Lightweight CNN for Image Classification with Preprocessing, Augmentation, and Feature Visualization

COSC-6324 Digital Image Processing

Team Members:

Bala Nithin Chennarapu(A04342051)
Prathyusha Pentam(A04342934)
Madhusudhan Poduturu(A04346304)

Faculty Guide: Dr. Minhua Huang

Contents

Team Members and Contributions

Madhusudhan poduturu

- Led dataset construction and preprocessing, including CIFAR-10 subset selection, stratified splitting, and directory organization.
- Implemented the CLAHE-based histogram equalization pipeline and integrated it into the data preprocessing functions.

Bala Nithin Chennarapu

- Designed and implemented the lightweight CNN architecture, including convolutional blocks, batch normalization, dropout,perfomed the epoch training and featuremaps.
- Configured the training procedure, callbacks (early stopping, learning rate reduction, model checkpointing), and monitored training/validation curves.And contributed in writing report.

Prathyusha Pentam

- Performed model evaluation and analysis: computed test metrics, generated the confusion matrix, and interpreted per-class performance.
- Developed the feature map visualization pipeline and contributed to the writing of the results, analysis, and contributed in writing the report.

1 Introduction

1.1 Background

Convolutional Neural Network have now are the standard approach for the image classification tasks, especially on benchmark datasets such as CIFAR-10. The models now learn very rich features directly from raw pixel data and achieve high accuracy when large amounts of labeled data and powerful hardware are available

However, the conditions in real-world and classroom environments are often very different from those in large-scale research settings. Instead of training very deep networks on millions of images, we commonly work with smaller datasets, limited compute budgets, and images collected under suboptimal conditions. This gap between "ideal" benchmark scenarios and realistic constraints motivates the need for pipelines that explicitly address data quality, model complexity, and interpretability. Hence this project focuses on developing a (CNN) for classifying grayscale images derived from 32×32 RGB images. The collection consists of 10,000 annotated images distributed among 10 object classes.

1.2 Challenges

CIFAR-10 itself is already a challenging dataset because as it consists of low-resolution (32×32) color images from ten everyday object categories. Objects are small, backgrounds are complex, and visual cues can be subtle. When we further restrict the training data to a subset of 10,000 images, several challenges become more pronounced: the model can easily memorize the training set instead of generalizing, noisy or low-contrast images can confuse the classifier, and class-specific patterns may be poorly represented here.

Small datasets and low-resolution images lead to a higher risk of overfitting and unstable performance. In addition, varying lighting, contrast, and background clutter make it harder for a network to extract stable features that generalize well to unseen samples. At the same time, in a resource-constrained environment, it is not practical to rely on very deep architectures or exhaustive hyperparameter searches. The key challenge is therefore to obtain reasonable classification performance under these constraints while keeping the model lightweight and training time manageable for us in this project.

1.3 Motivation

The motivation behind this project is to develop a model that closely mimics a realistic workflow for small-scale image classification. Starting from raw CIFAR-10 images, we first aim to enhance them using classical image processing techniques, then increase the effective dataset size using on-the-fly augmentation, and finally train a compact CNN model that is easy to understand, implement, and visualize.

By visualizing feature maps from intermediate convolutional layers, we can examine whether the network is focusing on meaningful structures such as edges, textures, and object parts. This interpretability aspect is especially important in this project, where understanding what the network learns is as valuable as the final performance metrics of what we have achieved.

1.4 Objectives

Based on this motivation, the objectives of the project are:

1. To design and implement a lightweight CNN architecture with a relatively small number of parameters that is with a dataset of 10000 images, so that it trains quickly while still being good enough to handle CIFAR-10-style images.

2. To integrate a preprocessing step using contrast-limited adaptive histogram equalization (CLAHE) and normalization, with the goal of improving image contrast and dynamic range so that important structures are more visible.

3. To apply data augmentation techniques such as random horizontal flips, small rotations, translations, and zooms, therefore increasing the effect of the training data and reducing overfitting.

4. To systematically verify our the trained data model on a held-out test using metrics like known as accuracy, precision, recall, and F1, and to generate visualizations of intermediate feature maps in order to analyze what the proposed CNN model learns at different depths.

1.5 Novelty

These lies in a way these components are combined into a coherent, lightweight, and interpretable pipeline. Instead of relying solely on raw CIFAR-10 images and a standard off-the-shelf CNN, we explicitly incorporate CLAHE-based contrast enhancement, structured data augmentation, and a small-parameter CNN architecture that is refined for fast training on a limited subset of data.

On top of that, we emphasize interpretability by extracting and visualizing feature maps from we added multiple convolutional layers, showing the progression from simple low-level edges and textures to more abstract, class-related patterns. This combination of image processing, efficient model design, and feature-level visualization turns the project into more than just a straightforward classification of a compact and extensible framework for experimenting with CNNs on small or noisy image datasets.

2 Methodology

2.1 Dataset construction and preprocessing

The further experiments in this project are based on the CIFAR-10 images dataset, which has 60,000 RGB imgs of size 32×32 pixels across 10 image categories (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). Here we Used TensorFlow loader `tf.keras.datasets.cifar10`, and all the 50,000 training images dataset and 10,000 test img dataset are first combined into a single folder.

From this folder , we selected a new subset of 10,000 images with 10 classes in it(airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). Stratified sampling is used so that the relative frequency of each of the 10 classes remains balanced in the subset. This choice is made to keep the realistic "small data" scenario and to work with small or noisy datasets instead of the full CIFAR-10.

The 10,000 images are then split into three sets:

- Training Dataset: 7,000 images
- Validation Dataset: 1,500 images
- Test Dataset: 1,500 images

Again, we used splitting so that all three splits preserve the class distribution. To make the data easy to load later, the images are written to disk in a directory structure such as:

- `cifar10_10000/train/<class_name>/...`
- `cifar10_10000/val/<class_name>/...`
- `cifar10_10000/test/<class_name>/...`

Each image is saved as an 8-bit RGB file using PIL. This folder layout allows us the project to use to automatically create TensorFlow datasets (`train_raw`, `val_raw`, `test_raw`) with images and categorical labels.

