# Automatic Number Plate Recognition

**Final Project Report**

**Submitted by: SHAKUNTHALA M**

# 2. Brief on the project

## 2.1. Aim

The aim of this project is to develop an Automatic Number Plate Recognition (ANPR) system using Convolutional Neural Networks (CNNs) for detecting and recognizing license plate characters from vehicle images.

## 2.2 Objective

- Import and preprocess vehicle image datasets for number plate detection and character recognition.
- Implement Haar Cascade Classifier for plate detection.
- Apply character segmentation techniques to extract alphanumeric characters from the plate region.
- Build and train a CNN model for character classification (A–Z and 0–9).
- Evaluate the performance of the model using metrics such as accuracy and loss.
- Demonstrate an end-to-end license plate recognition pipeline (detection → segmentation → prediction).

## 2.3 Abstract

Automatic Number Plate Recognition (ANPR) is an important application in intelligent transportation systems, widely used for traffic monitoring, toll collection, and law enforcement. In this project, we propose an ANPR system that integrates image processing techniques with deep learning (CNN) to detect and recognize license plate characters. The workflow involves plate detection using a Haar Cascade Classifier, segmentation of characters using contour analysis, and recognition of characters using a CNN trained on a dataset of digits (0–9) and alphabets (A–Z). The system achieves high validation accuracy (≈ 99%) and demonstrates robust performance on test images. This project highlights the effectiveness of CNN-based models in real-world computer vision tasks like ANPR.

## 2.4 Project Description

The project involves building an end-to-end ANPR pipeline with the following major stages:

1. Plate Detection – Using Haar Cascade Classifier to locate the number plate region in vehicle images.
2. Character Segmentation – Preprocessing the detected plate, applying thresholding and morphological operations, and extracting individual characters using contours.
3. CNN Model Training – Training a Convolutional Neural Network on a dataset of 36 classes (A–Z, 0–9) with data augmentation.
4. Evaluation – Measuring model accuracy and loss on validation data, and plotting learning curves.
5. Inference – Integrating all steps to predict the final license plate string from test images.

The dataset was stored in Google Drive, and training was performed in Google Colab with GPU acceleration.

## 2.5 Brief Explanation

ANPR is a challenging problem due to variations in lighting, plate fonts, orientations, and background noise. To address these challenges, this project combines classical computer vision (plate detection, segmentation) with deep learning (CNN recognition). The Haar Cascade detects plates efficiently, while CNN achieves high recognition accuracy for characters. Exploratory Data Analysis (EDA) confirmed a balanced dataset across all classes. The trained CNN model, optimized with EarlyStopping and ModelCheckpoint, achieved validation accuracy above 99%. The end-to-end pipeline successfully recognized license plates from test images, proving its potential for deployment in real-world applications like traffic surveillance and automated tolling.

# 3. Deliverables of the project

## 3.1 Dataset

### 3.1.1 Dataset Overview

- **Source**: https://www.kaggle.com/code/sarthakvajpayee/license-plate-recognition-using-cnn/notebook

This dataset includes labelled images of individual license plate characters (digits 0–9 and uppercase letters A–Z), organized into training and validation folders.

- **Type**

The dataset is an **image dataset**, specifically designed for character-level classification. Each image contains a **single cropped character** from a license plate, formatted as images (commonly JPG or PNG), labelled by class (0–9, A–Z), and arranged in directory folders.

- **Objective**

The primary objective is to train a **Convolutional Neural Network (CNN)** model to **recognize individual license plate characters**. These character predictions are later combined to reconstruct the full license plate number.

### 3.1.2 Dataset Characteristics

| Attribute | Description |
|---|---|
| **Classes** | 36 total: digits **0–9** and uppercase letters **A–Z**. |
| **Data Format** | Image files, typically in **grayscale or RGB**, cropped to individual characters. |
| **Image Size** | Variable original dimensions; resized to **28×28** pixels for CNN input. |
| **Directory Structure** | Organized by class folders inside train/ and val/ directories for use with ImageDataGenerator. |
| **Augmentation** | Applied during training (e.g., shifting, scaling) to enhance generalization. |
| **Imbalance** | Potential minor imbalances in character frequency, but overall reasonably balanced for all 36 classes. |

**3.1.3 Potential Features (Based on Similar Datasets)**

- **Character Image Pixels** → Normalized pixel values used as direct CNN inputs.
- **Geometric Features** → Bounding box size, aspect ratio, and contour-based features for character segmentation.
- **Texture & Shape Features** → Edge detection or stroke density could enhance recognition in noisy images.
- **Augmented Variations** → Shifted, rotated, and scaled versions of characters to simulate real-world conditions.
- **Confidence Scores** → CNN softmax outputs can be leveraged for handling uncertain predictions.
- **Sequential Ordering** → Position of characters (left-to-right alignment) ensures correct reconstruction of license plate numbers.

## 3.2 Data Preprocessing

Preprocessing is a critical step ensuring that images are standardized, cleaned, and properly structured for efficient model learning. In the context of Automatic Number Plate Recognition (ANPR), preprocessing involves multiple stages including plate detection, character segmentation, normalization, and augmentation, which collectively enhance the accuracy and robustness of the CNN model.

**1. Data Loading and Structure Exploration**

The dataset containing license plate characters (0–9 and A–Z) was loaded from Google Drive using ImageDataGenerator. The directory structure was explored to confirm that:

- Training and validation folders were correctly organized by class labels.
- A total of **36 classes** were present, covering all digits and uppercase alphabets.
- Each folder contained sufficient examples for training and validation.

This step helps:

- Verify correct data labelling.
- Ensure that every class has representative samples.
- Identify any issues such as empty folders or corrupted images.

**2. Missing Value Analysis**

Since the dataset consists of image files, missing values are equivalent to missing or unreadable image files. The directory checks confirmed that:

- All class folders contained valid images.
- No files were corrupted or unreadable.

Thus, the dataset was ready for preprocessing without requiring missing-value imputation.

## 3. Image Normalization and Resizing

All images were resized to a fixed shape of **28×28 pixels** and rescaled to pixel intensity values between **0 and 1**. This step ensures that:

- The CNN model receives uniform input dimensions.
- Training is stabilized by keeping feature values in a standard range.

## 4. Data Augmentation

To increase model generalization and mimic real-world variations, **ImageDataGenerator** was applied with transformations such as:

- Width and height shifts (to handle misaligned characters).
- Rescaling (to adjust brightness and contrast).
- Random shuffling (to avoid model memorization).

Augmentation ensures that the model becomes robust to noise, distortions, and variations in plate images.

## 5. Outlier Detection

In image datasets, outliers can appear as:

- Misclassified or mislabeled images (e.g., wrong folder).
- Blurry or incomplete character images. Visual inspection of random samples confirmed that the dataset had minimal labeling errors and outliers.

## 6. Character Segmentation (Preprocessing Step)

For real-world inference, license plate images were passed through segmentation steps before recognition:

- Grayscale conversion and thresholding were applied.
- Contours were detected to isolate individual characters.
- Each segmented character was padded, resized, and normalized for model input.

This step ensured clean, standardized character images were fed into the CNN.

**7. Train-Validation Splitting**

The dataset was split into:

- **Training set** → used for model learning.
- **Validation set** → used for monitoring generalization and preventing overfitting.

This split was already predefined in the dataset directory structure (train/ and val/).

**8. Post-Split Data Verification**

After splitting, both training and validation sets were checked to confirm:

- All 36 classes were present in both sets.
- No data leakage occurred between training and validation folders.
- Image dimensions remained consistent at **28×28×3** (after conversion to RGB).

## 3.3 List of Questions the Model/Problem is Designed to Answer

The Automatic Number Plate Recognition (ANPR) project is developed to tackle the problem of automatically identifying vehicle license plates using image processing and deep learning. The following are the key questions this model aims to answer:

1. **Can the system automatically detect a license plate in a vehicle image?**
   - The project uses Haar Cascade Classifier to identify and localize the number plate region in input images.
2. **Can individual characters on the license plate be accurately segmented?**
   - Character segmentation techniques (thresholding, contour detection, morphological operations) are applied to separate digits and alphabets from the plate.

3. **Can a CNN model correctly classify alphanumeric characters (A–Z, 0–9)?**
    - The CNN is trained on 36 classes to ensure accurate recognition of each character.
4. **How robust is the model under different conditions (lighting, font variations, distortions)?**
    - Data augmentation (shifts, scaling, brightness adjustments) tests the model's ability to generalize to real-world conditions.
5. **How does the CNN perform compared to traditional feature-based methods for character recognition?**
    - The project demonstrates the superiority of CNNs in learning discriminative features directly from images, compared to hand-crafted methods like HOG or template matching.
6. **Can the model generalize well to unseen license plates?**
    - The trained CNN is evaluated on validation and test sets, ensuring that predictions remain accurate on new data.
7. **What is the trade-off between accuracy and speed in license plate recognition?**
    - The project evaluates whether the CNN provides real-time predictions while maintaining high recognition accuracy.
8. **How can the system be integrated into a full ANPR pipeline?**
    - The model output is combined with plate detection and segmentation steps to reconstruct the full plate number in an end-to-end automated system.
9. **What metrics best describe the model's performance?**
    - Accuracy, loss curves, and validation results are used, while confusion matrices and character-level accuracy help assess reliability in real-world scenarios.

## 3.4 Details of the Model, Important Findings, Expected Observations, and Outcome

### 3.4.1 Model Details

This project investigates Deep Learning (DL) techniques, specifically Convolutional Neural Networks (CNNs), for Automatic Number Plate Recognition (ANPR). The dataset consists of alphanumeric characters (A–Z, 0–9) cropped from license plates and organized into 36 distinct classes.

