CASE STUDY

Traffic Sign Classification

LeNet derived Framework

Author: Sameer Hate
Date:07-06-2025

Table of Contents

Introduction

General

Traffic sign classification is a fundamental problem in the field of computer vision and intelligent transportation systems. It involves the automatic identification and categorization of traffic signs from images, a task that is critical for the development of autonomous vehicles, advanced driver-assistance systems (ADAS), and road safety monitoring. Correctly interpreting traffic signs allows vehicles to respond appropriately to road conditions, speed limits, warnings, and regulations, thereby reducing accidents and improving driving efficiency.

In this project, we tackle the traffic sign classification problem using the LeNet Convolutional Neural Network (CNN) architecture, one of the earliest and most influential deep learning models for image recognition. By leveraging supervised learning and a well-labeled dataset of traffic signs, we aim to build a model capable of accurately classifying images into their respective categories, such as speed limits, stop signs, pedestrian crossings, and more.

LeNet Framework

LeNet is one of the pioneering convolutional neural network (CNN) architectures, originally developed by Yann LeCun in the late 1980s for the task of handwritten digit recognition on the MNIST dataset. It laid the foundation for modern deep learning in image classification by introducing key concepts such as convolutional layers, pooling layers, and fully connected layers, which are now standard in CNN-based architectures.

The classic LeNet-5 architecture consists of seven layers (excluding input), including two convolutional layers, two average pooling layers, and three fully connected layers. The model uses the ReLU activation function in modern adaptations and is known for its simplicity and computational efficiency, making it ideal for small-scale image classification tasks like traffic sign recognition.

In this project, the LeNet framework is adapted and applied to the German Traffic Sign Recognition Benchmark (GTSRB) dataset to classify images into one of 43 traffic sign categories. Its well-defined structure and proven performance make it a suitable choice for developing a reliable and interpretable traffic sign classification model.

Author's Note

This case study presented in this work has been independently completed by me. This project serves as a medium to deepen my understanding of Data Science concepts and enhance my practical skills in applying machine learning and data analysis techniques to real-world problems.It may be prone to errors but I try my best to eliminate them as much as I can. In case you identify any such error or have a more optimal technique to solve this problem, your feedback and suggestions are always welcome as this will eventually help me strengthen my concepts.

Problem Statement & Expected Results

Objective

The primary objective of this case study is to develop a robust and efficient deep learning model based on the LeNet Convolutional Neural Network architecture for the accurate classification of traffic signs. By training the model on a labeled dataset of traffic sign images, the goal is to enable the automated recognition and categorization of various sign types, thereby demonstrating the applicability of CNNs in real-world computer vision tasks such as autonomous driving and intelligent traffic systems.

Goals

- To understand, modify and implement the LeNet Framework.
- To preprocess and prepare the traffic sign image dataset by applying grayscale conversion, normalization, and data shuffling to enhance model performance.
- To train the model on a labeled dataset with appropriate hyperparameters, including learning rate, batch size, and number of epochs.
- To draw conclusions about the effectiveness of LeNet in solving real-world computer vision problems like traffic sign recognition.

Expected Outcomes

The expected outcome of this case study is to develop a deep learning model based on the LeNet architecture that can accurately classify traffic sign images into their respective categories with high reliability and generalization capability. By leveraging convolutional neural networks, the model is anticipated to achieve strong performance on both training and unseen validation data

Additionally, the model should demonstrate the effectiveness of early CNN architectures like LeNet in solving real-world image recognition tasks, despite their simplicity compared to modern deep learning models. The case study also aims to highlight the importance of preprocessing techniques, regularization methods, and hyperparameter tuning in enhancing model accuracy and convergence. Ultimately, the results are expected to reinforce the applicability of CNNs in intelligent traffic systems and serve as a foundation for future enhancements using more advanced architectures.