Then for us to ensure reproducibility, random seeds are fixed at the beginning of the colab notebook for Python's `random`, NumPy, and TensorFlow. Environment variables such as `PYTHONHASHSEED` are also set, so that shuffling, splitting, and batch sampling behave as accordingly across the further runs.

Once the raw datasets are created, the project defines a preprocessing pipeline that is applied through the TensorFlow `tf.data` pipeline. here we did the main preprocessing step is contrast enhancement using CLAHE:

1. here we treated each image as an RGB NumPy array of type `uint8`.

2. The image is then converted from the RGB color using OpenCV (`cv2.cvtColor`). In LAB, the L channel encodes lightness (brightness), while the A and B channels carry color information.

3. The L channel is separated and passed through OpenCV's CLAHE function with:
   - `clipLimit = 2.0`
   - `tileGridSize = (8, 8)`

4. Here the CLAHE locally equalizes the histogram of the L channel, enhancing contrast in small regions while preventing extreme things of noise.

Now because CLAHE is implemented using OpenCV and NumPy, it is wrapped in a TensorFlow-compatible function `histogram_equalization_tf` using `tf.numpy_function`. This allows the CLAHE step to be done inside the `tf.data` pipeline and applied to each image in a batch. The image shape is again reset to $(32, 32, 3)$ after the operation.

Here we applied two preprocessing functions for the dataset:

- `prepare_train(images, labels)`:
  - Here we applied CLAHE to each image in the dataset using `tf.map_fn` and applied Histogram Equalization.
  - Then passed the equalized images through the data augmentation pipeline.
  - And applied a normalization layer (`Rescaling(1.0 / 255.0)`) to map pixel values from [0, 255] to [0, 1].

- `prepare_eval(images, labels)`:
  - Here also applies CLAHE in the same way as for the training dataset images.
  - Applies only to the normalization layer (no augmentation).

Now the raw datasets (`train_raw`, `val_raw`, `test_raw`) are mapped through the appropriate preprocessing function and then optimized. As a result, the model always receives images that have consistent preprocessing: contrast-enhanced and normalized, with augmentation only in the training pipeline for further.

2.2 Data augmentation

In our Project to address the 7,000-image training set and to simulate a larger, more diverse dataset, we used a data augmentation pipeline built with Keras Sequential augmentation layers. This augmentation is applied only to the training data and includes:

- `RandomFlip("horizontal")`
  Random horizontal flips that simulate the left–right variations.

- `RandomRotation(0.1)`
  And did Random rotations up to ±10% of a full turn (approximately ±18°) inorder to help the model become slightly rotation-invariant.

- `RandomTranslation(0.1, 0.1)`
  Small horizontal and vertical translations (up to 10% of the image size) to simulate the minor shifts in object position within the frame.

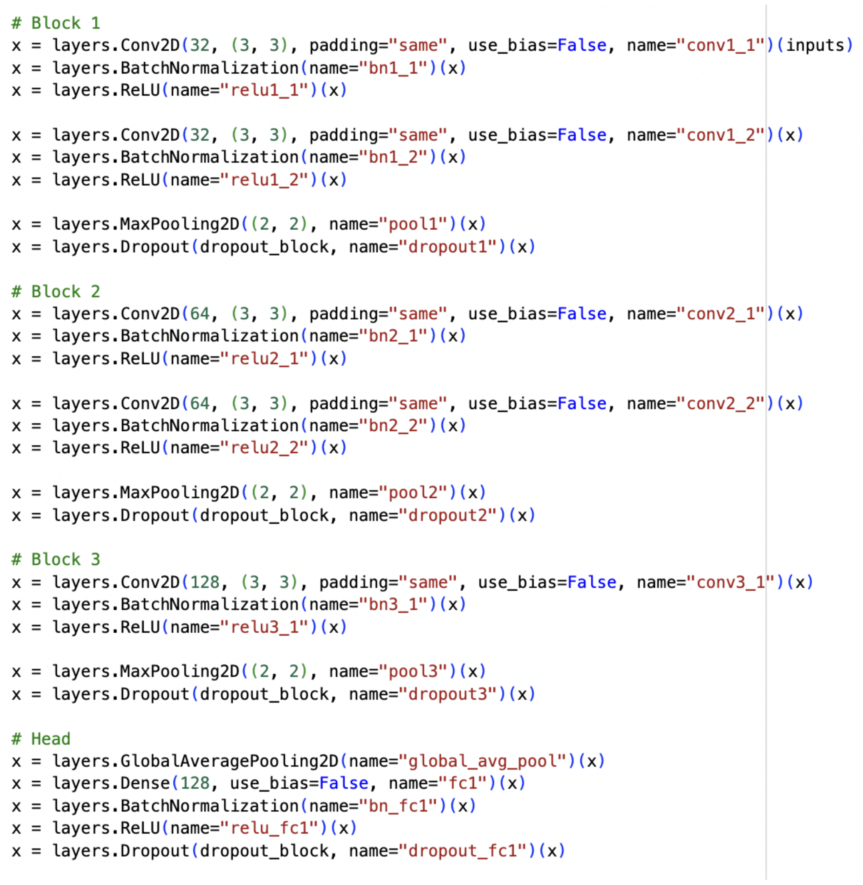
- `RandomZoom(0.1)`
  And did small random zooms to encourage changes in object scale and distance.

We applied these transformations during training, so each epoch sees slightly different versions of the same underlying images. This effectively increases the data diversity, making it harder for the model to memorize exact images and encouraging it to learn more generalizable features.

2.3 CNN architecture design

The classification model we took is a lightweight CNN defined in the function `build_cnn_model`. This is intentionally small as to satisfy the project requirement of being fast and efficient while still capable of learning useful features from 32×32 images.

