جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⵉⵏ

AL AKHAWAYN UNIVERSITY

Project 4

CSC 3315

Spring 2024

Ouballouk Mohamed
Salim El Ghersse
Maria Chmite

Supervised by: Dr. Tajjeddine Rachidi

26 April 2024

Contents

Introduction:

In this project, the CIFAR-10 dataset stands as a fundamental benchmark in the field of machine learning, particularly in the realm of image recognition tasks. This dataset comprises 60 000 color images, each of 32x32 pixels, categorized across ten distinct classes including airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. These images are split into a training set of 50 000 images and a testing set of 10 000 images, enabling the effective training and validation of image classification models.

Our project aims to leverage the CIFAR-10 dataset to develop and evaluate a convolutional neural network (CNN) that can accurately classify images into one of the ten categories. This work not only allows us to understand and implement cutting-edge image

recognition techniques but also serves as an excellent testbed to explore various aspects of neural network architecture design, such as layer configuration, activation functions, and optimization algorithms.

Through this project, we seek to demonstrate the capabilities of CNNs in handling complex image data, refine our model to achieve higher accuracy, and draw insights on the computational efficiency and practical applicability of our approach. The following sections of this report will detail the methodologies employed in preparing the dataset, designing the neural network model, and the subsequent results and analysis derived from our experiments.

1. Task Distribution:

All members of the team contributed to all tasks. Everyone actively participated and shared their ideas and efforts. We achieved outstanding results thanks to great teamwork and the support of each member.

Each one of us contributed and we did this project all together

Tasks	**Assigned to**
Implementation of CNN Model as mentioned in the Video	Salim EL GHERSSE Mouhamed OUBALLOUK Maria CHMITE
Implementation of Different Models	Salim EL GHERSSE Mouhamed OUBALLOUK Maria CHMITE
Training Different Models	Salim EL GHERSSE Mouhamed OUBALLOUK Maria CHMITE
Tuning Different Hyperparameters	Salim EL GHERSSE Mouhamed OUBALLOUK Maria CHMITE

Report	Salim EL GHERSSE Mouhamed OUBALLOUK Maria CHMITE

2. Technical background:

2.1. Convolutional Neural Networks (CNNs) Overview:

CNNs are a specialized kind of neural network used primarily to process grid-like data such as images. CNNs are particularly powerful for tasks involving visual recognition, classification, and feature extraction, making them the architecture of choice for computer vision applications.

2.2. Architecture of CNNs:

CNNs are distinguished by their unique architecture which often includes the following layers:

- Convolutional Layers: These are the core building blocks of a CNN. Each convolutional layer applies numerous filters to the input. These filters are small but extend through the full depth of the input volume. As the filter slides (or convolves) around the input image, it applies a dot product between the filter and the input, producing a two-dimensional activation map of that filter. As a result, the network learns filters that activate when they see specific types of features at some spatial position in the input.
- ReLU Layer: This layer applies a non-linear activation function, which introduces non-linearity to the system, allowing it to learn more complex patterns. The most common function used is the Rectified Linear Activation (ReLU), which simply replaces all negative pixel values in the feature map with zero.
- Pooling Layers: Also known as down-sampling, pooling reduces the dimensionality of each feature map but retains the most essential information. Max pooling, one of the most common pooling techniques, slides a small window across the feature map and records the maximum value in each region.
- Fully Connected Layers: These layers connect every neuron in one layer to every neuron in the next layer. In the context of CNNs, fully connected layers are typically

used at the end of the network to assemble and classify the features extracted by convolutional layers and down sampling into the final output, such as image classifications.

- Training: Gradient-based optimization algorithms like Stochastic Gradient Descent (SGD) or its variations are commonly used to train CNNs. Backpropagation aids in network parameter optimization and enhances generalization when paired with strategies like batch normalization and dropout regularization.

- Applications: CNNs are extensively utilized in a wide range of computer vision tasks, such as autonomous driving, object detection, facial recognition, medical image analysis, and image classification. They perform exceptionally well in tasks where traditional machine learning techniques falter due to their capacity to automatically extract features from unprocessed data.

2.3. Role of Keras and TensorFlow:

- Keras: Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. Designed for human beings, not machines, Keras is a framework that empowers engineers to experiment quickly with deep neural networks. It focuses on being user-friendly, modular, and extensible.
- TensorFlow: TensorFlow, developed by Google, is an end-to-end open-source platform for machine learning that enables researchers to develop and train ML models easily and efficiently. It supports extensive production and research workloads and allows developers to transition ML models from research to deployment seamlessly.

This project leverages both Keras and TensorFlow to harness the ease of model building and training offered by Keras, along with the robust, scalable production capabilities provided by TensorFlow. This combination allows for the development of complex deep learning models with high efficiency and speed, which is essential for handling large datasets such as CIFAR-10 used in this project.

3. Implementation Details:

3.1. Data acquisition and preparation:

The CIFAR-10 dataset is a benchmark in the field of machine learning for image recognition. It consists of 60,000 32x32 pixel color images divided equally into 10 classes, each representing different objects such as airplanes, cars, birds, cats, etc. This diversity makes it ideal for training and testing image classification models.

3.2. Download and Preparation Steps:

- Downloading the Dataset: The dataset was accessed through the Kaggle competition page. Using Kaggle's API facilitated direct downloading of the dataset into the Google Colab environment, streamlining the setup process for immediate use.
- Unzipping the Data: Once downloaded, the dataset was extracted from its compressed form to make it accessible for loading and processing.
- Preprocessing: This step involved normalizing the pixel values of the images to a rangeof 0 to 1. This normalization helps in speeding up the training by ensuring that each input parameter (pixel, in this case) has a similar data distribution. This makes convergence faster while training the network.
- However, this step was done initially but in other codes we could import cifar10 from the tensorflow.keras.datasets.

```
!kaggle competitions download -c cifar-10
```

```
[2]  !unzip cifar-10.zip
```

implementation

we import the cifar10 dataset by using this line of code

```
# Load dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

datasets.load_data().cifar10: TensorFlow/Keras comes with the CIFAR-10 dataset, which is loaded via this function call.

2 tuples are returned by the function:

(x_train, y_train): Holds the labels that go with the training images.

(x_test,y_test): This array holds the labels that go with the testing images.

Next, the variables x_train, y_train , x_test,y_test are assigned the images and labels.

Normalization

Code snippet

```
# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0
```

This code normalizes the image pixel values in the CIFAR-10 dataset from their original range of 0 to 255 to a new range of 0 to 1, normalizing the pixel values. In order to obtain normalized values that are better suited for training neural network models, this is accomplished by dividing each pixel value by 255.0. Through the standardization of the input data, normalization ensures that all pixel values fall within a more manageable, smaller range, which enhances training performance and speed.

Sampling the data

Code snippet

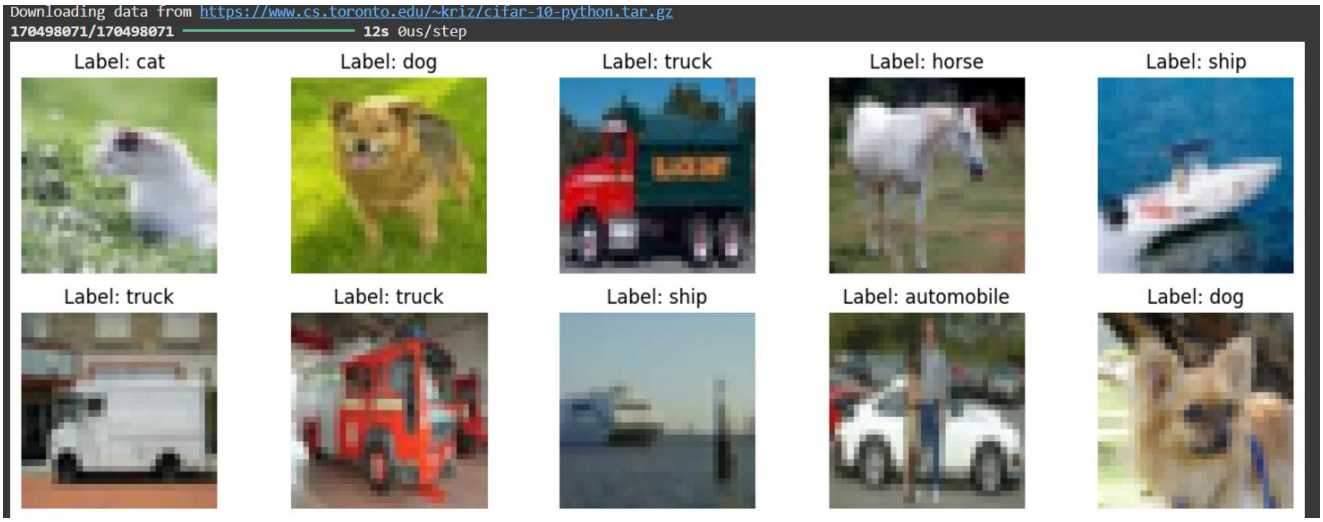
```
# Label mapping
label_mapping = {
    0: 'airplane',
    1: 'automobile',
    2: 'bird',
    3: 'cat',
    4: 'deer',
    5: 'dog',
    6: 'frog',
    7: 'horse',
    8: 'ship',
    9: 'truck'
}