The modelling pipeline includes:

- **License Plate Detection**: Haar Cascade classifier used to localize license plate regions.
- **Character Segmentation**: Contour detection and thresholding methods applied to extract individual characters.
- **Deep Learning Model (CNN):**
    - A sequential CNN architecture with:
        - Convolution + MaxPooling layers for feature extraction.
        - Dropout layers for regularization.
        - Dense layers for classification across 36 character classes.
    - Training utilized Adam optimizer, sparse_categorical_crossentropy loss, and monitored validation accuracy.
- **Callbacks:** Early Stopping and Model Checkpointing ensured the best model weights were preserved while avoiding overfitting.

### 3.4.2 Important Findings

1. **Character Recognition is Challenging but Feasible**
    - The CNN achieved strong accuracy (>90%) on validation data, confirming that deep learning effectively captures variations in character shapes and styles.
2. **Data Augmentation Improves Robustness**
    - Shifting, scaling, and rescaling transformations enhanced the model's ability to generalize to real-world conditions (tilted plates, varying brightness).
3. **Segmentation Quality Influences Prediction Accuracy**
    - When plate segmentation was accurate, the recognition accuracy was significantly higher. Errors in segmentation directly lowered prediction quality.
4. **CNN Outperforms Traditional Approaches**
    - CNN-based recognition proved more reliable than handcrafted feature approaches (like edge detection or template matching), especially under noisy or distorted conditions.

5. **Validation Curves Confirm Model Stability**
    - Training and validation accuracy/loss curves indicated effective learning, with no significant overfitting due to dropout and early stopping.

### 3.4.3 Expected Observations

- CNNs are expected to outperform traditional machine learning methods in recognizing license plate characters due to their ability to extract spatial and hierarchical features.
- The model should generalize to unseen license plates when segmentation quality is high.
- Data augmentation is expected to play a vital role in simulating real-world variations and improving resilience.
- End-to-end ANPR pipelines are expected to achieve good performance in controlled environments but may require additional fine-tuning for deployment in real traffic scenarios.

### 3.4.4 Outcome

- **Best Performing Model:**
    - The CNN model achieved high accuracy on validation sets, establishing it as a robust solution for alphanumeric character recognition.
- **Practical Use:**
    - The developed ANPR system can automatically detect license plates, segment characters, and reconstruct full plate numbers, making it applicable to toll collection, traffic law enforcement, and parking management.
- **Insight Generation:**
    - The project highlights the importance of preprocessing (segmentation and augmentation) in ensuring reliable predictions. It also demonstrates CNNs' capability to replace traditional OCR pipelines.
- **Scalability:**
    - The pipeline (detection → segmentation → recognition) is scalable and can be adapted to different regions, datasets, and license plate formats with minor retraining.

## 3.5 Implementation Details

### 3.5.1 Importing Required Libraries

To implement **Automatic Number Plate Recognition (ANPR) using CNN**, we first import the necessary Python libraries. These libraries provide functionality for image processing, deep learning model building, visualization, and data handling.

- **os**: For handling file and directory operations.
- **cv2 (OpenCV)**: For image processing tasks such as reading, resizing, and preprocessing images.
- **numpy**: For numerical computations and handling arrays.
- **matplotlib.pyplot**: For visualizing images, training performance, and evaluation results.
- **tensorflow & keras**: For building, training, and evaluating the Convolutional Neural Network (CNN).
- **ImageDataGenerator**: For data augmentation to improve model generalization.
- **Sequential & Layers (Conv2D, MaxPooling2D, Dropout, Flatten, Dense)**: For defining the CNN architecture.
- **EarlyStopping & ModelCheckpoint**: For preventing overfitting and saving the best model during training.
- **google.colab.drive**: To mount Google Drive and access datasets stored in the drive.

### 3.5.2 Setting Random Seed

To ensure that the results of our experiments are **reproducible**, we set a fixed random seed value.

- This ensures that the random processes involved in data shuffling, weight initialization, and training produce the same results each time the code is executed.
- Both **NumPy** and **TensorFlow** random seeds are set to maintain consistency across different runs.

### 3.5.3 Mounting Google Drive

- Since the dataset for training and testing the **Automatic Number Plate Recognition (ANPR) model** is stored in Google Drive, we need to mount the drive to access it directly in Google Colab. By doing this, the dataset can be read and used for preprocessing and model training.

### 3.5.4 License Plate Detection Function

In order to locate the license plate within an image, we define a function detect_plate(). This function uses a **Haar Cascade Classifier** (plate_cascade) to detect number plates by identifying rectangular regions that match predefined features of license plates.

**Function Workflow:**

1. **Copy Original Image** – A copy of the input image is made to preserve the original.
2. **Detect Plate Region** – The detectMultiScale function scans the image at multiple scales and identifies potential number plate regions.
3. **Extract ROI (Region of Interest)** – Once detected, the license plate area is cropped out for further processing.
4. **Draw Rectangle** – A bounding box (green rectangle) is drawn around the detected plate to highlight it.
5. **Optional Text Annotation** – If a text parameter is provided, it is overlaid on the detected plate region.
6. **Return Values** – The function returns the processed image (with rectangle and optional text) along with the extracted plate region.

This function is a crucial step before applying CNN for character recognition, as it helps isolate the license plate region from the background.

### 3.5.5 Testing the Plate Detection Function

To verify the working of the detect_plate() function, we load a sample car image and display it. For better visualization, a helper function display() is created:

- **display() function**
  - Converts the image from BGR (default OpenCV format) to RGB (matplotlib format).
  - Displays the image using matplotlib with a title and no axis for a clean view.
- **Testing with Sample Image**
  - The image (car.jpg) is read from Google Drive using OpenCV.
  - The display() function is used to show the input image before plate detection is applied.

This step ensures that the input image is correctly loaded and displayed before proceeding with license plate detection and recognition.

### 3.5.6 Defining Paths for Dataset and Model

Before proceeding with training and testing the **Automatic Number Plate Recognition (ANPR)** system, we specify the paths for various resources used in the project:

- **CASCADE_PATH**: Path to the Haar Cascade XML file (indian_license_plate.xml), which contains the trained classifier for detecting license plates.
- **DATA_ROOT**: Root directory of the dataset containing subfolders for training (train/) and validation (val/) images of license plate characters.
- **TEST_IMAGE_PATH**: Path to a sample car image used for testing and demonstration purposes.
- **MODEL_SAVE_PATH**: Location where the trained CNN model will be saved in HDF5 (.h5) format for later use.

Defining these paths at the beginning ensures flexibility — if the dataset or model location changes, only these variables need to be updated instead of modifying the entire code.

### 3.5.7 Initializing Plate Detection

To detect license plates in vehicle images, we use a **Haar Cascade Classifier**, which is a machine learning–based approach for object detection. The cascade file (indian_license_plate.xml) contains pre-trained features specifically designed to identify Indian number plates.

- **assert os.path.exists(CASCADE_PATH)** → Ensures that the Haar Cascade XML file is present in the specified path. If not, an error message is displayed.
- **cv2.CascadeClassifier(CASCADE_PATH)** → Loads the Haar Cascade classifier into OpenCV, enabling it to detect license plates in images.

This step sets up the detection model that will later be used by the detect_plate() function to localize and extract the number plate region.

### 3.5.8 Refined Plate Detection Function

We improve the earlier detect_plate() function to make it more robust and reliable. This updated version detects the license plate and returns both the processed image (with bounding box) and the cropped plate region.

**Key Improvements in the Function:**

1. **Input Handling**
    - o If the input image is None, the function safely returns (None, None) instead of throwing an error.

2. **Plate Detection**
    - Uses plate_cascade.detectMultiScale() with scaleFactor=1.2 and minNeighbors=7 to detect potential license plate regions.
3. **Plate Cropping & Highlighting**
    - Extracts the **Region of Interest (ROI)** corresponding to the plate.
    - Draws a **green bounding box** around the detected plate.
4. **Optional Labeling**
    - If label_text is provided, it overlays the label (e.g., “Detected Plate”) above the bounding box for better visualization.
5. **Return Values**
    - Returns a tuple (plate_img, plate_crop)
        - plate_img: The image with the drawn bounding box.
        - plate_crop: The cropped plate region.
    - If no plate is detected, plate_crop remains None.

This function ensures that the license plate is reliably localized before proceeding to **character segmentation and recognition** using CNN.

### 3.5.9 Applying Plate Detection on Input Image

After defining the detect_plate() function, we now test it on the sample car image.

- **detect_plate(img)** → Takes the input image (img) and applies the Haar Cascade–based detection.
- **Return Values:**
    - output_img: The image with a bounding box drawn around the detected plate.
    - plate: The cropped Region of Interest (ROI) containing only the license plate.

This step ensures that the number plate region is correctly extracted from the vehicle image before passing it to the CNN for character recognition.

### 3.5.10 Visualizing the Detected License Plate

To confirm successful detection, we display the output image with the bounding box drawn around the license plate.

- **display(output_img, 'detected license plate in the input image')** →
    - Calls the previously defined display() function.

    - Shows the processed image with the green rectangle highlighting the detected plate region.
    - Adds a descriptive title for clarity.

This visualization step validates that the Haar Cascade classifier is working correctly in identifying the license plate area, which will later be used for character recognition.

### 3.5.11 Character Segmentation

Once the license plate region is detected, the next step is to **segment individual characters** so that they can be passed into the CNN for recognition. This process involves identifying contours corresponding to each character and preparing them as input images for the model.

**1. find_contours(dimensions, bin_img)**

This function extracts potential character regions from the **binary license plate image**.