Here the network takes an input image of shape $(32, 32, 3)$ and we kept three main convolutional blocks and by a small classification head:



```
# Block 1
x = layers.Conv2D(32, (3, 3), padding="same", use_bias=False, name="conv1_1")(inputs)
x = layers.BatchNormalization(name="bn1_1")(x)
x = layers.ReLU(name="relu1_1")(x)

x = layers.Conv2D(32, (3, 3), padding="same", use_bias=False, name="conv1_2")(x)
x = layers.BatchNormalization(name="bn1_2")(x)
x = layers.ReLU(name="relu1_2")(x)

x = layers.MaxPooling2D((2, 2), name="pool1")(x)
x = layers.Dropout(dropout_block, name="dropout1")(x)

# Block 2
x = layers.Conv2D(64, (3, 3), padding="same", use_bias=False, name="conv2_1")(x)
x = layers.BatchNormalization(name="bn2_1")(x)
x = layers.ReLU(name="relu2_1")(x)

x = layers.Conv2D(64, (3, 3), padding="same", use_bias=False, name="conv2_2")(x)
x = layers.BatchNormalization(name="bn2_2")(x)
x = layers.ReLU(name="relu2_2")(x)

x = layers.MaxPooling2D((2, 2), name="pool2")(x)
x = layers.Dropout(dropout_block, name="dropout2")(x)

# Block 3
x = layers.Conv2D(128, (3, 3), padding="same", use_bias=False, name="conv3_1")(x)
x = layers.BatchNormalization(name="bn3_1")(x)
x = layers.ReLU(name="relu3_1")(x)

x = layers.MaxPooling2D((2, 2), name="pool3")(x)
x = layers.Dropout(dropout_block, name="dropout3")(x)

# Head
x = layers.GlobalAveragePooling2D(name="global_avg_pool")(x)
x = layers.Dense(128, use_bias=False, name="fc1")(x)
x = layers.BatchNormalization(name="bn_fc1")(x)
x = layers.ReLU(name="relu_fc1")(x)
x = layers.Dropout(dropout_block, name="dropout_fc1")(x)
```

Figure 1: Convolution Blocks in the Dataset.

Overall, our model has 158,570 parameters (about 619 KB), of which 157,674 are trainable. This confirms that the network is relatively small compared to typical deep CNNs, which helps keeps us the training times low(as we are computing this in a local PC) and reduces us the risk of severe overfitting on a 7,000-image training set.

2.4 Training details

Here, we did our model compilation with:

- Optimizer: Adam, with learning rate `1e-3`.
- Metric: we calculated accuracy across the training and val of our dataset.

We performed our training of our dataset for a maximum of 30 epochs (`EPOCHS = 30`), but we used early stopping so that training stops automatically when the validation loss stops improving. Specifically,we configured the following callbacks:

1. **EarlyStopping**

Here if the val did not change for 5 continous epochs, the training is stopped and the models are gone back to the epoch with the best val loss.

2. **ModelCheckpoint**

   - `filepath = "best_lightweight_cnn.keras"`

   The best data model (according to validation loss) is saved to disk during training.

3. **ReduceLossOnPlateau**

   Here if val loss does not improve for 2 epochs, the learning is decreased by half, down to a minimum of `1e-5`. This shows the Adam optimizer take smaller steps when the training starts to plateau.

During our training, both training accuracy/loss and val accuracy/loss are logged in the history object. After training, we used a helper function to visualize these curves, showing how the model's performance evolved over epochs and whether overfitting occurred.

2.5 Evaluation consideration

For our final evaluation, the trained model is applied to the held-out, which contains 1,500 examples which are not seen during training or validation. Then we did our evaluation in two steps:

1. **Raw accuracy and loss**
   Here the model's overall test accuracy and test loss are computed using standard Keras evaluation methods.

2. **Detailed metrics with scikit-learn**
   To obtain a deeper understanding of our performance across classes, we collected:

   - The true labels (`y_true`) by taking the argmax over the one-hot label vectors.
   - The predicted labels (`y_pred`) by taking the argmax over the model's softmax outputs.

   Using these, we then computed:

   - Confusion matrix, showing for each true class how many samples were predicted as each possible class.
   - Entire model accuracy, small precision, macro recall and F1-score.

The confusion matrix is then visualized with a heatmap, with class names along both axes. This matrix helped us identify which classes are confused with each other, providing results of the strengths and weaknesses of our model.

3 Results and Analysis

3.1 Training and validation curves

During our training of dataset, both of the training dataset accuracy and validate accuracy steadily improved over the epochs cycles. At the beginning of our training, the model starts around 22% training accuracy and about 13% validation accuracy, which is only slightly better than random guess over 10 classes (10%). As the model training progresses, the model learns useful patterns and the training accuracy gradually climbs up to around 60%, while the validation accuracy reaches approximately 45–46% by the final epochs.

