Image Classification on CIFAR-10 Using a Convolutional Neural Network

Course: CS 6840/4840 – Intro to Machine Learning

Student: Muhammad Islam Riaz

Dataset: CIFAR-10

Framework: TensorFlow/Keras

1. Objectives

The domain of this project is image classification in computer vision. The goal is to automatically assign each small color image to one of ten object categories, such as airplane, automobile, bird, cat, and so on. Even though CIFAR-10 images are only 32×32 pixels, the dataset is still challenging because the objects appear in different poses, backgrounds, and lighting conditions.

My main objectives in this project were:

• To build a supervised learning model that can classify CIFAR-10 images with reasonable accuracy.

• To learn how to design, train, and evaluate a Convolutional Neural Network (CNN) using TensorFlow and Keras.

• To perform a few experiments with architecture and training settings (number of epochs, size of the dense layer, and model depth) and compare the results.

• To analyze performance using accuracy, precision, recall, confusion matrices, and per-class error patterns.

A practical benefit of this kind of model is that similar CNNs are used in real applications like automatic tagging of photos, detecting objects in self-driving cars, or sorting images in large databases.

2. Dataset Description and Preprocessing

For this project I used the CIFAR-10 dataset, which is a standard benchmark dataset for image classification. It is available directly through tf. keras.datasets.

Number of images:
• 50,000 training images
• 10,000 test images

Image size: 32 × 32 pixels
Channels: 3 (RGB color images)
Number of classes: 10 (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck)

When I loaded the data, the shapes printed were:
• Train images: (50000, 32, 32, 3)
• Train labels: (50000, 1)
• Test images: (10000, 32, 32, 3)
• Test labels: (10000, 1)

To have a separate validation set, I split the original 50,000 training images into:
• Training set: 40,000 images
• Validation set: 10,000 images
• Test set: 10,000 images (kept untouched until final evaluation)

So the final shapes were:
• Train: (40000, 32, 32, 3) and (40000, 1)
• Validation: (10000, 32, 32, 3) and (10000, 1)
• Test: (10000, 32, 32, 3) and (10000, 1)

Preprocessing:

• Normalization: I converted the pixel values from integers in [0, 255] to floats in [0, 1] by dividing by 255.0. This helps the optimizer converge faster.

• Label format: The labels are integer class indices from 0 to 9, so I used sparse_categorical_crossentropy as the loss function (no one-hot encoding needed).

I did not apply data augmentation in this project, mostly to keep the experiments simple and to focus on understanding the core CNN model.

This dataset is interesting to me because it is small enough to train on my own laptop (MacBook Pro with M4 and 24 GB RAM), but still realistic enough to see the behavior of a real CNN model and overfitting/underfitting effects.

3. Model Choice and Architecture

For this project I chose a Convolutional Neural Network (CNN) implemented using TensorFlow/Keras. CNNs are a natural choice for image data because they can learn local patterns (edges, corners, textures) and then combine them into higher-level features (shapes, object parts).

I decided to use a simple, custom CNN instead of a very large pre-trained model because:
• It is easier to understand and explain each layer.
• Training time is reasonable on my machine.
• It still reaches around 75% test accuracy on CIFAR-10, which is acceptable for an intro project.

Baseline architecture (in Keras):

```
model = models.Sequential([
    # Block 1
    layers.Conv2D(32, (3, 3), activation="relu",
                  padding="same", input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),

    # Block 2
    layers.Conv2D(64, (3, 3), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    # Block 3
    layers.Conv2D(128, (3, 3), activation="relu", padding="same"),
    layers.MaxPooling2D((2, 2)),

    # Classifier
    layers.Flatten(),
    layers.Dense(128, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(10, activation="softmax")
])
```

Layer description:
• Conv2D (32 filters, 3×3, ReLU, same padding): learns low-level patterns like edges and color blobs.
• MaxPooling2D (2×2): downsamples the feature maps, making them smaller and more robust.
• Conv2D (64 filters, 3×3) + MaxPooling2D: learns more complex patterns and reduces size further.
• Conv2D (128 filters, 3×3) + MaxPooling2D: produces 4×4×128 feature maps representing higher-level concepts.
• Flatten: converts the 4×4×128 tensor into a vector of length 2048.
• Dense(128, ReLU): fully connected layer that combines all features.
• Dropout(0.5): drops 50% of the neurons during training to reduce overfitting.