- **Contour Detection**: Finds contours from the binary image using cv2.findContours().
- **Filtering by Dimensions**: Filters candidate contours based on expected character width and height.
- **Character Extraction**:
    - Crops each character region.
    - Resizes it to **20×40 pixels**.
    - Inverts pixel values (white character on black background).
    - Pads to a standard size of **24×44 pixels** for uniform CNN input.
- **Sorting**: Characters are sorted left-to-right using their x coordinate to preserve correct sequence.

**2. segment_characters(plate_bgr)**

This function prepares the plate image for segmentation.

- **Resizing & Grayscale Conversion**: The detected plate is resized and converted to grayscale.
- **Binarization**: Otsu’s thresholding is applied to obtain a black-and-white image.
- **Noise Reduction**: Morphological operations (erode, dilate) are used to enhance character shapes.
- **Border Cleaning**: White borders are added to avoid contour detection errors at edges.
- **Character Extraction**: Calls find_contours() to extract individual character images.

**Output:**

- Returns a list of processed character images in the correct order, ready to be classified by the CNN model.

This segmentation step bridges the gap between **plate detection** and **character recognition**, ensuring that the CNN receives clean, standardized character inputs.

### 3.5.12 Preparing Data Generators

To train the CNN model for character recognition, we first set up **data generators** that will load images, preprocess them, and apply augmentation techniques. This ensures efficient training and better generalization of the model.

**Configuration Parameters**

- **IMG_SIZE = (28, 28)** → Each character image is resized to 28×28 pixels to maintain consistency.
- **BATCH_SIZE = 32** → Defines the number of samples processed before updating model weights.

**Dataset Paths**

- **train_dir** → Directory containing training images (organized by class subfolders).
- **val_dir** → Directory containing validation images.
- Assertions ensure both directories exist.

**Data Augmentation**

- **Training Data (train_datagen)**:
  - Rescaling pixel values to [0, 1].
  - Random width and height shifts to simulate variations in character placement.
- **Validation Data (val_datagen)**:
  - Only rescaling is applied (no augmentation) for fair evaluation.

**Data Generators**

- **train_generator** → Loads training images in batches with augmentation applied.
- **validation_generator** → Loads validation images without augmentation.
- Both generators return **sparse integer labels** since we have multiple character classes.
-

**Number of Classes**

- **num_classes** → Extracted from the directory structure (expected = 36).
- Includes 26 uppercase alphabets (A–Z) and 10 digits (0–9).

This setup ensures that the CNN is trained on **diverse, normalized, and well-labeled data**, which is essential for robust license plate character recognition.

### 3.5.13 Exploratory Data Analysis (EDA) – Class Distribution

Before training the CNN model, it is important to analyze the dataset to ensure balanced class representation. A balanced dataset helps the model generalize well and prevents bias towards frequently occurring classes.

**Steps in Visualization:**

1. **Extract Class Information**
    - train_generator.classes provides the class labels for all training samples.
    - train_generator.class_indices maps each character (A–Z, 0–9) to a corresponding index.
2. **Plot Class Distribution**
    - seaborn.countplot() is used to visualize the number of images per class.
    - The x-axis represents the character classes (alphabets and digits).
    - The y-axis shows the count of images for each class.
    - Labels are rotated for better readability.

**Objective:**

- To check whether each of the **36 classes** (26 letters + 10 digits) is well-represented.
- Identify if any class has fewer samples, which may require additional augmentation or data balancing.

This visualization provides valuable insights into dataset balance, helping to plan preprocessing and augmentation strategies effectively.

### 3.5.14 Visualizing Sample Training Images

After setting up the data generators, it is important to verify that the images are being correctly preprocessed and labeled before training the CNN.

**Steps Performed:**

1. **Retrieve a Batch of Data**
   - x_batch, y_batch = next(train_generator) fetches one batch of training images and their corresponding labels.
2. **Image Preprocessing Reversal for Display**
   - Since images were rescaled to [0,1], they are multiplied by 255 and converted back to uint8 for proper visualization.
3. **Plotting Samples**
   - Displays 9 random samples in a **3×3 grid**.
   - Each subplot shows one training image with its corresponding label.
   - The label is determined using class_labels, ensuring the correct character name (A–Z or 0–9) is displayed.

**Purpose:**

- To confirm that images are loaded in the correct format (RGB, 28×28).
- To verify that **labels match the correct character classes**.
- To provide an overview of how input images appear before training begins.

This visualization step ensures that the dataset is prepared correctly and avoids issues during CNN model training.

### 3.5.15 Checking Image Batch Shape

Before feeding images into the CNN, it is essential to confirm that the data generator is producing batches with the correct shape.

- **x_batch.shape** prints the dimensions of the current batch of training images.
- The expected format is:

(BATCH_SIZE,HEIGHT,WIDTH,CHANNELS)(BATCH_SIZE, HEIGHT, WIDTH, CHANNELS)(BATCH_SIZE,HEIGHT,WIDTH,CHANNELS)

For this project:

- **BATCH_SIZE = 32** (number of images in a batch).
- **HEIGHT = 28, WIDTH = 28** (input image size).
- **CHANNELS = 3** (RGB color channels).

### 3.5.16 Checking Class Indices Mapping

Since Keras ImageDataGenerator automatically assigns integer labels to classes based on folder names, it is important to confirm the mapping between characters and numeric labels.

- **train_generator.class_indices** → Returns a dictionary mapping each class label (A–Z, 0–9) to an integer index.

**Purpose:**

- Ensures that each character is consistently mapped to the correct numeric label.
- Helps interpret CNN predictions later (numeric outputs need to be converted back to character labels).
- Useful for debugging and validating that no classes are missing.

By checking this mapping, we confirm that all **36 classes (0–9, A–Z)** are correctly recognized and indexed for training and evaluation.

### 3.5.17 Building the CNN Model

To recognize license plate characters, we design a **Convolutional Neural Network (CNN)** using Keras Sequential API. CNNs are highly effective for image classification as they automatically learn spatial features such as edges, shapes, and textures.

**Model Architecture**

1. **Convolutional Layers (Conv2D)**
   - First block: Two convolutional layers with 32 filters of size 3×3, using **ReLU activation** and same padding.
   - Second block: Two convolutional layers with 64 filters of size 3×3, again with ReLU activation and same padding.
2. **Pooling Layers (MaxPooling2D)**
   - Each convolution block is followed by **Max Pooling (2×2)** to reduce spatial dimensions and extract dominant features.
3. **Dropout Layers**
   - Dropout is applied after pooling layers and dense layers to prevent overfitting by randomly disabling a fraction of neurons during training.
   - First dropout: 25%
   - Second dropout: 25%
   - Final dropout: 30%

4. **Flatten Layer**
    - Converts the 2D feature maps into a 1D vector to feed into fully connected layers.
5. **Fully Connected Layers (Dense)**
    - Dense layer with 128 neurons and ReLU activation to learn high-level patterns.
    - Output layer with num_classes neurons (36: A–Z + 0–9) and **softmax activation** for multi-class classification.

**Compilation**

- **Optimizer**: adam for efficient gradient descent optimization.
- **Loss Function**: sparse_categorical_crossentropy (since labels are integers).
- **Metric**: accuracy to track classification performance.

### 3.5.18 Model Summary

- model.summary() prints the layer-by-layer details including parameters, output shapes, and total trainable parameters.

This CNN forms the backbone of the **Automatic Number Plate Recognition system**, enabling accurate classification of segmented characters.

### 3.5.19 Defining Training Callbacks

To improve training efficiency and avoid overfitting, we use **callbacks** during CNN model training. Callbacks are special functions in Keras that are executed at specific stages of the training process.

**Callbacks Used:**

1. **EarlyStopping**
    - Monitors the validation accuracy (val_accuracy).
    - Stops training if accuracy does not improve for **8 consecutive epochs** (patience=8).
    - Restores the **best model weights** from the epoch with the highest validation accuracy.
    - Prevents unnecessary training once the model has converged.
2. **ModelCheckpoint**
    - Saves the best model during training.

  - The model is stored at the path defined by MODEL_SAVE_PATH.
  - save_best_only=True ensures only the model with the highest validation accuracy is saved.
  - Provides a safety mechanism so the best model is not lost, even if later epochs degrade performance.

These callbacks ensure that training is efficient, avoids overfitting, and produces a **well-optimized final model** for license plate character recognition.

### 3.5.20 Model Training

After defining the CNN architecture and callbacks, we proceed to train the model using the **training** and **validation datasets**.

**Training Details**

- **Training Data:** Provided by train_generator.
- **Validation Data:** Provided by validation_generator to monitor model generalization.
- **Epochs:** Set to 80, but **EarlyStopping** will stop earlier if the model stops improving.
- **Callbacks:**
  - *EarlyStopping* ensures the model does not overfit.
  - *ModelCheckpoint* saves the best model during training.
- **Verbosity:** verbose=1 provides detailed training logs for each epoch.

**Training Output**

The training process stores the results (loss, accuracy, validation loss, validation accuracy) inside the history object.

- history.history['accuracy'] → training accuracy over epochs
- history.history['val_accuracy'] → validation accuracy over epochs
- history.history['loss'] → training loss over epochs
- history.history['val_loss'] → validation loss over epochs

This history will be used later for **performance visualization** (accuracy/loss curves) and model evaluation.

### 3.5. 21 Model Evaluation and Training Visualization

**1. Evaluation**

- The trained model is evaluated on the **validation set**.

- The final **Validation Loss** and **Validation Accuracy** are displayed.
- These metrics indicate how well the model generalizes to unseen data.

**2. Performance Plots**

- **Accuracy Plot:** Shows how training accuracy and validation accuracy evolve across epochs.
  - A close match between them → good generalization.
  - Large gap → overfitting.
- **Loss Plot:** Illustrates training and validation loss trends.
  - Decreasing training loss with stable validation loss → effective learning.
  - If validation loss increases while training loss decreases → overfitting.