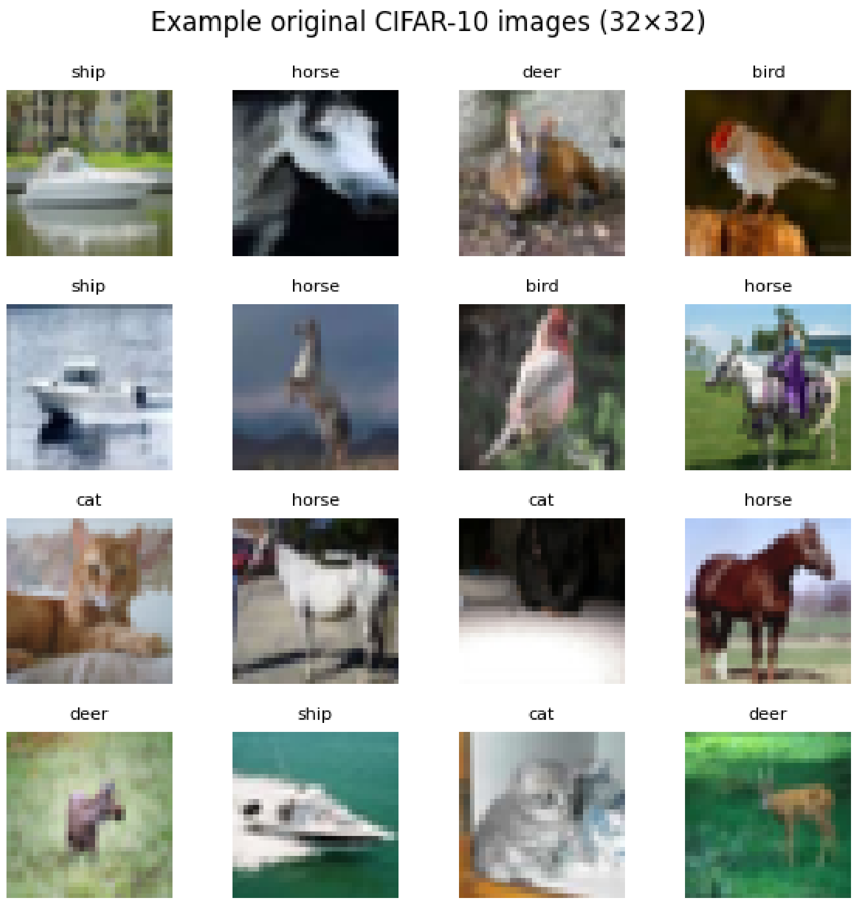


Figure 2: Original CIFAR-10 images (32 × 32) from the dataset.

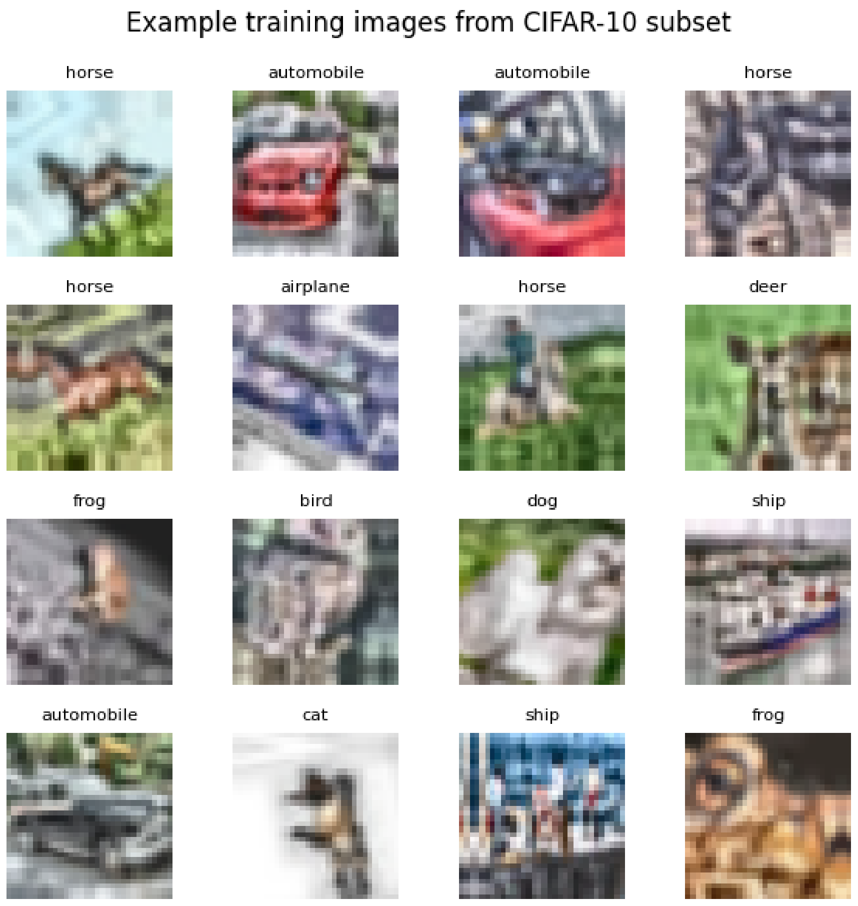
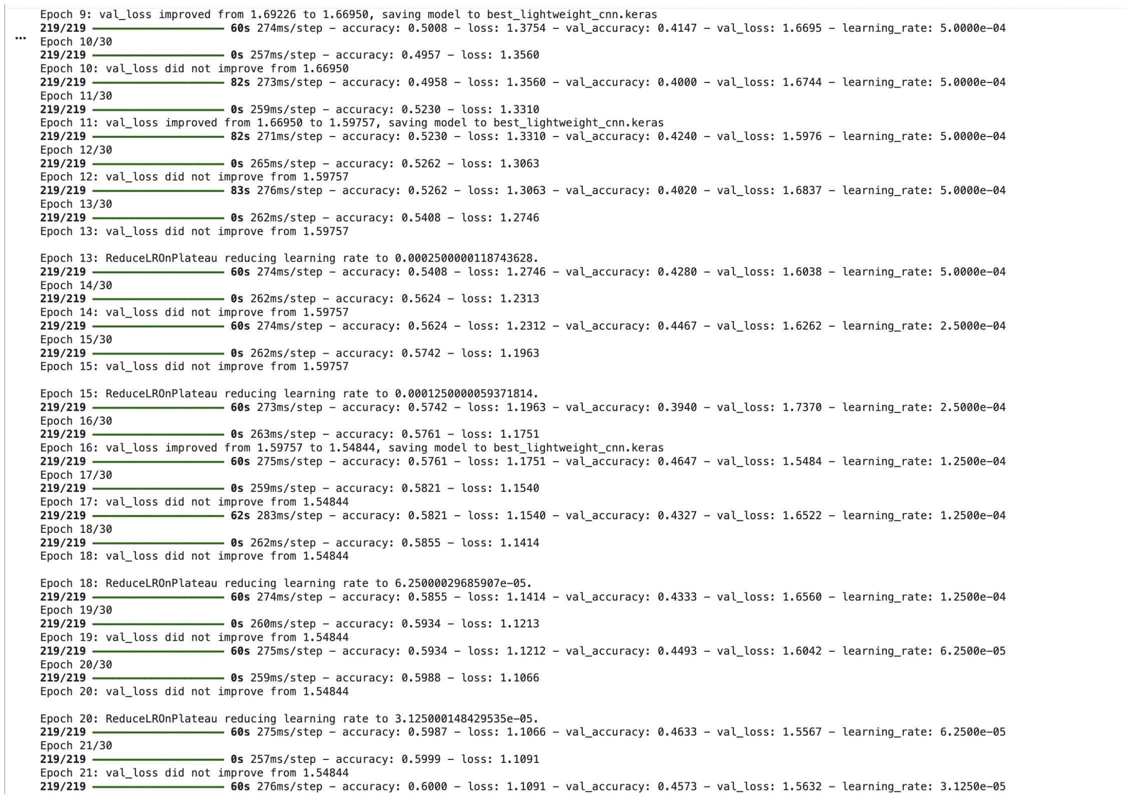