Dataset

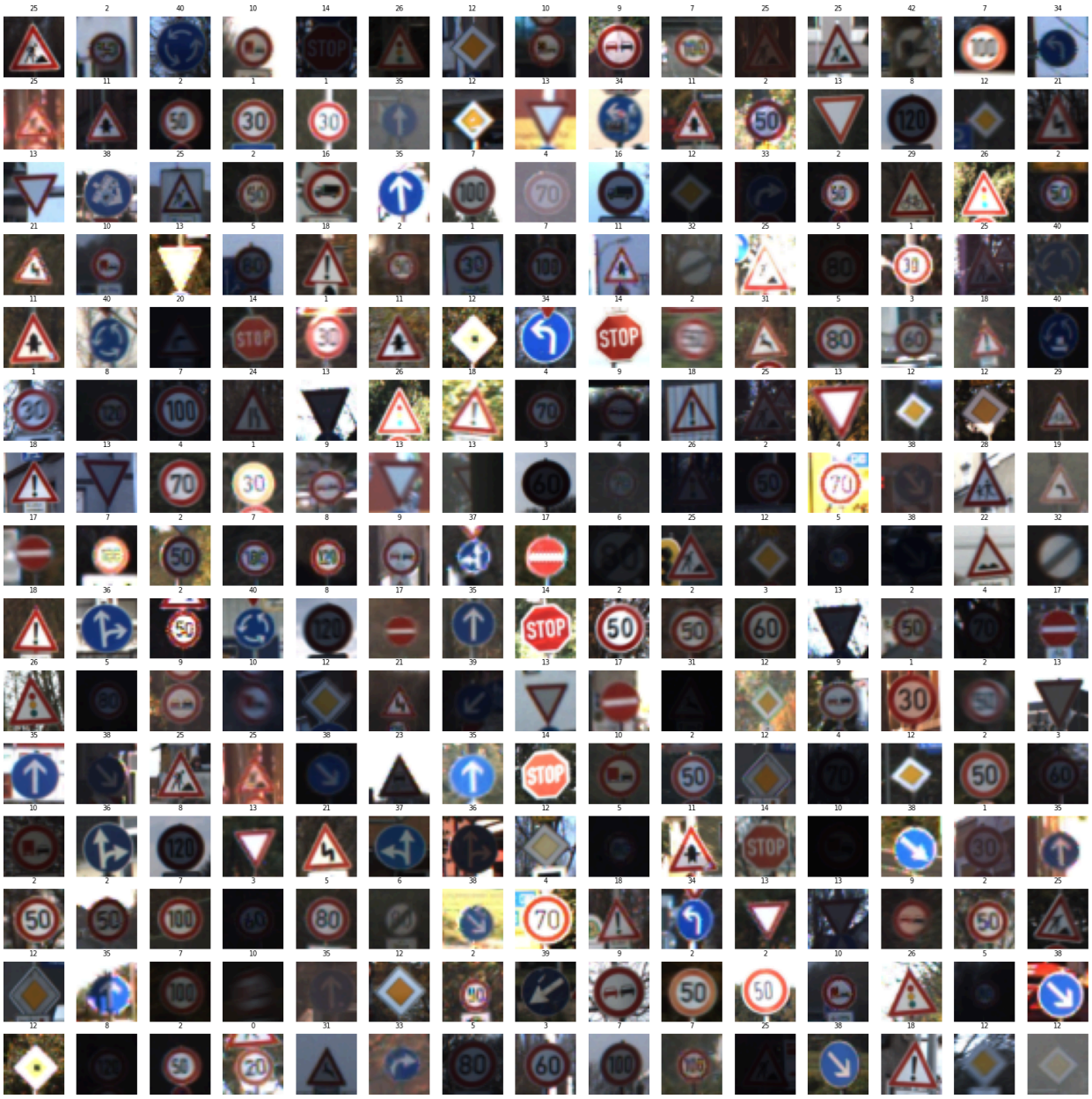
Description

The dataset used in this project is the German Traffic Sign Recognition Benchmark (GTSRB), which is a standardized dataset for multi-class image classification tasks in the domain of traffic sign recognition.

The dataset comprises over 50,000 color images of 43 different traffic sign classes, with each class representing a unique type of road sign, such as speed limits, warning signs, prohibitions, and direction indicators. Each image has a resolution of 32×32 pixels and varies in orientation, lighting, and background conditions, making it a challenging and realistic benchmark for computer vision models.

0 - Speed limit (20km/h) 1 - Speed limit (30km/h) 2 - Speed limit (50km/h) 3 - Speed limit (60km/h) 4 - Speed limit (70km/h) 5 - Speed limit (80km/h) 6 - End of speed limit (80km/h) 7 - Speed limit (100km/h) 8 - Speed limit (120km/h) 9 - No passing 10 - No passing for vehicles over 3.5 metric tons 11 - Right-of-way at the next intersection 12 - Priority road 13 - Yield 14 - Stop 15 - No vehicles 16 - Vehicles over 3.5 metric tons prohibited 17 - No entry 18 - General caution 19 - Dangerous curve to the left 20 - Dangerous curve to the right 21 - Double curve 22 - Bumpy road	23 - Slippery road 24 - Road narrows on the right 25 - Road work 26 - Traffic signals 27 - Pedestrians 28 - Children crossing 29 - Bicycles crossing 30 - Beware of ice/snow 31 - Wild animals crossing 32 - End of all speed and passing limits 33 - Turn right ahead 34 - Turn left ahead 35 - Ahead only 36 - Go straight or right 37 - Go straight or left 38 - Keep right 39 - Keep left 40 - Roundabout mandatory 41 - End of no passing 42 - End of no passing by vehicles over 3.5 metric tons

Dataset : The dataset is provided in the form of pre-processed Pickle (.p) files—namely, train.p, valid.p, and test.p—each containing dictionaries of image data and corresponding labels.



Pictures with their corresponding classes

Approach

- #IMPORTING LIBRARIES : This code cell is responsible for importing all the essential libraries required throughout the traffic sign classification project. The pandas library is used for data manipulation and handling tabular datasets, while numpy supports numerical operations and efficient array handling. Visualization of data is facilitated by matplotlib.pyplot and seaborn, with the latter offering aesthetically enhanced statistical plots. The pickle module is used to load serialized Python objects, particularly the pre-processed .p files containing the training, validation, and test datasets. Finally, tensorflow is imported to build and train the Convolutional Neural Network (CNN) model based on the LeNet architecture, which is the core framework used for classifying the traffic sign images.

- #IMPORTING THE DATASET : This code cell is responsible for importing the preprocessed dataset required for training and evaluating the traffic sign classification model. The dataset is provided in the form of serialized .p (pickle) files, which store Python objects efficiently. Using the pickle.load() function, the code loads three separate data files: train.p for training data, valid.p for validation data, and test.p for testing data. These files each contain dictionaries that include the image features (features) and their corresponding class labels (labels). This format allows for quick and organized access to the structured image data needed to train and evaluate the CNN model.

- #CREATING TRAINING, VALIDATION AND TESTING DATA : This code cell extracts the actual image data and corresponding labels from the loaded dataset dictionaries. Each dataset (training, validation, and testing) contains two main components: 'features', which represent the image data as NumPy arrays, and 'labels', which are integer values indicating the class of each traffic sign. By assigning these components to separate variables (X_train, y_train, etc.), the code prepares the data for preprocessing and model training. This separation makes it easier to manage input features and target outputs independently for the different stages of training and evaluation.