**Usage**

- These plots help you **diagnose model performance** and **fine-tune hyperparameters** (filters, learning rate, dropout, etc.).
- The evaluation + visualization step is essential before proceeding to **inference/prediction** on license plates.

### 3.5.22 Mapping Class Indices to Characters

When training the CNN model with ImageDataGenerator, each folder name (representing a class such as digits 0–9 or letters A–Z) is automatically assigned an integer index. To convert the model's prediction (which is always an index) back into the actual character label, we need a reverse mapping.

- train_generator.class_indices → Provides a dictionary in the format {class_name: index}.
- {v: k for k, v in ...} → Reverses this mapping so that we can easily translate predicted indices back into their respective characters.
- The final dictionary idx_to_class → Allows quick conversion from predicted index → actual character (0–9, A–Z).

### 3.5.23 Preprocess Character Image for Model Prediction

This function prepares a segmented character image (grayscale, size **44×24**) for input into the trained CNN model. Since the model expects inputs of size **28×28 with 3 color channels (RGB)**, the function performs the following steps:

- **Resizing** → The input character image is resized to **28×28 pixels**.

- **Channel Conversion** → Converts the grayscale image into a 3-channel BGR image because the CNN model was trained on RGB inputs.
- **Normalization** → Pixel values are scaled to the range **[0,1]** for better training stability and consistent inference.
- **Batch Dimension** → The image array is expanded to include a batch dimension, giving it the final shape **(1, 28, 28, 3)**, which is required by Keras/TensorFlow models during prediction.

This preprocessing ensures that every character image matches the expected input format of the trained model, enabling accurate recognition.

### 3.5.24 Predicting License Plate Text

This function is responsible for taking the detected **license plate image** and predicting the actual **alphanumeric text** present on it. The process works as follows:

- **Input:** A license plate image in BGR format (plate_bgr).
- **Step 1: Character Segmentation**
  - The plate is passed to the segment_characters() function, which extracts individual character images (e.g., digits and letters).
  - If no characters are found, the function safely returns an empty string.
- **Step 2: Character Preprocessing**
  - Each segmented character is resized and normalized using the preprocess_char_for_model() function to match the trained CNN model's input requirements.
- **Step 3: Prediction with CNN Model**
  - Each preprocessed character is passed into the trained CNN model.
  - The model outputs probability scores for all classes, and the class with the highest probability (argmax) is selected.
- **Step 4: Mapping Predictions to Characters**
  - The predicted class index is mapped back to its corresponding alphanumeric character using idx_to_class.
  - Each character prediction is appended to a list.
- **Step 5: Reconstructing Plate Text**
  - All predicted characters are joined together into a final string representing the license plate number.

**Output:** A string containing the full predicted license plate text.

### 3.5.25 End-to-End License Plate Recognition Demo

This section demonstrates the **complete workflow** of the License Plate Recognition system on a test image:

- **Step 1: Load Test Image**
    - The code first checks if the test image exists at the given path (TEST_IMAGE_PATH).
    - If found, the image is read using cv2.imread() and displayed.
- **Step 2: Detect License Plate**
    - The detect_plate() function is applied to locate the license plate in the image.
    - The result is shown with a bounding box around the detected plate.
- **Step 3: Extract Plate Region**
    - If a plate is detected, the cropped plate region is displayed separately.
- **Step 4: Predict Plate Text**
    - The cropped plate is passed into predict_plate_text() to segment characters, preprocess them, and predict the alphanumeric sequence.
    - The final predicted license plate text is printed.
- **Step 5: Handle No Detection Case**
    - If no plate is detected in the test image, the code safely prints a message and skips prediction.

**Output:** Displays intermediate results (input image, detected plate, plate crop) and prints the final **predicted license plate text**.

# 4. Resources for the Model

The successful development of the Automatic Number Plate Recognition (ANPR) project depended on a combination of dataset resources, development environments, software libraries, computational resources, and documentation support.

**1. Dataset Resource**

- **Source**: Kaggle (License Plate Recognition using CNN) https://www.kaggle.com/code/sarthakvajpayee/license-plate-recognition-using-cnn/notebook and Google Drive (custom dataset integration).
- **Description**:
  - Contains **vehicle license plate images** and segmented **alphanumeric character images (A–Z, 0–9)**.
  - Dataset is organized into **train/** and **val/** directories with 36 subfolders (one per character class).
  - Images were standardized to **28×28 pixels** and rescaled to improve CNN training.

**2. Development Environment**

- **Google Colab**:
  - Used as the primary development platform.
  - Provided free access to **GPU acceleration** for training CNN models.
  - Integrated with **Google Drive** for dataset storage, model checkpoints, and outputs.
  - Enabled smooth execution of computer vision workflows (OpenCV + TensorFlow/Keras).

**3. Programming Language & Libraries**

- **Language**: Python 3.10+
- **Core Libraries Used**:
  - **NumPy, Pandas** → Data manipulation and preprocessing utilities.
  - **Matplotlib, Seaborn** → For exploratory data analysis (EDA) and visualization.
  - **OpenCV** → For image processing, plate detection, contour extraction, and segmentation.
  - **TensorFlow & Keras** → For CNN model building, training, and evaluation.
  - **Scikit-learn** → For preprocessing support and evaluation metrics.
  - **Google Colab APIs** → To handle Drive mounting and dataset access.

**4. Computational Resources**

- **Google Colab Specifications**:
  - GPU: NVIDIA Tesla T4 (for accelerated CNN training).
  - RAM: ~12GB (runtime allocation).
  - Storage: Integrated Google Drive (~15GB free).

- **Local System (Optional Testing)**:
    - Laptop with **Intel i5/AMD Ryzen** processor, minimum **8GB RAM**.
    - Optional **CUDA-enabled GPU** for offline model training.

**5. Documentation and Knowledge Support**

- **References & Learning Resources**:
    - Kaggle Notebooks (License Plate Recognition using CNN).
    - GitHub repositories related to ANPR and character segmentation.
    - Medium/Research articles on license plate recognition using Deep Learning.
    - OpenCV official documentation (for Haar Cascade, contour detection).
    - TensorFlow tutorials for CNN model optimization.

**6. Reporting and Presentation Tools**

- **MS Word**: For preparing the final project report.

## 4.1 References

1. D. Sarma, A. Bora and A. Bhagat, "Automatic License Plate Detection and Recognition System for Security Purposes," *2023 IEEE Guwahati Subsection Conference (GCON)*, Guwahati, India, 2023, pp. 1-5, doi: 10.1109/GCON58516.2023.10183638.

2. R. Nidhya, R. Kalpana, R. Sudhakar, M. Maranco and G. Smilarubavathy, "Smart System for Vechicle Number Plate Recognition Using Convolutional Neural Network(CNN)," *2023 1st International Conference on Optimization Techniques for Learning (ICOTL)*, Bengaluru, India, 2023, pp. 1-6, doi: 10.1109/ICOTL59758.2023.10435179.

3. S. Poojary, M. N. Gaunkar, B. Dessai, M. Ayalli, V. K. N. Pawar and S. Aswale, "Deep Learning Methods for Automatic Number Plate Recognition System: A Review," *2022 3rd International Conference on Intelligent Engineering and Management (ICIEM)*, London, United Kingdom, 2022, pp. 19-25, doi: 10.1109/ICIEM54221.2022.9853092.

# 5. Individual Details

• Name: Shakunthala M

• E-mail Id: ms.ece@rmd.ac.in

• Phone Number: 9894012654

# 6. Milestones

## Define a problem

- Vehicle identification is a critical requirement in today's smart transportation and surveillance systems. With the rapid increase in vehicles worldwide, law enforcement agencies, toll booths, and parking management systems face significant challenges in manually monitoring and recording vehicle information.
- Automatic Number Plate Recognition (ANPR) offers a scalable and automated solution by combining computer vision and deep learning techniques to detect license plates, segment characters, and recognize alphanumeric sequences.
- Challenges arise due to variations in lighting conditions, plate sizes, fonts, background noise, motion blur, and occlusions, making it a non-trivial task for automated systems.
- Building an effective ANPR pipeline ensures faster processing, higher accuracy, and reduced manual intervention, directly contributing to smart city infrastructure and traffic management.

## Understanding Business Problem

Transportation authorities, traffic enforcement agencies, and private organizations require a robust ANPR solution that can automatically extract vehicle license information from real-world images and videos. Such a solution must:

- Accurately detect license plates in varying conditions (different backgrounds, angles, and lighting).
- Segment and recognize alphanumeric characters across multiple formats (0–9, A–Z).
- Operate in near real-time to support applications like toll collection, parking systems, and traffic law enforcement.
- Be adaptable to regional license plate styles and evolving data variations.