Figure 3: Training images after preprocessing and augmentation



```
    Epoch 9: val_loss improved from 1.69226 to 1.66950, saving model to best_lightweight_cnn.keras
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 274ms/step - accuracy: 0.5008 - loss: 1.3754 - val_accuracy: 0.4147 - val_loss: 1.6695 - learning_rate: 5.0000e-04
... Epoch 10/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 257ms/step - accuracy: 0.4957 - loss: 1.3560
    Epoch 10: val_loss did not improve from 1.66950
    219/219 ━━━━━━━━━━━━━━━━━━━━ 82s 273ms/step - accuracy: 0.4958 - loss: 1.3560 - val_accuracy: 0.4000 - val_loss: 1.6744 - learning_rate: 5.0000e-04
    Epoch 11/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 259ms/step - accuracy: 0.5230 - loss: 1.3310
    Epoch 11: val_loss improved from 1.66950 to 1.59757, saving model to best_lightweight_cnn.keras
    219/219 ━━━━━━━━━━━━━━━━━━━━ 82s 271ms/step - accuracy: 0.5230 - loss: 1.3310 - val_accuracy: 0.4240 - val_loss: 1.5976 - learning_rate: 5.0000e-04
    Epoch 12/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 265ms/step - accuracy: 0.5262 - loss: 1.3063
    Epoch 12: val_loss did not improve from 1.59757
    219/219 ━━━━━━━━━━━━━━━━━━━━ 83s 276ms/step - accuracy: 0.5262 - loss: 1.3063 - val_accuracy: 0.4020 - val_loss: 1.6837 - learning_rate: 5.0000e-04
    Epoch 13/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 262ms/step - accuracy: 0.5408 - loss: 1.2746
    Epoch 13: val_loss did not improve from 1.59757

    Epoch 13: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 274ms/step - accuracy: 0.5408 - loss: 1.2746 - val_accuracy: 0.4280 - val_loss: 1.6038 - learning_rate: 5.0000e-04
    Epoch 14/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 262ms/step - accuracy: 0.5624 - loss: 1.2313
    Epoch 14: val_loss did not improve from 1.59757
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 274ms/step - accuracy: 0.5624 - loss: 1.2312 - val_accuracy: 0.4467 - val_loss: 1.6262 - learning_rate: 2.5000e-04
    Epoch 15/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 262ms/step - accuracy: 0.5742 - loss: 1.1963
    Epoch 15: val_loss did not improve from 1.59757

    Epoch 15: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 273ms/step - accuracy: 0.5742 - loss: 1.1963 - val_accuracy: 0.3940 - val_loss: 1.7370 - learning_rate: 2.5000e-04
    Epoch 16/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 263ms/step - accuracy: 0.5761 - loss: 1.1751
    Epoch 16: val_loss improved from 1.59757 to 1.54844, saving model to best_lightweight_cnn.keras
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 275ms/step - accuracy: 0.5761 - loss: 1.1751 - val_accuracy: 0.4647 - val_loss: 1.5484 - learning_rate: 1.2500e-04
    Epoch 17/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 259ms/step - accuracy: 0.5821 - loss: 1.1540
    Epoch 17: val_loss did not improve from 1.54844
    219/219 ━━━━━━━━━━━━━━━━━━━━ 62s 283ms/step - accuracy: 0.5821 - loss: 1.1540 - val_accuracy: 0.4327 - val_loss: 1.6522 - learning_rate: 1.2500e-04
    Epoch 18/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 262ms/step - accuracy: 0.5855 - loss: 1.1414
    Epoch 18: val_loss did not improve from 1.54844

    Epoch 18: ReduceLROnPlateau reducing learning rate to 6.25000029685907e-05.
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 274ms/step - accuracy: 0.5855 - loss: 1.1414 - val_accuracy: 0.4333 - val_loss: 1.6560 - learning_rate: 1.2500e-04
    Epoch 19/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 260ms/step - accuracy: 0.5934 - loss: 1.1213
    Epoch 19: val_loss did not improve from 1.54844
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 275ms/step - accuracy: 0.5934 - loss: 1.1212 - val_accuracy: 0.4493 - val_loss: 1.6042 - learning_rate: 6.2500e-05
    Epoch 20/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 259ms/step - accuracy: 0.5988 - loss: 1.1066
    Epoch 20: val_loss did not improve from 1.54844

    Epoch 20: ReduceLROnPlateau reducing learning rate to 3.125000148429535e-05.
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 275ms/step - accuracy: 0.5987 - loss: 1.1066 - val_accuracy: 0.4633 - val_loss: 1.5567 - learning_rate: 6.2500e-05
    Epoch 21/30
    219/219 ━━━━━━━━━━━━━━━━━━━━ 0s 257ms/step - accuracy: 0.5999 - loss: 1.1091
    Epoch 21: val_loss did not improve from 1.54844
    219/219 ━━━━━━━━━━━━━━━━━━━━ 60s 276ms/step - accuracy: 0.6000 - loss: 1.1091 - val_accuracy: 0.4573 - val_loss: 1.5632 - learning_rate: 3.1250e-05
```

Figure 4: Training log showing epochs, accuracy, validation accuracy, loss, validation loss, and learning rate schedule.

