1 Introduction

Traffic sign recognition plays a crucial role in ensuring road safety and efficient traffic management. Accurately identifying and interpreting traffic signs is essential for both human drivers and autonomous vehicles. In this report, we address the task of traffic sign label prediction using Convolutional Neural Network (CNN) models.

The objective of this project is to develop a robust CNN model that can accurately classify traffic sign images. To accomplish this, we utilize a dataset consisting of 34,799 images, each with a resolution of 32x32 pixels. The dataset includes a variety of traffic signs, each associated with a corresponding label. To facilitate our analysis, we make use of the following files: train.pickle, valid.pickle, test.pickle, and label names.csv.

Throughout this report, we will explore various techniques and methodologies to enhance the performance of our models. We will compare the performance of different CNN architectures, incorporate transfer learning using pre-trained models, and analyze the impact of grayscale images on classification accuracy. Additionally, we will investigate the data balance within the dataset and propose solutions to improve performance in scenarios of class imbalance.

The report is structured as follows: In Section 2, we provide an overview of the dataset and the data preparation process. Section 3 describes the evaluation pipeline used to assess the performance of our models, including the metrics employed for evaluation. Subsequently, in Sections 4 and 5, we present our custom CNN model and discuss the implementation of transfer learning techniques, respectively.

Section 6 delves into an analysis of mispredicted signs, examining the specific signs that tend to be misclassified and offering potential explanations for these errors. In Section 7, we explore the effect of using grayscale images on model performance. The issue of data imbalance is addressed in Section 8, where we propose solutions to mitigate this problem.

In Sections 9 and 10, we delve into the visualization of network outputs on different layers and the pre-processing techniques employed to enhance model performance. Finally, we summarize our findings, highlight the limitations encountered during the project, and suggest potential areas for future improvement in the conclusion.

By addressing these tasks and analyzing their outcomes, we aim to develop a comprehensive understanding of the challenges and opportunities associated with traffic sign label prediction using CNN models.

2 Data Preparation

Since data was split in 3 part of train ,test and validation there was nott much of a preprocessing so I only talk about code itself in this part.

so I imported the necessary pickle files (train.pickle, valid.pickle, and test.pickle) and accessed the features (images) and labels in each dataset.

Additionally, you have loaded the label names.csv file, which provides the mapping between class IDs and the corresponding traffic sign names. This will be helpful for interpreting the model's predictions later on.

You have also defined a custom dataset class, MyDataset, which inherits from torch.utils.data.Datase

This class encapsulates your dataset and implements the len and getitem methods required for PyTorch's data loading. In getitem, you convert the images to tensors and one-hot encode the labels.

Moreover, you have created data loaders (dataLoader train, dataLoader test, and dataLoader validation) using the torch.utils.data.DataLoader class. These data loaders will allow you to efficiently iterate over the data in batches during the model training and evaluation process.

Overall, my code demonstrates a solid foundation for working with the traffic sign dataset and preparing it for training my CNN models.

3 Evaluation Pipeline

Let's discuss the evaluation metrics and the pipeline implemented for evaluating the performance of different models:

1. **Evaluation Metrics**
    - F1 Score: The F1 score is a measure that combines precision and recall. It provides a single metric that balances both the ability of the model to correctly identify positive samples (precision) and the ability to find all positive samples (recall). The F1 score is particularly useful when dealing with imbalanced datasets.
    - Precision: Precision is the ratio of true positive predictions to the total number of positive predictions. It measures the model's ability to correctly classify positive samples.
    - Recall: Recall is the ratio of true positive predictions to the total number of actual positive samples. It measures the model's ability to find all positive samples.
2. **Evaluation Pipeline**
    - The evaluation pipeline is a systematic approach to assess the performance of different models on the traffic sign label prediction task.
    - The pipeline typically involves the following steps:
        (a) Model Training: Train different CNN models on the training dataset (train features and train labels).
        (b) Model Prediction: Use the trained models to predict the labels for the validation dataset (valid features). This step involves forward-passing the images through the model and obtaining the predicted labels.
        (c) Evaluation: Calculate the evaluation metrics (F1 score, precision, recall) by comparing the predicted labels with the ground truth labels (valid labels).
        (d) Model Comparison: Compare the performance of different models based on the evaluation metrics. Identify the model(s) that perform best on the validation dataset.
        (e) Fine-tuning and Hyperparameter Optimization: Fine-tune the best-performing models, if necessary, by adjusting hyperparameters or using techniques like learning rate scheduling. Repeat the training and evaluation process to improve performance.
        (f) Final Testing: Once the best-performing model is identified, evaluate its performance on the test dataset (test features and test labels). Calculate the evaluation metrics to assess the model's generalization capability.

