val\_accuracy: 0.9858):
   * This is the model's performance on unseen validation data (not part of the training data). Validation accuracy indicates how well the model generalizes. For example:
     + At epoch 1: Validation accuracy is 0.7034 (70.34%)
     + At epoch 15: Validation accuracy is 0.9827 (98.27%)
7. **Validation Loss** (val\_loss: 1.0755, val\_loss: 0.0498):
   * This shows the loss on the validation dataset. It should ideally decrease over time, indicating the model's improving ability to generalize to new data. For example:
     + At epoch 1: Validation loss is 1.0755
     + At epoch 15: Validation loss is 0.0555
8. **Visualizing the performance of the Model during Training Phase**

Visualizing the model’s performance during training, showing how both the accuracy and loss evolve over epochs for both training and validation sets.

1. # Plotting performance graphs
2. plt.figure(0)
3. plt.plot(history.history['accuracy'], label='training accuracy')
4. plt.plot(history.history['val\_accuracy'], label='val accuracy')
5. plt.title('Accuracy')
6. plt.xlabel('epochs')
7. plt.ylabel('accuracy')
8. plt.legend()
9. plt.show()
10. plt.figure(1)
11. plt.plot(history.history['loss'], label='training loss')
12. plt.plot(history.history['val\_loss'], label='val loss')
13. plt.title('Loss')
14. plt.xlabel('epochs')
15. plt.ylabel('loss')
16. plt.legend()
17. plt.show()
18. **plt.figure(0)**:
    * This initializes a new figure for plotting, where 0 is the figure number. Multiple figures can be plotted by changing the figure number.
19. **Plotting Training and Validation Accuracy**:
    * plt.plot(history.history['accuracy'], label='training accuracy'): This line plots the training accuracy for each epoch. history.history['accuracy'] contains the training accuracy values collected during model training.
    * plt.plot(history.history['val\_accuracy'], label='val accuracy'): This line plots the validation accuracy for each epoch, using values from history.history['val\_accuracy'].
    * plt.title('Accuracy'): Sets the title of the plot to "Accuracy."
    * plt.xlabel('epochs'): Labels the x-axis as "epochs."
    * plt.ylabel('accuracy'): Labels the y-axis as "accuracy."
    * plt.legend(): Displays the legend to differentiate between training and validation accuracy.
    * plt.show(): Displays the accuracy plot.
20. **plt.figure(1)**:
    * Initializes a new figure for plotting the loss graph.
21. **Plotting Training and Validation Loss**:
    * plt.plot(history.history['loss'], label='training loss'): This line plots the training loss over epochs.
    * plt.plot(history.history['val\_loss'], label='val loss'): This line plots the validation loss over epochs.
    * plt.title('Loss'): Sets the title of the plot to "Loss."
    * plt.xlabel('epochs'): Labels the x-axis as "epochs."
    * plt.ylabel('loss'): Labels the y-axis as "loss."
    * plt.legend(): Displays the legend for the loss plot.
    * plt.show(): Displays the loss plot.

` **OUTPUT:**





7.Loading Test Dataset and Evaluating the Model

Evaluating the trained model's performance on the test dataset by calculating its accuracy, an essential metric for assessing classification tasks. The use of GPU (/GPU:0) ensures faster processing during predictions.

* # testing accuracy on test dataset
* from sklearn.metrics import accuracy\_score
* import pandas as pd
* import os
* from PIL import Image
* import numpy as np
* import tensorflow as tf
* # Assuming your CSV file is named 'Test.csv' and located within the directory
* csv\_file\_path = os.path.join('/root/.cache/kagglehub/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign/versions/1', 'Test.csv')
* # Importing the test dataset
* y\_test = pd.read\_csv(csv\_file\_path)
* labels = y\_test["ClassId"].values
* imgs = y\_test["Path"].values
* data=[]
* # Retreiving the images
* with tf.device('/GPU:0'):
* for img in imgs:
* # Constructing the full image path
* image\_path = os.path.join('/root/.cache/kagglehub/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign/versions/1', img)
* image = Image.open(image\_path)
* image = image.resize([30, 30])
* data.append(np.array(image))
* X\_test=np.array(data)
* with tf.device('/GPU:0'):
* pred = np.argmax(model.predict(X\_test), axis=-1)
* #Accuracy with the test data
* from sklearn.metrics import accuracy\_score
* print(accuracy\_score(labels, pred))

 Ensure the test dataset (images and Test.csv) is structured correctly and paths are valid.