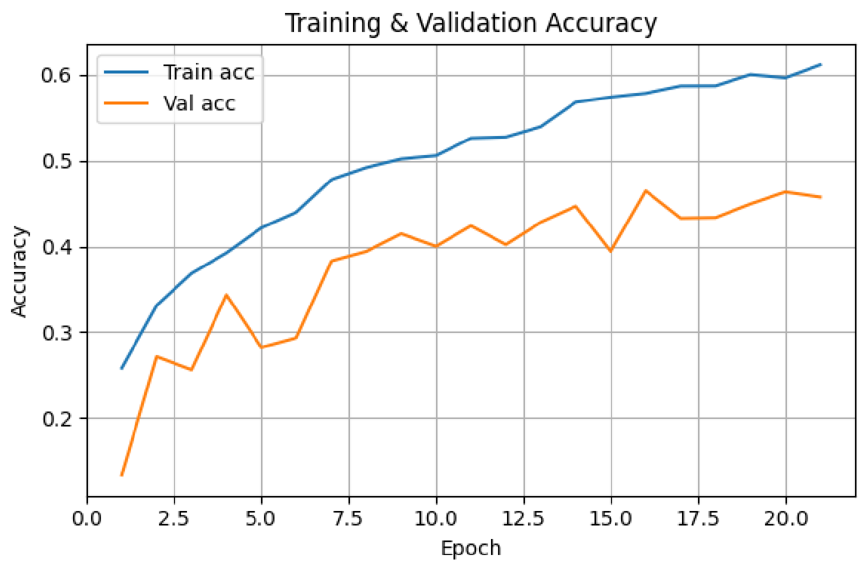


Figure 5: Training and val dataset accu curves across epochs for the lightweight CNN.

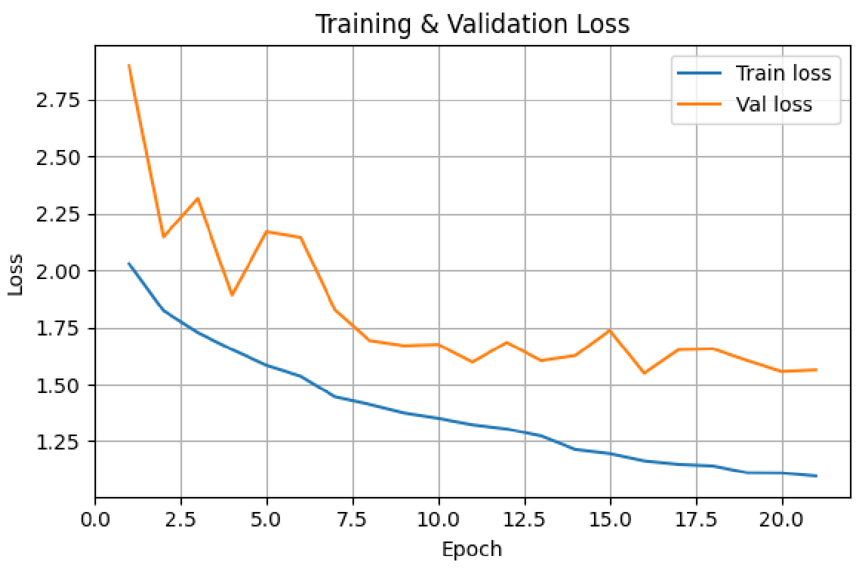


Figure 6: Training and validation loss curves across epochs for the lightweight CNN.

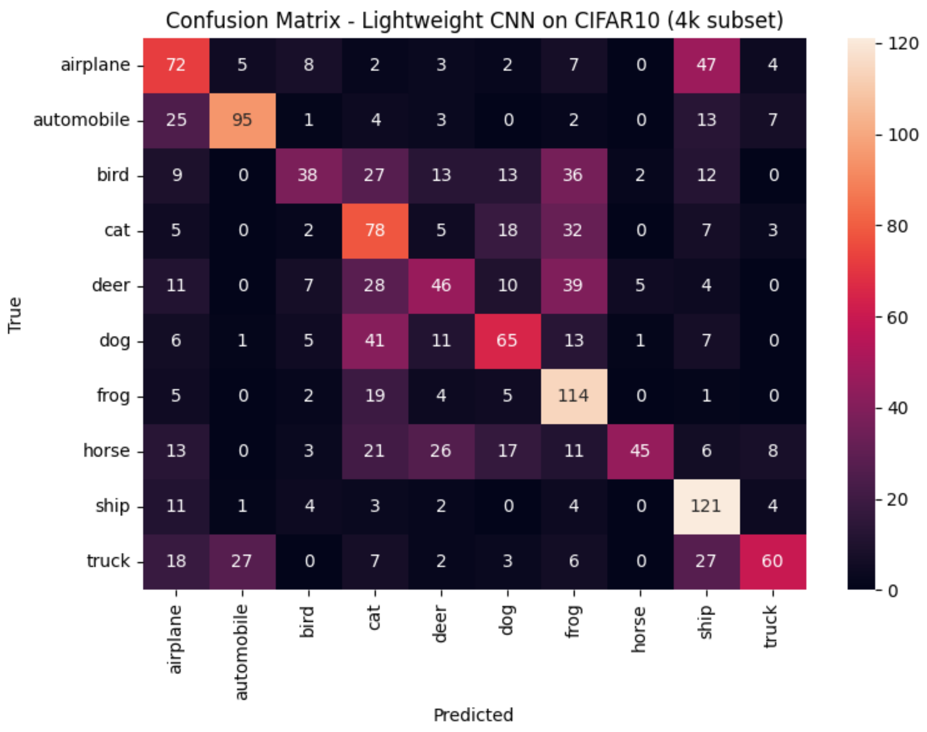


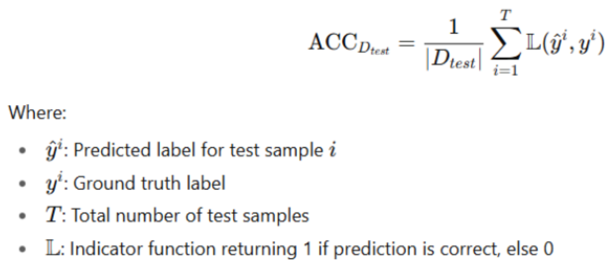
Figure 7: Confusion matrix of the lightweight CNN on the CIFAR-10 subset test set.