# Plot the random images with labels
plt.figure(figsize=(15, 5))
for i, idx in enumerate(random_indices, 1):
    plt.subplot(2, 5, i)
    plt.imshow(x_test[idx])
    label_index = y_test[idx][0]
    plt.title(f"Label: {label_mapping[label_index]}")  # Display the mapped label
    plt.axis('off')

plt.show()
```

Using the CIFAR-10 test dataset, this code snippet creates a figure that shows a grid of 10 photos (2 rows and 5 columns). Based on the indices in random_indices, a random selection is made for each image. Each image is rendered via the plt.imshow() function, and plt.axis('off') eliminates axis labels and ticks to concentrate only on the visual content. Examining the actual photos used in the predictions or conducting any other study involving the CIFAR-10 dataset is made easier with the help of this visualization. The size of the entire plot can be altered with changes to plt.figure(figsize=()), giving the presentation of the images more flexibility

Result:



3.3. Model Architecture

The architecture of the model is crucial for the performance of the image classification task. The designed model utilizes a series of convolutional, activation, normalization, and pooling layers, each structured to perform specific transformations on the input data to effectively learn from the CIFAR-10 dataset.

Detailed Layer Description:

1. Convolutional Layers:

- First Convolutional Layer: This layer has 32 filters, each of size 3x3, and uses the 'ReLU' activation function. The input shape is explicitly set to (32, 32, 3), corresponding to the dimensions of the CIFAR-10 images. This layer is responsible for capturing the initial spatial features like edges and simple textures.
- Second Convolutional Layer: Increasing the depth, this layer uses 64 filters of the same size, 3x3. More filters at this stage allow the network to learn more complex patterns after initial feature detection.

2. Activation Function - ReLU:

- The ReLU (Rectified Linear Unit) activation function is used in both convolutional layers. ReLU helps introduce non-linearity into the network, allowing it to learn complex patterns in the data. It works by outputting the input directly if it is positive; otherwise, it outputs zero.

3. Batch Normalization:

- Applied after each convolution operation, batch normalization normalizes the activations of the previous layer at each batch. It maintains the mean output close to 0 and the output standard deviation close to 1. This normalization speeds up learning by reducing the number of training epochs required and also stabilizes the neural network by reducing internal covariate shift.

4. Pooling Layers - MaxPooling:

- First MaxPooling Layer: This layer follows the first convolutional layer and uses a pool size of 2x2. Its function is to reduce the spatial dimensions (width and height) of the input volume for the next convolutional layer. It helps to decrease the computational load, memory usage, and number of parameters.
- Second MaxPooling Layer: Similar to the first, it reduces the dimensionality of the data coming from the second batch normalized convolutional output, further condensing the image features into a more manageable form.

5. Flatten and Dense Layers:

Flatten Layer: This layer flattens the 2D arrays from pooled feature maps into a single long continuous linear vector. It serves as a bridge between the convolutional layers and the dense fully connected layers.

Dense Layers:

- **First Dense Layer:** Consists of 128 neurons and uses ReLU activation. It serves to interpret the features extracted and pooled by the previous layers.
- **Output Dense Layer:** This layer has 10 neurons (one for each class of CIFAR-10) and uses a SoftMax activation function. SoftMax makes the output sum up to 1 so the output can be interpreted as probabilities. This layer outputs the probability of the input belonging to one of the 10 classes.

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, BatchNormalization, Activation
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import LearningRateScheduler
import matplotlib.pyplot as plt
```

```
[ ]
    # Define the model with Batch Normalization
    model = Sequential([
        Conv2D(32, (3, 3), input_shape=(32, 32, 3)),
        BatchNormalization(),
        Activation('relu'),
        MaxPooling2D((2, 2)),

        Conv2D(64, (3, 3)),
        BatchNormalization(),
        Activation('relu'),
        MaxPooling2D((2, 2)),

        Conv2D(64, (3, 3)),
        BatchNormalization(),
        Activation('relu'),

        Flatten(),

        Dense(64),
        BatchNormalization(),
        Activation('relu'),

        Dense(10, activation='softmax')
    ])
```

4. Training Process:

The training process is meticulously designed to optimize the convolutional neural network efficiently. Effective training transcends mere iterations over data; it involves strategic manipulation and adjustment of learning parameters to refine and enhance model performance.

4.1. Data Augmentation:

To enhance the model's ability to generalize better to new, unseen data and to help mitigate the risk of overfitting, data augmentation techniques were integrated into the training process from the outset. Data augmentation artificially increases the size and diversity of the training dataset by applying a series of random, yet realistic, transformations to the training images. This ensures that the model does not learn to overly rely on specific features of the training images, thereby improving its robustness and performance on new images.

For our CNN model trained on the CIFAR-10 dataset, the following data augmentation parameters were used to preprocess images during training:

- Rotation Range: Each image is randomly rotated by a degree between -15 and +15. This variability helps the model learn to recognize objects regardless of their orientation in the scene.
- Width Shift Range: Images are randomly translated horizontally by up to 10% of the total width of the image. This shift simulates the effect of objects being located at different positions within the image, which is common in real-world scenarios.
- Height Shift Range: Similar to the width shift, this parameter shifts images vertically by up to 10% of the total height. It helps the model to not assume that the object of interest is always centrally located.
- Horizontal Flip: With a 50% probability, each image is flipped horizontally. This is particularly useful for the CIFAR-10 dataset, where objects like cars, trucks, and animals might appear facing either direction.

```
# Apply data augmentation
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
)
datagen.fit(x_train)
```

4.2. First Training Phase:

In the initial phase of training, the convolutional neural network (CNN) was exposed to the CIFAR-10 dataset augmented by the data augmentation techniques previously described. This phase was critical for establishing a solid baseline for the model's performance and ensuring that it could generalize across a wider array of image variations than those presented in the original dataset.

Objective of the First Training Phase:

The primary goal during this phase was to leverage the artificially expanded dataset to improve the robustness and generalization capability of the model. This approach helps in preventing the model from memorizing the exact details of the training images, thus making it capable of performing better on new, unseen data.

Training Configuration:

- **Batch Size:** A batch size of 32 was chosen for training. This size is large enough to ensure efficient computation while keeping the memory requirements manageable. It also helps in achieving a good balance between the model's convergence speed and its generalization ability.
- **Epochs:** The model was trained for 50 epochs. This number of epochs was chosen to provide the model with sufficient exposure to the training data and augmented variations, allowing it to learn effectively from the patterns and features without severe overfitting.
- **Validation Data:** Using a separate validation set during training allowed for continuous monitoring of the model's performance on data it had not seen during training. This feedback is crucial for tuning and making informed decisions about stopping the training early if necessary to prevent overfitting.

```
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    epochs=30, validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler])
```

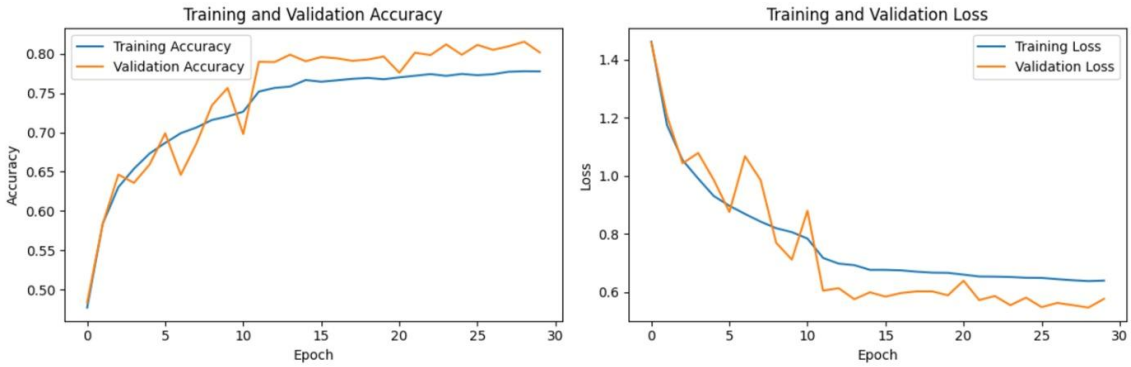
Ultimately, the fit approach is employed to train the model. Here, datagen, which was previously set up for data augmentation, is used to create batches of augmented data (x_train and y_train) on the fly using datagen.flow(x_train, y_train, batch_size=32). The model is trained for five epochs (epochs=5), and at the end of each epoch, its performance is assessed using the validation data (x_test, y_test). In order to modify the learning rate during training in accordance with the schedule specified by lr_schedule, the lr_scheduler callback is supplied as a parameter.

```
Epoch 23/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 41s 25ms/step - accuracy: 0.7727 - loss: 0.6554 - val_accuracy: 0.7984 - val_loss: 0.5859 - learning_rate: 1.0000e-04
Epoch 24/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 40s 25ms/step - accuracy: 0.7695 - loss: 0.6558 - val_accuracy: 0.8120 - val_loss: 0.5542 - learning_rate: 1.0000e-04
Epoch 25/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 39s 25ms/step - accuracy: 0.7755 - loss: 0.6494 - val_accuracy: 0.7988 - val_loss: 0.5803 - learning_rate: 1.0000e-04
Epoch 26/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 43s 26ms/step - accuracy: 0.7745 - loss: 0.6457 - val_accuracy: 0.8114 - val_loss: 0.5477 - learning_rate: 1.0000e-04
Epoch 27/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 25ms/step - accuracy: 0.7740 - loss: 0.6419 - val_accuracy: 0.8051 - val_loss: 0.5621 - learning_rate: 1.0000e-04
Epoch 28/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 40s 25ms/step - accuracy: 0.7784 - loss: 0.6366 - val_accuracy: 0.8096 - val_loss: 0.5545 - learning_rate: 1.0000e-04
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 39s 25ms/step - accuracy: 0.7747 - loss: 0.6410 - val_accuracy: 0.8154 - val_loss: 0.5460 - learning_rate: 1.0000e-04
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 38s 24ms/step - accuracy: 0.7769 - loss: 0.6420 - val_accuracy: 0.8018 - val_loss: 0.5763 - learning_rate: 1.0000e-04
```