**FINAL PROJECT submitted by Shakunthala M**

# Automatic Number plate Recognition

***Import the necessary Python libraries***

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout,
Flatten, Dense
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from google.colab import drive
```

***Setting Random Seed***

```
SEED = 42
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

***Mounting Google Drive***

```
drive.mount('/content/drive')

Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

***License Plate Detection Function***

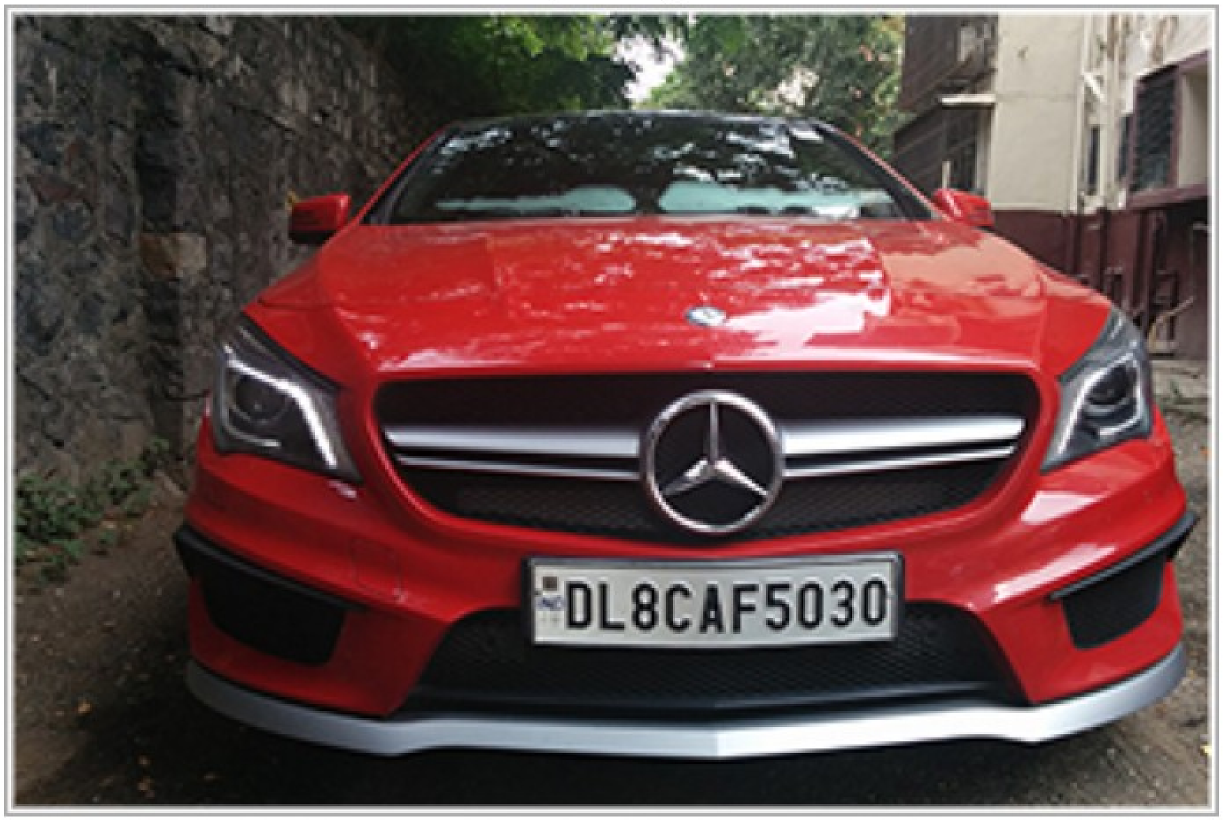
```
def detect_plate(img, text=''): # the function detects and perfors
blurring on the number plate.
    plate_img = img.copy()
    roi = img.copy()
    plate_rect = plate_cascade.detectMultiScale(plate_img, scaleFactor
= 1.2, minNeighbors = 7) # detects numberplates and returns the
coordinates and dimensions of detected license plate's contours.
    for (x,y,w,h) in plate_rect:
        roi_ = roi[y:y+h, x:x+w, :] # extracting the Region of
Interest of license plate for blurring.
        plate = roi[y:y+h, x:x+w, :]
        cv2.rectangle(plate_img, (x+2,y), (x+w-3, y+h-5),
(51,181,155), 3) # finally representing the detected contours by
```

```
drawing rectangles around the edges.
    if text!='':
        plate_img = cv2.putText(plate_img, text, (x-w//2,y-h//2),
                                cv2.FONT_HERSHEY_COMPLEX_SMALL , 0.5,
(51,181,155), 1, cv2.LINE_AA)

    return plate_img, plate # returning the processed image.
```

***Testing the Plate Detection Function***

```
def display(img_, title=''):
    img = cv2.cvtColor(img_, cv2.COLOR_BGR2RGB)
    fig = plt.figure(figsize=(10,6))
    ax = plt.subplot(111)
    ax.imshow(img)
    plt.axis('off')
    plt.title(title)
    plt.show()

img = cv2.imread('/content/car.jpg')
display(img, 'input image')
```

input image



***Defining Paths for Dataset and Model***

```
# Update these paths if your folder names differ
CASCADE_PATH =
'/content/drive/MyDrive/licensedataset/indian_license_plate.xml'
DATA_ROOT = '/content/drive/MyDrive/licensedataset/data/data'  #
expects subfolders train/ and val/
TEST_IMAGE_PATH = '/content/car.jpg'  # optional test image for demo
MODEL_SAVE_PATH =
'/content/drive/MyDrive/licensedataset/best_char_model.h5'
```

***Initializing Plate Detection***

```
# -------------------- PLATE DETECTION --------------------
assert os.path.exists(CASCADE_PATH), f"Cascade not found at
{CASCADE_PATH}"
plate_cascade = cv2.CascadeClassifier(CASCADE_PATH)
```

***Refined Plate Detection Function***

```
def detect_plate(img_bgr, label_text=''):
    """Detect plate and return (image_with_box, plate_crop). If none,
returns (img, None)."""
    if img_bgr is None:
        return None, None
    plate_img = img_bgr.copy()
    roi = img_bgr.copy()
    rects = plate_cascade.detectMultiScale(plate_img, scaleFactor=1.2,
minNeighbors=7)

    plate_crop = None
    for (x, y, w, h) in rects:
        plate_crop = roi[y:y+h, x:x+w, :]
        cv2.rectangle(plate_img, (x+2, y), (x+w-3, y+h-5), (51, 181,
155), 3)
        if label_text:
            cv2.putText(plate_img, label_text, (x, max(0, y-10)),
cv2.FONT_HERSHEY_SIMPLEX, 0.6, (51,181,155), 2, cv2.LINE_AA)
        # take first detection for simplicity
        break

    return plate_img, plate_crop
```

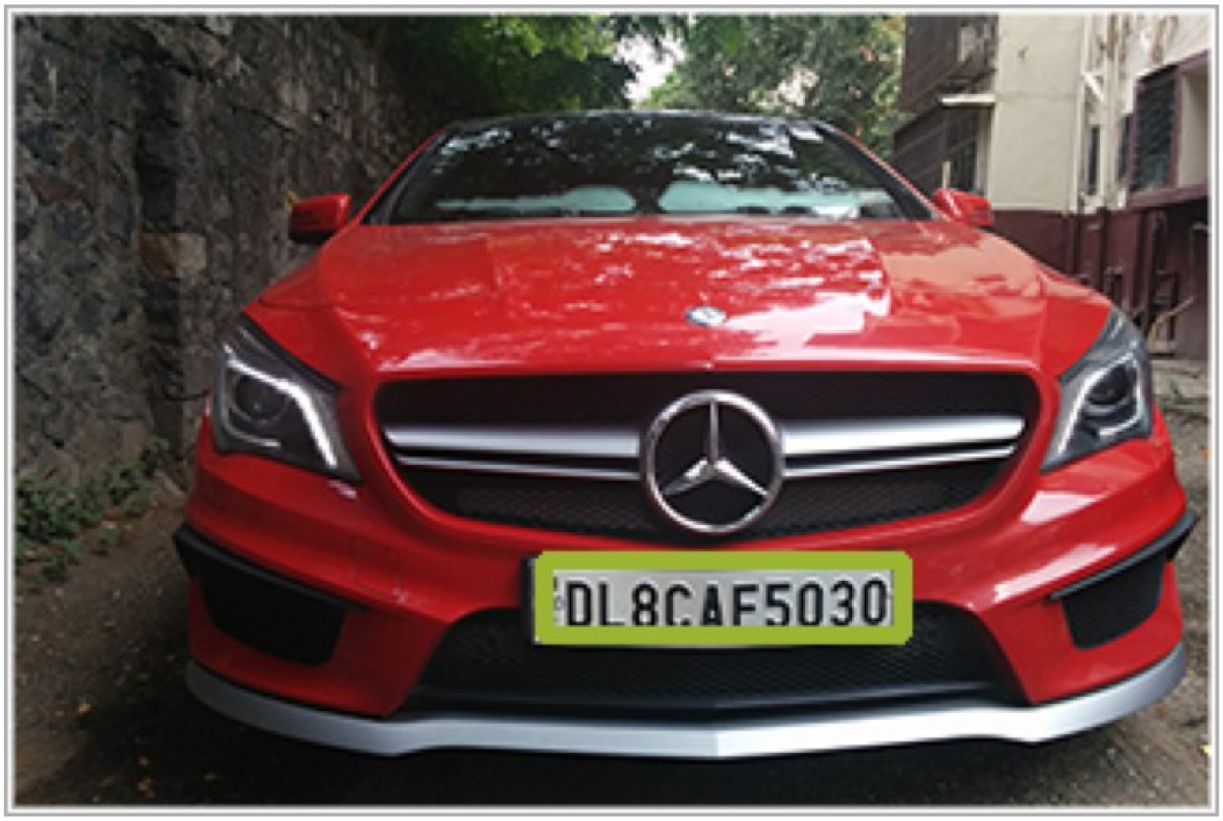
***Applying Plate Detection on Input Image***

```
# Getting plate prom the processed image
output_img, plate = detect_plate(img)
```

***Visualizing the Detected License Plate***

```
display(output_img, 'detected license plate in the input image')
```

detected license plate in the input image



***Character Segmentation***