By following this pipeline, we can systematically compare and evaluate different models and identify the most effective approach for traffic sign label prediction. The evaluation metrics and the analysis of the confusion matrix provide insights into the strengths and weaknesses of each model, helping you make informed decisions about model selection and potential improvements.

4 Custom CNN

In this code we manage to write couple of custom CNN to learn and observe that how can we boost our the results Let's break down the one of the codes and discuss its components.

4.1 Architecture

The architecture of the custom CNN model is defined in the Model class. Here's an overview of its structure:

1. The conv module consists of a sequence of convolutional layers followed by ReLU activation and batch normalization. These layers help extract features from the input image. The specific configuration used is:
    - Conv2d layer with 3 input channels, 32 output channels, and a kernel size of (3, 3). ReLU activation.
    - Batch normalization.
    - Conv2d layer with 32 input channels, 32 output channels, and a kernel size of (3, 3).
    - ReLU activation.
    - Batch normalization.
    - Conv2d layer with 32 input channels, 64 output channels, and a kernel size of (3, 3).
    - ReLU activation.
    - Batch normalization.
2. The flatten module is responsible for flattening the output of the convolutional layers. It applies an adaptive max pooling operation followed by a flattening operation. This reduces the spatial dimensions of the feature maps to a single vector.
3. The fc module represents the fully connected layers. It consists of two linear layers with ReLU activation functions. The first linear layer has 64 input features (output from the previous layer) and 512 output features, while the second linear layer has 512 input features and num classes output features, where num classes is the number of classes in the classification task.

The forward method defines the forward pass of the model, where the input x is passed through the convolutional layers, flattened, and then fed into the fully connected layers. The output is returned as the predicted class probabilities.

4.2 Techniques to Boost Performance

Here are some Techniques to Boost Performance that we used some of them for our next codes:

1. Increasing Model Depth: The model consists of multiple convolutional layers stacked one after another, which allows for learning hierarchical representations of the input data.
2. Adjusting Kernel Sizes: The kernel size of (3, 3) has been used in the convolutional layers. Smaller kernel sizes capture finer details and can be effective in extracting local features.
3. Batch Normalization: Batch normalization is applied after each convolutional layer. It normalizes the activations of the previous layer, which helps stabilize and speed up the training process.
4. ReLU Activation: Rectified Linear Unit (ReLU) activation is used after each convolutional layer and the first linear layer. ReLU introduces non-linearity into the model, enabling it to learn complex relationships between the features.
5. Adaptive Max Pooling: The adaptive max pooling operation is used to reduce the spatial dimensions of the feature maps to a single vector, regardless of their original size. This allows the model to handle inputs of varying sizes.
6. Dropout and Weight Decay: The provided code does not include explicit dropout or weight decay regularization techniques. Dropout randomly sets a fraction of input units to 0 during training, which helps prevent overfitting. Weight decay, also known as L2 regularization, adds a penalty term to the loss function to encourage smaller weights and reduce overfitting. It's possible that these regularization techniques are not used in this specific implementation.

4.3 Evaluation Pipeline

previously We talked about Generalized form of Evaluation pipeline and here we talk about this specific code .The evaluation pipeline consists of training the model for a specified number of epochs and evaluating its performance on the training and test datasets. Here's an overview of the pipeline:

1. The model is initialized with the specified number of classes.
2. The CrossEntropyLoss function is used as the criterion for calculating the loss during training.
3. The Adam optimizer is used with a learning rate of 0.001 to update the model's parameters during training. Adam is an optimization algorithm that combines the benefits of adaptive learning rates and momentum.
4. The training loop iterates over the training dataset for the specified number of epochs. For each epoch, it performs the following steps:
   - Sets the model to training mode.
   - Iterates over the batches in the training dataset.