The training loss consistently decreases, indicating that the model is fitting the training data well. The val data loss also goes down initially, then stabilizes and again changes slightly. Here we can observe there is a noticeable gap between training accuracy (about 60%) and validation accuracy (about 45–46%) in the later epochs, which suggests a certain level of overfitting: our dataset model performs good on the training set than on unknown validation data. However, as the gap is not much, and the validation accuracy does not collapse, so the model still generalizes to some extent.

Here the use of early stopping and learning rate reduction on plateau helps to control this behavior. Early stopping prevents the model from training too long where it is mostly overfitting, the training/validation curves overall show a reasonable results between learning capacity and generalization given as the dataset is small 7,000-image training set and the lightweight architecture.

3.2 Test set performance and overall metrics

The primary evaluation metric is classification accuracy on the test set D_{test}.



$$\text{ACC}_{D_{test}} = \frac{1}{|D_{test}|} \sum_{i=1}^{T} \mathbb{L}(\hat{y}^i, y^i)$$

Where:

- $\hat{y}^i$: Predicted label for test sample i
- y^i: Ground truth label
- T: Total number of test samples
- $\mathbb{L}$: Indicator function returning 1 if prediction is correct, else 0

Figure 8

where $T = |D_{\text{test}}|$ is the number of test samples.

After our training, the best model (according to validation loss) is evaluated on the held-out test set of 1,500 images. The results are:

- Test loss: 1.4838
- Test accuracy: 0.4893 (approximately 48.93%)

This means our model correctly classifies roughly half of the test images. Considering that:

- The dataset is a relatively small subset (10,000 images out of the full 60,000),
- The images are low-resolution (32×32),
- And the model is intentionally lightweight ($\approx$ 158k parameters),

this level of performance is considered as consistent for a compact CNN on a limited amount of data, without very deep architectures.

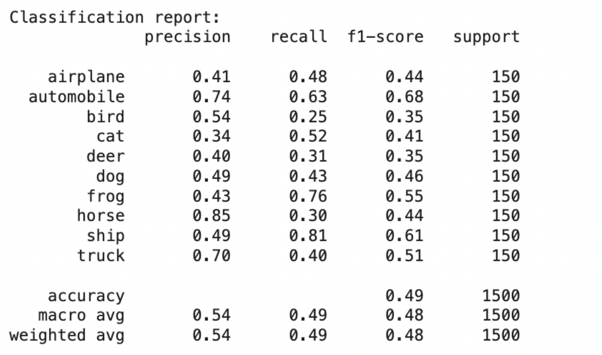
To get a deeper view of performance,we added additional metrics using scikit-learn:

- Macro precision: 0.5391
- Macro recall: 0.4893
- Macro F1-score: 0.4802

These values indicate that, on average across all 10 classes, the model has around 54% precision, 49% recall, and 48% F1-score.

3.3 Class-wise performance and confusion patterns

The classification report reveals how the model behaves for each individual class:



```
Classification report:
              precision    recall  f1-score   support

    airplane       0.41      0.48      0.44       150
  automobile       0.74      0.63      0.68       150
        bird       0.54      0.25      0.35       150
         cat       0.34      0.52      0.41       150
        deer       0.40      0.31      0.35       150
         dog       0.49      0.43      0.46       150
        frog       0.43      0.76      0.55       150
       horse       0.85      0.30      0.44       150
        ship       0.49      0.81      0.61       150
       truck       0.70      0.40      0.51       150

    accuracy                           0.49      1500
   macro avg       0.54      0.49      0.48      1500
weighted avg       0.54      0.49      0.48      1500
```

Figure 9: Classwise Performance.

A few clear patterns says that:

- Automobile, ship, and truck are among the strongest classes.
  - Automobile has high precision (0.74) and good recall (0.63), shows these that the model predicts "automobile", this is correct and it knows many of the true automobile images.
  - Ship has very high recall (0.81) with moderate precision (0.49), meaning the model successfully captures most ship examples, but also sometimes over-predicts "ship" for other classes.
  - Truck also has relatively strong precision (0.70).

  We can say that these patterns are typical for CIFAR-10: animal classes tend to be harder and more overlapping, while vehicles are easier to identify. The detailed metrics and confusion matrix confirm that the model behaves reasonably and reveal where it struggles.

3.4 Feature visualization and interpretability

To understand what the model actually learns, here we visualized feature maps from several convolutional layers using the activation model.

We examined the activations from the following layers:

- Early layer: `convol11`, `convol12`
- Middle layers: `convol21`, `convol22`
- Deepest layer: `convol31`

From the compilations from our outputs, we can say that the feature maps from the early layers show patterns consistent with typical CNN behavior: they respond strongly to simple edges, color gradients, and local texture.

In the middle layers, the feature maps become more abstract and localized.And in the deepest convolutional layer, the feature maps are even more focused.Even though CIFAR-10 images are small and sometimes noisy, these deeper maps still capture structured patterns rather than random noise.

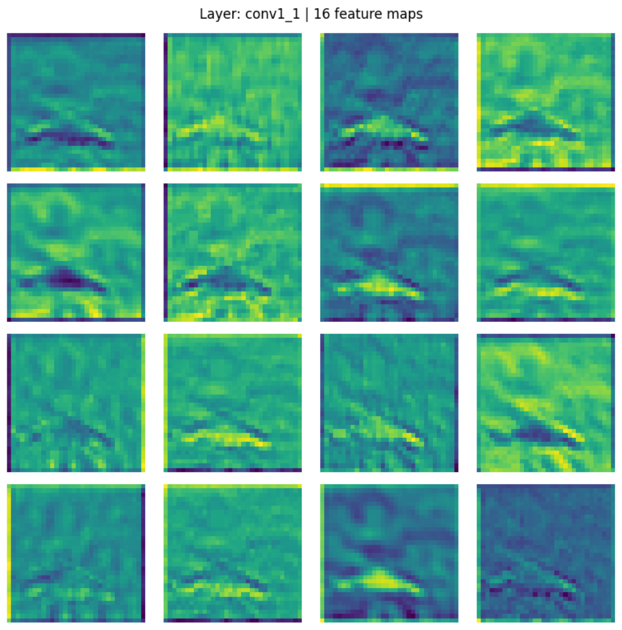


Figure 10: Layer-1 Convolution Feature map.