• Dense(10, softmax): output layer that produces a probability distribution over the 10 classes.

This architecture is a compromise between simplicity and capacity. It is small enough to understand and train quickly, but still deep enough to learn meaningful patterns from the images.

4. Training Process

I implemented and trained the model in Python using TensorFlow 2.20.0 and the Keras API.

Hyperparameters (baseline model):
• Optimizer: Adam
• Loss function: sparse_categorical_crossentropy
• Metrics: accuracy
• Batch size: 64
• Number of epochs (baseline): 15
• Train/validation split: 40,000 / 10,000

The model was compiled as:

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

Training was started with:

```
history = model.fit(
    x_train, y_train,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    validation_data=(x_val, y_val)
)
```

During training, Keras breaks the training data into mini-batches of 64 images. For each batch, the model makes predictions, computes the loss, and updates its weights using backpropagation and Adam. At the end of each epoch, the model is evaluated on the validation set (x_val, y_val) to track validation accuracy and loss.

After training, I evaluated the final model on the test set using:

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=1)
```

I also computed the classification report and confusion matrix using scikit-learn.

5. Experiments and Results

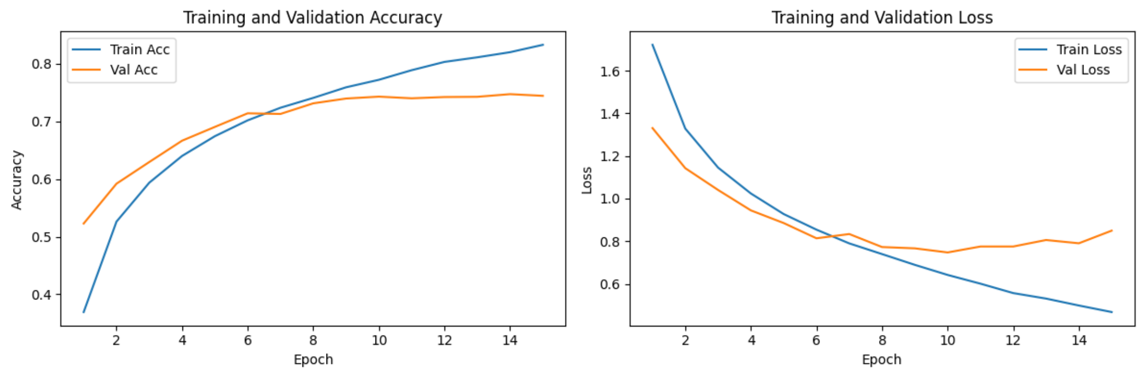
I ran several experiments around the baseline model to understand how different choices affect performance.

5.1 Baseline Model (3 Conv Blocks, Dense 128, 15 Epochs)

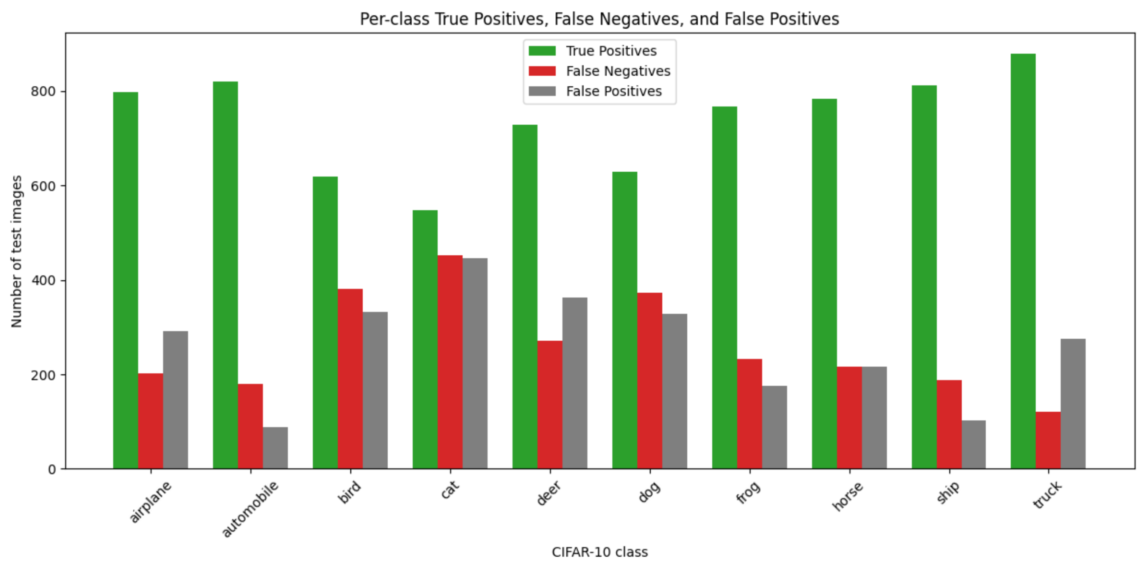
Architecture: same as described in Section 3. I trained this model for 15 epochs with batch size 64.

Test performance (baseline):
• Test accuracy: approximately 0.74–0.75
• Test loss: around 0.8
• Macro-average precision, recall, and F1-score: around 0.75

Training/validation accuracy and loss plot for the baseline:



TP/FN/FP bar graph for the baseline model here:



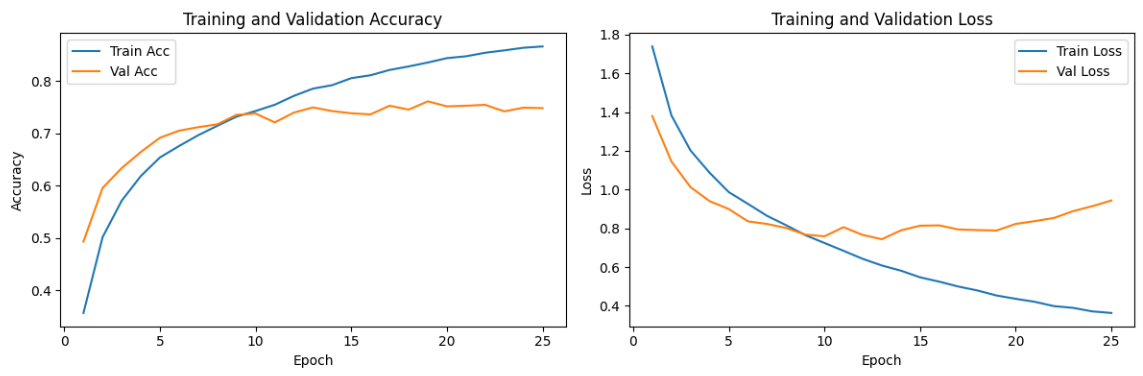
Classification Report:

Classes	precision	recall	f1-score	support
airplane	0.7325	0.7970	0.7634	1000
automobile	0.9032	0.8210	0.8601	1000
bird	0.6505	0.6180	0.6338	1000
cat	0.5509	0.5470	0.5489	1000
deer	0.6673	0.7280	0.6963	1000
dog	0.6562	0.6280	0.6418	1000
frog	0.8142	0.7670	0.7899	1000
horse	0.7832	0.7840	0.7836	1000
ship	0.8874	0.8120	0.8480	1000
truck	0.7617	0.8790	0.8162	1000

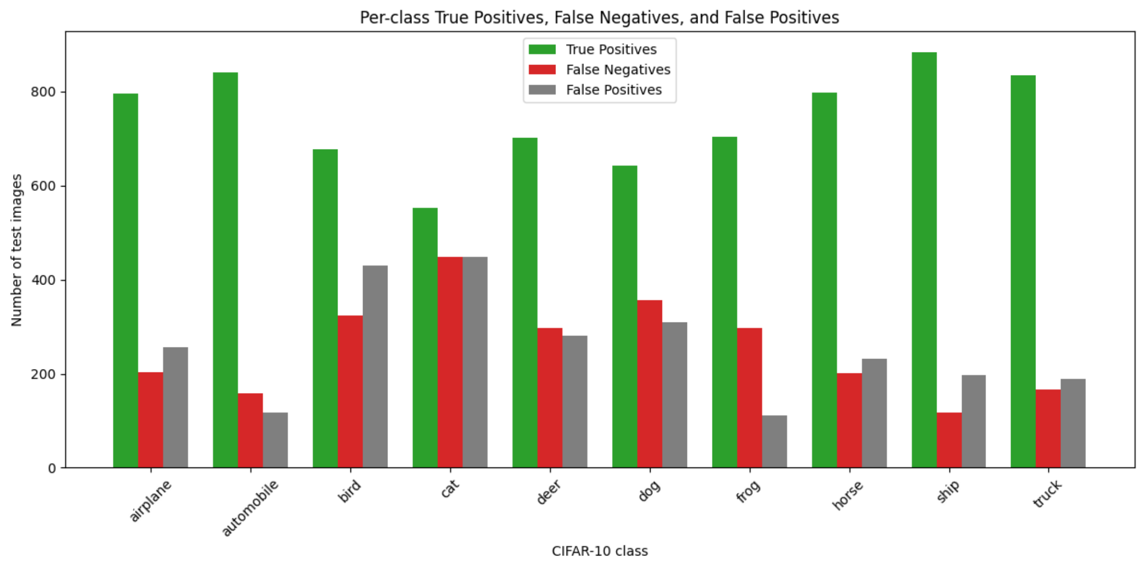
5.2 Experiment 1: More Epochs (25 Epochs)