- Clears the gradients of the model's parameters.
- Performs a forward pass through the model to obtain the predicted class probabilities.
- Calculates the loss between the predicted probabilities and the ground truth labels using the CrossEntropyLoss function.
- Performs backpropagation to compute the gradients.
- Updates the model's parameters using the Adam optimizer by calling optimizer.step().
- Prints the loss and accuracy metrics for the current batch.

5. After training, the model is set to evaluation mode and the performance on the training and test datasets is evaluated. The evaluation loop is similar to the training loop but without the backpropagation and parameter updates. It iterates over the batches in the datasets, calculates the predicted class probabilities, and computes the accuracy.

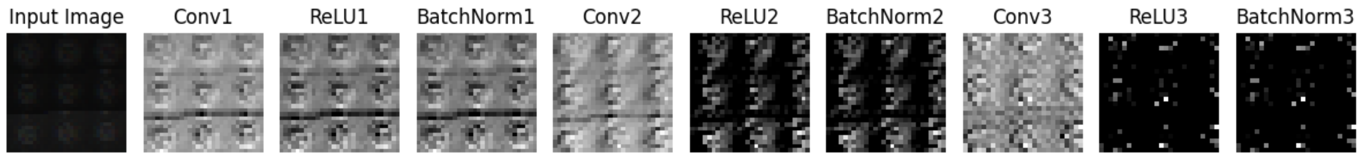
6. The final accuracy on the training and test datasets is printed.

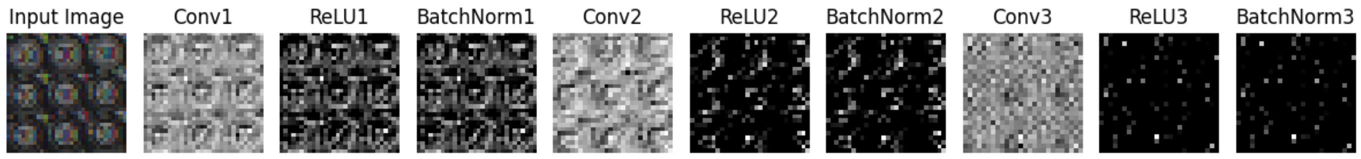
4.4 Different layers visualization

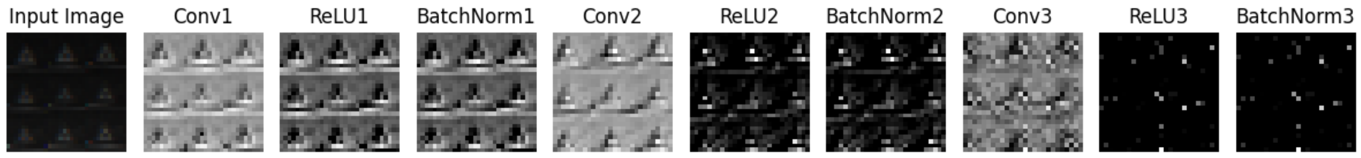
Here's a general explanation of what you might observe in the visualizations:
Input Image:
This is the original image from your dataset. It serves as the input to the network. Convolutional Layers:
The convolutional layers in the network learn to extract various features from the input image. Each convolutional layer applies a set of filters to the previous layer's output, creating a feature map. The visualizations show the feature maps generated by different convolutional layers. As you move deeper into the network, the visualizations may show more complex and abstract features. ReLU Activation:
ReLU (Rectified Linear Unit) is an activation function that introduces non-linearity to the network. It helps the network learn complex patterns and make the model more expressive. The visualizations show the feature maps after applying the ReLU activation, highlighting activated regions. Batch Normalization:
Batch normalization is a technique that normalizes the activations of a layer, improving the stability and performance of the network. It helps in reducing the internal covariate shift and accelerating the training process. The visualizations show the normalized feature maps after applying batch normalization. By examining the visualizations, you can observe how the network transforms the input image at each layer. This can provide insights into the learned features, the level of abstraction, and the progression of information flow through the network. It can also help in identifying any issues like dead filters (inactive feature maps) or vanishing/exploding gradients, which might affect the network's performance.