- #PRINTING A MATRIX OF IMAGES AND LABELS AT RANDOM : This code cell is used to visually depict a random sample of the training dataset by plotting a grid of traffic sign images along with their corresponding labels. The grid dimensions are set using W_grid and L_grid (15×15), meaning a total of 225 images will be displayed. The plt.subplots() function creates a large figure with multiple subplots, and the axes.ravel() flattens the axes array to allow easy iteration. For each subplot, a random index is chosen from the training set, and the corresponding image is displayed using imshow(), with the traffic sign label shown as the title.

- #CHECKING THE SHAPES OF THE DATA : This code cell displays the dimensions (shapes) of the training, validation, and testing datasets. By printing the shapes of both the image data (X_) and their corresponding labels (y_), it confirms that the data has been loaded correctly and is structured as expected. The image data (X_train, X_validation, X_test) typically has a 4-dimensional shape in the format (samples, height, width, channels)—in this case, it should be (n, 32, 32, 3) indicating RGB images of size 32×32 pixels. The label arrays (y_) are 1-dimensional and contain integer values representing the class index of each traffic sign. Verifying these shapes ensures compatibility with the input requirements of the neural network.

- #VISUALIZING THE DATA : This code cell is used to visualize a single image from the training dataset along with its corresponding label. The variable i is set to 1000, which means the 1000th image in X_train will be displayed using matplotlib.pyplot.imshow(). This provides a way to manually inspect individual samples and verify that the image data and labels align correctly. After the image is shown, the corresponding label from y_train[i] is output, indicating which traffic sign class the image belongs to. This step is helpful for sanity checking and understanding the raw input the model will learn from.

- #SHUFFLING THE ORDER OF THE DATA : This code cell shuffles the training data using Scikit-learn's shuffle function to randomize the order of the samples in X_train and y_train. Shuffling is a crucial preprocessing step that ensures the model does not learn any unintended sequence or order-based patterns from the data. Without shuffling, the model might be exposed to clusters of similar classes during training, which can lead to poor generalization and overfitting. By randomizing the data, each batch the model sees is more likely to be diverse, helping it learn more robust and generalized patterns.

- #GREYSCALING THE DATA : This code cell converts the original RGB images in the dataset to grayscale. Since the images have three color channels (Red, Green, Blue), dividing by 3 and summing across the third axis (axis=3) effectively averages the pixel intensities of all three channels to produce a single grayscale channel. The keepdims=True parameter retains the 4D shape of the input arrays (i.e., (samples, height, width, 1)), which is important for compatibility with convolutional layers that expect a channel dimension.

- #CHECKING NEW SHAPE OF THE DATA : This code cell prints the shapes of the grayscale image datasets—X_train_gray, X_validation_gray, and X_test_gray—after the grayscaling operation has been applied. Since each RGB image originally had a shape of (32, 32, 3), converting to grayscale reduces the number of channels from 3 to 1. However, due to keepdims=True, the shape is maintained as (32, 32, 1) for compatibility with convolutional layers that expect a 4D input (batch size, height, width, channels). This check confirms that the grayscale conversion was successful and that the data format is ready for input into the neural network.

- #NORMALIZING THE DATA : This code cell performs normalization on the grayscale image data by scaling the pixel values to a range of approximately -1 to 1. Originally, pixel values range from 0 to 255. By subtracting 128 and then dividing by 128, each pixel is transformed.
  This type of normalization helps the neural network converge faster and more efficiently during training by ensuring that input values are centered around zero. It also improves the stability of gradient-based optimization algorithms.

- #VISUALIZING THE GRAYSCALED AND NORMALIZED DATA : This code cell is used to visually compare the effects of the preprocessing steps—grayscaling and normalization—on a single sample image from the training dataset. It first displays the original RGB image using X_train[i], showing the traffic sign in its natural color form. Next, it displays the grayscale version of the same image using X_train_gray[i], which has been converted to a single-channel format and rendered using a grayscale color map. Finally, it displays the normalized grayscale image using X_train_gray_norm[i], where pixel values have been scaled to a range between approximately -1 and 1. Although normalization alters the underlying pixel values, the visual appearance remains suitable for human interpretation thanks to the grayscale rendering. This step serves as a useful verification to ensure that both grayscaling and normalization have been applied correctly before feeding the data into the CNN model.

- #BUILDING THE CNN - LE-NET DERIVED DEEP NETWORK : This code cell constructs a Convolutional Neural Network (CNN) based on an enhanced version of the LeNet architecture tailored for the traffic sign classification task. The model is defined using Keras' Sequential API, allowing layers to be stacked in a linear fashion.

  The network begins with a Conv2D layer containing 6 filters of size 5×5, followed by Batch Normalization to stabilize learning and speed up convergence. A MaxPooling2D layer then reduces the spatial dimensions, aiding in translation invariance. This pattern is repeated with a second convolutional layer (16 filters), another batch normalization, and pooling layer. A third Conv2D layer with 32 filters and smaller 3×3 kernel is added to capture more complex features, again followed by batch normalization and pooling.

  After the convolutional blocks, the model uses a Flatten layer to convert the 3D feature maps into a 1D vector, which is then passed through two fully connected (Dense) layers with 120 and 84 neurons respectively, both using ReLU activation. Dropout layers with a rate of 0.5 are inserted after each Dense layer to reduce overfitting by randomly disabling neurons during training. The final layer is a Dense output layer with 43 units—corresponding to the 43 traffic sign classes—and uses the softmax activation function to output a probability distribution over the classes.

  Overall, this model enhances the classic LeNet design by incorporating batch normalization and dropout, both of which improve training stability and generalization on real-world data.