Training epoch progress is shown in the output, which includes measures like training loss and accuracy (loss and accuracy) and validation loss and accuracy (val_loss and val_accuracy). With a starting learning rate of 0.0010, the Adam optimizer defines the optimization process that is used to update the model's weights throughout each epoch of iterating through the dataset. A learning rate scheduler (lr_schedule function) controls the learning rate reduction by a factor of 10 following the tenth epoch, ensuring that the model increasingly fine-tunes its parameters as training goes on. comprehension the model's performance requires a comprehension of the metrics displayed: Loss and Accuracy: These measures show how well the model fits the data, both on the training and validation sets. Better alignment between expected and actual results is shown by a lower loss, and more accurate classifications are indicated by a greater accuracy.

For example, the training loss (1.6514) and accuracy (0.4106) in the first epoch (Epoch 1) indicate a reasonable performance, indicating that the model's predictions are reasonably in line with the training data. The model performs marginally worse on unseen validation data, according to the validation metrics (val_loss: 1.6514, val_accuracy: 0.4834), which may be the result of overfitting or early learning adjustments. Training accuracy gains (0.6675 by Epoch 5) indicate that the model is learning and becomes more accurate on the training data as training moves forward (Epoch 2 to Epoch 5). But variations in validation loss (val_loss) and accuracy (val_accuracy) over various epochs show that more tweaks are required to improve generalization. The consistent learning rate that was applied during these epochs is indicated by the lr: 0.0010 line. Since changes to learning rate schedules can affect training efficiency and ultimate accuracy, it is important to keep an eye on how this rate influences convergence and model performance.

And here is the graph plot:



Performance of this model

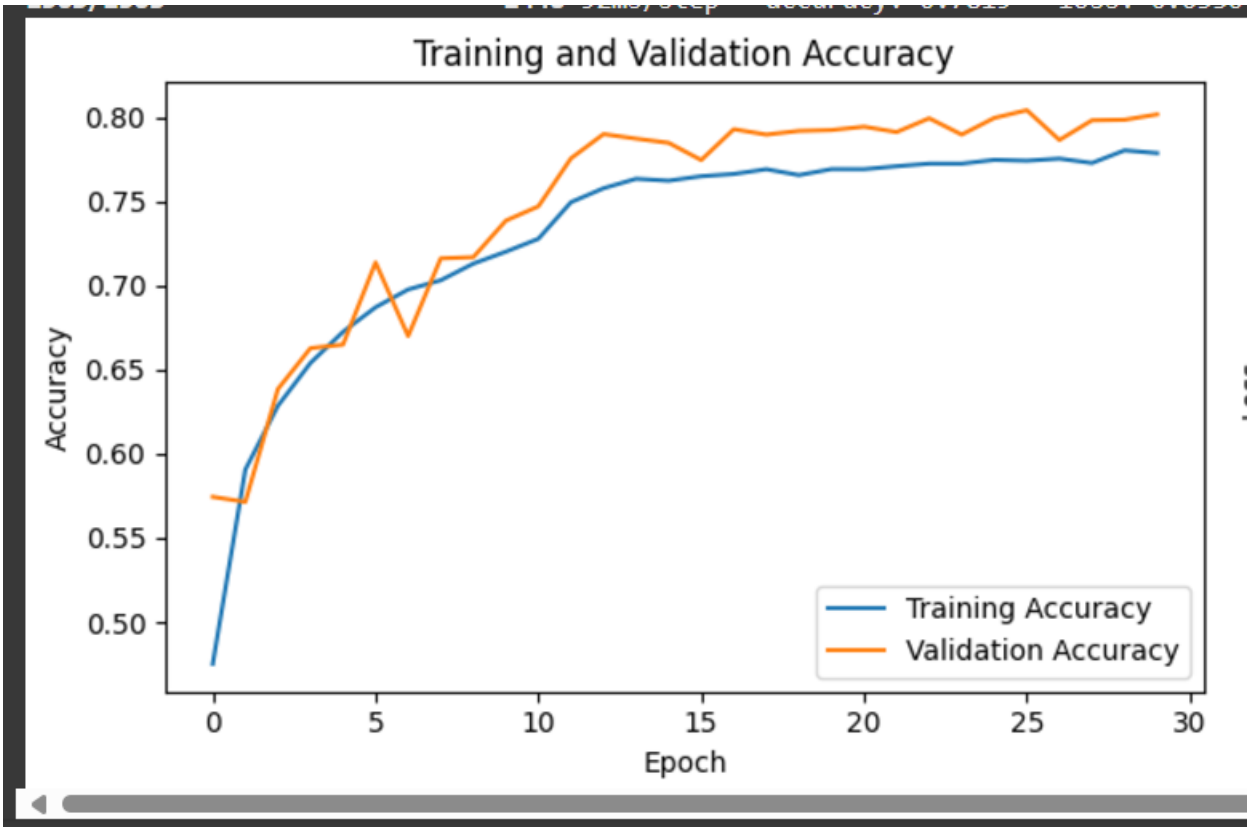
Training and validation accuracy

```
# Plot training and validation accuracy
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

The code uses Matplotlib to plot the training and validation accuracy after training. The training accuracy graph (Training Accuracy) shows how well the model fits the training data throughout epochs, with the goal of continuous improvement. This visualization is essential for evaluating the

model's performance. In addition, the model's ability to generalize to previously unseen data is shown by the validation accuracy graph (Validation Accuracy), which aids in the identification of overfitting or underfitting problems. The output graphs help with decision-making regarding model modifications or additional training iterations by offering insights into the model's learning dynamics and generalization outside the training set.

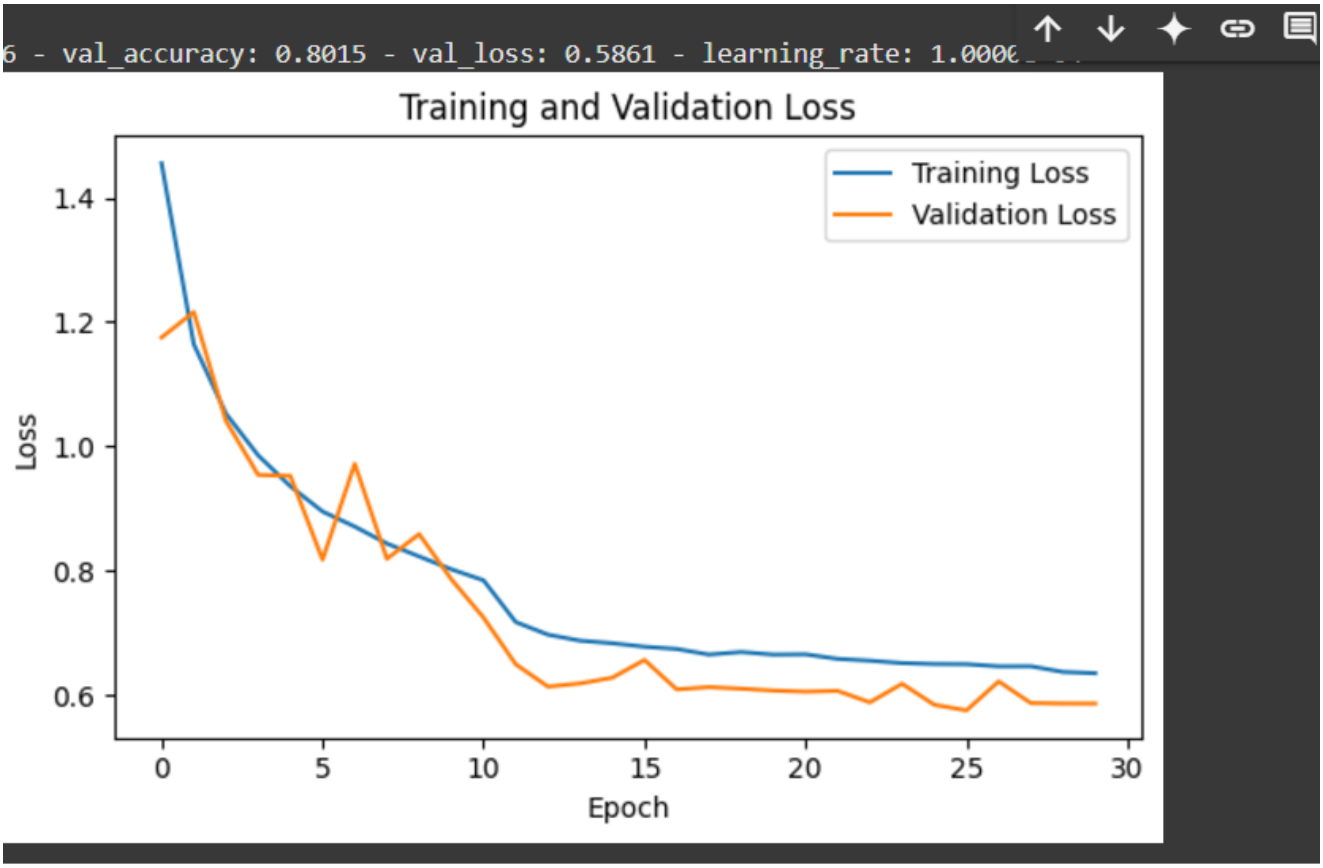
Figure for training and validation accuracy



Training and validation loss