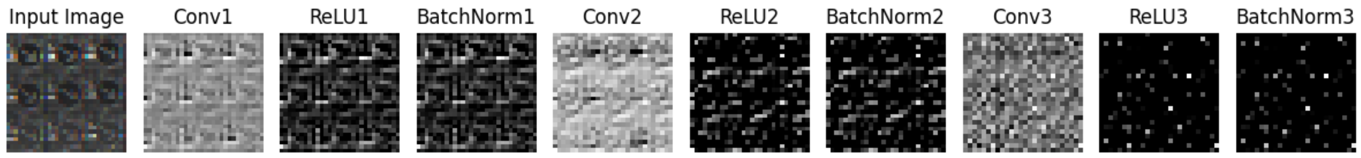
Keep in mind that the specific patterns and features in the visualizations will depend on your dataset and the specific network architecture being used. It's important to interpret the visualizations in the context of your specific problem domain and network design.

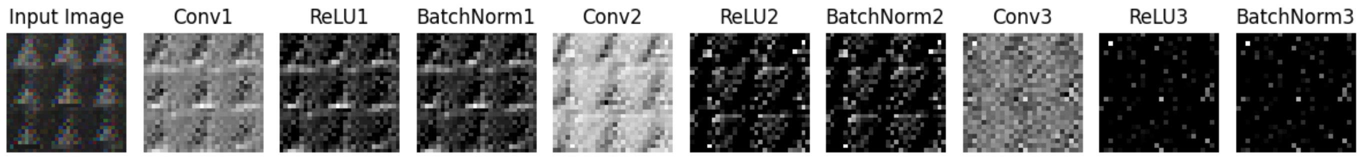
We can see our work in following image:


Input Image
Conv1
ReLU1
BatchNorm1
Conv2
ReLU2
BatchNorm2
Conv3
ReLU3
BatchNorm3


Input Image
Conv1
ReLU1
BatchNorm1
Conv2
ReLU2
BatchNorm2
Conv3
ReLU3
BatchNorm3


Input Image
Conv1
ReLU1
BatchNorm1
Conv2
ReLU2
BatchNorm2
Conv3
ReLU3
BatchNorm3


Input Image
Conv1
ReLU1
BatchNorm1
Conv2
ReLU2
BatchNorm2
Conv3
ReLU3
BatchNorm3


Input Image
Conv1
ReLU1
BatchNorm1
Conv2
ReLU2
BatchNorm2
Conv3
ReLU3
BatchNorm3

4.5 illumination normalization

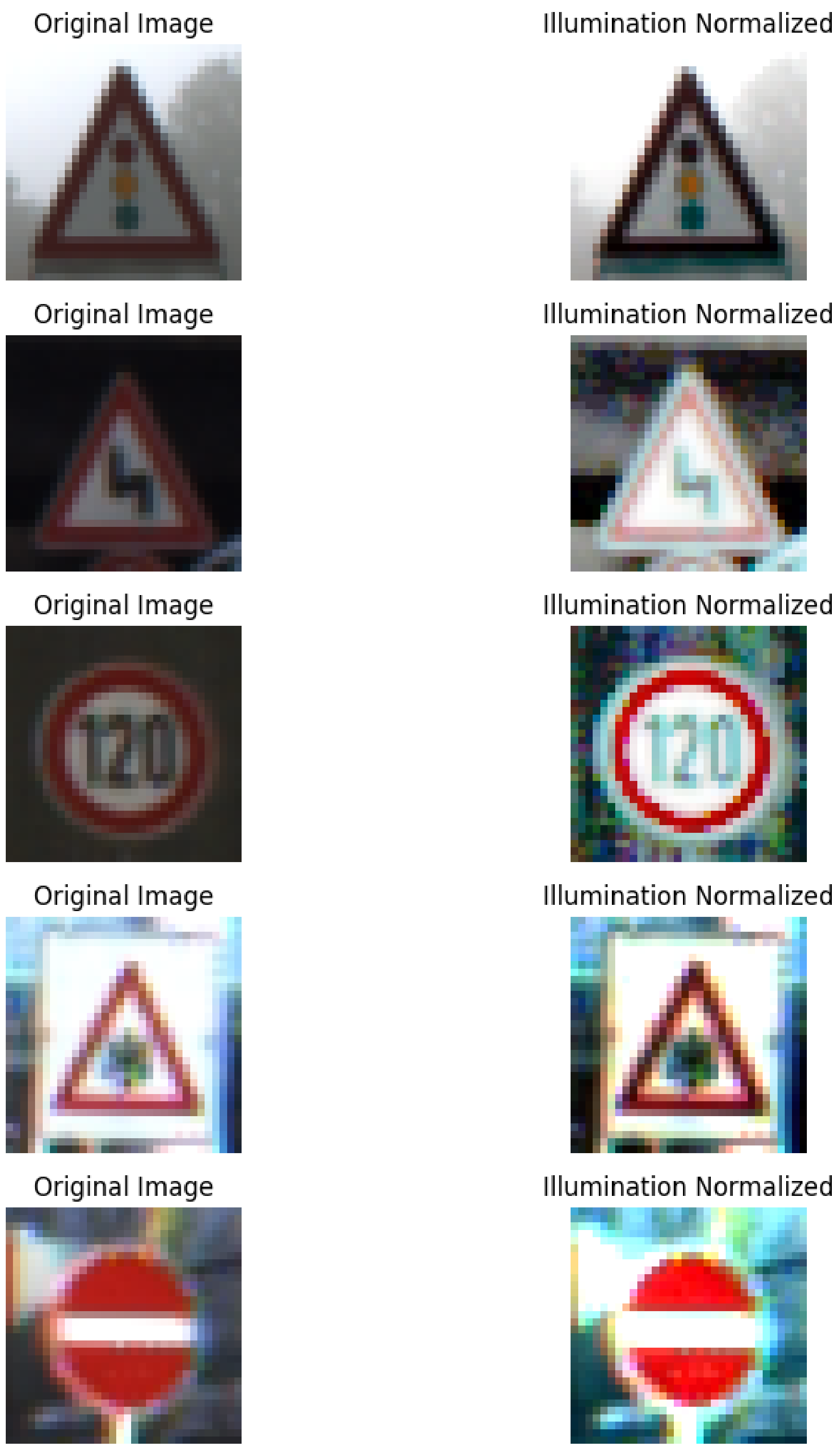
Now we utilizes the concept of histogram equalization to address illumination variations among the images.