- #COMPILING THE CNN : This code cell compiles the Convolutional Neural Network (CNN) using the Adam optimizer with a learning rate of 0.0003. The loss function used is sparse_categorical_crossentropy, which is suitable for multi-class classification problems where the labels are integers. The model is also configured to track accuracy as a performance metric during training and evaluation. This compilation step prepares the model for training by setting its optimization strategy and evaluation criteria.

- #ENHANCEMENTS - EARLY STOPPING : To improve training efficiency and prevent overfitting, EarlyStopping and ReduceLROnPlateau callbacks were implemented during model training. The EarlyStopping callback monitors the validation accuracy (val_accuracy) and halts the training process if there is no significant improvement for 30 consecutive epochs. Additionally, it restores the best weights observed during training to ensure optimal performance. The min_delta=0.0005 parameter ensures that only meaningful improvements are considered. Complementing this, ReduceLROnPlateau reduces the learning rate by half if the validation accuracy stagnates for 30 epochs, allowing the model to fine-tune its learning in later stages. Together, these enhancements help the model converge more efficiently while maintaining generalization and reducing the risk of overfitting.

- #ENHANCEMENTS - IMAGE AUGMENTATION : To increase the model's robustness and generalization capability, image augmentation was applied using TensorFlow's ImageDataGenerator. This technique artificially expands the training dataset by creating modified versions of the existing images. The augmentation settings included slight rotations (±10 degrees), zooming (±10%), horizontal and vertical shifts (10%), and horizontal flipping. These transformations simulate various real-world conditions such as slight camera angles or shifted views of traffic signs, helping the model become more invariant to such changes. By fitting the generator on the grayscale normalized training images (X_train_gray_norm), the model was exposed to a broader and more diverse set of training examples, which aids in reducing overfitting and improves its performance on unseen data.

- #TRAINING THE CNN - DATA AUGMENTATION : The CNN model was trained using the augmented dataset generated through ImageDataGenerator, which continuously supplies transformed batches of images during training. The fit method was used with datagen.flow(), which applies real-time data augmentation on the normalized grayscale training set (X_train_gray_norm). A batch size of 250 and a maximum of 300 epochs were specified, allowing the model ample opportunity to learn meaningful patterns. Validation was performed on the untouched validation set to monitor generalization. Additionally, EarlyStopping and ReduceLROnPlateau callbacks were included to prevent overfitting and dynamically adjust the learning rate when progress plateaued. This training strategy enhances the model's robustness by exposing it to varied inputs while optimizing training efficiency and stability.

- #EVALUATING THE CNN : After training, the performance of the Convolutional Neural Network was evaluated on the unseen test dataset using the evaluate method. This function computes the model's loss and accuracy on the test data (X_test_gray_norm, y_test), which had not been used during training or validation. The second value in the returned score list (score[1]) corresponds to the model's test accuracy, indicating how well the model can generalize to new, real-world data. A high test accuracy reflects that the model has successfully learned to recognize and classify traffic signs accurately.

- #CHECKING THE KEYS IN THE HISTORY : To gain insights into the model's training progress, the .history attribute of the training output was examined using history.history.keys(). This reveals the list of metrics that were tracked during training, such as 'loss', 'val_loss', 'accuracy', and 'val_accuracy'. These keys correspond to the training and validation loss and accuracy recorded at each epoch, and they are later used to visualize the learning curves. Monitoring these metrics helps assess how well the model is learning over time and whether issues like overfitting or underfitting are occurring.

- #EVALUATING THE HISTORY : To analyze the model's performance over the training period, the historical metrics stored during training were extracted from the history object. Specifically, training and validation accuracy ('accuracy', 'val_accuracy') and loss ('loss', 'val_loss') were retrieved. These metrics allow for a detailed examination of how the model's learning progressed across epochs, enabling the identification of key trends such as performance improvements, plateaus, or signs of overfitting. This data is especially useful when visualized, as it provides a clearer understanding of the effectiveness and stability of the training process.

- #VISUALIZING THE ACCURACY : To visually assess the model's learning performance, training and validation accuracy and loss were plotted over the number of epochs. The red curve represents the model's performance on the training set, while the blue curve shows how well the model performed on the validation set. The first plot illustrates the accuracy, revealing whether the model is learning to classify traffic signs effectively. The second plot shows the loss values, indicating how far the model's predictions are from the actual labels during training and validation. These visualizations help in identifying training trends, such as consistent improvement, convergence, or divergence between training and validation curves — which can signal overfitting or underfitting. Overall, these plots are critical for evaluating the model's learning behavior and generalization performance.

- #PREDICTING THE TESTING DATA : The trained Convolutional Neural Network was used to predict the classes of the unseen test dataset using the predict method on X_test_gray_norm. This step generates a set of probability scores for each class, for every input image in the test set. These predictions represent the model's confidence in assigning a given input to one of the 43 traffic sign categories. The variable y_true stores the actual class labels from the test set, which will later be used to compare with the predicted results and evaluate the model's classification performance through metrics such as accuracy, confusion matrix, and classification reports.