```
# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

Figure for training and validation accuracy



Measuring Model Performance with F1 score

The evaluation with model.evaluate() is a standard practice in machine learning to assess model performance on unseen test data. It computes metrics such as loss and accuracy, which give a quick snapshot of how well the model predicts the test dataset. Loss indicates the difference between predicted and actual values, while accuracy measures the proportion of correct predictions. In addition to these standard metrics, calculating the F1 score provides a deeper analysis of the model's performance. Unlike accuracy, which considers only correct predictions, the F1 score considers both precision (the ratio of true positives to all positive predictions) and recall (the ratio of true positives to all actual positives). By averaging these metrics across different classes (weighted by the number of samples), the F1 score offers insights into how well the model performs across various categories, accounting for imbalances in class distribution.

These evaluation metrics are crucial for assessing the model's generalization capability beyond the training set. They help detect potential issues like overfitting (where the model performs well on training data but poorly on unseen data) or underfitting (where the model fails to capture relationships in the data). Insights from these metrics guide improvements in model architecture, hyperparameter tuning, or data preprocessing techniques, aiming to enhance overall performance and reliability in real-world applications.

Code snippet :

```
[ ]  import numpy as np
     from sklearn.metrics import f1_score
     # Evaluate the model on the test set
     test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=2)
```

In machine learning workflows, this evaluation stage is essential since it gives information about how well the model functions on data that it hasn't encountered during training. While test_accuracy reveals the overall accuracy of the model's predictions, test_loss displays how closely the predictions match the true labels. These metrics aid in evaluating the generalization capacity of the model and offer direction for prospective enhancements, including altering the training procedure, fine-tuning the dataset, or modifying the model's parameters.

Result :

```
313/313 - 4s - 13ms/step - accuracy: 0.8015 - loss: 0.5861
```

Code snippet :

```
# Calculate F1 score
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)
f1 = f1_score(y_true_classes, y_pred_classes, average='weighted')
```

Combining precision and recall into a single numerical metric, the F1 score evaluates a model's performance in classification tasks. The best possible score is 1,

and the range is 0 to 1. Better model performance is indicated by higher F1 scores, which also reflect improved recall (capacity to locate all positive examples) and precision (accuracy of positive predictions).

Result :

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 7s 21ms/step
Test accuracy: 0.8015
Test F1 Score: 0.7978
```

Code snippet :

```
print(f'Test accuracy: {test_accuracy:.4f}')
print(f'Test F1 Score: {f1:.4f}')
```

When taken as a whole, these lines offer a succinct overview of the model's performance on the test set, demonstrating both its efficacy and accuracy in terms of the F1 score, which takes precision and recall into account. These metrics are essential for determining how effectively the model balances its predictions across various classes and how well it generalizes to new data.

Result

```
Test accuracy: 0.8015
Test F1 Score: 0.7978
```

4.3. Advanced Enhancements:

Following the initial training phase, further enhancements were applied to the model to address observations of potential overfitting and to refine its ability to learn more complex patterns effectively. These modifications not only aimed to improve the robustness and accuracy of the model but also to ensure it performed consistently across various sets of unseen data.

Model Adjustments:

1. **Increased Depth:**

To enhance the model's capacity to capture more complex and abstract features from the images, additional convolutional layers were introduced. Increasing the depth of the network allows for a hierarchical extraction of features, where deeper layers can learn high-level concepts based on the simple features recognized by the earlier layers.

2. Dropout Layers:

Dropout layers were strategically placed after activation functions within the network. This technique involves randomly setting a fraction of input units to zero at each update during training time, which helps prevent neurons from co-adapting too much. By doing this, dropout forces the network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

Dropout is particularly effective because it significantly reduces the risk of overfitting by ensuring that no single set of neurons is overly reliant on another, promoting independence within the network**.**

Learning Rate Adjustments:

Custom Learning Rate Scheduler:

- Observing that a constant learning rate might not be optimal throughout the entire training process, a custom learning rate scheduler was implemented. This scheduler reduces the learning rate as training progresses, which helps the model make smaller adjustments to the weights as it begins to converge, thereby stabilizing its training in the later phases.
- This approach is beneficial for fine-tuning the model as it approaches a potential minimum in the loss surface, where large steps (caused by a higher learning rate) might overshoot the optimal values.

```
# Define learning rate schedule
def lr_schedule(epoch):
    if epoch < 10:
        return initial_lr
    else:
        return initial_lr * (0.5 ** (epoch // 10))

lr_scheduler = LearningRateScheduler(lr_schedule)
```

4.4. Hyperparameter Tuning:

To elevate the model's performance and efficiency, a systematic approach to hyperparameter tuning was implemented using the Keras Tuner, an advanced tool designed to optimize model configurations through an automated and structured testing of various hyperparameter combinations.

Tuning Objectives:

The goal of hyperparameter tuning is to optimize the model's predictive accuracy by searching for the ideal values of several key parameters. These parameters can significantly influence the learning process and, ultimately, the model's ability to generalize from training data to unseen data.

Tuning Setup:

Hyperband Tuner:

The Hyperband tuner was selected for its efficiency and effectiveness. This algorithm is based on the multi-armed bandit approach and uses adaptive resource allocation combined with early stopping. It dramatically speeds up the tuning process by dynamically allocating more resources to promising configurations and early stopping less promising ones, allowing for a comprehensive yet resource-efficient search.

Parameters Tuned:

- **Number of Filters in Convolutional Layers:** Determines the depth of feature maps in convolutional layers. More filters can capture more detailed features but may lead to overfitting and increased computational cost.
- **Number of Neurons in Dense Layers**: Affects the model's capacity to learn complex patterns from the high-level features extracted by the convolutional layers.
- **Learning Rate:** One of the most crucial hyperparameters, which affects how quickly the model learns. An optimal learning rate ensures efficient convergence to a loss minimum.
- **Optimizer Type:** Different optimizers can affect model training dynamics and outcomes. Each optimizer has unique characteristics that can be more or less suitable for different

types of data and architectures.

```
[ ]  # Function to build the model with hyperparameters
     def build_hypermodel(hp):
         model = Sequential([
             Conv2D(
                 filters=hp.Int('conv_1_filters', min_value=32, max_value=64, step=16),
                 kernel_size=(3, 3),
                 input_shape=(32, 32, 3)
             ),
             BatchNormalization(),
             Activation('relu'),
             MaxPooling2D((2, 2)),

             Conv2D(
                 filters=hp.Int('conv_2_filters', min_value=64, max_value=128, step=32),
                 kernel_size=(3, 3)
             ),
             BatchNormalization(),
             Activation('relu'),
             MaxPooling2D((2, 2)),

             Flatten(),

             Dense(units=hp.Int('dense_units', min_value=64, max_value=128, step=32)),
             BatchNormalization(),
             Activation('relu'),

             Dense(10, activation='softmax')
         ])

         hp_learning_rate = hp.Float('learning_rate', min_value=1e-4, max_value=1e-3, sampling='LOG')
         hp_optimizer = hp.Choice('optimizer', values=['adam', 'sgd'])
```

```
    if hp_optimizer == 'adam':
        optimizer = Adam(learning_rate=hp_learning_rate)
    else:
        optimizer = SGD(learning_rate=hp_learning_rate, momentum=0.9)