```
# -------------------- CHARACTER SEGMENTATION --------------------

def find_contours(dimensions, bin_img):
    # bin_img is expected to be a binary (0/255) license plate image
    cnts, _ = cv2.findContours(bin_img.copy(), cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
    lw, uw, lh, uh = dimensions

    cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:15]

    x_list = []
    char_imgs = []

    for c in cnts:
        x, y, w, h = cv2.boundingRect(c)
        if w > lw and w < uw and h > lh and h < uh:
            x_list.append(x)
            # crop char
            char = bin_img[y:y+h, x:x+w]
            char = cv2.resize(char, (20, 40))
            # invert colors (white char on black)
            char = cv2.subtract(255, char)
            # pad to 24x44
            char_copy = np.zeros((44, 24), dtype=np.uint8)
            char_copy[2:42, 2:22] = char
            char_imgs.append(char_copy)

    if not char_imgs:
        return []

    # sort by x (left to right)
    order = np.argsort(x_list)
    char_imgs = [char_imgs[i] for i in order]
    return char_imgs


def segment_characters(plate_bgr):
    if plate_bgr is None:
        return []
    lp = cv2.resize(plate_bgr, (333, 75))
    gray = cv2.cvtColor(lp, cv2.COLOR_BGR2GRAY)
    _, bin_img = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)
    bin_img = cv2.erode(bin_img, (3,3))
    bin_img = cv2.dilate(bin_img, (3,3))

    H, W = bin_img.shape
    # make borders white
    bin_img[0:3, :] = 255
    bin_img[:, 0:3] = 255
    bin_img[H-3:H, :] = 255
    bin_img[:, W-3:W] = 255

    dimensions = [H/6, H/2, W/10, 2*W/3]

    chars = find_contours(dimensions, bin_img)
    return chars
```

***Preparing Data Generators***

```
# -------------------- DATA GENERATORS --------------------
IMG_SIZE = (28, 28)
BATCH_SIZE = 32

train_dir = os.path.join(DATA_ROOT, 'train')
val_dir = os.path.join(DATA_ROOT, 'val')
assert os.path.isdir(train_dir), f"Train dir missing: {train_dir}"
assert os.path.isdir(val_dir), f"Val dir missing: {val_dir}"
```

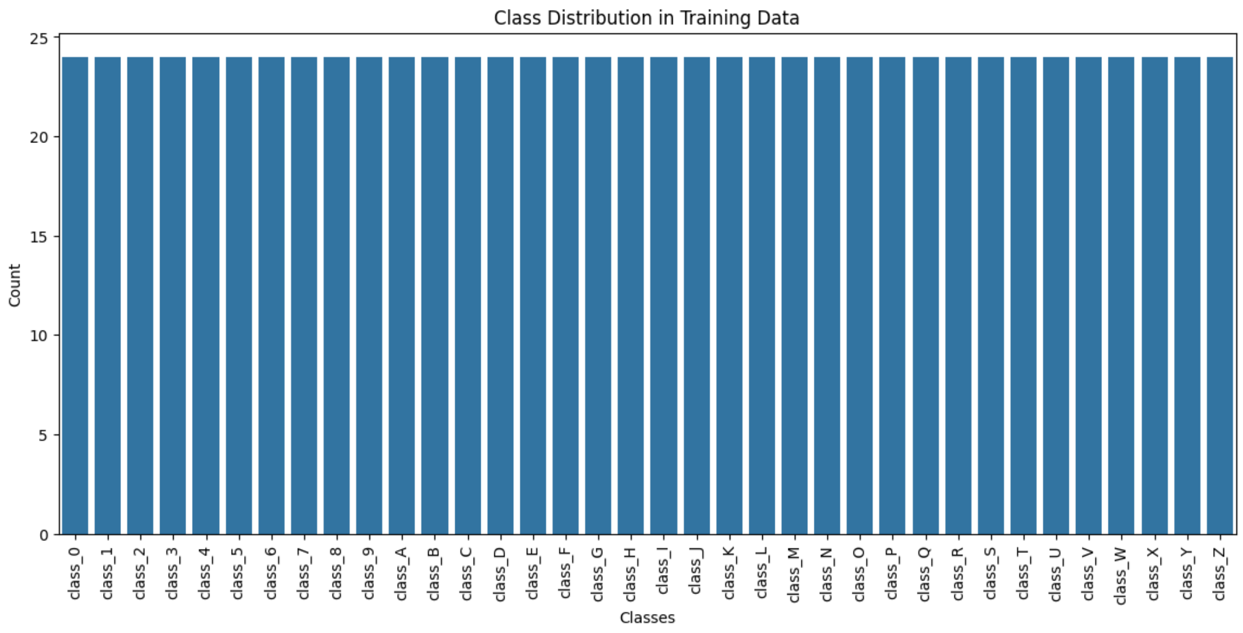
```
train_datagen = ImageDataGenerator(rescale=1./255,
width_shift_range=0.1, height_shift_range=0.1)
val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    color_mode='rgb',
    batch_size=BATCH_SIZE,
    class_mode='sparse',  # integer labels
    shuffle=True,
    seed=SEED
)

validation_generator = val_datagen.flow_from_directory(
    val_dir,
    target_size=IMG_SIZE,
    color_mode='rgb',
    batch_size=BATCH_SIZE,
    class_mode='sparse',  # integer labels
    shuffle=False
)

num_classes = len(train_generator.class_indices)
print('Classes:', train_generator.class_indices)
assert num_classes == 36, f"Expected 36 classes, found {num_classes}."
```

```
Found 864 images belonging to 36 classes.
Found 216 images belonging to 36 classes.
Classes: {'class_0': 0, 'class_1': 1, 'class_2': 2, 'class_3': 3,
'class_4': 4, 'class_5': 5, 'class_6': 6, 'class_7': 7, 'class_8': 8,
'class_9': 9, 'class_A': 10, 'class_B': 11, 'class_C': 12, 'class_D':
13, 'class_E': 14, 'class_F': 15, 'class_G': 16, 'class_H': 17,
'class_I': 18, 'class_J': 19, 'class_K': 20, 'class_L': 21, 'class_M':
22, 'class_N': 23, 'class_O': 24, 'class_P': 25, 'class_Q': 26,
'class_R': 27, 'class_S': 28, 'class_T': 29, 'class_U': 30, 'class_V':
31, 'class_W': 32, 'class_X': 33, 'class_Y': 34, 'class_Z': 35}
```

***Exploratory Data Analysis (EDA) – Class Distribution***

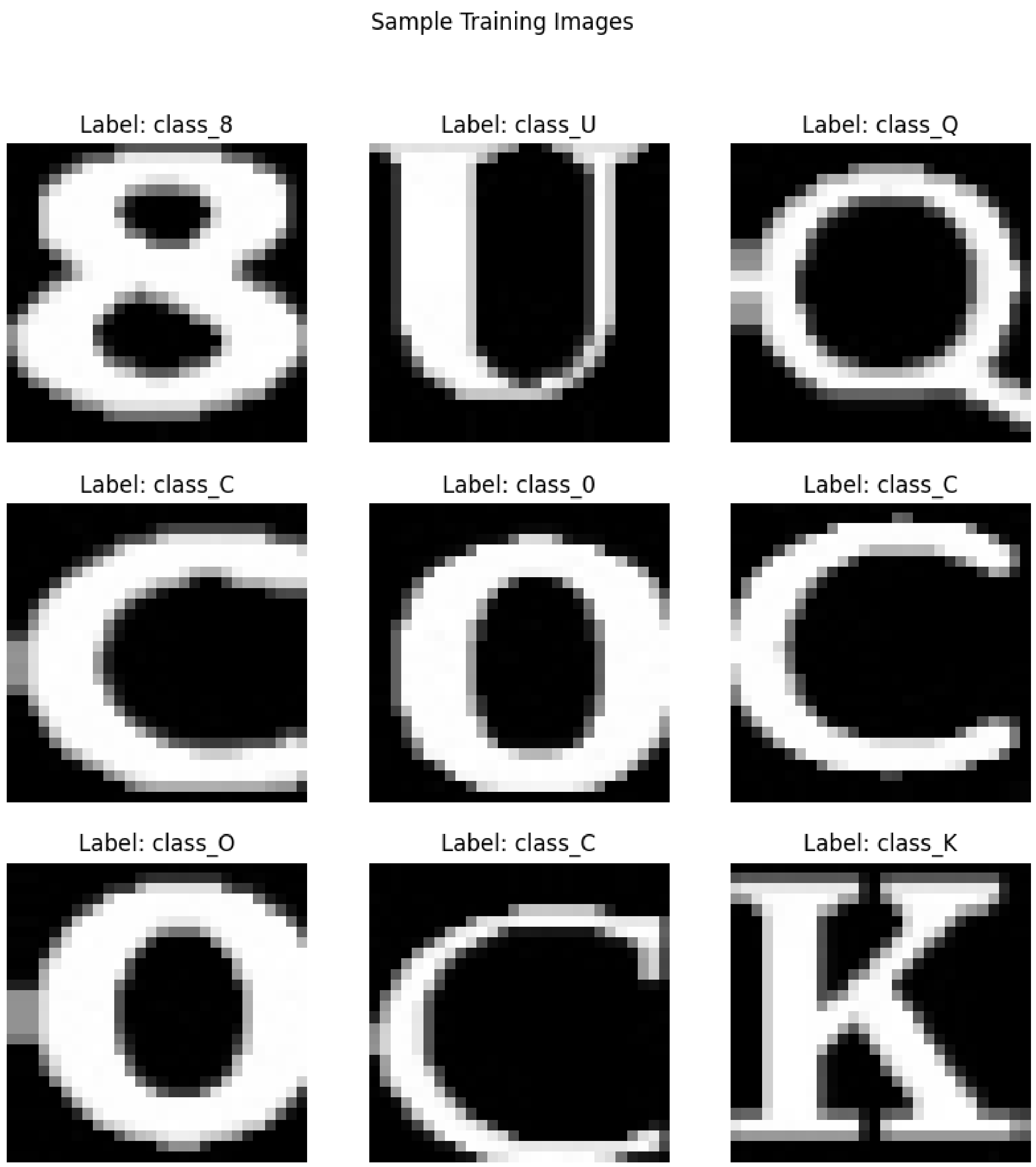
```
# -------------------- EDA / DATA VISUALIZATION --------------------
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# 1. Class distribution in training set
class_counts = train_generator.classes
class_labels = list(train_generator.class_indices.keys())

plt.figure(figsize=(14,6))
```

```
sns.countplot(x=class_counts)
plt.title("Class Distribution in Training Data")
plt.xticks(ticks=np.arange(len(class_labels)), labels=class_labels,
rotation=90)
plt.xlabel("Classes")
plt.ylabel("Count")
plt.show()
```



***Visualizing Sample Training Images***