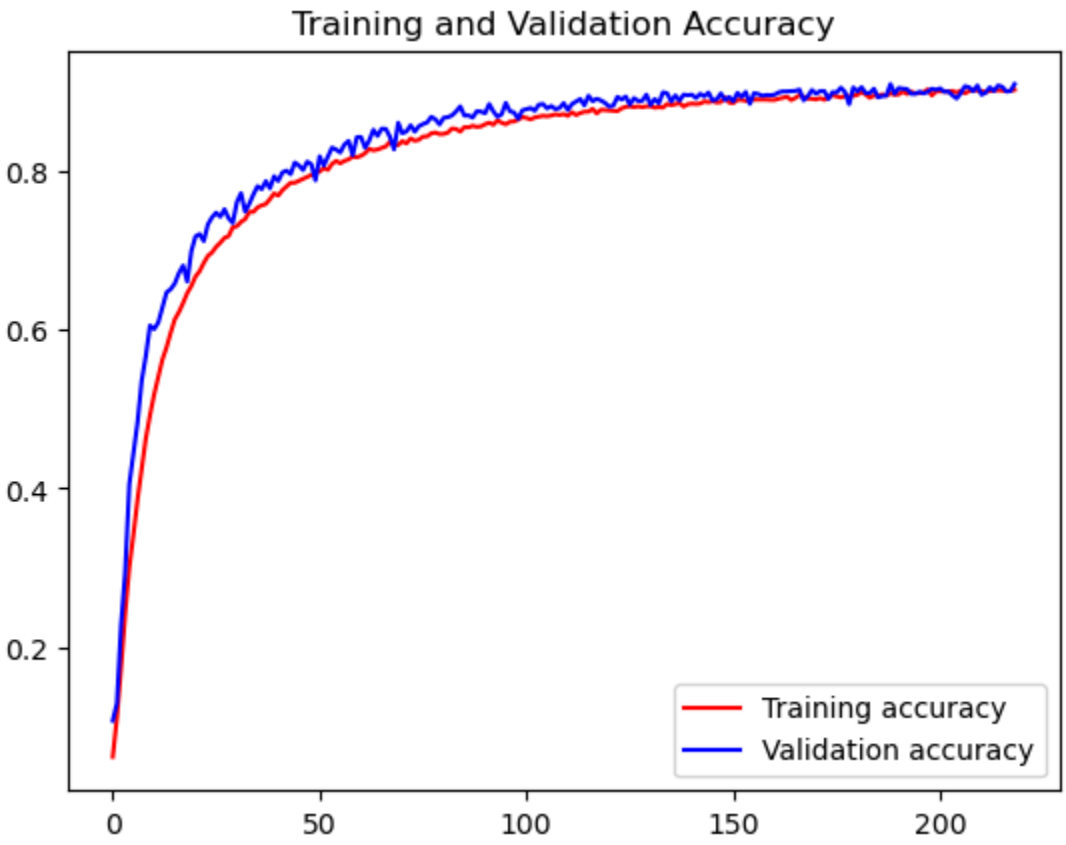
- #CONFUSION MATRIX : To further assess the performance of the trained CNN model, a confusion matrix was computed using scikit-learn's confusion_matrix function. This matrix compares the model's predicted class labels (obtained by applying np.argmax to the predicted probabilities) with the true class labels from the test set (y_true). Each row of the confusion matrix represents the actual class, while each column represents the predicted class. The diagonal elements indicate the number of correctly classified instances for each class. This matrix provides a detailed breakdown of how well the model performed across all 43 traffic sign categories, highlighting areas where the model may have struggled or misclassified certain signs.

- #PLOTTING THE CONFUSION MATRIX : To visualize the performance of the model across all classes, the confusion matrix was plotted using Seaborn's heatmap function. This heatmap provides an intuitive graphical representation of the classification results, where the x-axis represents the predicted class labels, and the y-axis shows the actual class labels. The values along the diagonal indicate correct predictions, while off-diagonal values highlight misclassifications. By setting the figure size to (20, 15) and enabling annotations, the heatmap clearly displays the count of predictions per class pair. This visual tool is especially helpful in identifying which specific traffic sign classes were more prone to confusion and which ones were recognized accurately by the model.