In this experiment I kept the same architecture but increased the training epochs from 15 to 25. This slightly improved test accuracy but also showed more signs of overfitting in the validation curves.

Training/validation accuracy and loss plots for Experiment 1



TP/FN/FP bar graph for Experiment 1:



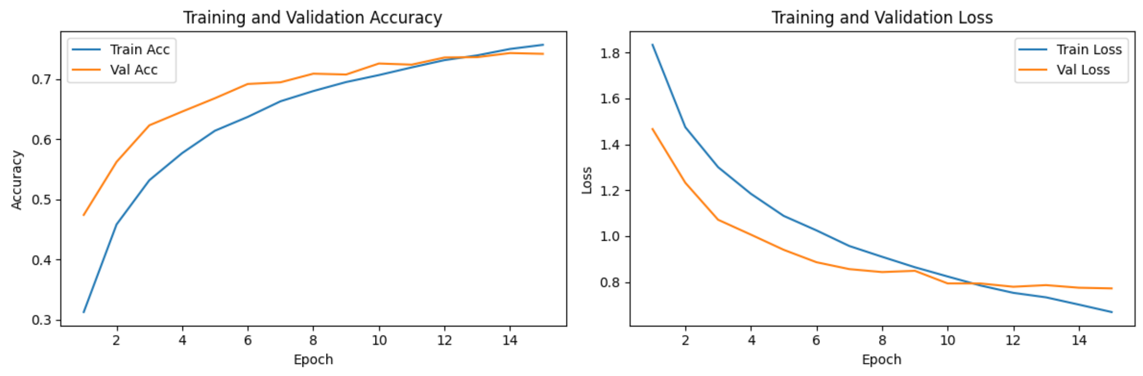
Classification report for Experiment 1:

label	precision	recall	f1-score	support
airplane	0.7567	0.7960	0.7758	1000
automobile	0.8779	0.8410	0.8590	1000
bird	0.6112	0.6760	0.6420	1000
cat	0.5514	0.5520	0.5517	1000
deer	0.7141	0.7020	0.7080	1000
dog	0.6747	0.6430	0.6585	1000
frog	0.8638	0.7040	0.7758	1000
horse	0.7755	0.7980	0.7866	1000
ship	0.8168	0.8830	0.8486	1000
truck	0.8151	0.8330	0.8239	1000

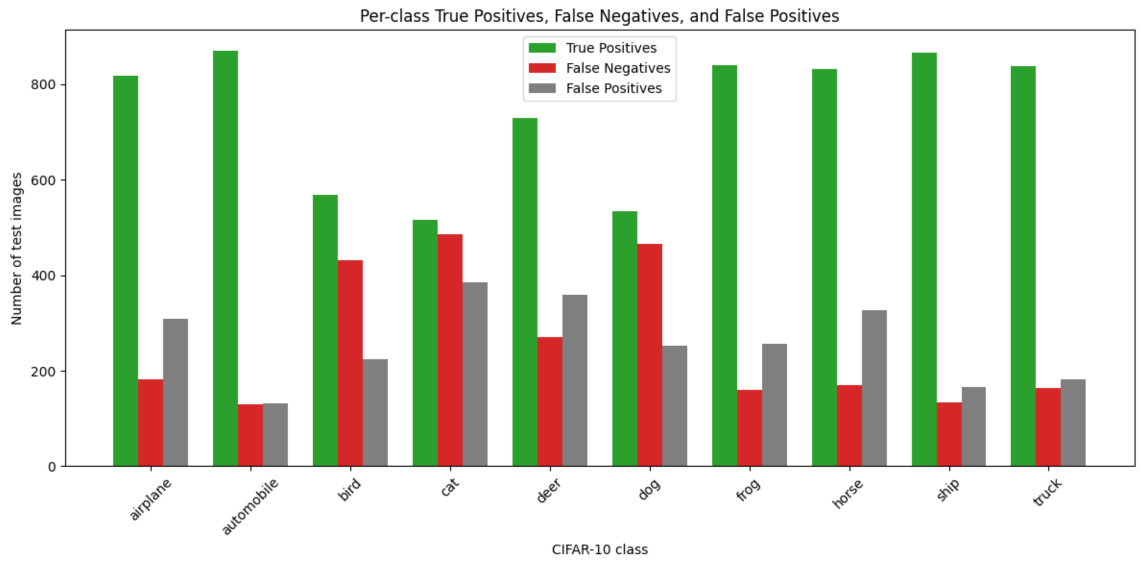
5.3 Experiment 2: Smaller Dense Layer (64 Units)

In this experiment I reduced the size of the dense layer from 128 units to 64 units. This lowered the model capacity and led to lower accuracy, which is a sign of underfitting.

Training/validation accuracy and loss plots for Experiment 2:



TP/FN/FP bar graph for Experiment 2:



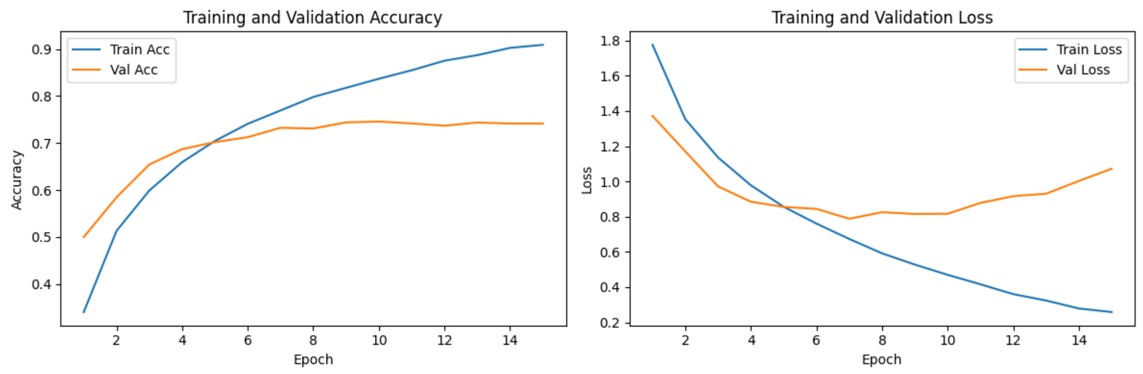
Classification report for Experiment 2:

Label	Precision	Recall	F1-score	Support
airplane	0.7256	0.8170	0.7686	1000
automobile	0.8683	0.8700	0.8691	1000
bird	0.7175	0.5690	0.6347	1000
cat	0.5716	0.5150	0.5418	1000