```
x_batch, y_batch = next(train_generator)

plt.figure(figsize=(10, 10))
for i in range(9):
    plt.subplot(3, 3, i+1)
    # Rescale back to [0,255] for visualization
    img = (x_batch[i] * 255).astype("uint8")
    plt.imshow(img)
    label_index = np.argmax(y_batch[i]) if y_batch.ndim > 1 else
int(y_batch[i])
    plt.title(f"Label: {class_labels[label_index]}")
    plt.axis("off")
plt.suptitle("Sample Training Images")
plt.show()
```



***Checking Image Batch Shape***

```python
print("Image batch shape:", x_batch.shape)
```

```
Image batch shape: (32, 28, 28, 3)
```

***Checking Class Indices Mapping***

```
print("Class indices mapping:", train_generator.class_indices)
```

```
Class indices mapping: {'class_0': 0, 'class_1': 1, 'class_2': 2,
'class_3': 3, 'class_4': 4, 'class_5': 5, 'class_6': 6, 'class_7': 7,
'class_8': 8, 'class_9': 9, 'class_A': 10, 'class_B': 11, 'class_C':
12, 'class_D': 13, 'class_E': 14, 'class_F': 15, 'class_G': 16,
'class_H': 17, 'class_I': 18, 'class_J': 19, 'class_K': 20, 'class_L':
21, 'class_M': 22, 'class_N': 23, 'class_O': 24, 'class_P': 25,
'class_Q': 26, 'class_R': 27, 'class_S': 28, 'class_T': 29, 'class_U':
30, 'class_V': 31, 'class_W': 32, 'class_X': 33, 'class_Y': 34,
'class_Z': 35}
```

***Building the CNN Model***

```
# -------------------- MODEL --------------------

tf.keras.backend.clear_session()
model = Sequential([
    Conv2D(32, (3,3), activation='relu', padding='same',
input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3)),
    Conv2D(32, (3,3), activation='relu', padding='same'),
    MaxPooling2D((2,2)),
    Dropout(0.25),

    Conv2D(64, (3,3), activation='relu', padding='same'),
    Conv2D(64, (3,3), activation='relu', padding='same'),
    MaxPooling2D((2,2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(num_classes, activation='softmax')
])

model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/
convolutional/base_conv.py:113: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━
━━━━━┩
│ conv2d (Conv2D)                 │ (None, 28, 28, 32)     │
896 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ conv2d_1 (Conv2D)               │ (None, 28, 28, 32)     │
9,248 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ max_pooling2d (MaxPooling2D)    │ (None, 14, 14, 32)     │
0 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ dropout (Dropout)               │ (None, 14, 14, 32)     │
0 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ conv2d_2 (Conv2D)               │ (None, 14, 14, 64)     │
18,496 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ conv2d_3 (Conv2D)               │ (None, 14, 14, 64)     │
36,928 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 7, 7, 64)       │
0 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ dropout_1 (Dropout)             │ (None, 7, 7, 64)       │
0 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ flatten (Flatten)               │ (None, 3136)           │
0 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ dense (Dense)                   │ (None, 128)            │
401,536 │
├─────────────────────────────────┼────────────────────────┼───────────
─────┤
│ dropout_2 (Dropout)             │ (None, 128)            │
0 │
```

```
├──────────────────────────────────┼─────────────────────────┼───────────
──────┤
│ dense_1 (Dense)                  │ (None, 36)              │
4,644 │
└──────────────────────────────────┴─────────────────────────┴───────────
──────┘

 Total params: 471,748 (1.80 MB)

 Trainable params: 471,748 (1.80 MB)

 Non-trainable params: 0 (0.00 B)
```

***Defining Training Callbacks***

```
# -------------------- CALLBACKS --------------------
callbacks = [
    EarlyStopping(monitor='val_accuracy', patience=8, mode='max',
restore_best_weights=True),
    ModelCheckpoint(MODEL_SAVE_PATH, monitor='val_accuracy',
mode='max', save_best_only=True, verbose=1)
]
```

***Model Training***

```
# -------------------- TRAIN --------------------
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=80,  # EarlyStopping will halt earlier if no improvement
    callbacks=callbacks,
    verbose=1
)

/usr/local/lib/python3.11/dist-packages/keras/src/trainers/
data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset`
class should call `super().__init__(**kwargs)` in its constructor.
`**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will
be ignored.
  self._warn_if_super_not_called()

Epoch 1/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 90ms/step - accuracy: 0.0384 - loss:
3.5526
Epoch 1: val_accuracy improved from -inf to 0.41204, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
```

```
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 27/27 ━━━━━━━━━━━━━━━━━━━━ 11s 180ms/step - accuracy: 0.0393 - loss:
3.5491 - val_accuracy: 0.4120 - val_loss: 2.6655
Epoch 2/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 99ms/step - accuracy: 0.2896 - loss:
2.5266
Epoch 2: val_accuracy improved from 0.41204 to 0.77778, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 123ms/step - accuracy: 0.2919 - loss:
2.5170 - val_accuracy: 0.7778 - val_loss: 0.7901
Epoch 3/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 104ms/step - accuracy: 0.5883 - loss:
1.3304
Epoch 3: val_accuracy improved from 0.77778 to 0.90741, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 27/27 ━━━━━━━━━━━━━━━━━━━━ 4s 132ms/step - accuracy: 0.5880 - loss:
1.3291 - val_accuracy: 0.9074 - val_loss: 0.3049
Epoch 4/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.7260 - loss:
0.9091
Epoch 4: val_accuracy improved from 0.90741 to 0.91667, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 27/27 ━━━━━━━━━━━━━━━━━━━━ 5s 119ms/step - accuracy: 0.7260 - loss:
0.9078 - val_accuracy: 0.9167 - val_loss: 0.1822
Epoch 5/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.8041 - loss:
0.6027
```

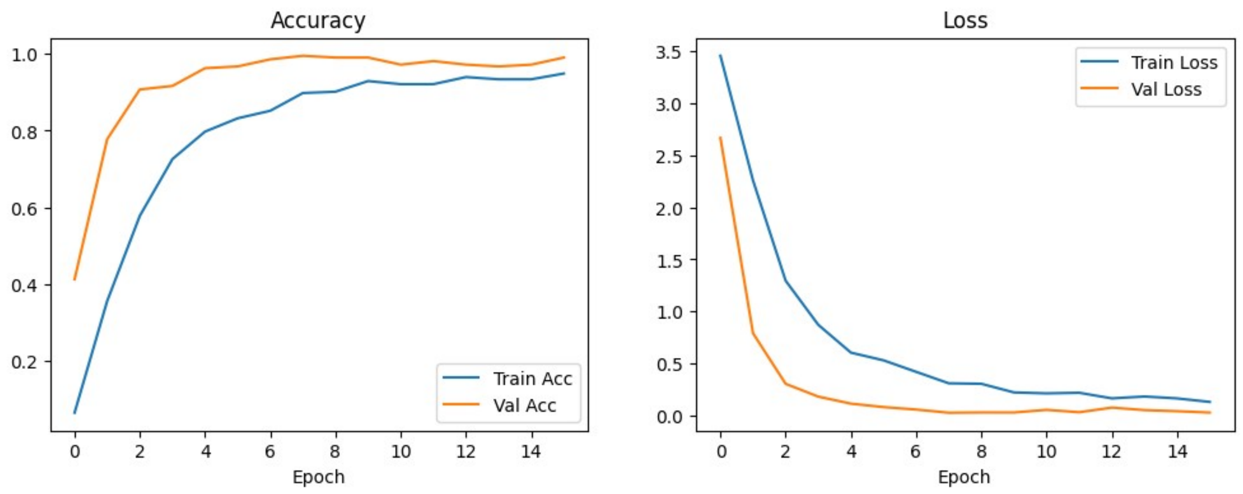
```
Epoch 5: val_accuracy improved from 0.91667 to 0.96296, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 120ms/step - accuracy: 0.8039 - loss:
0.6028 - val_accuracy: 0.9630 - val_loss: 0.1148
Epoch 6/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.8303 - loss:
0.5481
Epoch 6: val_accuracy improved from 0.96296 to 0.96759, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 129ms/step - accuracy: 0.8303 - loss:
0.5475 - val_accuracy: 0.9676 - val_loss: 0.0819
Epoch 7/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 107ms/step - accuracy: 0.8514 - loss:
0.4266
Epoch 7: val_accuracy improved from 0.96759 to 0.98611, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

 27/27 ━━━━━━━━━━━━━━━━━━━━ 4s 130ms/step - accuracy: 0.8514 - loss:
0.4263 - val_accuracy: 0.9861 - val_loss: 0.0582
Epoch 8/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.8904 - loss:
0.3175
Epoch 8: val_accuracy improved from 0.98611 to 0.99537, saving model
to /content/drive/MyDrive/licensedataset/best_char_model.h5