Results

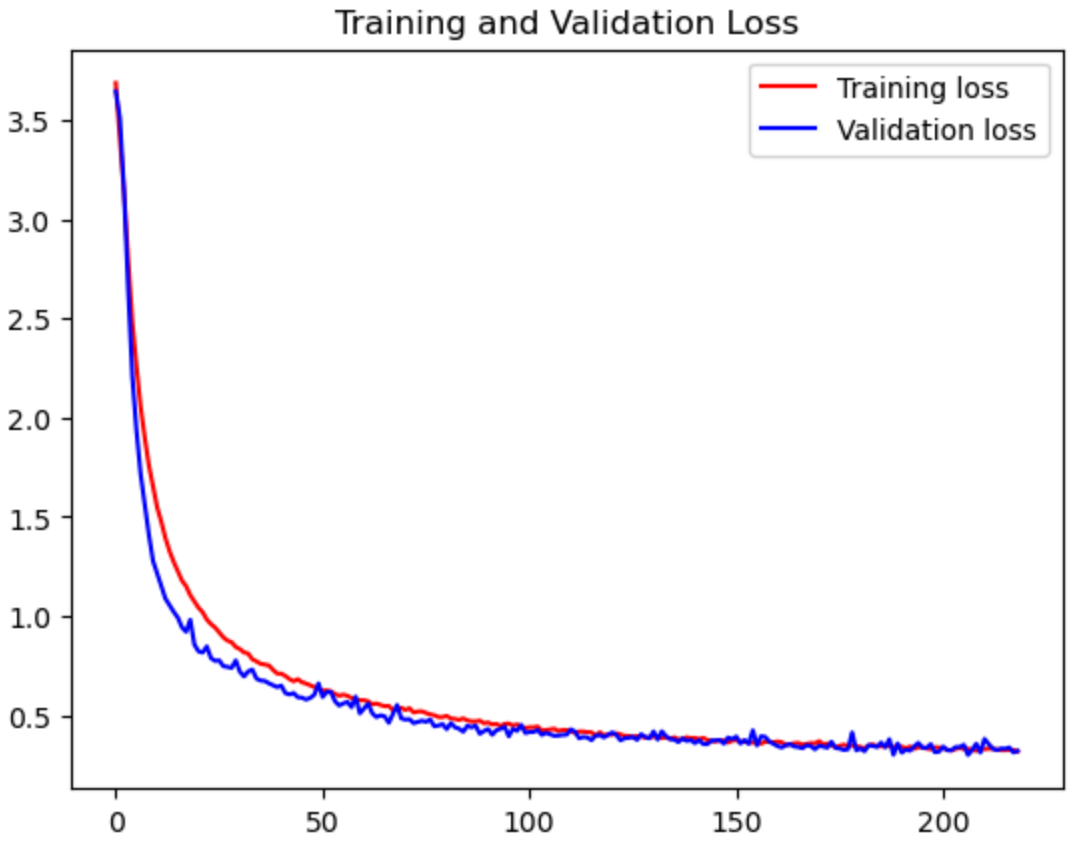
Overall

The final Convolutional Neural Network (CNN) model based on the LeNet architecture achieved a test accuracy of **88.23%** on the unseen dataset. This indicates that the model is capable of correctly classifying traffic signs across 43 categories with a high degree of reliability.



Training and Validation Accuracy

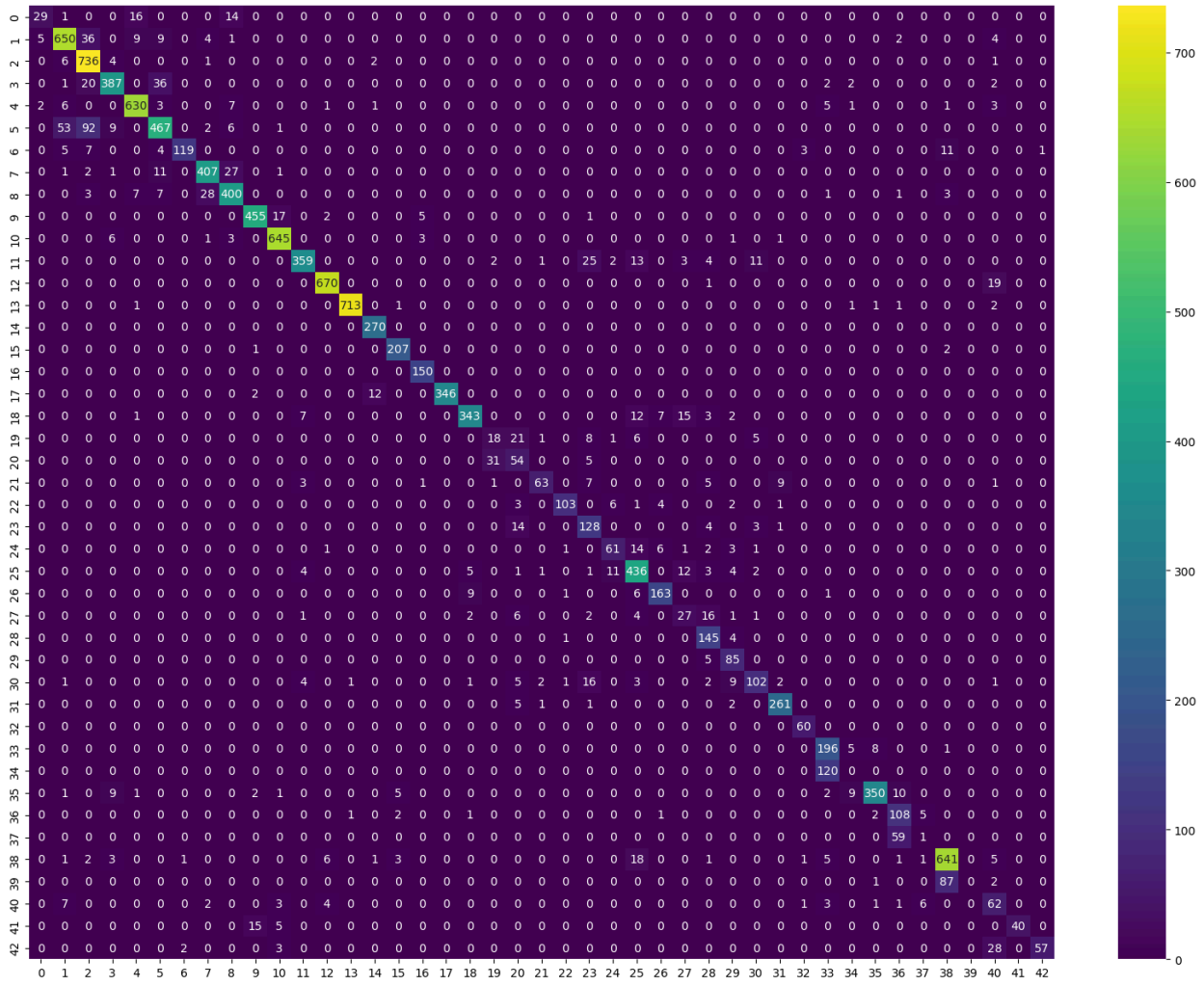
The accuracy graph shows how well the model is performing on both the training and validation datasets. The red line represents training accuracy, while the blue line indicates validation accuracy. Initially, both curves rise steeply, indicating that the model is quickly learning and improving its predictions. As the number of epochs increases, the curves gradually begin to plateau, suggesting that the model has reached a point of stable learning. By the end of training, both accuracies converge closely around 88%–89%, which is a strong indication of good generalization with minimal overfitting.