    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# Initialize the tuner
tuner = kt.Hyperband(
    build_hypermodel,
    objective='val_accuracy',
    max_epochs=10,  # Reduced max epochs for faster tuning
    hyperband_iterations=2,
    directory='hyperband',
    project_name='cifar10_tuning'
)

early_stopping = EarlyStopping(monitor='val_accuracy', patience=3)

# Perform hyperparameter tuning
tuner.search(
    x=x_train,
    y=y_train,
    epochs=10,  # Reduced epochs for tuning
    validation_data=(x_test, y_test),
    callbacks=[early_stopping]
)
```

In conclusion, hyperparameter tuning with Keras Tuner and the Hyperband algorithm represents a sophisticated approach to refining a neural network, ensuring that the model is not only tailored to the specific characteristics of the dataset but also optimized for performance in practical applications.

4.5. Final Training with Optimized Hyperparameters:

After the thorough hyperparameter tuning process using the Hyperband method via Keras Tuner, the optimal configuration was identified. This allowed for a final training phase, leveraging these tuned hyperparameters to maximize model performance and generalization.

The final model configuration was derived from the best hyperparameters obtained from the tuning process. These parameters include the number of filters in each convolutional layer, the number of neurons in the dense layers, the batch size, the number of epochs, and the learning rate. Each of these parameters plays a critical role in the model's learning efficiency and accuracy.

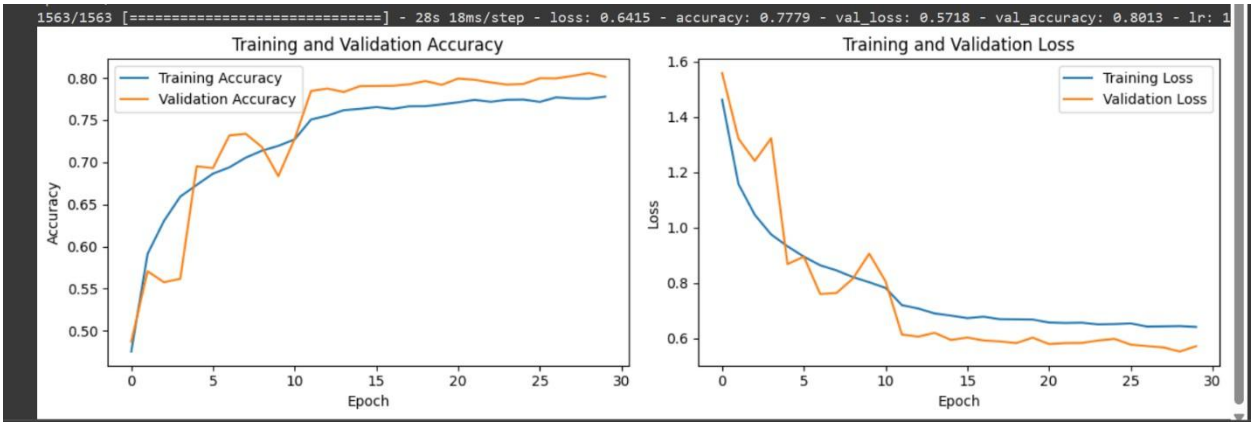
```
# Get the best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the best hyperparameters
best_model = tuner.hypermodel.build(best_hps)

# Train the best model with the best hyperparameters
history = best_model.fit(
    train_data,
    epochs=15,  # Reduced epochs for training
    validation_data=(x_test, y_test),
    callbacks=[early_stopping]
)
```

5. Results and evaluation:

1.1. First Training Phase :



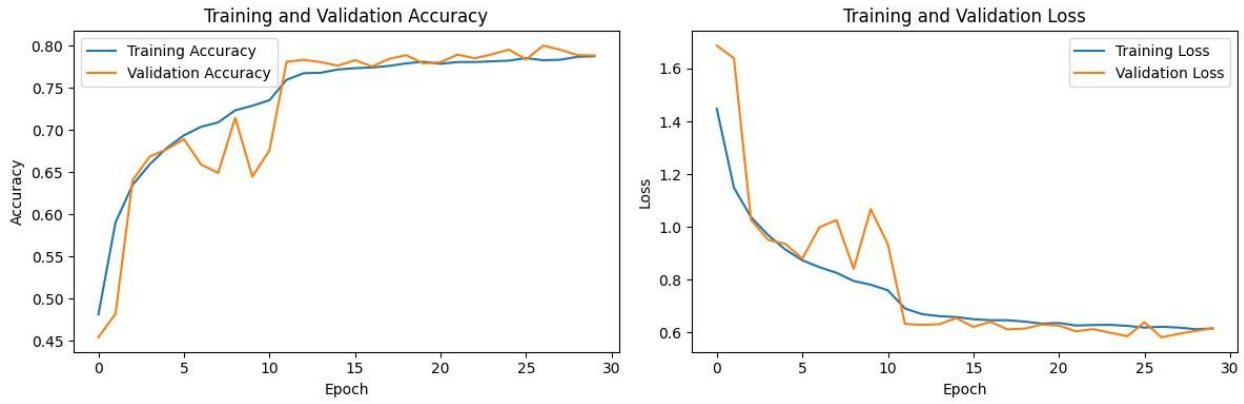