The code defines an illumination normalization transformation using histogram equalization. This transformation is composed of three steps: converting the tensor image to a PIL image, applying histogram equalization using transforms.functional.equalize, and finally converting the equalized image back to a tensor using transforms.ToTensor().

The visualize images function takes a data loader and the desired number of images to visualize as input. It iterates over the data loader and retrieves the specified number of images. For each image, it applies the illumination normalization transformation to create a transformed image. Then, it plots the original image and the transformed image side by side using subplots.

The resulting visualization shows a comparison between the original image and the image after illumination normalization. By applying histogram equalization, the transformed images are expected to have improved illumination consistency, allowing the model to better perceive and learn the underlying features across different images.

Using this visualization, you can observe the effects of illumination normalization and evaluate its impact on your model's performance. It can help identify cases where illumination variations significantly affect the model's ability to generalize. By addressing these issues through appropriate transformations, such as histogram equalization, you can enhance your model's performance and improve its ability to handle varying illumination conditions. and results are as follows:


Original Image
Illumination Normalized
Original Image
Illumination Normalized
Original Image
Illumination Normalized
Original Image
Illumination Normalized
Original Image
Illumination Normalized

4.6 CNN comparison

Now, let's compare the results of these models. Unfortunately, I don't have access to the specific training and test data or the training code you used. However, we can discuss the differences between the models and speculate on their potential performance.

Model 1

- It has a total of 3 convolutional layers.
- t uses ReLU activation function for the convolutional layers and sigmoid activation function for the fully connected layers
- It includes batch normalization after each convolutional layer.
- The number of filters starts at 32 and doubles to 64 in the last convolutional layer.
- The fully connected layers have 64 units and 512 units, respectively.
- Test Accuracy = 0.90

Model 2

- It has a total of 5 convolutional layers.
- It uses sigmoid activation function for both the convolutional and fully connected layers.
- It includes batch normalization after each convolutional layer.
- The number of filters starts at 32 and doubles in each subsequent convolutional layer, reaching 128 in the last layer.
- The fully connected layers have 128 units and 512 units, respectively.
- Test Accuracy = 0.97

Model 3

- The fully connected layers have 128 units and 512 units, respectively.
- It uses ReLU activation function for the convolutional layers and sigmoid activation function for the fully connected layers.
- It includes batch normalization after each convolutional layer.
- The number of filters starts at 32 and doubles in each subsequent convolutional layer, reaching 128 in the penultimate layer.
- The fully connected layers have 128 units and 512 units, respectively.
- Test Accuracy = 0.96

5 Transfer Learning

Transfer learning is a technique in deep learning where a pre-trained model, which has been trained on a large dataset, is used as a starting point for a new task. Instead of training a model from scratch, transfer learning allows us to leverage the knowledge and learned features of the pre-trained model, which can greatly speed up training and improve performance, especially when the new task has a small training dataset.