Training and Validation Loss

The loss graph tracks the error in predictions, where lower values indicate better performance. Here too, the red line represents training loss, and the blue line represents validation loss. Both losses decrease sharply during the early epochs and continue to decline steadily, indicating that the model is consistently learning to minimize errors. The close alignment of both loss curves suggests that the model is not overfitting, and it maintains consistent performance across both training and unseen validation data.

Conclusion : Together, these plots demonstrate a well-trained model. The near-convergence of training and validation metrics and the steady improvement over epochs indicate that the model is learning effectively without significant overfitting. This supports the final test accuracy of 88.23%, showing the model is robust in classifying traffic sign images across different categories.



Confusion matrix

This image represents the confusion matrix of the traffic sign classification model trained using the LeNet architecture.

- The x-axis (horizontal) represents the predicted class labels.
- The y-axis (vertical) represents the true (actual) class labels.
- Each cell (i, j) shows how many times a sample from class i was predicted as class j.

Diagonal Dominance : Most of the brighter (yellow-green) values lie on the diagonal, indicating that the majority of traffic sign images were correctly classified (true label = predicted label). This is a strong indicator of good model performance.

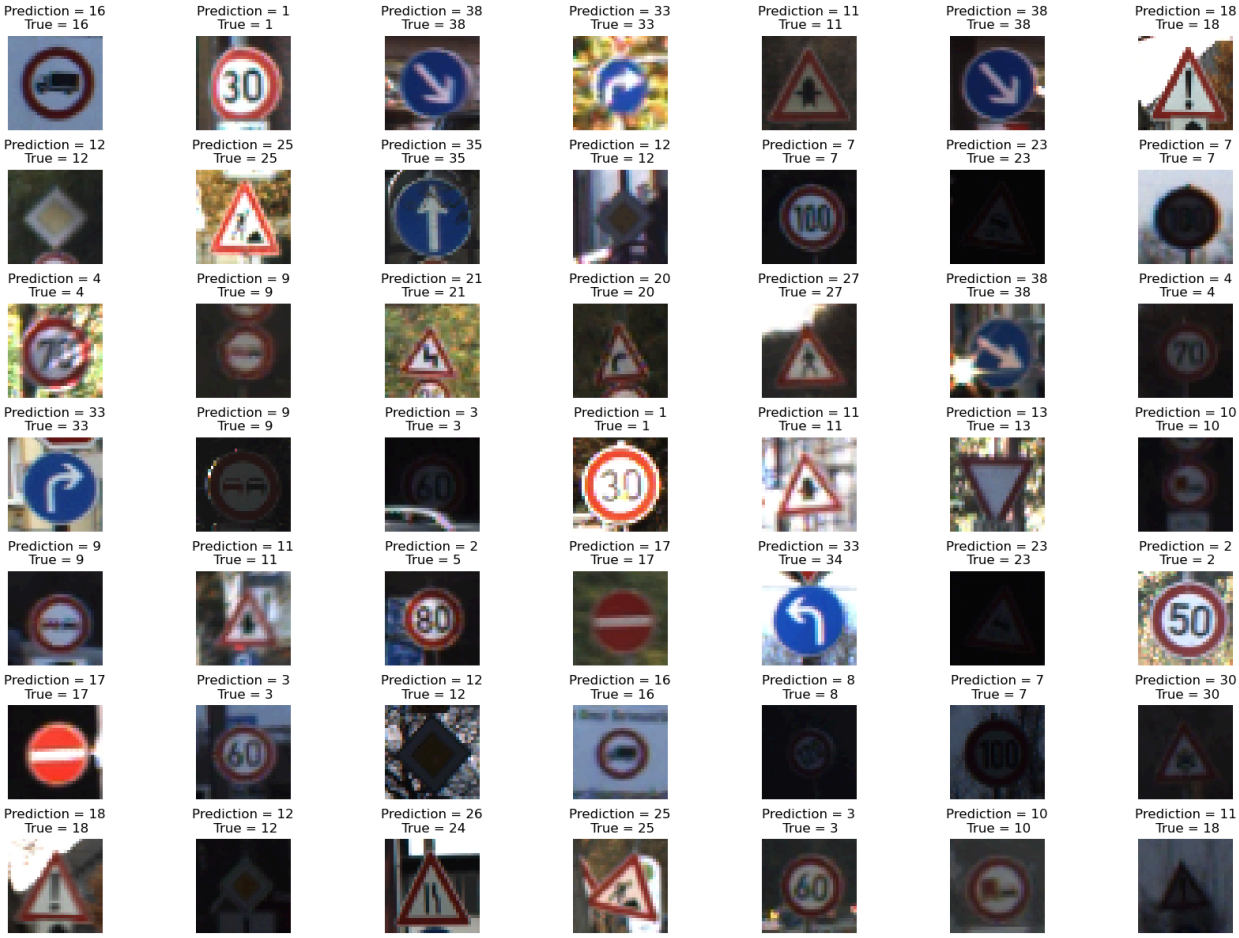
Off-Diagonal cells :

- Class 1 (Speed Limit 30 km/h) has some predictions scattered into classes 2 and 5.
- Class 5 (Speed Limit 80 km/h) has a few samples misclassified into class 2 or 3.

Key Takeaways :

- Strong diagonal: Indicates that the model is performing well, with high accuracy.
- Some confusion between similar classes: For example, speed limit signs with similar shapes or digits might be confused with each other (e.g., 60 km/h vs. 80 km/h).
- Most classes are correctly predicted: Classes like 1, 13, 14, and 38 show high counts with almost no confusion.

This matrix visually supports the model's test accuracy of 88.23%, indicating that it correctly classified a large proportion of the traffic signs, while also revealing the specific classes where improvements could be focused.