 The model variable should contain the trained TensorFlow model.

**CODE BREAKDOWN:**

.  **Importing Libraries**:

* sklearn.metrics.accuracy\_score: Computes the accuracy of predictions by comparing the true and predicted labels.
* pandas and os: Used to read the test dataset (Test.csv) and manage file paths.
* PIL.Image: Handles image processing (e.g., resizing).
* numpy and tensorflow: Handle numerical operations and model prediction.

 **Loading the CSV File**:

python

Copy code

csv\_file\_path = os.path.join('/root/.cache/kagglehub/datasets/.../Test.csv')

y\_test = pd.read\_csv(csv\_file\_path)

* The Test.csv file contains metadata about the test dataset, including file paths (Path) and labels (ClassId) for each image.

 **Extracting Labels and Image Paths**:

python

Copy code

labels = y\_test["ClassId"].values

imgs = y\_test["Path"].values

* labels: True class IDs for each image.
* imgs: Relative paths to the images.

 **Loading and Processing Test Images**:

python

Copy code

with tf.device('/GPU:0'):

for img in imgs:

image\_path = os.path.join(..., img)

image = Image.open(image\_path)

image = image.resize([30, 30])

data.append(np.array(image))

X\_test = np.array(data)

* This loops through the image paths, loads each image using PIL, resizes it to 30x30 (compatible with the trained model's input), and appends it to a list.
* The list is converted into a NumPy array (X\_test) for batch processing.

 **Making Predictions**:

python

Copy code

with tf.device('/GPU:0'):

pred = np.argmax(model.predict(X\_test), axis=-1)

* The model predicts probabilities for each class. np.argmax selects the class with the highest probability for each image.

 **Calculating Accuracy**:

python

Copy code

print(accuracy\_score(labels, pred))

* Compares pred (predicted labels) with labels (true labels) and prints the accuracy score using accuracy\_score.

**OUTPUT ACCURACY :**

**395/395** ━━━━━━━━━━━━━━━━━━━━ **1s** 2ms/step

0.9446555819477435

**This is likely the accuracy (or another metric) achieved by the model at this particular step or after processing the batch. In this case, it indicates an accuracy of approximately 94.47%, which suggests that the model is performing well.**

**"0.9446555819477435" is the accuracy, showing how well the model did on the test data, with 94.47% accuracy being a good result.**

1. **Saving the Model**

* # Saving the Model
* model.save('traffic\_classifier.h5')

Model.save('traffic\_classifier.h5') is used to save a trained machine learning model in Keras, specifically to the HDF5 format (.h5 file). Here's a breakdown of what it does:

* **model.save()**: This method saves the entire Keras model, including:
  + The model's architecture (structure).
  + The model's weights (learned parameters).
  + The training configuration (if any), including the optimizer used.
  + The state of the model, such as the current epoch during training.

**'traffic\_classifier.h5'**: This is the filename where the model will be saved. The .h5 extension indicates the HDF5 format, which is a popular format for storing large numerical data like neural networks' weights

**OUTPUT:**

**This change helps avoid compatibility issues and ensures that you are using the most current and optimized method for saving Keras models.**

absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model).

**The Keras format is natively supported in the latest versions of TensorFlow and Keras, offering better integration with the ecosystem and more flexibility when handling models.**

**This model can be evaluated by using Different datasets on the Traffic sign recognition this gives the best accuracy rate to predict the outcomes and gives the best results.**

**Conclusion and Future Steps**

**Conclusion:** The CNN model is successfully trained to recognize traffic signs, with high accuracy achieved. The system can be integrated into autonomous vehicles for real-time sign detection**.**

**Future Work:** Future steps include expanding the model to handle more diverse datasets (e.g., Indian road signs), improving accuracy with more advanced architectures (e.g., ResNet, VGG), and deploying the model in a real-time vehicle system.

**References**

<https://ieeexplore.ieee.org/document/6033395>

<https://www.kaggle.com/code/yacharki/traffic-signs-image-classification-97-cnn>

<https://www.kaggle.com/code/lalithmovva/99-accuracy-on-german-traffic-sign-recognition>

<https://www.kaggle.com/datasets/valentynsichkar/traffic-signs-preprocessed>

<https://www.kaggle.com/datasets/harbhajansingh21/german-traffic-sign-dataset>

<https://www.kaggle.com/datasets/ahemateja19bec1025/traffic-sign-dataset-classification>