deer	0.6700	0.7290	0.6983	1000
dog	0.6785	0.5340	0.5976	1000
frog	0.7664	0.8400	0.8015	1000
horse	0.7182	0.8310	0.7705	1000
ship	0.8400	0.8660	0.8528	1000
truck	0.8214	0.8370	0.8291	1000

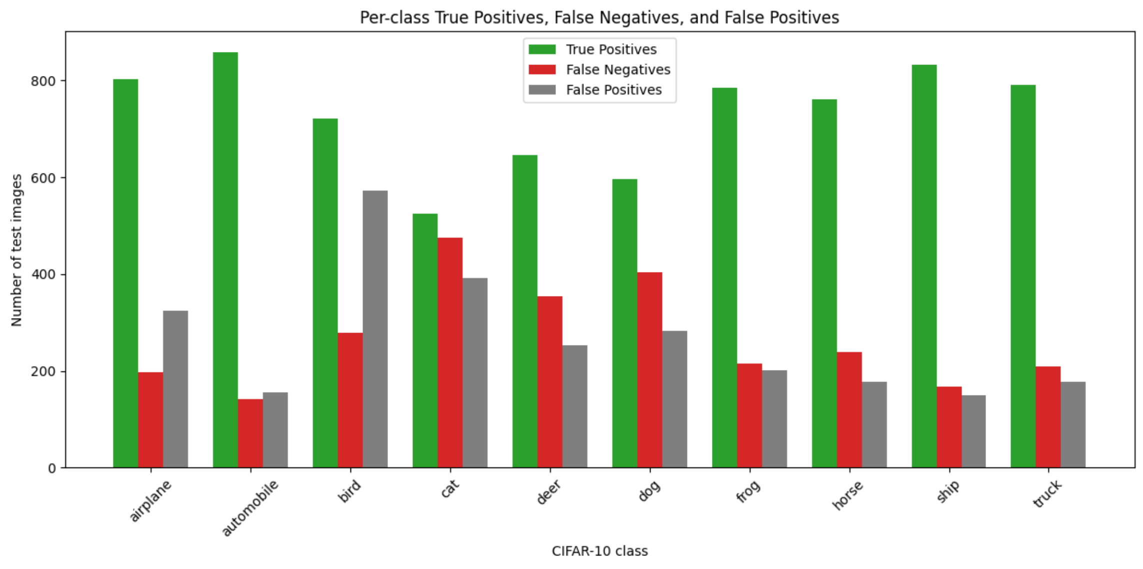
5.4 Experiment 3: Deeper CNN (Extra Conv Layer with 256 Filters)

In this experiment I added an extra convolutional block with 256 filters to make the network deeper. The model fit the training data very well but did not improve on the test set and showed more overfitting.

Training/validation accuracy and loss plots for Experiment 3:



TP/FN/FP bar graph for Experiment 3:



Classification report for Experiment 3:

Label	precision	recall	f1-score	support
airplane	0.7125	0.8030	0.7551	1000
automobile	0.8470	0.8580	0.8525	1000
bird	0.5572	0.7210	0.6286	1000
cat	0.5725	0.5250	0.5477	1000
deer	0.7183	0.6450	0.6797	1000
dog	0.6792	0.5970	0.6354	1000
frog	0.7961	0.7850	0.7905	1000
horse	0.8111	0.7600	0.7847	1000
ship	0.8473	0.8320	0.8396	1000
truck	0.8170	0.7900	0.8033	1000

5.5 Summary of Experiment Comparison

Across all models:
• The baseline model (3 conv blocks, Dense 128, 15 epochs) provided the best balance between accuracy, simplicity, and training time.
• Training longer (25 epochs) slightly improved accuracy but also increased overfitting.
• Reducing the dense layer size to 64 units led to underfitting and lower accuracy.

• Adding an extra conv block with 256 filters increased model capacity but did not generalize better to the test set.

6. Summary and Contributions

In this project, I implemented and evaluated a Convolutional Neural Network for CIFAR-10 image classification using TensorFlow and Keras. I went through the full pipeline from loading and preprocessing the data, to designing the model, training it, running experiments, and analyzing the results.

My main contributions in this project are:
• Implementing a full CNN classification pipeline for CIFAR-10 from scratch using TensorFlow/Keras.
• Designing and running multiple controlled experiments on architecture and training duration.
• Carefully evaluating the models using accuracy, precision, recall, confusion matrices, and TP/FN/FP bar graphs.
• Reflecting on overfitting and underfitting behavior and selecting a final model that balances performance and complexity.

If I had more time, I would like to try data augmentation (random flips, crops) and possibly transfer learning with a small pre-trained model to see how much I can further improve the accuracy. However, for this course project, I focused on understanding and explaining a custom CNN model in detail rather than using a very large pre-trained architecture.