Prediction vs True Labels

References & Glossary

- GitHub Repository - Click Here
- Sequential = Creates an empty model where you will add layers one by one — making it perfect for models like CNNs that have a straight-through architecture.
    - Think of Sequential like a conveyor belt where you place one tool after another — first a scanner, then a cutter, then a packager. Each tool performs a task in order.

- Conv2D = It is one of the most important layers in a Convolutional Neural Network (CNN) used to extract features from images, like edges, curves, or textures.
    - Imagine you're sliding a small square magnifying glass over an image. That magnifying glass looks at a small part of the image (say, 3x3 or 5x5 pixels) and takes notes (calculations) about what it sees — like lines, shapes, or corners.
        - It moves (or "slides") over the image.
        - At each position, it performs a dot product between the filter and the part of the image it covers.
        - This helps detect patterns like edges or textures.

- BatchNormalization = BatchNormalization is a layer in neural networks that helps your model learn faster and more reliably by keeping the data flowing through the network well-behaved — not too big, not too small.
    - Think of BatchNormalization as a leveling tool in cooking.
    - If one spoon has too much sugar and another too little, your cake won't turn out right. BatchNormalization makes sure every spoonful is just right, every time.
        - Making sure the data at each layer has a mean around 0 and a standard deviation around 1.
        - This makes learning smoother and faster for the model.

- ReLu = Rectified Linear Unit. It's a mathematical function used in neural networks to decide which values to keep and which to ignore.
    - Keep all positive numbers as they are, and turn all negative numbers into 0.

- MaxPooling = Is a layer in CNNs used to reduce the size of the image (or feature map) while keeping the most important information.
    - It looks at small patches of an image and picks the biggest (maximum) value from each patch.
    - This helps the network focus on the strongest features (like bright edges or bold patterns) and makes computation faster.

    - Imagine looking at a group of buildings and only noting the tallest one in each block.
- Flatten = Flatten is a layer in neural networks that converts a multi-dimensional array into a 1D vector.
    - Imagine you have a 2D or 3D image (like a grid or stack of numbers), and you want to unroll it into a single line of numbers so it can be passed to the next layer (usually a dense/fully connected layer).
    - Shape: (4, 4, 3) → height x width x channels
    - Shape: (48,) → A single 1D vector with 4 × 4 × 3 = 48 values
        - Because Dense (fully connected) layers only accept 1D vectors.
        - So, Flatten is the bridge between convolutional layers and dense layers.

- Dense = A Dense layer (also called a fully connected layer) is a layer where every neuron is connected to every neuron in the previous layer.It's used in neural networks to make decisions based on the features extracted from earlier layers (like convolution and pooling).

- Dropout = Dropout is a regularization technique used in neural networks to prevent overfitting.It works by randomly turning off (or "dropping") some neurons during training.
    - If you have 100 neurons in a layer and you set Dropout(0.5):
        - During each training step, 50 of those neurons will be randomly turned off.
        - In the next step, a different 50 might be dropped.
        - Prevent overfitting

- Softmax = Softmax is an activation function used in the output layer of classification models — especially when you have multiple classes (like classifying traffic signs into 43 categories).It converts the raw scores (also called logits) from the model into probabilities that add up to 1.
    - Let me look at all the class scores and assign a percentage (probability) to each one — whichever class has the highest percentage, I'll choose that as the final prediction.
    - [2.0, 1.0, 0.1] = Output raw scores
    - [0.65, 0.24, 0.11] = Softmax will turn them into this
    - Class 0 = 65%, Class 1 = 24%, Class 2 = 11%
    - So the model picks the highest i.e Class 0 with 65% chance

- Optimizer Adam = Adam (short for Adaptive Moment Estimation) is one of the most popular and powerful optimizers used in training neural networks.Its job is to adjust the model's weights during training to help the model learn and minimize loss as efficiently as possible.
    - Think of training like navigating a maze blindfolded.Adam is like a smart friend who not only remembers the wrong turns but also adjusts your pace based on how close you are to the exit — getting you out faster and more smoothly.
    - Imagine your model is climbing down a hill (loss function) trying to reach the bottom (minimum error).
    - Adam is like a smart guide that:
        - Watches the steepness of the hill (gradients)
        - Remembers the past steps (momentum)
        - And decides how big or small a step to take each time — automatically.
        - It combines the best parts of two other optimizers:
        - Momentum (uses past gradients to smooth updates)
        - RMSProp (adapts learning rates based on recent gradient magnitudes)

- Learning Rate = The learning rate determines how much the model's weights are updated in response to the calculated error during training.
    - Too high a learning rate → the model might overshoot the optimal values and never converge (unstable training).
    - Too low a learning rate → training becomes very slow and might get stuck in a suboptimal solution (local minima).
    - Analogy = Imagine you're descending a mountain in the dark with a flashlight
        - High learning rate: You take big steps. You might reach the bottom fast but also risk missing the path or falling off a cliff.
        - Low learning rate: You take small, careful steps. It's safer but takes much longer.

- Sparse_categorical_crossentropy = It's a loss function used in multi-class classification problems, where the target (true label) is a single integer representing the class — not a one-hot encoded vector.
    - It measures how far off your model's predictions are from the true class labels.
    - It's like comparing a guess sheet (your model's prediction) to the answer key (true labels) — and scoring how well your model did, so it can learn from its mistakes.
    - Imagine your model is trying to guess what kind of traffic sign it sees.If it says, "I'm 90% sure it's a Stop sign," but the actual label was "Go", this loss function says: "Okay, that's wrong — let me calculate how bad the guess was and adjust the model to do better next time."

****