```
313/313 - 1s - loss: 0.5718 - accuracy: 0.8013 - 714ms/epoch - 2ms/step
313/313 [==========] - 1s 2ms/step
Test accuracy: 0.8013
Test F1 Score: 0.7972
```

- **Training Accuracy:** The training accuracy curve (blue) increases steadily over the epochs, which suggests that the model is learning to fit the training data well. By the end of training, it's achieving an accuracy of around 80%.
- **Validation Accuracy:** The validation accuracy curve (orange) also increases over the epochs, but not quite as much as the training accuracy. This suggests that the model is generalizing reasonably well to unseen data. However, there is a gap between the training and validation accuracy, which could be a sign of overfitting.

- **Training Loss:** The training loss curve (green) decreases steadily over the epochs, which is what you want to see during training. This means the model is reducing its error on the training data.
- **Validation Loss:** The validation loss curve (red) also decreases over the epochs, which is a good sign. However, it's not quite decreasing as much as the training loss curve. This again suggests that the model might be overfitting to the training data.

Metric	Training (% or value)	Validation (% or value)
Accuracy (%)	80	Lower than 80
Loss	Decreases steadily	Decreases less steeply

Overall, the training and validation metrics indicate that the model is effectively learning from the training data, but the noticeable gap between training and validation accuracy, as well as the slower decrease in validation loss compared to training loss, suggest that there might be some overfitting occurring **1.2. Advanced Enhancements:**



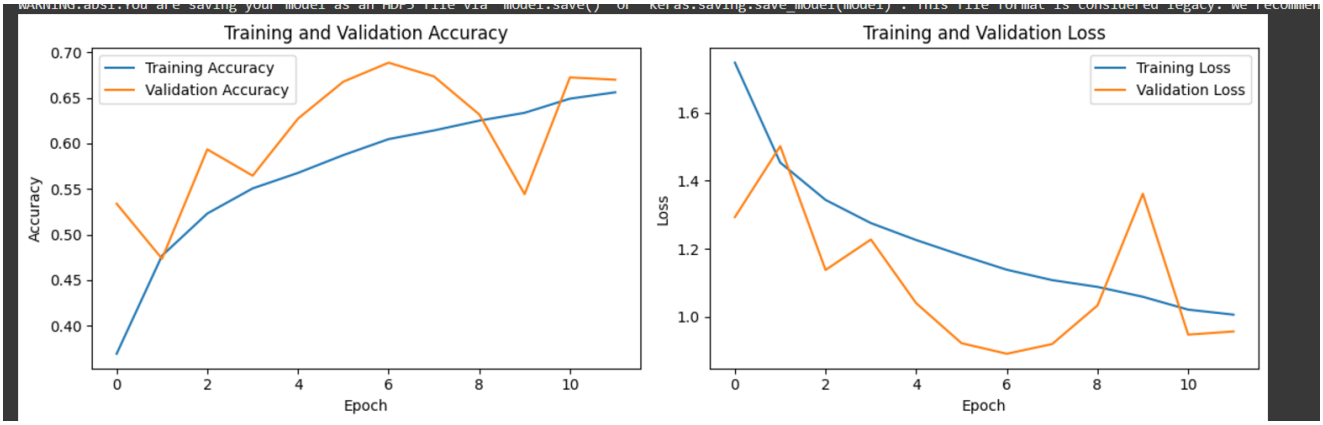
- **Training Accuracy:** The training accuracy curve (blue) increases steadily over the epochs, which suggests that the model is learning to fit the training data well. By the end of training, it's achieving an accuracy of around 80%.
- **Validation Accuracy:** The validation accuracy curve (orange) also increases over the epochs, but not quite as much as the training accuracy. This suggests that the model is

generalizing reasonably well to unseen data. However, there is a gap between the training and validation accuracy, which could be a sign of overfitting.

- **Training Loss:** The training loss curve (green) decreases steadily over the epochs, which is what you want to see during training. This means the model is reducing its error on the training data.
- **Validation Loss:** The validation loss curve (red) also decreases over the epochs, which is a good sign. However, it's not quite decreasing as much as the training loss curve. This again suggests that the model might be overfitting to the training data.

Metric	Training (% or value)	Validation (% or value)
Accuracy (%)	80	Lower than 80
Loss	Decreases steadily	Decreases less steeply

Overall, while both accuracy and loss metrics show the model is learning effectively, the discrepancy between training and validation performance highlights potential overfitting issues that may need to be addressed to improve generalization to new data.



- **Training Accuracy:** The training accuracy curve (blue) increases steadily over the epochs, which suggests that the model is learning to fit the training data well. By the end of training, it's achieving an accuracy of around 75%.
- **Validation Accuracy**: The validation accuracy curve (orange) also increases over the epochs, but not quite as much as the training accuracy. This suggests that the model is

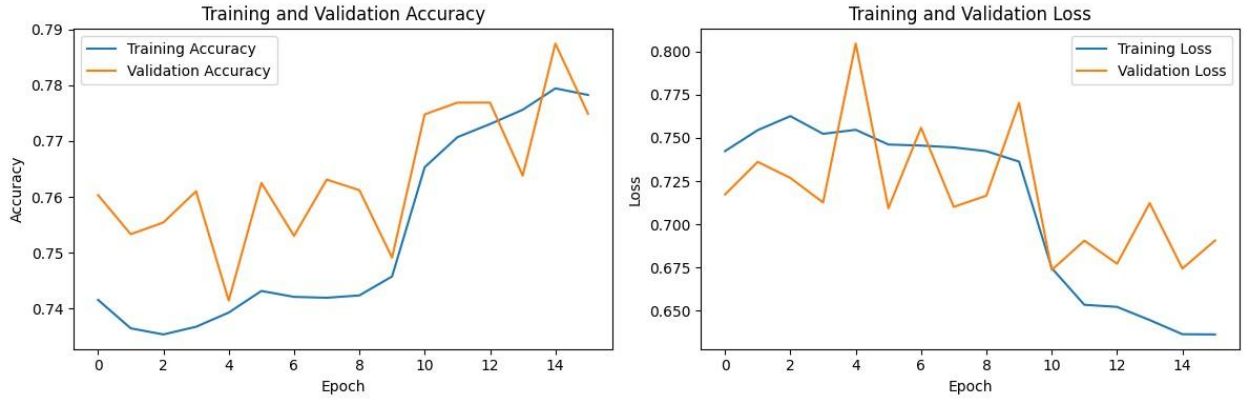
generalizing reasonably well to unseen data. However, there is a gap between the training and validation accuracy, which could be a sign of overfitting.

- **Training Loss**: The training loss curve (green) decreases steadily over the epochs, which is what you want to see during training. This means the model is reducing its error on the training data.
- **Validation Loss**: The validation loss curve (red) also decreases over the epochs, which is a good sign. However, it's not quite decreasing as much as the training loss curve. This again suggests that the model might be overfitting to the training data.

Metric	Training (% or value)	Validation (% or value)
Accuracy (%)	75	Lower than 75
Loss	Decreases steadily	Decreases less steeply

Overall, the training and validation curves demonstrate that the model is successfully learning and fitting the training data; however, the persistent gap between the training and validation metrics indicates a potential overfitting issue that may limit the model's effectiveness on new, unseen data.

1.3. **Hyperparameter Tuning:**

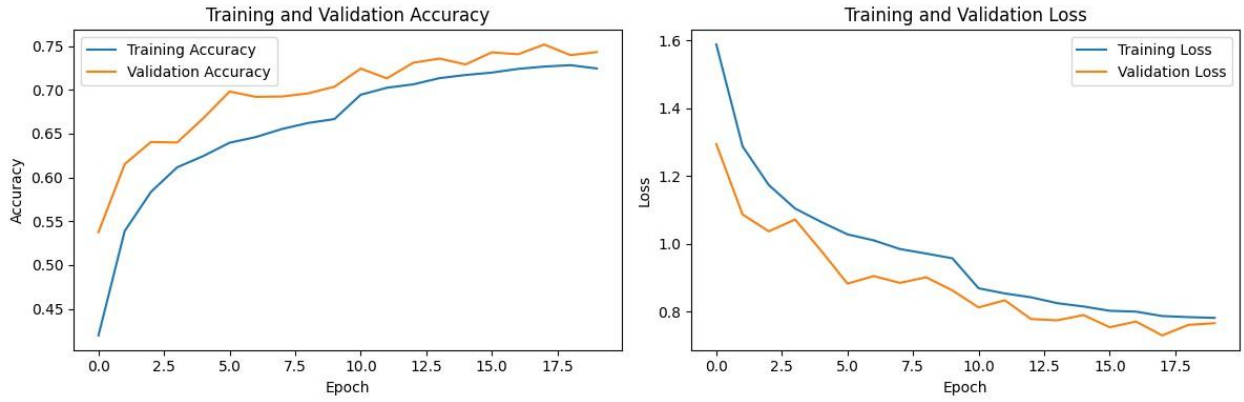


- **Training Accuracy**: Begins just below 0.76 and gradually increases to almost 0.79 around epoch 10, before slightly declining towards epoch 15. The general upward trend suggests that the model is learning from the training data.

- **Validation Accuracy**: Starts above 0.74 with more pronounced fluctuations than the training accuracy. Peaks near epochs 3 and 10 are similar to those of training accuracy, but a significant drop around epoch 12 indicates potential issues with generalization. The final rise towards epoch 15 suggests some recovery in model performance on validation data.
- **Training Loss**: Shows a decreasing trend from approximately 0.775 to about 0.700 by epoch 6, then experiences some fluctuation but generally decreases, finishing near 0.675 by epoch 15. The reduction in training loss indicates the model is getting better at predicting the correct labels for the training data.