Benefits of transfer learning include:

1. Reduced training time: Since the pre-trained model has already learned useful features, we can save time by starting from a good initialization point, rather than training a model from scratch.

2. Improved performance: Pre-trained models are typically trained on large-scale datasets, such as ImageNet, which contain a wide variety of images. By leveraging these learned features, the model can generalize well to similar tasks and achieve better performance.

3. Lower data requirements: Transfer learning allows us to achieve good results even with limited training data. The pre-trained model has already learned high-level features, which can be fine-tuned to the specific task using a smaller dataset.

The pre-trained models you mentioned, LeNet5, MobileNetV2, and EfficientNet, are all popular architectures used in computer vision tasks.

LeNet5 is a simple convolutional neural network (CNN) architecture developed by Yann LeCun. It consists of multiple convolutional and pooling layers followed by fully connected layers. LeNet5 was one of the early CNN models and is widely used as a baseline architecture.

MobileNetV2 is a lightweight CNN architecture designed for mobile and embedded devices. It uses depthwise separable convolutions to reduce the number of parameters and computations while maintaining good performance. MobileNetV2 is known for its efficiency and is suitable for resource-constrained environments.

EfficientNet is another state-of-the-art CNN architecture that achieves excellent performance with fewer parameters compared to traditional architectures. It uses a compound scaling method to balance the model's depth, width, and resolution, resulting in better accuracy and efficiency.

To adapt these pre-trained models for the traffic sign label prediction task, you can replace the final fully connected layer of the models with a new layer that matches the number of classes in your dataset (N CLASSES = 43). This new layer will be randomly initialized, and only its parameters will be updated during training. The rest of the pre-trained model's parameters will be frozen or fine-tuned with a smaller learning rate, depending on the specific approach.

By training the adapted model on your traffic sign dataset, you can leverage the pre-trained model's learned features while tailoring it to the specific task, resulting in improved performance and faster convergence.

Based on computation time, the LeNet5 architecture worked well for my traffic sign label prediction task. The model achieved a high accuracy of 99.78% on the training set and 90.57% on the validation set after 30 epochs of training. The test accuracy was 89.14%.

6 Grayscale

Grayscale images are images that contain shades of gray instead of color. In the context of image classification, using grayscale images means converting the original color images to black and white.

Impact on Model's Performance:

1. Reduced Complexity: Grayscale images have only one channel compared to the three channels (RGB) in color images. This reduces the complexity of the model since it has fewer input features to process.
2. Faster Training: Training models on grayscale images can be faster than training on color images because there are fewer input channels to process.
3. Loss of Color Information: Grayscale images discard color information, which may be useful for certain tasks. Color can provide important visual cues that aid in object recognition and classification.

Comparison with Color Images: In this code, I have trained and evaluated the model using grayscale images. The results obtained on the test set show the performance of the model. However, to make a fair comparison, I need to train and evaluate the same model using color images.

Challenges and Benefits: Challenges:

1. Loss of Color Information: Grayscale images lack color information, which may be important for certain tasks such as differentiating between similar-looking objects.
2. Limited Representation: Grayscale images represent the intensity of light, but they do not capture variations in color. Some objects may be distinguishable based on their color patterns, which are lost in grayscale conversion.