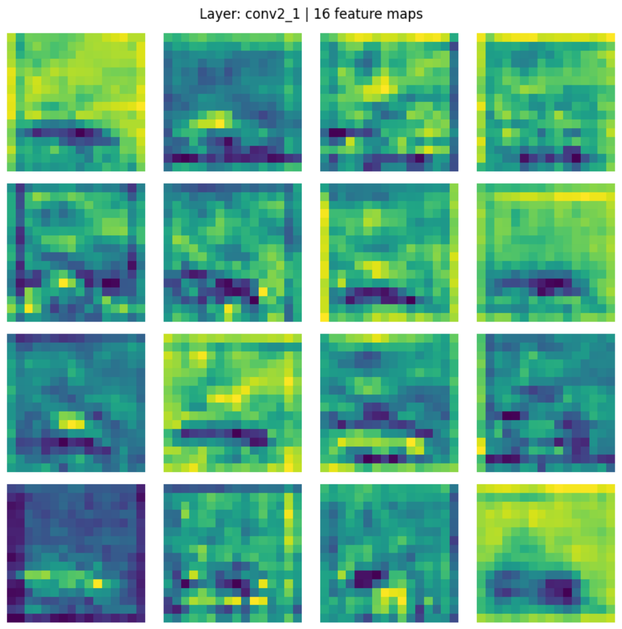


Figure 11: Layer-2 Convolution Feature map.

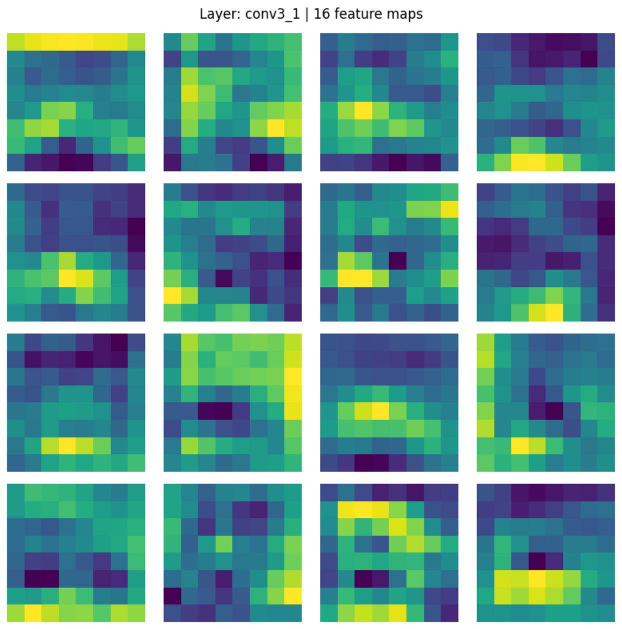


Figure 12: Layer-3 Convolution Feature map.

3.5 Overall performance details

In summary, the results show that the proposed model consisting of CLAHE-based preprocessing, data augmentation, and a lightweight CNN achieves around 49% test accuracy with macro F1-score ≈ 0.48 on a 10,000-image subset of CIFAR-10. While this can be lower than topend performance when comapared to big dataset with big models, it is consistent with expectations for:

- A relatively small training set (7,000 images),
- A compact architecture (≈ 158k parameters),
- And no aggressive hyperparameter tuning or ensembling.

Our results also demonstrate that:

- The preprocessing and augmentation strategy helps the model reach reasonable generalization without extreme overfitting.
- The confusion matrix and per-class metrics gave information into which classes are easier or harder to classify.
- Feature map visualization provides an interpretability component, showing that the model learns structured,from low-level edges to more abstract patterns.

This combination of performance, efficiency, and interpretability aligns with the goals of the project: to design a lightweight, fast-to-train CNN pipeline that works on small/noisy datasets.

4 Conclusion

4.1 Summary

This project gave us to design and evaluate a lightweight CNN pipeline for image classification on a small and potentially noisy subset of the CIFAR-10 dataset. By combining contrast-limited adaptive histogram equalization (CLAHE), data augmentation, and a compact convolutional architecture, our model achieved a test accuracy of about 48.9% with macro precision, recall, and F1-scores around 0.54, 0.49, and 0.48 respectively on a held-out set of 1,500 images. These results confirm that even with limited data (10,000 images total, 7,000 for training) and a relatively small network (∼158k parameters), it is possible to obtain reasonable performance without going to very deep or computationally expensive models.

The training and validation curves revealed moderate overfitting training accuracy reached around 60% while validation stabilized in the mid-40% range but early stopping and learning rate scheduling helped keep this under control. Overall, the model demonstrated a good knock off between accuracy, efficiency, and robustness for a constrained data and compute setting.

4.2 Insights on Interpretability

By visualizing feature maps from multiple convolutional layers, we observed the typical progression of learned representations: early layers responded to simple edges, color transitions, and textures, while deeper layers became more selective and concentrated on more abstract, object-related patterns. These visualizations provided evidence that our model was learning meaningful structure rather than random noise.

The feature maps also revealed how different channels specialize in different patterns, such as localized edges, blobs, and object parts. This kind of analysis helps us verify that the network uses sensible visual hints and can highlight situations where the model might be relying too heavily on background textures or spurious features.

4.3 Recommendations for Future Work

Although the results are competitive for a lightweight model on a small dataset, there is a scope for future improvement:

- Exploring alternative or stronger architectures (e.g., depthwise separable convolutions,) while still keeping the model relatively small and fast to train.
- Experimenting with different preprocessing strategies, such as per-channel normalization, different CLAHE parameters, or other contrast enhancement and denoising methods tailored to low-resolution images.
- Scaling up to a larger portion of CIFAR-10 or to different datasets to test the generality of the model across domains and image qualities, and comparing the lightweight approach with more complex baselines.