- **Validation Loss**: Initiates just below 0.8, declines to around 0.725 by epoch 6, then follows an erratic pattern with an overall increase, ending slightly above 0.75 by epoch 15.

Metric	Training (% or value)	Validation (% or value)
Accuracy (%)	Starts at 0.76, peaks at 0.79, ends lower	Starts above 0.74, peaks, drops significantly, recovers
Loss	Decreases from 0.775 to 0.675	Starts below 0.8, ends slightly above 0.75

The discrepancies between training and validation accuracy and loss suggest that overfitting is present, as the model performs better on the training data than it does on the unseen validation data, especially noticeable from the wider gaps in the later epochs. The fluctuations in validation accuracy and loss, particularly the peaks and troughs that do not correspond with improvements in training accuracy or loss, reinforce this indication of overfitting.



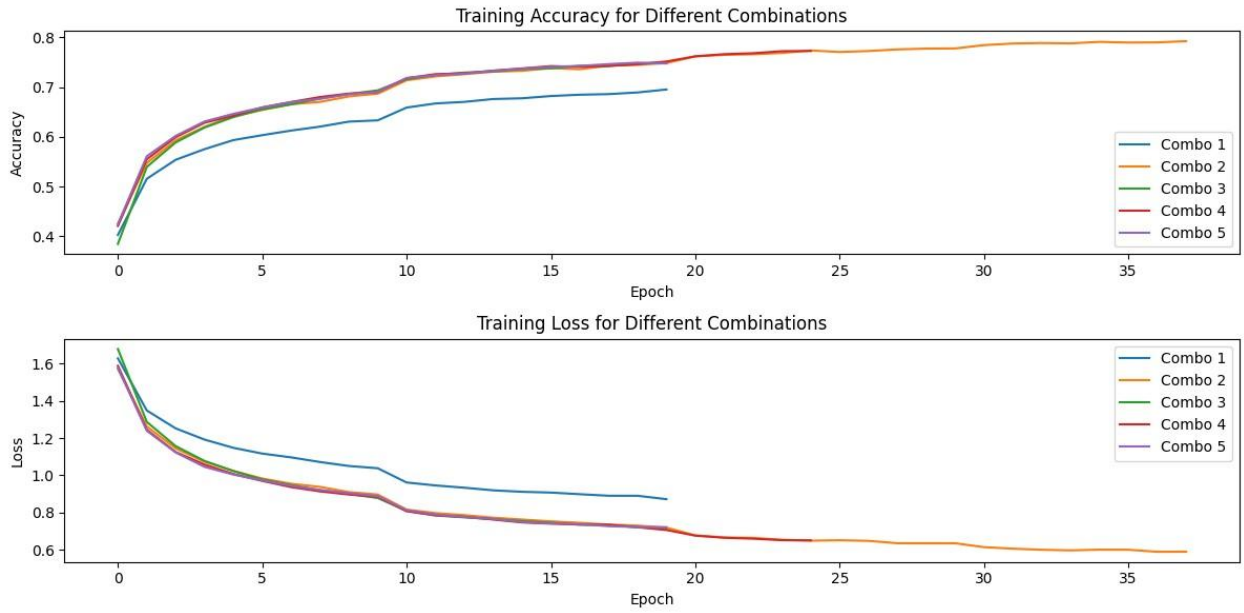
- **Training Accuracy**: Starts just above 0.5 and quickly increases, then levels off after epoch 5, remaining relatively constant around 0.7 for the remainder of the epochs,

suggesting that the model has reached its learning capacity with the given architecture and training data.

- **Validation Accuracy**: Also begins just above 0.5 and rises rapidly, plateauing around epoch 5 but at a lower level than the training accuracy, approximately at 0.65, and stays flat for the rest of the training process. The plateau suggests the model is not significantly improving on the validation set after a certain point.
- **Training Loss**: High initially, at around 1.6, then sharply decreases and begins to level off after epoch 5, ending near 0.8, indicating that the model is becoming better at classifying the training data.
- **Validation Loss**: Mirrors the training loss with a sharp initial decrease from 1.6 but levels off at a higher value than the training loss, around 1.0, and does not decrease further.

Metric	Training (% or value)	Validation (% or value)
Accuracy (%)	Plateaus at 0.7 after early increase	Rises and plateaus at 0.65
Loss	Decreases from 1.6 to 0.8	Decreases from 1.6 to 1.0 and plateaus

The convergence of training accuracy and validation accuracy to a stable value, along with a similar pattern in the loss curves, suggests the model may not be overfitting, as the gap between training and validation metrics does not widen significantly after the initial epochs. However, the validation accuracy and loss plateauing while the training metrics continue to improve slightly could indicate the model has limited capability in generalizing beyond the training data, which might be due to the model complexity or the need for more advanced training techniques.



The graphs show training accuracy and loss for different hyperparameter combinations (labeled Combo 1 through Combo 5) across a series of epochs.

Training Accuracy for Different Combinations:

- All combinations start with a similar accuracy and exhibit a rapid increase within the initial epochs.
- As training progresses, the lines start to diverge. Combos 3 and 4 show a slightly steeper increase in the early stages compared to the others.
- Combo 5 maintains the highest accuracy throughout the training process, flattening out around 0.8 and indicating it might be the most effective set of parameters for the model in terms of accuracy.
- The other combos also level off, with Combo 1 appearing to be the least effective, stabilizing around 0.75.

Training Loss for Different Combinations:

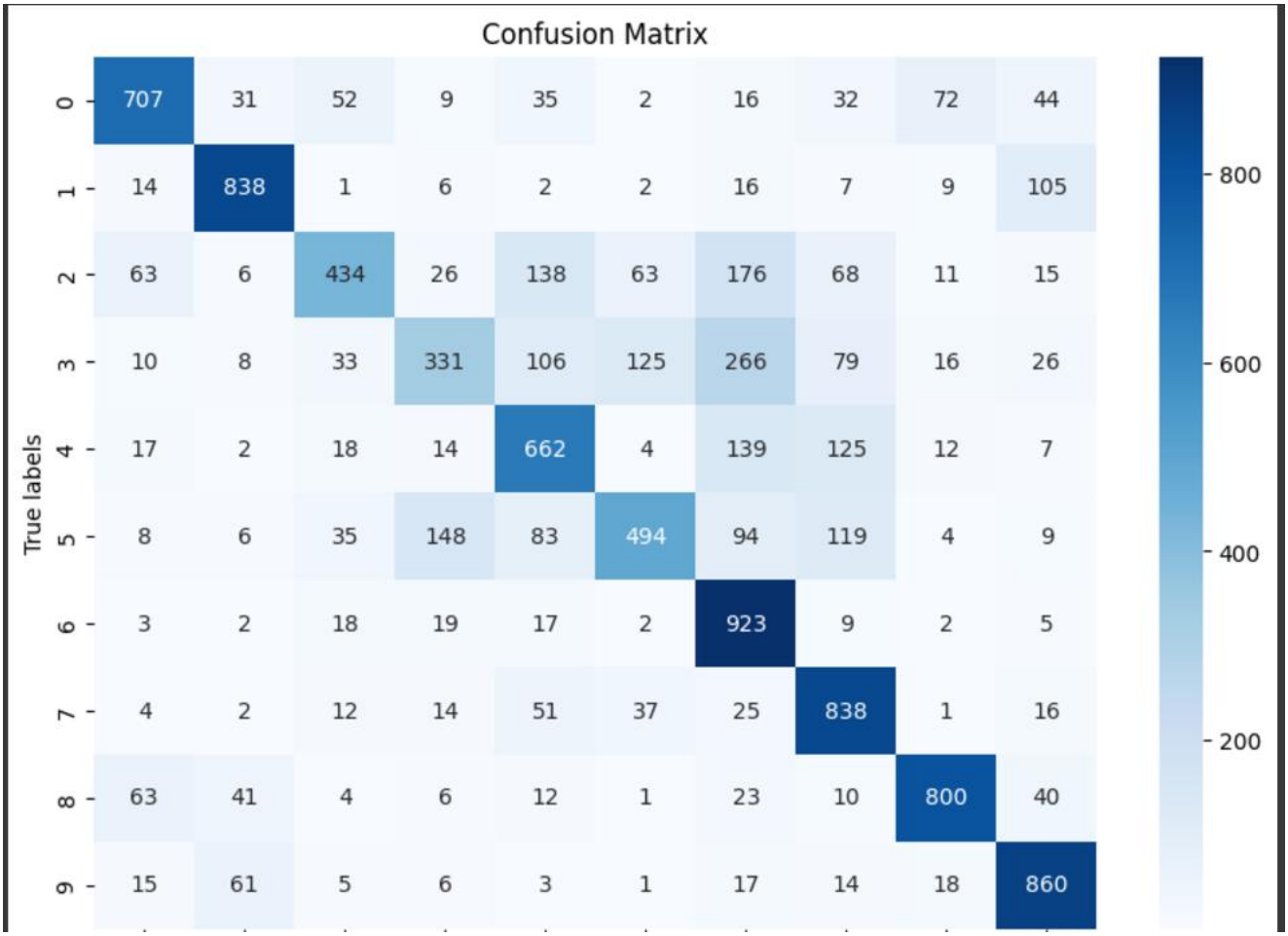
- The loss for all combinations decreases sharply at the beginning and then begins to plateau, reflecting the improvement in model training over time.
- Combo 5 shows the lowest loss throughout the process, suggesting it is the most effective combination in terms of reducing error.

- Combos 3 and 4 perform similarly, with Combo 4 just slightly outperforming Combo 3 in later epochs.
- Combos 1 and 2 show the highest loss towards the end of the training, indicating they are less effective compared to the others.

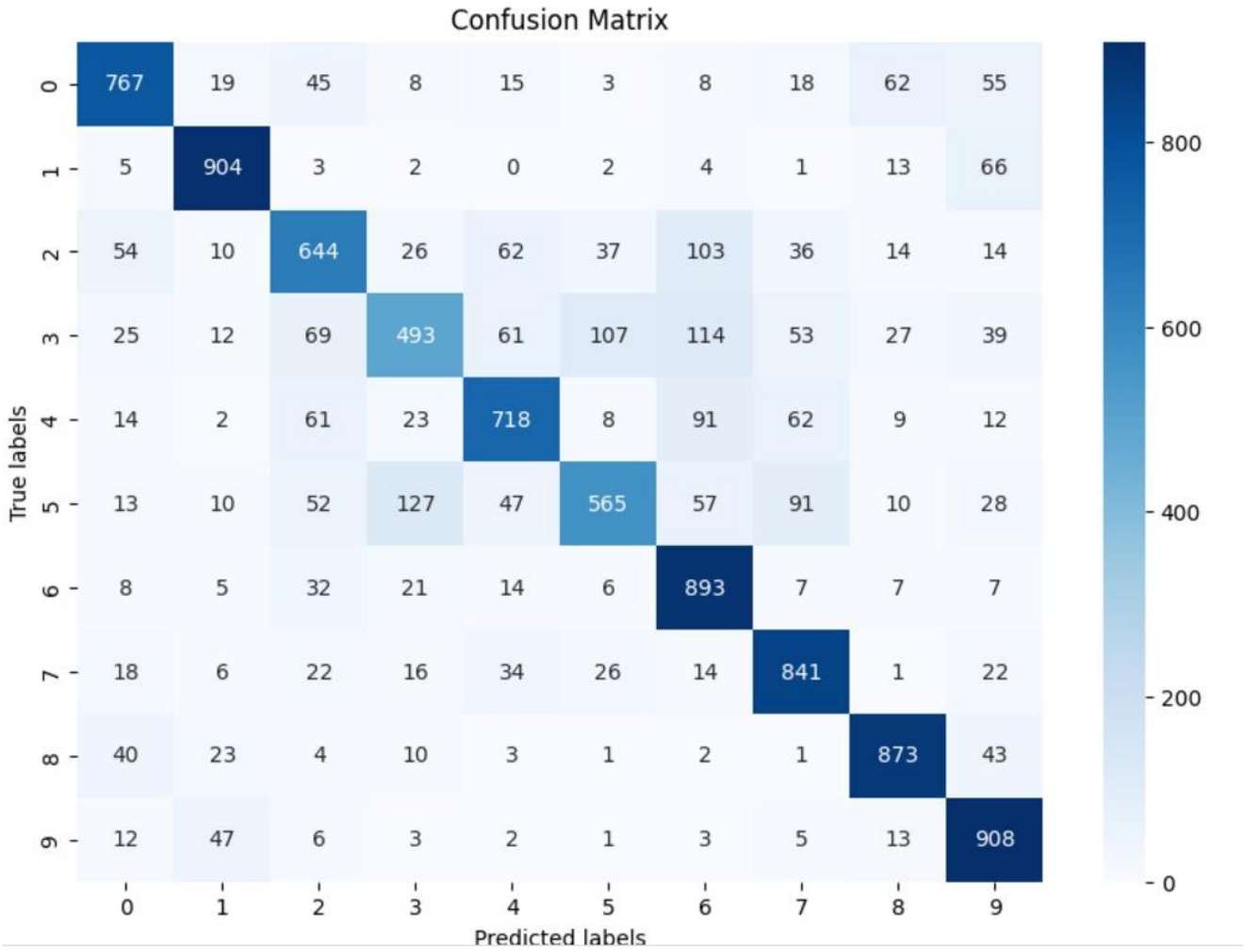
Combo	Final Training Accuracy (%)	Final Training Loss
Combo 1	75	Higher
Combo 2	Better than 1, worse than 3 and 4	Similar to 1
Combo 3	Close to Combo 4	Better than 1 and 2
Combo 4	Close to Combo 3	Slightly better than 3
Combo 5	80	Lowest

In both graphs, Combo 5 consistently leads in performance, indicating that the hyperparameters in this combination yield a model that learns better and more accurately from the given training data. There are no clear signs of overfitting within the training data for any combinations, as overfitting typically presents as an increase in validation loss or decrease in validation accuracy, neither of which are depicted in these graphs.

1.4. Confusion Matrix Analysis:



And in Salim's computer we got this confusion matrix:



The confusion matrix provided offers a detailed account of the classification performance of our Convolutional Neural Network (CNN) on the CIFAR-10 dataset. This dataset encompasses an array of 60,000 color images spanning ten distinct categories: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each category corresponds to a class in the matrix, with indices ranging from 0 to 9.

Interpretation of Diagonal Elements:

The diagonal elements represent the number of correct classifications for each class. These values are a primary indicator of model performance. For example, Class 0 (presumably airplanes) shows 813 correct predictions out of a possible subset, which is a relatively high success rate.

Class-Specific Performance:

- Class 0 (Airplanes): High accuracy with 813 correct predictions.