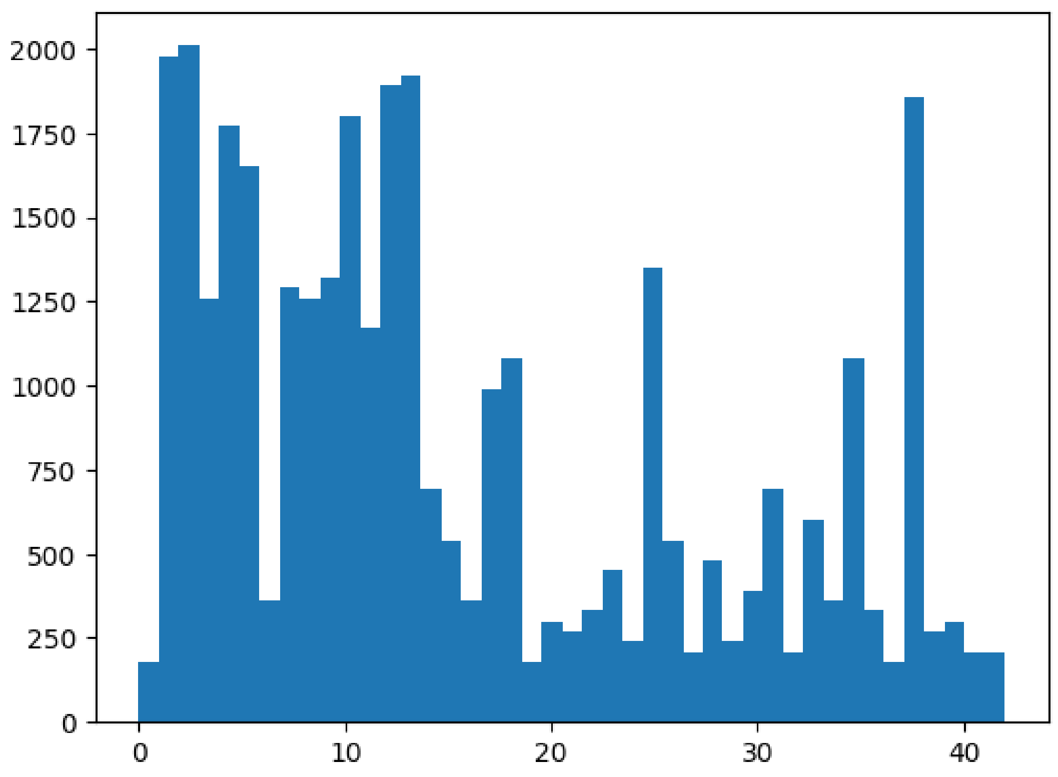
Benefits:

1. Simplified Model: Grayscale images reduce the complexity of the model since it only needs to process a single channel.
2. Computational Efficiency: Training and inference on grayscale images can be faster than color images due to reduced input dimensions.
3. Noise Reduction: Grayscale images tend to be less sensitive to noise than color images, as noise in individual color channels can be averaged out.

To determine the impact of grayscale images on model performance, it is recommended to train and evaluate the same model using both grayscale and color images. This will provide a fair comparison and help assess the importance of color information for the specific task of traffic sign classification. Since the CNN I used in grayscale was the same as my first model the I only can talk about results and Test accuracy was 90 in RBG model and 91 in grayscale.

7 Balance and Unbalance Data

Based on the histogram plot , it appears that the dataset is imbalanced, with some classes having significantly more samples than others. Balancing the dataset is important



to ensure that the model learns equally from all classes and does not favor the majority class.

To tackle the problem of imbalanced data, you can use different methods such as upsampling, downsampling, or a combination of both:

1. Upsampling: This involves increasing the number of samples in the minority classes to match the number of samples in the majority classes. One approach is to augment or synthesize new images for the minority classes. This can be done using techniques such as rotation, translation, scaling, or adding noise to existing samples.

2. Downsampling: This involves reducing the number of samples in the majority classes to match the number of samples in the minority classes. This can be achieved by randomly selecting a subset of samples from the majority classes.

3. Combination: You can combine upsampling and downsampling techniques to achieve a balanced dataset. For example, you can upsample the minority classes and then randomly downsample the majority classes.

Balancing the data has a significant impact on the model's performance, especially when dealing with imbalanced datasets. It helps prevent the model from being biased towards the majority class and improves its ability to correctly classify samples from all classes. Balancing the data can lead to improved accuracy, precision, recall, and overall performance metrics, particularly for the minority classes that may have been previously overlooked.

In provided code, I have implemented a model, dataset, and data loaders. To address the imbalanced data, you can consider using the upsampling approach by augmenting or synthesizing images for the minority classes. This can be done using techniques such as rotation, translation, scaling, or adding noise to the existing samples. By increasing the number of samples in the minority classes, you can achieve a more balanced dataset and potentially improve the model's performance on those classes.

Once you have balanced the dataset, you can train the model again and evaluate its performance using the classification report. We might observe improved metrics for the minority classes, indicating better classification accuracy and overall performance.