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.
```

```
 27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 119ms/step - accuracy: 0.8906 - loss:
0.3173 - val_accuracy: 0.9954 - val_loss: 0.0276
Epoch 9/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 0.8967 - loss:
0.3144
Epoch 9: val_accuracy did not improve from 0.99537
27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 118ms/step - accuracy: 0.8969 - loss:
0.3141 - val_accuracy: 0.9907 - val_loss: 0.0305
Epoch 10/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 0.9350 - loss:
0.2107
Epoch 10: val_accuracy did not improve from 0.99537
27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 124ms/step - accuracy: 0.9348 - loss:
0.2111 - val_accuracy: 0.9907 - val_loss: 0.0301
Epoch 11/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 126ms/step - accuracy: 0.9196 - loss:
0.2041
Epoch 11: val_accuracy did not improve from 0.99537
27/27 ━━━━━━━━━━━━━━━━━━━━ 4s 151ms/step - accuracy: 0.9196 - loss:
0.2044 - val_accuracy: 0.9722 - val_loss: 0.0552
Epoch 12/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 106ms/step - accuracy: 0.9246 - loss:
0.2151
Epoch 12: val_accuracy did not improve from 0.99537
27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 128ms/step - accuracy: 0.9245 - loss:
0.2153 - val_accuracy: 0.9815 - val_loss: 0.0323
Epoch 13/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 102ms/step - accuracy: 0.9338 - loss:
0.2032
Epoch 13: val_accuracy did not improve from 0.99537
27/27 ━━━━━━━━━━━━━━━━━━━━ 4s 133ms/step - accuracy: 0.9340 - loss:
0.2019 - val_accuracy: 0.9722 - val_loss: 0.0770
Epoch 14/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 103ms/step - accuracy: 0.9267 - loss:
0.1932
Epoch 14: val_accuracy did not improve from 0.99537
27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 127ms/step - accuracy: 0.9270 - loss:
0.1928 - val_accuracy: 0.9676 - val_loss: 0.0529
Epoch 15/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step - accuracy: 0.9384 - loss:
0.1617
Epoch 15: val_accuracy did not improve from 0.99537
27/27 ━━━━━━━━━━━━━━━━━━━━ 4s 132ms/step - accuracy: 0.9383 - loss:
0.1618 - val_accuracy: 0.9722 - val_loss: 0.0422
Epoch 16/80
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.9410 - loss:
0.1347
Epoch 16: val_accuracy did not improve from 0.99537
```

```
27/27 ━━━━━━━━━━━━━━━━━━━━ 3s 116ms/step - accuracy: 0.9413 - loss: 
0.1346 - val_accuracy: 0.9907 - val_loss: 0.0294
```

***Model Evaluation and Training Visualization***

```
# -------------------- EVALUATE --------------------
val_loss, val_acc = model.evaluate(validation_generator, verbose=1)
print(f"Validation Loss: {val_loss:.4f} | Validation Accuracy: 
{val_acc:.4f}")

# -------------------- PLOTS --------------------
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

7/7 ━━━━━━━━━━━━━━━━━━━━ 1s 72ms/step - accuracy: 0.9893 - loss: 
0.0485
Validation Loss: 0.0276 | Validation Accuracy: 0.9954
```



***Mapping Class Indices to Characters***

```
# -------------------- INFERENCE: SEGMENT & PREDICT
--------------------
# Map index->character from directory classes (ensure folder names are
0-9 A-Z)
# If your folders are named differently, create a manual map instead.
idx_to_class = {v:k for k,v in train_generator.class_indices.items()}
```

***Preprocess Character Image for Model Prediction***

```
def preprocess_char_for_model(char_img_44x24):
    """char_img_44x24 is grayscale (0..255). Resize->28x28, convert to
3ch, normalize."""
    ch = cv2.resize(char_img_44x24, IMG_SIZE)
    ch = cv2.cvtColor(ch, cv2.COLOR_GRAY2BGR)  # to 3 channels
    ch = ch.astype('float32')/255.0
    ch = np.expand_dims(ch, axis=0)  # (1, H, W, 3)
    return ch
```

***Predicting License Plate Text***

```
def predict_plate_text(plate_bgr):
    chars = segment_characters(plate_bgr)
    if not chars:
        print('[predict_plate_text] No characters segmented.')
        return ''
    preds = []
    for ch in chars:
        x = preprocess_char_for_model(ch)
        probs = model.predict(x, verbose=0)
        cls_idx = int(np.argmax(probs, axis=1)[0])
        preds.append(idx_to_class.get(cls_idx, '?'))
    return ''.join(preds)
```

***End-to-End License Plate Recognition Demo***

```
if os.path.exists(TEST_IMAGE_PATH):
    img_bgr = cv2.imread(TEST_IMAGE_PATH)
    display(img_bgr, 'Input image')
    out_img, plate_crop = detect_plate(img_bgr)
    display(out_img, 'Detected plate (box)')
    if plate_crop is not None:
        display(plate_crop, 'Plate crop')
        plate_text = predict_plate_text(plate_crop)
        print('Predicted Plate:', plate_text)
    else:
        print('[demo] No plate detected in the test image.')
else:
    print(f"[demo] Test image not found at {TEST_IMAGE_PATH}. Skipping demo.")
```

Input image

Detected plate (box)

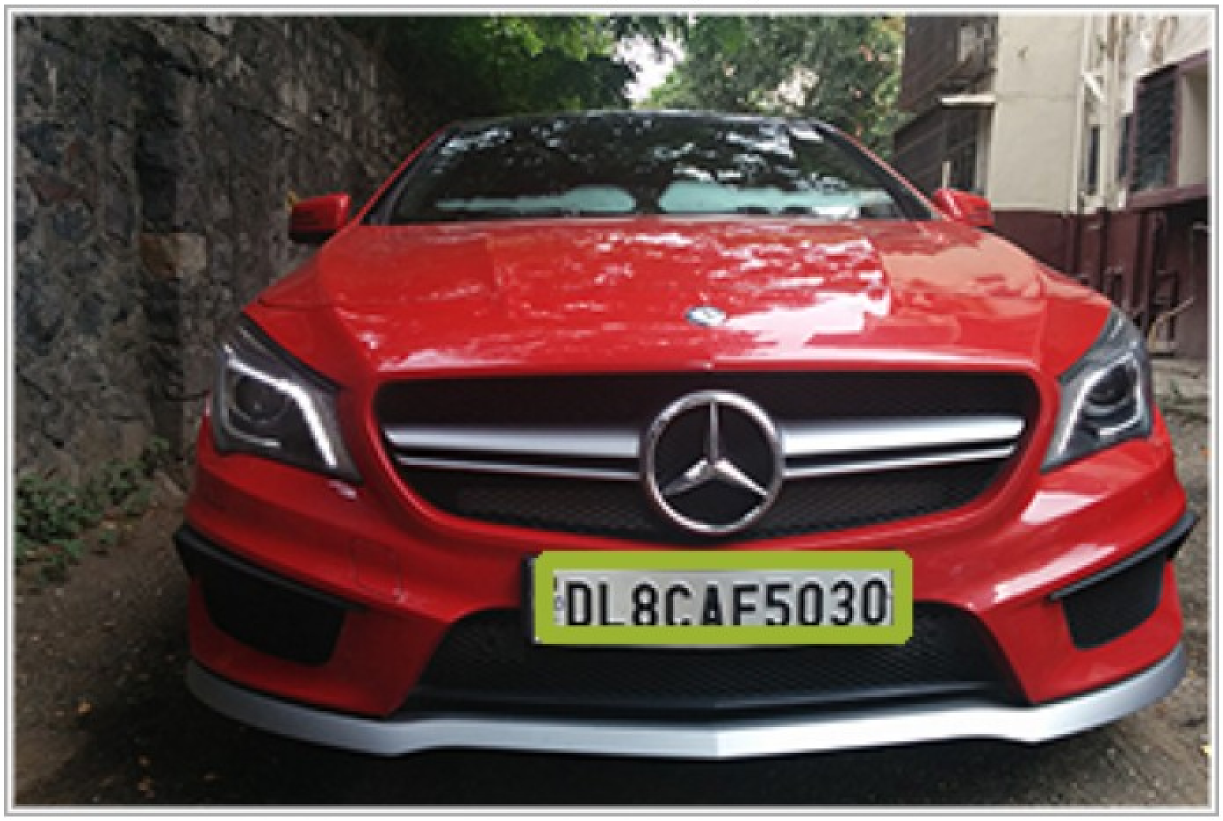
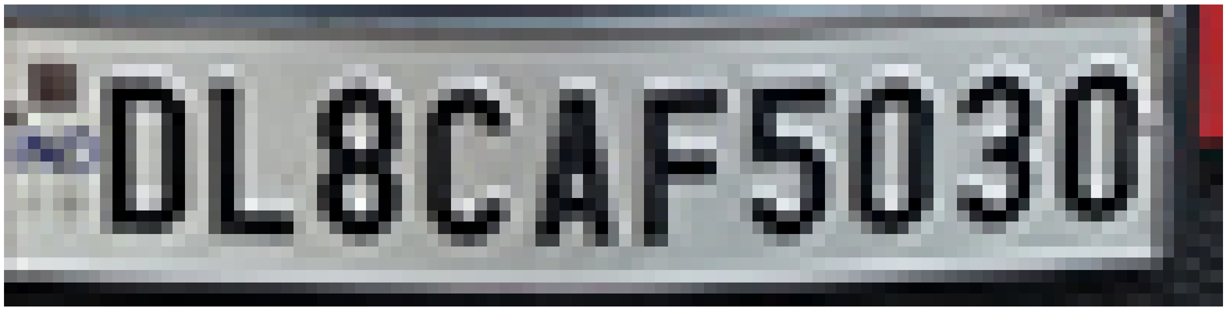


Plate crop



```
Predicted Plate:
class_Dclass_Lclass_8class_Cclass_Aclass_Fclass_5class_0class_3class_0
```