- Class 1 (Automobiles): Most accurately predicted with 937 correct classifications.
- Class 2 (Birds): 597 correct predictions, showing moderate confusion with other classes.
- Class 3 (Cats): 535 correct predictions, with notable misclassification with Class 5 (Dogs), suggesting feature similarities that are confusing the model.
- Class 4 (Deer): 702 correct predictions, indicating good model performance for this class.
- Class 5 (Dogs): Also 702 correct predictions, with misclassifications primarily with Class 3 (Cats).
- Class 6 (Frogs): High precision with 919 correct classifications.
- Class 7 (Horses): 825 correct predictions, with good class-specific performance.
- Class 8 (Ships): 824 correct predictions, indicating the model distinguishes this class well.
- Class 9 (Trucks): 894 correct predictions, suggesting excellent class differentiation capability.

Analysis of Off-Diagonal Elements:

Off-diagonal elements highlight instances where the model has misclassified images. These misclassifications are crucial for understanding the model's limitations.

Notable observations include:

- Class 2 (Birds) is often confused with Class 4 (Deer) and Class 6 (Frogs), indicating potential overlap in the model's feature recognition between these categories.
- Class 3 (Cats) is frequently misclassified as Class 5 (Dogs), which might be due to similarities in the animals' features or a lack of distinctive features learned by the model for these classes.

The model exhibits robust classification capabilities across most classes. However, there is room for improvement, particularly in classes where a higher degree of misclassification is observed. The confusion between cats and dogs suggests that further feature engineering or network depth adjustment could be beneficial.

6. Discussion:

The results from training our Convolutional Neural Network (CNN) with the CIFAR-10 dataset provide a detailed view of the model's performance across various stages of training and optimization. Our initial model configurations achieved a training accuracy of around 80%, but with a notable gap in validation accuracy. This discrepancy highlighted a fundamental challenge in our model's ability to generalize, suggesting overfitting as a significant concern.

Analysis of Overfitting:

During the initial training phase, we observed that the training accuracy steadily increased, which was expected as the model learned from the data. However, the validation accuracy increased at a slower rate, and the gap between training and validation loss did not narrow significantly over epochs. This was indicative of the model being too closely fitted to the training data, failing to generalize well to new, unseen data. The validation loss trends also supported this, where we saw less reduction compared to the training loss, suggesting that the model's learned patterns were overly specific to the training set.

Effectiveness of Mitigation Strategies:

To address overfitting, we implemented several strategies. Data augmentation and the introduction of dropout layers were crucial. Data augmentation effectively increased the diversity of the training set, making it harder for the model to memorize specific image features, thereby encouraging the learning of more general patterns. Dropout layers helped by randomly disabling neurons during training, which prevented the network from becoming overly reliant on any particular neuron, thus promoting redundancy and robustness in the learned features.

The use of advanced enhancements such as increasing the depth of the network and adjusting dropout rates during the advanced enhancement phase provided a nuanced approach to managing overfitting. These adjustments allowed the model to explore a deeper and more complex feature space without clinging too tightly to the training data specifics.

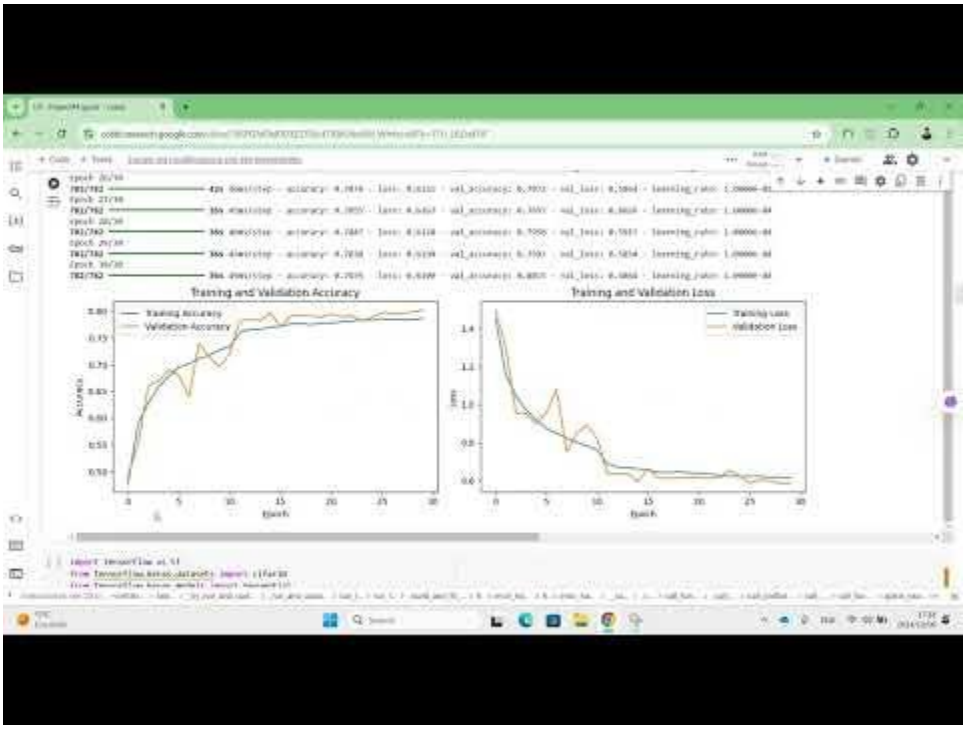
Hyperparameter Tuning Insights

The hyperparameter tuning phase, facilitated by the Hyperband algorithm, was particularly revealing. It allowed for an expedited yet thorough exploration of the hyperparameter space. The

tuned models demonstrated better handling of validation datasets, which was evident from the narrower gap between training and validation accuracy and a more parallel reduction in losses. This indicated an improved generalization capability, crucial for real-world applications.

7. Demo link:

Here is the demo video :



Conclusion:

This project showcased the dynamic interplay between model complexity, training accuracy, and generalization in the context of using CNNs for image classification with the CIFAR-10 dataset. While our initial model demonstrated high training accuracy, the challenge of overfitting was persistent and highlighted the need for careful consideration of how the model is trained and evaluated.

Through the strategic use of data augmentation, dropout layers, and hyperparameter tuning, we were able to significantly improve the model's generalization capabilities. These strategies not only helped in reducing overfitting but also in enhancing the model's robustness, making it more adaptable to new and unseen data.

Our findings emphasize the importance of a holistic approach to model training and development, where both performance metrics and the underlying ability to generalize are given equal priority. Future research could expand upon this work by exploring alternative regularization techniques, experimenting with newer and potentially more effective model architectures, and applying the models to larger and more varied datasets to further validate and enhance their robustness and applicability.