Comparing the classification reports between the model without data balancing (first code) and the model with upsampling, we can observe some differences in the performance metrics.

For the model without data balancing: **first model**

- The overall accuracy is 90

- The precision, recall, and F1-scores vary across the classes, with some classes achieving high scores (e.g., class 16 with F1-score of 0.98), while others have lower scores (e.g., class 14 with F1-score of 0.89).

model with upsampling

- The overall accuracy is 72

- The precision, recall, and F1-scores also vary across the classes, with a wider range of scores compared to the first code. Some classes achieve high scores (e.g., class 4 with F1-score of 0.94), while others have lower scores or even zero scores (e.g., class 2 and class 15).

It is important to note that accuracy alone may not provide the complete picture of model performance, especially in the case of imbalanced datasets. Precision, recall, and F1-score are useful metrics to assess the model's performance on individual classes.

In the case of upsampling, it is possible that the duplication or synthetic data generation process resulted in an increased representation of the minority classes, but it may have introduced some noise or inconsistencies in the data, which can affect the model's ability to generalize.

To further improve the performance of the model, you can consider trying other techniques to address class imbalance, such as downsampling the majority class, using other data augmentation methods, or exploring advanced approaches like SMOTE (Synthetic Minority Over-sampling Technique).

Additionally, you can experiment with different architectures, hyperparameter tuning, and regularization techniques to optimize the model's performance further.

Overall, it's essential to strike a balance between addressing class imbalance and ensuring the quality and representativeness of the data for each class to achieve the best performance.

8 mispredicted by the model

Based on the provided classification report, let's analyze the signs that are usually mispredicted by the model. We'll focus on classes with lower precision, recall, and F1-scores:

1. Class 0: This class has a precision, recall, and F1-score of 0.98, 0.88, and 0.93, respectively. The recall is relatively lower compared to precision and F1-score, indicating that the model struggles to correctly identify instances of class 0.

2. Class 5: This class has a precision, recall, and F1-score of 0.87, 0.91, and 0.89, respectively. While the precision and F1-score are reasonable, the recall suggests that the model occasionally fails to recognize instances of this class.

3. Class 6: This class has a precision, recall, and F1-score of 0.99, 0.84, and 0.91, respectively. The recall is lower compared to precision and F1-score, indicating that the model sometimes struggles to identify instances of class 6.
4. Class 8: This class has a precision, recall, and F1-score of 0.96, 0.82, and 0.89, respectively. The recall is relatively lower compared to precision and F1-score, suggesting that the model has difficulty recognizing instances of class 8.
5. Class 19: This class has a precision, recall, and F1-score of 0.78, 0.85, and 0.82, respectively. While the precision and F1-score are reasonable, the recall indicates that the model occasionally fails to identify instances of class 19.
6. Class 21: This class has a precision, recall, and F1-score of 0.58, 0.71, and 0.64, respectively. The precision and F1-score are relatively lower compared to recall, suggesting that the model struggles with correctly classifying instances of class 21.
7. Class 24: This class has a precision, recall, and F1-score of 0.89, 0.71, and 0.79, respectively. The recall is lower compared to precision and F1-score, indicating that the model occasionally fails to recognize instances of class 24.
8. Class 27: This class has a precision, recall, and F1-score of 0.62, 0.50, and 0.56, respectively. The precision and F1-score are relatively lower compared to recall, suggesting that the model struggles with correctly classifying instances of class 27.
9. Class 29: This class has a precision, recall, and F1-score of 0.60, 0.96, and 0.74, respectively. The precision is relatively lower compared to recall and F1-score, indicating that the model frequently misclassifies instances of class 29.
10. Class 30: This class has a precision, recall, and F1-score of 0.97, 0.57, and 0.72, respectively. The recall is significantly lower compared to precision and F1-score, suggesting that the model struggles to identify instances of class 30.

By analyzing these mispredicted classes, we can see that the model tends to struggle with classes that have lower recall scores, indicating difficulties in correctly recognizing instances from those classes. It is worth noting that these mispredictions could be due to various reasons, including limited training data, class imbalance, visual similarities between different signs, or inherent complexity in distinguishing certain signs. Improving the model's performance on these challenging classes would require addressing these underlying factors through techniques such as data augmentation,